Longest Common Subsequence (LCS)

Name: Cristhian Wiki Sánchez Sauñe

LCS Problem Statement

Given two sequences, find the length of longest subsequence present in both of them. A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous. For example, “abc”, “abg”, “bdf”, “aeg”, ‘”acefg”, .. etc are subsequences of “abcdefg”.

In order to find out the complexity of brute force approach, we need to first know the number of possible different subsequences of a string with length n, i.e., find the number of subsequences with lengths ranging from 1, 2, .. n-1.

LCS Problem Statement

Recall from theory of permutation and combination that number of combinations with 1 element are ${}^{n}C_1$. Number of combinations with 2 elements are ${}^{n}C_2$ and so forth and so on.

$$ {}^{n}C_0 + {}^{n}C_1 + {}^{n}C_2 + \ldots {}^{n}C_n = 2^n $$

So a string of length ***n*** has $2^n - 1$ different possible subsequences since we do not consider the subsequence with length 0.

LCS Problem Statement

This implies that the time complexity of the brute force approach will be $O(n * 2^n)$.

Take **O(*n*)** time to check if a subsequence is common to both the strings.

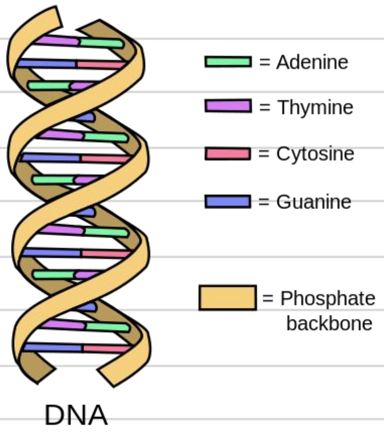
Examples:

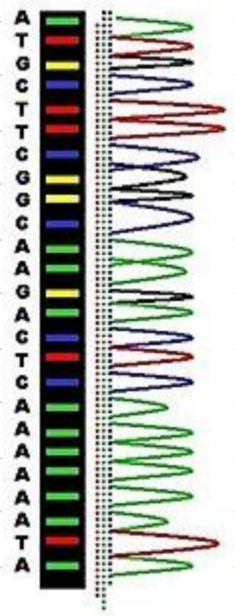
- LCS for input Sequences “ABCDGH” and “AEDFHR” is “ADH” of length 3.
- LCS for input Sequences “AGGTAB” and “GXTXAYB” is “GTAB” of length 4.

Applications

1. **Molecular biology**

DNA sequences (genes) can be represented as sequences of four letters ACGT, corresponding to the four sub molecules forming DNA. When biologists find a new sequences, they typically want to know what other sequences it is most similar to. One way of computing how similar two sequences are is to find the length of their longest common subsequence.



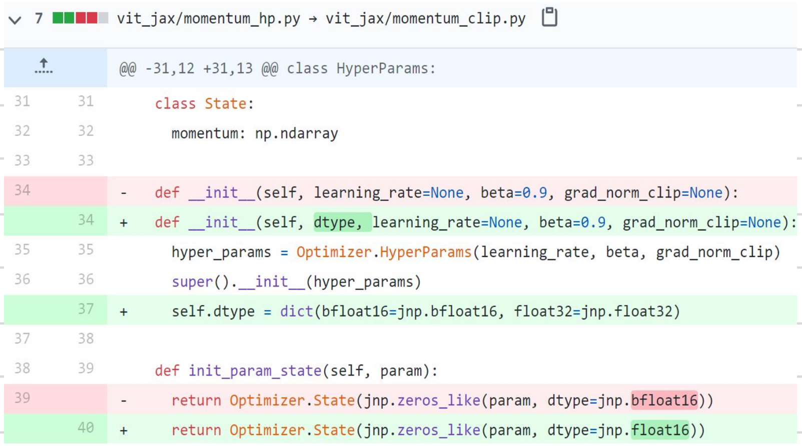


Applications

2. **File comparison**

Github has an option that allows you to two different versions of the same file, to determine what changes have been made to the file.

