Understanding Similarity Metrics: A Deep Exploration of Levenshtein Distance and Its Variants

Full Name: Fidan Karimli

Date: 7 March 2025

Institution: Baku Higher Oil School

(Note: Since resources and detailed explanations are primarily in English, this report is written in English to ensure clarity and depth.)

Table of Contents

1. Introduction

Ever wondered how the auto-suggest feature on your smartphone works? How does your phone always seem to know which word you're attempting to spell? The solution lies in the Edit Distance algorithm, which calculates the difference between the word being typed and words in the dictionary. The words with the smallest difference are suggested first, while those with larger differences are ranked lower.

Edit Distance is a fundamental concept in computer science that measures the similarity between two strings by calculating the minimum number of operations required to transform one string into another. This paper explores Edit Distance as a broad concept before delving deeply into the Levenshtein Distance. We examine its algorithmic foundations, implementation techniques, optimization strategies, and practical applications.

2. Understanding Edit Distance

2.1 Definition

In computational linguistics and computer science, edit distance is a string metric, i.e. a way of quantifying how dissimilar two strings (e.g., words) are to one another, that is measured by counting the minimum number of operations required to transform one string into the other. Edit distances find applications in natural language processing, where automatic spelling correction can determine candidate corrections for a misspelled word by selecting words from a dictionary that have a low distance to the word in question. In bioinformatics, it can be used to quantify the similarity of DNA sequences, which can be viewed as strings of the letters A, C, G and T.

Different definitions of an edit distance use different sets of like operations. Levenshtein distance operations are the removal, insertion, or substitution of a character in the string. Being the most common metric, the term Levenshtein distance is often used interchangeably with edit distance.

2.2 Types of Edit Distance

There are several variations of Edit Distance, each designed for specific applications:

- **Levenshtein Distance**: This metric allows three operations: insertion, deletion, and substitution. It is widely used in spell checking,DNA analysis,plagiarism detection and auto suggestion and in other fields.
- **Damerau-Levenshtein Distance**: This variation extends Levenshtein Distance by allowing transposition (swapping two adjacent characters) in addition to insertion, deletion, and substitution. It is particularly useful for correcting typos in Optical Character Recognition (OCR) and text input systems.
- **Hamming Distance**: Unlike other metrics, Hamming Distance only considers substitutions and requires strings of equal length. It is primarily used in error detection and correction in networking and data transmission.

- **Jaro-Winkler Distance**: This metric measures the similarity of two strings based on matching characters, transpositions, and prefix weighting. It is commonly used in name matching, record linkage, and fuzzy string matching.

Algorithm	Operations Allowed			
	Insertions	Deletions	Substitutions	Transposition
Levenshtein Distance	✓	✓	✓	
Longest Common Subsequence (LCS)	✓	✓		
Hamming Distance			✓	
Damerau–Levenshtein Distance	✓	✓	✓	✓
Jaro distance				✓

Among these variations, Levenshtein Distance is one of the most widely adopted methods, forming the foundation for numerous string similarity measures.

2.3 Formal definition and properties

Given two strings a and b over an alphabet Σ, the **edit distance** d(a,b) is the minimum-weight sequence of edit operations that transforms a into b. One of the simplest sets of edit operations is defined by **Levenshtein** in 1966:

- **Insertion** of a single symbol: If $a = uv$, then inserting symbol x produces uxv, denoted as ε→x, where ε represents the empty string.
- **Deletion** of a single symbol: Changing uxv to uv (denoted x→ε).
- **Substitution** of a single symbol x for a symbol y ≠ x. Changing uxv to uxv (denoted x→y).

In Levenshtein's original definition, each operation has unit cost (except when substituting a character by itself, which has zero cost). Thus, **Levenshtein distance** is the minimum number of operations needed to transform string a to string b. A more general version associates non-negative weight functions, such as $w_{ins}(x), w_{del}(x)\ and\ w_{sub}(x)$ with each operation.

Some additional primitive operations have been proposed. For example, **Damerau-Levenshtein distance** includes **transposition** of two adjacent characters as an edit. This operation, which changes $uxyv$ to $uyxv$, is useful for handling common typographical errors where adjacent characters are swapped (e.g., "acress" to "acres").

Other variations of edit distance arise from restricting the allowed operations:

- **Longest Common Subsequence (LCS) distance** involves only **insertions** and **deletions** as the operations with unit cost.

- **Hamming distance** restricts the operations to **substitutions** only (with unit cost), and it can only be used with strings of equal length.
- **Jaro-Winkler distance** can be seen as a variant where only **transpositions** are allowed

Note on Cost:

In the context of **edit distance**, the **cost** refers to the "weight" or "value" associated with each edit operation (insertion, deletion, substitution, or transposition).

Unit cost means that each operation (insertion, deletion, substitution) is assigned the same cost, typically **1 unit**. For example:

- Insertion: **1 unit**
- Deletion: **1 unit**
- Substitution: **1 unit**

Non-unit cost: In more general definitions, the cost of each operation can vary:

- **Insertion cost** could be different (e.g., 2 units if it's considered a more costly operation).
- **Deletion cost** might be assigned a different value (e.g., 3 units if deletion is more expensive).
- **Substitution cost** could vary depending on the characters involved (e.g., substituting 'a' for 'b' might have a cost of 1 unit, but substituting 'a' for 'z' could have a cost of 3 units).

Example

The Levenshtein distance between "horse" and "ros" is 3. A minimal edit script that transforms the former into the latter is:

1. horse → rorse (substitute "h" for "r")
2. rorse → rose (remove "r")
3. rose → ros (remove "e" at the end)

LCS distance (insertions and deletions only) gives a different distance and minimal edit script:

1. horse → orse (remove "h")
2. orse → rorse (insert "r")
3. rorse → rose (delete "e" at 4)
4. rose → ros (insert "i" at 4)

for a total cost/distance of 4 operations.

It is just a beginning example for understanding problem. But manually, sometimes without guidance, we may not do it in the minimum number of steps. We will discuss the DP Table for Longest Common Subsequence (LCS) ahead.

3. Mathematical Foundations of Levenshtein distance

3.1 Formal Definition

The Levenshtein algorithm, also known as the Edit Distance algorithm, is a method used to measure the similarity between two strings. Developed by the Russian mathematician Vladimir Levenshtein in 1965, it calculates the minimum number of single-character edits required to change one string into another. These edits include insertion, deletion, or substitution of a single character. The smaller the Levenshtein distance, the more similar the strings are.

The allowed operations are:

- **Insertion**: Adding a character.
- **Deletion**: Removing a character.
- **Substitution**: Replacing one character with another.

Mathematically, using Levenshtein's original operations, the (nonsymmetric) edit distance from $a = a_1 \ldots a_m$ to $b = b_1 \ldots b_n$ is given by d_{mn}, defined by the recurrence here

$$d_{i0} = \sum_{k=1}^{i} w_{\text{del}}(a_k), \qquad \text{for } 1 \le i \le m$$

$$d_{0j} = \sum_{k=1}^{j} w_{\text{ins}}(b_k), \qquad \text{for } 1 \le j \le n$$

$$d_{ij} = \begin{cases} d_{i-1,j-1} & \text{for } a_i = b_j \\ \min \begin{cases} d_{i-1,j} + w_{\text{del}}(a_i) \\ d_{i,j-1} + w_{\text{ins}}(b_j) \\ d_{i-1,j-1} + w_{\text{sub}}(a_i, b_j) \end{cases} & \text{for } a_i \ne b_j \end{cases} \qquad \text{for } 1 \le i \le m, 1 \le j \le n.$$

$w_{\text{del}}(a_k)$ → The cost of deleting a character a_k.

$w_{\text{ins}}(b_k)$ → The cost of inserting a character b_k.

$w_{\text{sub}}(a_i, b_j)$ → The cost of substituting a_i with b_j.

This is more generalized form called **Weighted Levenshtein Distance** so more **flexible** because it allows different costs for each operation.

