Problem Solving - IV - Hands On

1. Non-Repeating Character

Problem statement :

Given a string S consisting of lowercase Latin Letters. Return the first non-repeating character in S. If there is no non-repeating character, return '$'.

Example 1:

```
Input:
S = hello
Output: h
```

> Explanation: In the given string, the first character which is non-repeating is h, as it appears first and there is no other 'h' in the string.

Example 2:

```
Input:
S = zxvczbtxyzvy
Output: c
```

> Explanation: In the given string, 'c' is the character which is nonrepeating.

Constraints:

```
1 <= N <= 10^5
```

Analysis:

- We have given a string S consisting of lowercase Latin Letters.
- We have to find the first non-repeating character in S.
- If there is no non-repeating character, return '$'.
- We can solve this problem by using the brute force approach.
- We will iterate over the string and check the frequency of each character.
- If the frequency of any character is 1, then we will return that character.
- If we don't find any non-repeating character, then we will return '$'.
- The time complexity of this approach is O(N^2).

Code:

```
package com.hands_on;

import java.util.Scanner;

public class NonRepeatingCharacter {
    public static char nonRepeatingCharacter(String str) {
        for(int i = 0; i<str.length(); i++){
            int cnt = 0;
            for(int j = i+1; j<str.length();j++){
                if(str.charAt(i)==str.charAt(j)) {
                    cnt++;
                }
            }
            if (cnt <= 0){
                return str.charAt(i);
            }
        }
        return '$';
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter any string: ");
        String str = sc.nextLine();
        System.out.println(
            "The first non-repeating charater is "
            + nonRepeatingCharacter(str)
        );
    }
}
```

Output:

```
Enter any string: hello
The first non-repeating charater is h
```

---

2. Maximum Occurring Character

Problem statement:

Given a string str of lowercase alphabets. The task is to find the maximum occurring character in the string str. If more than one character occurs the maximum number of time then print the lexicographically smaller character.

Example 1:

```
Input:
str = testsample
Output: e
```

> Explanation: e is the character which is having the highest frequency.

Example 2:

```
Input:
str = output
Output: t
```

> Explanation: t and u are the characters with the same frequency, but t is lexicographically smaller.

Constraints:

```
1 ≤ |s| ≤ 100
```

Analysis:

- We have given a string str of lowercase alphabets.
- We have to find the maximum occurring character in the string str.
- If more than one character occurs the maximum number of time then print the lexicographically smaller character.
- We can solve this problem by using the brute force approach.
- We will iterate over the string and check the frequency of each character.
- We will maintain the maximum frequency and the lexicographically smaller character.

- If we find any character with a higher frequency, then we will update the maximum frequency and the lexicographically smaller character.
- The time complexity of this approach is O(N).

Code:

```java
package com.hands_on;

import java.util.Scanner;

public class MaximumOccurringCharacter {
    public static char maximumOccurringCharacter(String str) {
        int[] freq = new int[26];
        for (int i = 0; i < str.length(); i++) {
            freq[str.charAt(i) - 'a']++;
        }
        int max = 0;
        char res = 'a';
        for (int i = 0; i < 26; i++) {
            if (freq[i] > max) {
                max = freq[i];
                res = (char) (i + 'a');
            }
        }
        return res;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter any string: ");
        String str = sc.nextLine();
        System.out.println(
            "The maximum occurring character is "
            + maximumOccurringCharacter(str)
        );
    }
}
```

Output:

```
Enter any string: testsample
The maximum occurring character is e
```

---

3. Find First Palindromic String in the Array

Problem statement :

Given an array of strings words, return the first palindromic string in the array. If there is no such string, return an empty string "". A string is palindromic if it reads the same forward and backward.
Example 1:

```
Input: words = ["abc","car","ada","racecar","cool"]
Output: "ada"
```

> Explanation: The first string that is palindromic is "ada". Note that "racecar" is also palindromic, but it is not the first.

Example 2:

```
Input: words = ["notapalindrome","racecar"]
Output: "racecar"
```

> Explanation: The first and only string that is palindromic is "racecar".

Constraints:

```
1 <= words.length <= 100
```

Analysis:

- We have given an array of strings words.
- We have to return the first palindromic string in the array.
- If there is no such string, return an empty string "".
- We can solve this problem by using the brute force approach.
- We will iterate over the array and check whether the string is palindromic or not.
- If we find any palindromic string, then we will return that string.
- If we don't find any palindromic string, then we will return an empty string.
- The time complexity of this approach is O(N*M), where N is the number of strings in the array and M is the length of the string.

Code:

```java
package com.hands_on;

import java.util.Scanner;
```

```java
public class FindFirstPalindromicStringInTheArray {
    public static String findFirstPalindromicStringInTheArray(String[] str) {
        for(String s: str){
            if(checkPalindrome(s)){
                return s;
            }
        }
        return "None of the String is Palindrome";
    }

    public static boolean checkPalindrome(String word){
        StringBuilder rev = new StringBuilder();
        for (int i = word.length() - 1; i >= 0; i--) {
            rev.append(word.charAt(i));
        }
        return word.contentEquals(rev);
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter the size: ");
        int size = sc.nextInt();
        System.out.println("Enter the elements: ");
        String[] str = new String[size];
        for(int i=0;i<size;i++){
            str[i] = sc.next();
        }
        System.out.println("The first palindrome in the array is `"+ findFirstPalindromicStr
    }
}
```

Output

```
Enter the size: 5
Enter the elements:
abc
abz
ada
madam
cool
The first palindrome in the array is `ada`
```

---

4. Check If Two String Arrays are Equivalent

Problem statement :

Given two string arrays word1 and word2, return true if the two arrays represent the same string, and false otherwise. A string is represented by an array if the array elements concatenated in order forms the string.
Example 1:

```
Input: word1 = ["ab", "c"], word2 = ["a", "bc"]
Output: true
```

> Explanation:
>
> word1 represents string "ab" + "c" -> "abc"
>
> word2 represents string "a" + "bc" -> "abc"

> The strings are the same, so return true.

Example 2:

```
Input: word1 = ["a", "cb"], word2 = ["ab", "c"]
Output: false
```

Constraints:

```
1 <= word1.length, word2.length <= 10^3
1 <= word1[i].length, word2[i].length <= 10^3
1 <= sum(word1[i].length), sum(word2[i].length) <= 10^3
word1[i] and word2[i] consist of lowercase letters
```

Analysis:

- We have given two string arrays word1 and word2.
- We have to return true if the two arrays represent the same string, and false otherwise.
- We can solve this problem by using the brute force approach.
- We will concatenate all the strings in the array and compare the resultant strings.
- If both the strings are the same, then we will return true.
- If the strings are different, then we will return false.
- The time complexity of this approach is O(N*M), where N is the number of strings in the array and M is the length of the string.

Code:

```java
package com.hands_on;

import java.util.Scanner;

public class CheckIfTwoStringArraysAreEquivalent {
    public static boolean checkIfTwoStringArraysAreEquivalent(String[] str1, String[] str2)
        StringBuilder a = new StringBuilder();
        StringBuilder b = new StringBuilder();
        for(String s: str1){
            a.append(s);
        }
        for(String s: str2){
            b.append(s);
        }
        return a.toString().contentEquals(b);
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the size: ");
        int size = sc.nextInt();
        System.out.println("Enter the elements of array-1: ");
        String[] array1 = new String[size];
        for(int i=0;i<size; i++){
            array1[i] = sc.next();
        }
        System.out.println("Enter the elements of array-2: ");
        String[] array2 = new String[size];
        for(int i=0;i<size; i++){
            array2[i] = sc.next();
        }
        System.out.println(checkIfTwoStringArraysAreEquivalent(array1,array2));
    }
}
```

Output

```
Enter the size: 3
Enter the elements of array-1:
ab
c
d
Enter the elements of array-2:
a
bc
d
true
```

---

5. Make Two strings to Anagram

Problem statement:

Given two strings S1 and S2 in lowercase, the task is to make them anagram. The only allowed operation is to remove a character from any string. Find the minimum number of characters to be deleted to make both the strings anagram. Two strings are called anagrams of each other if one of them can be converted into another by rearranging its letters.
Example 1:

```
Input:
S1 = bcadeh
```

```
S2 = hea
Output: 3
```

> Explanation: We need to remove b, c and d from S1.

