```java
Codeium: Refactor | Explain
public class MatrixchainMultiplication {
    //Given a sequence of matrices, find the most efficient way to multiply these matrices together.
    //The problem is not actually to perform the multiplications, but merely to decide in which order
    // to perform the multiplications.
    //We have many options to multiply a chain of matrices because matrix multiplication is associative.
    //In other words, no matter how we parenthesize the product, the result will be the same.
    //For example, if we had four matrices A, B, C, and D, we would have:
    //    (ABC)D = (AB)(CD) = A(BCD) = ....
    //However, the order in which we parenthesize the product affects the number of simple arithmetic
    // operations needed to compute the product, or the efficiency.
    //For example, suppose A is a 10 × 30 matrix, B is a 30 × 5 matrix, and C is a 5 × 60 matrix.
    //Then,

    //    (AB)C = (10×30×5) + (10×5×60) = 1500 + 3000 = 4500 operations
    //    A(BC) = (30×5×60) + (10×30×60) = 9000 + 18000 = 27000 operations.
    //Clearly the first parenthesization requires less number of operations.
//Given an array p[] which represents the chain of matrices such that the ith matrix Ai is of
//dimension p[i-1] x p[i]. We need to write a function MatrixChainOrder() that should return the
//minimum number of multiplications needed to multiply the chain.

    //Approach: Recursive
    Codeium: Refactor | Explain | ×
    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 3};
        System.out.println(matrixChainOrder(arr, 1, arr.length - 1));
        System.out.println("matrixChainOrderMemo ->"+matrixChainOrderMemo(arr, 1, arr.length - 1));
        System.out.println("matrixChainOrderDP ->"+matrixChainOrderDP(arr));
    }


        Codeium: Refactor | Explain | Generate Javadoc | ×
        public static int matrixChainOrder(
                int[] arr, int i, int j) {
            if (i >= j) {
                return 0;
            }

            int min = Integer.MAX_VALUE;
            for (int k = i; k < j; k++) {
                int count = matrixChainOrder(arr, i, k) +
                        matrixChainOrder(arr, k + 1, j)
                        + arr[i - 1] * arr[k] * arr[j];
                if (count < min) {
                    min = count;
                }
            }
            return min;
        }
```

```java
//Approach: Memoization
Codeium: Refactor | Explain | ×
public static int matrixChainOrderMemo(
        int[] arr, int i, int j) {
    int[][] dp = new int[arr.length + 1][arr.length + 1];
    return matrixChainOrderMemo(arr, i, j, dp);
}

Codeium: Refactor | Explain | Generate Javadoc | ×
public static int matrixChainOrderMemo(
        int[] arr, int i, int j, int[][] dp) {
    if (i >= j) {
        return 0;
    }

    if (dp[i][j] != 0) {
        return dp[i][j];
    }

    int min = Integer.MAX_VALUE;
    for (int k = i; k < j; k++) {
        int count = matrixChainOrderMemo(arr, i, k, dp) +
                matrixChainOrderMemo(arr, k + 1, j, dp) +
                arr[i - 1] * arr[k] * arr[j];
        if (count < min) {
            min = count;
        }
    }
    dp[i][j] = min;
    return dp[i][j];
}

```

```java
    //Approach: Dynamic Programming
    Codeium: Refactor | Explain | ×
    public static int matrixChainOrderDP(int[] arr) {
        int n = arr.length;
        int[][] dp = new int[n][n];

        for (int i = 1; i < n; i++) {
            dp[i][i] = 0;
        }

        for (int l = 2; l < n; l++) {
            for (int i = 1; i < n - l + 1; i++) {
                int j = i + l - 1;
                if (j == n) {
                    continue;
                }
                dp[i][j] = Integer.MAX_VALUE;
                for (int k = i; k < j; k++) {
                    int count = dp[i][k] + dp[k + 1][j] + arr[i - 1] * arr[k] * arr[j];
                    if (count < dp[i][j]) {
                        dp[i][j] = count;
                    }
                }
            }
        }
        return dp[1][n - 1];
    }
}

