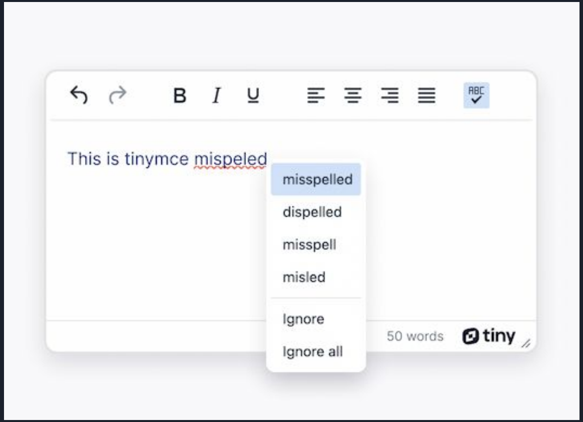
Levenshtein Edit Distance

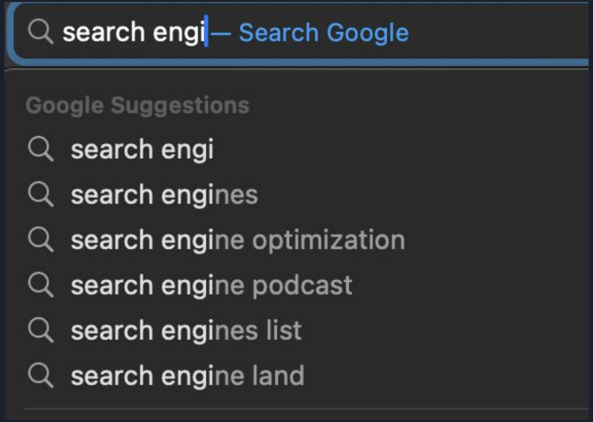
Shaurya Shrivastava

Purpose of Levenshtein Edit Distance

- Measures how similar or different two strings are
- Commonly used in spell checkers, search engines, DNA sequence matching
- Representation of dynamic programming



This is tinymce mispeled



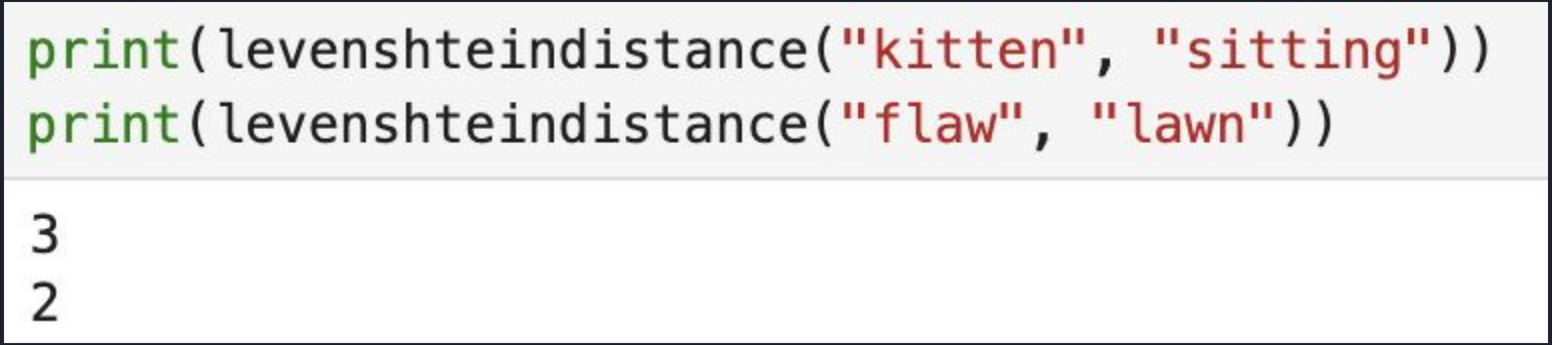
search engi — Search Google

Google Suggestions

search engi
search engines
search engine optimization
search engine podcast
search engines list
search engine land