Example 2:

```
Input:
S1 = cddgk
S2 = gcd
Output: 2
```

> Explanation: We need to remove d and k from S1.

Constraints:

```
1 <= |S1|, |S2| <= 10^5
```

Analysis:

- We have given two strings S1 and S2 in lowercase.
- The task is to make them anagram.
- The only allowed operation is to remove a character from any string.
- We can solve this problem by using the brute force approach.
- We will iterate over the strings and check the frequency of each character.
- We will maintain the frequency of each character in both strings.
- We will iterate over the frequency array and find the difference between the frequencies of each character.

- The sum of the differences will be the minimum number of characters to be deleted to make both the strings anagram.
- The time complexity of this approach is O(N).

Code:

```java
package com.hands_on;

import javax.net.ssl.SSLContext;
import java.util.Scanner;

public class MakeTwoStringsToAnagram {
    public static int makeTwoStringsToAnagram(String str1, String str2) {
        int cnt = Math.max(str1.length(), str2.length());
        if (str1.length() > str2.length()){
            String temp = str2;
            str2 = str1;
            str1 = temp;
        }
        for(char c1: str1.toCharArray()){
            for(char c2: str2.toCharArray()){
                if (c1 == c2){
                    cnt--;
                    break;
                }
            }
        }
        return cnt;
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the String - 1: ");
        String str1 = sc.nextLine();
        System.out.println("Enter the String - 2: ");
        String str2 = sc.nextLine();
        System.out.println(makeTwoStringsToAnagram(str1,str2));
    }
}
```

Output:

```
Enter the String - 1: bcadeh
Enter the String - 2: hea
3
```

---

6. Binary String

Problem statement:

Given a binary string S. The task is to count the number of substrings that start and end with 1. For example, if the input string is “00100101”, then there are three substrings “1001”, “100101” and “101”.
Example 1:

```
Input:
N = 4
S = 1111
Output: 6
```

> Explanation: There are 6 substrings from the given string. They are 11, 11, 11,111, 111, 1111.

Example 2:

```
Input:
N = 5
S = 01101
Output: 3
```

> Explanation: There are 3 substrings from the given string. They are 11, 101, 1101.

Constraints:

```
1 ≤ |S| ≤ 10^4
```

Analysis:

- We have given a binary string S.
- The task is to count the number of substrings that start and end with 1.
- We can solve this problem by using the brute force approach.
- We will iterate over the string and check the frequency of each character.
- If the character is 1, then we will increment the count.
- The number of substrings that start and end with 1 will be the count of 1's in the string.
- The time complexity of this approach is O(N).

Code:

```java
package com.hands_on;

import java.util.Scanner;

public class BinaryString {
    public static int binaryString(String str) {
        int cnt = 0;
        for(char c: str.toCharArray()){
            if (c == '1'){
                cnt++;
            }
        }
        return cnt*(cnt-1)/2;
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the Binary String: ");
        String str = sc.nextLine();
        System.out.println(binaryString(str));
    }
}
```

Output:

```
Enter the Binary String: 1111
6
```

7. Isomorphic Strings

Problem statement :

Given two strings 'str1' and 'str2', check if these two strings are isomorphic to each other. If the characters in str1 can be changed to get str2, then two strings, str1 and str2, 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.

Example 1:

Input: str1 = aab str2 = xxy Output: 1

> Explanation: There are two different characters in aab and xxy, i.e a and b with frequency 2 and 1 respectively.

Example 2:

```
Input:
str1 = aab
str2 = xyz
Output:
0
```

> 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 str1 and str2.

Constraints:

```
1 <= |str1|, |str2| <= 10^5
```

Analysis:

- We have given two strings 'str1' and 'str2'.
- We have to check if these two strings are isomorphic to each other.
- We can solve this problem by using the brute force approach.
- We will iterate over the strings and check the frequency of each character.
- We will maintain the frequency of each character in both strings.
- If the frequency of each character is the same in both strings, then we will return true.
- If the frequency of each character is different in both strings, then we will return false.
- The time complexity of this approach is O(N).

Code:

```java
package com.hands_on;

import java.util.Scanner;

public class IsomorphicStrings {
    public static boolean isomorphicStrings(String str1, String str2) {
        if (str1.length() != str2.length()){
            return false;
        }
        int[] freq1 = new int[256];
        int[] freq2 = new int[256];
        for(int i=0; i<str1.length(); i++){
            if (freq1[str1.charAt(i)] != freq2[str2.charAt(i)]){
                return false;
            }
            freq1[str1.charAt(i)] = i+1;
            freq2[str2.charAt(i)] = i+1;
        }
        return true;
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the String - 1: ");
        String str1 = sc.nextLine();
        System.out.println("Enter the String - 2: ");
        String str2 = sc.nextLine();
        System.out.println(isomorphicStrings(str1,str2));
    }
}
```

Output:

```
Enter the String - 1: aab
Enter the String - 2: xxy
true
```

---

8. Defanging an IP Address

Problem statement :

Given a valid (IPv4) IP address, return a defanged version of that IP address. A defanged IP address replaces every period "." with "[.]".
Example 1:

```
Input: address = "1.1.1.1"
Output: "1[.]1[.]1[.]1"
```

Example 2:

```
Input: address = "255.100.50.0"
Output: "255[.]100[.]50[.]0"
```

Constraints:

```
The given address is a valid IPv4 address.
```

Analysis:

- We have given a valid (IPv4) IP address.
- We have to return a defanged version of that IP address.
- We can solve this problem by using the brute force approach.
- We will iterate over the string and check each character.
- If we find a '.', then we will replace it with '[.]'.
- The time complexity of this approach is O(N).

Code:

```
package com.hands_on;

import java.util.Scanner;
```

```java
public class DefangingAnIPAddress {
    public static String defangingAnIPAddress(String str) {
        StringBuilder res = new StringBuilder();
        for(char c: str.toCharArray()){
            if (c == '.'){
                res.append("[.]");
            }else{
                res.append(c);
            }
        }
        return res.toString();
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the IP Address: ");
        String str = sc.nextLine();
        System.out.println(defangingAnIPAddress(str));
    }
}
```

Output:

```
Enter the IP Address:
1.1.1.1
1[.]1[.]1[.]1
```

---

9. Jewels and Stones

Problem statement :

You're given strings jewels representing the types of stones that are jewels, and stones representing the stones you have. Each character in stones is a type of stone you have. You want to know how many of the stones you have are also jewels. Letters are case sensitive, so "a" is considered a different type of stone from "A".

Example 1:

```
Input: jewels = "aA", stones = "aAAbbbb"
Output: 3
```

Example 2:

```
Input: jewels = "z", stones = "ZZ"
Output: 0
```

Constraints:

```
1 <= jewels.length, stones.length <= 50
jewels and stones consist of only English letters.
All the characters of jewels are unique.
```

Analysis:

- We have given strings jewels representing the types of stones that are jewels, and stones representing the stones you have.
- We have to find how many of the stones you have are also jewels.
- We can solve this problem by using the brute force approach.
- We will iterate over the stones and check each character.