```

```
30
matrixChainOrderMemo ->30
matrixChainOrderDP ->30
```

```java
Codeium: Refactor | Explain
public class BooleanParenthesis {
    //Given a boolean expression with following symbols.
    // Symbols 'T' ---> true 'F' ---> false And
    // following operators filled between symbols // Operators & ---> boolean AND | ---> boolean OR ^ ---> boolean XOR
    // Count the number of ways we can parenthesize the expression so that the value of expression evaluates to true.
    // For Example:
    // Expression: T|T&F^T
    // Ways: ((T|T)&(F^T)), (T|(T&(F^T))), (((T|T)&F)^T)
    // Expression: T^F|F
    // Ways: (T^(F|F)), ((T^F)|F)
    // Expression: T|F^T&T
    // Ways: (T|(F^(T&T))), (((T|F)^T)&T), (T|((F^T)&T)), (T|((F^T)&T))

    //Approach: Recursive
    Codeium: Refactor | Explain |
    public static void main(String[] args) {
        String s = "T|T&F^T";
        System.out.println(countWays(s, 0, s.length() - 1, true));
    }

    Codeium: Refactor | Explain | Generate Javadoc |
```

```java
Codeium: Refactor | Explain | Generate Javadoc | X
public static int countWays(
        String s, int i, int j, boolean isTrue) {
    if (i > j) {
        return 0;
    }

    if (i == j) {
        if (isTrue) {
            return s.charAt(i) == 'T' ? 1 : 0;
        } else {
            return s.charAt(i) == 'F' ? 1 : 0;
        }
    }

    int ans = 0;
    for (int k = i + 1; k < j; k += 2) {
        int lT = countWays(s, i, k - 1, true);
        int lF = countWays(s, i, k - 1, false);
        int rT = countWays(s, k + 1, j, true);
        int rF = countWays(s, k + 1, j, false);

        if (s.charAt(k) == '&') {
            if (isTrue) {
                ans += lT * rT;
            } else {
                ans += lT * rF + lF * rT + lF * rF;
            }
        } else if (s.charAt(k) == '|') {
            if (isTrue) {
                ans += lT * rT + lT * rF + lF * rT;
            } else {
                ans += lF * rF;
            }
        } else if (s.charAt(k) == '^') {
            if (isTrue) {
                ans += lT * rF + lF * rT;
            } else {
                ans += lT * rT + lF * rF;
            }
        }
    }
    return ans;
}

