Longest Common Subsequence

Ibrahim Albluwi

Definition. A subsequence of a sequence $X = < x_1, x_2, x_3, \ \dots \ , x_n >$ is a sequence produced from X by deleting zero or more elements (without changing the order of the elements).

> **Example.**
> $X = ABCA$
> Subsequences of X : $\{ABCA, ABC, ABA, ACA, BCA, AB, AA, CA, BA, BC, AC, A, B, C, \phi\}$

Problem Statement. Given two sequences $X = < x_1, x_2, x_3, \ \dots \ , x_n >$ and $Y = < y_1, y_2, y_3, \ \dots \ , y_m >$, find the length of the longest common subsequence between X and Y.

> **Example.**
> $X = ABCABC, \ \ Y = BCABCA$
> Examples of *common* subsequences: $\{A, B, C, ABC, ABCA, BCA, BC, BCABC, CAB, CABC\}$
> Longest Common Subsequence: $BCABC$ of length 5.

Applications. This is a classic computer science problem with many applications. It is widely used in Bioinformatics when comparing, matching, and searching for DNA fragments and in applications that require comparing text files, like version control systems (e.g. Git) and plagiarism detectors.

Brute Force Solution.

```
BRUTEFORCE_LCS(X, Y):
LCS = ϕ
FOR every subsequence S in X:
     IF SEARCH(S, Y):
          IF LENGTH(S) > LENGTH(LCS):
               LCS = S
```

```
SEARCH(S, Y):
i = 1
j = 1
WHILE j <= LENGTH(Y) AND i <= LENGTH(S):
     IF S[i] == Y[j] : ++i, ++j
     ELSE            : ++j
IF i > LENGTH(S): RETURN TRUE
ELSE            : RETURN FALSE
```

Running Time of the Brute Force Solution. There are 2^n subsequences in X. Checking if a subsequence of X is also a subsequence of Y requires $O(m)$ comparisons. The total is $O(m2^n)$.

Definition. Given two sequences $X = < x_1, x_2, x_3, \ \dots\ , x_n >$ and $Y = < y_1, y_2, y_3, \ \dots\ , y_m >$,
We define `LCS(i, j)` to be the length of the longest common subsequence between the first `i` characters of X and the first `j` characters of Y.

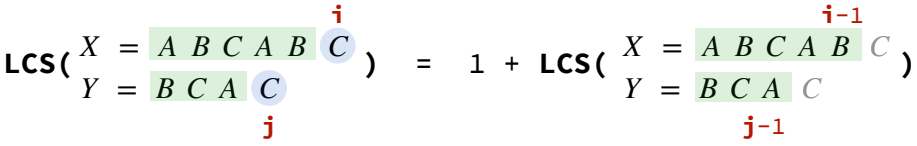
Hence, we are interested in solving `LCS(n, m)`.

Optimal Substructure.

```
LCS(i, j)  = 0                                  if  i == 0 or j == 0   (1)
           = 1 + LCS(i-1, j-1)                  if  X[i] == Y[j]       (2)
           = MAX(LCS(i-1, j), LCS(i, j-1))      if  X[i] != Y[j]       (3)
```

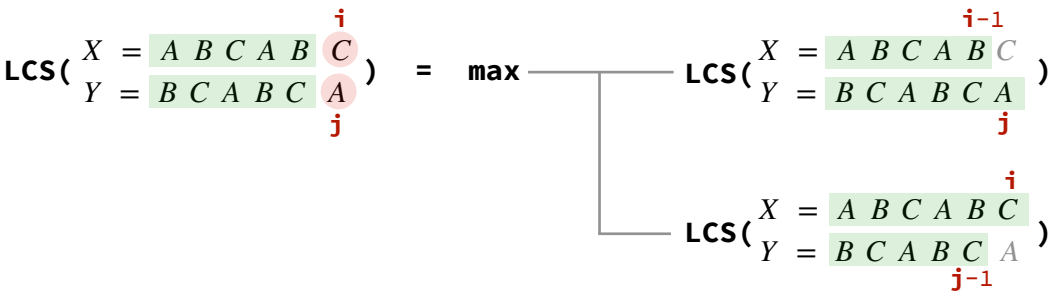
Case 1. The `LCS` is `0` because we are not considering any character from at least one of the two sequences.

Case 2.



