DAA CT4 ASSIGNMENT

SUBMITTED BY: HARSHIKA HASIJA(RA2111003010724)

KHAYATI SHARMA(RA2111003010710)

PROJECT TITLE- DNA SEQUENCE MATCHING

ALGORITHM USED- LONGEST COMMON SUBSEQUENCE(DYNAMIC AND RECURSIVE APPROACH)

LCS algorithm is an algorithm used to find the longest subsequence that is present in two given sequences. A subsequence is a sequence that can be derived from another sequence by deleting some or no elements without changing the order of the remaining elements.

TABLE OF CONTENTS

S.NO.	TITLE	PAGE NO.
1.	PROBLEM STATEMENT	3
2.	1) LCS(DYNAMIC APPROACH)	4
3.	1.1)DRY RUN	7
4.	1.2)TIME COMPLEXITY	8
5.	2)LCS(RECURSIVE APPROACH)	9
6.	2.1)DRY RUN	11
7.	2.2)TIME COMPLEXITY	12
8.	CONCLUSION	13

PROBLEM STATEMENT

The problem of DNA sequence matching involves identifying similarities between two or more DNA sequences. This is an important problem in genetics and bioinformatics, as it helps researchers understand the relationship between different organisms and the function of specific genes.One popular algorithm for solving this problem is the Longest Common Subsequence (LCS) algorithm, which identifies the longest subsequence that is common to two or more sequences. The LCS algorithm can be implemented using both dynamic and recursive approaches.In the dynamic approach, we use a two-dimensional array to store the lengths of the LCS between all possible pairs of prefixes of the input sequences. We then use this array to construct the LCS by backtracking from the bottom-right corner to the top-left corner.In the recursive approach, we define a recursive function that computes the LCS of two sequences by breaking the sequences into smaller and smaller subproblems. We then memoize the results of these subproblems to avoid redundant computation.

DYNAMIC APPROACH

ALGORITHM

To implement the dynamic programming approach for LCS on DNA sequences, you can follow these steps:

1. Define the input sequences: In the case of DNA sequences, you can represent them as strings of characters. For example, "ACGT" represents a DNA sequence that contains the nucleotides Adenine, Cytosine, Guanine, and Thymine.
2. Initialize the LCS table: Create a 2-dimensional table with dimensions (m+1)x(n+1), where m and n are the lengths of the two input sequences. Initialize the first row and the first column of the table to 0, as they correspond to empty prefixes.
3. Fill in the LCS table: Iterate over the table starting from the second row and second column. For each cell (i,j), if the characters at the i-th position in the first sequence and the j-th position in the second sequence match, set the value of the cell to the value of the diagonal cell (i-1,j-1) plus 1. Otherwise, set the value of the cell to the maximum value between the cell above (i-1,j) and the cell to the left (i,j-1).
4. Backtrack to find the LCS: Once the LCS table is filled, you can backtrack to find the actual LCS. Start at the bottom-right corner of the table (i.e., the cell (m+1,n+1)) and follow the path that gives the maximum LCS score. Whenever you move diagonally, add the character at the i-th position of the first sequence to the LCS.
5. Return the LCS: Once you reach the top-left corner of the table (i.e., the cell (1,1)), the LCS will be the reverse of the sequence you have constructed while backtracking.

PSEUDOCODE

```
function lcs(seq1, seq2):
    m = length(seq1)
    n = length(seq2)
     lcs_table = create_table(m+1, n+1)  // create a table with m+1 rows and n+1 columns
    for i = 0 to m:
        lcs_table[i,0] = 0  // initialize the first column to 0
    for j = 0 to n:
        lcs_table[0,j] = 0  // initialize the first row to 0
    for i = 1 to m:
        for j = 1 to n:
            if seq1[i] == seq2[j]:
                lcs_table[i,j] = lcs_table[i-1,j-1] + 1
            else:
                lcs_table[i,j] = max(lcs_table[i-1,j], lcs_table[i,j-1])
     lcs_length = lcs_table[m,n]  // the length of the LCS is in the bottom-right corner of the table
    lcs_seq = create_string(lcs_length)  // create a string to store the LCS
    i = m
    j = n
    while i > 0 and j > 0:
        if seq1[i] == seq2[j]:
```

```
            lcs_seq[lcs_length] = seq1[i]
            i = i - 1
            j = j - 1
            lcs_length = lcs_length - 1
        else:
            if lcs_table[i-1,j] > lcs_table[i,j-1]:
                i = i - 1
            else:
                j = j - 1
    return lcs_seq
```

DRY RUN

Taking an example here to show how DNA sequences can be matched using the dynamic approach of LCS algorithm.

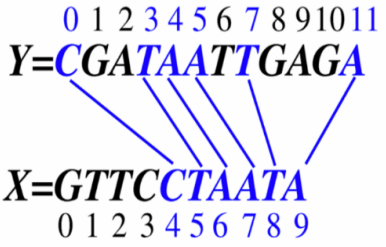
This sequence is made up of a string of nucleotides (A, T, C, and G) that represent the genetic information of an organism.

Y = CGATAATTGAGA
X = GTTCCTAATA

1. Create a table of size (m+1) x (n+1), where m and n are the lengths of Y and X, respectively. Initialize the first row and column of the table with 0's.

2. Iterate over each position (i, j) in the table, starting with (1, 1).
a. If Y[i-1] is equal to X[j-1], set table[i][j] to the value in the diagonal cell plus 1. Otherwise, set table[i][j] to the maximum of the value in the cell to the left and the value in the cell above it.
b. Repeat step (2a) for all positions (i, j) in the table.

L	-1	0	1	2	3	4	5	6	7	8	9	10	11
-1	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	1	1	1	1	1	1	1	1	1	1
1	0	0	1	1	2	2	2	2	2	2	2	2	2
2	0	0	1	1	2	2	2	3	3	3	3	3	3
3	0	1	1	1	2	2	2	3	3	3	3	3	3
4	0	**1**	1	1	2	2	2	3	3	3	3	3	3
5	0	1	1	1	**2**	2	2	3	4	4	4	4	4
6	0	1	1	2	2	**3**	3	3	4	4	5	5	5
7	0	1	1	2	2	3	**4**	4	4	4	5	5	6
8	0	1	1	2	3	3	4	5	**5**	5	5	5	6
9	0	1	1	2	3	4	4	5	5	5	6	6	**6**



TIME COMPLEXITY

- We have two nested loops
  The outer one iterates n times
  The inner one iterates m times
- A constant amount of work is done inside each iteration of the inner loop
- Thus, the total running time is O(nm)
- Answer is contained in L[n,m] (and the subsequence can be recovered from the L table).

RECURSIVE APPROACH
ALGORITHM

To implement the recursive programming approach for LCS on DNA sequences, you can follow these steps:

1. Define a function called lcs that takes two input sequences, seq1 and seq2, and returns the longest common subsequence of those sequences as a string.
2. Get the length of seq1 and seq2 and store them in variables m and n, respectively.
3. Check if either seq1 or seq2 is an empty sequence. If so, return an empty string, since the LCS of an empty sequence and any other sequence is an empty sequence.
4. Check if the last character of seq1 is equal to the last character of seq2. If so, recursively compute the LCS of seq1 with the last character removed and seq2 with the last character removed, and append the last character to the end of the LCS. This is because the LCS of two sequences that end in the same character is the LCS of the two sequences with that character removed, followed by that character.
5. If the last characters of seq1 and seq2 do not match, recursively compute the LCS of seq1 with the last character removed and seq2, and the LCS of seq1 and seq2 with the last character of seq2 removed. Compare the lengths of the resulting LCSs, and return the longer of the two.

PSEUDOCODE

```
function lcs(seq1, seq2):
    m = length(seq1)
    n = length(seq2)

    // If either sequence is empty, return an empty string
    if m == 0 or n == 0:
        return ""

    // If the last characters of seq1 and seq2 match, append the character to the LCS
    if seq1[m-1] == seq2[n-1]:
        return lcs(seq1[0:m-1], seq2[0:n-1]) + seq1[m-1]

     // If the last characters of seq1 and seq2 do not match, recursively call lcs on the two resulting sequences
    else:
        lcs1 = lcs(seq1[0:m-1], seq2)
        lcs2 = lcs(seq1, seq2[0:n-1])
        // Return the longer LCS of the two resulting sequences
        if length(lcs1) > length(lcs2):
            return lcs1
        else:
            return lcs2
```

DRY RUN

Taking an example here to show how DNA sequences can be matched using the recursive approach of LCS algorithm.

Sequence 1: ACGTACAG
Sequence 2: ACTAAGTCA

We start with the first character of both sequences:

- First character of sequence 1: A
- First character of sequence 2: A

Since both characters are the same, the length of LCS becomes 1 + LCS(ACGTACAG, ACTAAGTCA)

We move to the second character of both sequences:

- Second character of sequence 1: C
- Second character of sequence 2: C

Since both characters are the same, the length of LCS becomes 1 + LCS(GTACAG, GTAAGTCA)

We move to the third character of both sequences:

- Third character of sequence 1: G
- Third character of sequence 2: T

Since both characters are different, we have two options:

1. Remove the third character from sequence 1, so the problem reduces to LCS(ACGACAG, ACTAAGTCA)
2. Remove the third character from sequence 2, so the problem reduces to LCS(ACGTACAG, ACTAGTCA)

We calculate the length of LCS for both the options and choose the maximum:

- LCS(ACGACAG, ACTAAGTCA) = LCS(CGACAG, CTAAGTCA) = 2
- LCS(ACGTACAG, ACTAGTCA) = LCS(ACGACAG, ACTAGTCA) = 3

Therefore, the length of LCS becomes the maximum, which is 3.

Following this approach, we can recursively find the length of LCS for the entire sequence. However, note that the recursive approach has exponential time complexity and is not efficient for larger sequences.

TIME COMPLEXITY

The time complexity of the above algorithm is O(2^n) where n is the length of the input sequences. This is because for each element in the first sequence, we have two recursive calls, one with the next element of the first sequence and the same element of the second sequence, and another with the same element of the first sequence and the next element of the second sequence. This results in an exponential time complexity.

CONCLUSION

COMPARING THE TIME COMPLEXITIES TO FIND THE BETTER APPROACH FOR DNA MATCHING

The time complexity of the dynamic programming approach is O(mn), where m and n are the lengths of the input sequences. The dynamic programming approach uses a table to store previously computed LCS values, so that each subproblem is computed only once. The table is filled in a row-by-row or column-by-column manner, and each cell in the table is computed in constant time based on the values of previously computed cells.

On the other hand, the time complexity of the recursive approach is O(2^n). The recursive approach involves a large number of redundant function calls and subproblems, which results in exponential time complexity.

Therefore, the dynamic programming approach is preferred over the recursive approach for practical purposes, especially when dealing with longer sequences.