-

Indian Statistical Institute (ISI), Bangalore

The Levenshtein Distance Problem

Analysis and Implementation of Various Algorithms and their Applications

while(true);

Adrija Chatterjee and Suryansh Shirbhate

P29 of the Design and Analysis of Algorithms Class Project under Dr. Jaya Sreevalsan-Nair

October 19, 2025

Theoretical Statistics and Mathematics Unit (SMU)
Indian Statistical Institute, Bangalore

Contents

1 Introduction

1.1 Definition

In computer science and computational linguistics, the **Levenshtein distance** is a metric for measuring the difference between two sequences (usually strings). Let Σ be a finite alphabet and let Σ^* denote the set of all finite strings over Σ. For a string $s \in \Sigma^*$ we denote its length by $|s|$. A single *edit operation* on a string is one of:

- insertion of a single character,
- deletion of a single character,
- substitution (replacement) of one character by another.

The *Levenshtein distance* between two strings $x, y \in \Sigma^*$, denoted $d(x, y)$ or $\mathrm{lev}(x, y)$, is the minimum number of single-character edit operations required to transform x into y. This is also known as the (classic) edit distance.

1.2 Basic properties

We prove that (Σ^*, d) is a metric space by verifying the metric axioms.

Proposition 1.1 (Non-negativity and identity of indiscernibles)**.** *For all $x, y \in \Sigma^*$,*

$$d(x, y) \geq 0,$$

and

$$d(x, y) = 0 \iff x = y.$$

Proof. By definition $d(x, y)$ counts the minimum number of edit operations required to transform x into y. Hence $d(x, y)$ is an integer and $d(x, y) \geq 0$.

If $x = y$ then no edits are required so $d(x, y) = 0$. Conversely, if $d(x, y) = 0$ then the minimum number of edits to transform x into y is zero, which means x is already identical to y; therefore $x = y$. □

Proposition 1.2 (Symmetry)**.** *For all $x, y \in \Sigma^*$, $d(x, y) = d(y, x)$.*

Proof. Every edit operation has an inverse operation of the same cost: an insertion in one direction corresponds to a deletion in the other direction, and a substitution is its own inverse (replace a by b vs. replace b by a). If there exists a sequence of k edits converting x to y, applying the inverse operations in reverse order converts y to x with exactly k edits. Therefore the minimal number of edits from x to y equals the minimal number from y to x, i.e. $d(x, y) = d(y, x)$. □

Proposition 1.3 (Triangle inequality)**.** *For all $x, y, z \in \Sigma^*$,*

$$d(x, y) \leq d(x, z) + d(z, y).$$

Proof. Let p be a sequence of $d(x, z)$ edit operations that transforms x into z, and let q be a sequence of $d(z, y)$ edits that transforms z into y. Concatenating p followed by q produces a sequence of $d(x, z) + d(z, y)$ edits that transforms x into y. By minimality of $d(x, y)$ we therefore have

$$d(x, y) \leq |p| + |q| = d(x, z) + d(z, y),$$

which proves the triangle inequality. □

1.3 Useful Bounds

Proposition 1.4 (Length bounds)**.** *For all* $x, y \in \Sigma^*$*,*

$$\Big|\,|x| - |y|\,\Big| \le d(x,y) \le \max\{|x|, |y|\}.$$

Proof. The lower bound: each deletion or insertion changes string length by exactly one, while substitution does not change length. Hence to change $|x|$ to $|y|$ one needs at least $\Big||x| - |y|\Big|$ insertions/deletions, so $d(x,y) \ge \Big||x| - |y|\Big|$.

The upper bound: convert the longer string to the empty string (at cost equal to its length), then convert the empty string to the shorter string (cost equal to the shorter string's length). The total cost is at most $\max\{|x|, |y|\}$; a tighter trivial upper bound is $|x| + |y|$, but the max bound is often useful. □

This metric is widely used in *spell checking*, *natural language processing*, *bioinformatics* (e.g., DNA sequence alignment), and other applications where quantifying string similarity is important. In this report, we explore three algorithms for computing the Levenshtein distance efficiently: the Wagner–Fischer dynamic programming algorithm, Hirschberg's divide-and-conquer method, and Ukkonen's bounded edit distance algorithm. This report documents the implementation and analysis of three classical Levenshtein distance algorithms:

- Wagner–Fischer algorithm (Dynamic Programming)
- Hirschberg algorithm (Divide and Conquer)
- Ukkonen algorithm (Bounded Levenshtein Distance)

Implementation of the above algorithms in Python is given here.

2 Wagner-Fischer Algorithm

2.1 Overview

The Wagner-Fischer algorithm (1974) [WF74] is the classical dynamic programming formulation for computing the *Levenshtein edit distance* between two strings. It uses the same scoring principles as the Needleman-Wunsch algorithm, but interprets them in terms of *edit operations* rather than alignment scores. Specifically, it assigns a cost of 0 for matching characters, and a positive cost (usually 1) for each insertion, deletion, or substitution. The goal is to minimize the total cost of transforming one string into another. This approach has a time and space complexity of $\mathcal{O}(mn)$, where m and n are the lengths of the input strings.

2.2 Explanation

Let $D(i,j)$ denote the minimum edit distance between the prefixes $x_1x_2\ldots x_i$ and $y_1y_2\ldots y_j$. Then the recurrence relation is:

$$D(i,j) = \min \begin{cases} D(i-1,j) + 1, & \text{(deletion)} \\ D(i,j-1) + 1, & \text{(insertion)} \\ D(i-1,j-1) + \mathbf{1}_{\{x_i \neq y_j\}}, & \text{(match or substitution).} \end{cases}$$

Here $\mathbf{1}_{\{x_i \neq y_j\}}$ equals 0 if the characters match and 1 otherwise.

The Wagner-Fischer algorithm fills an $(m+1) \times (n+1)$ matrix row by row (or column by column), where the entry $D(i, j)$ represents the minimum edit cost to convert $x_1 \dots x_i$ into $y_1 \dots y_j$. The final value $D(m, n)$ gives the minimum number of edits. Although straightforward and exact, its quadratic memory makes it impractical for long genomic sequences, motivating space-efficient or "nearby" variants such as Hirschberg's and Ukkonen's algorithms.

