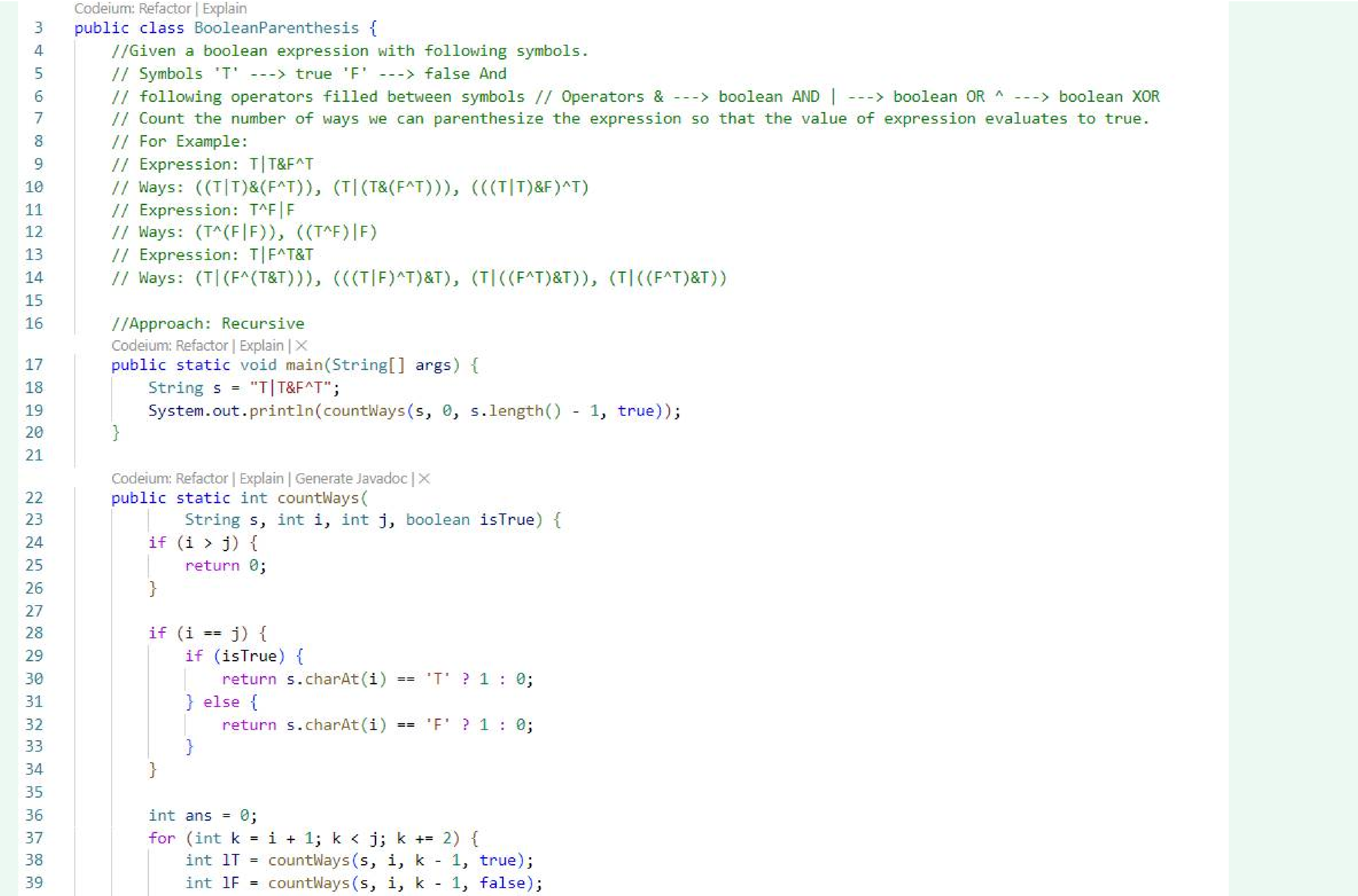
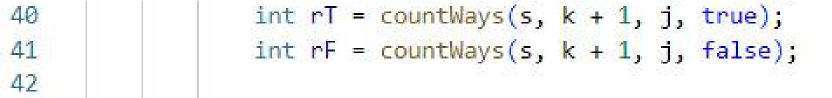
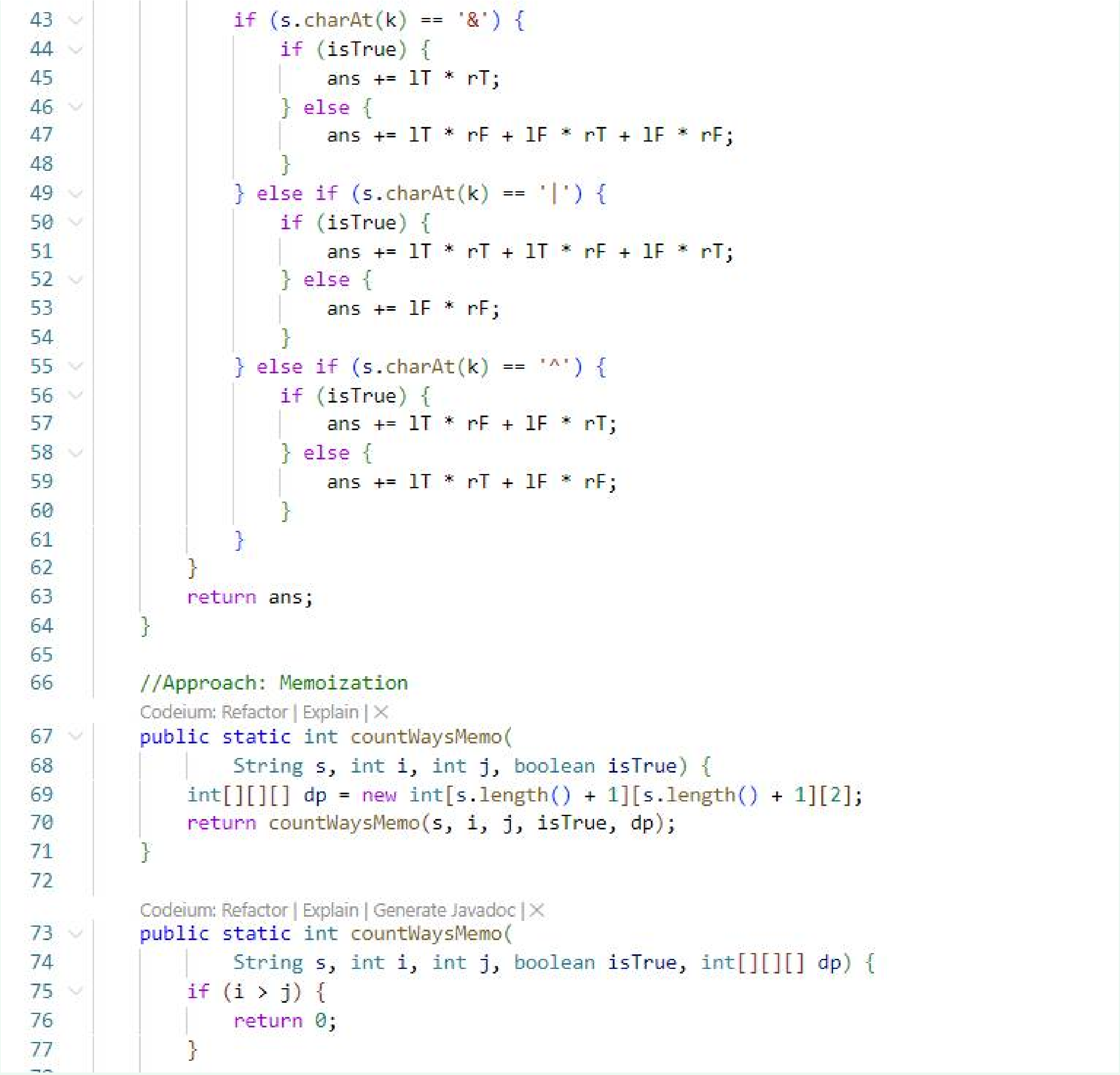
MatrixChainMultiplication-DP


Codeium: Refactor | Explain
Codeium: Refactor | Explain | ×
Codeium: Refactor | Explain | Generate Javadoc | ×

```java
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
    public static void main(String[] args) {
        String s = "T|T&F^T";
        System.out.println(countWays(s, 0, s.length() - 1, true));
    }

    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);
```

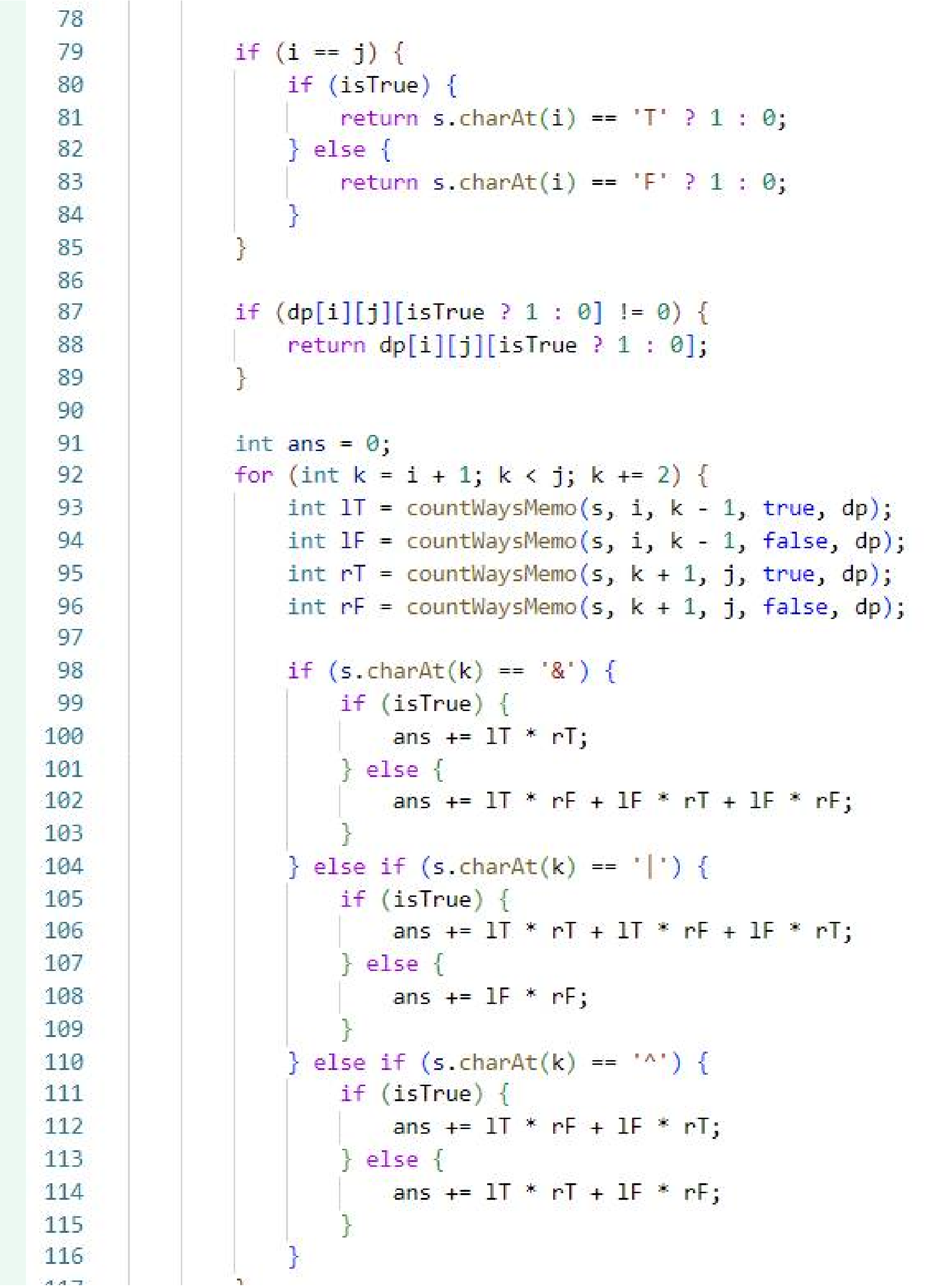


```
            int rT = countWays(s, k + 1, j, true);
            int rF = countWays(s, k + 1, j, false);

