Edit Distance

**Edit Distance.** Given two strings \( S = s_1 \ldots s_n \) and \( T = t_1 \ldots t_m \), the **edit distance** between \( S \) and \( T \) is defined as the **smallest number of atomic edit operations** necessary to transform \( S \) into \( T \). The **atomic edit operations** are

- **Character insertion.** An insertion of a single character from the alphabet into any position in the string.
- **Character deletion.** A removal of any character from the string.
- **Character replacement.** A replacement of any character in the string with another character from the alphabet.

**Example.** Given a word "cat", the following words have an edit distance of 1 from it:

- "at", obtained from "cat" by deleting its first character:

  \[
  \begin{array}{c}
  \text{cat} \\
  X| | \\
  _| at
  \end{array}
  \]

- "cast", obtained from "cat" by inserting a character "s" into the third position of the string:

  \[
  \begin{array}{c}
  \text{ca}_t \\
  | | X| \\
  \text{cast}
  \end{array}
  \]

- "vat", obtained from "cat" by replacing the first character with "v":

  \[
  \begin{array}{c}
  \text{vat} \\
  \end{array}
  \]
Computing the Edit Distance. We want to develop a dynamic programming algorithm for computing the edit distance. In preparation for this, we will consider using a data structure similar to the one we used when solving the LCS problem.

Let \( c[i, j] \) be the edit distance between the prefixes \( S_i = s_1 \ldots s_i \) and \( T_j = t_1 \ldots t_j \) of the strings \( S \) and \( T \). Our algorithm will construct the table \( c[i, j] \). When completed, \( c[n, m] \) will contain the edit distance between \( S \) and \( T \).

The construction of \( c[i, j] \) is guided by the following observations:

- \( c[0, 0] = 0 \). For the sake of consistency, \( S_0 \) and \( T_0 \) are empty strings. The edit distance between two empty strings is 0.
- \( c[0, j] = j \) for all \( 1 \leq j \leq m \). The edit distance between an empty string and any non-empty string of length \( j \) is \( j \): the string can be constructed via \( j \) consecutive insertions.
- \( c[i, 0] = i \): see above (the empty string is constructed from \( s_1 \ldots s_i \) via \( i \) consecutive deletions).
- If \( s_i = t_j \), then \( c[i, j] = c[i - 1, j - 1] \). If the last characters of the two prefixes coincide, then the edit distance between them is the same as the edit distance between the prefixes without the last characters.
- If \( s_i \neq t_j \), then an atomic edit is needed to match the last characters of the strings \( S_i \) and \( T_j \). We must select one of the three possible atomic edits (insertion, deletion, or replacement). When selecting which one to use, we basically are reducing computing the edit distance between \( S_i \) and \( T_j \) to:
  1. computing the edit distance between \( S_{i-1} \) and \( T_{j-1} \) if replacement is chosen.
  2. computing the edit distance between \( S_{i-1} \) and \( T_j \) if deletion is chosen.
  3. computing the edit distance between \( S_i \) and \( T_{j-1} \) if insertion is chosen.

These insights can be properly encoded as follows:

\[
c[i, j] = \begin{cases} 
0 & \text{if } i = j = 0 \\
i & \text{if } j = 0 \\
 j & \text{if } i = 0 \\
c[i - 1, j - 1] & \text{if } i, j \geq 1 \text{ and } s_i = t_j \\
\min(c[i - 1, j - 1], c[i - 1, j], c[i, j - 1]) + 1 & \text{if } i, j \geq 1 \text{ and } s_i \neq t_j
\end{cases}
\]

Algorithm for Edit Distance Computation

Using the formula derived above, we can write the following algorithm for computing the table \( c[i, j] \). The algorithm returns \( c[n, m] \), which contains the edit distance between the input strings \( S \) and \( T \).
**Algorithm** `EditDistance(S = s_1 \ldots s_n, T = t_1 \ldots t_m)`

begin
\[ c[0..n,0..m] := 0; \]
for \( i = 0 \) to \( n \) do
\[ c[i, 0] := 0; \]
end for
for \( j = 1 \) to \( m \) do
\[ c[0, j] := 0; \]
end for
for \( i = 1 \) to \( n \) do
for \( j = 1 \) to \( m \) do
  if \( s_i = t_j \) then
    \[ c[i, j] := c[i-1, j-1]; \]
  else
    \[ c[i, j] := \min(c[i-1, j], c[i, j-1], c[i-1, j-1]) + 1; \]
  end if
end for
end for
return \[ c[n, m]; \]
end

**Analysis.** The double nested loop executes \( n \cdot m \) times. Each iteration runs in \( O(1) \). Therefore, the algorithmic complexity of the `EditDistance` algorithm is \( O(nm) \).