



## **GUÍA DE APRENDIZAJE PARA CONCURSANTES ICPC Y IOI: DISTANCIA DE EDICIÓN (*EDIT DISTANCE*)**

---



## 1. Introducción

La distancia de edición o distancia de Levenshtein es el número mínimo de operaciones de edición necesarias para transformar una cadena en otra cadena. Las operaciones de edición permitidas son las siguientes:

- Insertar un caracter(  $ABC \rightarrow ABCA$  )
- Remover un caracter (  $ABC \rightarrow AB$  )
- Modificar un caracter (  $ABC \rightarrow ADC$  )

Por ejemplo, la distancia de edición entre LOVE y MOVIE es 2, porque primero podemos realizar la operación  $LOVE \rightarrow MOVE$  (modificar) y luego la operación  $MOVE \rightarrow MOVIE$  (insertar). Este es el menor número posible de operaciones, porque está claro que una sola operación no es suficiente.

## 2. Conocimientos previos

### 2.1. Programación dinámica

La programación dinámica es una técnica de optimización que se utiliza para resolver problemas complejos dividiéndolos en subproblemas más pequeños y resolviéndolos de manera recursiva. La idea principal detrás de la programación dinámica es almacenar los resultados de los subproblemas resueltos para evitar repetir los cálculos y mejorar la eficiencia del algoritmo.

### 2.2. Principio de óptimo

Enunciado por Bellman en 1957 y que dice *En una secuencia de decisiones óptima toda subsecuencia ha de ser también óptima*. Hemos de observar que aunque este principio parece evidente no siempre es aplicable y por tanto es necesario verificar que se cumple para el problema en cuestión

## 3. Desarrollo

Supongamos que tenemos una cadena  $x$  de longitud  $n$  y una cadena  $y$  de longitud  $m$ , y queremos calcular la distancia de edición entre  $x$  e  $y$ .

Para resolver el problema utilizando programación dinámica es necesario plantearlo como una sucesión de decisiones que satisfaga el principio de óptimo. Para plantearla, vamos a fijarnos en el último elemento de cada una de las cadenas. Si los dos son iguales, entonces tendremos que calcular el número de operaciones básicas necesarias para obtener de la primera cadena menos el último elemento, y la segunda cadena también sin el último elemento, es decir:

$$distance(n, m) = distance(n - 1, m - 1) \quad \text{si} \quad x_n = y_m$$

Pero si los últimos elementos fueran distintos habría que escoger la situación más beneficiosa de entre tres posibles: (i) considerar la primera cadena y la segunda pero sin el último elemento, o bien (ii) la primera cadena menos el último elemento y la segunda cadena, o bien (iii) las dos cadenas sin el último elemento. Esto da lugar a la siguiente relación en recurrencia para  $distance(n, m)$  para este caso:

$$distance(n, m) = 1 + \min\{distance(n-1, m), distance(n, m-1), distance(n-1, m-1)\} \quad \text{si } n \neq 0, m \neq 0, x_n \neq y_m$$

En cuanto a las condiciones iniciales, tenemos las tres siguientes:

$$distance(0, 0) = 0, \quad distance(n, 0) = n, \quad distance(0, m) = m,$$

Una vez disponemos de la ecuación en recurrencia necesitamos resolverla utilizando alguna estructura que nos permita reutilizar resultados intermedios.

Para resolver el problema, definimos una función  $distance(a, b)$  que proporciona la distancia de edición entre prefijos  $x[0 \dots a]$  y  $y[0 \dots b]$ . Por lo tanto, al usar esta función, la distancia de edición entre  $x$  e  $y$  es igual a la distancia  $(n-1, m-1)$ .

Podemos calcular los valores de distancia de la siguiente manera:

$$distance(a, b) = \min \begin{cases} distance(a, b-1) + 1, \\ distance(a-1, b) + 1, \\ distance(a-1, b-1) + cost(a, b) \end{cases}$$

Aquí  $cost(a, b) = 0$  si  $x[a] = y[b]$ , y en caso contrario  $cost(a, b) = 1$ . La fórmula considera las siguientes formas de editar la cadena  $x$ :

- $distance(a, b-1)$ : inserta un carácter al final de  $x$ .
- $distance(a-1, b)$ : remueve el último carácter de  $x$ .
- $distance(a-1, b-1)$ : coincide o modifica el último carácter de  $x$ .

En los dos primeros casos, es necesaria una operación de edición (insertar o eliminar). En el último caso, si  $x[a] = y[b]$ , podemos hacer coincidir los últimos caracteres sin editar; de lo contrario, se necesita una operación de edición (modificar).

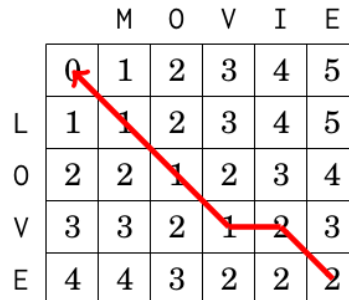
La siguiente tabla muestra los valores de distancia en el caso de ejemplo:

La esquina inferior derecha de la tabla nos dice que la distancia de edición entre LOVE y MOVIE es 2. La tabla también muestra cómo construir la secuencia más corta de operaciones de edición. En este caso el camino es el siguiente:

Los últimos caracteres de LOVE y MOVIE son iguales, por lo que la distancia de edición entre ellos es igual a la distancia de edición entre LOV y MOVI. Podemos usar una operación de edición para eliminar el carácter I de MOVI. Por lo tanto, la distancia de edición es uno mayor que la distancia de edición entre LOV y MOV, etc.

	M O V I E					
	0	1	2	3	4	5
L	1	1	2	3	4	5
O	2	2	1	2	3	4
V	3	3	2	1	2	3
E	4	4	3	2	2	2

	M O V I E					
	0	1	2	3	4	5
L	1	1	2	3	4	5
O	2	2	1	2	3	4
V	3	3	2	1	2	3
E	4	4	3	2	2	2



## 4. Implementación

### 4.1. C++

```

unsigned int edit_distance(string s1, string s2) {
    const size_t len1 = s1.size(), len2 = s2.size();
    vector< vector<unsigned int> > d(len1+1, vector <unsigned int>(len2 + 1));
    d[0][0] = 0;
    for (unsigned int i = 1; i <= len1; ++i) d[i][0] = i;
    for (unsigned int i = 1; i <= len2; ++i) d[0][i] = i;

    for (unsigned int i = 1; i <= len1; ++i)
        for (unsigned int j = 1; j <= len2; ++j)
            d[i][j] = min(min(d[i-1][j]+1,d[i][j-1]+1),
                           d[i-1][j-1]+(s1[i-1]== s2[j-1] ? 0 : 1));
    return d[len1][len2];
}

```

### 4.2. Java

```

public static int edit_distance(String s1, String s2) {
    int[][] dp = new int[s1.length()+1][s2.length()+1];
    for (int i = 0; i <= s1.length(); i++) dp[i][0] = i;
    for (int j = 0; j <= s2.length(); j++) dp[0][j] = j;

    for (int i = 1; i <= s1.length(); i++) {

```



```
for (int j = 1; j <= s2.length(); j++) {
    if (s1.charAt(i-1) == s2.charAt(j-1)) dp[i][j] = dp[i-1][j-1];
    else dp[i][j] = 1 + Math.min( Math.min( dp[i-1][j], dp[i][j-1]),
                                   dp[i-1][j-1]);
}
}
return dp[s1.length()][s2.length()];
}
```

Nótese que  $d[i-1][j] + 1$  representa un costo de 1 para la inserción,  $d[i][j-1] + 1$  costo 1 para eliminación, y  $d[i-1][j-1] + (s1[i-1] == s2[j-1]?0:1)$  representa costo 1 para reemplazo (en caso de que no sean iguales). Con estas consideraciones es fácil adaptar este problema a otros similares.

## 5. Aplicaciones

El cálculo de la distancia de edición, también conocida como distancia de Levenshtein, tiene diversas aplicaciones en diferentes campos. Algunas de estas aplicaciones incluyen:

1. **Corrección ortográfica:** La distancia de Levenshtein se utiliza para sugerir correcciones para palabras mal escritas. Se compara la palabra ingresada con las palabras en un diccionario y se sugiere la palabra más cercana en términos de distancia de edición.
2. **Búsqueda de similitud:** La distancia de edición se utiliza para encontrar la similitud entre dos cadenas de texto. Esto puede ser útil en la clasificación o agrupación de documentos basados en su similitud textual.
3. **Algoritmos de autocompletar:** La distancia de Levenshtein se puede utilizar para implementar algoritmos de autocompletar en campos como los motores de búsqueda o los sistemas de entrada de texto predictivo en dispositivos móviles.
4. **Comparación de secuencias genéticas:** La distancia de edición se utiliza en bioinformática para comparar secuencias genéticas y determinar su similitud o diferencia. Esto es útil en el estudio de la evolución genética y la identificación de relaciones entre diferentes especies.
5. **Corrección gramatical:** La distancia de Levenshtein se puede utilizar para detectar errores gramaticales en oraciones y sugerir correcciones gramaticales adecuadas.

## 6. Complejidad

El procedimiento descrito va a permitir la creación de la tabla que calcula el número mínimo de operaciones básicas para la distancia de edición. La solución se encuentra en  $[n, m]$ , y la tabla se construye fila a fila (a partir de los valores que definen las condiciones iniciales) para poder ir reutilizando los valores calculados previamente. Como el algoritmo se limita a dos bucles anidados que sólo incluyen operaciones constantes la complejidad de este algoritmo es de orden  $O(n \times m)$  con similar complejidad espacial. Siendo  $n$  y  $m$  la longitud de las cadenas iniciales.



## 7. Ejercicios

A continuación una lista de ejercicios que se resuelven aplicando este algoritmo

- [CSES - Edit Distance](#)