2.3 Pseudocode

```
# Compute Levenshtein distance with detailed operation log and transformations

function wagner_fischer_with_log(s[1..m], t[1..n]) -> dict:
    # Initialize matrices
    D[0..m, 0..n]; choice[0..m, 0..n]
    for i <- 0..m:
        D[i,0] <- i
        choice[i,0] <- "start" if i == 0 else "delete"
    for j <- 0..n:
        D[0,j] <- j
        choice[0,j] <- "start" if j == 0 else "insert"

    # Fill DP table with tie-breaking: diag > delete > insert
    for i <- 1..m:
        for j <- 1..n:
            cost_diag <- D[i-1,j-1] + (0 if s[i] == t[j] else 1)
            cost_del  <- D[i-1,j] + 1
            cost_ins  <- D[i,j-1] + 1
            best <- min(cost_diag, cost_del, cost_ins)
            D[i,j] <- best

            if best == cost_diag:
                choice[i,j] <- "match" if s[i] == t[j] else "substitute"
            elif best == cost_del:
                choice[i,j] <- "delete"
            else:
                choice[i,j] <- "insert"

    # Backtrack to get operations in reverse
    i <- m; j <- n; rev_ops <- []
    while i > 0 or j > 0:
        op <- choice[i,j]
        if op == "match":
            append {"op":"match", "pos":i-1, "char":s[i]} to rev_ops
            i <- i-1; j <- j-1
        elif op == "substitute":
```

```
            append {"op":"substitute", "pos":i-1, "from":s[i],
    ↪ "to":t[j]} to rev_ops
            i <- i-1; j <- j-1
        elif op == "delete":
            append {"op":"delete", "pos":i-1, "char":s[i]} to
    ↪ rev_ops
            i <- i-1
        elif op == "insert":
            append {"op":"insert", "pos":i, "char":t[j]} to
    ↪ rev_ops
            j <- j-1
        else: break

    ops <- reverse(rev_ops)

    # Adjust original positions as edits are applied sequentially
    function adjusted_pos(original_pos, applied_ops) -> int:
        shift <- 0
        for a in applied_ops:
            p <- a["pos"]
            if a["op"] == "insert" and p <= original_pos:
                shift <- shift + 1
            elif a["op"] == "delete" and p < original_pos:
                shift <- shift - 1
        return original_pos + shift

    # Apply operations to reconstruct transformations
    cur <- list(s)
    transformations <- [join(cur)]
    applied_ops <- []

    for action in ops:
        typ <- action["op"]
        if typ == "match":
            pos <- adjusted_pos(action["pos"], applied_ops)
            if 0 <= pos < len(cur):
                append {"op":"match", "pos":pos,
    ↪ "char":cur[pos]} to applied_ops
            else:
                append {"op":"match", "pos":pos, "char":None} to
    ↪ applied_ops
            transformations.append(join(cur))

        elif typ == "substitute":
            orig_pos <- action["pos"]
            pos <- adjusted_pos(orig_pos, applied_ops)
            to_char <- action["to"]
            if 0 <= pos < len(cur):
                old <- cur[pos]
                cur[pos] <- to_char
```

```
                append {"op":"substitute", "pos":pos,
↪ "from":old, "to":to_char} to applied_ops
            else:
                append {"op":"substitute", "pos":pos,
↪ "from":None, "to":to_char} to applied_ops
            transformations.append(join(cur))

        elif typ == "delete":
            orig_pos <- action["pos"]
            pos <- adjusted_pos(orig_pos, applied_ops)
            if 0 <= pos < len(cur):
                removed <- cur[pos]
                remove cur[pos]
                append {"op":"delete", "pos":pos,
↪ "char":removed} to applied_ops
            else:
                append {"op":"delete", "pos":pos, "char":None}
↪ to applied_ops
            transformations.append(join(cur))

        elif typ == "insert":
            orig_pos <- action["pos"]
            pos <- adjusted_pos(orig_pos, applied_ops)
            if pos < 0: pos <- 0
            if pos > len(cur): pos <- len(cur)
            insert action["char"] at cur[pos]
            append {"op":"insert", "pos":pos,
↪ "char":action["char"]} to applied_ops
            transformations.append(join(cur))

        else:
            transformations.append(join(cur))

    return {
        "distance": D[m,n],
        "D": D,
        "choice": choice,
        "ops": applied_ops,
        "transformations": transformations
    }
```

Listing 1: Wagner–Fischer algorithm with position-adjusted operation log and transformations

2.4 Python Implementation

Implementation of Wagner Fischer in Python

2.5 Wagner–Fischer Algorithm: Invariant and Correctness Proof

Let A and B be two strings of lengths m and n respectively, and let $d(i, j)$ denote the Levenshtein distance between prefixes $A[1..i]$ and $B[1..j]$. The Wagner–Fischer algorithm computes $d(i, j)$ iteratively using dynamic programming.

2.5.1 Invariant

After completion of iteration i of the outer loop (i.e. after computing the i-th row of the DP table), the following invariant holds:

Invariant (W): For all $0 \le p \le i$ and $0 \le q \le n$,

$$D[p, q] = d(p, q),$$

where $D[p, q]$ is the value stored in the DP table by the algorithm.

2.5.2 Formal Proof of the Levenshtein Recurrence

Let Σ be a finite alphabet and Σ^* the set of all finite strings over Σ. For $x = x_1 x_2 \cdots x_m$ and $y = y_1 y_2 \cdots y_n$ in Σ^*, define three primitive edit operations:

- deletion $\mathrm{del}(a)$, which removes a symbol $a \in \Sigma$;
- insertion $\mathrm{ins}(a)$, which inserts a symbol $a \in \Sigma$;
- substitution $\mathrm{sub}(a \mapsto b)$, which replaces a by b.

Each operation has unit cost:

$$c(\mathrm{del}(a)) = 1, \quad c(\mathrm{ins}(a)) = 1, \quad c(\mathrm{sub}(a \mapsto b)) = \begin{cases} 0, & a = b, \\ 1, & a \neq b. \end{cases}$$

An *edit script* S is a finite sequence of such operations. Let $\mathrm{cost}(S) = \sum_{o \in S} c(o)$ be its total cost. The Levenshtein distance between $x, y \in \Sigma^*$ is

$$d_{\mathrm{L}}(x, y) = \min\Big\{ \mathrm{cost}(S) : S(x) = y \Big\}.$$

For prefixes, define $d(i, j) := d_{\mathrm{L}}(x_1 \cdots x_i,\, y_1 \cdots y_j)$.

Base cases. Transforming an empty string to a length-j string requires j insertions, and conversely i deletions are needed to erase a length-i string. Hence

$$d(0, j) = j, \qquad d(i, 0) = i, \qquad d(0, 0) = 0.$$

Claim (Recurrence). For $i, j \ge 1$,

$$\boxed{d(i, j) = \min\Big\{ d(i-1, j) + 1,\ d(i, j-1) + 1,\ d(i-1, j-1) + \mathbf{1}_{x_i \neq y_j} \Big\}.}$$

Proof. Let $u = x_1 \cdots x_i$ and $v = y_1 \cdots y_j$. Let S^* be an optimal edit script such that $S^*(u) = v$ and $\text{cost}(S^*) = d(i, j)$. Write $u = u'a$ and $v = v'b$, where $a = x_i$ and $b = y_j$ denote the final symbols.

Step 1. The last operation must affect the final position. Note that after all edits, the resulting string equals $v'b$. If the last operation of S^* did not touch the last symbol, the resulting string before the last edit would already be $v'b$, contradicting the minimality of S^*. Thus the final operation necessarily produces or removes the last character b or a.

Step 2. Exhaustion of possible last operations. Exactly three situations can occur:

1. **Deletion.** The last edit deletes the final symbol a of u. The preceding portion of the script converts $x_1 \cdots x_{i-1}$ into v. Hence the total cost is $d(i-1, j) + 1$.
2. **Insertion.** The last edit inserts the final symbol b. Then the preceding script converts u into $y_1 \cdots y_{j-1}$, contributing cost $d(i, j-1) + 1$.
3. **Substitution (or match).** The last edit changes a to b (or leaves it unchanged if $a = b$). The preceding script converts $x_1 \cdots x_{i-1}$ into $y_1 \cdots y_{j-1}$, yielding cost $d(i-1, j-1) + c(\text{sub}(a \mapsto b)) = d(i-1, j-1) + \mathbf{1}_{a \neq b}$.

Step 3. Minimality over the three cases. Every valid edit sequence must end in one of these three types of operations, so $d(i, j)$ cannot be smaller than the minimum of their total costs:

$$d(i, j) \;\geq\; \min\Big\{ d(i-1, j) + 1,\; d(i, j-1) + 1,\; d(i-1, j-1) + \mathbf{1}_{x_i \neq y_j} \Big\}.$$

Conversely, each of the three cases above gives a legitimate script realizing the right-hand side with that cost. Hence equality holds, proving the recurrence.

Step 4. Sequential structure (justification). The argument relies on the *principle of optimal substructure*: if S^* is optimal for (i, j), then each prefix of S^* must be optimal for the intermediate prefix pair it produces. Otherwise one could replace that portion by a cheaper sequence, contradicting minimality. Because edit operations act locally on adjacent positions and costs are nonnegative, any optimal sequence can be rearranged (without changing total cost) so that it proceeds monotonically through prefix pairs $(0, 0) \to (1, *) \to \cdots \to (i, j)$. Thus the recurrence indeed characterizes $d(i, j)$ for all i, j.

