Date of Implementation: 26/06/2025
Date of Submission: 26/06/2025

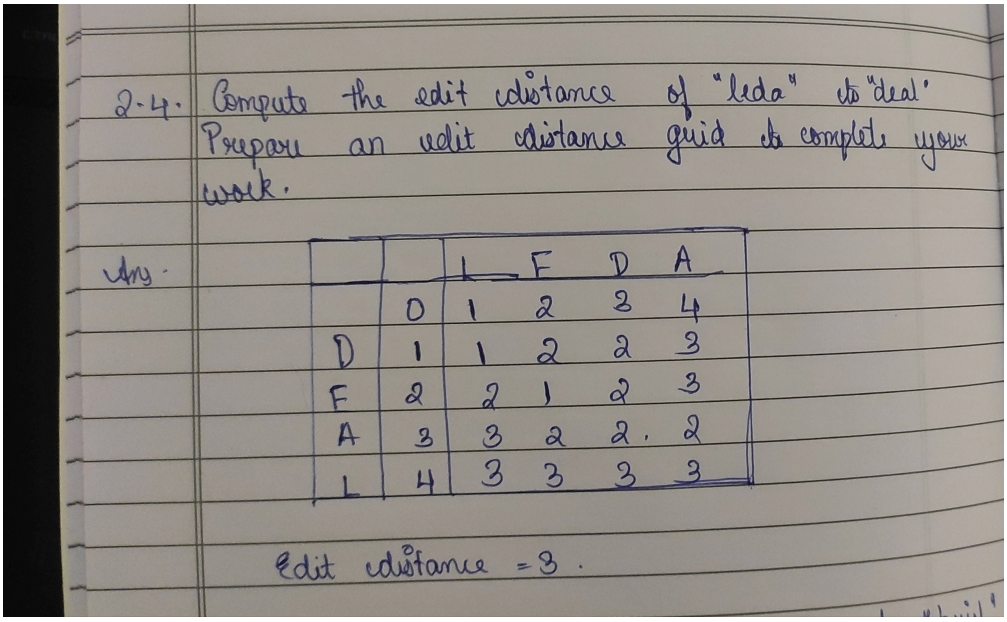
Lab 2: Edit Distance and Applications

Question 1. Edit distance (Manual)

2.4 Compute the edit distance (using insertion cost 1, deletion cost 1, substitution cost 1) of “leda” to “deal”. Prepare an edit distance grid to complete your work.

Program Description

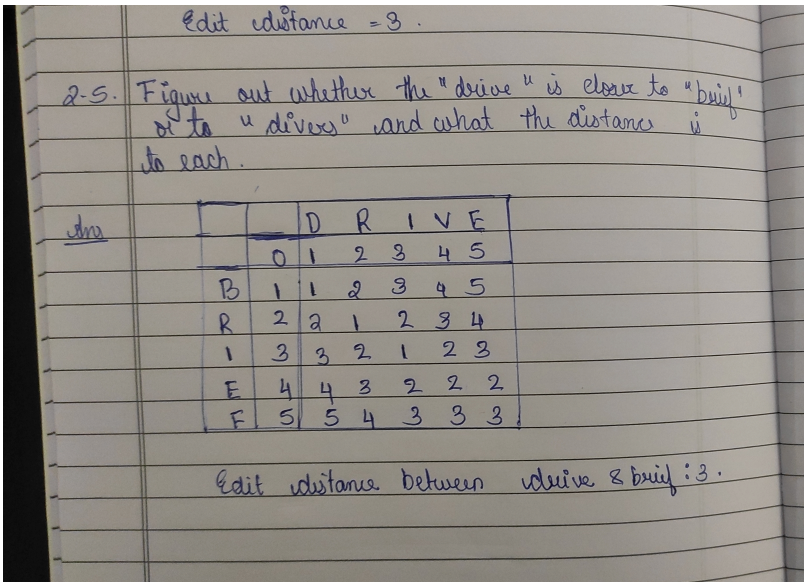
- Our aim is to compute the edit distance between two strings
- Program Logic
  - Create a matrix to hold distances.
  - Fill the first row and column with increasing numbers starting from zero.
  - Fill the matrix by comparing characters and applying edit operations.
  - The value in the matrix gives you the edit distance.

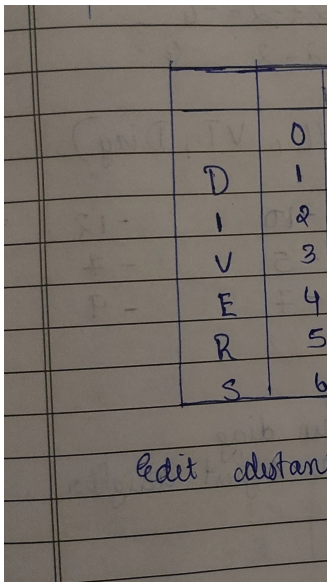


2.5 Figure out whether the “drive” is closer to “brief” or to “divers” and what the edit distance is to each. You may use any version of distance that you like. your work.