```



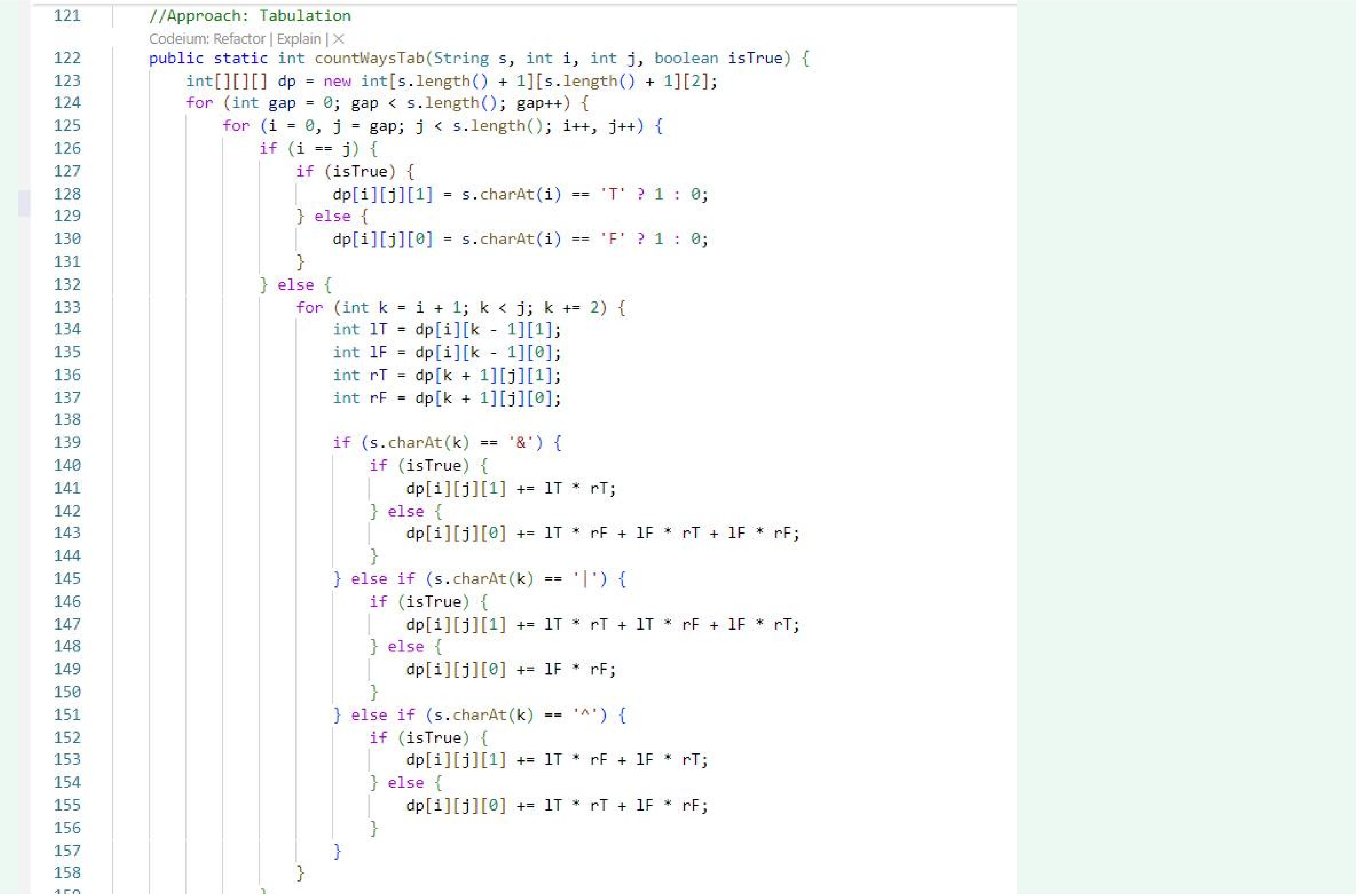
```
            if (s.charAt(k) == '&') {
                if (isTrue) {
                    ans += lT * rT;
                } else {
                    ans += lT * rF + lF * rT + lF * rF;
                }
            } else if (s.charAt(k) == '|') {
                if (isTrue) {
                  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

    //Approach: Memoization
    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);
    }

    public static int countWaysMemo(
            String s, int i, int j, boolean isTrue, int[][][] dp) {
        if (i > j) {
            return 0;
        }
```

Codeium: Refactor | Explain | ×
Codeium: Refactor | Explain | Generate Javadoc | ×

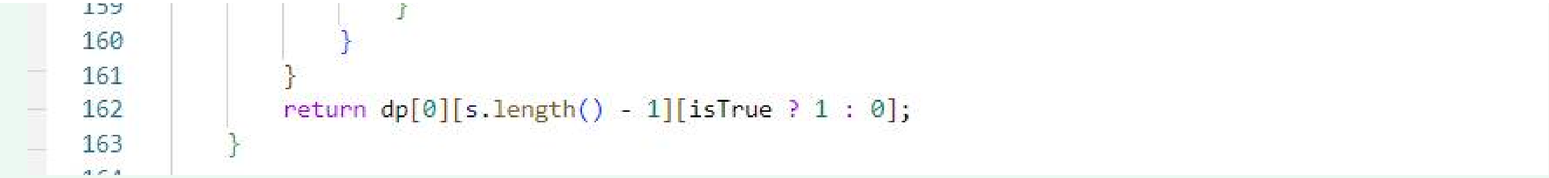


```

            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;
                    }
                }
```

```
        }
```



```java
    //Approach: Tabulation
    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];
}
```

Kindly write the above code from diagram and explain it with example and output

The code in the images is focused on solving the "Boolean Parenthesization Problem." Let me break it down for you, write the code, explain its functionality with an example, and provide the output.

Problem Description:

Given a boolean expression consisting of:

- Symbols: `` `'T'` `` (True) and `` `'F'` `` (False),
- Operators: `` `'&'` `` (AND), `` `'|'` `` (OR), and `` `'^'` `` (XOR),

Count the number of ways the expression can be parenthesized such that the result evaluates to `` `True` `` or `` `False` ``.

Approaches:

1. **Recursive**: Count the number of ways recursively by considering every operator as a potential splitting point.
2. **Memoization**: Use a 3D array to store results of overlapping subproblems for optimization.
3. **Tabulation**: Use bottom-up dynamic programming.

Full Code Implementation:

```java
public class BooleanParenthesis {

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

    // Memoization Approach
    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] != -1) 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 += lF * rT + lT * rF + lF * rF;
            } else if (s.charAt(k) == '|') {
                if (isTrue) ans += lT * rT + lF * rT + lT * rF;
                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;
    }

    // Tabulation Approach
    public static int countWaysTab(String s) {
        int n = s.length();
        int[][][] dp = new int[n][n][2];

        for (int gap = 0; gap < n; gap++) {
            for (int i = 0, j = gap; j < n; i++, j++) {
                if (i == j) {
                    dp[i][j][1] = s.charAt(i) == 'T' ? 1 : 0;
                    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) == '&') {
                            dp[i][j][1] += lT * rT;
                            dp[i][j][0] += lF * rT + lT * rF + lF * rF;
                        } else if (s.charAt(k) == '|') {
                            dp[i][j][1] += lT * rT + lF * rT + lT * rF;
                            dp[i][j][0] += lF * rF;
                        } else if (s.charAt(k) == '^') {
                            dp[i][j][1] += lT * rF + lF * rT;
                            dp[i][j][0] += lT * rT + lF * rF;
                        }
                    }
                }
            }
        }
        return dp[0][n - 1][1];
    }

    public static void main(String[] args) {
        String expression = "T|T&F^T";
        System.out.println("Recursive: " + countWays(expression, 0, expression.length() - 1, true));

        int[][][] dp = new int[expression.length()][expression.length()][2];
        for (int[][] row : dp)
            for (int[] col : row)
                java.util.Arrays.fill(col, -1);
        System.out.println("Memoization: " + countWaysMemo(expression, 0, expression.length() - 1, true, dp));

        System.out.println("Tabulation: " + countWaysTab(expression));
        }
    }
```

Example:

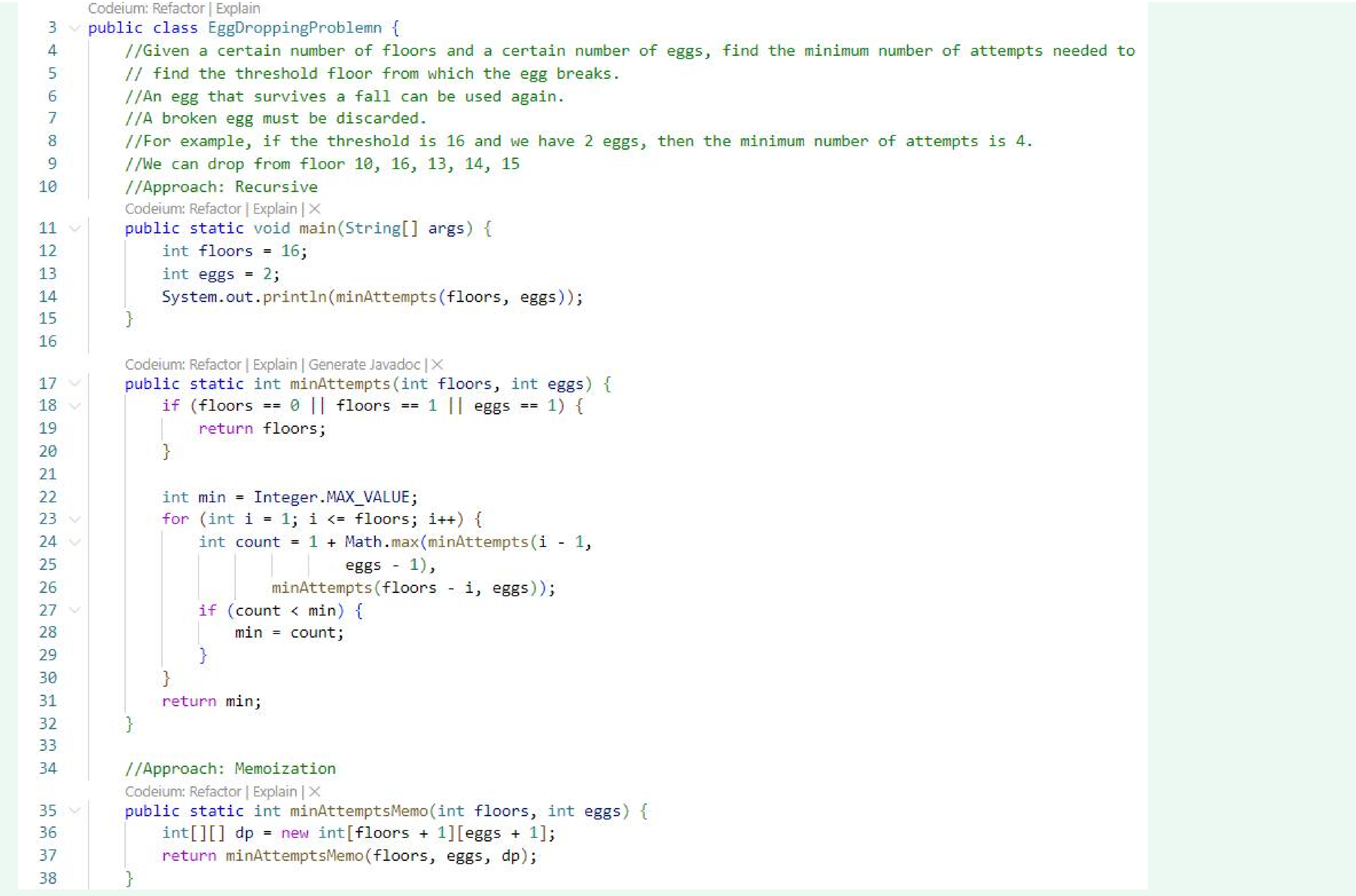
Input: `T|T&F^T`

Output:

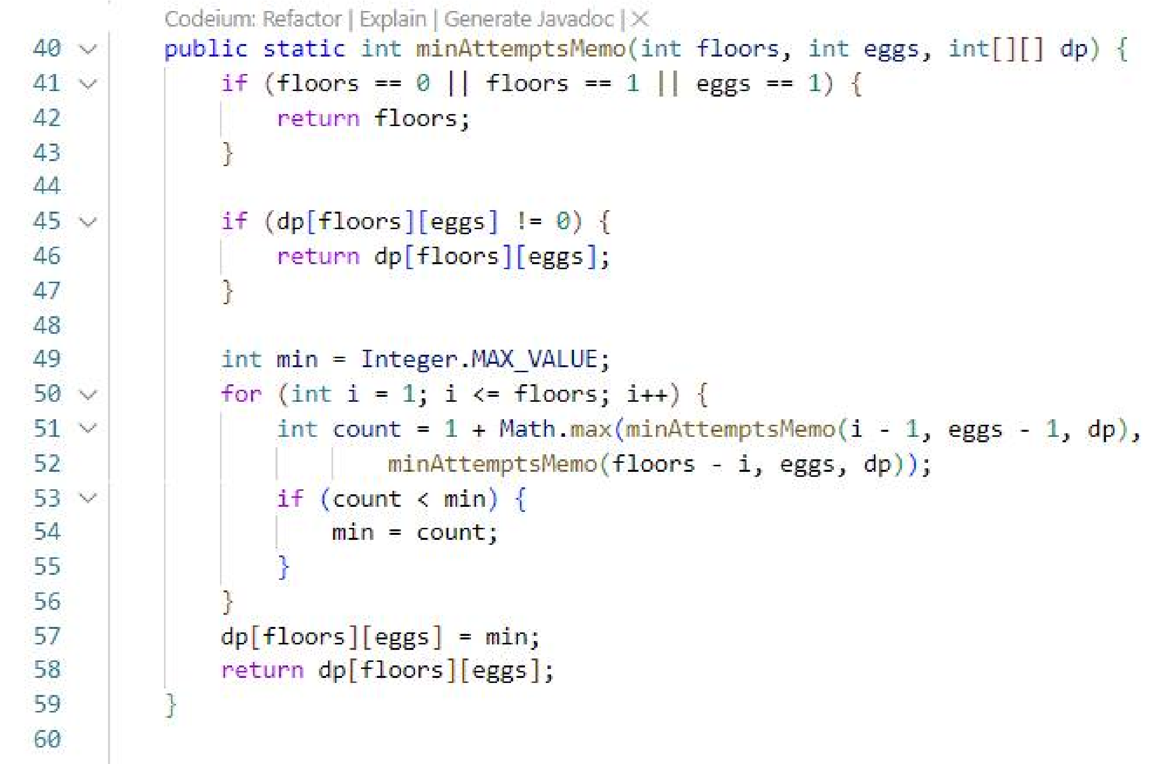
- Recursive: 4
- Memoization: 4
- Tabulation: 4

Explanation: There are **4 ways** to parenthesize the expression such that it evaluates to `True`:

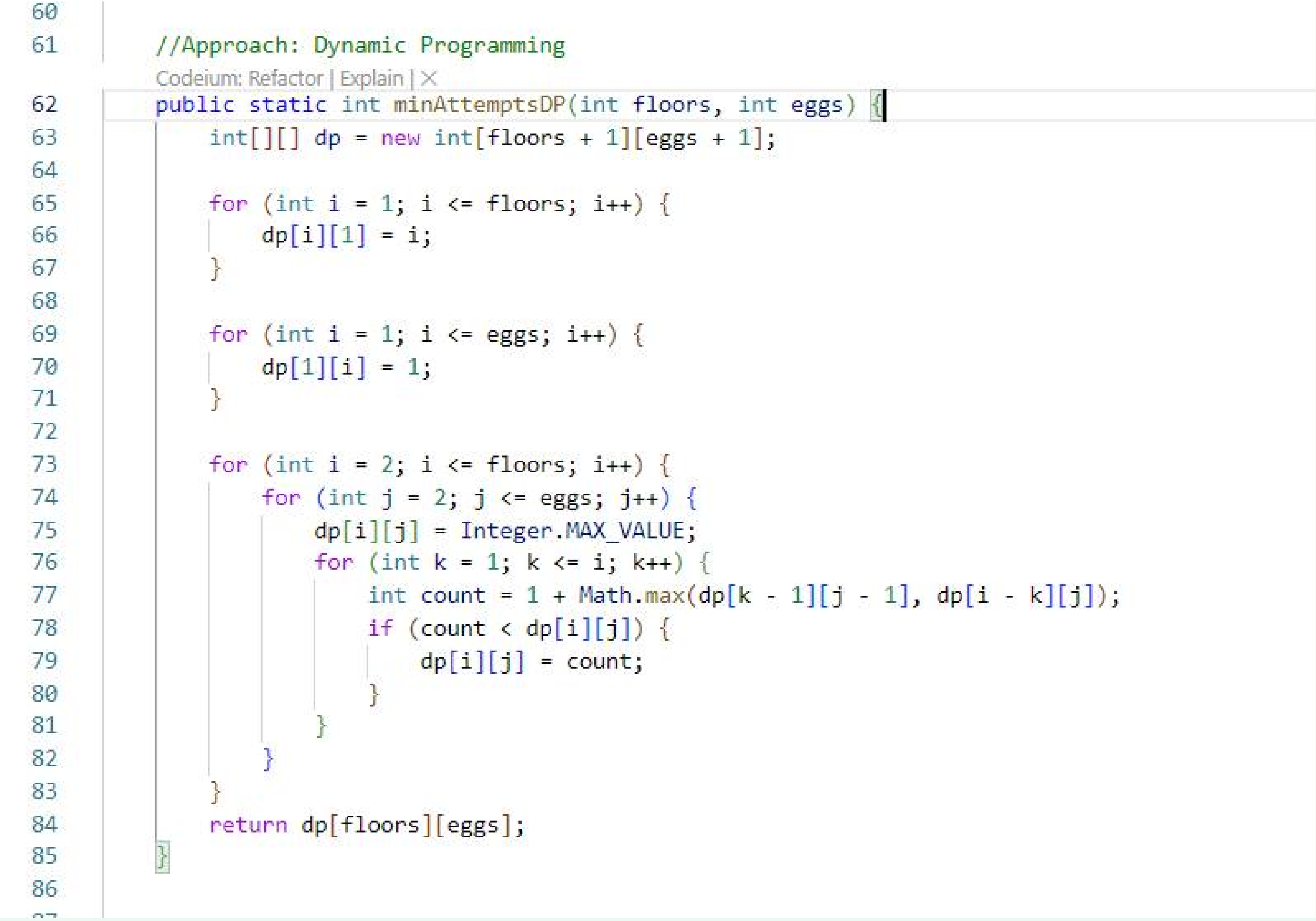
1. `(T|((T&F)^T))`
2. `((T|T)&(F^T))`
3. `(((T|T)&F)^T)`
4. `(T|(T&(F^T)))`


Codeium: Refactor | Explain
Codeium: Refactor | Explain | ×
Codeium: Refactor | Explain | Generate Javadoc | ×
Codeium: Refactor | Explain | ×

```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
    public static void main(String[] args) {
        int floors = 16;
        int eggs = 2;
        System.out.println(minAttempts(floors, eggs));
    }

    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
    public static int minAttemptsMemo(int floors, int eggs) {
        int[][] dp = new int[floors + 1][eggs + 1];
        return minAttemptsMemo(floors, eggs, dp);
    }
```



```java
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
    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];
    }

```

Kindly write the above code from diagram and explain it with example and output

Below is the Java code for the **Egg Dropping Problem** based on the diagrams you provided, with detailed explanations and an example.

