# Levenshtein distance

## From Wikipedia, the free encyclopedia

In information theory and computer science, the **Levenshtein distance** or **edit distance** between two strings is given by the minimum number of operations needed to transform one string into the other, where an operation is an insertion, deletion, or substitution of a single character. It is named after Vladimir Levenshtein, who considered this distance in 1965. It is useful in applications that need to determine how similar two strings are, such as spell checkers.

For example, the Levenshtein distance between "kitten" and "sitting" is 3, since these three edits change one into the other, and there is no way to do it with fewer than three edits:

1. kitten → sitten (substitution of 'k' for 's')
2. sitten → sittin (substitution of 'e' for 'i')
3. sittin → sitting (insert 'g' at the end)

It can be considered a generalization of the Hamming distance, which is used for strings of the same length and only considers substitution edits. There are also further generalizations of the Levenshtein distance that consider, for example, exchanging two characters as an operation, like in the Damerau-Levenshtein distance algorithm.

## The algorithm

A commonly-used bottom-up dynamic programming algorithm for computing the Levenshtein distance involves the use of an $(n + 1) \times (m + 1)$ matrix, where $n$ and $m$ are the lengths of the two strings. Here is pseudo code for a function *LevenshteinDistance* that takes two strings, *str1* of length *lenStr1*, and *str2* of length *lenStr2*, and computes the Levenshtein distance between them:

```
int LevenshteinDistance(char str1[1..lenStr1], char str2[1..lenStr2])
   // d is a table with lenStr1+1 rows and lenStr2+1 columns
   declare int d[0..lenStr1, 0..lenStr2]
   // i and j are used to iterate over str1 and str2
   declare int i, j, cost

   for i from 0 to lenStr1
       d[i, 0] := i
   for j from 0 to lenStr2
       d[0, j] := j

   for i from 1 to lenStr1
       for j from 1 to lenStr2
           if str1[i] = str2[j] then cost := 0
                                else cost := 1
           d[i, j] := minimum(
                                d[i-1, j  ] + 1,     // deletion
                                d[i  , j-1] + 1,     // insertion
                                d[i-1, j-1] + cost   // substitution
                            )

   return d[lenStr1, lenStr2]
```

Two examples of the resulting matrix (the minimum steps to be taken are highlighted):

| | | **k** | **i** | **t** | **t** | **e** | **n** |
|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| **s** | 1 | 1 | 2 | 3 | 4 | 5 | 6 |
| **i** | 2 | 2 | 1 | 2 | 3 | 4 | 5 |
| **t** | 3 | 3 | 2 | 1 | 2 | 3 | 4 |
| **t** | 4 | 4 | 3 | 2 | 1 | 2 | 3 |
| **i** | 5 | 5 | 4 | 3 | 2 | 2 | 3 |
| **n** | 6 | 6 | 5 | 4 | 3 | 3 | 2 |
| **g** | 7 | 7 | 6 | 5 | 4 | 4 | 3 |

| | | **S** | **a** | **t** | **u** | **r** | **d** | **a** | **y** |
|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| **S** | 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| **u** | 2 | 1 | 1 | 2 | 2 | 3 | 4 | 5 | 6 |
| **n** | 3 | 2 | 2 | 2 | 3 | 3 | 4 | 5 | 6 |
| **d** | 4 | 3 | 3 | 3 | 3 | 4 | 3 | 4 | 5 |
| **a** | 5 | 4 | 4 | 4 | 4 | 4 | 4 | 3 | 4 |
| **y** | 6 | 5 | 5 | 5 | 5 | 5 | 5 | 4 | 3 |

The invariant maintained throughout the algorithm is that we can transform the initial segment str1[1..i] into str2[1..j] using a minimum of d[i,j] operations. At the end, the bottom-right element of the array contains the answer.