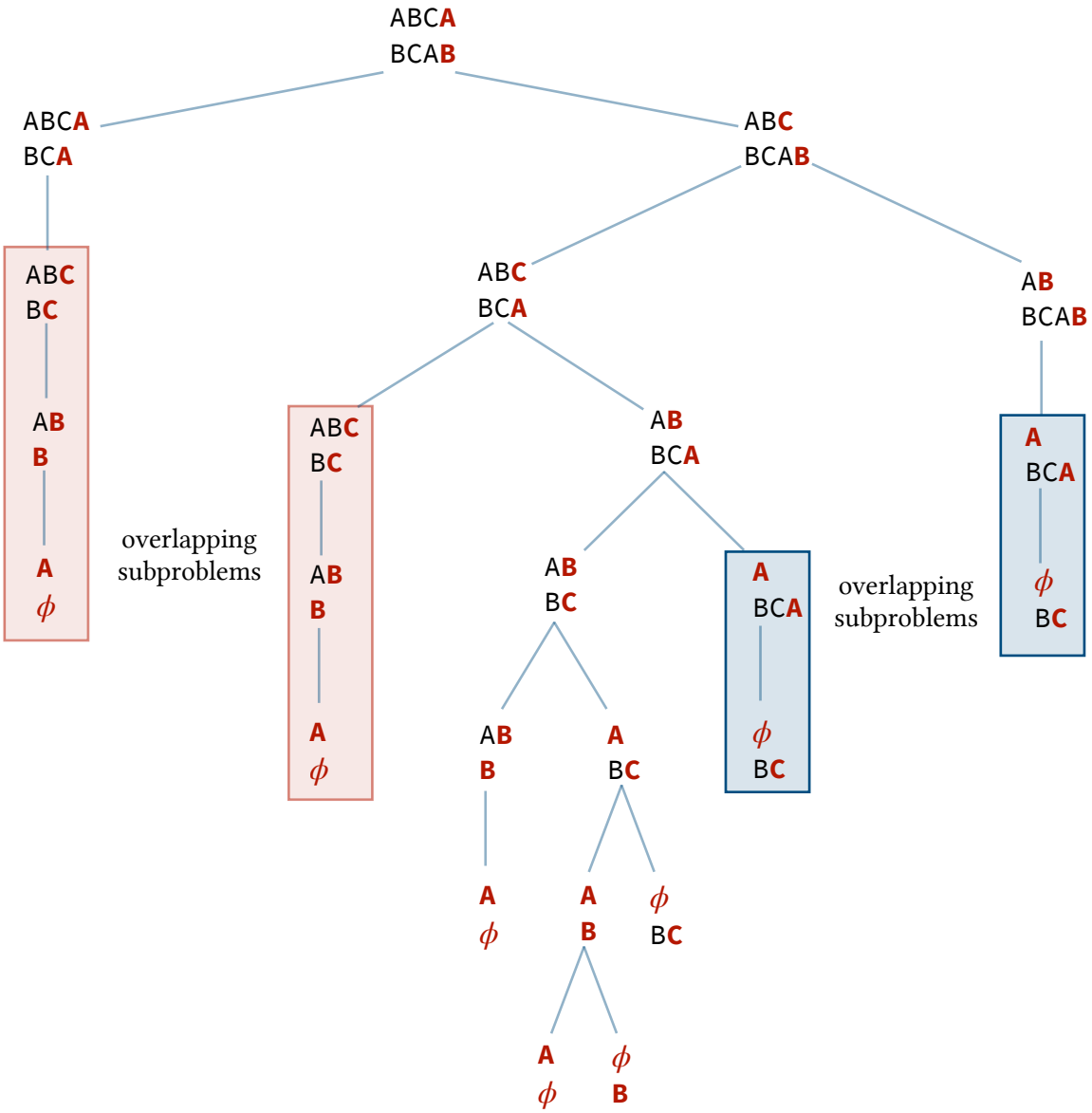
Since the last two characters match, we know for sure that the `LCS` ends with these two characters. Therefore, we can find the `LCS` between the rest of the characters in X and the rest of the characters in Y and then add `1` to the result.

Case 3.

