

String Questions



Basic Level

Reverse a String

Given a string, reverse it without using extra space.

Check if a String is a Palindrome

Check whether the given string is a palindrome or not.

Check for Anagrams

Given two strings, check if they are anagrams of each other.

Count the Number of Vowels and Consonants

Write a function to count vowels and consonants in a string.

Find Duplicate Characters in a String

Find and print all duplicate characters in the given string.

Find the First Non-Repeating Character

Return the first non-repeating character in a string.

Longest Word in a String

Given a sentence, find the longest word in it.

Check if a String is a Rotation of Another String

Given two strings, check if one is a rotation of the other.

Count the Frequency of Characters

Given a string, count the frequency of each character in the string.

Check if Two Strings are Isomorphic

Check if two strings are isomorphic (each character in one string can be mapped to a unique character in the other string).

give me explanation of problem statement with example use in simple way and also give java code optimized and easy to understand and also give time complexity of individual



1. Reverse a String

Problem Statement: Given a string, reverse it without using extra space.

- **Example:**

- Input: `"hello"`
- Output: `"olleh"`

Explanation: You have to reverse the characters in the original string itself, without using any additional data structure like another string or array.

Java Code:

```
java

public class ReverseString {
    public static String reverse(String str) {
        char[] arr = str.toCharArray();
        int left = 0, right = str.length() - 1;
        while (left < right) {
            char temp = arr[left];
            arr[left] = arr[right];
            arr[right] = temp;
            left++;
            right--;
        }
        return new String(arr);
    }

    public static void main(String[] args) {
        String str = "hello";
        System.out.println(reverse(str)); // Output: olleh
    }
}
```

Time Complexity: $O(n)$

Where `n` is the length of the string.

2. Check if a String is a Palindrome

Problem Statement: Check whether the given string is a palindrome or not (a string that reads the same backward and forward).

- **Example:**

- Input: `"madam"`
- Output: `true`
- Input: `"hello"`
- Output: `false`

Explanation: Compare the characters from the start and end of the string until the middle.

Java Code:

```
java

public class PalindromeCheck {
    public static boolean isPalindrome(String str) {
        int left = 0, right = str.length() - 1;
        while (left < right) {
            if (str.charAt(left) != str.charAt(right)) {
                return false;
            }
            left++;
            right--;
        }
        return true;
    }

    public static void main(String[] args) {
        String str = "madam";
    }
}
```

```
        System.out.println(isPalindrome(str)); // Output: true
    }
}
```

Time Complexity: $O(n)$

3. Check for Anagrams

Problem Statement: Given two strings, check if they are anagrams of each other (contain the same characters in different orders).

- **Example:**

- Input: `"listen"`, `"silent"`
- Output: `true`

Explanation: Two strings are anagrams if their sorted character arrays are identical.

Java Code:

```
java

import java.util.Arrays;

public class AnagramCheck {
    public static boolean areAnagrams(String str1, String str2) {
        if (str1.length() != str2.length()) return false;
        char[] arr1 = str1.toCharArray();
        char[] arr2 = str2.toCharArray();
        Arrays.sort(arr1);
        Arrays.sort(arr2);
        return Arrays.equals(arr1, arr2);
    }

    public static void main(String[] args) {
        String str1 = "listen";
        String str2 = "silent";
        System.out.println(areAnagrams(str1, str2)); // Output: true
    }
}
```

Time Complexity: $O(n \log n)$ (due to sorting)

4. Count the Number of Vowels and Consonants

Problem Statement: Write a function to count vowels and consonants in a string.

- **Example:**

- Input: `"hello"`
- Output: `Vowels: 2, Consonants: 3`

Explanation: Loop through the string and check each character, whether it's a vowel or a consonant.

Java Code:

```
java
```

```
public class VowelsConsonantsCount {
    public static void countVowelsAndConsonants(String str) {
        int vowels = 0, consonants = 0;
        str = str.toLowerCase();
        for (char ch : str.toCharArray()) {
            if (ch >= 'a' && ch <= 'z') {
                if ("aeiou".indexOf(ch) != -1) {
                    vowels++;
                } else {
                    consonants++;
                }
            }
        }
        System.out.println("Vowels: " + vowels + ", Consonants: " + consonants);
    }

    public static void main(String[] args) {
        String str = "hello";
        countVowelsAndConsonants(str); // Output: Vowels: 2, Consonants: 3
    }
}
```

Time Complexity: O(n)

5. Find Duplicate Characters in a String

Problem Statement: Find and print all duplicate characters in the given string.

- **Example:**

- Input: `"programming"`
- Output: `r, g`

Explanation: Use a frequency map to count occurrences of each character.