```

```java
    //Approach: Memoization
    Codeium: Refactor | Explain | ×
    public static int countWaysMemo(
            String s, int i, int j, boolean isTrue) {
        int[][][] dp = new int[s.length() + 1][s.length() + 1][2];
        return countWaysMemo(s, i, j, isTrue, dp);
    }

    Codeium: Refactor | Explain | Generate Javadoc | ×
    public static int countWaysMemo(
            String s, int i, int j, boolean isTrue, int[][][] dp) {
        if (i > j) {
            return 0;
        }

        if (i == j) {
            if (isTrue) {
                return s.charAt(i) == 'T' ? 1 : 0;
            } else {
                return s.charAt(i) == 'F' ? 1 : 0;
            }
        }

        if (dp[i][j][isTrue ? 1 : 0] != 0) {
            return dp[i][j][isTrue ? 1 : 0];
        }


        int ans = 0;
        for (int k = i + 1; k < j; k += 2) {
            int lT = countWaysMemo(s, i, k - 1, true, dp);
            int lF = countWaysMemo(s, i, k - 1, false, dp);
            int rT = countWaysMemo(s, k + 1, j, true, dp);
            int rF = countWaysMemo(s, k + 1, j, false, dp);

            if (s.charAt(k) == '&') {
                if (isTrue) {
                    ans += lT * rT;
                } else {
                    ans += lT * rF + lF * rT + lF * rF;
                }
            } else if (s.charAt(k) == '|') {
                if (isTrue) {
                    ans += lT * rT + lT * rF + lF * rT;
                } else {
                    ans += lF * rF;
                }
            } else if (s.charAt(k) == '^') {
                if (isTrue) {
                    ans += lT * rF + lF * rT;
                } else {
                    ans += lT * rT + lF * rF;
                }
            }
        }
        return dp[i][j][isTrue ? 1 : 0] = ans;
    }

    //Approach: Tabulation
```

```java
    //Approach: Tabulation
    Codeium: Refactor | Explain | ×
    public static int countWaysTab(String s, int i, int j, boolean isTrue) {
        int[][][] dp = new int[s.length() + 1][s.length() + 1][2];
        for (int gap = 0; gap < s.length(); gap++) {
            for (i = 0, j = gap; j < s.length(); i++, j++) {
                if (i == j) {
                    if (isTrue) {
                        dp[i][j][1] = s.charAt(i) == 'T' ? 1 : 0;
                    } else {
                        dp[i][j][0] = s.charAt(i) == 'F' ? 1 : 0;
                    }
                } else {
                    for (int k = i + 1; k < j; k += 2) {
                        int lT = dp[i][k - 1][1];
                        int lF = dp[i][k - 1][0];
                        int rT = dp[k + 1][j][1];
                        int rF = dp[k + 1][j][0];

                        if (s.charAt(k) == '&') {
                            if (isTrue) {
                                dp[i][j][1] += lT * rT;
                            } else {
                                dp[i][j][0] += lT * rF + lF * rT + lF * rF;
                            }
                        } else if (s.charAt(k) == '|') {
                            if (isTrue) {
                                dp[i][j][1] += lT * rT + lT * rF + lF * rT;
                            } else {
                                dp[i][j][0] += lF * rF;
                            }
                        } else if (s.charAt(k) == '^') {
                            if (isTrue) {
                                dp[i][j][1] += lT * rF + lF * rT;
                            } else {
                                dp[i][j][0] += lT * rT + lF * rF;
                            }
                        }
                    }
                }
            }
        }
        return dp[0][s.length() - 1][isTrue ? 1 : 0];
    }
```

.MatrixChainMultiplication.BooleanParenthesis

4

```java
public class EggDroppingProblemn {
    //Given a certain number of floors and a certain number of eggs, find the minimum number of attempts needed to
    // find the threshold floor from which the egg breaks.
    //An egg that survives a fall can be used again.
    //A broken egg must be discarded.
    //For example, if the threshold is 16 and we have 2 eggs, then the minimum number of attempts is 4.
    //We can drop from floor 10, 16, 13, 14, 15
    //Approach: Recursive
    Codeium: Refactor | Explain | ×
    public static void main(String[] args) {
        int floors = 16;
        int eggs = 2;
        System.out.println(minAttempts(floors, eggs));
    }

    Codeium: Refactor | Explain | Generate Javadoc | ×
    public static int minAttempts(int floors, int eggs) {
        if (floors == 0 || floors == 1 || eggs == 1) {
            return floors;
        }

        int min = Integer.MAX_VALUE;
        for (int i = 1; i <= floors; i++) {
            int count = 1 + Math.max(minAttempts(i - 1,
                                     eggs - 1),
                                 minAttempts(floors - i, eggs));
            if (count < min) {
                min = count;
            }
        }
        return min;
    }

    //Approach: Memoization
    Codeium: Refactor | Explain | ×
    public static int minAttemptsMemo(int floors, int eggs) {
        int[][] dp = new int[floors + 1][eggs + 1];
        return minAttemptsMemo(floors, eggs, dp);
    }

    Codeium: Refactor | Explain | Generate Javadoc | ×
    public static int minAttemptsMemo(int floors, int eggs, int[][] dp) {
        if (floors == 0 || floors == 1 || eggs == 1) {
            return floors;
        }

        if (dp[floors][eggs] != 0) {
            return dp[floors][eggs];
        }

        int min = Integer.MAX_VALUE;
        for (int i = 1; i <= floors; i++) {
            int count = 1 + Math.max(minAttemptsMemo(i - 1, eggs - 1, dp),
                                     minAttemptsMemo(floors - i, eggs, dp));
            if (count < min) {
                min = count;
            }
        }
        dp[floors][eggs] = min;
        return dp[floors][eggs];
    }

