Universidad Nacional de Colombia
Departamento de Matemáticas
Topics on Advanced Algorithms
Edit Ds (2025-II)

Juliana Alejandra Nieto Cárdenas

El problema de *Edit Distance* de LeetCode, consiste en determinar el número mínimo de operaciones necesarias para transformar una palabra en otra, permitiendo tres tipos de operaciones: inserción, eliminación y reemplazo de caracteres. Por ejemplo, para convertir la palabra `horse` en `ros`, se requieren tres operaciones:

$$\texttt{horse} \to \texttt{rorse} \to \texttt{rose} \to \texttt{ros}$$

Idea principal

Comparar carácter por carácter dos palabras. Si los caracteres actuales son iguales, avanzamos en ambos sin costo. Si difieren, consideramos tres operaciones (*insertar*, *eliminar*, *reemplazar*) y elegimos la que deja el costo total mínimo.

Paso 1: Definir el estado del DP

Sea $dp[i][j]$ el costo mínimo para convertir el sufijo `word1[i:]` en el sufijo `word2[j:]`. Ese estado captura "dónde voy" en cada palabra.

Paso 2: Casos base (bordes de la matriz)

Si una palabra se agotó, sólo quedan inserciones o eliminaciones:

$$dp[i][m] = n-i \quad \text{(eliminar lo que queda en } \texttt{word1}\text{)}, \qquad dp[n][j] = m-j \quad \text{(insertar lo que falta de } \texttt{word2}\text{)}.$$

Paso 3: Transición cuando los caracteres coinciden

Si `word1[i] == word2[j]`, no pagamos costo aquí y avanzamos en ambos:

$$dp[i][j] = dp[i+1][j+1].$$

Paso 4: Transición cuando los caracteres difieren

Si `word1[i] != word2[j]`, probamos las tres operaciones sobre el carácter actual y tomamos el mínimo total:

$$\begin{array}{rll} \textbf{Insertar:} & \text{agregar el carácter } \texttt{word2[j]} & \Rightarrow\ 1 + dp[i][j+1] \\ \textbf{Eliminar:} & \text{quitar el carácter } \texttt{word1[i]} & \Rightarrow\ 1 + dp[i+1][j] \\ \textbf{Reemplazar:} & \text{sustituir } \texttt{word1[i]} \text{ por } \texttt{word2[j]} & \Rightarrow\ 1 + dp[i+1][j+1] \end{array}$$

$$\Rightarrow \quad dp[i][j] = 1 + \min\Big(dp[i][j+1],\ dp[i+1][j],\ dp[i+1][j+1]\Big)$$

Paso 5: Orden de llenado (bottom-up)

Llenamos *dp* desde la esquina inferior-derecha hacia arriba e izquierda, garantizando que los subproblemas necesarios estén listos al usar cada transición. El resultado final queda en $dp[0][0]$.

Ejemplo: `horse` vs `ros`

Matriz *dp* de tamaño 6×4 (incluye cadenas vacías). Filas: sufijos de `horse`; columnas: sufijos de `ros`. Los bordes reflejan los casos base.

`word1\word2`	r	o	s	
h	**3**	3	4	5
o	3	**2**	3	4
r	**2**	3	2	3
s	3	2	**1**	2
e	4	3	2	1
	3	2	1	0

Interpretación: $dp[0][0] = 3$ (reemplazar $h \to r$, eliminar r, eliminar e):

`horse` → `rorse` → `rose` → `ros`.

Por qué esto es Programación Dinámica

Subestructura óptima: cada estado (i, j) se resuelve a partir de soluciones óptimas de subestados $(i+1, j)$, $(i, j+1)$ y $(i+1, j+1)$ (o sólo $(i+1, j+1)$ si coinciden los caracteres).
Subproblemas superpuestos: las mismas parejas (i, j) reaparecen en un enfoque recursivo ingenuo; memorizarlas en la tabla evita recomputar.

Complejidad

Tiempo $O(nm)$ y espacio $O(nm)$ (puede reducirse a $O(m)$ manteniendo solo la fila actual y la siguiente).

Código en C++ (implementación bottom-up)

```cpp
class Solution {
public:
    int minDistance(string word1, string word2) {
        int n = word1.size();
        int m = word2.size();
        vector<vector<int>> matrix(n+1, vector<int>(m+1, 0));
        // llenar casos base
        for(int j = 0; j <= m; j++) matrix[n][j] = m - j;
        for(int i = 0; i <= n; i++) matrix[i][m] = n - i;

        // programaci n din mica "bottom - up"
        for(int i = n - 1; i >= 0; i--){
            for(int j = m - 1; j >= 0; j--){
                if (word1[i] == word2[j]){
                    matrix[i][j] = matrix[i+1][j+1];
                } else {
                    int insert_op = matrix[i][j+1];   // inserto word2[j]
                    int delete_op = matrix[i+1][j];   // elimino word1[i]
                    int replac_op = matrix[i+1][j+1]; // reemplazo word1[i] por word2[j]
                    matrix[i][j] = 1 + std::min({insert_op, delete_op, replac_op});
                }
            }
        }
        return matrix[0][0];
    }
};
```