```java
Codeium: Refactor | Explain
public class LongestCommonSubsequence {
    //Given two sequences, find the length of longest subsequence present in both of them.
    //A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous.
    //For example, “abc”, “abg”, “bdf”, “aeg”, ‘”acefg”, .. etc are subsequences of “abcdefg”.
    //So a string of length n has 2^n different possible subsequences.
    //Example: s1 = "abcdgh"; s2 = "abedfhr" -> "abdh" -> 4
    //Approach: Recursive


    Codeium: Refactor | Explain | Generate Javadoc | ×
    public static void main(String[] args) {
        String s1 = "abcdgh";
        String s2 = "abedfhr";
        System.out.println(lcs(s1, s2, s1.length(), s2.length()));
    }

    Codeium: Refactor | Explain | Generate Javadoc | ×
    public static int lcs(String s1, String s2, int m, int n) {
        if (m == 0 || n == 0) {
            return 0;
        }

        if (s1.charAt(m - 1) == s2.charAt(n - 1)) {
            return 1 + lcs(s1, s2, m - 1, n - 1);
        } else {
            return Math.max(lcs(s1, s2, m - 1, n), lcs(s1, s2, m, n - 1));
        }
    }

    //Approach: Memoization
    Codeium: Refactor | Explain | ×
    public static int lcsMemo(String s1, String s2, int m, int n) {
        int[][] dp = new int[m + 1][n + 1];
        return lcsMemo(s1, s2, m, n, dp);
    }

    Codeium: Refactor | Explain | ×
    public static int lcsMemo(String s1, String s2, int m, int n) {
        int[][] dp = new int[m + 1][n + 1];
        return lcsMemo(s1, s2, m, n, dp);
    }

    Codeium: Refactor | Explain | Generate Javadoc | ×
    public static int lcsMemo(String s1, String s2, int m, int n, int[][] dp) {
        if (m == 0 || n == 0) {
            return 0;
        }
        if (dp[m][n] != 0) {
            return dp[m][n];
        }

        if (s1.charAt(m - 1) == s2.charAt(n - 1)) {
            dp[m][n] = 1 + lcsMemo(s1, s2, m - 1, n - 1, dp);
            return dp[m][n];
        } else {
            dp[m][n] = Math.max(lcsMemo(s1, s2, m - 1, n, dp), lcsMemo(s1, s2, m, n - 1, dp));
            return dp[m][n];
        }
    }
    //Approach: Dynamic Programming
```

```java
//Approach: Dynamic Programming
    Codeium: Refactor | Explain | ×
    public static int lcsDP(String s1, String s2) {
        int m = s1.length();
        int n = s2.length();

        int[][] dp = new int[m + 1][n + 1];

        //Fill the dp table in bottom up manner
        for (int i = 1; i <= m ; i++) {
            for (int j = 1; j <= n ; j++) {
                if (s1.charAt(i - 1) == s2.charAt(j - 1)) {
                    //If the characters match, then add 1 to the result and move diagonally
                    dp[i][j] = 1 + dp[i - 1][j - 1];
                } else {
                    //If the characters don't match, then take the max of the two results
                    dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
                }

            }
        }

        return dp[m][n];
    }


```

```
.LongestCommonSubsequence
4
```

```java
package a1Dynamic.LCS;

Codeium: Refactor | Explain
public class LongestCommonSubsequenceDP {
    //Given two sequences, find the length of longest subsequence present in both of them.
    //A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous.
    //Approach: Dynamic Programming

    Codeium: Refactor | Explain | Generate Javadoc | ×
    public static void main(String[] args) {
        String s1 = "abcdgh";
        String s2 = "abedfhr";
        System.out.println(lcs(s1, s2));
    }

    Codeium: Refactor | Explain | Generate Javadoc | ×
    public static int lcs(String s1, String s2) {
        int m = s1.length();
        int n = s2.length();

        int[][] dp = new int[m + 1][n + 1];

        //Fill the dp table in bottom up manner
        for (int i = 1; i <= m ; i++) {
            for (int j = 1; j <= n ; j++) {
                if (s1.charAt(i - 1) == s2.charAt(j - 1)) {
                    //If the characters match, then add 1 to the result and move diagonally
                    dp[i][j] = 1 + dp[i - 1][j - 1];
                } else {
                    //If the characters don't match, then take the max of the two results
                    dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
                }

            }
        }

        return dp[m][n];
    }
}