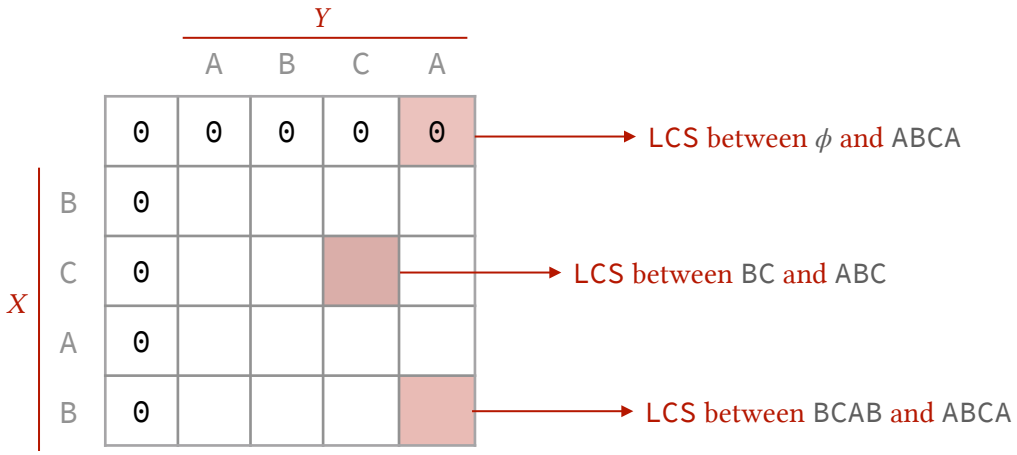


Since the last two characters do not match, we know for sure that *at least* one of them is not the last character in the `LCS`, but we don't know which. Therefore, we try to solve the problem once ignoring the last character in X and once ignoring the last character in Y. We then pick the choice that resulted in the maximum result.

Example Trace. The following is an illustration for the LCS between $Y = ABCA$ and $X = BCAB$. The highlighted branches are overlapping subproblems.



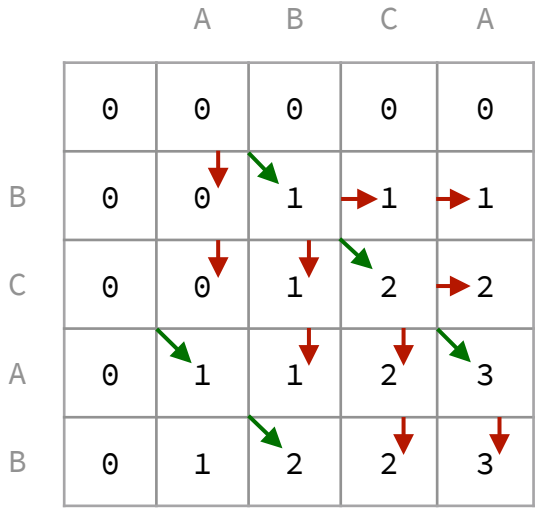
Bottom-up Dynamic Programming. To avoid computing the same subproblem twice, we store the solutions for the subproblems in a table and solve the subproblems from smallest to largest. The table stores at index `[i][j]` the result of `LCS(i, j)`.




```
COMPUTE_LCS(X, Y, n, m, i, j)

Create an array LCS[n+1][m+1]
Initialize the first row and first column to 0's
FOR i = 1 to n:
      FOR j = 1 to m:
            IF X[i] == Y[j]:
                  LCS[i][j] = 1 + LCS[i-1][j-1]
            ELSE:
                  LCS[i][j] = MAX(LCS[i-1][j], LCS[i][j-1])
RETURN LCS[n][m]
```

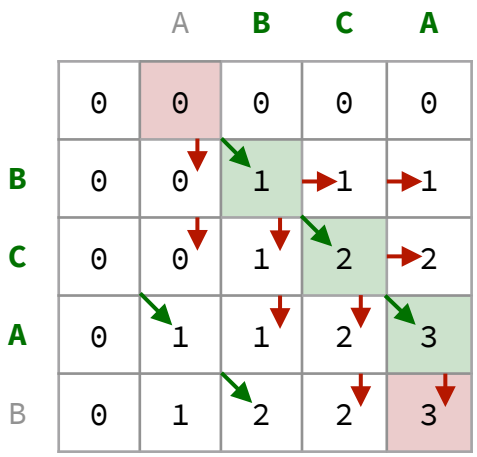
Example Trace. The following table shows the result of finding the `LCS` between `ABCA` and `BCAB`. A red arrow indicates a mismatch between the `X[i]` and `Y[j]` and a green arrow indicates a match.



Finding the Longest Common Subsequence. The result at index `[n][m]` is the length of the longest common subsequence. To find the characters constituting the `LCS`, we can follow the path that led to the result at index `[n][m]`.

```
REBUILD_LCS(X, Y, n, m, i, j, LCS[][])
```

```
i = n, j = m
WHILE i > 0 AND j > 0
    IF X[i] == Y[j]:
          result = X[i] + result
          i--
          j--
    ELSEIF LCS[i-1][j] >= LCS[i][j-1]:
          i--
    ELSE: j--
RETURN result
```



Running Time.

Building and filling the table requires $\Theta(n\,m)$ work.

Finding characters constituting the `LCS` requires $O(n+m)$ work.