```

```java
    //Approach: Dynamic Programming
    Codeium: Refactor | Explain | ×
    public static int minAttemptsDP(int floors, int eggs) {
        int[][] dp = new int[floors + 1][eggs + 1];

        for (int i = 1; i <= floors; i++) {
            dp[i][1] = i;
        }

        for (int i = 1; i <= eggs; i++) {
            dp[1][i] = 1;
        }

        for (int i = 2; i <= floors; i++) {
            for (int j = 2; j <= eggs; j++) {
                dp[i][j] = Integer.MAX_VALUE;
                for (int k = 1; k <= i; k++) {
                    int count = 1 + Math.max(dp[k - 1][j - 1], dp[i - k][j]);
                    if (count < dp[i][j]) {
                        dp[i][j] = count;
                    }
                }
            }
        }
        return dp[floors][eggs];
    }


}

```

```
.MatrixChainMultiplication.EggDroppingProblemn
6
```

```java
Codeium: Refactor | Explain
public class PallindromePartition {
    // Given a string, a partitioning of the string is a palindrome partitioning if every substring of the partition is a palindrome.
    // For example, “aba|b|bbabb|a|b|aba” is a palindrome partitioning of “ababbbabbababa”.
    // Determine the fewest cuts needed for palindrome partitioning of a given string.
    // For example, minimum 3 cuts are needed for “ababbbabbababa”.

    //Approach: Recursive
    Codeium: Refactor | Explain | ×
    public static void main(String[] args) {
        String s = "ababbbabbababa";
        System.out.println(minCuts(s, 0, s.length() - 1));
    }

    Codeium: Refactor | Explain | Generate Javadoc | ×
    public static int minCuts(String s, int i, int j) {
        if (i >= j) {
            return 0;
        }

        if (isPallindrome(s, i, j)) {
            return 0;
        }

        int min = Integer.MAX_VALUE;
        for (int k = i; k < j; k++) {
            int count = minCuts(s, i, k) + minCuts(s, k + 1, j) + 1;
            if (count < min) {
                min = count;
            }
        }
        return min;
    }

```

```java
public static boolean isPallindrome(String s, int i, int j) {
    while (i < j) {
        if (s.charAt(i++) != s.charAt(j--)) {
            return false;
        }
    }
    return true;
}

//Approach: Memoization
Codeium: Refactor | Explain | ×
public static int minCutsMemo(String s, int i, int j) {
    int[][] dp = new int[s.length() + 1][s.length() + 1];
    return minCutsMemo(s, i, j, dp);
}

Codeium: Refactor | Explain | Generate Javadoc | ×
public static int minCutsMemo(String s, int i, int j, int[][] dp) {
    if (i >= j) {
        return 0;
    }

    if (isPallindrome(s, i, j)) {
        return 0;
    }

    if (dp[i][j] != 0) {
        return dp[i][j];
    }

    int min = Integer.MAX_VALUE;
    for (int k = i; k < j; k++) {
        int count = minCutsMemo(s, i, k, dp) +
                minCutsMemo(s, k + 1, j, dp) + 1;
        if (count < min) {
            min = count;
        }
    }
    dp[i][j] = min;
    return dp[i][j];
}