```

```
.LongestCommonSubsequenceDP
4
```

```java
Codeium: Refactor | Explain
public class LongestCommonString01 {
    //  Given two strings ‘X’ and ‘Y’, find the length of the longest common substring.
    //  Input : X = “abcdxyz”, y = “xyzabcd”
    //  Output : 4
    //  The longest common substring is “abcd” and is of length 4.

    //  Input : X = “zxabcdezy”, y = “yzabcdezx”    //  Output : 6

    Codeium: Refactor | Explain | Generate Javadoc | ×
    public static void main(String[] args) {
        String s1 = "abcdxyz";
        String s2 = "xyzabcd";
        System.out.println(lcs(s1, s2));
    }

    Codeium: Refactor | Explain | Generate Javadoc | ×
    public static int lcs(String s1, String s2) {
        int m = s1.length();
        int n = s2.length();

        int[][] dp = new int[m + 1][n + 1];

        //Fill the dp table in bottom up manner
        int max = 0;
        for (int i = 1; i <= m ; i++) {
            for (int j = 1; j <= n ; j++) {
                if (s1.charAt(i - 1) == s2.charAt(j - 1)) {
                    //If the characters match, then add 1 to the result and move diagonally
                    dp[i][j] = 1 + dp[i - 1][j - 1];
                    max = Math.max(max, dp[i][j]);
                } else {
                    //If the characters don't match, then take the max of the two results
                    dp[i][j] = 0;
                }

            }
        }

        return max;
    }
}

```

```
.LongestCommonString01
4
```

```java
Codeium: Refactor | Explain
public class PrintLongestCommonSubsequence02 {
//Given two sequences, print the longest subsequence present in both of them.
    //A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous.
    //Approach: Dynamic Programming

    Codeium: Refactor | Explain | Generate Javadoc | ×
    public static void main(String[] args) {
        String s1 = "abcdgh";
        String s2 = "abedfhr";
        System.out.println(lcs(s1, s2));
    }

    Codeium: Refactor | Explain | Generate Javadoc | ×
    public static String lcs(String s1, String s2) {
        int m = s1.length();
        int n = s2.length();

        int[][] dp = new int[m + 1][n + 1];

        //Fill the dp table in bottom up manner
        for (int i = 1; i <= m ; i++) {
            for (int j = 1; j <= n ; j++) {
                if (s1.charAt(i - 1) == s2.charAt(j - 1)) {
                    //If the characters match, then add 1 to the result and move diagonally
                    dp[i][j] = 1 + dp[i - 1][j - 1];
                } else {
                    //If the characters don't match, then take the max of the two results
                    dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
                }

            }
        }

        //Now, we need to print the longest common subsequence
        //Start from the bottom right corner of the dp table
        int i = m;
        int j = n;
        StringBuilder sb = new StringBuilder();
        while (i > 0 && j > 0) {
            //If the characters match, then add the character to the result and move diagonally
            if (s1.charAt(i - 1) == s2.charAt(j - 1)) {
                sb.append(s1.charAt(i - 1));
                i--;
                j--;
            } else {
                //If the characters don't match, then move in the direction of the larger result
                if (dp[i - 1][j] > dp[i][j - 1]) {
                    i--;
                } else {
                    j--;
                }
            }
        }

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

```
.PrintLongestCommonSubsequence02
abdh
```

```java
Codeium: Refactor | Explain
public class ShortestCommonSupersequence03 {
    //Given two strings str1 and str2, find the shortest string that has both str1 and str2 as subsequences.
    // Examples : Input:   str1 = "geek",  str2 = "eke"  Output: "geeke"
    //Approach: Dynamic Programming