Also, we can use **Standard Levenshtein Distance** if we assume that all operations (insertion, deletion, and substitution) have the same cost of **1**. The recurrence formula is

$$D(i,j) = \begin{cases} \max(i,j), & \text{if } \min(i,j) = 0 \\ \min \begin{cases} D(i-1,j) + 1, & \text{(deletion)} \\ D(i,j-1) + 1, & \text{(insertion)} \\ D(i-1,j-1) + 1(a_i \ne b_j) & \text{(substitution)} \end{cases}, & \text{otherwise} \end{cases}$$

3.2 Dynamic Programming Recurrence Relation

To compute the Levenshtein distance efficiently, a dynamic programming approach is used. The recurrence relation is:

$$d_{i,j} = \min \begin{cases} d_{i-1,j} + 1, & \text{(Deletion)} \\ d_{i,j-1} + 1, & \text{(Insertion)} \\ d_{i-1,j-1} + \text{cost}(a_i, b_j), & \text{(Substitution)} \end{cases}$$

where cost (a_i, b_j) is 0 if $a_i = b_j$, and 1 otherwise. This recurrence ensures that we explore all possible transformations and choose the minimum cost path to convert one string into another.

4. Algorithmic Implementations of Levenshtein Distance

4.1 Naïve Recursive Approach (Top-Down Without Memoization)

The **naïve recursive approach** computes the Levenshtein Distance by breaking down the problem into smaller subproblems. The basic idea is to perform recursive calls for all possible operations (insert, delete, substitute) at each position of the strings and return the minimum number of operations required. Here's how it works:

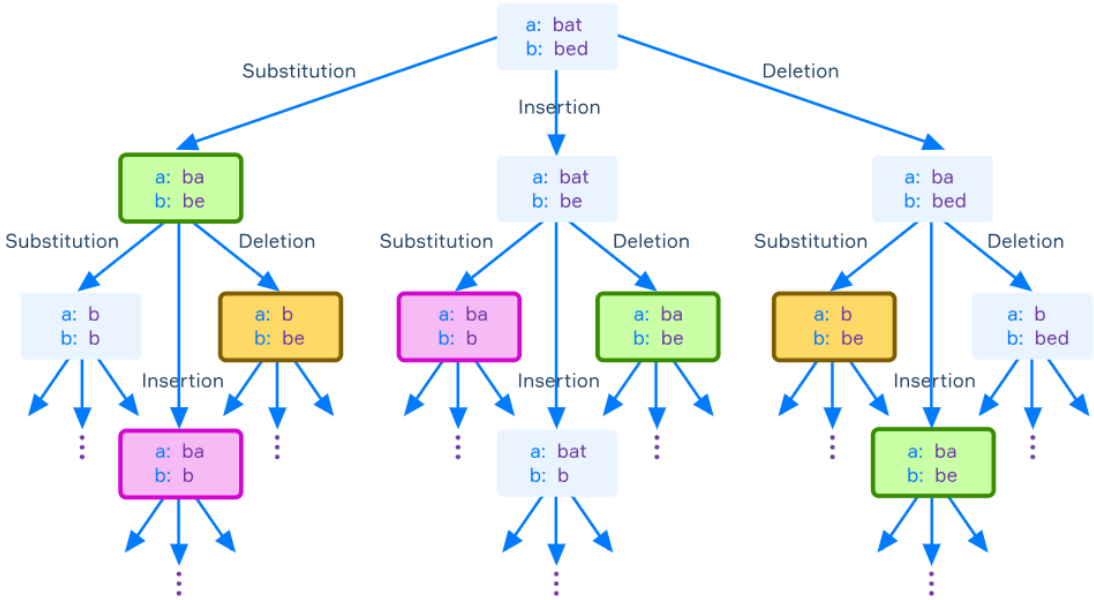
In the **base case**, if either of the two strings is empty, the distance is the length of the other string, since converting an empty string into a non-empty one requires inserting all characters of the non-empty string. If both strings are empty, the distance is 0, as no operations are needed.

In the **recursive case**, if the characters at the current positions of both strings match, no operation is needed, and the algorithm proceeds to the next character in both strings. If the characters differ, the algorithm recursively computes the Levenshtein Distance for three possible operations:

- **Insertion**: Insert a character into the first string to match the second string.
- **Deletion**: Remove a character from the first string to match the second string.
- **Substitution**: Replace the character in the first string with the corresponding character from the second string.

For each of these three operations (insertion, deletion, and substitution), the algorithm makes recursive calls to compute the respective distances and returns the minimum of the three values, adding 1 to account for the operation performed.

Let’s take a look at the following recursion tree of the **naive recursive approach** for calculating Levenshtein distance, that we observed in a previous topic. Here, a = "bat" and b = "bed".



The recursion tree highlights repetitive calculations using the same color. There are multiple evaluations of substring combinations of strings a and b, leading to unnecessary redundant computations.

4.1.1 Complexity Analysis of the Naïve Recursive Approach

Time Complexity: $\boldsymbol{O(3^{min(m,n)})}$

The naive approach leads to repeated subproblems because of overlapping recursive calls, resulting in exponential time complexity. Specific ally, for strings of length mmm and n, there are approximately $\boldsymbol{3^{min(m,n)}}$recursive calls, as at each position, the algorithm branches into three recursive calls (insert, delete, and substitute). This exponential growth in recursive calls makes the naive approach infeasible for long strings.

Space Complexity: $\boldsymbol{O(min(m,n))}$

The space complexity of the naive recursive approach is determined by the maximum recursion depth, which is at most $\boldsymbol{min(m,n)}$. Unlike memoization or dynamic programming approaches, this method does not store intermediate results, meaning it does not use additional memory beyond the recursion stack. As a result, the overall space complexity is $\boldsymbol{O(min(m,n))}$.

Inefficiency for Long Strings:

As the length of the strings increases, the number of subproblems grows exponentially. For larger strings, this inefficiency results in extremely slow computation times, making the naive recursive approach impractical for real-world applications.

However, we can optimize these redundant recursive calls and avoid such inefficiencies. To achieve this optimization, we can implement a technique called **memoization**. Memoization enables us to store the results of previously computed sub-problems, allowing us to reuse these results and avoid redundant calculations.

4.2 Optimized Top-Down Approach (Memoization)

The top-down dynamic programming approach is simply an extension of the naive recursive approach. In fact, it closely resembles the naive recursive approach, with only a few minor adjustments. This approach involves incorporating caching within the recursive function calls. To store the result of each sub-problem, we employ a data structure known as a **memoization table** or **cache**. By using this cache, we optimize the calculations by storing and reusing previously computed results.

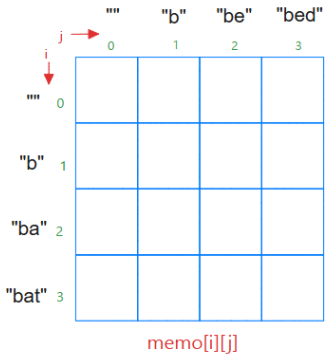
Memoization enables us to store the results of previously computed sub-problems, allowing us to reuse these results and avoid redundant calculations.By **caching** or memoizing the results of each sub-problem, the algorithm becomes more efficient. Before computing the result for a particular sub-problem, it checks if the result is already present in the cache. If so, it terminates the computation and directly retrieves the pre-computed result from the cache, eliminating the need for repeated evaluations.

Now, let's delve into the implementation steps of the top-down dynamic programming approach.

1. Caching Mechanism

- In a recursive call with specific parameters (e.g. minDistance(a, b, i, j)), we store the result of the sub-problem with string a ending at index i and string b ending at index j in the cache.
- Before computing the result of a sub-problem, we check if it already exists in the cache (e.g. memo[i][j]). If the result is cached, we directly retrieve it, avoiding redundant computations.
- After computing the result of a sub-problem, we store the result in the cache for future use. So, when encountering the same sub-problem in subsequent recursive calls, we can quickly retrieve its result from the cache (e.g. memo[i][j]).