Conclusion. Combining the base cases and recurrence yields a well-defined dynamic programming formulation of the Levenshtein distance:

$$\begin{cases} d(0, j) = j, & 0 \leq j \leq n, \\ d(i, 0) = i, & 0 \leq i \leq m, \\ d(i, j) = \min\{\, d(i-1, j) + 1,\; d(i, j-1) + 1,\; d(i-1, j-1) + \mathbf{1}_{x_i \neq y_j} \,\}, & i, j \geq 1. \end{cases}$$

□

2.6 Example Output 1

```
Enter source string:  scavenger
Enter target string:  avenge

Levenshtein distance: 3

Step-by-step transformations (initial -> ... -> target):
  [ 0] scavenger
  [ 1] cavenger
  [ 2] avenger
  [ 3] avenger
  [ 4] avenger
  [ 5] avenger
  [ 6] avenger
  [ 7] avenger
  [ 8] avenger
  [ 9] avenge

Operations applied (in order):
[{'char': 's', 'op': 'delete', 'pos': 0},
 {'char': 'c', 'op': 'delete', 'pos': 0},
 {'char': 'a', 'op': 'match', 'pos': 0},
 {'char': 'v', 'op': 'match', 'pos': 1},
 {'char': 'e', 'op': 'match', 'pos': 2},
 {'char': 'n', 'op': 'match', 'pos': 3},
 {'char': 'g', 'op': 'match', 'pos': 4},
 {'char': 'e', 'op': 'match', 'pos': 5},
 {'char': 'r', 'op': 'delete', 'pos': 6}]
```

2.7 Complexity Analysis

- Time: $\mathcal{O}(m \cdot n)$
- Space: $\mathcal{O}(m \cdot n)$

2.8 Two-row Wagner Fischer

The standard Wagner-Fischer dynamic programming algorithm for computing the Levenshtein distance requires an $m \times n$ matrix to store all intermediate edit distances between prefixes of the input strings S and T. However, observe that d(i,j) is only a function of d(i-1,j), d(i,j-1), d(i-1,j-1), x_i , y_j and therefore each entry in the current row depends only on the values from the *previous* row and the *current* row itself. Hence, instead of retaining the full matrix, one can iteratively reuse two one-dimensional arrays: one representing the previous row and the other representing the current row. This **reduces the space complexity from $\mathcal{O}(\mathbf{mn})$ to $\mathcal{O}(\min(\mathbf{m}, \mathbf{n}))$**, while the time complexity remains $\mathcal{O}(mn)$. This optimization is particularly effective when the goal is only to obtain the edit distance value, rather than reconstructing the sequence of operations leading to it. Here is the implementation of Two Row Wagner Fischer in Python.

2.9 Example Output 2

```
Enter first string:  heraclitus
Enter second string:  hercules
Levenshtein distance between 'heraclitus' and 'hercules' = 5
```

3 Hirschberg Algorithm

3.1 Needleman-Wunsch Scoring Scheme

The Needleman-Wunsch algorithm (1970) provides the classical dynamic programming framework for *global sequence alignment*. It assigns a score to each possible alignment between two sequences $X = x_1x_2 \dots x_m$ and $Y = y_1y_2 \dots y_n$ using a substitution function $s(x_i, y_j)$ and a gap penalty d. In order to calculate the Levenshtein distance, a match contributes a $s(x_i, y_j) = 0$ score, a mismatch incurs a penalty of $s(x_i, y_j) = 1$ and a gap incurs a penalty of $d = 1$. The recurrence relation is:

$$F(i,j) = \max \begin{cases} F(i-1, j-1) - s(x_i, y_j), & \text{(match/mismatch)} \\ F(i-1, j) - d, & \text{(deletion)} \\ F(i, j-1) - d, & \text{(insertion).} \end{cases}$$

The value $F(m, n)$ gives the optimal global alignment score between X and Y.

3.2 Overview

Hirschberg's algorithm [Hir75] computes the same Needleman-Wunsch alignment score but with drastically reduced space complexity. While Needleman-Wunsch requires $\mathcal{O}(mn)$ memory to store the full dynamic programming matrix, Hirschberg's divide-and-conquer strategy reduces this to $\mathcal{O}(n)$. It achieves this by computing only the forward and backward Needleman-Wunsch score vectors for partial alignments, then locating an optimal split point that partitions the problem into two smaller sub-alignments.

3.3 Explanation

Hirschberg's algorithm applies the Needleman-Wunsch scoring rules recursively. For a string X of length m and Y of length n, it divides X into two halves, say X_{left} and X_{right}, and computes:

1. the **forward Needleman-Wunsch scores** for aligning X_{left} with every prefix of Y, and
2. the **backward Needleman-Wunsch scores** for aligning X_{right} with every suffix of Y.

These two linear-space score vectors are then added elementwise to find the index in Y that yields the highest combined Needleman-Wunsch score, identifying the optimal split point for recursion.

The algorithm then recursively aligns the two subproblems $(X_{\text{left}}, Y_{\text{left}})$ and $(X_{\text{right}}, Y_{\text{right}})$ using the same scoring scheme until reaching base cases of size 0 or 1, where the alignment is trivial. Finally, the full alignment (and thus the total Needleman-Wunsch score)

is reconstructed from the partial solutions. This method achieves identical alignment results to the original Needleman-Wunsch algorithm but with linear memory, making it efficient for long genomic sequences.

3.4 Pseudocode

```
# Compute last row of NW (Wagner-Fischer) DP
function NW_Score(A[1..m], B[1..n]) -> array[0..n]:
    prev[0..n] <- [0, 1, 2, ..., n]
    for i <- 1 to m:
        cur[0..n] <- [i] + [0]*(n)
        ai <- A[i]
        for j <- 1 to n:
            cost <- 0 if ai == B[j] else 1
            cur[j] <- min(prev[j-1] + cost, prev[j] + 1,
    ↪ cur[j-1] + 1)
        prev <- cur
    return prev

# Base case alignment using standard DP
function Align_Base(A[1..m], B[1..n]) -> list of operations:
    D[0..m][0..n] <- 0
    choice[0..m][0..n] <- ""

    for i <- 0 to m:
        D[i][0] <- i
        choice[i][0] <- "start" if i == 0 else "delete"
    for j <- 0 to n:
        D[0][j] <- j
        choice[0][j] <- "start" if j == 0 else "insert"

    for i <- 1 to m:
        for j <- 1 to n:
            cost <- 0 if A[i] == B[j] else 1
            diag <- D[i-1][j-1] + cost
            delete <- D[i-1][j] + 1
            insert <- D[i][j-1] + 1
            best <- min(diag, delete, insert)
            D[i][j] <- best

            if best == diag and cost == 0:
                choice[i][j] <- "match"
            elif best == diag:
                choice[i][j] <- "substitute"
            elif best == delete:
                choice[i][j] <- "delete"
            else:
                choice[i][j] <- "insert"

    # Backtrack to generate operations
    i <- m; j <- n
```

```
    rev_ops <- []
    while i > 0 or j > 0:
        op <- choice[i][j]
        if op == "match":
            rev_ops.append({"op":"match", "pos":i, "char":A[i]})
            i <- i - 1; j <- j - 1
        elif op == "substitute":
            rev_ops.append({"op":"substitute", "pos":i,
    ↪ "from":A[i], "to":B[j]})
            i <- i - 1; j <- j - 1
        elif op == "delete":
            rev_ops.append({"op":"delete", "pos":i, "char":A[i]})
            i <- i - 1
        elif op == "insert":
            rev_ops.append({"op":"insert", "pos":i, "char":B[j]})
            j <- j - 1
        else:
            break
    return reverse(rev_ops)

# Hirschberg recursion with step-by-step log
function Hirschberg_With_Log(S[1..m], T[1..n]) -> (ops,
    ↪ transformations):
    function Rec(A[1..m], B[1..n]) -> list of operations:
        if length(A) == 0:
            return [{"op":"insert","pos":i,"char":B[i]} for i in
    ↪ 0..length(B)-1]
        if length(B) == 0:
            return [{"op":"delete","pos":0,"char":A[i]} for i in
    ↪ 0..length(A)-1]
        if length(A) == 1 or length(B) == 1:
            return Align_Base(A, B)

        mid <- floor(length(A)/2)
        scoreL <- NW_Score(A[0..mid-1], B)
        scoreR <- NW_Score(reverse(A[mid..]), reverse(B))
        nB <- length(B)

        best_k <- 0; best_val <- \infty
        for k <- 0 to nB:
            val <- scoreL[k] + scoreR[nB - k]
            if val < best_val:
                best_val <- val
                best_k <- k

        left_ops <- Rec(A[0..mid-1], B[0..best_k-1])
        right_ops <- Rec(A[mid..], B[best_k..])

        # Shift right_ops positions
        shifted_right_ops <- []
        for op in right_ops:
```

```
            op_copy <- copy(op)
            if "pos" in op_copy:
                op_copy["pos"] <- op_copy["pos"] + mid
            shifted_right_ops.append(op_copy)

        return left_ops + shifted_right_ops

    ops <- Rec(S, T)

    # Build transformations
    cur <- list(S)
    transformations <- [join(cur)]
    applied <- []

    for action in ops:
        typ <- action["op"]
        if typ == "match":
            applied.append({"op":"match","pos":action["pos"],
            "char":action["char"]})
        elif typ == "substitute":
            p <- action["pos"]
            old <- cur[p]
            cur[p] <- action["to"]
            applied.append({"op":"substitute","pos":p,"from":old,
            "to":action["to"]})
        elif typ == "delete":
            p <- action["pos"]
            removed <- cur.pop(p)

    ↪ applied.append({"op":"delete","pos":p,"char":removed})
        elif typ == "insert":
            p <- action["pos"]
            if p < 0: p <- 0
            if p > length(cur): p <- length(cur)
            cur.insert(p, action["char"])

    ↪ applied.append({"op":"insert","pos":p,"char":action["char"]})
        transformations.append(join(cur))

    return (applied, transformations)

# Print summary of alignment
function Print_Summary(applied, transformations):
    edit_ops <- [op for op in applied if op["op"] in
    ↪ ["insert","delete","substitute"]]
    print("Edit distance:", length(edit_ops))
    print("Step-by-step transformations:")
    for idx <- 0 to length(transformations)-1:
        print("[" + idx + "] " + transformations[idx])
    print("Operations applied (in order):")
    print(applied)
```

```

# Main entry point
function Main():
    S <- input("Enter source string S:")
    T <- input("Enter target string T:")

    applied, transformations <- Hirschberg_With_Log(S, T)
    Print_Summary(applied, transformations)
```