It works by finding a longest common subsequence of the lines of the two files; any line in the subsequence has not been changed, so what it displays is the remaining set of lines that have changed.



```
7 vit_jax/momentum_hp.py → vit_jax/momentum_clip.py
@@ -31,12 +31,13 @@ class HyperParams:
   class State:
     momentum: np.ndarray

-  def __init__(self, learning_rate=None, beta=0.9, grad_norm_clip=None):
+  def __init__(self, dtype, learning_rate=None, beta=0.9, grad_norm_clip=None):
     hyper_params = Optimizer.HyperParams(learning_rate, beta, grad_norm_clip)
     super().__init__(hyper_params)
+    self.dtype = dict(bfloat16=jnp.bfloat16, float32=jnp.float32)

   def init_param_state(self, param):
-    return Optimizer.State(jnp.zeros_like(param, dtype=jnp.bfloat16))
+    return Optimizer.State(jnp.zeros_like(param, dtype=jnp.float16))
```

Dynamic Programming

Let us see how this problem possesses both important properties of a Dynamic Programming (DP) Problem.

1. **Optimal Substructure:**

Let the input sequences be X[0..m-1] and Y[0..n-1] of lengths m and n respectively. And let L(X[0..m-1], Y[0..n-1]) be the length of LCS of the two sequences X and Y. Following is the recursive definition of L(X[0..m-1], Y[0..n-1]).

- If last characters of both sequences match (or X[m-1] == Y[n-1]) then
  L(X[0..m-1], Y[0..n-1]) = 1 + L(X[0..m-2], Y[0..n-2])

- If last characters of both sequences do not match (or X[m-1] != Y[n-1]) then
  L(X[0..m-1], Y[0..n-1]) = *max* (L(X[0..m-2], Y[0..n-1]), L(X[0..m-1], Y[0..n-2]))

Dynamic Programming

Examples:

a) Consider the input strings “AGGTAB” and “GXTXAYB”.
Last characters match for the strings. So length of LCS can be written as:

L(“AGGTAB”, “GXTXAYB”) = 1 + L(“AGGTA”, “GXTXAY”)

b) Consider the input strings “DGH” and “FHR. Last characters do not match for the strings. So length of LCS can be written as:

L(“DGH”, “FHR”) = *max* (L(“DG”, “FH**R**”), L(“DG**H**”, “FH”))

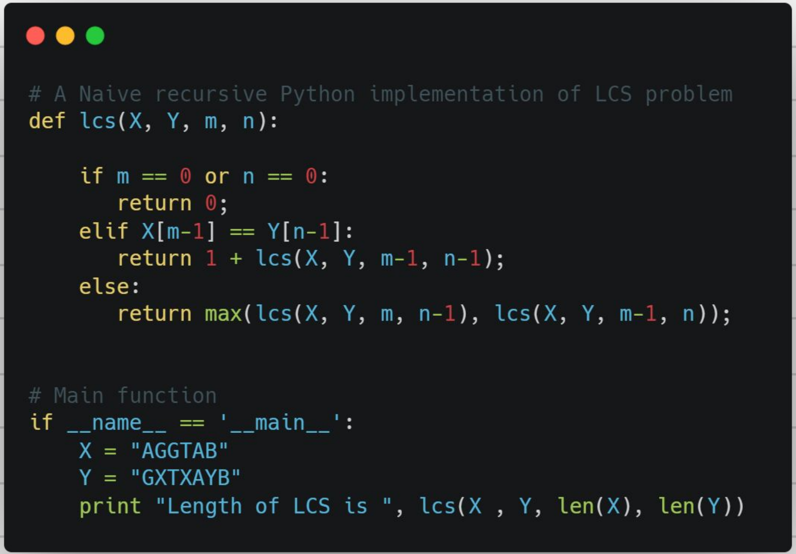
So the LCS problem has optimal substructure property as the main problem can be solved using solutions to subproblems.