Let’s see Cache array (or memorization table):



2. Base cases: We define the base cases for the recursive function:

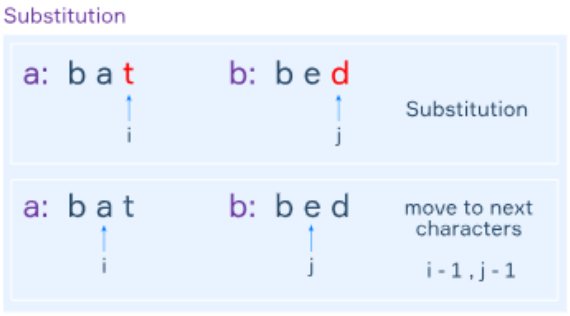
- If a is empty, the edit distance equals the number of characters in b.
- If b is empty, the edit distance equals the number of characters in a.

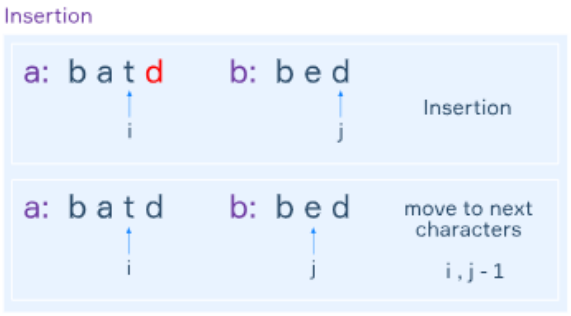
3. Character comparison: When comparing characters at indices i and j, two possibilities arise:

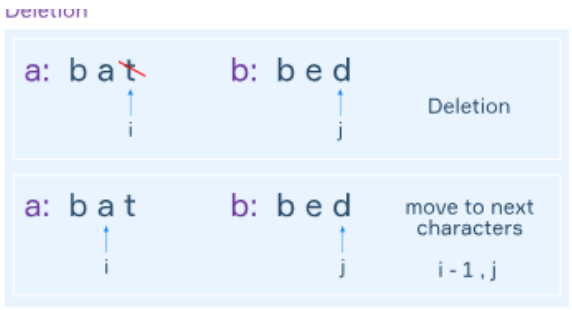
- If the characters match ($a_i = b_j$), we move to the next index without performing any operation.
- If the characters don't match ($a_i \neq b_j$), we need to consider the three operations: deletion, insertion, and substitution.

4.Recurrence relations: The recurrence relation for each possible operation is as follows:

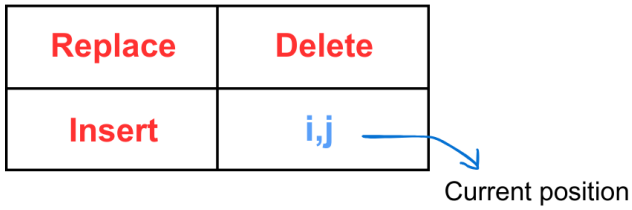
Let's derive it using an example with actual strings. Let’s take a = "bat" and b = "bed”.

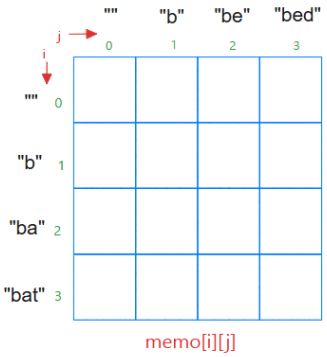


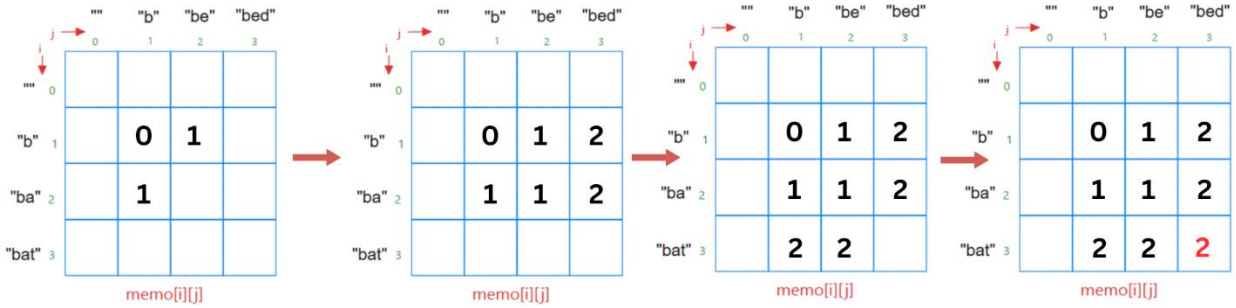


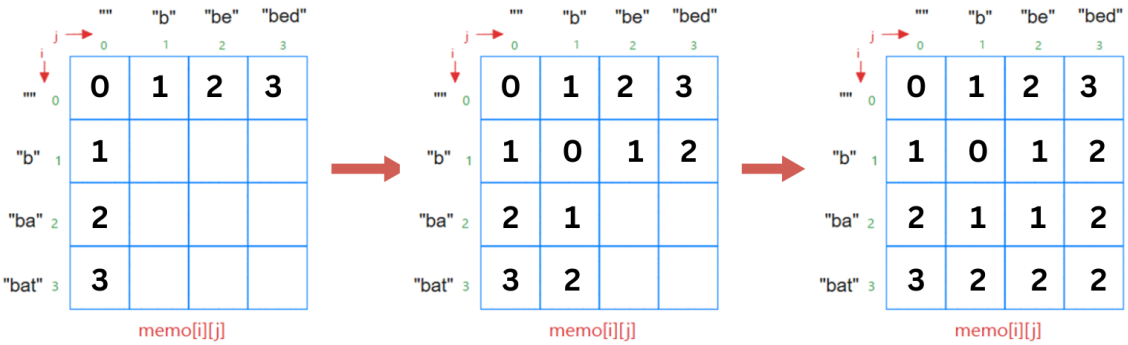


Let’s fill Cache array:

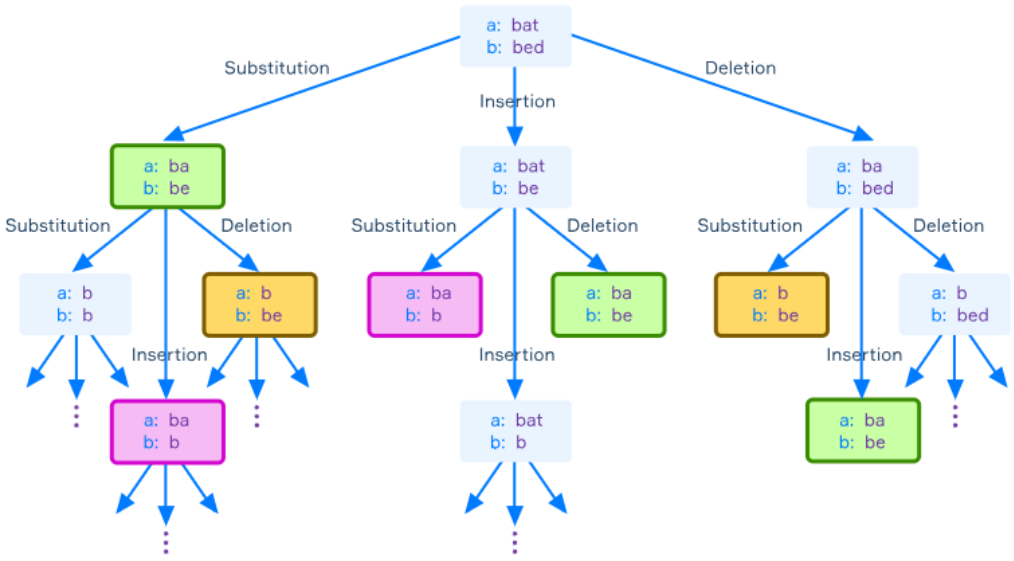








Let’s look at the recursion tree of the top-down dynamic programming approach for calculating the Levenshtein distance.This optimization ensures that computations are performed only once, and the results are stored for future reference.