Listing 2: Hirschberg Alignment with Step-by-Step Log

3.5 Python Implementation

Implementation of Hirschberg Algorithm in Python

3.6 Hirschberg's Algorithm: Invariant and Correctness Proof

Hirschberg's algorithm computes the optimal edit sequence (alignment) with only linear space, using the divide-and-conquer principle.

3.6.1 Invariant

Let A and B be strings of lengths m and n. Let $\mathsf{Forward}(i)$ denote the DP array computed for prefixes $A[1..i]$ and all of B, and $\mathsf{Backward}(i)$ denote the DP array for suffixes $A[i+1..m]$ and B (reversed). At each divide step, the algorithm maintains the following invariant:

Invariant (H): For every $0 \le j \le n$,

$$\mathsf{Forward}(i)[j] = d(A[1..i], B[1..j]), \qquad \mathsf{Backward}(i)[n-j] = d(A[i+1..m], B[j+1..n]).$$

The split index j^* minimizing

$$\mathsf{Forward}(i)[j] + \mathsf{Backward}(i)[n-j]$$

corresponds to the optimal partition of B into two halves at which the global alignment path crosses the i-th row.

3.6.2 Proof of Invariant and Minimality

Base case. When $m = 0$ or $n = 0$, the edit distance is trivially $|m - n|$, and the algorithm returns this directly, satisfying (H).

Recursive step. Assume (H) holds for all subproblems of smaller size. For a given $A[1..m]$ and $B[1..n]$, Hirschberg splits A at $i = \lfloor m/2 \rfloor$ and computes:

$\mathsf{Forward}(i)$ using the standard Wagner–Fischer recurrence forward,

$\mathsf{Backward}(i)$ using the same recurrence backward on reversed strings.

By correctness of Wagner–Fischer (proved above), both arrays store exact subproblem distances, satisfying (H).

The split position j^* is chosen as

$$j^* = \arg\min_{0 \le j \le n} \Big(\mathsf{Forward}(i)[j] + \mathsf{Backward}(i)[n-j] \Big),$$

ensuring that the total distance

$$d(m,n) = \mathsf{Forward}(i)[j^*] + \mathsf{Backward}(i)[n-j^*]$$

is minimal, since every edit path from $(0,0)$ to (m,n) must cross row i exactly once.

Recursive correctness. The algorithm then recursively computes optimal alignments for the two subproblems:

$$(A[1..i], B[1..j^*]) \quad \text{and} \quad (A[i+1..m], B[j^*+1..n]).$$

By the inductive assumption, both recursive calls yield optimal alignments and minimal distances. Their concatenation forms an optimal alignment for (A, B).

Termination. Recursion ends when one of the strings is of length 1, at which point the alignment and distance can be computed directly.

Conclusion. By induction, invariant (H) holds at every recursive level. Since at each step the split index j^* minimizes the total distance, Hirschberg's algorithm yields the globally minimal edit sequence using only linear space.

3.7 Example Output

```
=== Hirschberg Edit/Alignment (log) ===

Enter source string S:  abcde
Enter target string T:  cbdeg

Hirschberg Alignment Summary
============================
Edit distance: 3

Step-by-step transformations:
  [ 0] abcde
  [ 1] cbcde
  [ 2] cbcde
  [ 3] cbde
  [ 4] cbde
  [ 5] cbde
  [ 6] cbdeg

Operations applied (in order):
[{'from': 'a', 'op': 'substitute', 'pos': 0, 'to': 'c'},
```

```
{'char': 'b', 'op': 'match', 'pos': 1},
{'char': 'c', 'op': 'delete', 'pos': 2},
{'char': 'd', 'op': 'match', 'pos': 3},
{'char': 'e', 'op': 'match', 'pos': 4},
{'char': 'g', 'op': 'insert', 'pos': 4}]
```

3.8 Complexity Analysis

- Time: $\mathcal{O}(m \cdot n)$
- Space: $\Theta(\min(m, n))$

4 Ukkonen's Algorithm (Edit Distance up to Threshold k)

4.1 Overview

Ukkonen's algorithm (1985) [Ukk85] provides an optimized dynamic programming approach for computing the edit distance when a maximum threshold k is known in advance. It is based on the same recurrence as the Wagner-Fischer algorithm but restricts computation to a narrow diagonal band of width $2k + 1$ around the main diagonal of the DP matrix. If the true edit distance exceeds k, the algorithm can terminate early without fully filling the matrix. This drastically reduces the time and space complexity from $\mathcal{O}(mn)$ to $\mathcal{O}(k \cdot \min(m, n))$.

4.2 Explanation

The algorithm leverages the fact that if two strings differ by at most k edits, the optimal alignment path must lie within k cells of the main diagonal. Therefore, only entries $D(i, j)$ satisfying $|i - j| \leq k$ need to be computed. The recurrence is the same as in Wagner-Fischer:

$$D(i,j) = \min \begin{cases} D(i-1,j) + 1, & \text{(deletion)} \\ D(i,j-1) + 1, & \text{(insertion)} \\ D(i-1,j-1) + \mathbf{1}_{\{x_i \neq y_j\}}, & \text{(match or substitution).} \end{cases}$$

However, computation and storage are limited to the $(2k + 1)$ diagonals centered on the main diagonal. If at any step all candidate cells exceed k, the algorithm terminates early, concluding that the edit distance $> k$.

In terms of scoring, Ukkonen's algorithm still corresponds to the Needleman-Wunsch global alignment with a match score of 0, mismatch penalty 1, and gap penalty 1. It differs only in efficiency: by constraining the search space, it can decide equality within k edits much faster than full dynamic programming, making it particularly useful in DNA sequence comparison and real-time approximate string matching.

