What is LCS( longest common subsequence)

LCS using recursion

LCS using dynamic programing ( memorization top down method)

LCS using dynamic programing (Tabulation bottom up method)

Longest Common Subsequence

In this study we will see different solution of longest common subsequence algorithm and difference between their time complexity. The longest common subsequence is an algorithm which finds the common string subsequence between two or more strings. For an example: two strings abcdefghi and

cdgi has the longest common sequence cdgi. An recursive method is given billow of this problem:

Let us consider two arrays A and B.

|  |  |
| --- | --- |
| b | d |

|  |  |  |  |
| --- | --- | --- | --- |
| a | b | c | d |

The longest common subsequence is b,d .

int Lcs(i,j)

{

if (A[i]== '/0' || B[j] == '/0')

return 0;

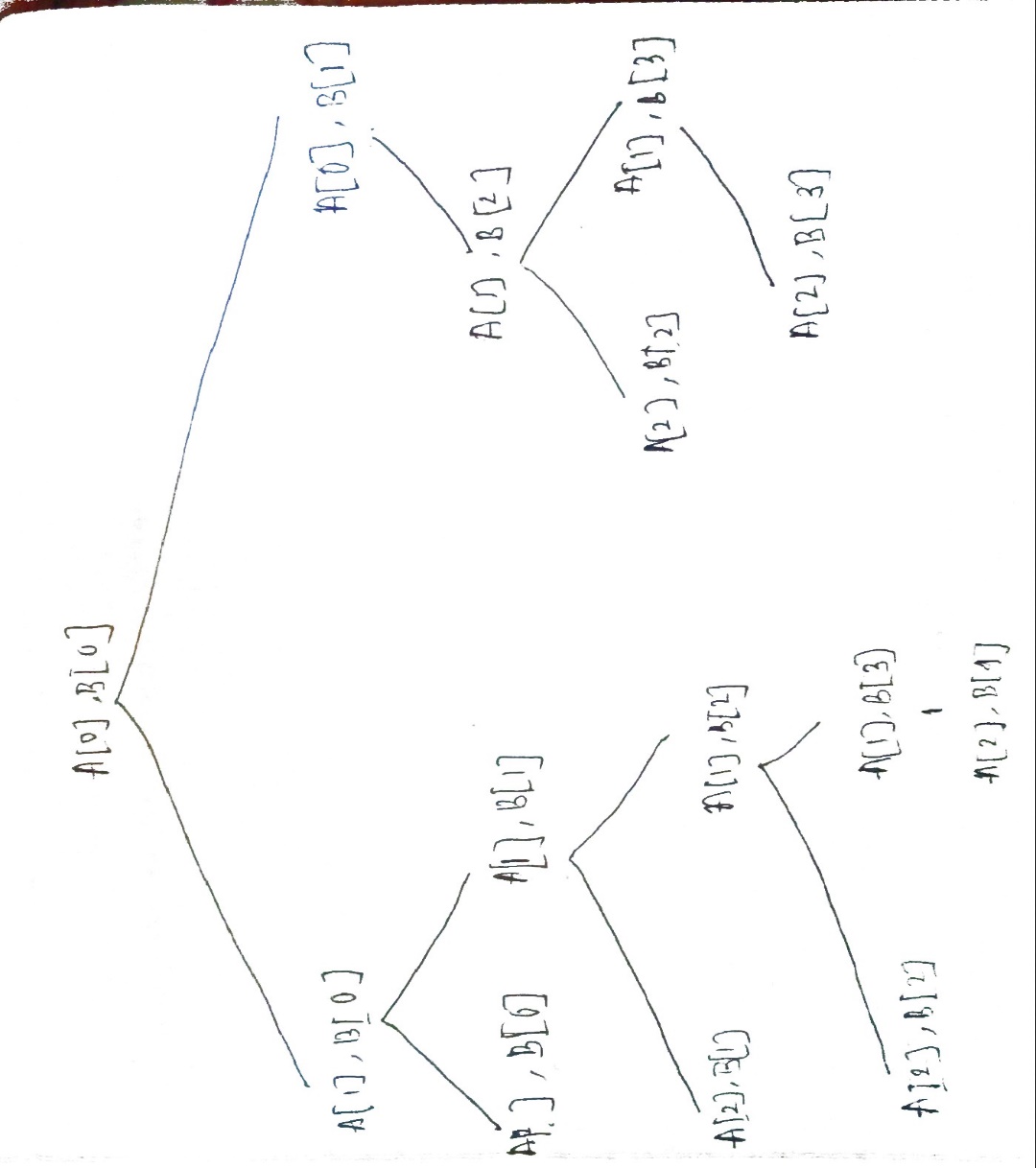
else if(A[i] == B[j])

return 1=Lcs(i+1,j+1)

else

return max(Lcs(i+1,j), Lcs(i,j+1));

}



Here we can see same task is happening in the right sub tree . This method is too slow and exponential time complexity . In worse case when there is no matching common subsequence between A and B time complexity of the algorithm is : 2m\*n

We can do it with linear time with memorization and other dynamic programing methods.

