Please explain the following code so I may use it in an interview.

class B0072 {

    public int **minDistance**(String word1, String word2) {

        int n = word1.**length**(); // length of first string

        int m = word2.**length**(); // length of second string

        // create a 1D array to store intermediate results

        int[] dp = new int[m + 1];

        // initialize the first column with 0,1,2,...,m

        for (int i = 0; i <= m; i++) {

            dp[i] = i;

        }

        for (int i = 1; i <= n; i++) {

            int prev = dp[0]; // store previous value

            dp[0] = i;  // update first column

            for (int j = 1; j <= m; j++) {

                int temp = dp[j];

                if (word1.**charAt**(i - 1) == word2.**charAt**(j - 1)) {

                    dp[j] = prev; // if the characters are equal, no operation is needed

                } else {

                    // calculate minimum edit distance for insertion, deletion and replacement

                    dp[j] = 1 + Math.**min**(Math.**min**(dp[j - 1], dp[j]), prev);

                }

                prev = temp;

            }

        }

        return dp[m]; // return the minimum edit distance

    }

}

This code is a solution to the problem of calculating the minimum number of operations required to convert one string (word1) to another string (word2). The operations that are permitted on a word are: insert a character, delete a character, and replace a character.

The code uses a dynamic programming approach to solve the problem, which is an efficient method for solving problems that have overlapping subproblems. It creates a 1D array (dp) to store the intermediate results, which reduces the memory usage and improves the execution time.

The first step is to initialize the length of the first string (n) and the second string (m). Next, it creates a 1D array (dp) with the length of the second string + 1, and initializes the first column with values 0,1,2,3,...,m. This is done to ensure that the first row of the dp array represents the case where the first string is empty and the second string has i characters, the minimum number of operations required in this case is i.

Then, it starts a nested loop that iterates through the characters of the first string. Inside the inner loop, it iterates through the characters of the second string. For each pair of characters, it checks if they are equal. If they are equal, it assigns the previous value to the current cell in the dp array. If they are not equal, it calculates the minimum edit distance for insertion, deletion, and replacement. It stores the minimum of the three operations in the current cell of the dp array.

Finally, it returns the last element of the dp array, which represents the minimum number of operations required to convert the first string to the second string.

This code is a solution to the problem of calculating the minimum number of operations required to convert one string (word1) to another string (word2). The operations that are permitted on a word are: insert a character, delete a character, and replace a character.

The code uses a dynamic programming approach to solve the problem, which is an efficient method for solving problems that have overlapping subproblems. It creates a 1D array (dp) to store the intermediate results, which reduces the memory usage and improves the execution time.

The first step is to initialize the length of the first string (n) and the second string (m). Next, it creates a 1D array (dp) with the length of the second string + 1, and initializes the first column with values 0,1,2,3,...,m. This is done to ensure that the first row of the dp array represents the case where the first string is empty and the second string has i characters, the minimum number of operations required in this case is i.

Then, it starts a nested loop that iterates through the characters of the first string.

Inside the inner loop, it iterates through the characters of the second string.

For each pair of characters, it checks if they are equal.

If they are equal, it assigns the previous value to the current cell in the dp array.

If they are not equal, it calculates the minimum edit distance for insertion, deletion, and replacement.

It stores the minimum of the three operations in the current cell of the dp array.

Finally, it returns the last element of the dp array, which represents the minimum number of operations required to convert the first string to the second string.

Explain to me this logic. i kind of confuse how this line can calculate for insertion, deletion and replacement Am i missing something?

// calculate minimum edit distance for insertion, deletion and replacement dp[j] = 1 + Math.min(Math.min(dp[j - 1], dp[j]), prev);

Those code were taken from :

class B0072 {

    public int **minDistance**(String word1, String word2) {

        int n = word1.**length**(); // length of first string

        int m = word2.**length**(); // length of second string

        // create a 1D array to store intermediate results

        int[] dp = new int[m + 1];

        // initialize the first column with 0,1,2,...,m

        for (int i = 0; i <= m; i++) {

            dp[i] = i;

        }

        for (int i = 1; i <= n; i++) {

            int prev = dp[0]; // store previous value

            dp[0] = i;  // update first column

            for (int j = 1; j <= m; j++) {

                int temp = dp[j];

                if (word1.**charAt**(i - 1) == word2.**charAt**(j - 1)) {

                    dp[j] = prev; // if the characters are equal, no operation is needed

                } else {

                    // calculate minimum edit distance for insertion, deletion and replacement

                    dp[j] = 1 + Math.**min**(Math.**min**(dp[j - 1], dp[j]), prev);

                }

                prev = temp;

            }

        }

        return dp[m]; // return the minimum edit distance

    }

}

This line of code, dp[j] = 1 + Math.min(Math.min(dp[j - 1], dp[j]), prev);, is calculating the minimum edit distance for insertion, deletion, and replacement using dynamic programming.