4.3 Pseudocode

```
# Compute Levenshtein distance between strings a and b,
# up to a maximum threshold k. Returns distance or None if > k.

function Ukkonen_Levenshtein(a[1..m], b[1..n], k) -> int or None:
    # Quick rejection if length difference exceeds threshold
    if abs(m - n) > k:
        return None

    # Initialize previous row for DP (0..min(n, k))
    prev[0..min(n, k)] := [0, 1, 2, ..., min(n, k)]
    curr[0..n] := [0, 0, ..., 0]
    INF := k + 1

    # Iterate over characters of a
    for i := 1 to m:
        low  := max(1, i - k)
        high := min(n, i + k)

        # Initialize first column
        if low == 1:
            curr[0] := i
        else:
            curr[0] := INF

        # Compute DP values only within diagonal band [low, high]
        for j := low to high:
            cost := 0 if a[i] == b[j] else 1

            if j < length(prev):
                deletion := prev[j] + 1
            else:
                deletion := INF

            insertion := curr[j - 1] + 1

            if (j - 1) < length(prev):
                substitution := prev[j - 1] + cost
            else:
                substitution := INF

            curr[j] := min(deletion, insertion, substitution)

        # Early termination: all values in band exceed k
        if min(curr[low .. high]) > k:
            return None

        # Prepare for next iteration
        prev := curr
        curr := [INF, INF, ..., INF] of length (n + 1)
```

```
50
51      # Return result if within threshold
52      if prev[n] <= k:
53          return prev[n]
54      else:
55          return None
56  # CLI interface: read input strings and threshold, call
        ↪ Ukkonen_Levenshtein
57  function main(argv):
58      if length(argv) >= 3:
59          a := argv[0]
60          b := argv[1]
61          k := int(argv[2])
62      else:
63          a := input("Enter string a:")
64          b := input("Enter string b:")
65          k := int(input("Enter threshold k:"))
66
67      dist := Ukkonen_Levenshtein(a, b, k)
68
69      if dist is None:
70          print("No alignment within distance", k)
71          return 1
72      else:
73          print("Levenshtein distance =", dist)
74          return 0
```

Listing 3: Ukkonen's Bounded Levenshtein Algorithm (Python equivalent)

4.4 Python Implementation

Implementation of Ukkonen Algorithm in Python

4.5 Ukkonen's Algorithm: Invariant and Correctness Proof

4.5.1 Invariant

Let A and B be two strings of lengths m and n, respectively, and let $k \in \mathbb{N}$ be the distance threshold. Denote by $d(i, j)$ the Levenshtein edit distance between the prefixes $A[1..i]$ and $B[1..j]$.

After processing row i in Ukkonen's algorithm (i.e. after updating and swapping the arrays so that *prev* holds the row corresponding to prefix length i), the following invariant holds:

Invariant (I): For all indices j satisfying $|i - j| \leq k$,

$$prev[j] = d(i, j),$$

and for indices outside this band ($|i - j| > k$),

$$prev[j] \geq k + 1.$$

That is, after iteration i, the algorithm maintains the exact dynamic programming values $d(i, j)$ inside the diagonal band of width $2k + 1$, and values at least $k + 1$ (effectively ∞) outside the band.

4.5.2 Proof of the Invariant

We prove (I) by induction on i.

Base case ($i = 0$). When $i = 0$, the edit distance between the empty prefix $A[1..0]$ and $B[1..j]$ is

$$d(0, j) = j \quad \text{for all } 0 \leq j \leq n.$$

The algorithm initializes

$$prev[j] = j, \quad 0 \leq j \leq \min(n, k),$$

and sets other entries to ∞ (or equivalently $\geq k + 1$). Hence, the invariant holds for $i = 0$.

Inductive step. Assume the invariant holds for $i - 1$; that is,

$$prev[j] = d(i - 1, j) \quad \text{for all } |(i - 1) - j| \leq k.$$

We show it holds for i.

For $i \geq 1$, Ukkonen's algorithm computes the new row *curr* for indices

$$j \in [\text{low}, \text{high}], \quad \text{low} = \max(0, i - k), \quad \text{high} = \min(n, i + k),$$

using the recurrence

$$curr[j] = \min \begin{cases} prev[j] + 1, & \text{(deletion)} \\ curr[j - 1] + 1, & \text{(insertion)} \\ prev[j - 1] + \mathbb{1}_{A_i \neq B_j}, & \text{(substitution/match)} \end{cases}$$

which is exactly the standard Levenshtein recurrence

$$d(i, j) = \min \begin{cases} d(i - 1, j) + 1, \\ d(i, j - 1) + 1, \\ d(i - 1, j - 1) + \mathbb{1}_{A_i \neq B_j}. \end{cases}$$

By the inductive hypothesis:

- $prev[j] = d(i - 1, j)$ for $|(i - 1) - j| \leq k$.
- $curr[j - 1]$ stores $d(i, j - 1)$ since it was computed earlier in the same iteration.
- $prev[j - 1] = d(i - 1, j - 1)$ for $|(i - 1) - (j - 1)| \leq k$.

These three quantities are sufficient to compute the correct $d(i, j)$ whenever $|i - j| \leq k$.

If any of the required neighbors lies outside the band ($|i - j| > k$), then the true distance $d(i, j) > k$, and the algorithm stores a value $\geq k + 1$ there, which is consistent with (I).

Thus, after completing the inner loop and swapping arrays, the invariant (I) holds for iteration i.

Early termination. At the end of iteration i, the algorithm checks whether

$$\min_{j \in [\text{low},\text{high}]} curr[j] > k.$$

By the invariant, $curr[j] = d(i, j)$ for all j in the band. If all these values exceed k, then for this row and all subsequent rows, any alignment path will have cost $> k$. Hence, it is correct to terminate early and report "$> k$".

Initialization check. Finally, if $|m - n| > k$, then the edit distance must satisfy $d(m, n) \geq |m - n| > k$, so the algorithm correctly returns "$> k$" immediately.

Conclusion. By induction, the invariant (I) holds for all $i = 0, 1, \ldots, m$. Therefore, upon termination:

$$prev[n] = \begin{cases} d(m, n), & \text{if } d(m, n) \leq k, \\ > k, & \text{otherwise.} \end{cases}$$

This proves the correctness of Ukkonen's algorithm.

4.6 Example Output

4.6.1 Output 1:

```
Enter string a:  algorithm
Enter string b:  gotham
Enter threshold k:  5
Levenshtein distance = 5
```

4.6.2 Output 2:

```
Enter string a:  algorithms
Enter string b:  gotham
Enter threshold k:  5
No alignment within distance 5.
```

4.7 Complexity Analysis

- Time: $O(k \cdot \min(m, n))$
- Space: $O(\min(n, k))$
- Early termination possible if distance $> k$

5 Complexity Analysis

5.1 Summary Table

Algorithm	Time	Space	Operations Stored?
Classical Wagner-Fischer	$\Theta(mn)$	$\Theta(mn)$	Yes
Two-Row Wagner-Fischer	$\Theta(mn)$	$\Theta(\min(m,n))$	No
Hirschberg	$\mathcal{O}(mn)$	$\mathcal{O}(\min(m,n))$	Yes
Ukkonen	$O(k\min(m,n))$	$O(\min(n,k))$	No

Table 1: Complexities Summarised

5.2 Derivations of Complexities

Let two input strings be denoted by

$$x = x_1x_2\ldots x_m, \qquad y = y_1y_2\ldots y_n,$$

of lengths m and n respectively.

5.2.1 Classical Wagner-Fischer Algorithm

The Wagner-Fischer dynamic programming (DP) table $D \in \mathbb{N}^{(m+1)\times(n+1)}$ is defined by

$$D_{i,j} = \begin{cases} 0, & i=0, j=0, \\ j, & i=0, \\ i, & j=0, \\ \min\Big(D_{i-1,j}+1,\ D_{i,j-1}+1,\ D_{i-1,j-1}+\mathbb{1}_{x_i\neq y_j}\Big), & \text{otherwise}, \end{cases}$$

The Levenshtein distance between x and y is $D_{m,n}$.

Time Complexity. The DP matrix D contains $(m+1)(n+1) = \Theta(mn)$ entries. Each cell requires $\mathcal{O}(1)$ time to compute, involving a constant-time min operation over three possible transitions (substitution, insertion, deletion). Therefore, the total running time is

$$T_{\mathrm{WF}}(m,n) = \Theta(mn).$$

Space Complexity and Reconstruction. The standard algorithm stores the entire DP matrix D, requiring

$$S_{\mathrm{WF}}(m,n) = \Theta(mn)$$

space. To reconstruct the sequence of edit operations (alignment path), a second matrix (often called `choice`) is maintained, storing the direction of the minimum for each cell. This doubles the constant factor but preserves the asymptotic bound.