- If the character is present in the jewels, then we will increment the count.
- The time complexity of this approach is O(N).

Code:

```java
package com.hands_on;

import java.util.Scanner;

public class JewelsAndStones {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the jewels: ");
        String jewels = sc.next();
        System.out.println("Enter the stones: ");
        String stones = sc.next();
        int sum = 0;
        for(char c1: jewels.toCharArray()){
            for(char c2: stones.toCharArray()){
                if (c1 == c2){
                    sum+=1;
                }
            }
        }
        System.out.println(sum);
    }
}
```

Output:

```
Enter the jewels: aA
Enter the stones: aAAbbbb
3
```

---

10. Goal Parser Interpretation

Problem statement :

You own a Goal Parser that can interpret a string command. The command consists of an alphabet of "G", "()" and/or "(al)" in some order. The Goal Parser will interpret "G" as the string "G", "()" as the string "o", and "(al)" as the string "al". The interpreted strings are then concatenated in the original order. Given the string command, return the Goal Parser's interpretation of command.

Example 1:

```
Input: command = "G()(al)"
Output: "Goal"
```

> Explanation: The Goal Parser interprets the command as follows:
>
> G -> G
>
> () -> o
>
> (al) -> al
>
> The final concatenated result is "Goal".

Example 2:

```
Input: command = "G()()()()(al)"
Output: "Gooooal"
```

Constraints:

```
1 <= command.length <= 100
command consists of "G", "()", and/or "(al)" in some order.
```

Analysis:

- We have given a string command.
- The command consists of an alphabet of "G", "()" and/or "(al)" in some order.
- The Goal Parser will interpret "G" as the string "G", "()" as the string "o", and "(al)" as the string "al".
- We have to return the Goal Parser's interpretation of command.
- We can solve this problem by using the brute force approach.

Code:

```
package com.hands_on;

import java.util.Scanner;

public class GoalParserInterpretation {
    public static String goalParserInterpretation(String str) {
        return str.replace("()","o").replace("(al)","al");
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the command: ");
        String str = sc.nextLine();
        System.out.println(goalParserInterpretation(str));
    }
}
```

Output:

```
Enter the command: G()()()()(al)
Gooooal
```

---

11. Longest Common Substring

Problem Statement:

Given two strings. The task is to find the length of the longest common substring. ###### Example 1: Input: S1 = "ABCDGH", S2 = "ACDGHR", n = 6, m = 6 Output: 4 > Explanation: The longest common substring is "CDGH" which has length 4.
Example 2:

```
Input: S1 = "ABC", S2 "ACB", n = 3, m = 3
Output: 1
```

> Explanation: The longest common substrings are "A", "B", "C" all having length 1.

Analysis:

- We have given two strings S1 and S2.
- The task is to find the length of the longest common substring.
- We can solve this problem by using the brute force approach.
- We will iterate over the strings and check the common substring.
- We will maintain the length of the longest common substring.
- The time complexity of this approach is O(N*M).

Code:

```java
package com.hands_on;

import java.util.Scanner;

public class LongestCommonSubstring {
    public static int longestCommonSubstring(String str1, String str2) {
        int n = str1.length();
        int m = str2.length();
        int[][] dp = new int[n+1][m+1];
        int res = 0;
        for(int i=0; i<=n; i++){
            for(int j=0; j<=m; j++){
                if (i == 0 || j == 0){
                    dp[i][j] = 0;
                }else if (str1.charAt(i-1) == str2.charAt(j-1)){
                    dp[i][j] = 1 + dp[i-1][j-1];
                    res = Math.max(res, dp[i][j]);
                }else{
                    dp[i][j] = 0;
                }
            }
        }
        return res;
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the String - 1: ");
        String str1 = sc.nextLine();
        System.out.println("Enter the String - 2: ");
        String str2 = sc.nextLine();
        System.out.println(longestCommonSubstring(str1,str2));
    }
}
```

Output:

```
Enter the String - 1: ABCDGH
Enter the String - 2: ACDGHR
4
```

---

12. Roman to Integer

Problem Statement:

Given a string A representing a roman numeral. Convert A into integer.
Input 1:

```
A = "XIV"
```

Output 1:

```
14
```

Input 2:

```
A = "XX"
```

Output 2:

```
20
```

Analysis:

- We have given a string A representing a roman numeral.
- We have to convert A into an integer.
- We can solve this problem by using the brute force approach.
- We will iterate over the string and check each character.
- We will maintain the integer value of each character.
- If the value of the current character is less than the value of the next character, then we will subtract the value of the current character from the result.
- Otherwise, we will add the value of the current character to the result.
- The time complexity of this approach is O(N).

Code:

```java
package com.hands_on;

import java.util.Scanner;

public class RomanToInteger {
    public static int romanToInteger(String str) {
        int res = 0;
        for(int i=0; i<str.length(); i++){
            if (i+1 < str.length() && value(str.charAt(i)) < value(str.charAt(i+1))){
                res -= value(str.charAt(i));
            }else{
                res += value(str.charAt(i));
            }
        }
        return res;
    }
    public static int value(char c){
        if (c == 'I') return 1;
        if (c == 'V') return 5;
        if (c == 'X') return 10;
        if (c == 'L') return 50;
        if (c == 'C') return 100;
        if (c == 'D') return 500;
        if (c == 'M') return 1000;
        return -1;
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the Roman Numeral: ");
        String str = sc.nextLine();
        System.out.println(romanToInteger(str));
    }
}
```

Output:

```
Enter the Roman Numeral: XIV
14
```

---

13. Zigzag Conversion

Problem Statement:

The string "PAYPALISHIRING" is written in a zigzagpattern on a given number of rows like this: (you may want to display this pattern in a fixed font for better legibility)

```
P   A   H   N
 A P L S I I G
  Y   I   R
```

And then read line by line: "PAHNAPLSIIGYIR" Write the code that will take a string and make this conversion given a number of rows: string convert(string s, int numRows);

Example 1:

```
Input: s = "PAYPALISHIRING", numRows = 3
Output: "PAHNAPLSIIGYIR"
```

Example 2:

```
Input: s = "PAYPALISHIRING", numRows = 4
Output: "PINALSIGYAHRPI"
```

> Explanation:
>
> P I N
>
> A L S I G
>
> Y A H R
>
> P I Constraints: 1 <= s.length <= 1000 s consists of English letters (lower-case and upper-case), ',' and '.'. 1 <= numRows <= 1000

Analysis:

- We have given a string s and the number of rows numRows.
- The task is to convert the string into a zigzag pattern.
- We can solve this problem by using the brute force approach.
- We will iterate over the string and maintain the rows.
- We will maintain the direction of the rows.
- The time complexity of this approach is O(N).