Problem Description:

The **Egg Dropping Problem** involves determining the minimum number of attempts needed to find the critical floor from which an egg will break when dropped. The rules are:

1. If an egg breaks when dropped, all eggs dropped from floors above will also break.
2. If an egg does not break, all eggs dropped from floors below will not break.
3. You need to minimize the worst-case number of attempts to determine the critical floor.

Code Implementation

java

```java
public class EggDroppingProblem {

    // Recursive Approach
    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;
    }

    // Memoization Approach
    public static int minAttemptsMemo(int floors, int eggs) {
        int[][] dp = new int[floors + 1][eggs + 1];
        return minAttemptsMemo(floors, eggs, dp);
    }

    private 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 min;
    }

    // Dynamic Programming Approach
    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; // If we have one egg, we must try every floor
        }

        for (int i = 1; i <= eggs; i++) {
            dp[1][i] = 1; // If we have one floor, we need only one attempt
        }

        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];
    }

    public static void main(String[] args) {
        int floors = 16;
        int eggs = 2;

        // Recursive Approach
        System.out.println("Recursive: " + minAttempts(floors, eggs));

        // Memoization Approach
        System.out.println("Memoization: " + minAttemptsMemo(floors, eggs));

        // Dynamic Programming Approach
        System.out.println("Dynamic Programming: " + minAttemptsDP(floors, eggs));
    }
}
```

Explanation:

1. **Recursive Approach**:
    - Try every possible floor as a drop point (`k`).
    - For each floor, calculate:
        - If the egg breaks: Check floors below (`i - 1`) with one less egg.
        - If the egg does not break: Check floors above (`floors - i`) with the same number of eggs.
    - Return the minimum of all worst-case attempts.
2. **Memoization Approach**:
    - Store results of subproblems in a 2D array `dp` to avoid recomputation.
3. **Dynamic Programming Approach**:
    - Build a table (`dp`) iteratively where `dp[i][j]` represents the minimum attempts needed for `i` floors and `j` eggs.
    - Use previously computed values to fill the table.

Example:

Input:

- Floors = `16`
- Eggs = `2`

Output:

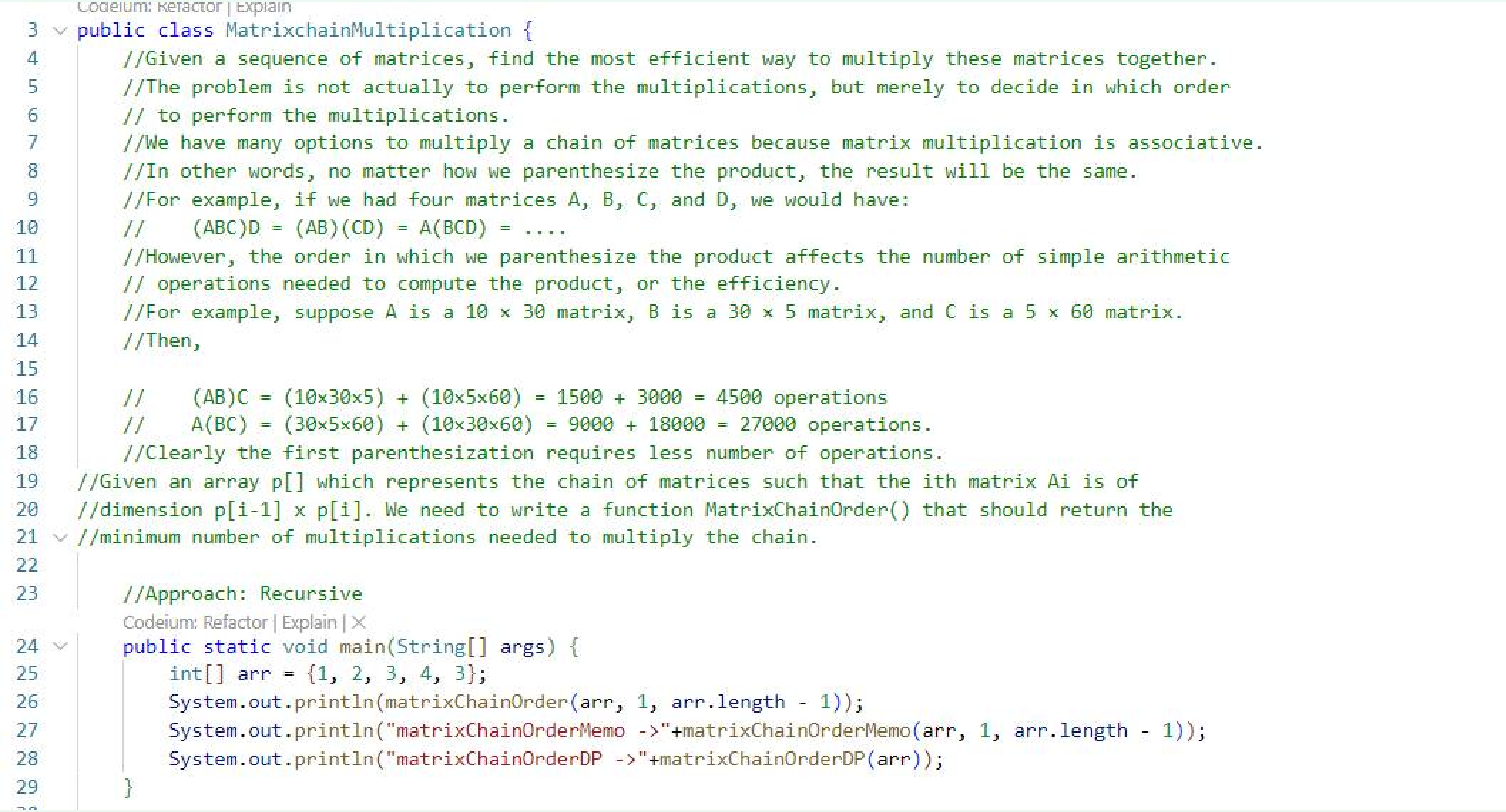
- Recursive: `5`
- Memoization: `5`
- Dynamic Programming: `5`

Explanation of Output:

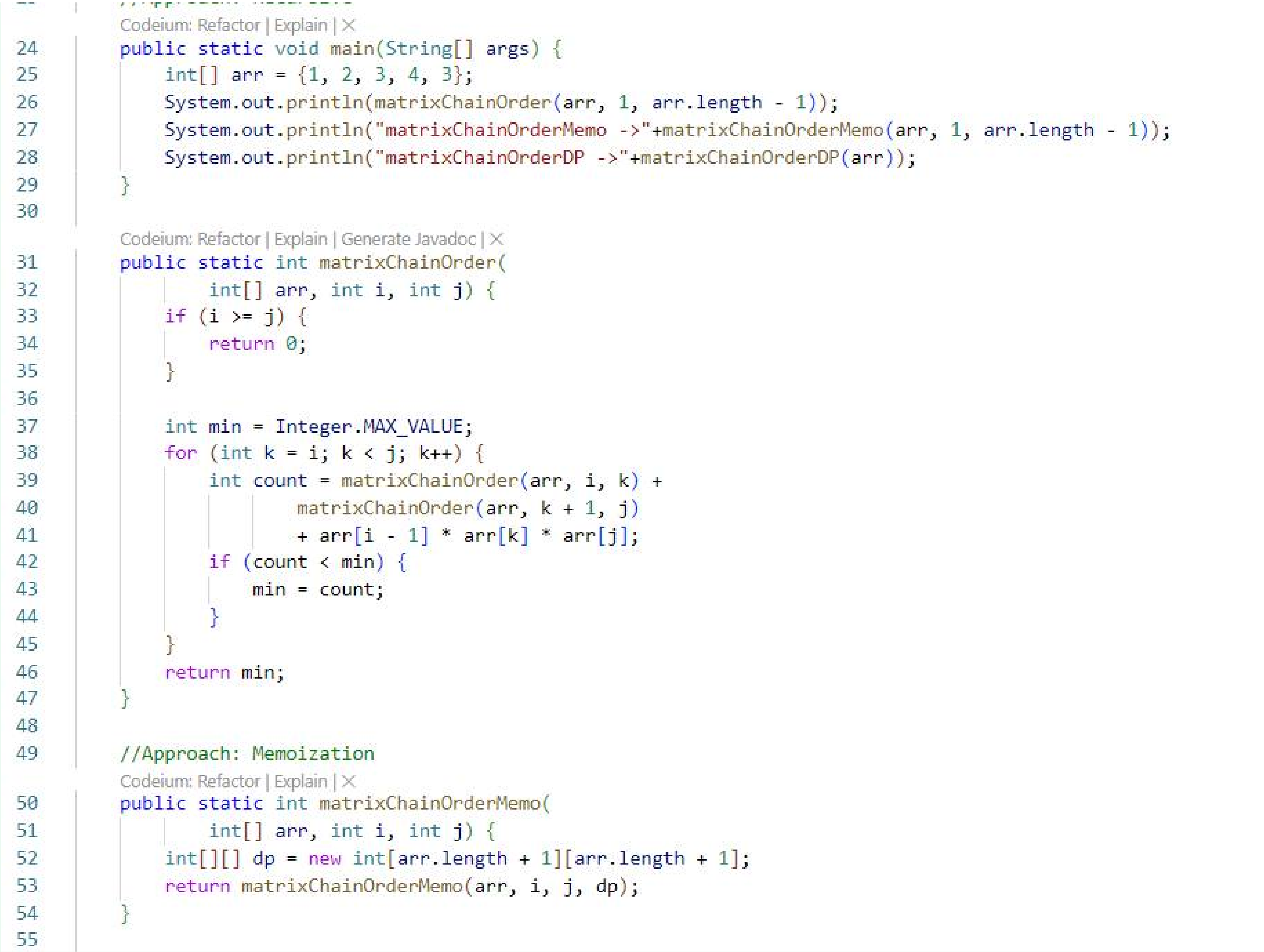
The critical floor is determined in a maximum of `5` attempts in the worst case:

1. Start from the middle floor to minimize attempts.

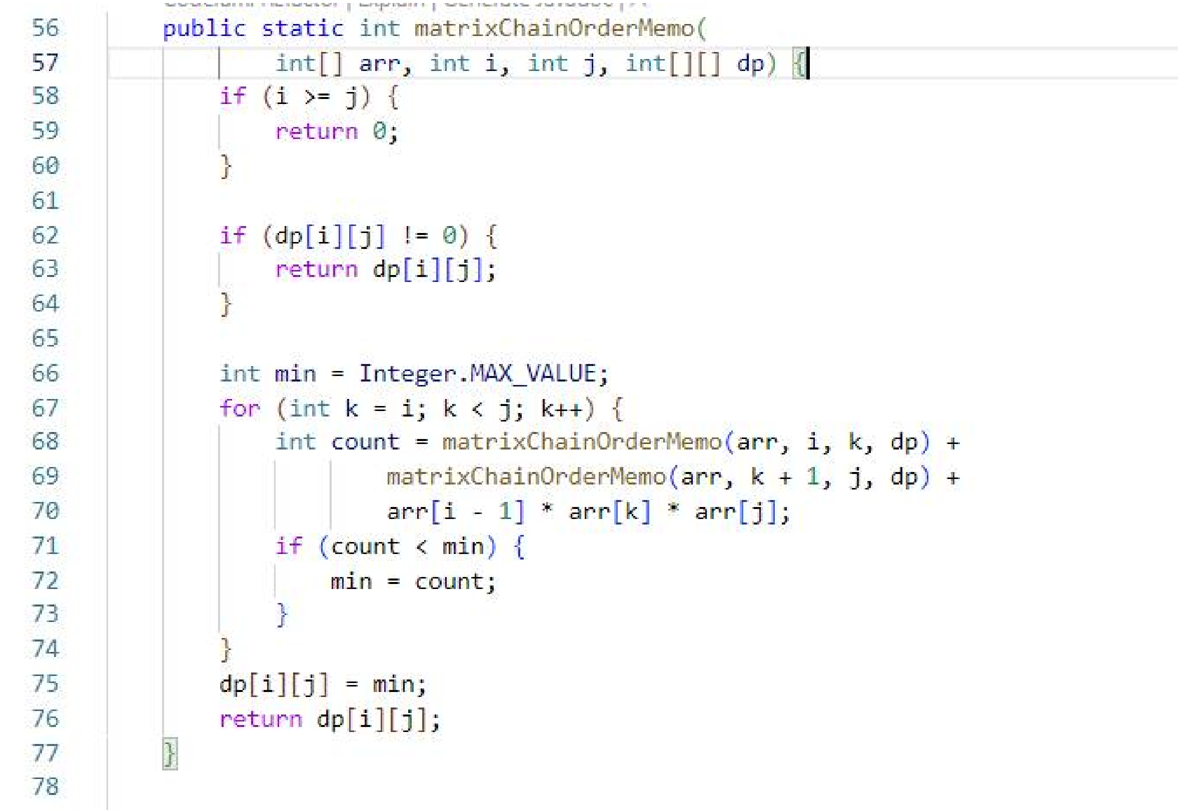
2. Drop eggs based on whether they break or not, reducing the problem size dynamically.



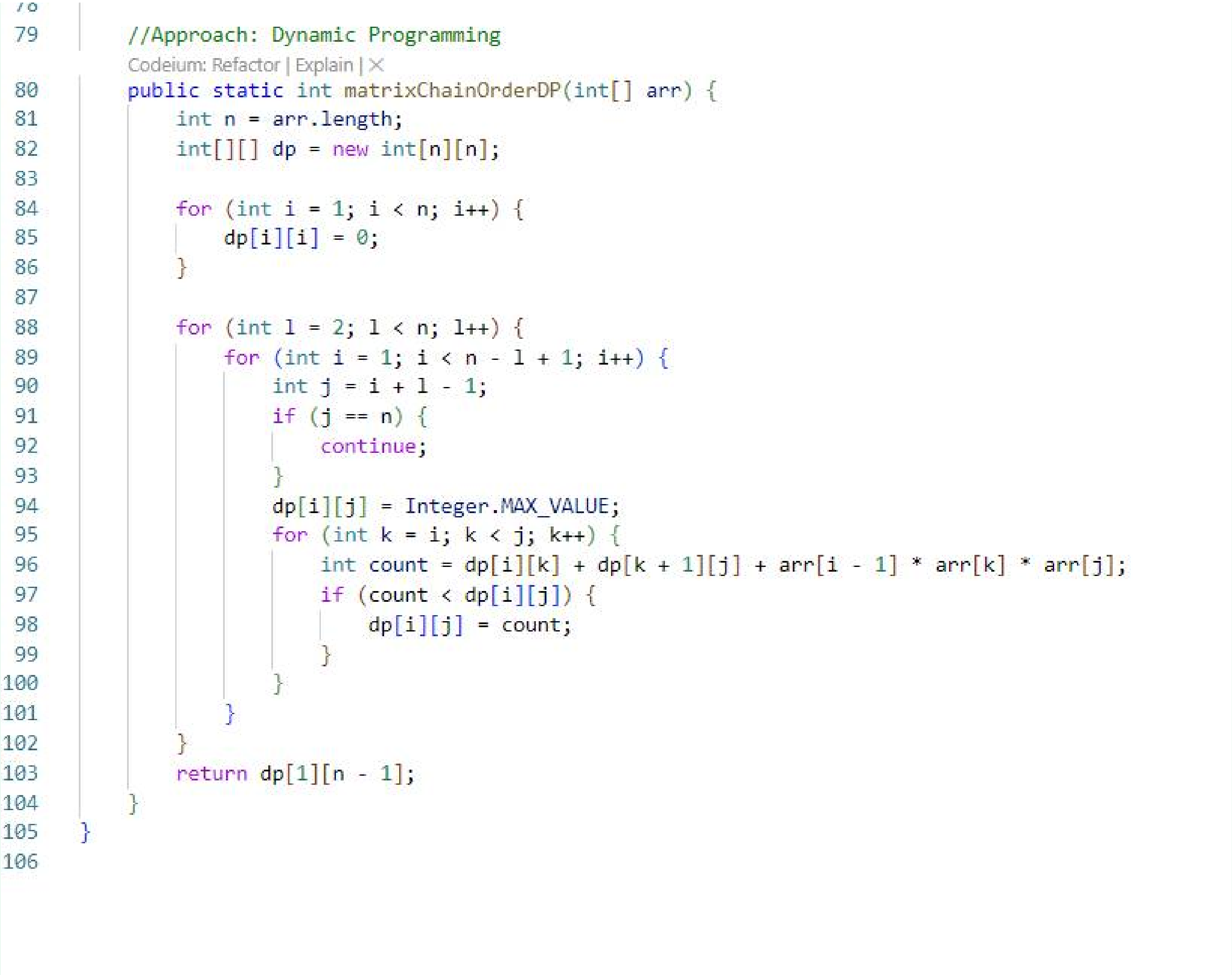
```
public class MatrixchainMultiplication {
    //Given a sequence of matrices, find the most efficient way to multiply these matrices together.
    //The problem is not actually to perform the multiplications, but merely to decide in which order
    // to perform the multiplications.
    //We have 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
    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 | ×
Codeium: Refactor | Explain | Generate Javadoc | ×
Codeium: Refactor | Explain | ×

```java
    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));
    }

    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;
    }

    //Approach: Memoization
    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);
    }

```



```java
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];
}
```



```
    //Approach: Dynamic Programming
    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];
    }
}
```

Kindly write the above code from diagram and explain it with example and output

Here is the Java implementation of the **Matrix Chain Multiplication Problem** from the diagrams, along with a detailed explanation and example:

Problem Description:

Given an array `` `arr[]` `` where the dimensions of the matrices are stored such that:

- The matrix `` `Ai` `` has dimensions `` `arr[i-1] x arr[i]` ``.

The goal is to find the **minimum number of scalar multiplications** required to multiply the chain of matrices using the best possible parenthesization.

Code Implementation

java

```java
public class MatrixChainMultiplication {

    // Recursive Approach
    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;
    }

    // Memoization Approach
    public static int matrixChainOrderMemo(int[] arr, int i, int j) {
        int[][] dp = new int[arr.length][arr.length];
        return matrixChainOrderMemo(arr, i, j, dp);
    }

    private 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 min;
    }

    // Dynamic Programming Approach
    public static int matrixChainOrderDP(int[] arr) {
        int n = arr.length;
        int[][] dp = new int[n][n];

        for (int l = 2; l < n; l++) {
            for (int i = 1; i < n - l + 1; i++) {
                int j = i + l - 1;
                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];
    }

    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 3};

        // Recursive Approach
        System.out.println("Recursive: " + matrixChainOrder(arr, 1, arr.length - 1));

        // Memoization Approach
        System.out.println("Memoization: " + matrixChainOrderMemo(arr, 1, arr.length - 1));