Longest common subsequence using memoization(Top Down)

int lcs(string X, string Y, int m, int n, int dp[][maximum])

{

    // base case

    if (m == 0 || n == 0)

        return 0;

    // if the same state has already been

    // computed

    if (dp[m - 1][n - 1] != -1)

        return dp[m - 1][n - 1];

    // if equal, then we store the value of the

    // function call

    if (X[m - 1] == Y[n - 1]) {

        // store it in arr to avoid further repetitive

        // work in future function calls

        dp[m - 1][n - 1] = 1 + lcs(X, Y, m - 1, n - 1, dp);

        return dp[m - 1][n - 1];

    }

    else {

        // store it in arr to avoid further repetitive

        // work in future function calls

        dp[m - 1][n - 1] = max(lcs(X, Y, m, n - 1, dp),

                               lcs(X, Y, m - 1, n, dp));

        return dp[m - 1][n - 1];

    }

}

Use a 2-D array to store the computed lcs(m, n) value at arr[m-1][n-1] because the string index starts from 0.

Whenever the function with an equivalent argument m and n are called again, don't perform any longer recursive call and return arr[m-1][n-1] because the previous computation of the lcs(m, n) has already been stored in arr[m-1][n-1], hence reducing the recursive calls that happen more then once.

**Time Complexity:** O(M\*N), where M and N is length of the first and second string respectively.

**Auxiliary Space:** (M\*N)

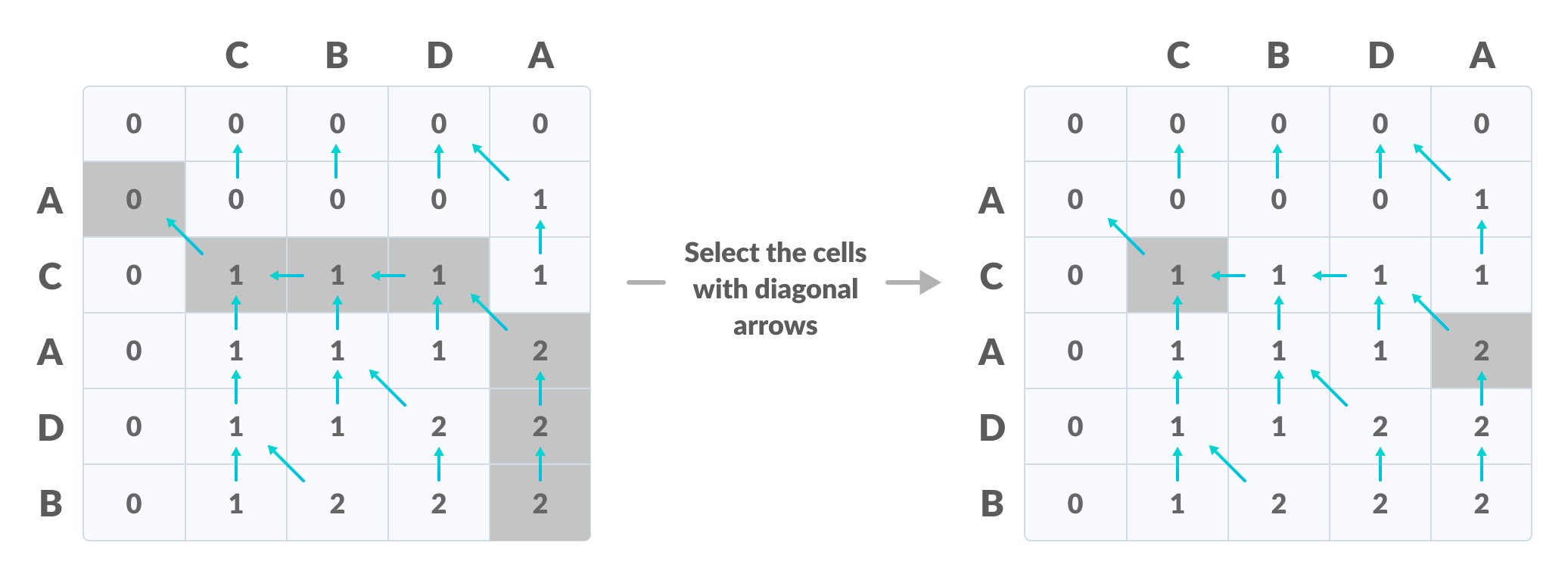
LCS with tabulation method ( Bottom Up)

First we create a table of dimension n+1\*m+1 where n and m are the lengths of strings X and Y respectively. The first row and the first column are filled with 0.

If the character of previous rows and column are matching then fill the slot with previous diagonal slots number and adding 1 with it.

Else fill the slot with the maximum number of previous slot of rows and column.

After filling the table the last number of last row and last column is the length of longest common subsequence . And if we want to find the sequence we need to backtrack the table.



Here the longest common subsequence is CA.

void lcs(char \*S1, char \*S2, int m, int n) {

int LCS\_table[m + 1][n + 1];

// Building the mtrix in bottom-up way

for (int i = 0; i <= m; i++) {

for (int j = 0; j <= n; j++) {

if (i == 0 || j == 0)

LCS\_table[i][j] = 0;

else if (S1[i - 1] == S2[j - 1])

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]);

}

}

int index = LCS\_table[m][n];

char lcs[index + 1];

lc[index] = '\0';

int i = m, j = n;

while (i > 0 && j > 0) {

if (S1[i - 1] == S2[j - 1]) {

lcs[index - 1] = S1[i - 1];

i--;

j--;

index--;

}

else if (LCS\_table[i - 1][j] > LCS\_table[i][j - 1])

i--;

else

j--;

}

// Printing the sub sequences

cout << "S1 : " << S1 << "\nS2 : " << S2 << "\nLCS: " << lcsAlgo << "\n";

}

The time complexity of this method is O(m\*N) . m stands for rows and n stands of column. Because we need to backtrack m\*n element in the worst case.