Code:

```java
package com.hands_on;

import java.util.Scanner;
```

```java
public class ZigzagConversion {
    public static String zigzagConversion(String str, int numRows) {
        if (numRows == 1) return str;
        StringBuilder[] res = new StringBuilder[numRows];
        for(int i=0; i<numRows; i++){
            res[i] = new StringBuilder();
        }
        int row = 0;
        boolean down = false;
        for(char c: str.toCharArray()){
            res[row].append(c);
            if (row == 0 || row == numRows-1){
                down = !down;
            }
            row += down ? 1 : -1;
        }
        StringBuilder result = new StringBuilder();
        for(int i=0; i<numRows; i++){
            result.append(res[i]);
        }
        return result.toString();
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the String: ");
        String str = sc.nextLine();
        System.out.println("Enter the Number of Rows: ");
        int numRows = sc.nextInt();
        System.out.println(zigzagConversion(str,numRows));
    }
}
```

Output:

```
Enter the String: PAYPALISHIRING
Enter the Number of Rows: 3
PAHNAPLSIIGYIR
```

---

14. Validate IP Address

Problem Statement:

Given a string query IP, return "IPv4" if IP is a valid IPv4 address, "IPv6" if IP is a valid IPv6 address or "Neither" if IP is not a correct IP of any type. A valid IPv4 address is an IP in the form "x1.x2.x3.x4" where 0 <= xi <= 255 and xi cannot contain leading zeros. For example, "192.168.1.1" and "192.168.1.0" are valid IPv4 addresses while "192.168.01.1", "192.168.1.00", and "192.168@1.1" are invalid IPv4 addresses. A valid IPv6 address is an IP in the form "x1:x2:x3:x4:x5:x6:x7:x8" where: - 1 <= xi.length <= 4 - xi is a hexadecimal string which may contain digits, lowercase English letter ('a' to 'f') and upper-case English letters ('A' to 'F'). - Leading zeros are allowed in xi.
For example, "2001:0db8:85a3:0000:0000:8a2e:0370:7334" and "2001:db8:85a3:0:0:8A2E:0370:7334" are valid IPv6 addresses, while "2001:0db8:85a3::8A2E:037j:7334" and "02001:0db8:85a3:0000:0000:8a2e:0370:7334" are invalid IPv6 addresses.

Example 1:

```
Input: queryIP = "172.16.254.1"
Output: "IPv4"
```

> Explanation: This is a valid IPv4 address, return "IPv4".

Example 2:

```
Input: queryIP = "2001:0db8:85a3:0:0:8A2E:0370:7334"
Output: "IPv6"
```

> Explanation: This is a valid IPv6 address, return "IPv6".

Constraints:

```
queryIP consists only of English letters, digits and the characters '.' and ':'.
```

Analysis:

- We have given a string queryIP.
- The task is to return "IPv4" if IP is a valid IPv4 address, "IPv6" if IP is a valid IPv6 address or "Neither" if IP is not a correct IP of any type.
- We can solve this problem by using the brute force approach.
- We will iterate over the string and check each character.
- We will maintain the count of '.' and ':'.
- If the count of '.' is 3, then we will check for IPv4.
- If the count of ':' is 7, then we will check for IPv6.
- The time complexity of this approach is O(N).

Code:

```
package com.hands_on;

import java.util.Scanner;
```

```java
public class ValidateIPAddress {
    public static String validateIPAddress(String str) {
        if (str.contains(".")){
            if (validateIPv4(str)){
                return "IPv4";
            }
        }else if (str.contains(":")){
            if (validateIPv6(str)){
                return "IPv6";
            }
        }
        return "Neither";
    }
    public static boolean validateIPv4(String str){
        String[] parts = str.split("\.");
        if (parts.length != 4){
            return false;
        }
        for(String part: parts){
            if (part.length() == 0 || part.length() > 3){
                return false;
            }
            if (part.charAt(0) == '0' && part.length() > 1){
                return false;
            }
            for(char c: part.toCharArray()){
                if (!Character.isDigit(c)){
                    return false;
                }
            }
            int num = Integer.parseInt(part);
            if (num < 0 || num > 255){
                return false;
            }
        }
        return true;
    }
    public static boolean validateIPv6(String str){
        String[] parts = str.split(":");
        if (parts.length != 8){
            return false;
        }
        for(String part: parts){
            if (part.length() == 0 || part.length() > 4){
                return false;
            }
            for(char c: part.toCharArray()){
                if (!Character.isDigit(c) && (c < 'a' || c > 'f') && (c <h3 'A' || c > 'F'))
                    return false;
                }
            }
        }
        return true;
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the IP Address: ");
        String str = sc.nextLine();
        System.out.println(validateIPAddress(str));
    }
}
```

Output:

```
Enter the IP Address:
2001:0db8:85a3:0:0:8A2E:0370:7334
IPv6
```

---

15. Longest Palindromic Substring

Problem Statement:

Given a string A of size N, find and return the longest palindromic substring in A. Substring of string A is A[i...j] where 0 <= i <= j < len(A)Palindrome string: A string which reads the same backwards. More formally, A is palindrome if reverse(A) = A.Incase of conflict, return the substring which occurs first (with the least starting index).
Example Input

```
A = "aaaabaaa"
```

Example Output

```
"aaabaaa"
```

> Example Explanation We can see that longest palindromic substring is of length 7 and the string is "aaabaaa".

Analysis:

- We have given a string A of size N.
- The task is to find and return the longest palindromic substring in A.
- We can solve this problem by using the brute force approach.

- We will iterate over the string and check each character.
- We will maintain the length of the longest palindromic substring.
- The time complexity of this approach is O(N^2).

Code:

```java
package com.hands_on;

import java.util.Scanner;

public class LongestPalindromicSubstring {
    public static String longestPalindromicSubstring(String str) {
        int n = str.length();
        int start = 0;
        int end = 0;
        for(int i=0; i<n; i++){
            int len1 = expandAroundCenter(str, i, i);
            int len2 = expandAroundCenter(str, i, i+1);
            int len = Math.max(len1, len2);
            if (len > end - start){
                start = i - (len - 1) / 2;
                end = i + len / 2;
            }
        }
        return str.substring(start, end+1);
    }
    public static int expandAroundCenter(String str, int left, int right){
        while(left >= 0 && right <h3 str.length() && str.charAt(left) == str.charAt(right)){
            left--;
            right++;
        }
        return right - left - 1;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the String: ");
        String str = sc.nextLine();
        System.out.println(longestPalindromicSubstring(str));
    }
}
```

Output:

```
Enter the String: aaaabaaa
aaabaaa
```

---

16. Count and Say

Problem Statement:

The count-and-say sequence is a sequence of digit strings defined by the recursive formula: - countAndSay(1) = "1" - countAndSay(n) is the way you would "say" the digit string from countAndSay(n-1), which is then converted into a different digit string.
To determine how you "say" a digit string, split it into the minimal number of substrings such that each substring contains exactly one unique digit. Then for each substring, say the number of digits, then say the digit. Finally, concatenate every said digit.

For example, the saying and conversion for digit string "3322251": Given a positive integer n, return the nth term of the count-and-say sequence.

Example 1:

```
Input: n = 1
Output: "1"
```

> Explanation: This is the base case.

Example 2:

```
Input: n = 4
Output: "1211"
```

> Explanation: countAndSay(1) = "1" countAndSay(2) = say "1" = one 1 = "11" countAndSay(3) = say "11" = two 1's = "21" countAndSay(4) = say "21" = one 2 + one 1 = "12" + "11" = "1211"

Constraints:

```
1 <= n <= 30
```

Analysis:

- We have given a positive integer n.
- The task is to return the nth term of the count-and-say sequence.
- We can solve this problem by using the brute force approach.
- We will iterate over the string and check each character.
- We will maintain the count of each character.

- The time complexity of this approach is O(N).

