Intro

The “**Minimum Edit Distance**” is a measure of similarity between two strings. It represents the minimum number of single-character edits:

- *Insertions*
- *Deletions*
- *Substitutions*

These 3 are required to change one string into another. This concept is used alot in different types of fields, such as computational biology, spell checking, and what we are currently studying- natural language processing.

Dynamic Programming (DP) is a method for solving complex problems by breaking them down into smaller, simpler subproblems. It is useful when these small subproblems overlap, and when the problem itself has an optimal substructure. Dynamic Programming solutions typically involve creating a table to store the results of subproblems, which can then be used to construct the overall solution to the problem.

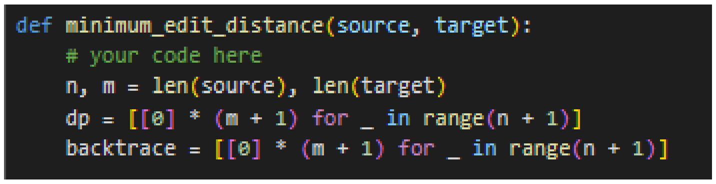
So how do they intertwine?

The **Minimum Edit Distance** problem is a perfect candidate for **Dynamic Programming** as it shows both overlapping subproblems and optimal substructure. By using a DP approach, we can calculate the edit distance between the two given strings by building a matrix where each cell represents the edit distance between prefixes of the two strings. This allows us to avoid redundant calculations and build up to the final solution in a **bottom-up** manner.

Name: Homssi, Yazan M.
Course || Section: NLP1000 || S15

How the Algorithm Works

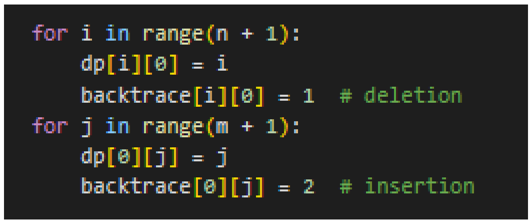
- We first initialize a 2D matrix `dp` where `dp[i][j]` will store the minimum edit distance between the first `i` characters of the source string and the first `j` characters of the target string.



```
def minimum_edit_distance(source, target):
    # your code here
    n, m = len(source), len(target)
    dp = [[0] * (m + 1) for _ in range(n + 1)]
    backtrace = [[0] * (m + 1) for _ in range(n + 1)]
```

Figure 1.1 (Matrix Initialization)

- Then initialize the first row & column of the matrix to represent the cost of inserting/deleting all characters.

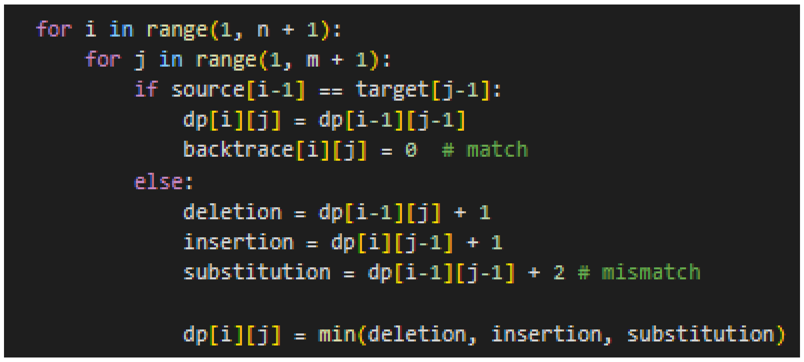


```
for i in range(n + 1):
    dp[i][0] = i
    backtrace[i][0] = 1  # deletion
for j in range(m + 1):
    dp[0][j] = j
    backtrace[0][j] = 2  # insertion
```

Figure 1.2 (Row and Column Initialization)

- For each cell (`i, j`), we calculate the minimum cost based on three possible operations:
    - *Match/Mismatch:* `dp[i-1][j-1] + (0 if chars match, 2 if they don't)`
    - *Insertion:* `dp[i][j-1] + 1`
    - *Deletion:* `dp[i-1][j] + 1`

Then choose the minimum of these three options for each cell.



```
for i in range(1, n + 1):
    for j in range(1, m + 1):
        if source[i-1] == target[j-1]:
            dp[i][j] = dp[i-1][j-1]
            backtrace[i][j] = 0  # match
        else:
            deletion = dp[i-1][j] + 1
            insertion = dp[i][j-1] + 1
            substitution = dp[i-1][j-1] + 2 # mismatch

            dp[i][j] = min(deletion, insertion, substitution)
```