Java Code:

```
java
```

```
import java.util.HashMap;

public class DuplicateCharacters {
    public static void findDuplicates(String str) {
        HashMap<Character, Integer> charCount = new HashMap<>();
        for (char ch : str.toCharArray()) {
            charCount.put(ch, charCount.getOrDefault(ch, 0) + 1);
        }
        for (char ch : charCount.keySet()) {
            if (charCount.get(ch) > 1) {
                System.out.println(ch + " occurs " + charCount.get(ch) + " times");
            }
        }
    }

    public static void main(String[] args) {
        String str = "programming";
        findDuplicates(str); // Output: r, g
    }
}
```

Time Complexity: O(n)

6. Find the First Non-Repeating Character

Problem Statement: Return the first non-repeating character in a string.

- **Example:**
 - Input: `"swiss"`
 - Output: `w`

Explanation: First, count the occurrences of each character using a map, then traverse again to find the first character with a count of 1.

Java Code:

```
java

import java.util.LinkedHashMap;

public class FirstNonRepeatingCharacter {
    public static char firstNonRepeating(String str) {
        LinkedHashMap<Character, Integer> charCount = new LinkedHashMap<>();
        for (char ch : str.toCharArray()) {
            charCount.put(ch, charCount.getOrDefault(ch, 0) + 1);
        }
        for (char ch : charCount.keySet()) {
            if (charCount.get(ch) == 1) {
                return ch;
            }
        }
        return '-'; // return a placeholder if no non-repeating character
    }

    public static void main(String[] args) {
        String str = "swiss";
        System.out.println(firstNonRepeating(str)); // Output: w
    }
}
```

Time Complexity: $O(n)$

7. Longest Word in a String

Problem Statement: Given a sentence, find the longest word in it.

- **Example:**
 - Input: `"I love programming"`
 - Output: `"programming"`

Explanation: Split the sentence into words and find the longest one.

Java Code:

```
java

public class LongestWord {
    public static String findLongestWord(String sentence) {
        String[] words = sentence.split(" ");
        String longest = "";
        for (String word : words) {
            if (word.length() > longest.length()) {

```

```

        longest = word;
    }
    }
    return longest;
}

public static void main(String[] args) {
    String sentence = "I love programming";
    System.out.println(findLongestWord(sentence)); // Output: programming
}
}

```

Time Complexity: $O(n)$

8. Check if a String is a Rotation of Another String

Problem Statement: Given two strings, check if one is a rotation of the other.

- **Example:**
 - Input: `"abcd"`, `"dabc"``
 - Output: `true`

Explanation: Concatenate the original string with itself and check if the second string is a substring of it.

Java Code:

java

```

public class RotationCheck {
    public static boolean isRotation(String str1, String str2) {
        if (str1.length() != str2.length()) return false;
        String concatenated = str1 + str1;
        return concatenated.contains(str2);
    }

    public static void main(String[] args) {
        String str1 = "abcd";
        String str2 = "dabc";
        System.out.println(isRotation(str1, str2)); // Output: true
    }
}

```

Time Complexity: $O(n)$

9. Count the Frequency of Characters

Problem Statement: Given a string, count the frequency of each character in the string.

- **Example:**
 - Input: `"apple"``
 - Output: `a:1, p:2, l:1, e:1`

Explanation: Use a map to store the frequency of each character.

Java Code:

```
java
```

```
import java.util.HashMap;

public class CharacterFrequency {
    public static void countFrequency(String str) {
        HashMap<Character, Integer> freqMap = new HashMap<>();
        for (char ch : str.toCharArray()) {
            freqMap.put(ch, freqMap.getOrDefault(ch, 0) + 1);
        }
        for (char ch : freqMap.keySet()) {
            System.out.println(ch + ": " + freqMap.get(ch));
        }
    }

    public static void main(String[] args) {
        String str = "apple";
        countFrequency(str); // Output: a:1, p:2, l:1, e:1
    }
}
```

Time Complexity: $O(n)$

10. Check if Two Strings are Isomorphic

Problem Statement: Check if two strings are isomorphic (each character in one string can be mapped to a unique character in the other string).

- **Example:**
 - Input: `"egg"`, `"add"`
 - Output: `true`

Explanation: For each character in the first string, there should be a one-to-one mapping with the characters in the second string.

Java Code:

```
java
```

```
import java.util.HashMap;

public class IsomorphicStrings {
    public static boolean isIsomorphic(String str1, String str2) {
        if (str1.length() != str2.length()) return false;
        HashMap<Character, Character> map1 = new HashMap<>();
        HashMap<Character, Character> map2 = new HashMap<>();
        for (int i = 0; i < str1.length(); i++) {
            char c1 = str1.charAt(i);
            char c2 = str2.charAt(i);
            if (map1.containsKey(c1)) {
                if (map1.get(c1) != c2) return false;
            } else {
                if (map2.containsKey(c2)) return false;
                map1.put(c1, c2);
                map2.put(c2, c1);
            }
        }
        return true;
    }

    public static void main(String[] args) {
        String str1 = "egg";
        String str2 = "add";
    }
}
```

```
        System.out.println(isIsomorphic(str1, str2)); // Output: true
    }
}
```

Time Complexity: $O(n)$

These problems and solutions cover a range of basic string manipulation tasks, each with an efficient approach and clear explanation of time complexity.