Here, dp[j] represents the minimum edit distance between the first i characters of word1 and the first j characters of word2. The current value of dp[j] is being updated with the minimum of the following three values:

1. dp[j - 1] + 1: represents the minimum edit distance for insertion, as it inserts a character in word2 to match the current character of word1.
2. dp[j] + 1: represents the minimum edit distance for deletion, as it deletes a character from word1 to match the current character of word2.
3. prev + 1: represents the minimum edit distance for replacement, as it replaces the current character of word1 with the current character of word2.

prev is the previous value of dp[j], which is stored in a separate variable, before updating it to the current value. This is done to use the previous value of dp[j] in the current calculation.

The final value of dp[m] will be the minimum edit distance between the two strings, word1 and word2.

Below is 2 statements by you

Statement 1 :

The line dp[j] = 1 + Math.min(Math.min(dp[j - 1], dp[j]), prev); is calculating the minimum edit distance between the substrings of word1 and word2. The variable dp[j] represents the minimum edit distance between the substrings ending at index i-1 of word1 and j-1 of word2. The minimum edit distance is calculated by considering the minimum of three possible operations: insertion, deletion, and replacement. The Math.min(Math.min(dp[j - 1], dp[j]), prev) calculates the minimum of the three operations by considering the minimum edit distance of the substrings ending at i and j-1, i-1 and j, and i-1 and j-1. And then adding 1 to it because one operation is done to make the strings equal.

Statement 2 :

This line of code, dp[j] = 1 + Math.min(Math.min(dp[j - 1], dp[j]), prev);, is calculating the minimum edit distance for insertion, deletion, and replacement using dynamic programming. Here, dp[j] represents the minimum edit distance between the first i characters of word1 and the first j characters of word2. The current value of dp[j] is being updated with the minimum of the following three values: 1. dp[j - 1] + 1: represents the minimum edit distance for insertion, as it inserts a character in word2 to match the current character of word1. 2. dp[j] + 1: represents the minimum edit distance for deletion, as it deletes a character from word1 to match the current character of word2. 3. prev + 1: represents the minimum edit distance for replacement, as it replaces the current character of word1 with the current character of word2. prev is the previous value of dp[j], which is stored in a separate variable, before updating it to the current value. This is done to use the previous value of dp[j] in the current calculation. The final value of dp[m] will be the minimum edit distance between the two strings, word1 and word2. My questions is : What is “at i and j-1, i-1 and j, and i-1 and j-1. And then adding 1 to it because one operation is done to make the strings equal.”

How can dp[j] = 1 + Math.min(Math.min(dp[j - 1], dp[j]), prev); same as dp[j - 1] + 1, dp[j] + 1 and prev + 1 ?

Is majority dynamic programming often involving this similar kind of expression, dp[j] = 1 + Math.min(Math.min(dp[j - 1], dp[j]), prev); ?

In the first statement, “at i and j-1, i-1 and j, and i-1 and j-1” refers to the substrings ending at those positions of word1 and word2, respectively. The idea is that the minimum edit distance between two substrings ending at those positions is calculated by considering the minimum of three possible operations: insertion, deletion, and replacement.

Regarding the second question, dp[j] = 1 + Math.min(Math.min(dp[j - 1], dp[j]), prev); represents the minimum edit distance between the first i characters of word1 and the first j characters of word2. The current value of dp[j] is being updated with the minimum of the three possible operations: insertion, deletion and replacement. As dp[j - 1] + 1 represents the minimum edit distance for insertion, dp[j] + 1 represents the minimum edit distance for deletion, prev + 1 represents the minimum edit distance for replacement.

Yes, many dynamic programming problems involve using similar expressions to calculate the minimum or maximum value for a given problem by considering multiple subproblems. This is a common technique used in dynamic programming to break down a problem into smaller subproblems and use the solutions to these subproblems to calculate the solution to the main problem.