If only the numeric distance $D_{m,n}$ is required, the memory can be reduced to

$$S'_{\text{WF}}(m,n) = \mathcal{O}(\min(m,n)),$$

by maintaining only two consecutive rows (or columns). However, this optimization prevents backtracking to reconstruct the full edit sequence.

Output Storage. The resulting sequence of edit operations has length at most $m+n$, giving

$$S_{\text{output}} = \Theta(m+n).$$

If each intermediate transformed string is also recorded, the total storage becomes

$$S_{\text{transformations}} = \mathcal{O}\Big((m+n)\cdot\max(m,n)\Big),$$

which is effectively quadratic in the input size.

5.2.2 Two-Row Wagner-Fischer Algorithm

Consider again the DP table $D \in \mathbb{N}^{(m+1)\times(n+1)}$. Suppose we only require the Levenshtein distance $D_{m,n}$ and not the full alignment path.

Observation. From the recurrence

$$D_{i,j} = \min\Big(D_{i-1,j}+1,\ D_{i,j-1}+1,\ D_{i-1,j-1}+[x_i \neq y_j]\Big),$$

we see that computing row i only requires entries from row $i-1$ (previous row) and the current row . All other rows $(0,\ldots,i-2)$ are never referenced again.

Algorithm. - Maintain two arrays of length $n+1$: `prev` and `curr`. - Initialize `prev` with the base case $D_{0,j} = j$. - For each $i = 1,\ldots,m$:

1. Compute `curr[j]` using `prev[j]`, `curr[j-1]`, and `prev[j-1]`.
2. After finishing row i, copy `curr` to `prev` (or swap pointers).

- At the end, $D_{m,n}$ is stored in `curr[n]`.

Time Complexity. Each entry $D_{i,j}$ is computed exactly once in $\mathcal{O}(1)$ time. There are $(m+1)(n+1)$ entries, so the total time is

$$T_{\text{2row}}(m,n) = (m+1)(n+1)\cdot\mathcal{O}(1) = \Theta(mn).$$

Space Complexity. At any moment, we only store two rows of length $n+1$. Hence, the memory requirement is

$$S_{\text{2row}}(m,n) = 2\cdot(n+1)\cdot\mathcal{O}(1) = \Theta(n).$$

By symmetry, choosing the shorter string as the row dimension yields

$$S_{\text{2row}}(m,n) = \Theta(\min(m,n)).$$

5.2.3 Hirschberg Algorithm

Hirschberg's algorithm applies a divide-and-conquer approach to reduce memory usage while preserving the optimal alignment.

Method. The algorithm splits the longer string (say x) at its midpoint and recursively computes two linear-space DP arrays (forward and backward scores) for string y of length n. The midpoint of the optimal path is found by combining these two partial results.

Complexity. Each level of recursion requires $\mathcal{O}(m'n)$ time for a subproblem of size m', and the total across all levels satisfies

$$T_{\text{Hirsch}}(m, n) = \mathcal{O}(mn).$$

The working memory consists of two arrays of length n (for forward and backward computations) plus recursion stack overhead $\mathcal{O}(\log m)$, yielding

$$S_{\text{Hirsch}}(m, n) = \mathcal{O}(n).$$

The reconstructed edit sequence still occupies $\Theta(m + n)$ space.

5.2.4 Ukkonen Algorithm (Banded Distance Computation)

Method. If an upper bound k on the Levenshtein distance is known in advance, Ukkonen's algorithm restricts computation to a diagonal band of width $2k + 1$, containing only cells (i, j) satisfying $|i - j| \leq k$. This significantly reduces the number of states considered.

Complexity. For each row i, only $\mathcal{O}(k)$ columns are updated, giving total complexity

$$T_{\text{Ukk}}(m, n, k) = \mathcal{O}(k \cdot \min(m, n)), \qquad S_{\text{Ukk}}(m, n, k) = \mathcal{O}(k).$$

If at any step all values in the active band exceed k, the algorithm terminates early, certifying that the true distance is greater than k. In practice, Ukkonen's approach is extremely efficient for small thresholds or highly similar strings.

5.3 Practical recommendations

- If inputs are small (both m, n moderate) and you need the alignment plus a full transformation trace, use Wagner-Fischer with logging; it's simplest to implement and understand.
- If only figuring out the distance is required then two row Wagner Fischer would be the best.
- If one or both inputs are large but you still need the full optimal alignment, use Hirschberg to lower working memory to linear.
- If you only care whether the distance is $\leq k$ (approximate matching) and k is much smaller than the lengths, use Ukkonen for significant speed and memory savings.

- Avoid storing all intermediate transformed strings unless required for visualization: storing these can blow up memory to quadratic in the worst case.
- The Ukkonen algorithm is useful in hypothesis testing scenarios where the goal is to quantify similarity between data sequences under a tolerance threshold. For example, in plagiarism detection, it can test the hypothesis that two documents are identical or nearly identical by measuring their edit distance and checking if it crosses a chosen threshold k. Similarly in bioinformatics, it can test whether two genetic sequences differ by only a limited number of mutations, supporting hypotheses about evolutionary relatedness or variant classification.

6 Experiments

We will do the following with Wagner-Fischer (last one using the Two-Row variant) which has a time complexity of $\Theta(mn)$. We **run the Wagner-Fischer algorithm on string pairs**, **reconstruct minimal edit sequences** and **validate edit operations** below. From the internet, we found famous string pairs with their Levenshtein distances: (kitten,sitting,3), (intention,execution,5) and (strange,france,3)

1. **(kitten,sitting,**3**):**

```
Source string: kitten
Target string: sitting

Levenshtein distance: 3

Step-by-step transformations (initial -> ... -> target):
  [ 0] kitten
  [ 1] sitten
  [ 2] sitten
  [ 3] sitten
  [ 4] sitten
  [ 5] sittin
  [ 6] sittin
  [ 7] sitting

Operations applied (in order):
[{'from': 'k', 'op': 'substitute', 'pos': 0, 'to': 's'},
 {'char': 'i', 'op': 'match', 'pos': 1},
 {'char': 't', 'op': 'match', 'pos': 2},
 {'char': 't', 'op': 'match', 'pos': 3},
 {'from': 'e', 'op': 'substitute', 'pos': 4, 'to': 'i'},
 {'char': 'n', 'op': 'match', 'pos': 5},
 {'char': 'g', 'op': 'insert', 'pos': 6}]
```

2. **(intention,execution,**5**):**

```
Source string: intention
```

```
Target string: execution

Levenshtein distance: 5

Step-by-step transformations (initial -> ... -> target):
  [ 0] intention
  [ 1] entention
  [ 2] extention
  [ 3] exeention
  [ 4] execntion
  [ 5] execution
  [ 6] execution
  [ 7] execution
  [ 8] execution
  [ 9] execution

Operations applied (in order):
[{'from': 'i', 'op': 'substitute', 'pos': 0, 'to': 'e'},
 {'from': 'n', 'op': 'substitute', 'pos': 1, 'to': 'x'},
 {'from': 't', 'op': 'substitute', 'pos': 2, 'to': 'e'},
 {'from': 'e', 'op': 'substitute', 'pos': 3, 'to': 'c'},
 {'from': 'n', 'op': 'substitute', 'pos': 4, 'to': 'u'},
 {'char': 't', 'op': 'match', 'pos': 5},
 {'char': 'i', 'op': 'match', 'pos': 6},
 {'char': 'o', 'op': 'match', 'pos': 7},
 {'char': 'n', 'op': 'match', 'pos': 8}]
```