	A	G	G	T	A	B
G	-	-	4	-	-	-
X	-	-	-	-	-	-
T	-	-	-	3	-	-
X	-	-	-	-	-	-
A	-	-	-	-	2	-
Y	-	-	-	-	-	-
B	-	-	-	-	-	1

Dynamic Programming

2. **Overlapping Subproblems:**

   Following is simple recursive implementation of the LCS problem. The implementation simply follows the recursive structure mentioned above.



```
# A Naive recursive Python implementation of LCS problem
def lcs(X, Y, m, n):

    if m == 0 or n == 0:
       return 0;
    elif X[m-1] == Y[n-1]:
       return 1 + lcs(X, Y, m-1, n-1);
    else:
       return max(lcs(X, Y, m, n-1), lcs(X, Y, m-1, n));


# Main function
if __name__ == '__main__':
    X = "AGGTAB"
    Y = "GXTXAYB"
    print "Length of LCS is ", lcs(X , Y, len(X), len(Y))
```

Dynamic Programming

Time complexity of the above naive recursive approach is O(2^n) in worst case and worst case happens when all characters of X and Y mismatch i.e., length of LCS is 0.

Considering the above implementation, following is a partial recursion tree for input strings “AXYT” and “AYZX”.

```
                         lcs("AXYT", "AYZX")
                       /
         lcs("AXY", "AYZX")            lcs("AXYT", "AYZ")
         /                              /
lcs("AX", "AYZX") lcs("AXY", "AYZ")   lcs("AXY", "AYZ") lcs("AXYT", "AY")
```

Dynamic Programming

In the tree, lcs("AXY", "AYZ") is being solved twice. If we draw the complete recursion tree, then we can see that there are many subproblems which are solved again and again.

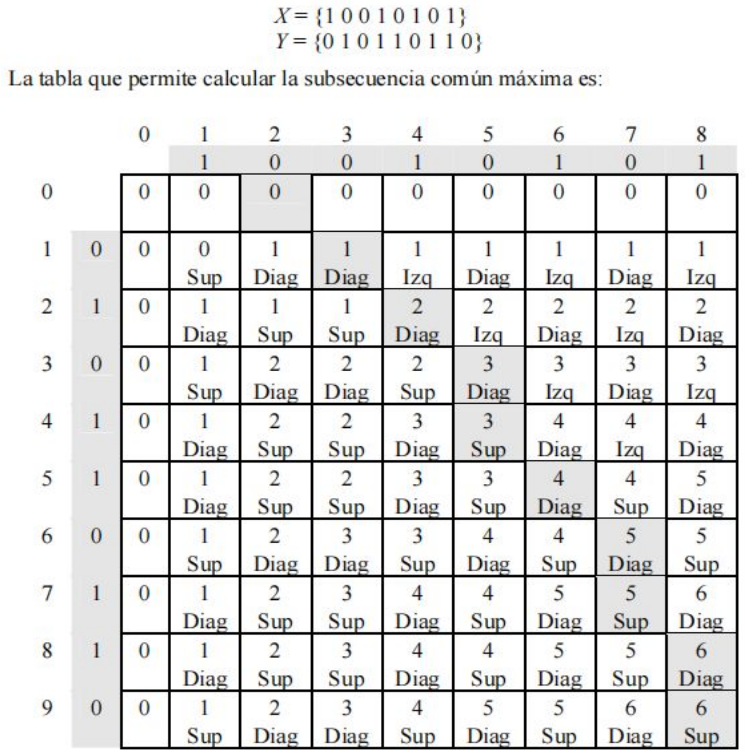
So this problem has Overlapping Substructure property and recomputation of same subproblems can be avoided by either using **Memoization**.

```
                         lcs("AXYT", "AYZX")
                       /
         lcs("AXY", "AYZX")            lcs("AXYT", "AYZ")
         /                              /
lcs("AX", "AYZX") lcs("AXY", "AYZ")   lcs("AXY", "AYZ") lcs("AXYT", "AY")
```