Program Description

- Using 'levenshtein distance' we compare the distance between drive and brief to drive and divers.
- Program Logic
  - Create a matrix to hold distances.
  - Fill the first row and column with increasing numbers starting from zero.
  - Fill the matrix by comparing characters and applying edit operations.
  - The value in the matrix gives you the edit distance.





Question 2. Edit distance (Implementation)

2.6 Implement a minimum edit distance algorithm and use your hand-computed results to check your code..

Program Description

- Our aim is to compute the edit distance between two strings

Program Logic:

- Libraries used: numpy
- FLow of the program
  - Create a matrix to hold distances.
  - Initialize the first row and column with increasing numbers.
  - Fill the matrix by comparing characters and applying edit operations.
  - Print the matrix after each row is filled.
  - Return the final edit distance.

Test Cases

- Case 1:
  - str1 = "apple"
  - str2 = "apple"
  - Expected Output: 0
- Case 2:
  - str1 = "apple"
  - str2 = "apples"
  - Expected Output: 1
- Case 3:
  - str1 = "abc"
  - str2 = "xyz"
  - Expected Output: 3
- Case 4:

- str1 = "kitten"
- str2 = "sitting"
- Expected Output: 3

In [5]:
```
import numpy as np
```

In [8]:
```
def levenshtein_distance(str1, str2):
    len1, len2 = len(str1),len(str2)
    mat = np.zeros((len1+1,len2+1), dtype = int)
    for i in range(0, len1+1):
        mat[i][0] = i
    for j in range(0, len2+1):
        mat[0][j] = j
    for i in range(1, len1+1):
        for j in range(1, (len2+1)):
            if str1[i-1] == str2[j-1]:
                mat[i][j] = mat[i-1][j-1]
            else:
                mat[i][j] = 1+min(mat[i-1][j-1], mat[i][j-1], mat[i-1][j])
        # print(f"Matrix after {i}th iteration: {mat}")


    return mat[len1][len2]
str1 = "leda"
str2 = "deal"
distance = levenshtein_distance(str1.lower(), str2.lower())
print(f"The edit distance between '{str1}' and '{str2}' is {distance}")
```

```
The edit distance between 'leda' and 'deal' is 3
```

In [ ]:
```
test_cases = [
    ("apple", "apple", 0),
    ("apple", "apples", 1),
    ("abc", "xyz", 3),
    ("kitten", "sitting", 3),
]
for s1, s2, expected in test_cases:
    result = levenshtein_distance(s1, s2)
    print(f"levenshtein_distance('{s1}', '{s2}') = {result} | Expected: {expecte
```

```
levenshtein_distance('apple', 'apple') = 0 | Expected: 0 | True
levenshtein_distance('apple', 'apples') = 1 | Expected: 1 | True
levenshtein_distance('abc', 'xyz') = 3 | Expected: 3 | True
levenshtein_distance('kitten', 'sitting') = 3 | Expected: 3 | True
```

Question 3. Implement Sequence Alignment

Write a program to align the given sequence of input text A and B

Program Description:

Global alignment is a fundamental concept in bioinformatics and string comparison, used to align two sequences (e.g., DNA, RNA, protein, or text strings) from start to end in their entirety. The goal is to find the best possible match between the two sequences over their full length, even if that requires inserting gaps.

Program Logic:

- Libraries used: `numpy`
- Flow of the program:
  - Create a matrix (`mat`) to store alignment scores using `numpy.zeros` .
  - Initialize the **first row** and **first column** of the matrix with cumulative gap penalties (−2 per gap).
  - Iterate through the matrix and compute scores for each cell based on:
    - Diagonal move (match/mismatch):
      - Add `+1` if characters match,
      - Subtract `1` if characters mismatch.
    - Top move (gap in `seq2`): subtract `2`
    - Left move (gap in `seq1`)**: subtract `2`
    - Assign the maximum of these three values to the current cell.
  - After the matrix is filled, backtrack from the bottom-right to the top-left:
    - If characters match or mismatch, take diagonal.
    - If current cell came from top, insert gap in `seq2` .
    - If from left, insert gap in `seq1` .
  - Handle any remaining characters in `seq1` or `seq2` by aligning them with gaps.
  - Return the two aligned sequences (`res1` , `res2`) with inserted gaps where needed.