Overview of Algorithm

- Input: two strings
- Output: # of edits
- Allowed operations: insert, delete, substitute
- Goal: minimum # edits to transform one into the other



```
print(levenshteindistance("kitten", "sitting"))
print(levenshteindistance("flaw", "lawn"))
```

```
3
2
```

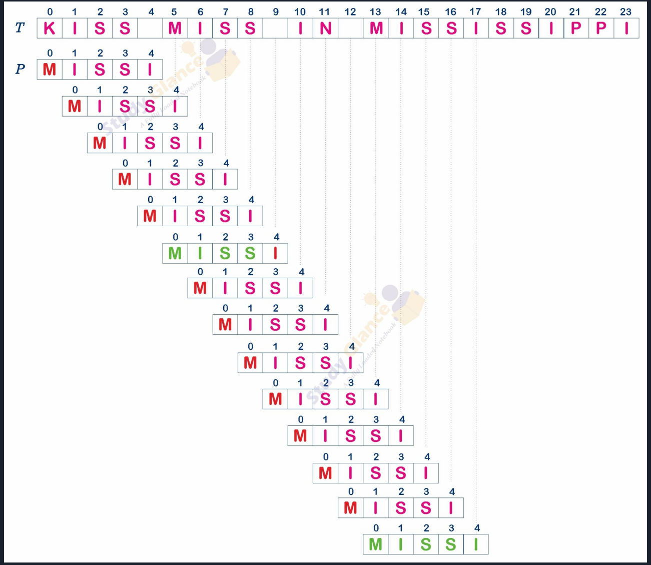
Problem Statement

Levenshtein Distance solves the problem of measuring similarity between text efficiently, when when certain text or sequences may differ due to formatting or typos. These are the fields that Levenshtein Distance is valuable to:

- Data Management: Can detect and merge duplicate records in databases, like customer names, DNA sequences, and product catalogs
- Text Processing: Drives autocorrect and spell-check on digital platforms
- Information Retrieval: Helps predict text using edit similarity on search engines

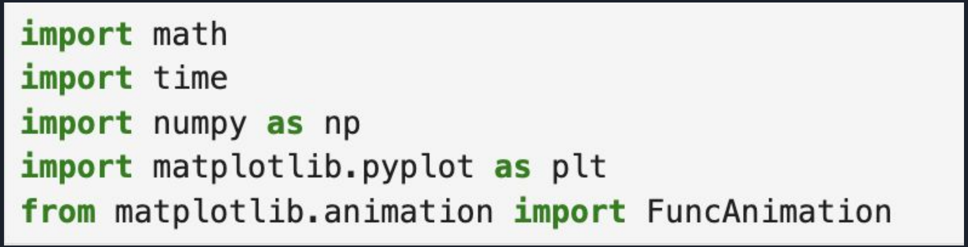
Brute Force Approach

- Requires trying all possible edit sequences at every step (insert, delete, substitute)
- Total possibilities grow exponentially with string length
- Identical subproblems repeat many times, based on trial and error
- DP reuses results of overlapping subproblems - Polynomial Time