Dynamic Programming

The classic dynamic programming solution to LCS problem was invented by **Wagner and Fischer**. Using dynamic programming, the values in this matrix can be computed in O(mn) time and space.

$$L(i,j) = \begin{cases} 0 & \text{si } i = 0 \text{ o } j = 0 \\ L(i-1, j-1) + 1 & \text{si } i \neq 0,\ j \neq 0 \text{ y } x_i = y_j \\ Max\{L(i, j-1), L(i-1, j)\} & \text{si } i \neq 0,\ j \neq 0 \text{ y } x_i \neq y_j \end{cases}$$

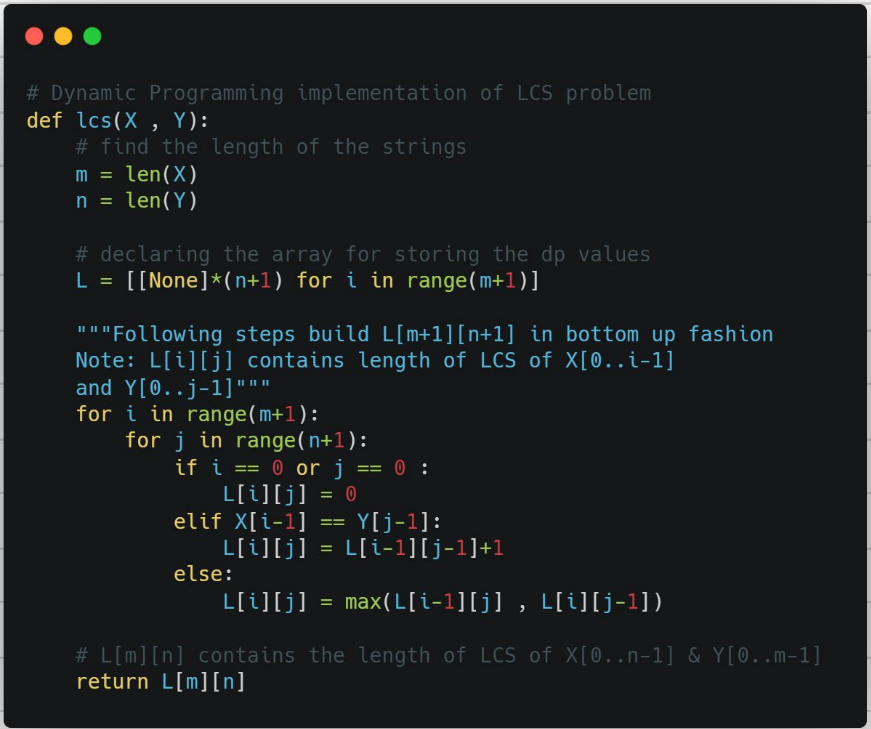


$X = \{1\,0\,0\,1\,0\,1\,0\,1\}$
$Y = \{0\,1\,0\,1\,1\,0\,1\,1\,0\}$

La tabla que permite calcular la subsecuencia común máxima es:

		0	1	2	3	4	5	6	7	8
			1	0	0	1	0	1	0	1
0		0	0	0	0	0	0	0	0	0
1	0	0	0 Sup	1 Diag	1 Diag	1 Izq	1 Diag	1 Izq	1 Diag	1 Izq
2	1	0	1 Diag	1 Sup	1 Sup	2 Diag	2 Izq	2 Diag	2 Izq	2 Diag
3	0	0	1 Sup	2 Diag	2 Diag	2 Sup	3 Diag	3 Izq	3 Diag	3 Izq
4	1	0	1 Diag	2 Sup	2 Sup	3 Diag	3 Sup	4 Diag	4 Izq	4 Diag
5	1	0	1 Diag	2 Sup	2 Sup	3 Diag	3 Sup	4 Diag	4 Sup	5 Diag
6	0	0	1 Sup	2 Diag	3 Diag	3 Sup	4 Diag	4 Sup	5 Diag	5 Sup
7	1	0	1 Diag	2 Sup	3 Sup	4 Diag	4 Sup	5 Diag	5 Sup	6 Diag
8	1	0	1 Diag	2 Sup	3 Sup	4 Diag	4 Sup	5 Diag	5 Sup	6 Diag
9	0	0	1 Sup	2 Diag	3 Diag	4 Sup	5 Diag	5 Sup	6 Diag	6 Sup

