JavaScript Edit Distance

Challenge

Given two strings `word1` and `word2`, return the minimum number of operations required to convert `word1` to `word2`.

You have the following three operations permitted on a word:

- `insert` a character.
- `delete` a character.
- `replace` a character.

1st Example

```
Input: word1 = 'horse', word2 = 'ros'
Output: 3
Explanation: horse -> rorse (replace 'h' with 'r')
             rorse -> rose (remove 'r')
             rose -> ros (remove 'e')
```

2nd Example

```
Input: word1 = 'intention', word2 = 'execution'
Output: 5
Explanation: intention -> inention (remove 't')
             inention -> enention (replace 'i' with 'e')
             enention -> exention (replace 'n' with 'x')
             exention -> exection (replace 'n' with 'c')
             exection -> execution (insert 'u')
```

Constraints

- `0 <= word1.length, word2.length <= 500`
- `word1` and `word2` consist of lowercase English letters.

Solution

```
const minDistance = (word1, word2) => {
    const memo = new Map();

    const run = (w1, w2) => {
        let insert  = Infinity,
            del     = Infinity,
            replace = Infinity;

        if (memo.has(`${w1} - ${w2}`)) {
            return memo.get(`${w1} - ${w2}`);
        }

        if (w1 >= word1.length && w2 >= word2.length) {
            return 0;
        }

        if (word1[w1] === word2[w2]) {
            return run(w1 + 1, w2 + 1);
        }

        if (w2 < word2.length) {
            insert = run(w1, w2 + 1);
        }
```

Solution continues on next page...

```
        if (w1 < word1.length) {
            del = run(w1 + 1, w2);
        }

        if (w1 < word1.length && w2 < word2.length) {
            replace = run(w1 + 1, w2 + 1);
        }

        const res = Math.min(insert, del, replace) + 1;

        memo.set(`${w1} - ${w2}`, res);

        return res;
    };

    return run(0, 0);
};
```

Explanation

I've coded a function called `minDistance` that calculates the minimum number of operations required to transform one word into another. It takes two input parameters: `word1` and `word2`.

Inside the function, a `Map` called `memo` is created to store previously computed results.

The function `run` is defined within `minDistance`, which represents the recursive calculation. It takes two parameters: `w1` and `w2`.

Three variables, `insert`, `del`, and `replace`, are initialized with `Infinity`. These variables will store the minimum number of

operations required for each operation type.

The function checks if the result for the current combination of `w1` and `w2` is already present in the `memo` `Map`. If it is, the previously computed result is returned.

If both `w1` and `w2` have reached the end of their respective words (`word1` and `word2`), the function returns `0`, as no further operations are needed.

If the characters at `w1` and `w2` are the same, the function recursively calls `run` with incremented `w1` and `w2` to proceed to the next characters.

If `w2` is less than the length of `word2`, the function recursively calls `run` with the same `w1` and incremented `w2` to simulate an insert operation.

If `w1` is less than the length of `word1`, the function recursively calls `run` with incremented `w1` and the same `w2` to simulate a delete operation.

If both `w1` and `w2` are less than the lengths of `word1` and `word2` respectively, the function recursively calls `run` with incremented `w1` and `w2` to simulate a replace operation.

The minimum number of operations required among the three types (`insert`, `delete`, and `replace`) is calculated by taking the minimum among `insert`, `del`, and `replace` and adding `1` to it.

The result is stored in the `memo` `Map` using the combination of `w1` and `w2` as the key.

Finally, the result is returned as the output of the `minDistance` function.

In summary, the `minDistance` function uses recursion and memoization to calculate the minimum number of operations required to transform one word into another. It considers insertions, deletions, and replacements of characters. The function stores previously computed results to optimize performance.

Author: Trevor Morin