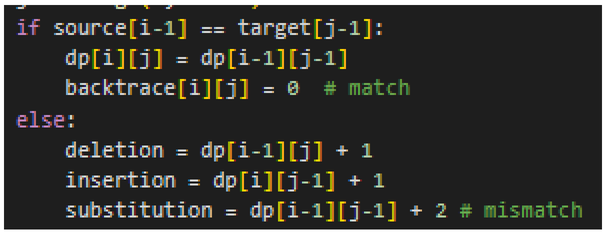
Figure 1.3 (Filling in the matrix)

Backtracking:

- *Create a separate* `backtrace` *matrix to keep track of which operation was chosen for each cell;*
- *After filling the* `dp` *matrix, start from the bottom-right cell and follow the backtrace to reconstruct the alignment; then*
- *Build the alignment strings by adding characters (or gaps) based on the operations in the backtrace.*

I would say that the main modification in this implementation is the use of a separate backtrace matrix. This is good for a more efficient reconstruction of the alignment without actually affecting the distance calculation. Plus, it makes it easier to prioritize certain operations when multiple optimal paths exist.

In the algorithm, substitution has a penalty of 2, while deletion and insertion both have a penalty of 1. Let's take a look at it here:



```
if source[i-1] == target[j-1]:
    dp[i][j] = dp[i-1][j-1]
    backtrace[i][j] = 0  # match
else:
    deletion = dp[i-1][j] + 1
    insertion = dp[i][j-1] + 1
    substitution = dp[i-1][j-1] + 2 # mismatch
```

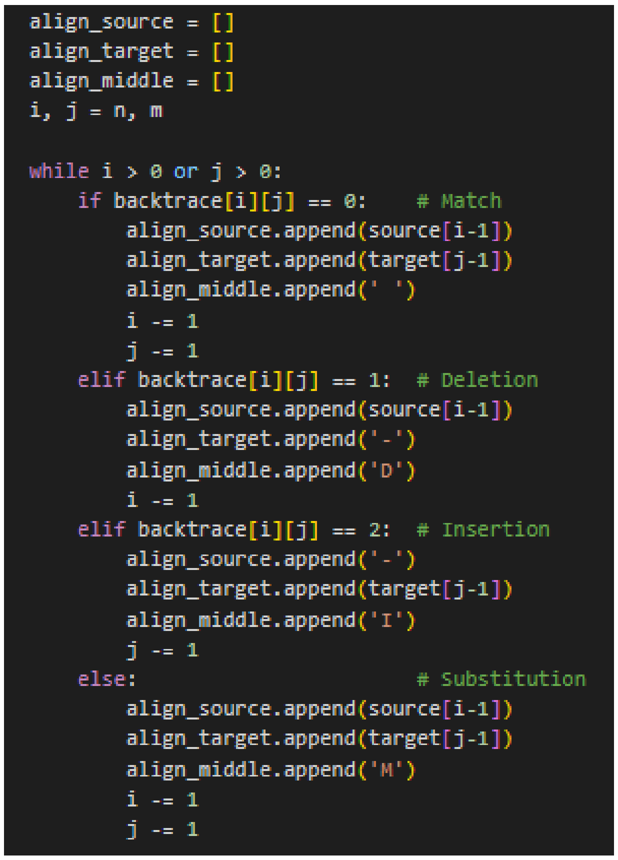
Figure 1.4 (Equal weight of Deletion & Insertion)

This means that the algorithm favors **deletion** and **insertion** over **substitution**. It treats deletion and insertion with the same weight (1), implying that removing a character from source(n) and adding a character to the target (m) is equally costly.

When it comes to **matching** characters though, the algorithm will assign 0 cost.