Dynamic Programming



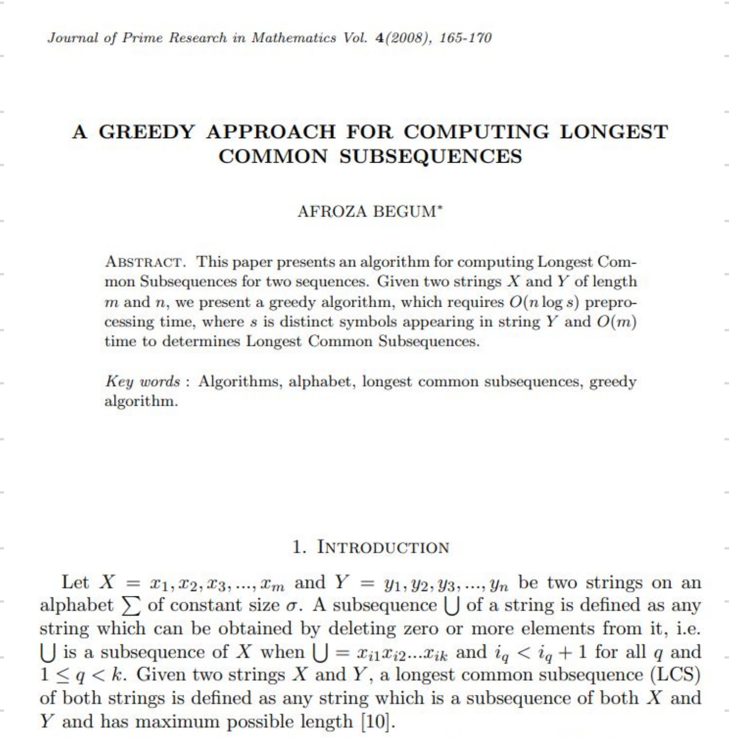
```python
# Dynamic Programming implementation of LCS problem
def lcs(X , Y):
    # find the length of the strings
    m = len(X)
    n = len(Y)

    # declaring the array for storing the dp values
    L = [[None]*(n+1) for i in range(m+1)]

    """Following steps build L[m+1][n+1] in bottom up fashion
    Note: L[i][j] contains length of LCS of X[0..i-1]
    and Y[0..j-1]"""
    for i in range(m+1):
        for j in range(n+1):
            if i == 0 or j == 0 :
                L[i][j] = 0
            elif X[i-1] == Y[j-1]:
                L[i][j] = L[i-1][j-1]+1
            else:
                L[i][j] = max(L[i-1][j] , L[i][j-1])

    # L[m][n] contains the length of LCS of X[0..n-1] & Y[0..m-1]
    return L[m][n]
```

Greedy



Journal of Prime Research in Mathematics Vol. **4***(2008), 165-170*

A GREEDY APPROACH FOR COMPUTING LONGEST COMMON SUBSEQUENCES

AFROZA BEGUM*

ABSTRACT. This paper presents an algorithm for computing Longest Common Subsequences for two sequences. Given two strings X and Y of length m and n, we present a greedy algorithm, which requires $O(n \log s)$ preprocessing time, where s is distinct symbols appearing in string Y and $O(m)$ time to determines Longest Common Subsequences.

Key words : Algorithms, alphabet, longest common subsequences, greedy algorithm.