        // Dynamic Programming Approach
        System.out.println("Dynamic Programming: " + matrixChainOrderDP(arr));
    }
}
```

Explanation:

1. **Recursive Approach**:
   - Divide the array into two parts (`i to k` and `k+1 to j`) for each possible `k`.
   - Calculate the cost recursively for each subproblem and return the minimum cost.
2. **Memoization Approach**:
   - Use a 2D array `dp` to store the results of previously solved subproblems.
   - Avoid recomputation by checking if the result is already in `dp`.
3. **Dynamic Programming Approach**:
   - Use a bottom-up approach to fill the `dp` table.
   - `dp[i][j]` represents the minimum cost to multiply matrices from `i` to `j`.

Example:

Input:

`arr[] = {1, 2, 3, 4, 3}`

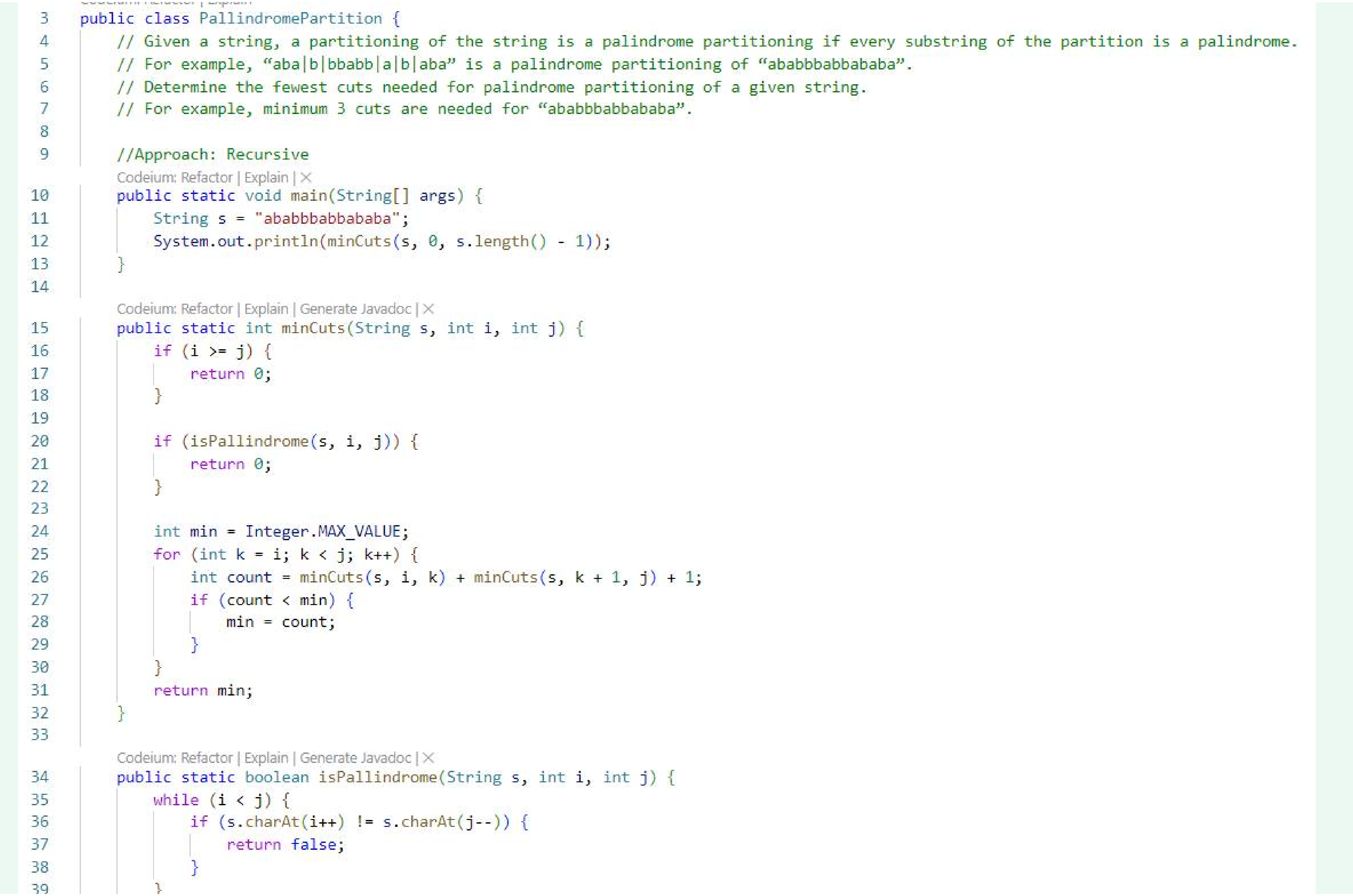
Output:

- Recursive: `30`
- Memoization: `30`
- Dynamic Programming: `30`

Explanation of Output:

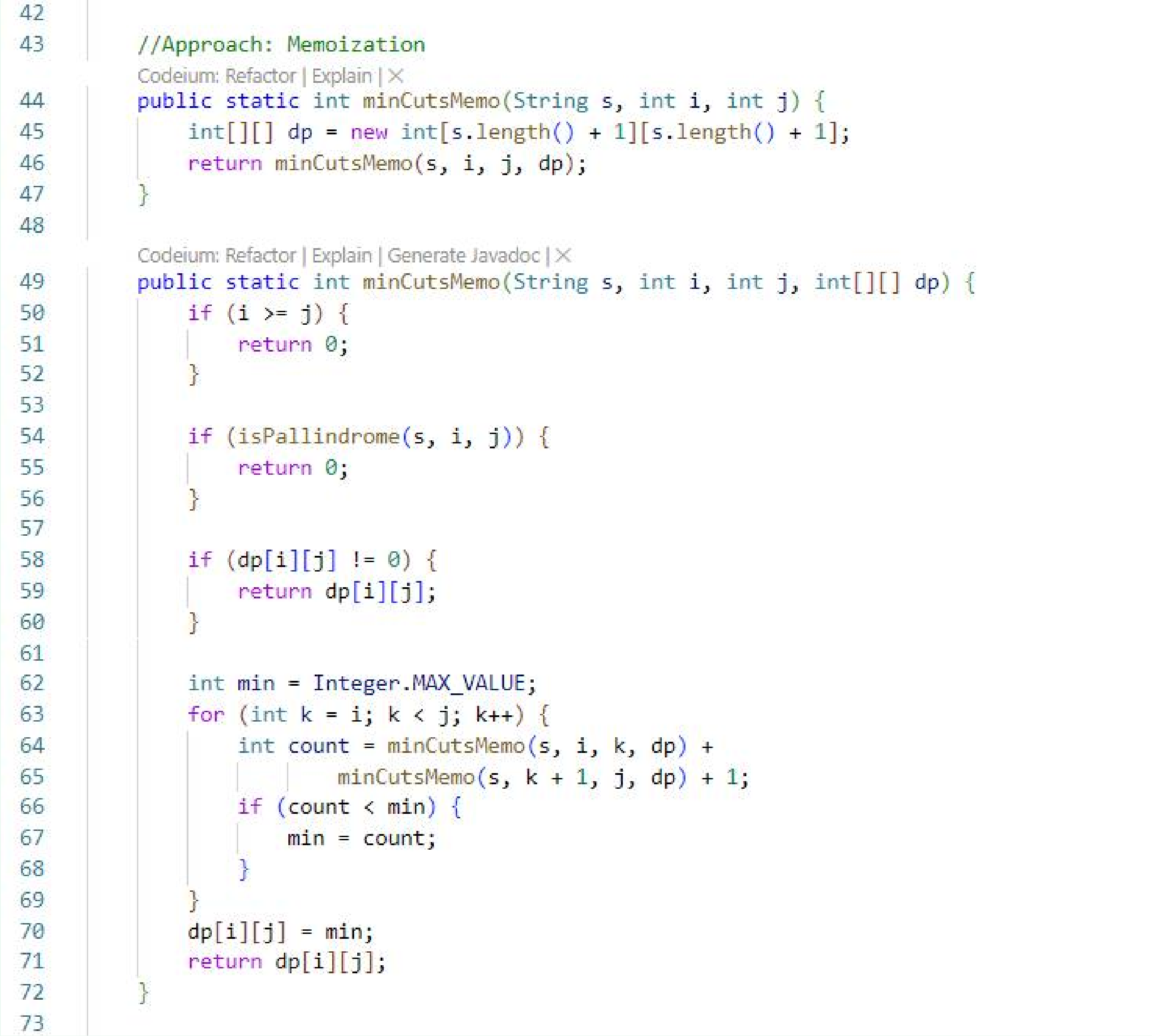
1. Parenthesization: `((A1 * A2) * (A3 * A4))`
   - Multiply `A1` (1x2) and `A2` (2x3): Cost = `1*2*3 = 6`.
   - Multiply `A3` (3x4) and `A4` (4x3): Cost = `3*4*3 = 36`.
   - Multiply the results: `6 + 36 = 42`.

2. Optimal Parenthesization minimizes operations:
    - Final minimum cost = `30`.


Codeium: Refactor | Explain
Codeium: Refactor | Explain | ×
Codeium: Refactor | Explain | Generate Javadoc | ×
Codeium: Refactor | Explain | Generate Javadoc | ×

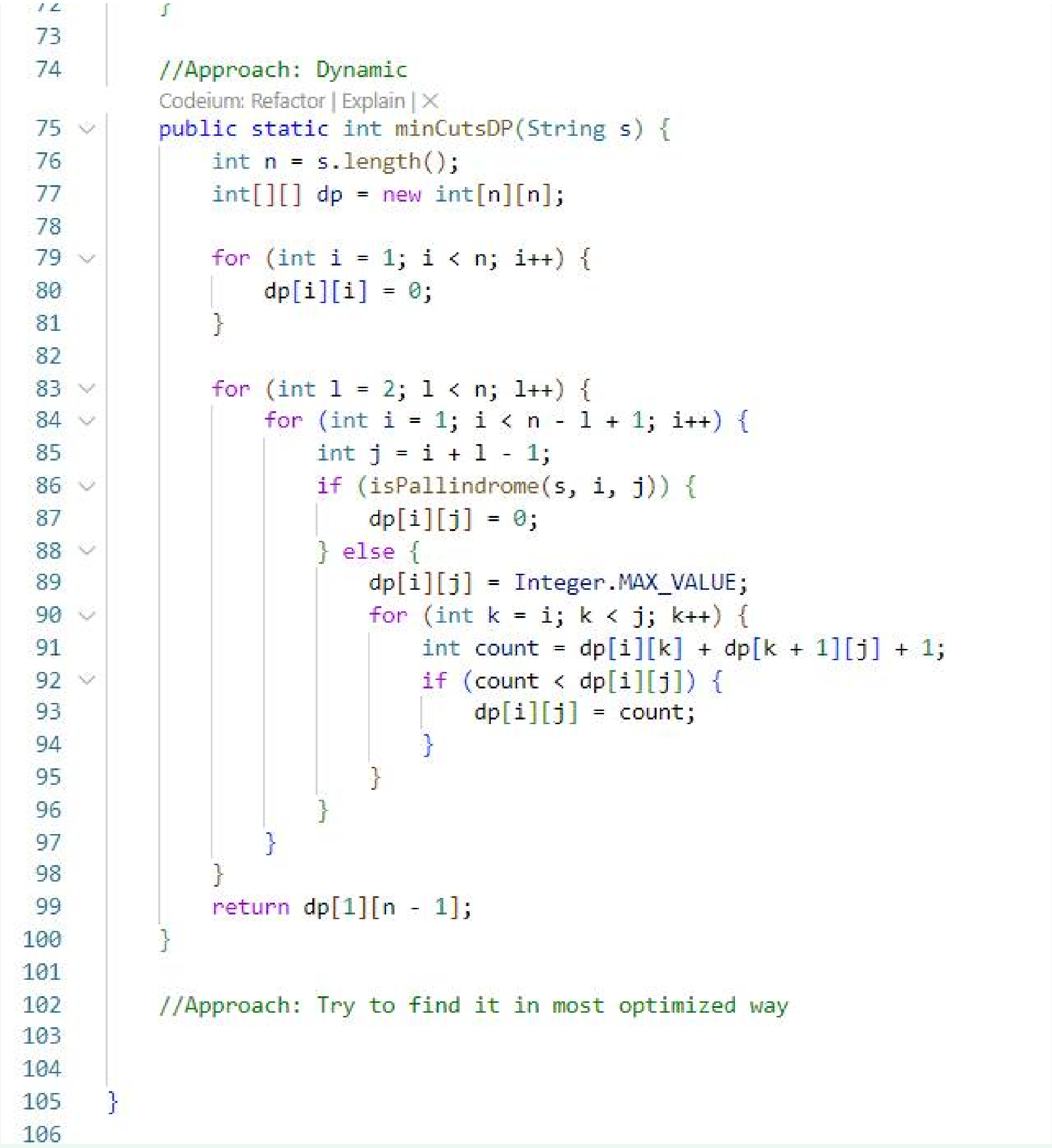
```java
public class PallindromePartition {
    // Given a string, a partitioning of the string is a palindrome partitioning if every substring of the partition is a palindrome.
    // For example, “aba|b|bbabb|a|b|aba” is a palindrome partitioning of “ababbbabbababa”.
    // Determine the fewest cuts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

