**TIME COMPLEXITY OF LEVENSHTEIN DISTANCE**

**ALGORITHM:**

function LevenshteinDistance(char array str1[1..m], char array str2[1..n])

declare int d[0..m, 0..n]

for i from 0 to m

d[i, 0] = i

for j from 0 to n

d[0, j] = j

for j from 1 to n

for i from 1 to m

if str1[i] = str2[j] then cost := 0

else cost := 1

d[i, j] := minimum(

d[i-1, j] + 1,

d[i, j-1] + 1,

d[i-1, j-1] + cost

)

return d[m, n]

**Time Complexity:**

The time complexity of the Levenshtein distance algorithm is **O(mn)**, where **m** and **n** are the lengths of the two input strings.

**Here’s why:**

The algorithm uses a 2D array **d[m+1][n+1]** to store the Levenshtein distances between prefixes of the input strings. It iterates over this array in a nested loop: the outer loop runs **n+1** times and the inner loop runs **m+1** times.

In each iteration of the inner loop, it performs a constant amount of work: it calculates the minimum of three values (which takes constant time) and stores the result in the array.

Therefore, the total amount of work done is proportional to the number of cells in the array, which is **(m+1)(n+1)**. Ignoring the “+1” (since we usually ignore lower-order terms when calculating time complexity), we get a time complexity of **O(mn)**.

This means that the running time of the algorithm grows linearly with the size of the input strings, making it quite efficient for computing the Levenshtein distance between two strings. However, keep in mind that it can still be slow if the strings are very long, because it has to fill in a large 2D array.

**Space complexity:**

The space complexity of the Levenshtein distance algorithm is **O(mn)**.

**Here’s why:**

The algorithm uses a 2D array **d[m+1][n+1]** to store the Levenshtein distances between prefixes of the input strings. Each cell in this array stores a single integer, so the amount of space used is proportional to the number of cells in the array.

Since the array has **(m+1)** rows and **(n+1)** columns, the total number of cells is **(m+1)(n+1)**. Ignoring the “+1” (as we usually do when calculating space complexity), we get a space complexity of **O(mn)**.

This means that the amount of memory used by the algorithm grows linearly with the size of the input strings.