Algorithm Project

Team Members / Section 2

1000287803	احمد محمد مسعد الشافعي
1000287783	احمد محمد محمود بدوي
1000287763	احمد محمد شاكر لاظ
1000288289	احمد محمد احمد ابراهيم بيصار

---

1. Problem Description

The **Edit Distance problem** computes the minimum number of operations required to transform one string into another.
The allowed operations are:

- Insert a character
- Delete a character
- Replace a character

This problem is widely used in spell checking, text correction, and bioinformatics.

2.Algorithms Used

2.1 Naive Recursive Algorithm

A pure recursive solution that explores all possible edit operations.
It recomputes the same subproblems many times, making it inefficient for large inputs.

```python
# Edit Distance Naive

import time

def Edit_Distance_Naive(w1, w2, m, n):

    if m == 0:
        return n # If w1 is empty
    if n == 0:
        return m # If w2 is empty

    if w1[m-1] == w2[n-1]: # If characters are the same, ignore them
        return Edit_Distance_Naive(w1, w2, m-1, n-1)

    return 1 + min(
        Edit_Distance_Naive(w1, w2, m, n-1),     # Insert
        Edit_Distance_Naive(w1, w2, m-1, n),     # Delete
        Edit_Distance_Naive(w1, w2, m-1, n-1)    # Replace
    )
```

```
Algorithm EditDistanceNaive(w1, w2, m, n)

Input:
    w1, w2 : strings
    m : length of w1
    n : length of w2

Output:
    Minimum edit distance between w1 and w2

Begin
    if m == 0 then
        return n
    if n == 0 then
        return m

    if w1[m-1] == w2[n-1] then
        return EditDistanceNaive(w1, w2, m-1, n-1)

    return 1 + minimum(
        EditDistanceNaive(w1, w2, m, n-1),      // Insert
        EditDistanceNaive(w1, w2, m-1, n),      // Delete
        EditDistanceNaive(w1, w2, m-1, n-1)     // Replace
    )
End
```

- **Time Complexity:** `O(3^(m+n))`
- **Space Complexity:** `O(m+n)`

```
execution -> intention
- Naive Result:
- Distance: 5
- Time Complexity: O(3^(m+n))
- Space Complexity: O(m + n)
- Time: 0.00015593 seconds
```

2.2 Optimized Algorithm

Uses dynamic programming and stores only two rows of the DP table to reduce memory usage.

```python
# Edit Distance Optimized

import time

def Edit_Distance_Optimized(word1, word2):
    m, n = len(word1), len(word2)

    prev = list(range(n + 1)) # Previous row
    curr = [0] * (n + 1) # Current row

    for i in range(1, m + 1):
        curr[0] = i # If word2 is empty
        for j in range(1, n + 1):
            if word1[i - 1] == word2[j - 1]: # If characters are the same, ignore them
                curr[j] = prev[j - 1]
            else:
                curr[j] = 1 + min(
                    prev[j],     # Insert
                    curr[j - 1], # Delete
                    prev[j - 1]  # Replace
                )
        prev, curr = curr, prev  # Swap rows

    return prev[n]
```

```
Algorithm EditDistanceOptimized(word1, word2)

Input:
    word1, word2 : strings

Output:
    Minimum edit distance between word1 and word2

Begin
    m ← length(word1)
    n ← length(word2)

    prev ← array of size n+1
    curr ← array of size n+1

    for j from 0 to n do
        prev[j] ← j

    for i from 1 to m do
        curr[0] ← i
        for j from 1 to n do
            if word1[i-1] == word2[j-1] then
                curr[j] ← prev[j-1]
            else
                curr[j] ← 1 + minimum(
                    prev[j],      // Insert
                    curr[j-1],    // Delete
                    prev[j-1]     // Replace
                )
        swap(prev, curr)

    return prev[n]
End
```

- **Time Complexity:** `O(m × n)`
- **Space Complexity:** `O(n)`

```
execution -> intention
- Optimized Result:
- Distance: 5
- Time Complexity: O(m * n)
- Space Complexity: O(n)
- Time: 0.00001836 seconds
```