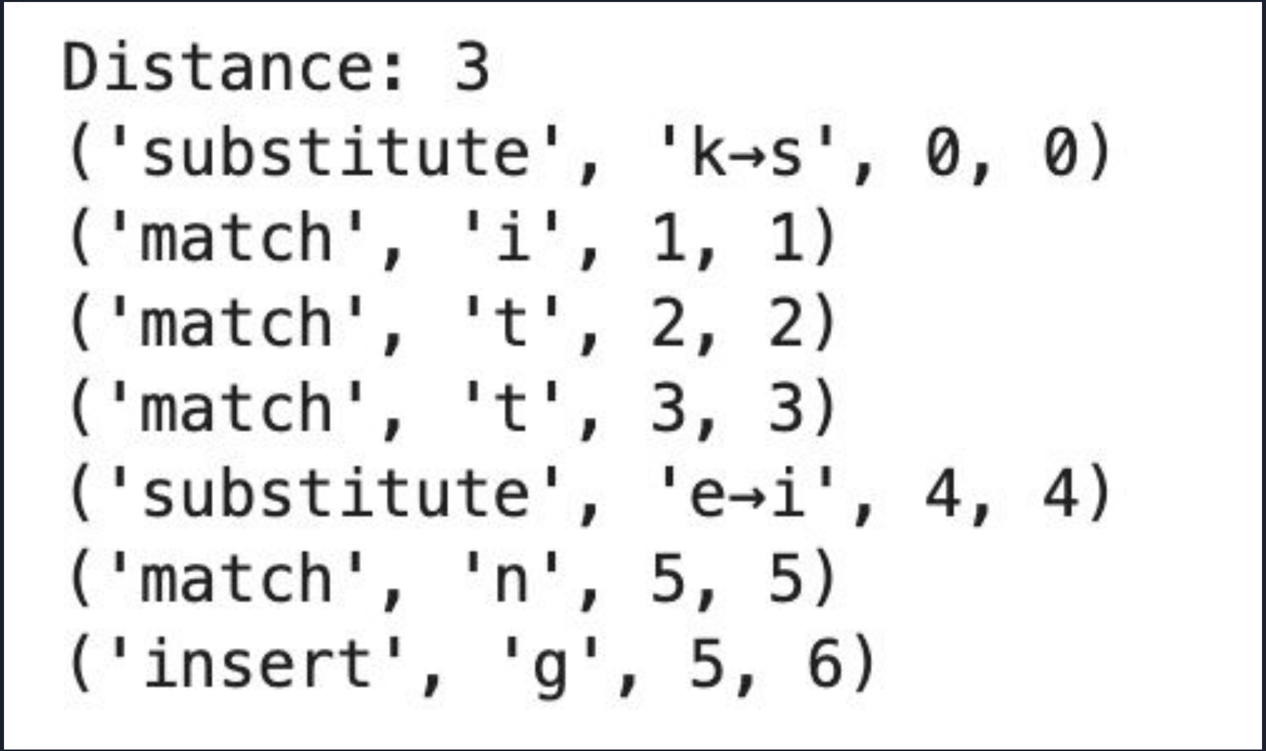
Libraries & Imports

- math – basic mathematical functions used in calculations
- time – for measuring performance
- numpy – efficient array operations and matrix handling for the DP table
- matplotlib.pyplot – plotting and visualization of the DP grid



```python
import math
import time
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
```

Operations Sequence



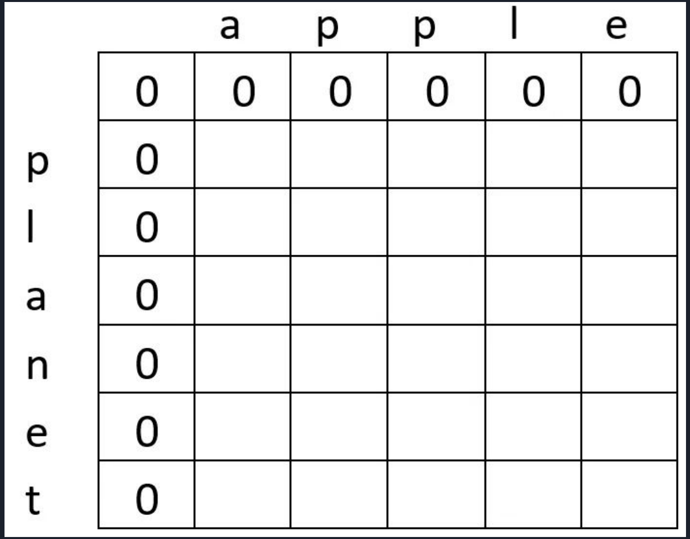
```
Distance: 3
('substitute', 'k→s', 0, 0)
('match', 'i', 1, 1)
('match', 't', 2, 2)
('match', 't', 3, 3)
('substitute', 'e→i', 4, 4)
('match', 'n', 5, 5)
('insert', 'g', 5, 6)
```

Initialization

1. Let m = length of string a and n = length of string b
2. Create a 2D matrix dp of size (m+1) × (n+1)
3. Fill the first column: dp[i][0] = i, cost of deleting i characters to match an empty string
4. Fill the first row: dp[0][j] = j, cost of inserting j characters to match string b of length j.