Intermediate Level

Remove Duplicates from a String

Write a function to remove duplicate characters from a string.

Implement strstr() (Substring Search)

Implement the strstr() function which returns the first occurrence of a substring in a string.

Longest Common Prefix

Given an array of strings, find the longest common prefix.

Z-Algorithm for Pattern Matching

Implement the Z-algorithm for pattern matching and substring searching.

Count and Say Problem

Implement the "Count and Say" sequence for a given integer n.

Permutations of a String

Generate all permutations of a given string.

Longest Palindromic Substring

Find the longest substring which is a palindrome.

Check for Balanced Parentheses in an Expression

Given a string containing just the characters (,), {, }, [,], determine if the input string is valid.

Group Anagrams

Given an array of strings, group the anagrams together.

Roman to Integer Conversion

Convert a Roman numeral to an integer.



Memory updated

Here are the explanations and solutions to the intermediate-level string problems:

1. Remove Duplicates from a String

Problem Statement: Write a function to remove duplicate characters from a string.

- **Example:**
 - Input: `"programming"`
 - Output: `"progamin"`

Explanation: Use a boolean array to keep track of visited characters while iterating through the string.

Java Code:

```
java

public class RemoveDuplicates {
    public static String removeDuplicates(String str) {
        boolean[] seen = new boolean[256]; // Assuming ASCII characters
        StringBuilder result = new StringBuilder();
        for (char ch : str.toCharArray()) {
            if (!seen[ch]) {
                seen[ch] = true;
                result.append(ch);
            }
        }
        return result.toString();
    }

    public static void main(String[] args) {
        String str = "programming";
        System.out.println(removeDuplicates(str)); // Output: progamin
    }
}
```

Time Complexity: $O(n)$

2. Implement `strstr()` (Substring Search)

Problem Statement: Implement the `strstr()` function which returns the first occurrence of a substring in a string.

- **Example:**
 - Input: `str = "hello", substr = "ll"`
 - Output: `2`

Explanation: Use a sliding window to match the substring within the main string.

Java Code:

```
java

public class SubstringSearch {
    public static int strstr(String haystack, String needle) {
        int n = haystack.length(), m = needle.length();
        if (m == 0) return 0;
        for (int i = 0; i <= n - m; i++) {
            if (haystack.substring(i, i + m).equals(needle)) {
                return i;
            }
        }
        return -1;
    }

    public static void main(String[] args) {
        String str = "hello";
        String substr = "ll";
        System.out.println(strstr(str, substr)); // Output: 2
    }
}
```

```
}
}
```

Time Complexity: $O(n * m)$

3. Longest Common Prefix

Problem Statement: Given an array of strings, find the longest common prefix.

- **Example:**
 - Input: `["flower", "flow", "flight"]`
 - Output: `"fl"`

Explanation: Start with the first string and compare it with each subsequent string, shrinking the prefix as needed.

Java Code:

```
java

public class LongestCommonPrefix {
    public static String longestCommonPrefix(String[] strs) {
        if (strs.length == 0) return "";
        String prefix = strs[0];
        for (int i = 1; i < strs.length; i++) {
            while (strs[i].indexOf(prefix) != 0) {
                prefix = prefix.substring(0, prefix.length() - 1);
                if (prefix.isEmpty()) return "";
            }
        }
        return prefix;
    }

    public static void main(String[] args) {
        String[] strs = {"flower", "flow", "flight"};
        System.out.println(longestCommonPrefix(strs)); // Output: fl
    }
}
```

Time Complexity: $O(n * m)$

4. Z-Algorithm for Pattern Matching

Problem Statement: Implement the Z-algorithm for pattern matching and substring searching.

- **Example:**
 - Input: `text = "abababab", pattern = "abab"`
 - Output: `Pattern found at index 0, 2, 4`

Explanation: Construct the Z-array and identify the pattern's positions by comparing with the length of the pattern.

Java Code:

```
java
```

```
public class ZAlgorithm {
    public static void ZAlgorithm(String text, String pattern) {
        String concat = pattern + "$" + text;
        int[] Z = new int[concat.length()];
        calculateZ(Z, concat);
        for (int i = 0; i < Z.length; i++) {
            if (Z[i] == pattern.length()) {
                System.out.println("Pattern found at index " + (i - pattern.length() -
1));
            }
        }
    }

    private static void calculateZ(int[] Z, String str) {
        int L = 0, R = 0, K;
        for (int i = 1; i < str.length(); i++) {
            if (i > R) {
                L = R = i;
                while (R < str.length() && str.charAt(R) == str.charAt(R - L)) R++;
                Z[i] = R - L;
                R--;
            } else {
                K = i - L;
                if (Z[K] < R - i + 1) {
                    Z[i] = Z[K];
                } else {
                    L = i;
                    while (R < str.length() && str.charAt(R) == str.charAt(R - L)) R++;
                    Z[i] = R - L;
                    R--;
                }
            }
        }
    }

    public static void main(String[] args) {
        String text = "abababab";
        String pattern = "abab";
        ZAlgorithm(text, pattern);
    }
}
```

Time Complexity: $O(n + m)$

5. Count and Say Problem

Problem Statement: Implement the "Count and Say" sequence for a given integer `n`.