    Codeium: Refactor | Explain | Generate Javadoc | ×
    public static void main(String[] args) {
//        String s1 = "abcdgh";
//        String s2 = "abedfhr";
        String s1 = "geek";
        String s2 = "eke";
     //   System.out.println(shortestCommonSupersequence(s1, s2));
        System.out.println(printShortestCommonSupersequence(s1, s2));
    }

    Codeium: Refactor | Explain | Generate Javadoc | ×
    public static int shortestCommonSupersequence(String s1, String s2) {
        int m = s1.length();
        int n = s2.length();

        int[][] dp = new int[m + 1][n + 1];

        //Fill the dp table in bottom up manner
        for (int i = 1; i <= m ; i++) {
            for (int j = 1; j <= n ; j++) {
                if (s1.charAt(i - 1) == s2.charAt(j - 1)) {
                    //If the characters match, then add 1 to the result and move diagonally
                    dp[i][j] = 1 + dp[i - 1][j - 1];
                } else {
                    //If the characters don't match, then take the max of the two results
                    dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
                }

            }
        }

        return m + n - dp[m][n];
    }

    //print the shortest common supersequence
    Codeium: Refactor | Explain | ×
    public static String printShortestCommonSupersequence(String s1, String s2) {
        int m = s1.length();
        int n = s2.length();

        int[][] dp = new int[m + 1][n + 1];

        //Fill the dp table in bottom up manner
        for (int i = 1; i <= m ; i++) {
            for (int j = 1; j <= n ; j++) {
                if (s1.charAt(i - 1) == s2.charAt(j - 1)) {
                    //If the characters match, then add 1 to the result and move diagonally
                    dp[i][j] = 1 + dp[i - 1][j - 1];
                } else {
                    //If the characters don't match, then take the max of the two results
                    dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
                }

            }
        }

        //print the shortest common supersequence
        int i = m;
        int j = n;
        StringBuilder sb = new StringBuilder();
        while (i > 0 && j > 0) {
            if (s1.charAt(i - 1) == s2.charAt(j - 1)) { //if the characters match, then add the character to the result and move diagonally
                sb.append(s1.charAt(i - 1));
                i--;
                j--;
            } else {
                if (dp[i - 1][j] > dp[i][j - 1]) { //if the character doesn't match, then move in the direction of the max result
                    sb.append(s1.charAt(i - 1));
                    i--;
                } else {
                    sb.append(s2.charAt(j - 1));
                    j--;
                }
            }
        }
```

```
        }

        while (i > 0) { //if there are any characters left in s1, then add them to the result
            sb.append(s1.charAt(i - 1));
            i--;
        }

        while (j > 0) { //if there are any characters left in s2, then add them to the result
            sb.append(s2.charAt(j - 1));
            j--;
        }

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

    //print the shortest common supersequence

}

```

```
.ShortestCommonSupersequence03
geeke
```

```java
Codeium: Refactor | Explain
public class MinimumInsertionDeletionToConvertStrAToStrB04 {
    //Given two strings ‘str1’ and ‘str2’ of size m and n respectively.
    // The task is to remove/delete and insert the minimum number of characters from/in str1 to transform it into str2.
    // It could be possible that the same character needs to be removed/deleted from one point of str1 and inserted to some another point.
    // Example 1: Input : str1 = "heap", str2 = "pea"
    // Output : Minimum Deletion = 2 and
    //          Minimum Insertion = 1
    // p and h deleted from heap
    // Then, p is inserted at the beginning
    // One thing to note, though p was required yet
    // it was removed/deleted first from its position and
    // Example 2: Input : str1 = "geeksforgeeks", str2 = "geeks"\ Output : Minimum Deletion = 8
    //          Minimum Insertion = 0
    // Explanation:  String s1 = "geeksforgeeks" and s2 = "geeks" 1. Delete 7 characters from s1 so that string becomes "geeks"
    //Approach: Dynamic Programming
    Codeium: Refactor | Explain | ×
    public static void main(String[] args) {
        String s1 = "heap";
        String s2 = "pea";
        System.out.println(minimumInsertionDeletionToConvertStrAToStrB(s1, s2));
    }