Defining the DP Table

- dp[i][j] = min edits to convert a[:i] → b[:j]
- Table size: (len(a)+1) × (len(b)+1)
- dp[0][j] = j → insert j characters
- dp[i][0] = i → delete i characters



		a	p	p	l	e
	0	0	0	0	0	0
p	0					
l	0					
a	0					
n	0					
e	0					
t	0					

DP Cell Dependencies

Each DP cell depends on three neighbors:

- Top (dp[i-1][j]): cost if we delete a character from string a.
- Left (dp[i][j-1]): cost if we insert a character into string a (to match b).
- Diagonal (dp[i-1][j-1]): cost if we substitute one character for another, or do nothing if they match.

Cases

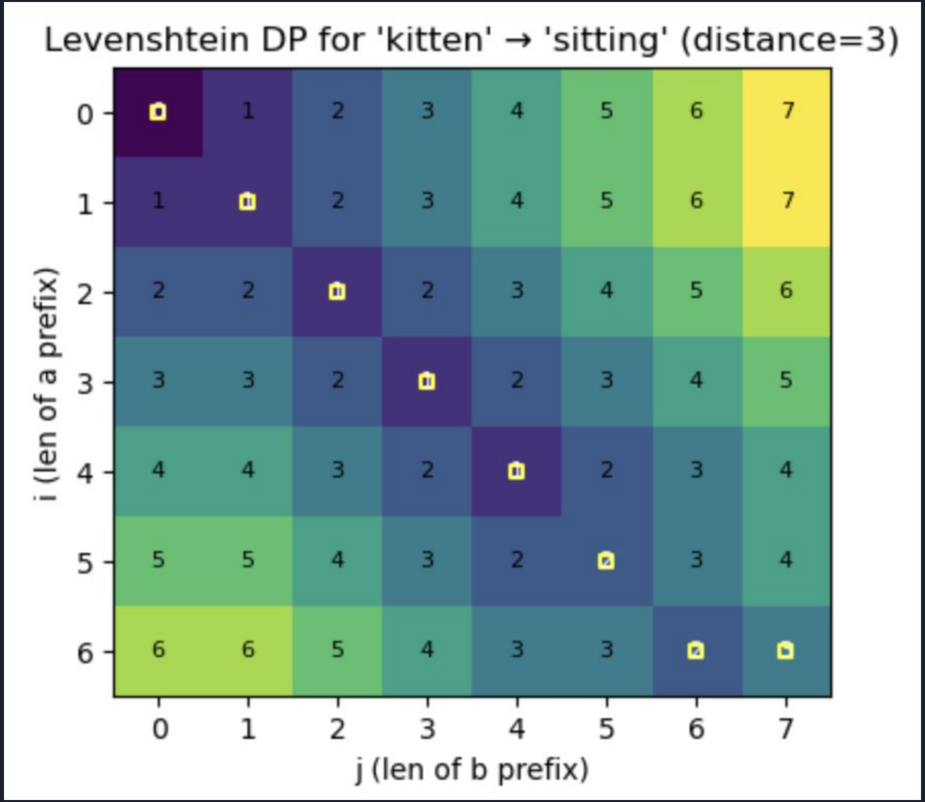
Match Case: a[i-1] == b[j-1]

- Results in no cost
- Copy diagonal: dp[i][j] = dp[i-1][j-1]

Mismatch Case: If a[i-1] != b[j-1]

- Must edi: insert, delete, or substitute
- dp[i][j] = 1 + min(top, left, diagonal)