4.2.1 Complexity Analysis of the Top-Down Approach (Memoization)

The top-down approach with memoization optimizes the naive recursive approach by storing previously computed results to avoid redundant calculations. This significantly improves the time complexity.

Time Complexity: $O(m \cdot n)$

The time complexity of the memoization approach is $O(n \cdot m)$ because for every combination of string *a* and string *b*, the result is computed only once and then stored in the memoization table. This optimization eliminates redundant calculations by reusing the previously computed results, significantly reducing the number of recursive calls.

Space Complexity: $O(m \cdot n)$

The space complexity is $O(m \cdot n)$ due to the additional 2D array memo, of size $n \times m$. This memoization table is used to cache the results of previously computed sub-problems. Consequently, the algorithm consumes additional memory to store and retrieve these cached results, but it ensures efficient reuse of the computed values.

4.3 From Memoization to Iterative Dynamic Programming

Memoization and dynamic programming both optimize recursive solutions by storing results. Memoization follows a **top-down** approach, using recursion with caching to avoid redundant calculations. However, it still relies on the call stack, leading to overhead and potential stack overflow for large inputs.

In contrast, **bottom-up dynamic programming** builds the solution iteratively, filling a table from smaller subproblems to larger ones. This avoids recursion, improves cache efficiency, and eliminates function call overhead. While both methods fall under dynamic programming, the bottom-up approach is generally preferred for better performance and scalability.

4.4 Bottom-Up Dynamic Programming (Iterative Approach)

Dynamic Programming (DP) is like solving a big puzzle, where instead of jumping straight to the final piece, you take the time to solve the smaller pieces first and then gradually build your way up to the solution. The **Bottom-Up** approach is a strategic and efficient way to solve these kinds of puzzles, where you begin by tackling the simplest, most basic pieces and then work your way toward the final picture.

This approach uses the tabulation technique to implement the dynamic programming solution. It addresses the same problems as before, but without recursion. The recursion is replaced with iteration in this approach. Hence, there is no stack overflow error or overhead of recursive procedures. We maintain a table (3D matrix) to solve the problem in this method.

$$dp[i][j] = \min \begin{cases} dp[i-1][j] + 1 & \text{(deletion)} \\ dp[i][j-1] + 1 & \text{(insertion)} \\ dp[i-1][j-1] + \text{cost} & \text{(substitution)} \end{cases}$$

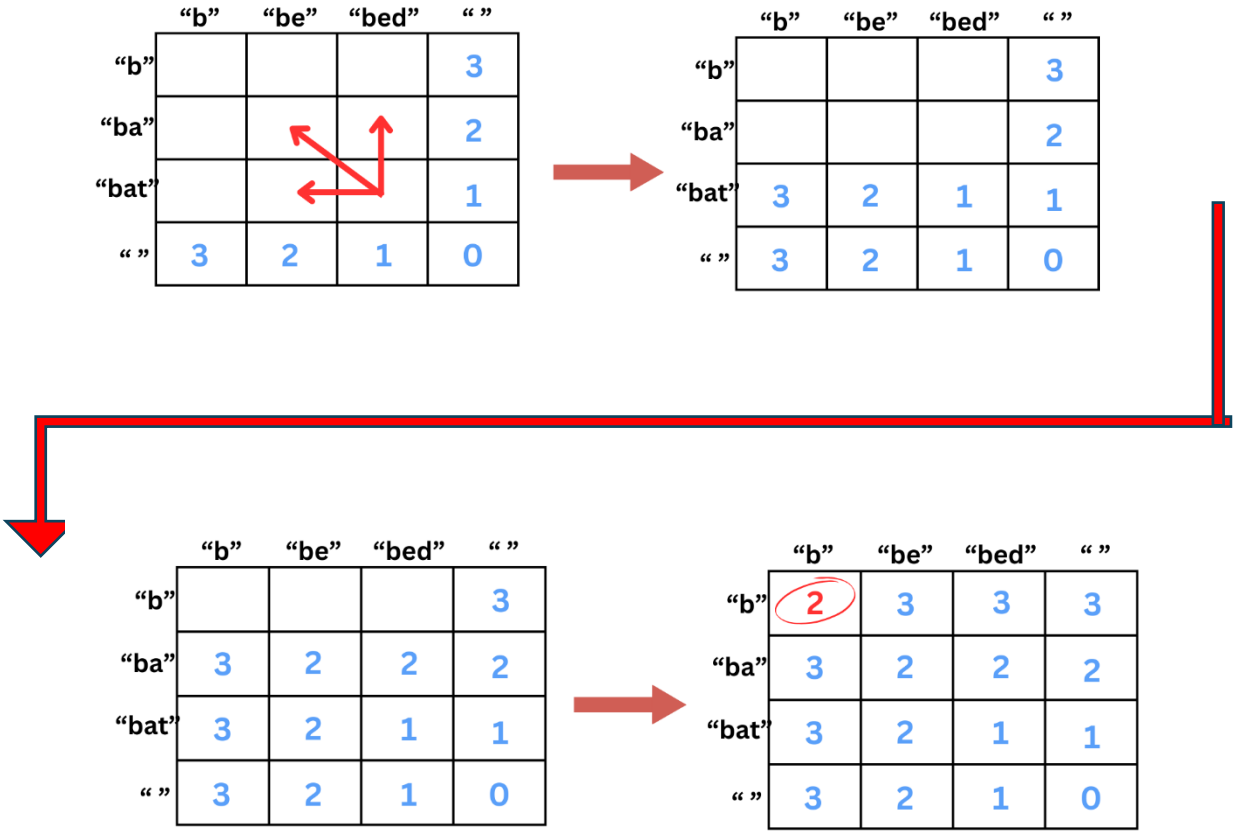
Let’s dive deep how it works:

1.**Building from the Ground Up**: Imagine trying to solve a maze. If you start from the exit and work backwards, you can figure out how to navigate to the start by breaking it down into smaller paths. This is essentially what Bottom-Up DP does: you begin by solving the simplest possible subproblems (the base cases) and then combine their solutions to solve increasingly complex subproblems, all the way up to the final solution. It's like starting with a tiny building block and stacking them higher, one by one.

2. **Iterative, Not Recursive**: Instead of using recursion (where functions call themselves and build on each other like a chain), the Bottom-Up approach uses iteration. This means you build solutions step by step, filling in a table to keep track of the work you've already done. No need for a long series of function calls or the risk of "losing track" because everything is done systematically, one step after the other.

3. **A Table of Solutions**: Think of the DP table as a giant grid, where each cell contains the solution to a subproblem. As you work through the problem, you fill in this grid from the simplest cases (like an empty string) and gradually solve the larger, more complex ones. By the time you reach the final cell, you've already solved everything you need to reach the solution.

Let's see a DP table with an example for word "bat" to "bed".



The top-left cell **dp[0][0]** contains the final Levenshtein distance.This approach ensures an **optimal** and **efficient** solution using **O(m × n) time and space complexity**.

4.4.1 Comparison of Bottom-Up vs. Top-Down Dynamic Programming for Levenshtein Distance

The bottom-up dynamic programming approach, or the Wagner-Fisher Algorithm, is indeed more efficient than the top-down approach because it avoids the overhead of recursion and utilizes a tabulation technique to build the solution iteratively. By iteratively building the solution from smaller subproblems to larger ones, the bottom-up approach ensures that each subproblem is solved exactly once, eliminate redundancy by calculating and reusing values in a

tabulated manner. Additionally, the tabulation technique used in the bottom-up approach lends itself well to parallelization, making it suitable for parallel computing environments where multiple computations can be performed simultaneously. Overall, the bottom-up approach is preferred for its efficiency, especially in scenarios where performance and scalability are critical.

