# The Levenshtein distance

Vikaskomera
5 min read · Just now



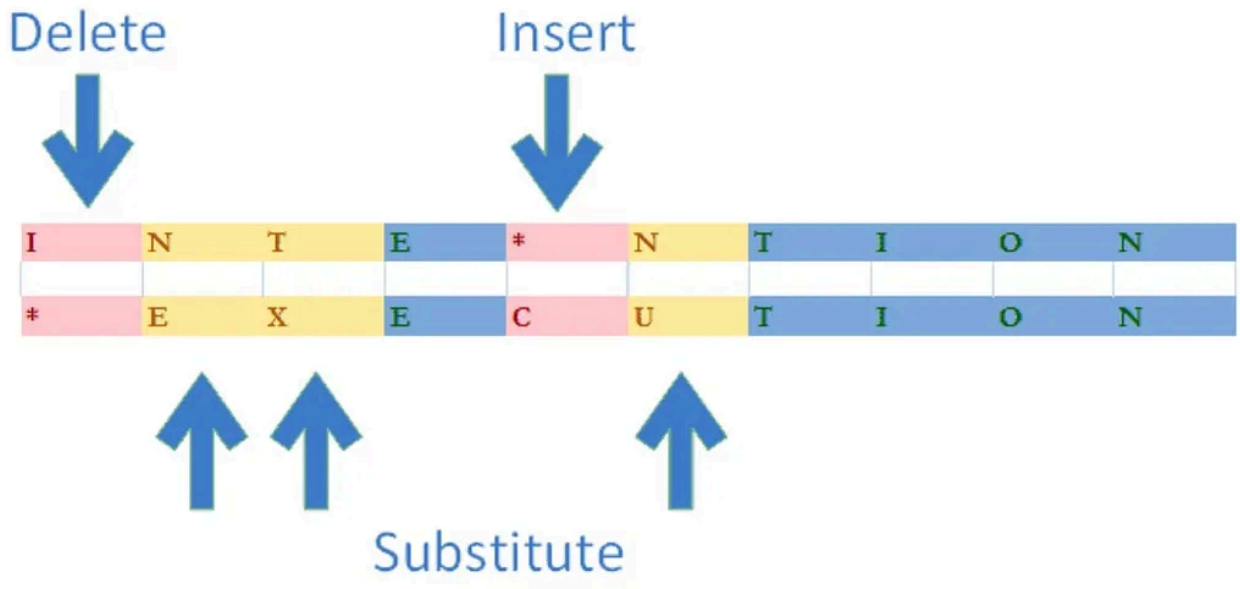
## Introduction:

The Levenshtein distance is a string metric for measuring difference between two sequences. Informally, the Levenshtein distance between two words is the minimum number of single-character edits (i.e. insertions, deletions or substitutions) required to change one word into the other. It is named after Vladimir Levenshtein, who considered this distance in 1965.

Levenshtein distance may also be referred to as edit distance, although it may also denote a larger family of distance metrics. It is closely related to pairwise string alignments.

## Problem:

Informally, the Levenshtein distance between two words is the minimum number of single-character edits (insertions, deletions, or substitutions) required to change one word into the other.




## Definition:

Mathematically, the Levenshtein distance between two strings a, b (of length |a| and |b| respectively) is given by leva,b(|a|,|b|) where:

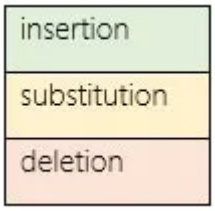
$$\operatorname{lev}_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if } \min(i,j) = 0, \\ \min \begin{cases} \operatorname{lev}_{a,b}(i-1,j) + 1 \\ \operatorname{lev}_{a,b}(i,j-1) + 1 \\ \operatorname{lev}_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)} \end{cases} & \text{otherwise.} \end{cases}$$

where 1(ai≠bi) is the indicator function equal to 0 when ai≠bi and equal to 1 otherwise, and leva, b(i,j) is the distance between the first i characters of a and the first j characters of b.

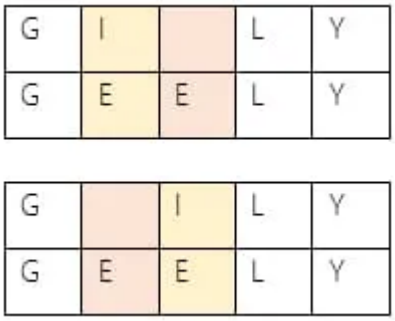
Note that the first element in the minimum corresponds to deletion (from a to b), the second to insertion and the third to match or mismatch, depending on whether the respective symbols are the same.