```

```java

    //Approach: Dynamic
    Codeium: Refactor | Explain | ×
    public static int minCutsDP(String s) {
        int n = s.length();
        int[][] dp = new int[n][n];

        for (int i = 1; i < n; i++) {
            dp[i][i] = 0;
        }

        for (int l = 2; l < n; l++) {
            for (int i = 1; i < n - l + 1; i++) {
                int j = i + l - 1;
                if (isPallindrome(s, i, j)) {
                    dp[i][j] = 0;
                } else {
                    dp[i][j] = Integer.MAX_VALUE;
                    for (int k = i; k < j; k++) {
                        int count = dp[i][k] + dp[k + 1][j] + 1;
                        if (count < dp[i][j]) {
                            dp[i][j] = count;
                        }
                    }
                }
            }
        }
        return dp[1][n - 1];
    }

    //Approach: Try to find it in most optimized way


}


```

.MatrixChainMultiplication.PallindromePartition

3

```java
Codeium: Refactor | Explain
public class ScrambledString {
    //Given two strings s1 and s2 of the same length, determine if s2 is a scrambled string of s1.
    //A string is a scrambled string of another string if it can be obtained by swapping some characters of the other string.
    //Example: s1 = "great", s2 = "rgeat" -> true
    //Example: s1 = "abcde", s2 = "caebd" -> false

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

    //Approach: Recursive
    Codeium: Refactor | Explain | ×
    public static boolean isScrambled(String s1, String s2) {
        if (s1.equals(s2)) {
            return true;
        }

        if (s1.length() <= 1) {
            return false;
        }

        int n = s1.length();
        boolean flag = false;
        for (int i = 1; i < n; i++) {
            if ((isScrambled(s1.substring(0, i), s2.substring(n - i))
                    && isScrambled(s1.substring(i), s2.substring(0, n - i))) ||
                    (isScrambled(s1.substring(0, i), s2.substring(0, i)) &&
                            isScrambled(s1.substring(i), s2.substring(i)))) {
                flag = true;
                break;
            }
        }
        return flag;
    }

```

```java
    //Approach: Memoization
    Codeium: Refactor | Explain |
    public static boolean isScrambledMemo(String s1, String s2) {
        int n = s1.length();
        int[][][] dp = new int[n + 1][n + 1][n + 1];
        return isScrambledMemo(s1, s2, dp, 0, 0, n);
    }

    Codeium: Refactor | Explain | Generate Javadoc |
    public static boolean isScrambledMemo(
            String s1, String s2, int[][][] dp, int i, int j, int len) {
        if (i >= len || j >= len) {
            return false;
        }

        if (s1.substring(i).equals(s2.substring(j))) {
            return true;
        }

        if (len == 1) {
            return s1.charAt(i) == s2.charAt(j);
        }

        if (dp[i][j][len] != 0) {
            return dp[i][j][len] == 1;
        }

        boolean flag = false;
        for (int k = 1; k < len; k++) {
            if ((isScrambledMemo(s1, s2, dp, i, j + len - k, k) &&
                    isScrambledMemo(s1, s2, dp, i + k, j, len - k)) ||
                    (isScrambledMemo(s1, s2, dp, i, j, k) &&
                            isScrambledMemo(s1, s2, dp, i + k, j + k, len - k))) {
                flag = true;
                break;
            }
        }
        dp[i][j][len] = flag ? 1 : -1;
        return flag;
    }

```

```java

    //Approach: Dynamic Programming
    Codeium: Refactor | Explain | ×
    public static boolean isScrambledDP(String s1, String s2) {
        int n = s1.length();
        boolean[][][] dp = new boolean[n][n][n + 1];

        for (int len = 1; len <= n; len++) {
            for (int i = 0; i <= n - len; i++) {
                for (int j = 0; j <= n - len; j++) {
                    if (len == 1) {
                        dp[i][j][len] = s1.charAt(i) == s2.charAt(j);
                    } else {
                        for (int k = 1; k < len; k++) {
                            if ((dp[i][j + len - k][k] && dp[i + k][j][len - k]) ||
                                    (dp[i][j][k] && dp[i + k][j + k][len - k])) {
                                dp[i][j][len] = true;
                                break;
                            }
                        }
                    }
                }
            }
        }
        return dp[0][0][n];
    }


}

```

```
 .MatrixChainMultiplication.ScrambledString
true
```