4.5 Wagner-Fischer Algorithm (Space-Optimized DP)

Vladimir Levenshtein introduced the Levenshtein distance algorithm in 1965 for spell-checking and calculating the similarity between two strings. However, the algorithm's recursive nature led to inefficiencies, with an exponential time complexity of O(3^n), where n is the length of the strings.

In 1971, Robert Wagner and Michael Fischer revolutionized string comparison with the Wagner-Fischer algorithm. This dynamic programming approach vastly improved efficiency by solving each subproblem just once and storing their solutions. This innovation eliminated the excessive repetitive work of Levenshtein's recursive approach, propelling string comparison into a new era of efficiency.

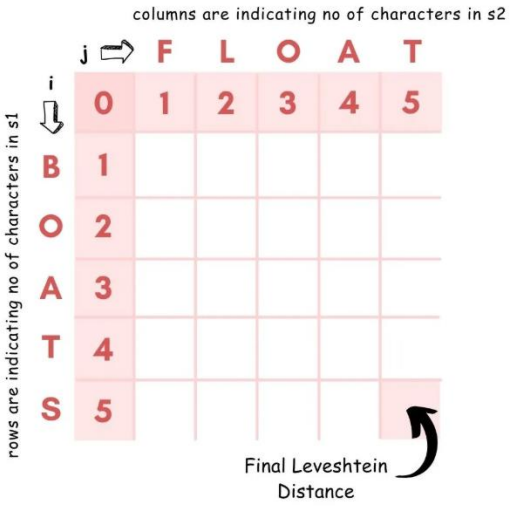
Wagner's Fischer algorithm is based on dynamic programming breaking down the complex problem into simpler sub problems solving each of these sub problems just once and storing their solutions by doing this the the algorithm avoids the excessive repetitive work seen in Levin's recursive approach.

How it works:

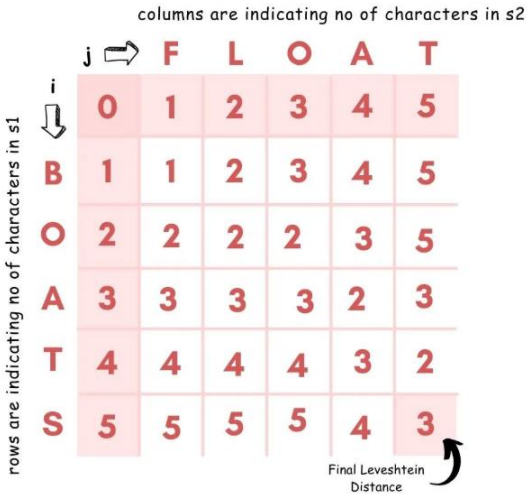
The key observation is that at any point while filling the DP table, we only need information from the **previous row** (for deletions) and the **current row being computed** (for insertions and substitutions). Instead of storing the entire matrix, we maintain only **two rows at a time**, drastically reducing memory usage.

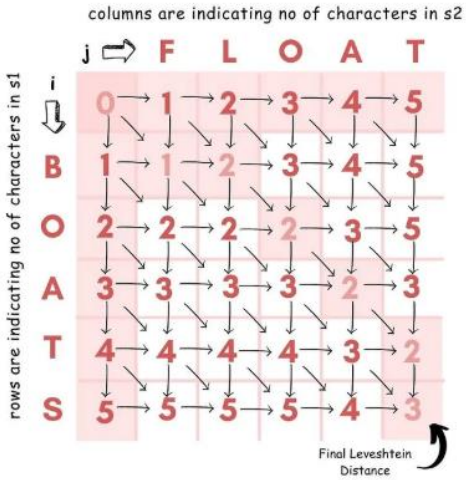
For even further optimization, we can use **just one row** and update it iteratively, since each row depends only on the values from the previous iteration. This allows us to achieve **O(min(m, n)) space complexity**, making the algorithm scalable for very large strings.

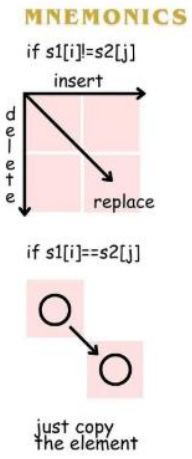
Example: we have two strings, "boats" and "float", and we want to find the minimum number of operations (insertions, deletions, or substitutions) required to transform "boats" into "float".



3.







The minimum edit distance for boats → float is 3.

5. Variants, Extensions, and Optimizations of Levenshtein Distance

5.1 Damerau-Levenshtein Distance

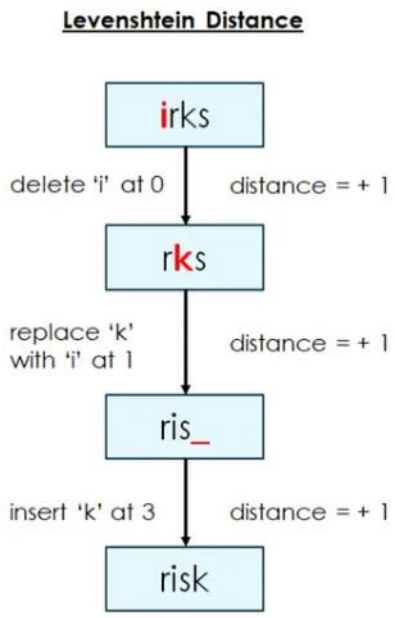
The Damerau-Levenshtein distance is an extension of the classic Levenshtein distance, making it more effective in handling common real-world errors. While Levenshtein distance measures the number of insertions, deletions, and substitutions needed to transform one string into another, Damerau-Levenshtein goes one step further—it also considers adjacent transpositions (swapping two neighboring characters).

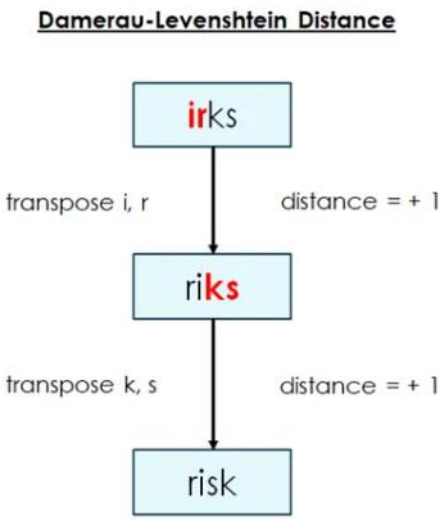
Key Difference: Transposition Handling

The Damerau-Levenshtein distance extends the standard Levenshtein distance by including an additional operation: transposition**.** This allows for the correction of adjacent character swaps as a single operation rather than two separate edits.

This refinement makes Damerau-Levenshtein more accurate for applications like spell-checkers and OCR correction**,** where transposition errors are common.

For example, the Levenshtein distance between "CA" and "AC" is 2 (one deletion + one insertion). However, the Damerau-Levenshtein distance is 1, as it recognizes the transposition as a single edit.