3. **(strange,france,**3**):**

```
Source string: strange
Target string: france

Levenshtein distance: 3

Step-by-step transformations (initial -> ... -> target):
  [ 0] strange
  [ 1] trange
  [ 2] frange
  [ 3] frange
  [ 4] frange
  [ 5] frange
  [ 6] france
  [ 7] france

Operations applied (in order):
[{'char': 's', 'op': 'delete', 'pos': 0},
 {'from': 't', 'op': 'substitute', 'pos': 0, 'to': 'f'},
 {'char': 'r', 'op': 'match', 'pos': 1},
```

```
{'char': 'a', 'op': 'match', 'pos': 2},
{'char': 'n', 'op': 'match', 'pos': 3},
{'from': 'g', 'op': 'substitute', 'pos': 4, 'to': 'c'},
{'char': 'e', 'op': 'match', 'pos': 5}]
```

6.1 Analyzing runtime scaling with input length

We take strings of lengths $5000-20000$ and check their run times on the Two-Row Wagner Fischer algorithm. The data is given below -

1. $[1(5000 \text{ times}), 2(5000 \text{ times})]$: 6.4475 seconds
2. $[1(5000 \text{ times}), 2(10000 \text{ times})]$: 12.4623 seconds
3. $[1(10000 \text{ times}), 2(5000 \text{ times})]$: 12.4578 seconds
4. $[1(5000 \text{ times}), 2(20000 \text{ times})]$: 23.9839 seconds
5. $[1(20000 \text{ times}), 2(5000 \text{ times})]$: 24.9947 seconds
6. $[1(10000 \text{ times}), 2(10000 \text{ times})]$: 22.6140 seconds
7. $[1(10000 \text{ times}), 2(20000 \text{ times})]$: 49.3412 seconds
8. $[1(20000 \text{ times}), 2(10000 \text{ times})]$: 52.1164 seconds
9. $[1(20000 \text{ times}), 2(20000 \text{ times})]$: 107.1135 seconds

P.S.: The inputs for the above analysis are provided here. Please paste the inputs in the code beforehand to prevent input time lag.

We can see that as the product of the lengths (mn) doubles, the runtime also doubles approximately. This is in accordance to the claim that the time complexity of the Two-Row Wagner Fischer algorithm is $\Theta(mn)$. We can repeat the same experiment for the other distance algorithms and verify the same.

7 Applications in DNA Edit Mutation Data

7.1 Overview

To obtain the genomic dataset for this study, we first identified the major SARS-CoV-2 variants (Alpha/B.1.1.7, Beta/B.1.351, Gamma/P.1, Delta/B.1.6.7, Lambda/Lineage C.37, Eta/B.1.525, Epsilon/B.1.427 and Omicron/B.1.1.529) from Wikipedia [wik]. For each variant, the corresponding PANGO lineage was recorded, and the complete genomic sequences were retrieved from the National Center for Biotechnology Information (NCBI) Virus database [nih] in FASTA format. These individual genome sequences were subsequently organized into a structured CSV file, where each row represented a single variant and its associated nucleotide sequence. Using this dataset, we implemented the Wagner-Fischer algorithm to compute the Levenshtein distance between every pair of variant genomes. The resulting pairwise distances were stored in a symmetric distance matrix D_{ij}, where each entry D_{ij} quantifies the minimal number of edit operations (insertions, deletions, or substitutions) required to transform the i^{th} variant genome into the j^{th} genome. This matrix serves as a quantitative measure of genomic dissimilarity among SARS-CoV-2 variants and forms the basis for subsequent comparative analysis.

7.2 Implementation in `Python`

The pairwise edit distances were computed using an efficient two-row implementation of the Wagner-Fischer dynamic programming algorithm. This approach reduces the space complexity from $\mathcal{O}(mn)$ to $\mathcal{O}(\min(m, n))$ by storing only two rows of the dynamic programming table at each step. Each genome sequence was compared against all others, and the resulting distances were normalized by the total length of the two sequences to ensure comparability across genomes of varying lengths. The final symmetric distance matrix was then exported as a CSV file for further analysis. The dataset is given here and its implementation in Python is given here. We have further modified the table in R to create a heatmap showcasing the distances. The final table is attached below.

7.3 Output

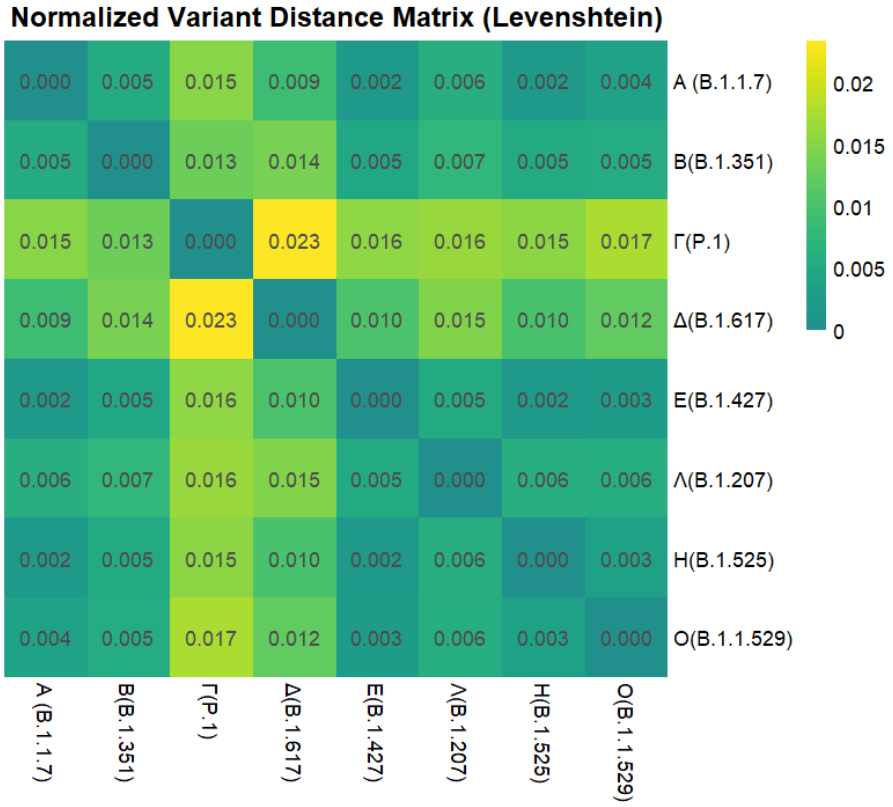

Figure 1: Distance Chart of SARS-CoV-2 Variants

7.4 Inference

The data leads us to guess the following:

1. The Γ variant diverges (has more mutations) from the other variants
2. E, Λ, H & O variants are similar to eachother (less mutations)
3. All variants have very low normalized Levenshtein distances, i.e., they did not mutate significantly

8 Applications in Comparing Text Documents

8.1 Overview

We explored the practical utility of this algorithm in evaluating the accuracy of automatically generated textual outputs against human references. We collected real-world datasets comprising pairs of *machine-generated* and *human-authored* translations in **Marathi**, **Bengali**, **Gujarati**, **Marwadi**, **Tamil** and **Hindi**. By computing the normalized Levenshtein distance between generated and reference strings, we derived a quantitative measure of accuracy and linguistic similarity. This approach allowed us to evaluate how closely machine models approximate human language.

8.2 Implementation in Python

To evaluate the accuracy of machine translation across different Indian languages, we employed the Wagner-Fischer algorithm to compute the Levenshtein distance between human and machine-translated English texts. The chosen source languages were Marathi, Bengali, Gujarati, Marwadi, Tamil and Hindi, each represented by a paragraph extracted from regional literary sources. Each paragraph was translated into English through two distinct methods:

1. **Manual translation:** We translated each paragraph as a human reference (assumed to be 100% accurate).

2. **Automated translation:** The same paragraphs were translated into English using Google Translate.

The goal was to quantitatively assess the deviation between human and machine translations. Using the Wagner-Fischer algorithm, we computed the minimum number of edit operations (insertions, deletions, and substitutions) required to transform the Google-translated text into the corresponding human-translated version. This Levenshtein distance serves as an interpretable measure of translation similarity, where a smaller value indicates higher accuracy.

The implementation was carried out in Python. Each pair of translations (human, machine) was processed line by line, and the total edit distance was computed and normalized by sentence length to ensure comparability.