Operational Bias

The `align_source`, `align_target`, and `align_middle` arrays trace back through the operations to produce the final alignment. The algorithm is biased toward showing **deletions**, **insertions**, and **substitutions** explicitly in the alignment output (using 'D', 'I', and 'M'). This can help in visualizing the edit operations, but it also emphasizes differences rather than similarities.



```python
align_source = []
align_target = []
align_middle = []
i, j = n, m

while i > 0 or j > 0:
    if backtrace[i][j] == 0:    # Match
        align_source.append(source[i-1])
        align_target.append(target[j-1])
        align_middle.append(' ')
        i -= 1
        j -= 1
    elif backtrace[i][j] == 1:  # Deletion
        align_source.append(source[i-1])
        align_target.append('-')
        align_middle.append('D')
        i -= 1
    elif backtrace[i][j] == 2:  # Insertion
        align_source.append('-')
        align_target.append(target[j-1])
        align_middle.append('I')
        j -= 1
    else:                       # Substitution
        align_source.append(source[i-1])
        align_target.append(target[j-1])
        align_middle.append('M')
        i -= 1
        j -= 1
```

Figure 1.5 (Operational Bias in Alignment Output)

Weighting Scheme Analysis

The **1-1-2-0** weighting scheme (**1** for **insertion/deletion**, **2** for **mismatch**, **0** for **match**) is common, where it penalizes substitutions more heavily compared to insertions or deletions. The 1-1-2-0 scheme is often appropriate for tasks like spell checking or DNA sequence alignment. Although, the optimal weighting can depend on the specific application:

- *In biological applications, insertions/deletions might be more penalized than substitutions;*
- *In text processing, certain substitutions might be more likely than others (e.g., common typos).*

Sources? The slides 😎

When the 1-1-2-0 scheme might be appropriate:

- General Text Comparison: Comparing strings (spell-checking , string similarity).
  - Deletions and Insertions are equally important.
  - Substitutions cost more since they alter the meaning of the word or string itself.

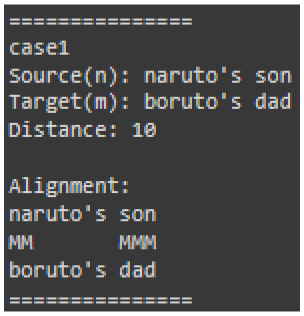
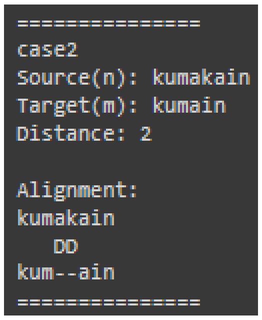
When it might not be ideal:

- DNA/Sequence Alignment in fields like bioinformatics.
  - This is unsure actually, insertions/deletions might carry different weights, while substitution might also need to be tweaked to reflect the biological cost of changes.
- Historical Documents
  - Insertions/Deletions may occur due to transcription errors that should carry lower penalties.
  - Substitutions may be more common and thus should have a lower penalty.

After analyzing the output for each test case, which made use of the **1-1-2-0 weighting scheme**, we could say that outputs are **as expected** based on our algorithm because:

1. *The algorithm will highly prioritizes matches (0 cost);*
2. *It prefers insertions/deletions (cost 1) over substitutions (cost 2) when a match isn't possible; and*
3. *The alignment shows the minimum number of operations needed to transform the source(n) into the target(m).*

Let's take a look at some test cases I find simple to explain, and will not clog this document:

 `===============` `case1` `Source(n): naruto's son` `Target(m): boruto's dad` `Distance: 10` `Alignment:` `naruto's son` `MM       MMM` `boruto's dad` `===============`	This alignment is expected because: • *Substitute "n" and "a" with "b" and "o" respectively (4)* • *The last 3 characters "son" should be substituted with "dad" (6)* • *Total: 2 + 2 + 0 + 2 + 2 + 2 = 10*
 `===============` `case2` `Source(n): kumakain` `Target(m): kumain` `Distance: 2` `Alignment:` `kumakain` `   DD` `kum--ain` `===============`	This alignment is expected because: • *The first 4 characters "kuma" match (0)* • *Delete 'k' (1)* • *Delete 'a' (1)* • *The last 2 characters "in" match (0)* • *Total: 0 + 1 + 1 + 0 = 2*

Reflection

This implementation of the **Minimum Edit Distance** algorithm using **Dynamic Programming** provides us with an efficient and flexible solution to the MP3 problem of string similarity and alignment. The use of a separate backtrace matrix makes it easy for reconstruction of the alignment & provides flexibility in prioritizing certain operations.

The algorithm's **time complexity** is *O(nm)*, where `n` and `m` are the lengths of the input strings, which is optimal for this problem. The **space complexity** is also O(nm) due to the two matrices used.