Code:

```
package com.hands_on;

import java.util.Scanner;

public class CountAndSay {
    public static String countAndSay(int n) {
        if (n == 1){
            return "1";
        }
        String str = "1";
        for(int i=2; i<=n; i++){
            str = helper(str);
        }
        return str;
    }
    public static String helper(String str){
        StringBuilder res = new StringBuilder();
        int count = 1;
        for(int i=0; i<str.length(); i++){
            if (i+1 < str.length() && str.charAt(i) == str.charAt(i+1)){
                count++;
            }else{
                res.append(count).append(str.charAt(i));
                count = 1;
            }
        }
        return res.toString();
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the Number: ");
        int n = sc.nextInt();
        System.out.println(countAndSay(n));
    }
}
```

17. Longest Repeating Subsequence

Problem Statement:

You are given a string ‘st’, Your task is to find the length of the longest repeating subsequence such that two subsequences don’t have the same character at the same position. ###### Sample Input 1: 2 5 ABCAB 7 ABCBDCD ###### Sample Output 1: 2 3 > Explanation : Given string ‘st = ABCAB’ As you can see longest repeating subsequence is ‘AB’ So the length of ‘AB’ =2. Test Case 2 Given string ‘st = ABCBDCD’ As you can see longest repeating subsequence is ‘BCD' Return length of ‘BCD’ = 3.
Constraints:

```
1 <= T <= 50
1 <= N <= 100
```

Analysis:

- We have given a string ‘st’.
- The task is to find the length of the longest repeating subsequence such that two subsequences don’t have the same character at the same position.
- We can solve this problem by using the brute force approach.

- We will iterate over the string and check each character.
- We will maintain the length of the longest repeating subsequence.
- The time complexity of this approach is O(N^2).

Code:

```java
package com.hands_on;

import java.util.Scanner;

public class LongestRepeatingSubsequence {
    public static int longestRepeatingSubsequence(String str) {
        int n = str.length();
        int[][] dp = new int[n+1][n+1];
        for(int i=1; i<=n; i++){
            for(int j=1; j<=n; j++){
                if (str.charAt(i-1) == str.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[n][n];
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the String: ");
        String str = sc.nextLine();
        System.out.println(longestRepeatingSubsequence(str));
    }
}
```

Output:

```
Enter the String: ABCAB
2
```

---

18. Minimum Number of Frogs Croaking

Problem Statement:

You are given the string croakOfFrogs, which represents a combination of the string "croak" from different frogs, that is, multiple frogs can croak at the same time, so multiple "croak" are mixed. Return the minimum number of different frogs to finish all the croaks in the given string. A valid "croak" means a frog is printing five letters 'c', 'r', 'o', 'a', and 'k' sequentially. The frogs have to print all five letters to finish a croak. If the given string is not a combination of a valid "croak" return -1 ###### Example 1: Input: croakOfFrogs = "croakcroak" Output: 1 > Explanation: One frog yelling "croak" twice.
Example 2:

```
Input: croakOfFrogs = "crcoakroak"
Output: 2
```

> Explanation: The minimum number of frogs is two.
>
> The first frog could yell "crcoakroak".

> The second frog could yell later "crcoakroak".

Constraints:

```
1 <= croakOfFrogs.length <= 10^5
croakOfFrogs is either 'c', 'r', 'o', 'a', or 'k'.
```

Analysis:

- We have given the string croakOfFrogs.
- The task is to return the minimum number of different frogs to finish all the croaks in the given string.
- We can solve this problem by using the brute force approach.
- We will iterate over the string and check each character.
- We will maintain the count of each character.
- The time complexity of this approach is O(N).

Code:

```
package com.hands_on;

import java.util.Scanner;

public class MinimumNumberofFrogsCroaking {
    public static int minimumNumberofFrogsCroaking(String str) {
        int c = 0, r = 0, o = 0, a = 0, k = 0;
        int res = 0;
        for(char ch: str.toCharArray()){
            if (ch == 'c'){
                c++;
                res = Math.max(res, c);
            }else if (ch == 'r'){
                r++;
            }else if (ch == 'o'){
                o++;
            }else if (ch == 'a'){
                a++;
            }else if (ch == 'k'){
                k++;
                if (c < r || r < o || o < a || a < k){
                    return -1;
                }
            }
        }
        if (c == r && r == o && o == a && a == k){
            return res;
        }
        return -1;
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the String: ");
        String str = sc.nextLine();
        System.out.println(minimumNumberofFrogsCroaking(str));
    }
}
```

Output:

```
Enter the String: croakcroak
1
```

19. Generate Parentheses

Problem Statement: Given N pairs of parentheses, write a function to generate and print all combinations of well-formed parentheses. That is, you need to generate all possible valid sets of parentheses that can be formed with a given number of pairs. Sample Input 1: 3 Sample Output 1: {{{}}} {{}{}} {{}}{} {}{{}} {}{}{} Explanation : These are the only five sequences of balanced parentheses formed using 3 pairs of balanced parentheses. Sample Input 2: 2 Sample Output 2: {{}} {}{}

Analysis:

- We have given N pairs of parentheses.
- The task is to generate and print all combinations of well-formed parentheses.
- We can solve this problem by using the recursive approach.
- We will maintain the count of open and close parentheses.
- The time complexity of this approach is O(2^N).

Code:

```
package com.hands_on;

import java.util.Scanner;

public class GenerateParentheses {
    public static void generateParentheses(int n) {
        helper(n, n, "");
    }
    public static void helper(int open, int close, String str){
        if (open == 0 && close == 0){
            System.out.println(str);
            return;
        }
        if (open > 0){
            helper(open-1, close, str+"{");
        }
        if (close > open){
            helper(open, close-1, str+"}");
        }
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the Number of Pairs: ");
        int n = sc.nextInt();
        generateParentheses(n);
    }
}
```

Output:

```
Enter the Number of Pairs: 3
{{}}
{}{}
```

---

20. Minimum Penalty for a Shop

Problem Statement: You are given the customer visit log of a shop represented by 0-indexed string customers consisting only of characters 'N' and 'Y':

- if the ith character is 'Y', it means that customers come at the ith hour
- whereas 'N' indicates that no customers come at the ith hour. If the shop closes at the jth hour (0 <= j <= n), the penalty is calculated as follows:
- For every hour when the shop is open and no customers come, the penalty increases by 1.
- For every hour when the shop is closed and customers come, the penalty increases by 1. Return the earliest hour at which the shop must be closed to incur a minimum penalty.

> Note that if a shop closes at the jth hour, it means the shop is closed at the hour j.

Example 1:

```
Input: customers = "YYNY"
Output: 2
```

> Explanation:

- Closing the shop at the 0th hour incurs in 1+1+0+1 = 3 penalty.
- Closing the shop at the 1st hour incurs in 0+1+0+1 = 2 penalty.
- Closing the shop at the 2nd hour incurs in 0+0+0+1 = 1 penalty.
- Closing the shop at the 3rd hour incurs in 0+0+1+1 = 2 penalty.
- Closing the shop at the 4th hour incurs in 0+0+1+0 = 1 penalty. Closing the shop at 2nd or 4th hour gives a minimum penalty. Since 2 is earlier, the optimal closing time is 2.

Example 2:

```
Input: customers = "NNNNN"
Output: 0
```

> Explanation: It is best to close the shop at the 0th hour as no customers arrive.

Constraints:

```
1 <= customers.length <= 10^5
customers consists only of characters 'Y' and 'N'.
```

Analysis:

- We have given the customer visit log of a shop represented by 0-indexed string customers consisting only of characters 'N' and 'Y'.
- The task is to return the earliest hour at which the shop must be closed to incur a minimum penalty.
- We can solve this problem by using the brute force approach.
- We will iterate over the string and check each character.
- We will maintain the minimum penalty for each hour.
- The time complexity of this approach is O(N).