- **Example:**
 - Input: `n = 4`
 - Output: `"1211"``

Explanation: The sequence starts as "1", and each next term describes the previous term.

Java Code:

```
java
```

```
public class CountAndSay {
    public static String countAndSay(int n) {
        if (n == 1) return "1";
        String previous = countAndSay(n - 1);
```

```

Stringbuilder result = new StringBuilder();
int count = 1;
for (int i = 1; i < previous.length(); i++) {
    if (previous.charAt(i) == previous.charAt(i - 1)) {
        count++;
    } else {
        result.append(count).append(previous.charAt(i - 1));
        count = 1;
    }
}
result.append(count).append(previous.charAt(previous.length() - 1));
return result.toString();
}

public static void main(String[] args) {
    System.out.println(countAndSay(4)); // Output: 1211
}

```

Time Complexity: $O(n * m)$

Where `n` is the number of terms, and `m` is the length of each term.

6. Permutations of a String

Problem Statement: Generate all permutations of a given string.

- **Example:**

- Input: `"abc"`
- Output: `"abc", "acb", "bac", "bca", "cab", "cba"`

Explanation: Use recursion to generate all permutations.

Java Code:

java

```

public class StringPermutations {
    public static void permute(String str, String result) {
        if (str.length() == 0) {
            System.out.println(result);
            return;
        }
        for (int i = 0; i < str.length(); i++) {
            char ch = str.charAt(i);
            String ros = str.substring(0, i) + str.substring(i + 1);
            permute(ros, result + ch);
        }
    }

    public static void main(String[] args) {
        String str = "abc";
        permute(str, "");
    }
}

```

Time Complexity: $O(n!)$

7. Longest Palindromic Substring

Problem Statement: Find the longest substring which is a palindrome.

- **Example:**
 - Input: ``"babad"```
 - Output: ``"bab"```

Explanation: Use dynamic programming or expand around the center approach.

Java Code:

```
java

public class LongestPalindromicSubstring {
    public static String longestPalindrome(String s) {
        if (s == null || s.length() < 1) return "";
        int start = 0, end = 0;
        for (int i = 0; i < s.length(); i++) {
            int len1 = expandAroundCenter(s, i, i);
            int len2 = expandAroundCenter(s, i, i + 1);
            int len = Math.max(len1, len2);
            if (len > end - start) {
                start = i - (len - 1) / 2;
                end = i + len / 2;
            }
        }
        return s.substring(start, end + 1);
    }

    private static int expandAroundCenter(String s, int left, int right) {
        while (left >= 0 && right < s.length() && s.charAt(left) == s.charAt(right)) {
            left--;
            right++;
        }
        return right - left - 1;
    }

    public static void main(String[] args) {
        String str = "babad";
        System.out.println(longestPalindrome(str)); // Output: bab
    }
}
```

Time Complexity: $O(n^2)$

8. Check for Balanced Parentheses in an Expression

Problem Statement: Given a string containing ``(``,`)``,`{`,`}`,`,`[`,`]`,``, determine if the input string is valid.

- **Example:**
 - Input: ``"({[]})"```
 - Output: ``true``

Explanation: Use a stack to validate the sequence.

Java Code:

```
java

import java.util.Stack;

public class BalancedParentheses {
    public static boolean isValid(String s) {
        Stack<Character> stack = new Stack<>();
        for (char c : s.toCharArray()) {
            if (c == '(' || c == '{' || c == '[') {
                stack.push(c);
            }
        }
        return stack.isEmpty();
    }
}
```

```

        } else {
            if (stack.isEmpty()) return false;
            char top = stack.pop();
            if (c == ')' && top != '(' || c == '}' && top != '{' || c == ']' && top !=
'[') {
                return false;
            }
        }
    }
    return stack.isEmpty();
}

public static void main(String[] args) {
    String str = "({[]})";
    System.out.println(isValid(str)); // Output: true
}

```

Time Complexity: $O(n)$

9. Group Anagrams

Problem Statement: Given an array of strings, group the anagrams together.

- **Example:**

- Input: `["eat", "tea", "tan", "ate", "nat", "bat"]`
- Output: `[["eat", "tea", "ate"], ["tan", "nat"], ["bat"]]`

Explanation: Use a HashMap to group anagrams.