    Codeium: Refactor | Explain | Generate Javadoc | ×
    public static int minimumInsertionDeletionToConvertStrAToStrB(String s1, String s2) {
        int m = s1.length();
        int n = s2.length();

        int[][] dp = new int[m + 1][n + 1];

        //Fill the dp table in bottom up manner
        for (int i = 1; i <= m ; i++) {
            for (int j = 1; j <= n ; j++) {
                if (s1.charAt(i - 1) == s2.charAt(j - 1)) {
                    //If the characters match, then add 1 to the result and move diagonally
                    dp[i][j] = 1 + dp[i - 1][j - 1];
                } else {
                    //If the characters don't match, then take the max of the two results
                    dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
                }

            }
        }

        return m + n - 2 * dp[m][n];
    }

}

```

```
.MinimumInsertionDeletionToConvertStrAToStrB04
3
```

```java
Codeium: Refactor | Explain
public class LongestPallindromicSubsequence05 {
    //Given a sequence, find the length of the longest palindromic subsequence in it.
    //As another example, if the given sequence is “BBABCBCAB”, then the output should be 7 as
    // “BABCBAB” is the longest palindromic subsequence in it.
    //“BBBBB” and “BBCBB” are also palindromic subsequences of the given sequence, but not the longest ones.
    //Approach: Dynamic Programming

    Codeium: Refactor | Explain | Generate Javadoc | ×
    public static void main(String[] args) {
        String s1 = "BBABCBCAB";
        System.out.println(lcs(s1));
        printLCS(s1);
    }

    Codeium: Refactor | Explain | Generate Javadoc | ×
    public static int lcs(String s1) {
        int m = s1.length();
        int n = m;

        int[][] dp = new int[m + 1][n + 1];

        //Fill the dp table in bottom up manner
        for (int i = 1; i <= m ; i++) {
            for (int j = 1; j <= n ; j++) {
                if (s1.charAt(i - 1) == s1.charAt(n - j)) {
                    //If the characters match, then add 1 to the result and move diagonally
                    dp[i][j] = 1 + dp[i - 1][j - 1];
                } else {
                    //If the characters don't match, then take the max of the two results
                    dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
                }

            }
        }

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

```java
    //Print the longest pallindromic subsequence
    //Approach: Dynamic Programming

    Codeium: Refactor | Explain | Generate Javadoc | ×
    public static void printLCS(String s1) {
        int m = s1.length();
        int n = m;

        int[][] dp = new int[m + 1][n + 1];

        //Fill the dp table in bottom up manner
        for (int i = 1; i <= m ; i++) {
            for (int j = 1; j <= n ; j++) {
                if (s1.charAt(i - 1) == s1.charAt(n - j)) {
                    //If the characters match, then add 1 to the result and move diagonally
                    dp[i][j] = 1 + dp[i - 1][j - 1];
                } else {
                    //If the characters don't match, then take the max of the two results
                    dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
                }

            }
        }
```

```java

        //Print the longest pallindromic subsequence
        int i = m;
        int j = n;
        String result = "";
        while (i > 0 && j > 0) {
            if (s1.charAt(i - 1) == s1.charAt(n - j)) {
                result = s1.charAt(i - 1) + result;
                i--;
                j--;
            } else {
                if (dp[i - 1][j] > dp[i][j - 1]) {
                    i--;
                } else {
                    j--;
                }
            }
        }

        System.out.println(result);
    }

}
```

```
 .LongestPallindromicSubsequence05
7
BACBCAB
```

```java
Codeium: Refactor | Explain
public class MinInsDelToMakePallindrome06 {
    //Given a string, find the minimum number of characters to be inserted to convert it to palindrome.
    //For Example:
    //ab: Number of insertions required is 1. bab or aba
    //aa: Number of insertions required is 0. aa
    //abcd: Number of insertions required is 3. dcbabcd
    //abcd: Number of insertions required is 3. dcbabcd or abcacba or abcdcba or abcba
    //abcda: Number of insertions required is 2. adcbcda which is same as number of deletions in adcbcba
    //abcde: Number of insertions required is 4. edcbabcde
    //Approach: Dynamic Programming

    Codeium: Refactor | Explain | Generate Javadoc | ×
    public static void main(String[] args) {
        String s1 = "abcd";
        String s2 = new StringBuilder(s1).reverse().toString();
        System.out.println("Min Insertions: " + minInsertions(s1, s2));
    }

    Codeium: Refactor | Explain | Generate Javadoc | ×
    public static int minInsertions(String s1, String s2) {
        int m = s1.length();
        int n = s2.length();

        int[][] dp = new int[m + 1][n + 1];

        //Fill the dp table in bottom up manner
        for (int i = 1; i <= m ; i++) {
            for (int j = 1; j <= n ; j++) {
                if (s1.charAt(i - 1) == s2.charAt(j - 1)) {
                    //If the characters match, then add 1 to the result and move diagonally
                    dp[i][j] = 1 + dp[i - 1][j - 1];
                } else {
                    //If the characters don't match, then take the max of the two results
                    dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
                }

            }
        }

        return m - dp[m][n];
    }