## Example

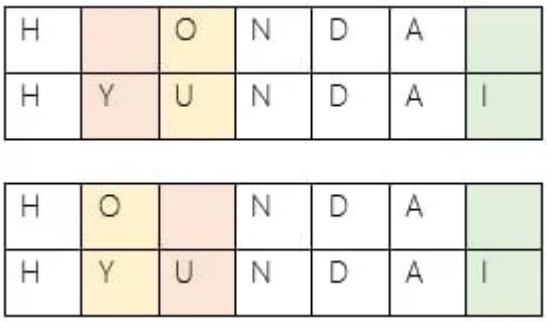
The Levenshtein distance between “FLOMAX” and “VOLMAX” is 3, since the following three edits change one into the other, and there is no way to do it with fewer than three edits:




Levenshtein distance between “GILY” and “GEELY” is 2.




*Levenshtein distance between “HONDA” and “HYUNDAI” is 3.*




## Dynamic Programming Approach:

The Levenshtein algorithm calculates the least number of edit operations that are necessary to modify one string to obtain another string. The most common way of calculating this is by the dynamic programming approach:

1. A matrix is initialized measuring in the (m, n) cell the Levenshtein distance between the m-character prefix of one with the n-prefix of the other word.
2. The matrix can be filled from the upper left to the lower right corner.
3. Each jump horizontally or vertically corresponds to an insert or a delete, respectively.
4. The cost is normally set to 1 for each of the operations.

5. The diagonal jump can cost either one, if the two characters in the row and column do not match else 0, if they match. Each cell always minimizes the cost locally.

6. This way the number in the lower right corner is the Levenshtein distance between both words.

## Algorithm:

**Algorithm 2:** Levenshtein Distance - Iterative with Full Matrix

```
Function LevDist (a, b) is
    n ← a.size();
    m ← b.size();
    distance[i, 0] ← i ∀ i ∈ 0..n;
    distance[0, j] ← j ∀ j ∈ 0..m;
    for j ← 1 to m do
        for i ← 1 to n do
            indicator ← a[i] ≠ b[j] ? 1 : 0;
            distance[i, j] ← Minimum(
                    distance[i − 1, j] + 1,                  // deletion
                    distance[i, j − 1] + 1,                  // insertion
                    distance[i − 1, j − 1] + indicator // substitution
                    );
        end
    end
    return distance[n, m];
end
```