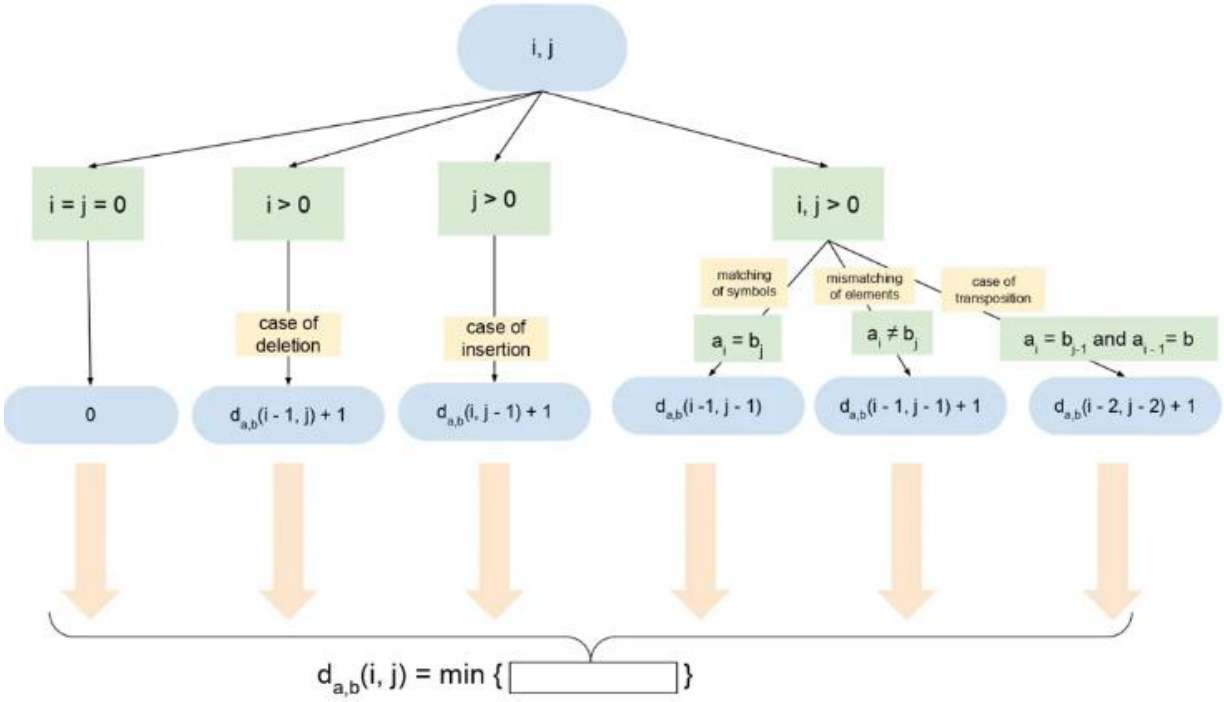


Formal Defiinition

$$a = a_1 ... a_i ... a_n$$
$$b = b_1 ... b_j ... b_m$$

$$d_{a,b}(i,j) = min \begin{cases} 0 & \text{if } i = j = 0 \\ d_{a,b}(i-1,j) + 1 & \text{if } i > 0 \\ d_{a,b}(i,j-1) + 1 & \text{if } j > 0 \\ d_{a,b}(i-1,j-1) & \text{if } i,j > 0 \text{ and } a_i = b_j \\ d_{a,b}(i-1,j-1) + 1 & \text{if } i,j > 0 \text{ and } a_i \neq b_j \\ d_{a,b}(i-2,j-2) + 1 & \text{if } i,j > 0 \text{ and } a_i = b_{j-1} \text{ and } a_{i-1} = b_j \end{cases}$$

For a better understanding, let's have a look at an illustration of the algorithm:



To calculate the Damerau–Levenshtein distance, we need to fill the distance matrix with values. The idea is the following:

1. Define a distance matrix, where the number of rows and columns is equal to the length of the provided strings.
2. Add an extra row and column that will hold characters' indices to count.
3. Calculate the values of the function $d_{a,b}(i,j)$, which depend on the row i*i* and the column *j*, using the formula.
4. Choose the minimum among all possible $d_{a,b}(i,j)$ values.

5. Fill up the cell at the intersection of the column and the row with the found minimum value.
6. After filling up the matrix, the score in the lower-right cell will be the answer.

Optimization Technique

Similar to the Wagner-Fischer algorithm, the Damerau-Levenshtein distance can be efficiently computed using dynamic programming**.** However, the full DP table requires O(m × n) time and space, which can be impractical for large-scale applications.

Space-Optimized DP (Using Two Rows)

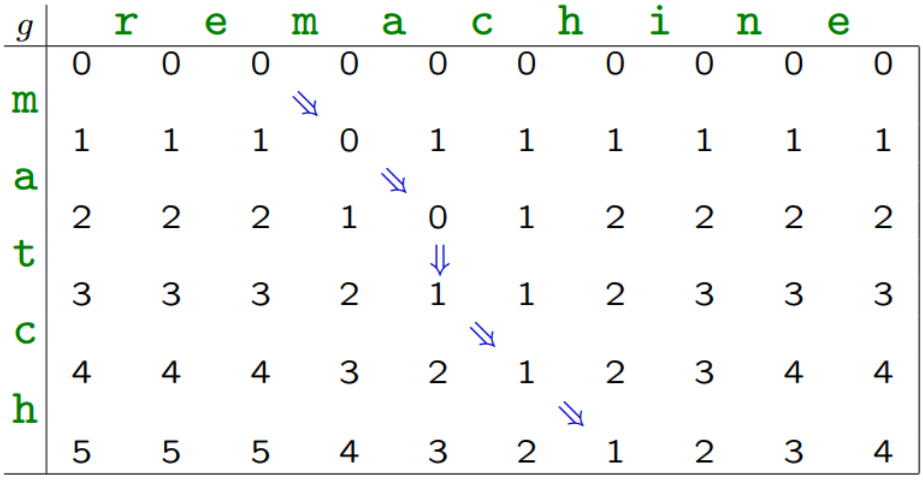
Rather than storing the entire DP table, we can reduce space complexity from O(m × n) to O(min(m, n)) by keeping only the current row and the previous row. This optimization works because each DP cell depends only on nearby values from the previous row and column, making full table storage unnecessary.

5.2 Ukkonen's Algorithm (Fast Approximate Matching)

When working with large-scale text processing tasks—such as real-time spell-checking, fuzzy search, or DNA sequence alignment—traditional dynamic programming (DP) methods like Wagner-Fischer can be too slow. Ukkonen's Algorithm, introduced by Esko Ukkonen in 1985, optimizes the Levenshtein distance calculation by eliminating redundant computations and narrowing the search space.

This makes it a great choice for fast approximate string matching, especially when dealing with long texts and large datasets where exact matches are rare, and we only need "good enough" results.Approximate string matching, or fuzzy string searching, involves finding occurrences of a pattern within a text where the match may not be exact. The degree of similarity between the pattern and the text is often quantified using metrics like the Levenshtein distance, which counts the minimum number of single-character edits (insertions, deletions, or substitutions) required to transform one string into another.

Let's take a look at an example where P= "match" and T=" remachine", k=1. We want to find all positions j ∈ [1..m] such that $ed(P, T(j - l...j]) \leq k$ for some $l \geq 0$.



One occurrence ending at position 6.

Challenges with Traditional Methods

Traditional dynamic programming approaches to approximate string matching typically operate with a time complexity of O(mn), where *m* is the length of the pattern and *n* is the length of the text. This quadratic complexity can be computationally intensive, especially when dealing with large texts or when the pattern and text lengths are similar.

Ukkonen's Contribution

Ukkonen's algorithm addresses these challenges by introducing an "output-sensitive" approach. Instead of evaluating all possible substrings, it focuses on those within a specified edit distance threshold, effectively reducing unnecessary computations. This results in a time complexity of O(nd), where *d* is the maximum allowed edit distance, making the algorithm more efficient, particularly when *d* is small relative to *n*.

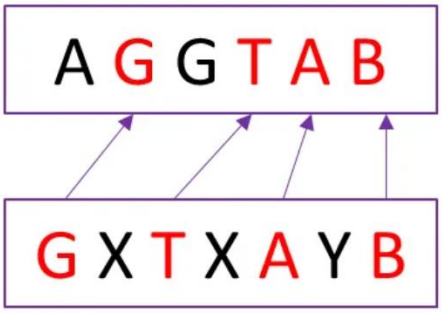
Key Features of Ukkonen's Algorithm

1. **Edit Distance Threshold**: Users can define a maximum allowable edit distance, enabling the algorithm to concentrate on substrings that meet this criterion and disregard those that don't.
2. **Diagonal Monotonicity**: The algorithm leverages the property that the edit distance matrix's diagonals are monotonically non-decreasing. This insight allows for pruning certain computations, further enhancing efficiency.
3. **Space Optimization**: By storing only relevant portions of the edit distance matrix, Ukkonen's algorithm reduces memory usage, making it more practical for large-scale applications.