Java Code:

```

java

import java.util.*;

public class GroupAnagrams {
    public static List<List<String>> groupAnagrams(String[] strs) {
        if (strs == null || strs.length == 0) return new ArrayList<>();
        Map<String, List<String>> map = new HashMap<>();
        for (String s : strs) {
            char[] chars = s.toCharArray();
            Arrays.sort(chars);
            String sorted = new String(chars);
            if (!map.containsKey(sorted)) {
                map.put(sorted, new ArrayList<>());
            }
            map.get(sorted).add(s);
        }
        return new ArrayList<>(map.values());
    }

    public static void main(String[] args) {
        String[] strs = {"eat", "tea", "tan", "ate", "nat", "bat"};
        System.out.println(groupAnagrams(strs));
    }
}

```

Time Complexity: $O(n * m * \log(m))$

10. Roman to Integer Conversion

Problem Statement: Convert a Roman numeral to an integer.

- **Example:**

- Input: `"IX"`
- Output: `9`

Explanation: Map Roman symbols to integers and handle subtraction cases.

Java Code:

java

```
import java.util.HashMap;

public class RomanToInteger {
    public static int romanToInt(String s) {
        HashMap<Character, Integer> map = new HashMap<>();
        map.put('I', 1);
        map.put('V', 5);
        map.put('X', 10);
        map.put('L', 50);
        map.put('C', 100);
        map.put('D', 500);
        map.put('M', 1000);
        int sum = 0;
        for (int i = 0; i < s.length(); i++) {
            if (i > 0 && map.get(s.charAt(i)) > map.get(s.charAt(i - 1))) {
                sum += map.get(s.charAt(i)) - 2 * map.get(s.charAt(i - 1));
            } else {
                sum += map.get(s.charAt(i));
            }
        }
        return sum;
    }

    public static void main(String[] args) {
        String roman = "IX";
        System.out.println(romanToInt(roman)); // Output: 9
    }
}
```

Time Complexity: O(n)

Let me know if you need further explanations or refinements on any of these solutions!



Advanced Level

Longest Repeating Subsequence

Given a string, find the longest subsequence that repeats itself.

Edit Distance (Levenshtein Distance)

Find the minimum number of edits required to convert one string into another.

KMP (Knuth-Morris-Pratt) Pattern Matching Algorithm

Implement the KMP algorithm for pattern matching.

Minimum Window Substring

Given two strings s and t, find the minimum window in s which contains all the characters of t.

Wildcard Pattern Matching

Implement a function to match a wildcard pattern with a string.

Aho-Corasick Algorithm for Multiple Pattern Matching

Implement the Aho-Corasick algorithm for searching multiple patterns in a string.

Manacher's Algorithm

Find the longest palindromic substring using Manacher's algorithm.

Suffix Array Construction

Construct a suffix array for a given string.

Suffix Tree for Pattern Searching

Implement a suffix tree to search for a pattern in a string efficiently.

Rabin-Karp Algorithm for Pattern Matching

Implement the Rabin-Karp algorithm for pattern matching using a rolling hash.



Memory updated

Here are explanations and solutions for the advanced-level string manipulation problems:

1. Longest Repeating Subsequence

Problem Statement: Given a string, find the longest subsequence that repeats itself.

- **Example:**

- Input: `"aabb"`
- Output: `2` (The longest repeating subsequence is `"ab"`)

Explanation: This is similar to the Longest Common Subsequence (LCS) problem, but the two subsequences must come from the same string with different positions.

Java Code:

```
java
```

```
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) {
        String str = "aabb";
    }
}
```



```

        System.out.println(longestRepeatingSubsequence(str)); // Output: 2
    }
}

```

Time Complexity: $O(n^2)$

2. Edit Distance (Levenshtein Distance)

Problem Statement: Find the minimum number of edits required to convert one string into another.

- **Example:**
 - Input: `str1 = "kitten"`, `str2 = "sitting"``
 - Output: `3`

Explanation: Use dynamic programming to calculate the minimum number of insertions, deletions, or substitutions required to transform one string into another.

Java Code:

```

java

public class EditDistance {
    public static int editDistance(String str1, String str2) {
        int m = str1.length(), n = str2.length();
        int[][] dp = new int[m+1][n+1];

        for (int i = 0; i <= m; i++) {
            for (int j = 0; j <= n; j++) {
                if (i == 0) {
                    dp[i][j] = j; // Insert all characters
                } else if (j == 0) {
                    dp[i][j] = i; // Remove all characters
                } else if (str1.charAt(i-1) == str2.charAt(j-1)) {
                    dp[i][j] = 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[m][n];
    }

    public static void main(String[] args) {
        String str1 = "kitten";
        String str2 = "sitting";
        System.out.println(editDistance(str1, str2)); // Output: 3
    }
}

```

Time Complexity: $O(m * n)$

3. KMP (Knuth-Morris-Pratt) Pattern Matching Algorithm

Problem Statement: Implement the KMP algorithm for pattern matching.