2.3 Optimized Algorithm with Backtracking

Uses a full DP table and backtracking to display the sequence of edit operations.

```python
# Edit Distance Optimized with Backtracking

import time

def edit_distance_with_Backtracking(word1, word2):
    m, n = len(word1), len(word2)

    dp = [[0] * (n + 1) for _ in range(m + 1)] # DP table

    for i in range(m + 1):
        dp[i][0] = i # If word2 is empty
    for j in range(n + 1):
        dp[0][j] = j # If word1 is empty

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if word1[i - 1] == word2[j - 1]: # If characters are the same, ignore them
                dp[i][j] = dp[i - 1][j - 1]
            else:
                dp[i][j] = 1 + min(
                    dp[i - 1][j],     # Insert
                    dp[i][j - 1],     # Delete
                    dp[i - 1][j - 1]  # Replace
                )

    steps = []
    i, j = m, n
    while i > 0 or j > 0:
        if i > 0 and j > 0 and word1[i - 1] == word2[j - 1]: # Characters are the same
            i -= 1
            j -= 1
        elif i > 0 and j > 0 and dp[i][j] == dp[i - 1][j - 1] + 1: # Replace
            steps.append(f"Replace '{word2[j-1]}' with '{word1[i-1]}'") #Display replace operation
            i -= 1
            j -= 1
        elif i > 0 and dp[i][j] == dp[i - 1][j] + 1: # Insert
            steps.append(f"Insert '{word1[i-1]}'") #Display insert operation
            i -= 1
        else: # Delete
            steps.append(f"Delete '{word2[j-1]}'") #Display delete operation
            j -= 1

    steps.reverse()
    return dp[m][n], steps
```

```
Algorithm EditDistanceWithBacktracking(word1, word2)

Input:
    word1, word2 : strings

Output:
    Edit distance and list of operations

Begin
    m ← length(word1)
    n ← length(word2)

    create DP table dp[m+1][n+1]

    for i from 0 to m do
        dp[i][0] ← i
    for j from 0 to n do
        dp[0][j] ← j

    for i from 1 to m do
        for j from 1 to n do
            if word1[i-1] == word2[j-1] then
                dp[i][j] ← dp[i-1][j-1]
            else
                dp[i][j] ← 1 + minimum(
                    dp[i-1][j],     // Insert
                    dp[i][j-1],     // Delete
                    dp[i-1][j-1]    // Replace
                )

    operations ← empty list
    i ← m, j ← n

    while i > 0 or j > 0 do
        if i > 0 and j > 0 and word1[i-1] == word2[j-1] then
            i ← i-1, j ← j-1
        else if i > 0 and j > 0 and dp[i][j] == dp[i-1][j-1] + 1 then
            add "Replace" to operations
            i ← i-1, j ← j-1
        else if i > 0 and dp[i][j] == dp[i-1][j] + 1 then
            add "Insert" to operations
            i ← i-1
        else
            add "Delete" to operations
            j ← j-1

    reverse operations
    return dp[m][n], operations
End
```

- **Time Complexity:** `O(m × n)`
- **Space Complexity:** `O(m × n)`

```
execution -> intention
- Optimized with Backtracking Result:
- Distance: 5
- Operations:
- Replace 'e' with 'i'
- Replace 'x' with 'n'
- Replace 'e' with 't'
- Replace 'c' with 'e'
- Replace 'u' with 'n'
- Time Complexity: O(m * n)
- Space Complexity: O(m * n)
- Time: 0.00002694 seconds
```