5.3 Longest Common Subsequence (LCS) Distance

The Longest Common Subsequence (LCS) distance measures the similarity between two sequences by identifying the longest subsequence common to both. Unlike the Levenshtein distance, which accounts for insertions, deletions, and substitutions, LCS distance considers only insertions and deletions. This makes it simpler but potentially less flexible in capturing certain types of differences between sequences.

Example:



Dynamic Programming Approach

A standard method for solving the LCS problem utilizes dynamic programming. This approach constructs a two-dimensional table where each entry (i, j) represents the length of the LCS of the prefixes X[1..i] and Y[1..j]. The table is filled using the following recurrence relations:

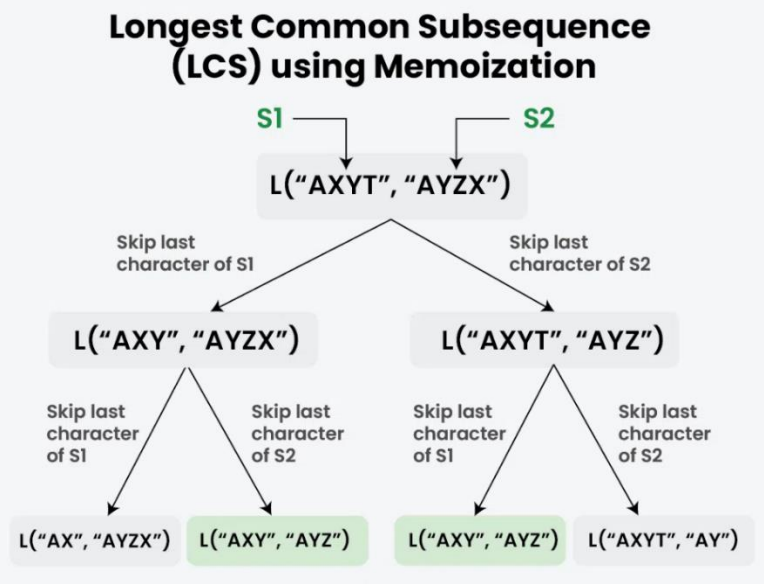
- If X[i] = Y[j], then C[i][j] = C[i-1][j-1] + 1.
- Otherwise, C[i][j] = max(C[i-1][j], C[i][j-1]).

This method has a time complexity of O(mn), where m and n are the lengths of sequences X and Y, respectively. The space complexity is also O(mn) due to the storage requirements of the table.

We can do some optimizations.

1. Space Optimization: By observing that the computation of each table entry depends only on the current and previous rows or columns, the space complexity can be reduced to O(min(m, n)). This is achieved by storing only the necessary portions of the table at any given time.

2. Hirschberg's Algorithm: This algorithm leverages the divide-and-conquer technique to compute the LCS in linear space while maintaining a time complexity of O(mn). It is particularly useful when memory usage is a concern. When wwe need the LCS for massive datasets but can’t afford O(n * m) space, Hirschberg’s Algorithm is the go-to solution.



6. Comparisons with Other Metrics

6.1 Jaro-Winkler Distance

The Jaro-Winkler distance is a string metric designed to measure the similarity between two strings, particularly optimizing for short strings and personal names. It is widely used in record linkage tasks, where identifying duplicate records or matching similar entries is crucial.

Jaro Similarity

The Jaro similarity sim_j of two given strings s_1 and s_2 is

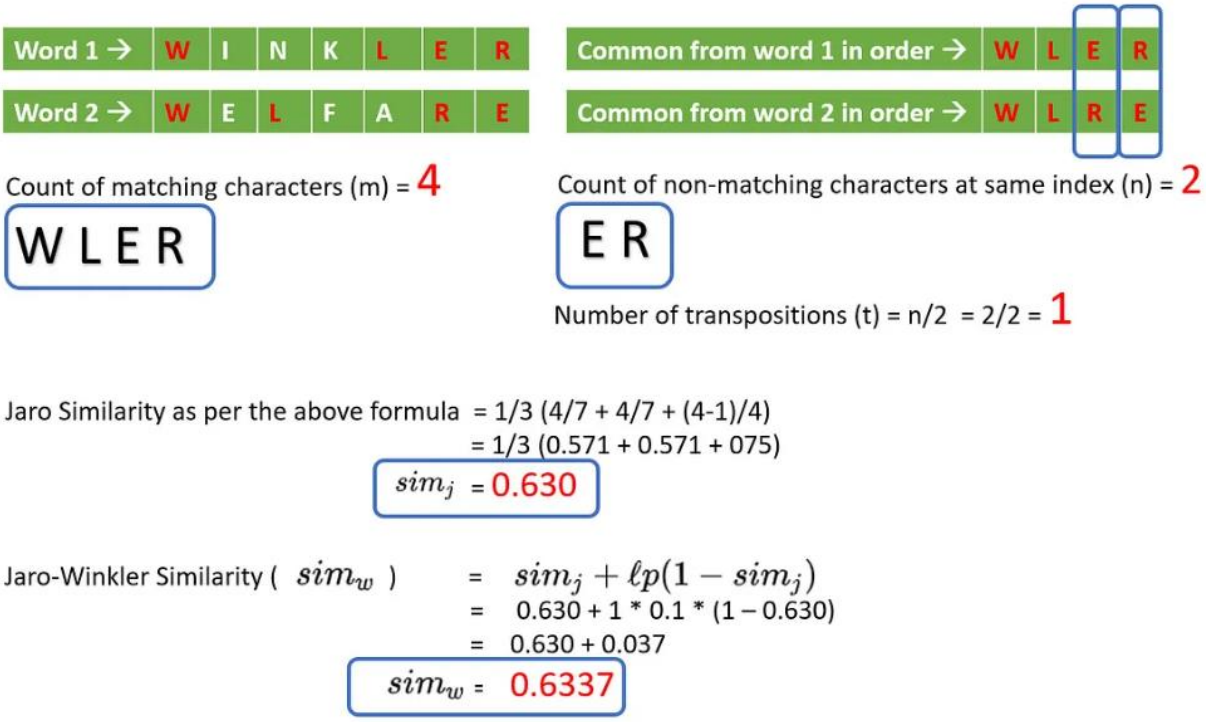
$$sim_j = \begin{cases} 0 & \text{if } m = 0 \\ \frac{1}{3}\left(\frac{m}{|s_1|} + \frac{m}{|s_2|} + \frac{m-t}{m}\right) & \text{otherwise} \end{cases}$$

- $|s_i|$ is the length of the string s_i;
- m is the number of matching characters;
- t is the number of transpositions

Jaro–Winkler similarity

$$sim_w = sim_j + \ell p(1 - sim_j),$$

- $|sim_j|$ is the Jaro similarity for strings s_1 and s_2;
- l is the length of the common prefix at the start of the string up to a maximum of 4 characters ;
- p sfb is a constant scaling factor for how much the score is adjusted upwards for having common prefixes ;



Applications of Jaro-Winkler Distance

The Jaro-Winkler distance is particularly effective in scenarios where minor typographical errors are common, such as:

- Record Linkage and Deduplication: Identifying duplicate records in datasets where names or short strings may have slight variations.
- Data Cleaning: Correcting or standardizing data entries by detecting and merging similar strings.
- Search Autocorrect: Providing suggestions or corrections for user input in search engines or databases.

Jaro-Winker v/s Levenshtein — Which one to use?

Jaro-Winkler takes into account only matching characters and any required transpositions (swapping of characters). Also, it gives more priority to prefix similarity. Levenshtein counts the number of edits to convert one string to another. So, it's really a choice based on use cases and

there is no perfect answer of one vs the other. As a general guideline, I have seen Jaro-Winkler work well for single word comparisons and is more dependable. Also, in terms of performance Jaro-Winkler gives better performance than Levenshtein. But I would go with Levenshtein distance for longer string comparisons since I really get to know how different they are in terms of character replacements.