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

    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;
    }

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

```java
        return true;
    }

```



Codeium: Refactor | Explain | ×
Codeium: Refactor | Explain | Generate Javadoc | ×

```java

    //Approach: Memoization
    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);
    }

    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
    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


}
```

Kindly write the above code from diagram and explain it with example and output

Here is the Java code for the **Palindrome Partitioning Problem** from the diagrams, along with a detailed explanation and example:

Problem Description:

Given a string, partition it such that every substring of the partition is a palindrome. Return the **minimum cuts** required to partition the string.

For example:

- Input: `"ababbababababa"`
- Output: `3`

Code Implementation:

```java
public class PalindromePartition {

    // Recursive Approach
    public static int minCuts(String s, int i, int j) {
        if (i >= j || isPalindrome(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;
    }

    // Helper method to check if a substring is a palindrome
    public static boolean isPalindrome(String s, int i, int j) {
        while (i < j) {
            if (s.charAt(i++) != s.charAt(j--)) {
                return false;
            }
        }
```

```
            return true;
        }

        // Memoization Approach
        public static int minCutsMemo(String s, int i, int j) {
            int[][] dp = new int[s.length()][s.length()];
            return minCutsMemo(s, i, j, dp);
        }

        private static int minCutsMemo(String s, int i, int j, int[][] dp) {
            if (i >= j || isPalindrome(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 min;
        }

        // Dynamic Programming Approach
        public static int minCutsDP(String s) {
            int n = s.length();
            int[][] dp = new int[n][n];

            for (int i = 0; i < n; i++) {
                dp[i][i] = 0; // Single character is a palindrome
            }

            for (int l = 2; l <= n; l++) { // Length of substring
                for (int i = 0; i < n - l + 1; i++) {
                    int j = i + l - 1;
                    if (isPalindrome(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[0][n - 1];
        }

        public static void main(String[] args) {
            String s = "ababbbababa";

            // Recursive Approach
            System.out.println("Recursive: " + minCuts(s, 0, s.length() - 1));

            // Memoization Approach
            System.out.println("Memoization: " + minCutsMemo(s, 0, s.length() - 1));

            // Dynamic Programming Approach
            System.out.println("Dynamic Programming: " + minCutsDP(s));
        }
    }
```

Explanation:

1. **Recursive Approach**:
    - For each substring `(i, j)`, check if it is a palindrome.
    - If yes, no cuts are needed for this substring.
    - Otherwise, try every possible split point `k` to partition the string and calculate the minimum cuts.
2. **Memoization Approach**:
    - Use a 2D array `dp` to store results of overlapping subproblems.
    - Avoid recomputing the results for the same `(i, j)`.
3. **Dynamic Programming Approach**:
    - Use a bottom-up approach.
    - `dp[i][j]` represents the minimum cuts required to partition the substring from index `i` to `j`.

Example:

Input:

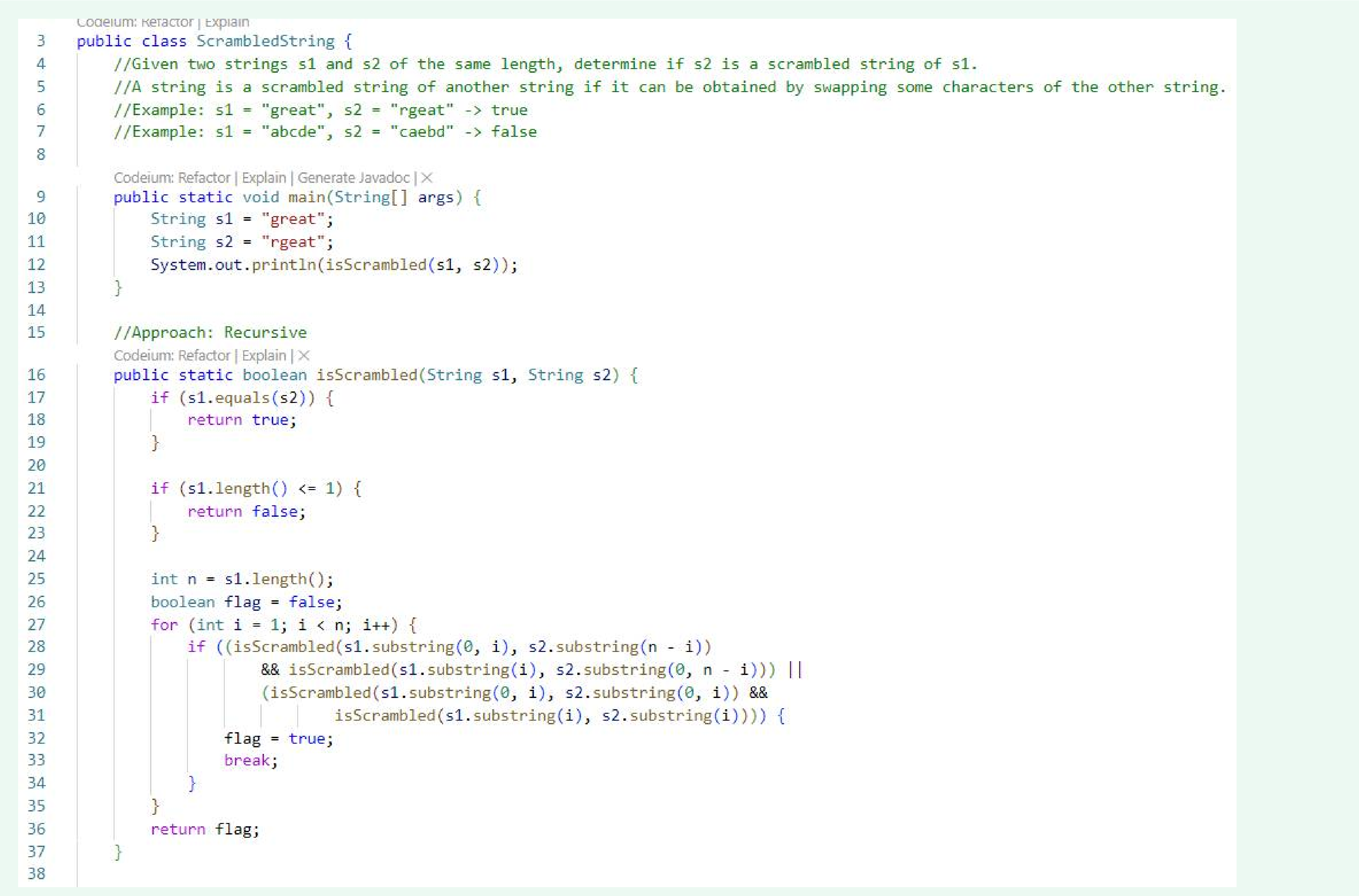
`"ababbbababa"`

Output:

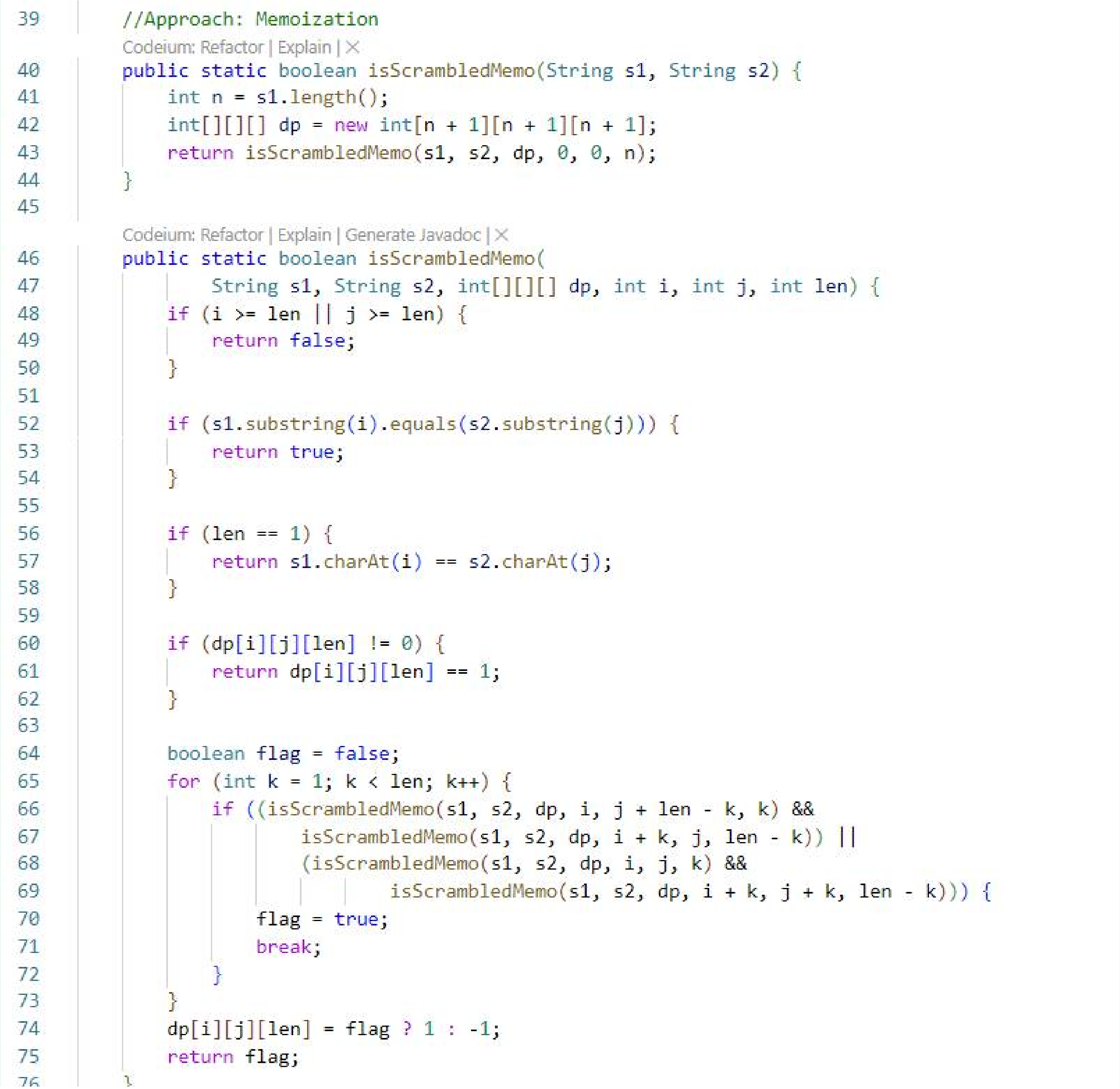
- Recursive: `3`
- Memoization: `3`
- Dynamic Programming: `3`