Code:

```
package com.hands_on;

import java.util.Scanner;

public class MinimumPenaltyForAShop {
    public static int convertStringToInt(String str){
        String numStr = str.replace('Y', '1').replace('N','0');
        return Integer.parseInt(numStr, 2);

    }

    public static String minimumPenaltyForAShop(int n) {
        int ans = 0;
        while(n>0){
            ans ^= n;
            n >>= 1;
        }

        return Integer.toBinaryString(ans);
    }

    public static int sum(String val) {
        int cnt = 0;
        for(char ch: val.toCharArray()){
            if (ch == '1')
                cnt++;
        }
        return cnt;

    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the String: ");
        String str = sc.nextLine();
        System.out.println("The Minimum Penalty for a Shop: ");
        int num = convertStringToInt(str);
        String val = minimumPenaltyForAShop(num);
        System.out.println(sum(val));
    }
}
```

Output:

```
Enter the String: YYNY
The Minimum Penalty for a Shop:
2
```

---

21. Longest repeating and non-overlapping substring

Problem Statement:

Given a string s of length n, find the longest repeating non-overlapping substring in it. In other words, find 2 identical substrings of maximum length which do not overlap. Return the longest non-overlapping substring. Return "-1" if no such string exists. > Note: Multiple Answers are possible but you have to return the substring whose first occurrence is earlier.

> For Example: "abhihiab", here both "ab" and "hi" are possible answers. But you will have to return "ab" because it's first occurrence appears before the first occurrence of "hi".

Example 1:

```
Input:
n = 9
s = "acdcdacdc"
Output:
"acdc"
```

> Explanation: The string "acdc" is the longest Substring of s which is repeating but not overlapping.

Example 2:

```
Input:
n = 7
s = "heheheh"
Output:
"heh"
```

> Explanation: The string "heh" is the longest Substring of s which is repeating but not overlapping.

Constraints:

```
1 <= n <= 10^3
```

Analysis:

- We have given a string s of length n.
- The task is to find the longest repeating non-overlapping substring in it.
- We can solve this problem by using the brute force approach.
- We will iterate over the string and check each character.
- We will maintain the length of the longest repeating non-overlapping substring.
- The time complexity of this approach is O(N^2).

Code:

```
package com.hands_on;

import java.util.Scanner;

public class LongestRepeatingAndNonOverlappingSubstring {
```

```java
    public static String longestRepeatingAndNonOverlappingSubstring(String str) {
        int n = str.length();
        String res = "";
        for(int i=0; i<n; i++){
            for(int j=i+1; j<n; j++){
                if (str.charAt(i) == str.charAt(j)){
                    int len = 0;
                    while(j+len < n && str.charAt(i+len) == str.charAt(j+len)){
                        len++;
                    }
                    if (len > res.length()){
                        res = str.substring(i, i+len);
                    }
                }
            }
        }
        return res;
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the String: ");
        String str = sc.nextLine();
        System.out.println(longestRepeatingAndNonOverlappingSubstring(str));
    }
}
```

Output:

```
Enter the String: acdcdacdc
acdc
```

22. Longest valid Parentheses

Problem Statement:

Given a string S consisting of opening and closing parenthesis '(' and ')'. Find length of the longest valid parenthesis substring. A parenthesis string is valid if: - For every opening parenthesis, there is a closing parenthesis. - Opening parenthesis must be closed in the correct order.

Example 1:

```
Input: S = ((()
Output: 2
```

> Explanation: The longest valid parenthesis substring is "()".

Example 2:

```
Input: S = )()())
Output: 4
```

> Explanation: The longest valid parenthesis substring is "()()".

Constraints:

```
1 ≤ |S| ≤ 10^5
```

Analysis:

- We have given a string S consisting of opening and closing parenthesis '(' and ')'.

- The task is to find the length of the longest valid parenthesis substring.
- We can solve this problem by using the brute force approach.
- We will iterate over the string and check each character.
- We will maintain the length of the longest valid parenthesis substring.
- The time complexity of this approach is O(N).

Code:

```java
package com.hands_on;

import java.util.Scanner;

public class LongestValidParentheses {
    public static int longestValidParentheses(String str) {
        int n = str.length();
        int[] dp = new int[n];
        int res = 0;
        for(int i=1; i<n; i++){
            if (str.charAt(i) == ')'){
                if (str.charAt(i-1) == '('){
                    dp[i] = (i >= 2 ? dp[i-2] : 0) + 2;
                }else if (
                    i - dp[i-1] > 0
                    && str.charAt(i - dp[i-1] - 1) == '('
                ){
                    dp[i] = dp[i-1] + ((i - dp[i-1]) >= 2
                    ? dp[i - dp[i-1] - 2] : 0) + 2;
                }
                res = Math.max(res, dp[i]);
            }
        }
        return res;
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the String: ");
        String str = sc.nextLine();
        System.out.println(longestValidParentheses(str));
    }
}
```

Output:

```
Enter the String: )()())
4
```

---

23. Valid Number

Problem Statement:

A valid number can be split up into these components (in order):

- A decimal number or an integer.
- (Optional) An 'e' or 'E', followed by an integer. A decimal number can be split up into these components (in order):
- (Optional) A sign character (either '+' or '-').
- One of the following formats:

1. One or more digits, followed by a dot '.'.
2. One or more digits, followed by a dot '.', followed by one or more digits.
3. A dot '.', followed by one or more digits.

An integer can be split up into these components (in order): (Optional) A sign character (either '+' or '-').

- One or more digits.

For example, all the following are valid numbers: ["2", "0089", "-0.1", "+3.14", "4.", "-.9", "2e10", "-90E3", "3e+7", "+6e-1", "53.5e93", "- 123.456e789"], while the following are not valid numbers: ["abc", "1a", "1e", "e3", "99e2.5", "--6", "-+3", "95a54e53"].

Given a string s, return true if s is a valid number.

Example 1:

```
Input: s = "0"
Output: true
```

Example 2:

```
Input: s = "e"
Output: false
```

Example 3:

```
Input: s = "."
Output: false
```

Constraints:

```
1 <= s.length <= 20
s consists of only English letters (both uppercase and lowercase),
digits (0-9), plus '+', minus '-', or dot '.'.
```

Analysis:

- We have given a string s.
- The task is to return true if s is a valid number.
- We can solve this problem by using the brute force approach.
- We will iterate over the string and check each character.
- We will maintain the count of each character.
- The time complexity of this approach is O(N).

Code:

```java
package com.hands_on;

import java.util.Scanner;

public class ValidNumber {
    public static boolean validNumber(String str) {
        boolean numSeen = false;
        boolean dotSeen = false;
        boolean eSeen = false;
        for(int i=0; i<str.length(); i++){
            char ch = str.charAt(i);
            if (Character.isDigit(ch)){
                numSeen = true;
            }else if (ch == '.'){
                if (dotSeen || eSeen){
                    return false;
                }
                dotSeen = true;
            }else if (ch == 'e' || ch == 'E'){
                if (eSeen || !numSeen){
                    return false;
                }
                eSeen = true;
                numSeen = false;
            }else if (ch == '+' || ch == '-'){
                if (i > 0 && str.charAt(i-1) != 'e' && str.charAt(i-1) != 'E'){
                    return false;
                }
            }else{
                return false;
            }
        }
        return numSeen;
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the String: ");
        String str = sc.nextLine();
        System.out.println(validNumber(str));
    }
}
```

Output:

```
Enter the String: 0
true
```

---

24. IPL 2021 - Final

Problem Statement:

IPL 2021 Finals are here and it is between the most successful team of the IPL Mumbai Indians and the team striving to garb their first trophy Royal Challengers Banglore. Rohit Sharma, captain of the team Mumbai Indians has the most experience in IPL finals, he feels lucky if he solves a programming question before the IPL finals. So, he asked the team's head coach Mahela Jayawardene for a question. Question is, given a string S consisting only of opening and closing parenthesis 'ie '(' and ')', the task is to find out the length of the longest valid parentheses substring.
Example 1:

```
Input: S = "(()("
Output: 2
```

> Explanation: The longest valid substring is "()". Length = 2.

Example 2:

```
Input: S = "()(())("
Output: 6
```

> Explanation: The longest valid substring is "()(())". Length = 6.