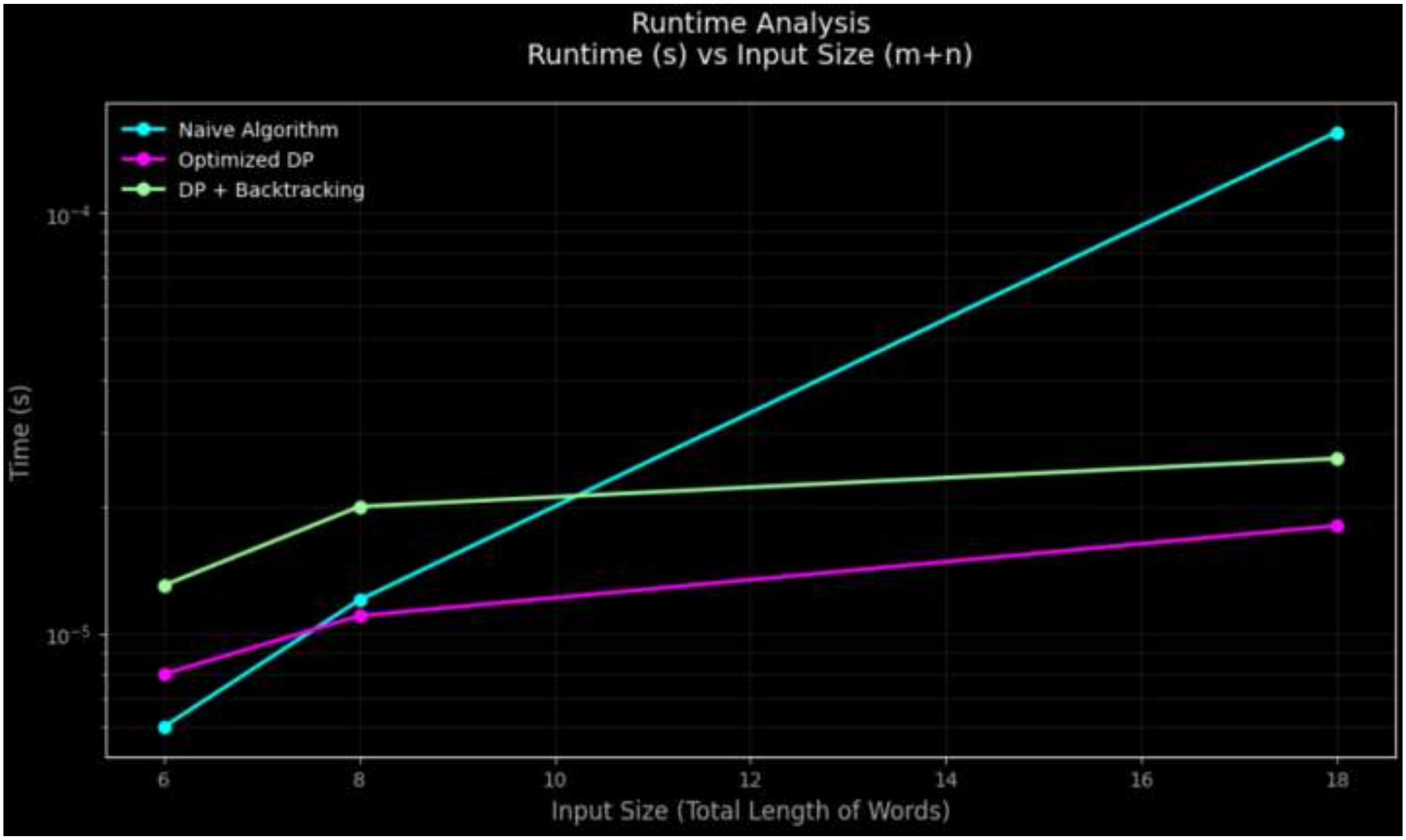
3.Empirical Time Analysis

All algorithms were executed on the same machine using Python.

Execution Time Comparison

word2 → word1	Naive Time (s)	Optimized Time (s)	Optimized + Backtracking (s)
cat → cut	0.000006	0.000008	0.000013
book → back	0.000012	0.000011	0.000020
execution → intention	0.000155	0.000018	0.000026

The naive algorithm was **not executed** for large inputs because its exponential time makes it impractical.



3.1 Naive Recursive Algorithm

- **Simple Explanation: This is a "slow" approach because it repeats the same calculations many times.**
- **Observation: It works fine for short words, but for longer words (like "execution"), the time jumps significantly to 0.000155s.**
- **Graph Result: The blue line in the graph rises very steeply, showing it is not practical for large data.**

3.2 Optimized DP Algorithm

- **Simple Explanation: This is a "smart and fast" approach because it saves results and never repeats a calculation.**
- **Observation: Even with long words, the time stays very low and stable at around 0.000018s.**
- **Graph Result: The purple line stays low and almost flat, proving it is the most efficient for speed and memory.**

3.3 Optimized DP with Backtracking

- **Simple Explanation: It is fast like the optimized version, but it does "extra work" to show you the exact steps (Insert, Delete, Replace).**
- **Observation: Because of this extra work, it is slightly slower (0.000026s) than the simple optimized version.**
- **Graph Result: The green line is always slightly above the purple line, showing the small "time cost" for getting more details.**

The empirical results and the graph prove that **Dynamic Programming (DP)** is the only practical solution. It transforms the algorithm from being very slow (Naive) into a highly efficient and fast solution, no matter how long the words are.

---

4. Discussion

- The naive recursive algorithm works only for very small inputs.
- The optimized dynamic programming algorithm is extremely fast and scalable.
- The backtracking version is slightly slower due to extra memory usage, but provides detailed transformation steps.

The empirical results clearly match the theoretical complexity analysis.

---

5. Conclusion

This project demonstrates how algorithm optimization dramatically improves performance.
While the naive solution illustrates the basic concept, dynamic programming provides a practical and efficient solution suitable for real-world applications.
