

Name-Shiva Nema

Section-A3-B3

Rollno.-40

DAA LAB-PRACTICAL 5

1143. Longest Common Subsequence

Medium Topics Companies Hint

Given two strings `text1` and `text2`, return the length of their longest **common subsequence**. If there is no common subsequence, return 0.

A **subsequence** of a string is a new string generated from the original string with some characters (can be none) deleted without changing the relative order of the remaining characters.

- For example, `"ace"` is a subsequence of `"abcde"`.

A **common subsequence** of two strings is a subsequence that is common to both strings.

Example 1:

Input: `text1 = "abcde", text2 = "ace"`
Output: 3
Explanation: The longest common subsequence is "ace" and its length is 3.

Example 2:

Input: `text1 = "abc", text2 = "abc"`
Output: 3
Explanation: The longest common subsequence is "abc" and its length is 3.

Example 3:

Input: `text1 = "abc", text2 = "def"`
Output: 0
Explanation: There is no such common subsequence, so the result is 0.

```
class Solution:
    def longestCommonSubsequence(self, text1: str, text2: str) -> int:
        len_text1, len_text2 = len(text1), len(text2)
        dp_matrix = [[0] * (len_text2 + 1) for _ in range(len_text1 + 1)]
        for i in range(1, len_text1 + 1):
            for j in range(1, len_text2 + 1):
                if text1[i - 1] == text2[j - 1]:
                    dp_matrix[i][j] = dp_matrix[i - 1][j - 1] + 1
                else:
                    dp_matrix[i][j] = max(dp_matrix[i - 1][j], dp_matrix[i][j - 1])
        return dp_matrix[len_text1][len_text2]
```

Testcase Test Result

Accepted Runtime: 0 ms

Case 1 Case 2 Case 3

Input

text1 =
"abcde"

text2 =
"ace"

Output

```
[1]: from typing import List, Tuple

def lcs_table(X: str, Y: str) -> Tuple[List[List[int]], List[List[str]]:
    m, n = len(X), len(Y)
    dp = [[0] * (n + 1) for _ in range(m + 1)]
    direction = [[''] * (n + 1) for _ in range(m + 1)]
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if X[i - 1] == Y[j - 1]:
                dp[i][j] = dp[i - 1][j - 1] + 1
                direction[i][j] = '^'
            else:
                if dp[i - 1][j] >= dp[i][j - 1]:
                    dp[i][j] = dp[i - 1][j]
                    direction[i][j] = '↑'
                else:
                    dp[i][j] = dp[i][j - 1]
                    direction[i][j] = '←'
    return dp, direction

def lcs_backtrack(X: str, Y: str, dp: List[List[int]]) -> str:
    i, j = len(X), len(Y)
    lcs_chars: List[str] = []
    while i > 0 and j > 0:
        if X[i - 1] == Y[j - 1]:
            lcs_chars.append(X[i - 1])
            i -= 1
            j -= 1
        else:
            if dp[i - 1][j] >= dp[i][j - 1]:
                i -= 1
            else:
                j -= 1
    return ''.join(reversed(lcs_chars))

def print_cost_matrix_with_directions(X: str, Y: str, dp: List[List[int]], direction: List[List[str]]) -> None:
    m, n = len(X), len(Y)
    header = [" "]
    header += ["f" + " " * (ch) for ch in (' ' + Y)]
```

```

header += [f" {ch} " for ch in (' ' + Y)]
print(''.join(header))
for i in range(m + 1):
    row_label = ' ' if i == 0 else X[i - 1]
    line = [f" {row_label} "]
    for j in range(n + 1):
        arrow = direction[i][j]
        cell = f"{dp[i][j]}{arrow if not (i == 0 or j == 0) else '.'}"
        line.append(f"{cell:>4} ")
    print(''.join(line))

def lcs(X: str, Y: str) -> Tuple[int, str, List[List[int]], List[List[str]]:
    dp, direction = lcs_table(X, Y)
    seq = lcs_backtrack(X, Y, dp)
    return dp[-1][-1], seq, dp, direction

if __name__ == "__main__":
    X = "AGCCCTAAGGGCTACCTAGCTT"
    Y = "GACAGCCTACAAGCGTTAGCTTG"
    length, seq, dp, direction = lcs(X, Y)
    print("LCS Cost Matrix with Directions:")
    print_cost_matrix_with_directions(X, Y, dp, direction)
    print()
    print(f"Final LCS Length: {length}")
    print(f"LCS: {seq}")

```

LCS Cost Matrix with Directions:

	G	A	C	A	G	C	C	T	A	C	A	A	G	C	G	T	T	A	G	C	T	T	G
A	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
G	0	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
C	0	1	1	1	2	2	2	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
C	0	1	1	1	2	2	2	3	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
C	0	1	1	1	2	2	2	3	4	4	4	5	5	5	5	5	5	5	5	5	5	5	5
T	0	1	1	1	2	2	2	3	4	5	5	5	5	5	5	6	6	6	6	6	6	6	6
A	0	1	2	2	3	3	3	4	5	6	6	6	6	6	6	6	6	7	7	7	7	7	7
A	0	1	2	2	3	3	3	4	5	6	6	7	7	7	7	7	7	7	7	7	7	7	7
G	0	1	2	2	3	4	4	4	5	6	6	7	7	8	8	8	8	8	8	8	8	8	8
G	0	1	2	2	3	4	4	4	5	6	6	7	7	8	8	9	9	9	9	9	9	9	9
G	0	1	2	2	3	4	4	4	5	6	6	7	7	8	8	9	9	9	9	10	10	10	10
C	0	1	2	3	3	4	5	5	5	6	7	7	7	8	9	9	9	9	9	10	11	11	11
T	0	1	2	3	3	4	5	5	6	6	7	7	7	8	9	9	10	10	10	10	11	12	12
A	0	1	2	3	4	4	5	5	6	7	7	8	8	8	9	9	10	10	11	11	11	12	12
C	0	1	2	3	4	4	5	6	6	7	7	8	8	8	9	9	10	10	11	11	12	12	12
C	0	1	2	3	4	4	5	6	6	7	7	8	8	8	9	9	10	10	11	11	12	12	12
T	0	1	2	3	4	4	5	6	7	7	8	8	8	8	9	9	10	11	11	12	13	13	13
A	0	1	2	3	4	4	5	6	7	7	8	8	9	9	9	9	10	11	11	12	12	13	13
G	0	1	2	3	4	5	5	6	7	7	8	8	9	9	10	10	10	11	11	12	13	13	14
C	0	1	2	3	4	5	6	6	7	8	8	9	9	10	11	11	11	12	13	14	14	14	14
T	0	1	2	3	4	5	6	7	7	8	9	9	9	10	11	11	12	12	13	14	15	15	15
T	0	1	2	3	4	5	6	7	8	8	9	9	10	11	11	12	13	13	14	15	16	16	16

Final LCS Length: 16

LCS: AGCCCAAGGTTAGCTT

```
[2]: from typing import List, Tuple

def lrs_table(S: str) -> List[List[int]]:
    n = len(S)
    dp = [[0] * (n + 1) for _ in range(n + 1)]
    for i in range(1, n + 1):
        for j in range(1, n + 1):
            if S[i - 1] == S[j - 1] and i != j:
                dp[i][j] = 1 + dp[i - 1][j - 1]
            else:
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])
    return dp

def lrs_backtrack(S: str, dp: List[List[int]]) -> str:
    i, j = len(S), len(S)
    out: List[str] = []
    while i > 0 and j > 0:
        if S[i - 1] == S[j - 1] and i != j:
            out.append(S[i - 1])
            i -= 1
            j -= 1
        else:
            if dp[i - 1][j] >= dp[i][j - 1]:
                i -= 1
            else:
                j -= 1
    return ''.join(reversed(out))

def lrs(S: str) -> Tuple[int, str, List[List[int]]]:
    dp = lrs_table(S)
    seq = lrs_backtrack(S, dp)
    return dp[-1][-1], seq, dp

def print_matrix(S: str, dp: List[List[int]]) -> None:
    n = len(S)
    header = [" " * n] + [f" {ch} " for ch in (' ' + S)]
    print(''.join(header))
    for i in range(n + 1):
        row_label = ' ' if i == 0 else S[i - 1]
```

```

def lrs(S: str) -> Tuple[int, str, List[List[int]]]:
    dp = lrs_table(S)
    seq = lrs_backtrack(S, dp)
    return dp[-1][-1], seq, dp

def print_matrix(S: str, dp: List[List[int]]) -> None:
    n = len(S)
    header = [" " + ch for ch in S]
    print(''.join(header))
    for i in range(n + 1):
        row_label = ' ' if i == 0 else S[i - 1]
        line = [f"{row_label} " + str(dp[i][j]) for j in range(n + 1)]
        for j in range(n + 1):
            line.append(f"{dp[i][j]:>4} ")
        print(''.join(line))

if __name__ == "__main__":
    S = "AABEBCDD"
    length, seq, dp = lrs(S)
    print("LRS DP Matrix (values):")
    print_matrix(S, dp)
    print()
    print(f"Final LRS Length: {length}")
    print(f"LRS: {seq}")

```

LRS DP Matrix (values):

		A	A	B	E	B	C	D	D
	0	0	0	0	0	0	0	0	0
A	0	0	1	1	1	1	1	1	1
A	0	1	1	1	1	1	1	1	1
B	0	1	1	1	1	2	2	2	2
E	0	1	1	1	1	2	2	2	2
B	0	1	1	2	2	2	2	2	2
C	0	1	1	2	2	2	2	2	2
D	0	1	1	2	2	2	2	2	3
D	0	1	1	2	2	2	2	3	3

Final LRS Length: 3
LRS: ABD