Algorithms: Design and Analysis - CS 412

Weekly Challenge 09: Dynamic Programming

Ali Muhammad Asad - aa07190

In compilers, the lexical analyzer scans the input source code character by character. When it encounters whitespace characters, it discards them without creating tokens. Whitespace characters do not contribute to the code's meaning and are often used for formatting and readability purposes only. After removing whitespace, the lexical analyzer continues to tokenize the remaining non-whitespace characters in the source code, identifying keywords, identifiers, literals, operators, and other language constructs.

The lexical analyzer generates the sequence of tokens (without spaces) as an output. Consider the following pair of correct and incorrect programming language statements represented as a sequence of tokens, and devise a dynamic programming algorithm to identify the minimum number of edits required to transform an incorrect sequence to the correct sequence.

Incorrect sequence: `<int><int><;>`
Correct sequence: `<int><var><.>`

1. Identify whether the problem has an optimal substructure property.
2. Identify the smallest subproblem.
3. Formulate a dynamic programming algorithm to solve the problem.
4. Generate a dynamic programming table for the given problem.

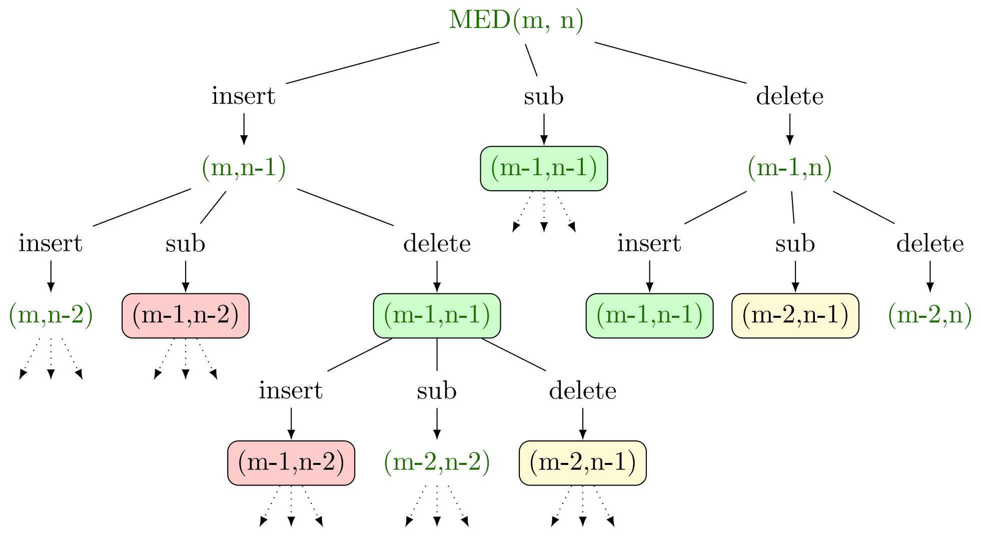
Solution: We are given two sequences; an incorrect sequence and a correct sequence, and we need to find the minimum number of edits required to transform the incorrect sequence to the correct sequence, including insertion, deletion, and substitutions. Thus, this problem can be easily reduced to the minimum edit distance problem that deals with finding the smallest number of edits; insertion, deletion, substitutions, required to transform one string into the other (which is exactly our problem). This problem already has an optimal substructure, and can be solved using dynamic programming. In this case, the strings are the incorrect and correct sequences of tokens.

For the sake of this weekly challenge, we will show its recursion tree and that it has an optimal substructure.

Consider two strings I and C representing the incorrect and correct sequences, respectively. Let m and n be the lengths of the incorrect and correct sequences, respectively. Then we can solve this problem by processing all characters one by one starting from either left or right sides of both the strings. For every pair of characters they either match or they don't match. If the last characters of both the strings match, then no operation is needed. So we recursively calculate the answer for the rest of the string. When last characters don't match, we can try and perform all three operations to match the last character, and then from there, recursively calculate the result for the remaining part of the string. Upon completion of these operations, the `min` of these values will be our final answer. We can do this easily with recursion:

- Match: when the last characters of the strings match, make a recursive call `MED(m-1,n-1)` to calculate the answer to the remaining parts of the string
- When they don't match, make 3 recursive calls:
  1. *insert* `I[n-1]` at last of `C`: `MED(m, n-1)`
  2. *replace* `C[m-1]` with `I[n-1]`: `MED(m-1,n-1)`
  3. *delete* `C[m-1]`: `MED(m-1,n)`

We can then make the recursion tree for our problem, `MED(m, n)` as follows:



1. By analyzing the above recursion tree, we can easily identify that it has an optimal substructure, since the minimum edits to transform a sequence `I[1..i]` to `C[1..i]` can be computed using the solutions of its subproblems, i.e., `I[1..i-1] to C[1..j]`, `I[1..i] to C[1..j - 1]`, `I[1..i-1] to C[1..j-1]`. Further, nodes with the same frame highlight show that there exists overlapping subproblems that are occurring on both sides of the tree, which again shows that there is an optimal substructure, and by solving the subproblems, we can easily solve the bigger problem.

2. The smallest subproblem is when either string is empty, i.e., $m = 0$, or $n = 0$ in which case the number of edits will be the length of the non-empty sequence as it requires that many characters to convert an empty string to a non-empty string.

3. We can use a bottom-up approach to solve this efficiently as follows:

```
MED(inc, cor)
  m <- inc.length
  n <- cor.length
  dp = [[0 for _ in range(n + 1)] for _ in range(m + 1)]

  for i <- 0 to m + 1
    for j <- 0 to n + 1
      if i is equal to 0:
        dp[i, j] = j
      else if j is equal to 0:
        dp[i, j] = i
      else if inc[i - 1] is equal to cor[j - 1]:
        dp[i, j] = dp[i - 1, j - 1]
      else:
        dp[i, j] = 1 + min(dp[i, j - 1], dp[i - 1, j], dp[i - 1, j - 1])

  return dp[m, n]
```

4. We have a table of `m x n`, and where `dp[i, j]` represents the minimum number of edits required to transform `I[1..i] to C[1..j]`. If `I[i - 1] = C[j - 1]`, then they match, so no additional edits required. If `i` or `j` is 0, then the edits are just the length of the other string. Else, we take the minimum of the three operations, *insert*, *delete*, and *replace* and add 1 to it to denote the addition in the edit distance. The final answer will be stored in `dp[m][n]`.

   For the given example, the table looks like this:

	#	**<**	**i**	**n**	**t**	**>**	**<**	**v**	**a**	**r**	**>**	**<**	**.**	**>**
#	0	1	2	3	4	5	6	7	8	9	10	11	12	13
<	1	0	1	2	3	4	5	6	7	8	9	10	11	12
i	2	1	0	1	2	3	4	5	6	7	8	9	10	11
n	3	2	1	0	1	2	3	4	5	6	7	8	9	10
t	4	3	2	1	0	1	2	3	4	5	6	7	8	9
>	5	4	3	2	1	0	1	2	3	4	5	6	7	8
<	6	5	4	3	2	1	0	1	2	3	4	5	6	7
i	7	6	5	4	3	2	1	1	2	3	4	5	6	7
n	8	7	6	5	4	3	2	2	2	3	4	5	6	7
t	9	8	7	6	5	4	3	3	3	3	4	5	6	7
>	10	9	8	7	6	5	4	4	4	4	3	4	5	6
<	11	10	9	8	7	6	5	5	5	5	4	3	4	5
;	12	11	10	9	8	7	6	6	6	6	5	4	4	5
>	13	12	11	10	9	8	7	7	7	7	6	5	5	4

   The '#' denotes an empty string.