- **Example:**
 - Input: `text = "abcxabcdabcbcdabcy"`, `pattern = "abcdabcy"``
 - Output: `Pattern found at index 8`

Explanation: Preprocess the pattern to create the "longest prefix which is also suffix" (LPS) array, then use it to search the pattern efficiently.

Java Code:

```
java
```

```
public class KMP {
    public static void KMPSearch(String pat, String txt) {
        int M = pat.length();
        int N = txt.length();
        int[] lps = new int[M];
        int j = 0; // index for pat[]

        computeLPSArray(pat, M, lps);
        int i = 0; // index for txt[]
        while (i < N) {
            if (pat.charAt(j) == txt.charAt(i)) {
                j++;
                i++;
            }
            if (j == M) {
                System.out.println("Pattern found at index " + (i - j));
                j = lps[j - 1];
            } else if (i < N && pat.charAt(j) != txt.charAt(i)) {
                if (j != 0) {
                    j = lps[j - 1];
                } else {
                    i++;
                }
            }
        }
    }

    private static void computeLPSArray(String pat, int M, int[] lps) {
        int len = 0;
        lps[0] = 0; // lps[0] is always 0
        int i = 1;
        while (i < M) {
            if (pat.charAt(i) == pat.charAt(len)) {
                len++;
                lps[i] = len;
                i++;
            } else {
                if (len != 0) {
                    len = lps[len - 1];
                } else {
                    lps[i] = 0;
                    i++;
                }
            }
        }
    }

    public static void main(String[] args) {
        String txt = "abcxabcabcdabcy";
        String pat = "abcdabcy";
        KMPSearch(pat, txt);
    }
}
```

Time Complexity: $O(n + m)$

4. Minimum Window Substring

Problem Statement: Given two strings `s` and `t`, find the minimum window in `s` which contains all the characters of `t`.

- **Example:**

- Input: `s = "ADOBECODEBANC"`, `t = "ABC"`
- Output: `"BANC"`

Explanation: Use two pointers and a sliding window technique to find the smallest window containing all characters of `t`.

Java Code:

```
java
```

```
import java.util.HashMap;

public class MinimumWindowSubstring {
    public static String minWindow(String s, String t) {
        if (s.length() == 0 || t.length() == 0) return "";

        HashMap<Character, Integer> mapT = new HashMap<>();
        for (char c : t.toCharArray()) {
            mapT.put(c, mapT.getOrDefault(c, 0) + 1);
        }

        int required = mapT.size();
        int left = 0, right = 0, formed = 0;
        HashMap<Character, Integer> windowCounts = new HashMap<>();
        int[] ans = {-1, 0, 0}; // length, left, right

        while (right < s.length()) {
            char c = s.charAt(right);
            windowCounts.put(c, windowCounts.getOrDefault(c, 0) + 1);

            if (mapT.containsKey(c) && windowCounts.get(c).intValue() ==
mapT.get(c).intValue()) {
                formed++;
            }

            while (left <= right && formed == required) {
                c = s.charAt(left);

                if (ans[0] == -1 || right - left + 1 < ans[0]) {
                    ans[0] = right - left + 1;
                    ans[1] = left;
                    ans[2] = right;
                }

                windowCounts.put(c, windowCounts.get(c) - 1);
                if (mapT.containsKey(c) && windowCounts.get(c).intValue() <
mapT.get(c).intValue()) {
                    formed--;
                }
                left++;
            }
            right++;
        }

        return ans[0] == -1 ? "" : s.substring(ans[1], ans[2] + 1);
    }

    public static void main(String[] args) {
        String s = "ADOBECODEBANC";
        String t = "ABC";
        System.out.println(minWindow(s, t)); // Output: BANC
    }
}
```

Time Complexity: O(n)

These examples should help you understand the problems and approaches to solving them using optimized Java code. Let me know if you'd like further assistance with the other problems!



give remaining



Here are the solutions for the remaining advanced-level string problems:

5. Wildcard Pattern Matching

Problem Statement: Implement a function to match a wildcard pattern with a string. The wildcard characters are `'*'` (matches zero or more characters) and `'?'` (matches exactly one character).

- **Example:**

- Input: `s = "adceb"`, `p = "*a*b"``
- Output: `true`

Explanation: Use dynamic programming to match the pattern with the string by handling wildcard characters.

Java Code:

java

```
public class WildcardPatternMatching {
    public static boolean isMatch(String s, String p) {
        int m = s.length();
        int n = p.length();
        boolean[][] dp = new boolean[m + 1][n + 1];
        dp[0][0] = true;

        for (int j = 1; j <= n; j++) {
            if (p.charAt(j - 1) == '*') {
                dp[0][j] = dp[0][j - 1];
            }
        }

        for (int i = 1; i <= m; i++) {
            for (int j = 1; j <= n; j++) {
                if (p.charAt(j - 1) == '*') {
                    dp[i][j] = dp[i][j - 1] || dp[i - 1][j];
                } else if (p.charAt(j - 1) == '?' || s.charAt(i - 1) == p.charAt(j - 1)) {
                    dp[i][j] = dp[i - 1][j - 1];
                }
            }
        }
        return dp[m][n];
    }

    public static void main(String[] args) {
        String s = "adceb";
        String p = "*a*b";
        System.out.println(isMatch(s, p)); // Output: true
    }
}
```

Time Complexity: $O(m * n)$

6. Aho-Corasick Algorithm for Multiple Pattern Matching

Problem Statement: Implement the Aho-Corasick algorithm for searching multiple patterns in a string.