Optimal Path Overlay

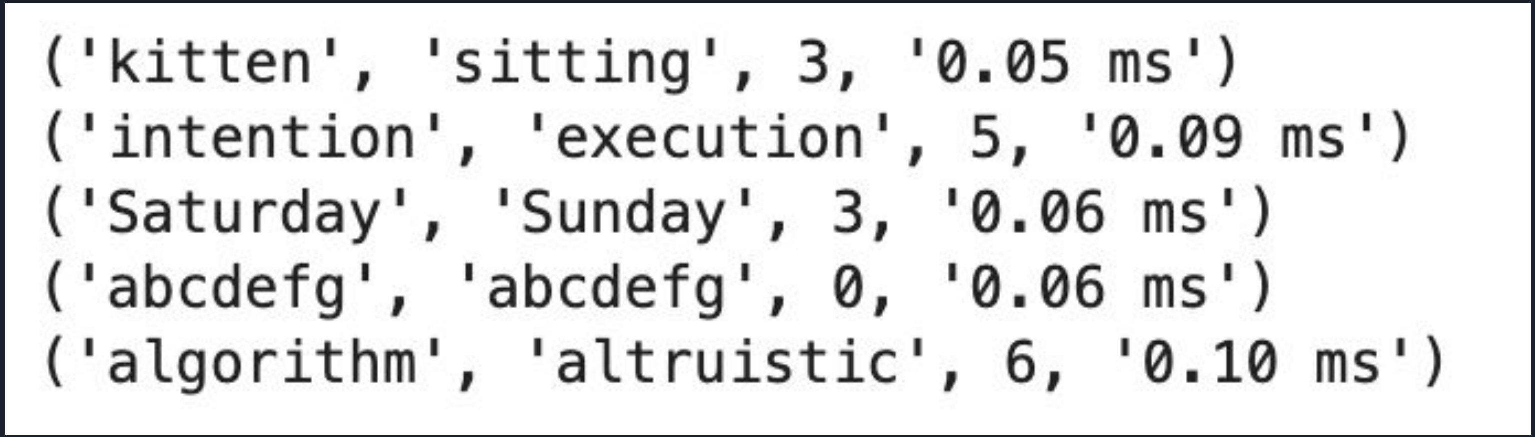


	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	1	1	2	3	4	5	6	7
2	2	2	1	2	3	4	5	6
3	3	3	2	1	2	3	4	5
4	4	4	3	2	1	2	3	4
5	5	5	4	3	2	2	3	4
6	6	6	5	4	3	3	2	3

Visualization Code

- Build a DP table for prefixes of the two strings.
- Fill cells using insert, delete, substitute rules.
- Bottom-right cell = final edit distance.
- Backtrack to reconstruct optimal sequence of edits.

Performance Measurement



```
('kitten', 'sitting', 3, '0.05 ms')
('intention', 'execution', 5, '0.09 ms')
('Saturday', 'Sunday', 3, '0.06 ms')
('abcdefg', 'abcdefg', 0, '0.06 ms')
('algorithm', 'altruistic', 6, '0.10 ms')
```

Demonstrates efficiency of Dynamic Programming, with all test cases running under or at 0.1 ms.

Limitations

Computational Cost:

- Time complexity and space complexity are O(m·n)
- Expensive when comparing strings with millions of characters

Lack of Context:

- Measures edit distance without being able to consider a string’s semantic meaning

Scope:

- When scaling to large databases, requires indexing or heuristics

Credits

Macroxela, Computing. Visualizing Dynamic Programming as Tables. 9 Apr. 2025. Medium, https://medium.com/@compuxela/visualizing-dynamic-programming-as-tables-d50f4c8567c8.

Team, Shutterstock. dna strand in scientific illustration. 17 Jan. 2025. Https://Www.Shutterstock.Com/Image-Photo/Dna-Strand-Scientific-Illustration-2573116321.

Team, Tiny. Spell Checker. Tiny, https://www.tiny.cloud/tinymce/features/spell-checker/.

Team, Study Glance. "Brute Force Pattern Matching." Study Glance, studyglance.in/ds/display.php?tno=40&topic=Brute-force-Pattern-Matching.