6.2 Hamming Distance

The Hamming distance is a metric that quantifies the number of positions at which the corresponding symbols differ between two strings of equal length. It effectively measures the minimum number of substitutions required to transform one string into the other, or equivalently, the number of errors that could have led from one string to the other.

In everyday terms, the Hamming distance tells us how many positions two equal-length strings differ. For instance, if two DNA sequences of the same length have a Hamming distance of 5, it means there are five positions where the nucleotides differ between the two sequences. This metric is straightforward and particularly useful in scenarios where substitutions are the primary concern, and the strings being compared are of equal length.

Examples:

"**karolin**" and "**kathrin**" is 3.

"**karolin**" and "**kerstin**" is 3.

"**kathrin**" and "**kerstin**" is 4.

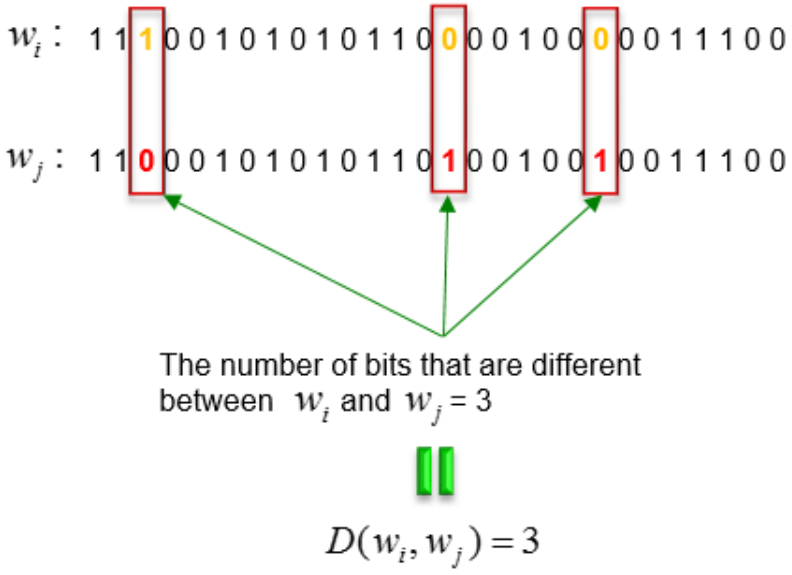
0000 and **1111** is 4.

2173896 and **2233796** is 3.

Applications

The Hamming distance is primarily utilized in coding theory, particularly in error detection and correction schemes. In this context, it determines the error-detecting and error-correcting capabilities of block codes.

1. **Error Detection and Correction**: Hamming distance is used in coding theory to detect and correct errors in data transmission. Hamming…
2. **DNA Sequence Analysis**: In bioinformatics, Hamming distance is used to compare DNA sequences and identify mutations or similarities between genetic codes.
3. **Cryptography**: Hamming distance is used to measure the similarity between cryptographic keys or hash values.



Comparison with Other String Similarity Metrics

While the Hamming distance is effective for strings of equal length, other metrics are more suitable for strings of varying lengths or when different types of edits are considered:

- **Levenshtein Distance**: Measures the minimum number of single-character edits (insertions, deletions, or substitutions) required to transform one string into another. Unlike the Hamming distance, it accommodates strings of different lengths and accounts for a broader range of edit operations.
- **Jaro-Winkler Distance**: Specifically designed for short strings and particularly effective in matching names, this metric considers character transpositions and common prefixes, offering a more nuanced similarity measure in certain contexts.

7. Fuzzy Matching

Fuzzy matching, also known as approximate string matching, refers to the process of identifying strings that are approximately equal, allowing for minor errors or differences. This technique is essential in various fields such as data cleaning, record linkage, spell checking, and natural language processing.

In computer science, fuzzy matching involves finding strings that match a pattern approximately rather than exactly. The closeness of a match is typically measured using the concept of edit distance, which quantifies the number of primitive operations (insertions, deletions, substitutions, or transpositions) required to transform one string into another. The Levenshtein distance is a well-known metric used to calculate this edit distance.

For that reason, I also implemented Python code for it

Tools and Libraries for Fuzzy Matching

Several tools and libraries facilitate fuzzy matching:

FuzzyWuzzy: A Python library that uses Levenshtein Distance to calculate the differences between sequences. It provides simple interfaces for string matching tasks.

RapidFuzz: An alternative to FuzzyWuzzy, offering faster performance and additional features for fuzzy string matching.

Apache Lucene: A high-performance, full-featured text search engine library written in Java, which includes fuzzy search capabilities.

With FuzzyWuzzy library:

```
Simple Ratio between 'Azərbaycan Respublikası' and 'Azerbaycan Respublikasi': 91
Partial Ratio between 'Azərbaycan Respublikası' and 'Azerbaycan Respublikasi': 91
Token Sort Ratio between 'Azərbaycan Respublikası' and 'Azerbaycan Respublikasi': 91
Token Set Ratio between 'Azərbaycan Respublikası' and 'Azerbaycan Respublikasi': 91
Best match for 'Bakiii' in ['Bakı', 'Gəncə', 'Sumqayıt', 'Şəki']: ('Bakı', 68)
```

With RapidFuzz:

```
Simple Ratio between 'Azərbaycan Respublikası' and 'Azerbaycan Respublikasi': 91.30
Partial Ratio between 'Azərbaycan Respublikası' and 'Azerbaycan Respublikasi': 93.33
Token Sort Ratio between 'Azərbaycan Respublikası' and 'Azerbaycan Respublikasi': 91.30
Token Set Ratio between 'Azərbaycan Respublikası' and 'Azerbaycan Respublikasi': 91.30
Best match for 'Bakiii' in ['Bakı', 'Gəncə', 'Sumqayıt', 'Şəki']: ('Bakı', 77.14285714285715, 0)
```

8. References

https://medium.com/@tejaswiyadav221/the-levenshtein-distance-algorithm-a-string-metric-for-measuring-the-difference-between-two-269afbbddd34

https://youtu.be/ocZMDMZwhCY?si=aAIF02XVlrBBHPUu - **Lecture 21: Dynamic Programming III: Parenthesization, Edit Distance, Knapsack**

https://en.wikipedia.org/wiki/Levenshtein_distance

https://www.geeksforgeeks.org/edit-distance-dp-using-memoization/

https://hien-luu.medium.com/dissecting-dynamic-programming-top-down-bottom-up-3d3a1d62fbd7

https://en.wikipedia.org/wiki/Jaro%E2%80%93Winkler_distance#:~:text=In%20computer%20science%20and%20statistics,Winkler.

https://srinivas-kulkarni.medium.com/jaro-winkler-vs-levenshtein-distance-2eab21832fd6#:~:text=The%20Jaro%E2%80%93Winkler%20distance%20is,means%20there%20is%20no%20similarity.

https://www.cs.helsinki.fi/u/tpkarkka/teach/16-17/SPA/lecture06.pdf

https://medium.com/@whyamit404/efficient-algorithms-for-finding-the-longest-common-subsequence-lcs-0eea5e44023a

https://medium.com/@m.nath/fuzzy-matching-algorithms-81914b1bc498

https://chrisyandata.medium.com/understanding-hamming-distance-a-measure-of-similarity-698ae2cb0ef6

https://rahulmadhani20.medium.com/coding-algorithm-question-longest-common-subsequence-lcs-1e0a4b3c47c7

https://www.researchgate.net/publication/221314127_A_Fast_Algorithm_for_Approximate_String_Matching_on_Gene_Sequences (I have just read it because so interesting)

https://youtu.be/XuWPdFteIkQ?si=5S2ZPq5q3JdMpDkR - **Complete Guide to Fuzzy Matching - Methods, Limitations & SEO Use cases**

https://www.analyticsvidhya.com/blog/2021/07/fuzzy-string-matching-a-hands-on-guide/