1. INTRODUCTION

Let $X = x_1, x_2, x_3, ..., x_m$ and $Y = y_1, y_2, y_3, ..., y_n$ be two strings on an alphabet $\sum$ of constant size σ. A subsequence $\bigcup$ of a string is defined as any string which can be obtained by deleting zero or more elements from it, i.e. $\bigcup$ is a subsequence of X when $\bigcup = x_{i1}x_{i2}...x_{ik}$ and $i_q < i_q + 1$ for all q and $1 \leq q < k$. Given two strings X and Y, a longest common subsequence (LCS) of both strings is defined as any string which is a subsequence of both X and Y and has maximum possible length [10].

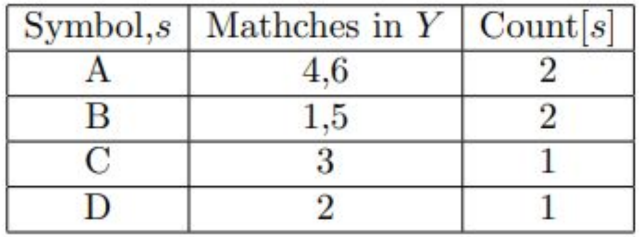
Greedy

- **Preprocessing**

First the lists of coincident points or matches for each distinct symbol in **Y** are computed. Lists of matches are the lists of ordered pairs of integers **(i, j)** such that $\mathbf{x_i = y_j}$. It is sufficient to record only the set of **j** values (the positions in **Y**) corresponding to each distinct symbol, since from this set, the set of **i** values (the positions in **X**) can easily be obtained.

For example, let two strings X = ABCBDABE and Y = BDCABA . The lists of matches for string Y are shown in Table 1.

TABLE 1. Lists of matches for string Y



Symbol,s	Mathches in Y	Count$[s]$
A	4,6	2
B	1,5	2
C	3	1
D	2	1

Greedy

- ### Algorithm

Only the matched symbol can constitute a LCS.

So, at the time of scanning the string **X**, when we want to decide whether the symbol being examined is to select next, we only look at the lists of matches for that symbol. We can only consider the symbols for which lists of matches are constructed. All other symbols can be disregarded.

<u>This is greedy because it leaves as much opportunity as possible for the remaining symbols to be selected</u>.

Greedy

- **Algorithm**

The proposed greedy algorithm uses the following steps to compute LCS:

1. Preprocessing phase: Constructs lists of matches for all distinct symbols in **Y** .
2. Scan **X** from left to right. For those symbols of **X** that have match lists do the following:
    a. Let P_i **and** P_{i+1} be the two positions of **i** and **i + 1** symbol respectively that are obtained from lists of matches.

    b. Compare them. If P_i is larger than P_{i+1} then we disregard P_i and P_{i+1} is added to the set **L**. **L** is the set of positions of selected symbol that will constitute LCS, which is initially empty.

    c. If P_i is smaller than P_{i+1} then P_i is added to the set **L**.

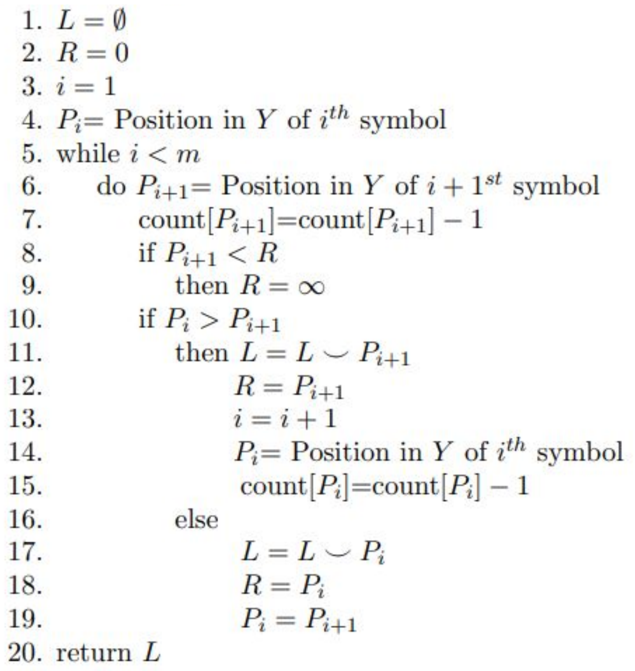
This algorithm records the position of last selected symbol in **Y**. If any of P_i and P_{i+1} denotes a position that belongs to the left of last selected position, it is assigned ∞ to ignore that position. Also, match lists traces the next matched positions of each symbol.