```

```
.MinInsDelToMakePallindrome06
Min Insertions: 3
```

```java
Codeium: Refactor | Explain
public class PrintShortestSuperSequence07 {
    //Given two strings ‘X’ and ‘Y’, yprin the shortest supersequence such that both strings are subsequence
    // of the supersequence.
    // Examples : Input:   str1 = "geek",  str2 = "eke"  Output: "geeke"
    //Approach: Dynamic Programming

    Codeium: Refactor | Explain | Generate Javadoc | ×
    public static void main(String[] args) {
//        String s1 = "abcdgh";
//        String s2 = "abedfhr";
      String s1 = "geek";
      String s2 = "eke";
      System.out.println(printShortestCommonSupersequence(s1, s2));
    }

    Codeium: Refactor | Explain | Generate Javadoc | ×
    public static String printShortestCommonSupersequence(String s1, String s2) {
        int m = s1.length();
        int n = s2.length();

        int[][] dp = new int[m + 1][n + 1];

        //Fill the dp table in bottom up manner
        for (int i = 1; i <= m ; i++) {
            for (int j = 1; j <= n ; j++) {
                if (s1.charAt(i - 1) == s2.charAt(j - 1)) {
                    //If the characters match, then add 1 to the result and move diagonally
                    dp[i][j] = 1 + dp[i - 1][j - 1];
                } else {
                    //If the characters don't match, then take the max of the two results
                    dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
                }

            }
        }

        //print the shortest common supersequence
        int i = m;
        int j = n;
        StringBuilder sb = new StringBuilder();
        while (i > 0 && j > 0) {
            if (s1.charAt(i - 1) == s2.charAt(j - 1)) {
                sb.append(s1.charAt(i - 1));
                i--;
                j--;
            } else {
                if (dp[i - 1][j] > dp[i][j - 1]) {
                    sb.append(s1.charAt(i - 1));
                    i--;
                } else {
                    sb.append(s2.charAt(j - 1));
                    j--;
                }
            }
        }

        while (i > 0) {
            sb.append(s1.charAt(i - 1));
            i--;
        }

        while (j > 0) {
            sb.append(s2.charAt(j - 1));
            j--;
        }

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

```
.PrintShortestSuperSequence07
geeke
```

```java
Codeium: Refactor | Explain
public class LongestRepeatingSubsequence08 {
    //Given a string, find the length of the longest repeating subsequence such that the two subsequences don't have
    //same string character at the same position, i.e., any i'th character in the two subsequences shouldn't have the
    //same index in the original string.
    //Example: s1 = "aabebcdd" -> "abd" -> 3
    //Example: s1 = "aabb" -> "ab" -> 2
    //Approach: Dynamic Programming