Explanation of Output:

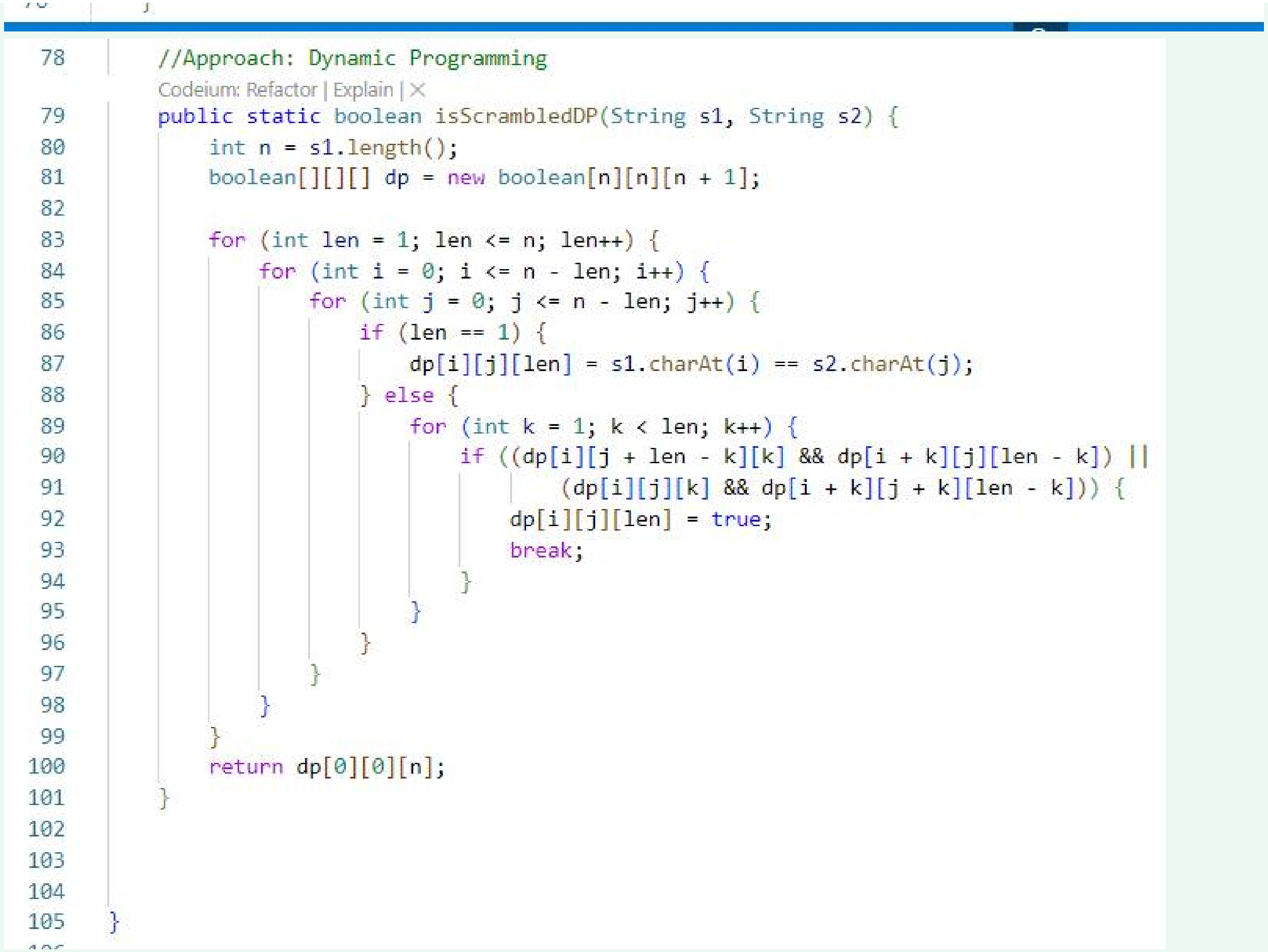
- Partition the string as: `"ababbab | abab | a"`.
- Each segment is a palindrome. The minimum number of cuts required is `3`.


Codeium: Refactor | Explain
Codeium: Refactor | Explain | Generate Javadoc | ×
Codeium: Refactor | Explain | ×

```java
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

    public static void main(String[] args) {
        String s1 = "great";
        String s2 = "rgeat";
        System.out.println(isScrambled(s1, s2));
    }

    //Approach: Recursive
    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;
    }

```


Codeium: Refactor | Explain | ×
Codeium: Refactor | Explain | Generate Javadoc | ×

```
    //Approach: Memoization
    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);
    }

    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
    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];
    }




}
```

Kindly write the above code from diagram and explain it with example and output

Here is the **Scrambled String Problem** implementation from the diagrams, with explanations and examples.

Problem Description:

A string `s2` is a scrambled version of `s1` if it can be obtained by recursively swapping some characters of `s1`.

For example:

- Input: `s1 = "great", s2 = "rgeat"` → **true** (swap "gr" and "eat").
- Input: `s1 = "abcde", s2 = "caebd"` → **false**.

The goal is to determine if `s2` is a scrambled string of `s1`.

Code Implementation:

```java
public class ScrambledString {

    // Recursive Approach
    public static boolean isScrambled(String s1, String s2) {
        if (s1.equals(s2)) {
            return true;
        }

        if (s1.length() != s2.length() || s1.length() <= 1) {
            return false;
        }

        int n = s1.length();
        boolean flag = false;

        for (int i = 1; i < n; i++) {
            // Check both cases:
            // Case 1: No swap
            // Case 2: Swap
            if ((isScrambled(s1.substring(0, i), s2.substring(0, i)) &&
                 isScrambled(s1.substring(i), s2.substring(i))) ||
                (isScrambled(s1.substring(0, i), s2.substring(n - i)) &&
                 isScrambled(s1.substring(i), s2.substring(0, n - i)))) {
                flag = true;
                break;
            }
        }
        return flag;
    }

    // Memoization Approach
    public static boolean isScrambledMemo(String s1, String s2) {
        int n = s1.length();
        int[][][] dp = new int[n + 1][n + 1][n + 1]; // 3D array for memoization
        return isScrambledMemo(s1, s2, dp, 0, 0, n);
    }

    private static boolean isScrambledMemo(String s1, String s2, int[][][] dp, int i, int j, int len) {
        if (len == 1) {
            return s1.charAt(i) == s2.charAt(j);
        }

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

        for (int k = 1; k < len; k++) {
            // Check both cases:
            // Case 1: No swap
            // Case 2: Swap
            if ((isScrambledMemo(s1, s2, dp, i, j, k) &&
                 isScrambledMemo(s1, s2, dp, i + k, j + k, len - k)) ||
                (isScrambledMemo(s1, s2, dp, i, j + len - k, k) &&
                 isScrambledMemo(s1, s2, dp, i + k, j, len - k))) {
                dp[i][j][len] = 1;
                return true;
            }
        }
        dp[i][j][len] = -1;
        return false;
    }

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

        // Initialization for substrings of length 1
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                dp[i][j][1] = s1.charAt(i) == s2.charAt(j);
            }
        }

        // Solve for substrings of length 2 to n
        for (int len = 2; len <= n; len++) {
            for (int i = 0; i <= n - len; i++) {
                for (int j = 0; j <= n - len; j++) {
                    for (int k = 1; k < len; k++) {
                        if ((dp[i][j][k] && dp[i + k][j + k][len - k]) ||
                            (dp[i][j + len - k][k] && dp[i + k][j][len - k])) {
                            dp[i][j][len] = true;
                            break;
                        }
                    }
                }
            }
        }

        return dp[0][0][n];
    }

    public static void main(String[] args) {
        String s1 = "great";
        String s2 = "rgeat";

        // Recursive Approach
        System.out.println("Recursive: " + isScrambled(s1, s2));

        // Memoization Approach
        System.out.println("Memoization: " + isScrambledMemo(s1, s2));

        // Dynamic Programming Approach
        System.out.println("Dynamic Programming: " + isScrambledDP(s1, s2));
    }
}
```

Explanation:

1. **Recursive Approach**:
    - Base case: If `s1.equals(s2)`, return `true`.
    - For each possible split index `k`, check:
        - Case 1: No swap.
        - Case 2: Swap.
    - Return `true` if any split makes `s2` a scrambled string of `s1`.
2. **Memoization Approach**:
    - Use a 3D DP table to store results of subproblems.
    - Avoid recomputation by checking the DP table before recursive calls.
3. **Dynamic Programming Approach**:
    - Bottom-up approach using a 3D DP table.
    - `dp[i][j][len]` indicates if `s1.substring(i, i + len)` is a scrambled string of `s2.substring(j, j + len)`.

Example:

Input:

`s1 = "great"`, `s2 = "rgeat"`

Output:

- Recursive: **true**
- Memoization: **true**
- Dynamic Programming: **true**

Explanation:

1. Split `s1 = "great"` into `["gr", "eat"]`.
2. Swap "gr" and "eat" → `"rgeat"`. This matches `s2`.

The output is `true`.