## Code:

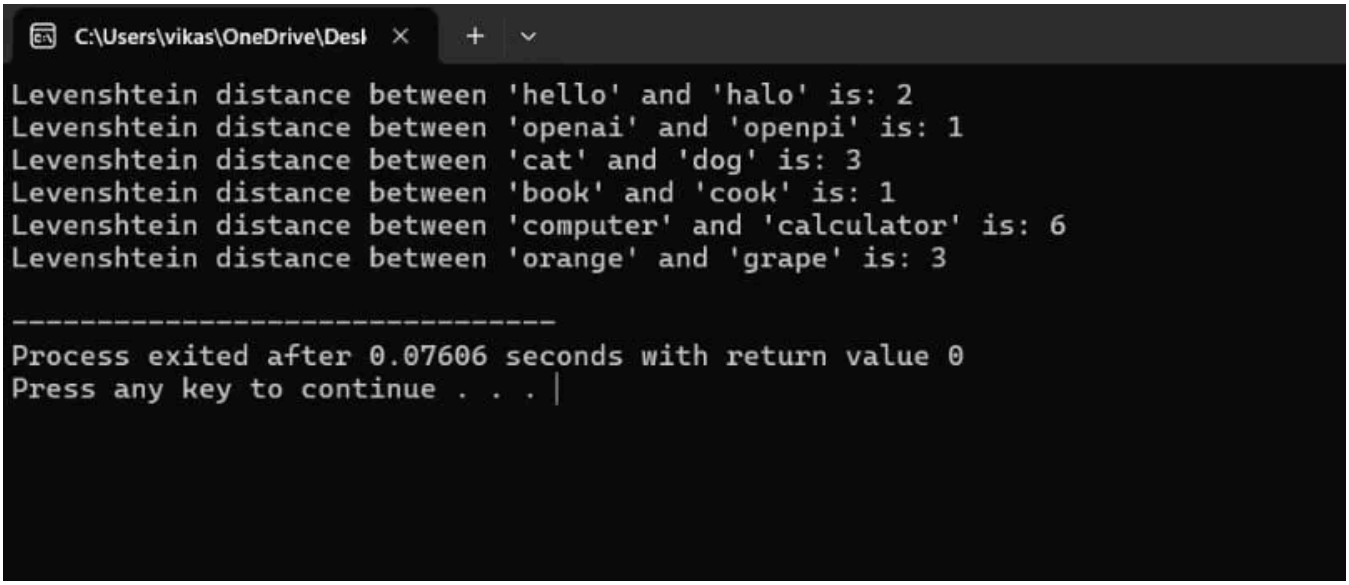
```cpp
#include <iostream>
#include <fstream>
#include <vector>
#include <string>
#include <algorithm>
using namespace std;
int minDistance(int a, int b, int c) {
    int minimum = a;
    if (b < minimum)
        minimum = b;
    if (c < minimum)
        minimum = c;
```

```
        return minimum;
    }
    int calculateDistance(const string& str1, const string& str2) {
        int len1 = str1.length() + 1;
        int len2 = str2.length() + 1;
        vector<vector<int> > distanceMatrix(len1, vector<int>(len2, 0));
        for (int i = 1; i < len1; i++)
            distanceMatrix[i][0] = i;
        for (int j = 1; j < len2; j++)
            distanceMatrix[0][j] = j;
        for (int col = 1; col < len2; col++) {
            for (int row = 1; row < len1; row++) {
                int cost = (str1[row - 1] == str2[col - 1]) ? 0 : 1;
                distanceMatrix[row][col] = minDistance(distanceMatrix[row - 1][col]
                                        distanceMatrix[row][col - 1] + 1,      // inse
                                        distanceMatrix[row - 1][col - 1] + cost); // s
            }
        }
        return distanceMatrix[len1 - 1][len2 - 1];
    }
    int main() {
        ifstream inputFile("input.txt");
        if (!inputFile.is_open()) {
            cerr << "Error opening file." << endl;
            return 1;
        }
        vector<string> lines;
        string currentLine;
        while (getline(inputFile, currentLine)) {
            lines.push_back(currentLine);
        }
        inputFile.close();
        int numPairs = lines.size() / 2;
        for (int i = 0; i < numPairs; i++) {
            string str1 = lines[i * 2];
            string str2 = lines[i * 2 + 1];
            str1.erase(remove(str1.begin(), str1.end(), '\n'), str1.end());
            str2.erase(remove(str2.begin(), str2.end(), '\n'), str2.end());
            int distance = calculateDistance(str1, str2);
            cout << "Levenshtein distance between '" << str1 << "' and '" << str2 <
        }
        return 0;
    }
```

## Example:(input.txt)

```
hello
halo
openai
openpi
cat
dog
book
cook
computer
calculator
orange
grape
```

## Output:




## Time Complexity:

The time complexity of the Levenshtein distance algorithm implemented in your code can be analyzed as follows:

Best Case: If the two input strings are identical, meaning there are no insertions, deletions, or substitutions needed, the algorithm will only perform initialization of the distance matrix, which takes O(m * n) time, where m and n are the lengths of the input strings. Therefore, the best-case time complexity is O(m * n), where m and n are the lengths of the input strings.

Worst Case: In the worst case, the algorithm iterates through each cell of the distance matrix, performing constant time operations for each cell. Therefore, the

worst-case time complexity is O(m * n), where m and n are the lengths of the input strings.

Average Case: The average-case time complexity is also O(m * n), as the algorithm typically involves iterating through each cell of the distance matrix, with the number of iterations being proportional to the product of the lengths of the input strings.

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

## GitHub Repository link:

**GitHub - vikaskomeraa/LEVENSTEIN-DISTANCE**

Contribute to vikaskomeraa/LEVENSTEIN-DISTANCE development by creating an account on GitHub.

github.com

## Conclusion:

In this tutorial, we've discussed different algorithms to implement the Levenshtein distance. We've seen that the worst-case complexity is quadratic and thus the question of possible optimizations is crucial to our efforts to provide a usable implementation. Finally, we observed that besides some techniques to directly reduce the computational cost, different approaches may be used to address the tasks on the agenda.

Edit profile

## Written by Vikaskomera

0 Followers