- **Example:**

- Input: `text = "ababcababcabc"`, `patterns = ["abc", "bca"]`
- Output: `Pattern 'abc' found at indices: 2, 8, 12; Pattern 'bca' found at indices: 1, 7`

Explanation: Build a Trie for the patterns, create failure links, and search through the text using the Trie.

Java Code:

java

```
import java.util.*;

public class AhoCorasick {
    static class TrieNode {
        Map<Character, TrieNode> children = new HashMap<>();
        TrieNode fail;
        List<Integer> output = new ArrayList<>();
    }

    private static void buildTrie(TrieNode root, String[] patterns) {
        for (int i = 0; i < patterns.length; i++) {
            String pattern = patterns[i];
            TrieNode node = root;
            for (char ch : pattern.toCharArray()) {
                node = node.children.computeIfAbsent(ch, c -> new TrieNode());
            }
            node.output.add(i);
        }
    }

    private static void buildFailureLinks(TrieNode root) {
        Queue<TrieNode> queue = new LinkedList<>();
        root.fail = null;
        for (TrieNode child : root.children.values()) {
            child.fail = root;
            queue.add(child);
        }

        while (!queue.isEmpty()) {
            TrieNode current = queue.poll();
            for (Map.Entry<Character, TrieNode> entry : current.children.entrySet()) {
                char ch = entry.getKey();
                TrieNode child = entry.getValue();
                TrieNode fail = current.fail;

                while (fail != null && !fail.children.containsKey(ch)) {
                    fail = fail.fail;
                }
                child.fail = fail == null ? root : fail.children.get(ch);
                if (child.fail != null) {
                    child.output.addAll(child.fail.output);
                }
                queue.add(child);
            }
        }
    }

    public static void search(String text, String[] patterns) {
        TrieNode root = new TrieNode();
        buildTrie(root, patterns);
        buildFailureLinks(root);

        for (int i = 0; i < text.length(); i++) {
            TrieNode node = root;
            for (int j = i; j < text.length(); j++) {
                char ch = text.charAt(j);
                node = node.children.get(ch);
                if (node == null) break;
                for (int k : node.output) {
                    System.out.println("Pattern '" + patterns[k] + "' found at indices: " + i);
                }
            }
        }
    }
}
```

```

buildTrie(root, patterns);
buildFailureLinks(root);

TrieNode node = root;
for (int i = 0; i < text.length(); i++) {
    char ch = text.charAt(i);
    while (node != null && !node.children.containsKey(ch)) {
        node = node.fail;
    }
    if (node == null) {
        node = root;
        continue;
    }
    node = node.children.get(ch);
    for (int patternIndex : node.output) {
        System.out.println("Pattern '" + patterns[patternIndex] + "' found at
index " + (i - patterns[patternIndex].length() + 1));
    }
}

public static void main(String[] args) {
    String text = "ababcababcabc";
    String[] patterns = {"abc", "bca"};
    search(text, patterns);
}
}

```

Time Complexity: $O(n + m)$ where n is the length of the text and m is the total length of all patterns.

7. Manacher's Algorithm

Problem Statement: Find the longest palindromic substring using Manacher's algorithm.

- **Example:**
 - Input: `"babad"`
 - Output: `"bab"`

Explanation: Manacher's algorithm finds the longest palindromic substring in linear time using a transformed string.

Java Code:

java

```

public class Manacher {
    public static String longestPalindrome(String s) {
        StringBuilder sb = new StringBuilder("#");
        for (char c : s.toCharArray()) {
            sb.append(c).append('#');
        }
        String T = sb.toString();
        int n = T.length();
        int[] L = new int[n];
        int C = 0, R = 0; // center and right edge of the current rightmost palindrome

        for (int i = 0; i < n; i++) {
            int mirror = 2 * C - i;
            if (R > i) {
                L[i] = Math.min(R - i, L[mirror]);
            } else {
                L[i] = 0;
            }
            while (i + L[i] + 1 < n && i - L[i] - 1 >= 0 && T.charAt(i + L[i] + 1) ==

```

```

T.charAt(i - L[i] - 1)) {
    L[i]++;
}
if (i + L[i] > R) {
    C = i;
    R = i + L[i];
}
}

int maxlen = 0, center = 0;
for (int i = 0; i < n; i++) {
    if (L[i] > maxlen) {
        maxlen = L[i];
        center = i;
    }
}

return s.substring((center - maxlen) / 2, (center + maxlen) / 2);
}

public static void main(String[] args) {
    String s = "babad";
    System.out.println(longestPalindrome(s)); // Output: bab
}
}

```

Time Complexity: $O(n)$

8. Suffix Array Construction

Problem Statement: Construct a suffix array for a given string.