8.3 Output

1. **Marathi:** Human and google generated translations for the children's story "Chimney anhi Kavlyachi Goshta" are given here.

   ```
   Levenshtein distance = 976
   Normalized Levenshtein distance = 0.29819737244118544
   ```

2. **Bengali:** Human and google generated translations for an excerpt of the children's story "HJBRL" are given here.

   ```
   Levenshtein distance = 797
   Normalized Levenshtein distance = 0.2979439252336449
   ```

3. **Gujarati:** Human and google generated translations for the poem "Kaagda to Badhey" are given here.

   ```
   Levenshtein distance = 576
   Normalized Levenshtein distance = 0.3282051282051282
   ```

4. **Marwadi:** Human and google generated translations for the song "Kesariya Baalam" are given here.

```
Levenshtein distance = 108
Normalized Levenshtein distance = 0.33962264150943394
```

5. **Tamil:** Human and google generated translations for an excerpt of the epic "Kamba Ramayanam" are given here.

   ```
   Levenshtein distance = 443
   Normalized Levenshtein distance = 0.4090489381348107
   ```

6. **Hindi:** Human and google generated translations for the children's story "Jhute Gavah" are given here.

   ```
   Levenshtein distance = 657
   Normalized Levenshtein distance = 0.24689966178128522
   ```

8.4 Inference

This tells us that the average normalized Levenshtein distance between the accurate translations and google generated translations is around 0.32 which is quite significant, given

$$\hat{d}(s,t) = \frac{d(s,t)}{\mid s \mid + \mid t \mid}.$$

We should be wary of auto-translated documents and also work on improving the technology.

9 Spelling Error Correction

9.1 Overview

To develop an effective spell correction system, we utilized the Levenshtein distance as the primary metric for quantifying dissimilarity between words. The fundamental idea is to measure how many single-character operations (insertions, deletions, or substitutions) are required to transform a misspelled word into a valid dictionary entry. To construct the reference vocabulary, a comprehensive English word list was obtained from the Natural Language Toolkit (NLTK) corpus in Python. Each user-input word was compared against this dictionary using the Wagner-Fischer dynamic programming algorithm to compute pairwise Levenshtein distances.

9.2 Implementation in Python

The code implements a simple spell-checker using the NLTK English word corpus and the Levenshtein distance. It first imports the NLTK dictionary of words and then takes a sentence as input from the user. Each word in the sentence is checked against the dictionary: if it exists, it is marked correct; otherwise, the code computes the Levenshtein distance between the word and all dictionary words to find the closest matches. Words with the minimum distance are suggested as possible corrections, allowing the user to identify and correct spelling errors efficiently.

9.3 Outputs

1. `Enter a sentence to spell-check: Continous distribushan`
   `'Continous': Spelling error. Suggestions (distance 1): ['continuous']`
   `'distribushan': Spelling error. Suggestions (distance 3): ['distribution']`

2. `Enter a sentence to spell-check: my nime is karl`
   `'nime': Spelling error. Suggestions (distance 1): ['lime', 'wime', 'nim',`
   `'sime','nide', 'Niue', 'name', 'dime', 'Nile', 'mime']...`
   `'karl': Spelling error. Suggestions (distance 0): ['Karl']`

3. `Enter a sentence to spell-check: Synchronize`
   `No spelling error.`

10 Practical Challenges Faced and Lessons Learnt

10.1 Practical Challenges Faced

During the course of this project, several practical obstacles were encountered in the process of implementing, analysing, and documenting the Levenshtein distance algorithms.

1. **Gathering Real-World Genome Datasets.** Obtaining accurate SARS-CoV-2 genome data for testing the algorithms required careful collection from public biological databases. Ensuring data integrity across variant samples involved additional preprocessing steps, making this a time-intensive part of the study.

2. **Computational Constraints on Large Datasets.** Executing the algorithms on massive genome sequences proved highly resource-intensive. For particularly long DNA samples, the dynamic programming routines took several hours to complete. This highlighted the computational limits of quadratic-time algorithms when applied to real biological data.

3. **Input Lag in Runtime Analysis.** While analyzing the runtime of the two-row Wagner–Fischer algorithm, it proved tricky to avoid input-related time lag caused by manual user entry. To ensure accurate benchmarking, a separate version of the code was written with preset input data, thereby eliminating delays from user interaction and allowing precise runtime measurement.

4. **Difficulties in Algorithm Typesetting.** Expressing the algorithms in formal LaTeX pseudocode format required heavy formatting. Ensuring consistent indentation, alignment of mathematical symbols, and readability within the `algorithmic` and `lstlisting` environments posed repeated challenges.

5. **Collecting and Translating Reference Material.** Curating appropriate literary and technical references in several languages demanded careful filtering of sources. Several foundational works and linguistic materials were not readily available in English, necessitating manual translation and paraphrasing to ensure accuracy.

10.2 Lessons Learnt

Despite the difficulties, the challenges offered substantial learning opportunities and practical insights.

1. **Practical Exposure to Real Data Handling.** The process of acquiring and processing authentic SARS-CoV-2 genome sequences revealed how theoretical algorithms must often be adapted to the quirks of real-world data, such as incomplete sequences, file encoding differences, and inconsistent annotations.

2. **Importance of Algorithmic Efficiency.** The long runtime of the initial implementations demonstrated the real-world consequences of time and space complexity, motivating the exploration of more time efficient or approximate algorithms such as Ukkonen's banded variant.

3. **Precision in Scientific Communication.** The process of typesetting algorithms reinforced the need for clarity and consistency in presenting technical material. Small formatting details often influenced how easily the logic could be understood by a reader.

4. **Value of Independent Research.** Working through translations and diverse references improved both analytical comprehension and research independence, illustrating how theoretical computer science often intersects with linguistic interpretation.

Overall, the experience demonstrated how theoretical understanding, practical implementation, and effective communication must converge to realise a complete computational study.

11 Conclusion

Working on this project gave us an in-depth understanding of the Levenshtein distance problem, not just as a mathematical idea but as a practical computational tool. Through the process, we explored and implemented several classical algorithms, including the Wagner–Fischer algorithm, its two-row optimization, Hirschberg's space-efficient divide-and-conquer method, and Ukkonen's banded approach. Each method revealed its own strengths and trade-offs in terms of time, memory, and ease of reconstruction.

The experiments showed that while the original Wagner–Fischer algorithm works well for short sequences, it quickly becomes slow and memory-heavy for large inputs such as genome data. The two-row version was a simple yet powerful improvement when only the edit distance was needed, while Hirschberg's algorithm balanced both space and accuracy. Ukkonen's method stood out for approximate matching — it was the most efficient when a small error tolerance was acceptable.

Applying these algorithms to real-world data, especially SARS-CoV-2 genome sequences, was a particularly valuable experience. It showed how theoretical computer science connects to real biological problems — for example, quantifying genetic differences across viral variants. Using Levenshtein distance for language translation data also revealed how similar principles can apply beyond biology, offering insights into inaccuracies in machine translations.

Beyond the technical side, this project taught us a great deal about the process of research itself — collecting datasets, managing computational limitations and learning new programming skills. It was equally about persistence and problem-solving as it was about algorithms and proofs. In the end, the project strengthened both our theoretical understanding and our practical confidence in designing, analyzing, and applying algorithms to real data.

References

[Hir75] Daniel S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, 1975.

[nih] NCBI Virus — ncbi.nlm.nih.gov. https://www.ncbi.nlm.nih.gov/labs/virus/vssi/#/virus?SeqType_s=Nucleotide&VirusLineage_ss=Severe%20acute%20respiratory%20syndrome%20coronavirus%202,%20taxid:2697049.

[Ukk85] Esko Ukkonen. Algorithms for approximate string matching. *Information and control*, 64(1-3):100–118, 1985.

[WF74] Robert A Wagner and Michael J Fischer. The string-to-string correction problem. *Journal of the ACM (JACM)*, 21(1):168–173, 1974.

[wik] Variants of SARS-CoV-2 - Wikipedia — en.wikipedia.org. https://en.wikipedia.org/wiki/Variants_of_SARS-CoV-2#. [Accessed 18-10-2025].