    Codeium: Refactor | Explain | Generate Javadoc | ×
    public static void main(String[] args) {
        String s1 = "aabebcdd";
        System.out.println(lrs(s1));
    }

    Codeium: Refactor | Explain | Generate Javadoc | ×
    public static int lrs(String s1) {
        String s2 = s1;
        int m = s1.length();
        int n = s2.length();

        int[][] dp = new int[m + 1][n + 1];

        //Fill the dp table in bottom up manner
        for (int i = 1; i <= m ; i++) {
            for (int j = 1; j <= n ; j++) {
                if (s1.charAt(i - 1) == s2.charAt(j - 1) && i != j) {
                    //If the characters match, then add 1 to the result and move diagonally
                    dp[i][j] = 1 + dp[i - 1][j - 1];
                } else {
                    //If the characters don't match, then take the max of the two results
                    dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
                }

            }
        }

        return dp[m][n];
    }

}

```

```
.LongestRepeatingSubsequence08
3
```

```java
Codeium: Refactor | Explain
public class SequencePatternMatching09 {
    //Given two strings s and t, return true if s is a subsequence of t, or false otherwise.
    //A subsequence of a string is a new string that is formed from the original string by
    // deleting some (can be none) of the characters without disturbing the
    // relative positions of the
    // remaining characters. (i.e., "ace" is a subsequence of "abcde" while "aec" is not).
    //
    //Example 1:
    //Input: s = "abc", t = "ahbgdc"
    //Output: true

    Codeium: Refactor | Explain | Generate Javadoc | ×
    public static void main(String[] args) {
        String s = "abc";
        String t = "ahbgdc";
        System.out.println(isSubsequence(s, t));
    }

    Codeium: Refactor | Explain | Generate Javadoc | ×
    public static boolean isSubsequence(String s, String t) {
        int m = s.length();
        int n = t.length();

        int[][] dp = new int[m + 1][n + 1];

        //Fill the dp table in bottom up manner
        for (int i = 1; i <= m ; i++) {
            for (int j = 1; j <= n ; j++) {
                if (s.charAt(i - 1) == t.charAt(j - 1)) {
                    //If the characters match, then add 1 to the result and move diagonally
                    dp[i][j] = 1 + dp[i - 1][j - 1];
                } else {
                    //If the characters don't match, then take the max of the two results
                    dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
                }

            }
        }

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

```
.SequencePatternMatching09
true
```

```java
Codeium: Refactor | Explain
public class MinDeletionsToMakePallindrome10 {
   //Minimum number of deletions to make a string palindrome
    //Given a string of size ‘n’. The task is to remove or delete minimum number of
    // characters from the string so that the resultant string is palindrome.
    //Examples :
    //
    //Input : aebcbda
    //Output : 2
    //Remove characters 'e' and 'd'
    //Resultant string will be 'abcba'
    //which is a palindromic string

    //Approach: Dynamic Programming
    Codeium: Refactor | Explain | ×
    public static void main(String[] args) {
        String s1 = "aebcbda";
        String s2 = new StringBuilder(s1).reverse().toString();
        System.out.println("Min Deletions: " + minDeletions(s1, s2));
    }

    Codeium: Refactor | Explain | Generate Javadoc | ×
    public static int minDeletions(String s1, String s2) {
        int m = s1.length();
        int n = s2.length();

        int[][] dp = new int[m + 1][n + 1];

        //Fill the dp table in bottom up manner
        for (int i = 1; i <= m ; i++) {
            for (int j = 1; j <= n ; j++) {
                if (s1.charAt(i - 1) == s2.charAt(j - 1)) {
                    //If the characters match, then add 1 to the result and move diagonally
                    dp[i][j] = 1 + dp[i - 1][j - 1];
                } else {
                    //If the characters don't match, then take the max of the two results
                    dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
                }

            }
        }

        return m - dp[m][n];
    }


```

```
.MinDeletionsToMakePallindrome10
Min Deletions: 2
```