- **Example:**

- Input: `"banana"`
- Output: `[5, 3, 1, 0, 4, 2]`

Explanation: A suffix array is an array of the starting indices of all suffixes of a string sorted in lexicographical order.

Java Code:

java

```

import java.util.Arrays;

public class SuffixArray {
    public static int[] buildSuffixArray(String s) {
        int n = s.length();
        Integer[] suffixes = new Integer[n];
        for (int i = 0; i < n; i++) {
            suffixes[i] = i;
        }
        Arrays.sort(suffixes, (a, b) -> s.substring(a).compareTo(s.substring(b)));
        return Arrays.stream(suffixes).mapToInt(Integer::intValue).toArray();
    }

    public static void main(String[] args) {
        String s = "banana";
        int[] suffixArray = buildSuffixArray(s);
        System.out.println(Arrays.toString(suffixArray)); // Output: [5, 3, 1, 0, 4, 2]
    }
}

```

Time Complexity: $O(n * \log(n) * \log(n))$

9. Suffix Tree for Pattern Searching

Problem Statement: Implement a suffix tree to search for a pattern in a string efficiently.

- **Example:**

- Input: `text = "bananas"`, `pattern = "ana"`
- Output: `Pattern found at indices: 1, 3`

Explanation: Construct a suffix tree and use it to find the occurrences of a pattern.

Java Code:

```
java

import java.util.*;

public class SuffixTree {
    static class Node {
        Map<Character, Node> children = new HashMap<>();
        int start, end;
        int suffixIndex = -1;

        Node(int start, int end) {
            this.start = start;
            this.end = end;
        }
    }

    public static void buildSuffixTree(String s) {
        Node root = new Node(-1, -1);
        for (int i = 0; i < s.length(); i++) {
            insertSuffix(root, s, i);
        }
        printSuffixTree(root, s);
    }

    private static void insertSuffix(Node root, String s, int suffixStart) {
        Node node = root;
        for (int i = suffixStart; i < s.length(); i++) {
            char c = s.charAt(i);
            if (!node.children.containsKey(c)) {
                node.children.put(c, new Node(i, s.length()));
            }
            node = node.children.get(c);
        }
        node.suffixIndex = suffixStart;
    }

    private static void printSuffixTree(Node node, String s) {
        if (node.suffixIndex != -1) {
            System.out.println("Suffix at index " + node.suffixIndex + ": " +
s.substring(node.suffixIndex));
        }
        for (Node child : node.children.values()) {
            printSuffixTree(child, s);
        }
    }

    public static void main(String[] args) {
        String s = "bananas";
        buildSuffixTree(s);
    }
}
```


Time Complexity: $O(n^2)$

10. Rabin-Karp Algorithm for Pattern Matching

Problem Statement: Implement the Rabin-Karp algorithm for pattern matching using a rolling hash.

- **Example:**
 - Input: `text = "abcdefg"`, `pattern = "bcd"``
 - Output: `Pattern found at index 1`

Explanation: Use a rolling hash to efficiently find the pattern in the text.

Java Code:

java

```
public class RabinKarp {
    private static final int BASE = 256;
    private static final int MOD = 101;

    public static void search(String text, String pattern) {
        int m = pattern.length();
        int n = text.length();
        int patternHash = 0, textHash = 0, h = 1;

        for (int i = 0; i < m - 1; i++) {
            h = (h * BASE) % MOD;
        }

        for (int i = 0; i < m; i++) {
            patternHash = (BASE * patternHash + pattern.charAt(i)) % MOD;
            textHash = (BASE * textHash + text.charAt(i)) % MOD;
        }

        for (int i = 0; i <= n - m; i++) {
            if (patternHash == textHash) {
                boolean match = true;
                for (int j = 0; j < m; j++) {
                    if (text.charAt(i + j) != pattern.charAt(j)) {
                        match = false;
                        break;
                    }
                }
                if (match) {
                    System.out.println("Pattern found at index " + i);
                }
            }
            if (i < n - m) {
                textHash = (BASE * (textHash - text.charAt(i) * h) + text.charAt(i + m)) %
MOD;

                if (textHash < 0) {
                    textHash += MOD;
                }
            }
        }
    }

    public static void main(String[] args) {
        String text = "abcdefg";
        String pattern = "bcd";
        search(text, pattern);
    }
}
```

Time Complexity: $O(n)$

These implementations cover various advanced string manipulation problems. If you need any more details or have additional questions, feel free to ask!

ChatGPT can make mistakes. Check important info.