Test Cases

- Case 1:
  - seq1 = "ACTG"
  - seq2 = "ACTG"
  - Expected Output:
    ACTG
    ACTG
- Case 2:
  - seq1 = "ACG"
  - seq2 = "ACTG"
  - Expected Output:
    A-CG
    ACTG
- Case 3:
  - seq1 = "ACTG"
  - seq2 = "ACG"
  - Expected Output:
    ACTG

A-CG

- Case 4:
  - seq1 = "AAA"
  - seq2 = "TTT"
  - Expected Output:
    AAA
    TTT

- Case 5:
  - seq1 = ""
  - seq2 = "ACTG"
  - Expected Output:
    ACTG
- Case 6:
  - seq1 = "GATTACA"
  - seq2 = "TACTAGA"
  - Expected Output:
    GATTACA
    TACTAGA

- Case 7:
  - seq1 = "AGGCTATCACCTGACCTCCAGGCCGATGCCC"
  - seq2 = "TAGCTATCACGACCGCGGTCGATTTGCCCGAC"
  - Expected Output:
    -AGGCTATCACCTGACCTCCAGGCCGA--TGCCC---
    TAG-CTATCAC--GACCGC--GGTCGATTTGCCCGAC

In [ ]:
```
def alignment(seq1, seq2):
    len1, len2 = len(seq1), len(seq2)
    mat = np.zeros((len1+1, len2+1), dtype=int)
    for i in range(1, len1+1):
        mat[i][0] = mat[i-1][0]-2
    for j in range(1, len2+1):
        mat[0][j] = mat[0][j-1]-2

    for i in range(1, len1+1):
        for j in range(1, (len2+1)):
            top_value = mat[i-1][j]-2
            left_value = mat[i][j-1]-2
            diag_value = mat[i-1][j-1]
            if seq1[i-1] == seq2[j-1]:
                diag_value += 1
            else:
                diag_value -= 1
            mat[i][j] = max(top_value, left_value, diag_value)
    res1 = ""
    res2 = ""
```

```
    match=1
    mismatch=-1
    gap=-2
    i, j = len1, len2
    while i > 0 and j > 0:
        if seq1[i - 1] == seq2[j - 1]:
            score_diag = match
        else:
            score_diag = mismatch

        if mat[i][j] == mat[i - 1][j - 1] + score_diag:
            res1 = seq1[i - 1] + res1
            res2 = seq2[j - 1] + res2
            i -= 1
            j -= 1
        elif mat[i][j] == mat[i - 1][j] + gap:
            res1 = seq1[i - 1] + res1
            res2 = "-" + res2
            i -= 1
        else:
            res1 = "-" + res1
            res2 = seq2[j - 1] + res2
            j -= 1

    # Finish remaining
    while i > 0:
        res1 = seq1[i - 1] + res1
        res2 = "-" + res2
        i -= 1

    while j > 0:
        res1 = "-" + res1
        res2 = seq2[j - 1] + res2
        j -= 1

    return res1, res2
seq1 = "AGGCTATCACCTGACCTCCAGGCCGATGCCC"
seq2 = "TAGCTATCACGACCGCGGTCGATTTGCCCGAC"

result1, result2 = alignment(seq1, seq2)
print(f"The Sequence alignment for {seq1} and {seq2} are:\n {result1}, {result2}")
```

```
The Sequence alignment for AGGCTATCACCTGACCTCCAGGCCGATGCCC and TAGCTATCACGACCGCGG
TCGATTTGCCCGAC are:
 AGGCTATCACCTGACCTCCAGGCCGA--TG-CC--C, TAGCTATCA-C-GACC-GC-GGTCGATTTGCCCGAC
```

In [13]:

```
test_cases = [
    ("ACTG", "ACTG"),
    ("ACTG", "ACCG"),
    ("ACG", "ACTG"),
    ("ACTG", "ACG"),
    ("AAA", "TTT"),
    ("GATTACA", "TACTAGA"),
    ("", "ACTG"),
    ("AGGCTATCACCTGACCTCCAGGCCGATGCCC", "TAGCTATCACGACCGCGGTCGATTTGCCCGAC")
]

for s1, s2 in test_cases:
    res1, res2 = alignment(s1, s2)
    print(f"Alignment for '{s1}' and '{s2}':\n{res1}\n{res2}\n{'-'*40}")
```

```
Alignment for 'ACTG' and 'ACTG':
ACTG
ACTG
----------------------------------------
Alignment for 'ACTG' and 'ACCG':
ACTG
ACCG
----------------------------------------
Alignment for 'ACG' and 'ACTG':
AC-G
ACTG
----------------------------------------
Alignment for 'ACTG' and 'ACG':
ACTG
AC-G
----------------------------------------
Alignment for 'AAA' and 'TTT':
AAA
TTT
----------------------------------------
Alignment for 'GATTACA' and 'TACTAGA':
GATTACA
TACTAGA
----------------------------------------
Alignment for '' and 'ACTG':
----
ACTG
----------------------------------------
Alignment for 'AGGCTATCACCTGACCTCCAGGCCGATGCCC' and 'TAGCTATCACGACCGCGGTCGATTTGCC
CGAC':
AGGCTATCACCTGACCTCCAGGCCGA--TG-CC--C
TAGCTATCA-C-GACC-GC-GGTCGATTTGCCCGAC
----------------------------------------
```

In [ ]: