


Edit Distance (ED) - Distância de Edição

O algoritmo de distância de edição (ou distância de Levenshtein) mede o quão diferentes são duas sequências, geralmente strings, com base no número mínimo de operações necessárias para transformar uma sequência na outra.



Essas operações básicas incluem:

➤ Inserção de um caractere.

➤ Remoção de um caractere.

➤ Substituição de um caractere por outro.

Por exemplo, a distância de edição entre as palavras "gato" e "pato" é 1, pois basta uma substituição (trocar o "g" por "p") para transformar uma na outra.

Exemplos:

FLOR --> FLORES ----> 2 operações

LIVRO --> LIVRARIA ----> 4 operações

XZW --> ABC XZW ----> 4 operações

XZW --> " " ----> ? operações

Aplicações para o algoritmo Distância de Edição:

- . Verificação ortográfica e correção automática
- . Alinhamento de sequência de DNA
- . Detecção de plágio
- . Processamento de linguagem natural
 - .. Busca por similaridade entre palavras
 - .. Análise de afinidade entre frases.
- . Sistemas de controle de versão
- . Correspondência de strings

S1

CASA

S2

CARRO

ETAPAS

1.

IGUAL
A S2

2.

IGUAL
A S2

3.

DIFERENTE
DE S2

4.

Substituir o caractere
's' pelo caractere 'r':
"casa" "cara"

5.

Substituir o caractere
'r' após o último 'a':
"cara" "carr"

6.

Inserir o caractere 'o' no final:
"carr" "carro"

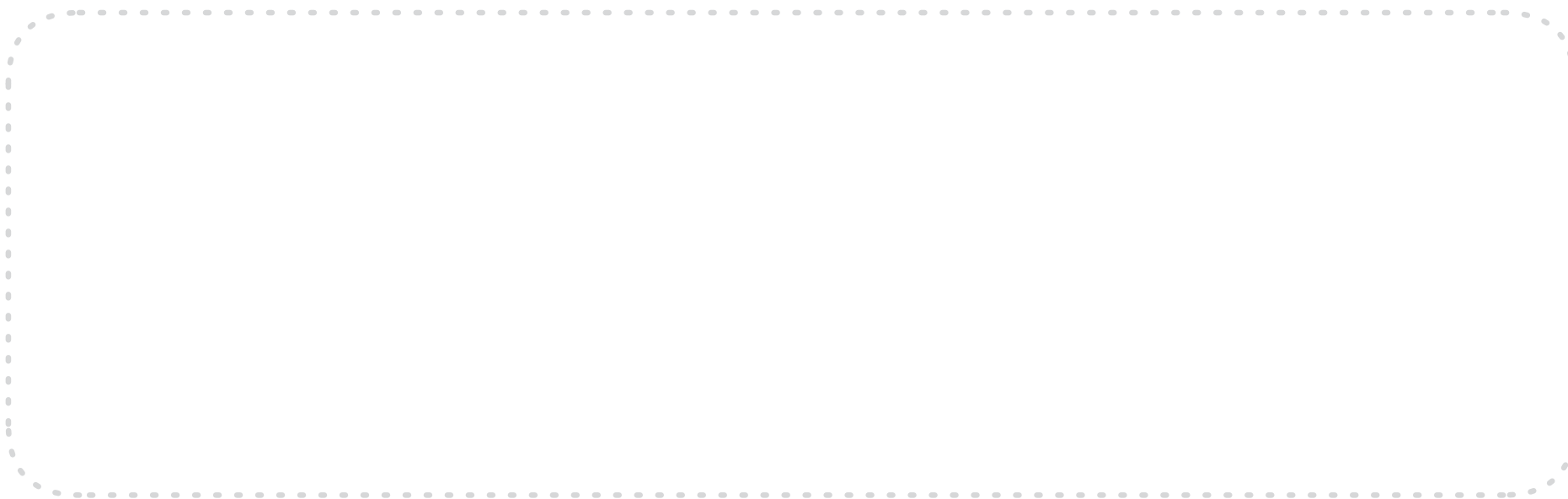
Quantas operações foram
necessárias para transformar
s1 em s2?

Entendemos o problema. Como resolvê-lo?

1. Força bruta
2. Recursão
3. Programação Dinâmica
 - 3a. Memoização + Recursão
 - 3b. Memoização + Bottom-Up (Iterativa)
 - 3c. Memoização + BU com espaço otimizado

Recusão:

1. Quais são os casos de base?



Recusão:

1. Quais são os casos de base?

Caso 1: Quando $s1$ se torna vazio, ou seja, $m=0$ retorna n , pois requer n inserções para converter uma string vazia em $s2$ de tamanho n

Caso 2: Quando $s2$ se torna vazio, ou seja, $n=0$ retorna m , pois requer m remoções para converter $s1$ de tamanho m em uma string vazia.

Recusão:

2. Quais são os dois casos que devem ser tratados?

Recusão:

2. Quais são os dois casos que devem ser tratados?

Caso 1: Quando o último caractere de ambas as strings são iguais:

$$\text{editDist}(s1, s2, m, n) = \text{editDist}(s1, s2, m-1, n-1)$$

Caso 2: Quando os últimos caracteres são diferentes:

$$\begin{aligned} \text{editDist}(s1, s2, m, n) = 1 + \text{Minimum} \{ & \text{editDist}(s1, s2, m, n-1), // \text{ inserir} \\ & \text{editDist}(s1, s2, m-1, n), // \text{ remover} \\ & \text{editDist}(s1, s2, m-1, n-1) \} // \text{ substituir} \end{aligned}$$

LeetCode: 72. Edit Distance

[Description](#) | [Editorial](#) | [Solutions](#) | [Submissions](#)

72. Edit Distance

Medium

Topics

Companies

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

Example 1:

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

```

1 class Solution {
2     private Map<String, Integer> memo = new HashMap<>();
3
4     public int minDistance(String word1, String word2) {
5         return minDistanceHelper(word1, word2, word1.length(), word2.length());
6     }
7
8     private int minDistanceHelper(String word1, String word2, int i, int j) {
9         // Caso base: transformar uma string vazia em outra
10        if (i == 0) return j;
11        if (j == 0) return i;
12
13        // Cria uma chave única para as coordenadas atuais
14        String key = i + "," + j;
15
16        // Verifica se o valor já foi calculado
17        if (memo.containsKey(key)) {
18            return memo.get(key);
19        }
20
21        int result;
22
23        // Se os caracteres são iguais, move para a próxima posição sem custo adicional
24        if (word1.charAt(i - 1) == word2.charAt(j - 1)) {
25            result = minDistanceHelper(word1, word2, i - 1, j - 1);
26        } else {
27            // Calcula o mínimo entre as três operações possíveis
28            int insert = minDistanceHelper(word1, word2, i, j - 1);
29            int delete = minDistanceHelper(word1, word2, i - 1, j);
30            int replace = minDistanceHelper(word1, word2, i - 1, j - 1);
31            result = 1 + Math.min(insert, Math.min(delete, replace));
32        }
33
34        // Armazena o resultado na tabela de memoização e retorna
35        memo.put(key, result);
36        return result;
37    }
38 }
39

```