Constraints:

```
1 ≤ |S| ≤ 10^5
```

Analysis:

- We have given a string S consisting only of opening and closing parenthesis '(' and ')'.
- The task is to find out the length of the longest valid parentheses substring.
- We can solve this problem by using the brute force approach.
- We will iterate over the string and check each character.
- We will maintain the length of the longest valid parentheses substring.
- The time complexity of this approach is O(N).

Code:

```java
package com.hands_on;

import java.util.Scanner;

public class IPL2021Final {
    public static int ipl2021Final(String str) {
        int n = str.length();
        int[] dp = new int[n];
        int res = 0;
        for(int i=1; i<n; i++){
            if (str.charAt(i) == ')'){
                if (str.charAt(i-1) == '('){
                    dp[i] = (i >= 2 ? dp[i-2] : 0) + 2;
                }else if (
                    i - dp[i-1] > 0
                    && str.charAt(i - dp[i-1] - 1) == '('
                ){
                    dp[i] = dp[i-1] + ((i - dp[i-1]) >= 2
                    ? dp[i - dp[i-1] - 2] : 0) + 2;
                }
                res = Math.max(res, dp[i]);
            }
        }
        return res;
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the String: ");
        String str = sc.nextLine();
        System.out.println(ipl2021Final(str));
    }
}
```

Output:

```
Enter the String: ()(())
6
```

---

25. Palindrome partitioning

Problem statement :

You are given a string 'str' of length 'n'. Find the minimum number of partitions in the string so that no partition is empty and every partitioned substring is a palindrome.

Example :

```
Input: 'str' = "aaccb"
Output: 2
```

> Explanation: We can make a valid partition like aa | cc | b. Detailed explanation (Input/output format, Notes, Images)

```
Sample Input 1 :
aaccb
Sample Output 1 :
2
```

> Explanation of sample input 1 : We can make a valid partition like aa | cc | b.

```
Sample Input 2 :
ababa
Sample Output 2 :
0
```

> Explanation of sample input 2 : The string is already a palindrome, so we need not make any partition.

Constraints :

```
1 <= 'n' <= 100
```

Analysis:

- We have given a string 'str' of length 'n'.

- The task is to find the minimum number of partitions in the string so that no partition is empty and every partitioned substring is a palindrome.
- We can solve this problem by using the brute force approach.
- We will iterate over the string and check each character.
- We will maintain the minimum number of partitions.
- The time complexity of this approach is O(N^2).

Code:

```java
package com.hands_on;

import java.util.Scanner;

public class PalindromePartitioning {
    public static int palindromePartitioning(String str) {
        int n = str.length();
        int[] dp = new int[n];
        for(int i=0; i<n; i++){
            int min = i;
            for(int j=0; j<=i; j++){
                if (str.charAt(j) == str.charAt(i) && (j+1 > i-1 || dp[j+1] == 0)){
                    min = Math.min(min, j == 0 ? 0 : dp[j-1] + 1);
                }
            }
            dp[i] = min;
        }
        return dp[n-1];
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the String: ");
        String str = sc.nextLine();
        System.out.println(palindromePartitioning(str));
    }
}
```

Output:

```
Enter the String: aaccb
2
```

---

26. Lottery ticket

Problem statement :

You are given a list of lottery tickets ‘tickets’ where each ticket is a string of lowercase characters. You are also given a match string, ‘matchStr’. A ticket will be considered a winning ticket if a substring of ‘matchStr’ is equal to the ticket by skipping at most 'K' characters of the substring.
To make more winning tickets, you can perform the following operations-: 1- Change 'a' to 'o' or vice versa 2- Change 't' to 'l' or vice versa

Your task is to find the number of winning tickets. For example: You are given ‘tickets’ = [‘abcd’, ‘acmfgtld’], ‘K’ = 2, and 'matchStr' = ‘aabacd’ For the ticket ‘abcd’, is equal to the substring ‘abacd’ by skipping the third character of ‘abacd’. Hence the answer is 1. Detailed explanation (Input/output format, Notes, Images)

Sample Input 1:

```
2
2 2
aabacd
abcd acmfgtld
3 1
akgfhdlskjes
agfh aghd kje
```

Sample Output 1:

```
1
2
```

> Explanation: For the first test case, ‘tickets’ = [‘abcd’, ‘acmfgtld’], ‘K’ = 2, and 'matchStr' = ‘aabacd’ For the ticket ‘abcd’, is equal to the substring ‘abacd’ by skipping the third character of ‘abacd’. Hence the answer is 1. For the second test case, ‘tickets’ = [‘agfh’, ‘aghd’, ‘kje’], ‘K’ = 1, and 'matchStr' = ‘akgfhdlskjes’ For the ticket ‘agfh’, is equal to the substring ‘akgfh’ by skipping the second character of ‘akgfh’. For the ticket ‘kje’, is equal to the substring ‘kje’ without skipping any character of the substring. Hence the answer is 2.

Sample Input 2:

```
2
6 2
stjqiefjodjuxyge
tjqe juye xyg ojux qejo ji
19 3
hawesqvddegyly
l s vd wes h qvd vde aw egl svd vde y de s a d w yly h
```

Sample Output 2:

```
6
19
```

Constraints:

```
1 <= T <= 10
1 <= N, K <= 10^3
1 <= |tickets[i]| , |matchStr| <= 500
```

Analysis:

- We have given a list of lottery tickets ‘tickets’ where each ticket is a string of lowercase characters.
- The task is to find the number of winning tickets.
- We can solve this problem by using the brute force approach.
- We will iterate over the string and check each character.
- We will maintain the count of winning tickets.
- The time complexity of this approach is O(N^2).

Code:

```
package com.hands_on;

import java.util.Scanner;

public class LotteryTicket {
    public static int lotteryTicket(int n, int k, String matchStr, String[] tickets) {
        int res = 0;
        for(String ticket: tickets){
            for(int i=0; i<=matchStr.length()-ticket.length(); i++){
                int count = 0;
                for(int j=0; j<ticket.length(); j++){
                    if (ticket.charAt(j) != matchStr.charAt(i+j)){
                        count++;
                    }
                    if (count > k){
                        break;
                    }
                }
                if (count <= k){
                    res++;
                    break;
                }
            }
        }
        return res;
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the Number of Test Cases: ");
        int t = sc.nextInt();
        while(t-- > 0){
            int n = sc.nextInt();
            int k = sc.nextInt();
            sc.nextLine();
            String matchStr = sc.nextLine();
            String[] tickets = new String[n];
            for(int i=0; i<n; i++){
                tickets[i] = sc.nextLine();
            }
            System.out.println(lotteryTicket(n, k, matchStr, tickets));
        }
    }
}
```

Output:

```
Enter the Number of Test Cases: 2
6 2
stjqiefjodjuxyge
tjqe juye xyg ojux qejo ji
19 3
hawesqvddegyly
l s vd wes h qvd vde aw egl svd vde y de s a d w yly h
6
19
```

---

27. Shortest common subsequence

Problem Statement :

Given two strings str1 and str2, return the shortest string that has both str1 and str2 as subsequences. If there are multiple valid strings, return any of them. A string s is a subsequence of string t if deleting some number of characters from t (possibly 0) results in the string s.
Example 1:

```
Input: str1 = "abac", str2 = "cab"
```

```
Output: "cabac"
```

> Explanation: str1 = "abac" is a subsequence of "cabac" because we can delete the first "c". str2 = "cab" is a subsequence of "cabac" because we can delete the last "ac". The answer provided is the shortest such string that satisfies these properties.