Greedy

- **Algorithm**

Lists of matches of all distinct symbols in **Y** are provided by preprocessing phase. A one dimensional array **count[s]** maintains the total number of coincident points for symbol **s**. **R** records the position of last selected symbol.

L is the set of positions of selected symbol that will constitute LCS, which is initially empty.

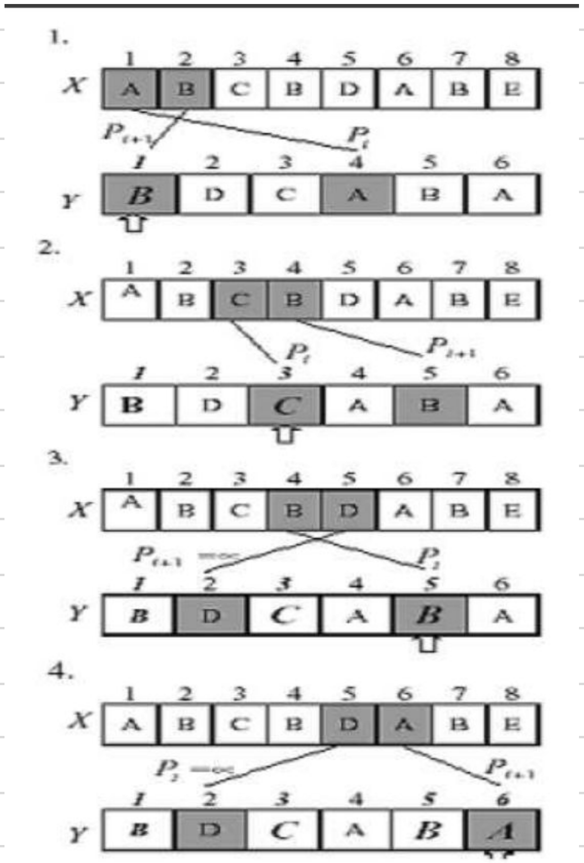


```
1.  $L = \emptyset$
2.  $R = 0$
3.  $i = 1$
4.  $P_i$= Position in $Y$ of $i^{th}$ symbol
5.  while $i < m$
6.      do $P_{i+1}$= Position in $Y$ of $i + 1^{st}$ symbol
7.          count[$P_{i+1}$]=count[$P_{i+1}$] $- 1$
8.          if $P_{i+1} < R$
9.              then $R = \infty$
10.         if $P_i > P_{i+1}$
11.             then $L = L \smile P_{i+1}$
12.                 $R = P_{i+1}$
13.                 $i = i + 1$
14.                 $P_i$= Position in $Y$ of $i^{th}$ symbol
15.                 count[$P_i$]=count[$P_i$] $- 1$
16.             else
17.                 $L = L \smile P_i$
18.                 $R = P_i$
19.                 $P_i = P_{i+1}$
20. return $L$
```

Greedy

- **Result**

The execution greedy on two given strings **X** and **Y**. Lightly shaded elements denote the symbols being examined. In each iteration, hollow arrows indicate the last selected position. The resulting set of selected positions is {1, 3, 5, 6}.



Greedy

- **Conclusion**

The paper proposes a greedy algorithm for the computation of the Longest Common Subsequences of two strings **X** and **Y** that achieves complexity of **O(m)** time with **O(n log s)** preprocessing time, where **m** and **n** are the lengths of two original strings and **s** denotes the total number of distinct symbols in string **Y** .