Example 2:

```
Input: str1 = "aaaaaaaa", str2 = "aaaaaaaa"
Output: "aaaaaaaa"
```

Constraints:

```
1 <= str1.length, str2.length <= 1000
str1 and str2 consist of lowercase English letters.
```

Analysis:

- We have given two strings str1 and str2.
- The task is to return the shortest string that has both str1 and str2 as subsequences.
- We can solve this problem by using the brute force approach.
- We will iterate over the string and check each character.
- We will maintain the shortest string that has both str1 and str2 as subsequences.
- The time complexity of this approach is O(N).

Code:

```java
package com.hands_on;

import java.util.Scanner;

public class ShortestCommonSubsequence {
    public static String shortestCommonSubsequence(String str1, String str2) {
        int n = str1.length();
        int m = str2.length();
        int[][] dp = new int[n+1][m+1];
        for(int i=1; i<=n; i++){
            for(int j=1; j<=m; j++){
                if (str1.charAt(i-1) == str2.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]);
                }
            }
        }
        int i = n, j = m;
        StringBuilder res = new StringBuilder();
        while(i > 0 && j > 0){
            if (str1.charAt(i-1) == str2.charAt(j-1)){
                res.append(str1.charAt(i-1));
                i--;
                j--;
            }else if (dp[i-1][j] > dp[i][j-1]){
                res.append(str1.charAt(i-1));
                i--;
            }else{
                res.append(str2.charAt(j-1));
                j--;
            }
        }
        while(i > 0){
            res.append(str1.charAt(i-1));
            i--;
        }
        while(j > 0){
            res.append(str2.charAt(j-1));
            j--;
        }
        return res.reverse().toString();
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the String 1: ");
        String str1 = sc.nextLine();
        System.out.println("Enter the String 2: ");
        String str2 = sc.nextLine();
        System.out.println(shortestCommonSubsequence(str1, str2));
    }
}
```

Output:

```
Enter the String 1: abac
Enter the String 2: cab
cabac
```

28. Find the closest Palindrome

Problem Statement :

Given a string n representing an integer, return the closest integer (not including itself), which is a palindrome. If there is a tie, return the smaller one. The closest is defined as the absolute difference minimized between two integers.
Example 1:

```
Input: n = "123"
Output: "121"
```

Example 2:

```
Input: n = "1"
Output: "0"
```

> Explanation: 0 and 2 are the closest palindromes but we return the smallest which is 0.

Constraints:

```
1 <= n.length <= 18
n consists of only digits.
n does not have leading zeros.
n is representing an integer in the range [1, 10^18 - 1].
```

Analysis:

- We have given a string n representing an integer.
- The task is to return the closest integer (not including itself), which is a palindrome.
- We can solve this problem by using the brute force approach.

- We will iterate over the string and check each character.
- We will maintain the closest palindrome integer.
- The time complexity of this approach is O(N).

Code:

```
package com.hands_on;

import java.util.Scanner;

public class FindTheClosestPalindrome {
    public static String findTheClosestPalindrome(String str) {
        int n = str.length();
        long num = Long.parseLong(str);
        long left = Long.parseLong(str.substring(0, (n+1)/2));
        long res = Long.MAX_VALUE;
        long minDiff = Long.MAX_VALUE;
        for(long i=-1; i<=1; i++){
            long newNum = Long.parseLong(String.valueOf(left+i) + new StringBuilder(String.va
            if (newNum == num){
                continue;
            }
            long diff = Math.abs(newNum - num);
            if (diff < minDiff){
                minDiff = diff;
                res = newNum;
            }else if (diff == minDiff){
                res = Math.min(res, newNum);
            }
        }
        return String.valueOf(res);
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the String: ");
        String str = sc.nextLine();
        System.out.println(findTheClosestPalindrome(str));
    }
}
```

Output:

```
Enter the String: 123
121
```

---

29. Regular Expression Matching

Problem Statement:

Given an input string s and a pattern p, implement regular expression matching with support for '.' and '*' where:

- '.' Matches any single character.
- '*' Matches zero or more of the preceding element.

The matching should cover the entire input string (not partial).

Example 1:

```
Input: s = "aa", p = "a"
```

```
Output: false
```

> Explanation: "a" does not match the entire string "aa".

Example 2:

```
Input: s = "aa", p = "a*"
Output: true
```

> Explanation: '*' means zero or more of the preceding element, 'a'. Therefore, by repeating 'a' once, it becomes "aa".

Example 3:

```
Input: s = "ab", p = ".*"
Output: true
```

> Explanation: "." *means "zero or more* () of any character (.)".

Constraints:

```
1 <= s.length <= 20
1 <= p.length <= 20
s contains only lowercase English letters.
p contains only lowercase English letters, '.', and '*'.
It is guaranteed for each appearance of the character '*', there will be a previous valid cha
```

Analysis:

- We have given an input string s and a pattern p.

- The task is to implement regular expression matching with support for '.' and '*'.
- We can solve this problem by using the recursive approach.
- The time complexity of this approach is O(N).

Code:

```
package com.hands_on;

import java.util.Scanner;

public class RegularExpressionMatching {
    public static boolean regularExpressionMatching(String s, String p) {
        if (p.isEmpty()){
            return s.isEmpty();
        }
        boolean firstMatch = !s.isEmpty() && (s.charAt(0) == p.charAt(0) || p.charAt(0) == '
        if (p.length() >= 2 && p.charAt(1) == '*'){
            return (firstMatch && regularExpressionMatching(s.substring(1), p)) || regularEx
        }else{
            return firstMatch && regularExpressionMatching(s.substring(1), p.substring(1));
        }
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the String: ");
        String s = sc.nextLine();
        System.out.println("Enter the Pattern: ");
        String p = sc.nextLine();
        System.out.println(regularExpressionMatching(s, p));
    }
}
```

Output:

```
Enter the String: aa
Enter the Pattern: a*
true
```

---

30. Wildcard Matching

Problem Statement:

Given an input string (s) and a pattern (p), implement wildcard pattern matching with support for '?' and '*' where:

- '?' Matches any single character.
- '*' Matches any sequence of characters (including the empty sequence).

The matching should cover the entire input string (not partial).

Example 1:

```
Input: s = "aa", p = "a"
Output: false
```

> Explanation: "a" does not match the entire string "aa".

Example 2:

```
Input: s = "aa", p = "*"
Output: true
```

> Explanation: '*' matches any sequence.

Example 3:

```
Input: s = "cb", p = "?a"
Output: false
```

> Explanation: '?' matches 'c', but the second letter is 'a', which does not match 'b'.

Constraints:

```
0 <= s.length, p.length <= 2000
s contains only lowercase English letters.
p contains only lowercase English letters, '?' or '*'.
```

Analysis:

- We have given an input string (s) and a pattern (p).
- The task is to implement wildcard pattern matching with support for '?' and '*'.
- We can solve this problem by using the recursive approach.
- The time complexity of this approach is O(N).

Code:

```java
package com.hands_on;

import java.util.Scanner;

public class WildcardMatching {
    public static boolean wildcardMatching(String s, String p) {
        if (p.isEmpty()){
            return s.isEmpty();
        }
        boolean firstMatch = !s.isEmpty() && (s.charAt(0) == p.charAt(0) || p.charAt(0) == '
        if (p.length() >= 2 && p.charAt(1) == '*'){
            return wildcardMatching(s, p.substring(2)) || (firstMatch && wildcardMatching(s.
        }else{
            return firstMatch && wildcardMatching(s.substring(1), p.substring(1));
        }
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the String: ");
        String s = sc.nextLine();
        System.out.println("Enter the Pattern: ");
        String p = sc.nextLine();
        System.out.println(wildcardMatching(s, p));
    }
}
```

Output:

```
Enter the String: aa
Enter the Pattern: *
```

```
true
```