Longest Common Subsequence

Tanveer Muttaqueen
Student ID : 1505002

Subangkar Karmaker Shanto
Student ID : 1505015

August 3, 2018

Contents

Chapter 1

Introduction

This report briefly illustrates the well known problem *Longest Common Subsequence*. Longest Common Subsequence, in short LCS is the problem of finding the common subsequence of some sequences that has the longest length among all other subsequences of them.

Basically in LCS, we find longest common subsequence of two string. We can also find longest common subsequence of more than two strings but that is much more complicated. So here we will limit out discussion to finding longest common subsequence of two string.

So in this problem we will have two string as input and our output will be a single string that is the longest common subsequence of the above two.

Chapter 2

Subsequence

2.1 Subsequence

2.1.1 Definition:

Given two sequences $X = \langle\ x_1, x_2,...,x_m\ \rangle$ and $Z = \langle\ z_1, z_2,...,z_k\ \rangle$, we say that Z is a subsequence of X if there is a strictly increasing sequence of k indices i_1, i_2,...,i_k $(1 \leq i_1 < i_2 < ... < i_k \leq m)$ such that $Z = \langle\ x_{i_1}, x_{i_2},..., x_{i_k}\ \rangle$.

Informally sequential elements in subsequence must have strictly increasing sequence in the original sequence.

2.1.2 Example of Subsequences:

Let's consider a string $X = \langle$BEGINNING$\rangle$

Main Sequence	$\langle$BEGINNING$\rangle$
Subsequences	$\langle$BGN$\rangle$, $\langle$INNING$\rangle$, $\langle$BIS$\rangle$ etc.
Not Subsequence	$\langle$EBG$\rangle$, $\langle$NINNIG$\rangle$, $\langle$BGL$\rangle$ etc.

Let's consider another string $Y = \langle$BACDB$\rangle$

Main Sequence	$\langle$BACDB$\rangle$
Subsequences	$\langle$BDB$\rangle$, $\langle$CDB$\rangle$, $\langle$BCB$\rangle$ etc.
Not Subsequence	$\langle$DCA$\rangle$, $\langle$BDA$\rangle$, $\langle$CDA$\rangle$ etc.

2.2 Longest Common Subsequence

A longest common subsequence is any string that is of the longest length among any common subsequences. Obviously there can be more than on subsequence with the longest length. In that case any of those subsequences is a optimal solution by definition. Here are some examples,

1. **Original Sequences:** $S = \langle \text{BDCB} \rangle$, $T = \langle \text{BACDB} \rangle$
   - **Common Subsequences:** $\langle \text{B} \rangle$, $\langle \text{BD} \rangle$, $\langle \text{BCB} \rangle$ etc.
   - **Longest Common Subsequence:** $\langle \text{BCB} \rangle$

2. **Original Sequences:** $S = \langle \text{DABKC} \rangle$, $T = \langle \text{APBCK} \rangle$
   - **Common Subsequences:** $\langle \text{A} \rangle$, $\langle \text{AB} \rangle$, $\langle \text{ABK} \rangle$, $\langle \text{ABC} \rangle$ etc.
   - **Longest Common Subsequence:** $\langle \text{ABK} \rangle$, $\langle \text{ABC} \rangle$

3. **Original Sequences:** $S = \langle \text{ABCD} \rangle$, $T = \langle \text{PQRS} \rangle$
   - **Common Subsequences:** NO common subsequences
   - **Longest Common Subsequence:** NO longest common subsequences

Chapter 3

Problem Solving Techniques

There are many well established methodology and algorithms for problem solving. Brute force technique, Divide and conquer, Dynamic Programming are some popular examples of such methodologies. Here we will discuss Brute force technique and Dynamic Programming approach as both of them are relevant to our problem.

3.1 Brute Force Technique

Brute force is a type of algorithm that tries a large number of patterns to solve a problem. It is often mentioned as *Brute Force Search* or *Exhaustive Search*. Brute Force Search is a very general problem-solving technique that consists of systematically enumerating all possible candidates for the solution and checking whether each candidate satisfies the problem's statement.
For example, a brute-force algorithm to find the divisors of a natural number n would enumerate all integers from 1 to n, and check whether each of them divides n without remainder. A brute-force approach for the eight queens puzzle would examine all possible arrangements of 8 pieces on the 64-square chessboard, and, for each arrangement, check whether each (queen) piece can attack any other.

While a brute-force search is simple to implement, and will always find a solution if it exists, its cost is proportional to the number of candidate solutions – which in many practical problems tends to grow very quickly as the size of the problem increases. Therefore, brute-force search is typically used when the problem size is limited, or when there are problem-specific heuristics that can be used to reduce the set of candidate solutions to a manageable size. The method is also used when the simplicity of implementation is more important than speed.

Basic algorithm

In order to apply brute-force search to a specific class of problems, one must implement four procedures, first,next, valid, and output. These procedures should take as a parameter the data P for the particular instance of the problem that is to be solved, and should do the following:

1. first (P): generate a first candidate solution for P.
2. next (P, c): generate the next candidate for P after the current one c.
3. valid (P, c): check whether candidate c is a solution for P.
4. output (P, c): use the solution c of P as appropriate to the application.

The next procedure must also tell when there are no more candidates for the instance P, after the current one c. A convenient way to do that is to return a "null candidate", some conventional data value $\wedge$ that is distinct from any real candidate. Likewise the first procedure should return $\wedge$ if there are no candidates at all for the instance P. The brute-force method is then expressed by the algorithm.

```
1 c ← first(P)
2 while c ≠ ∧ do
3     if valid(P, c) then
4         output(P, c)
5     c ← next(P, c)
```

3.2 Dynamic Programming Approach

Dynamic programming is both a mathematical optimization method and a computer programming method. The method was developed by Richard Bellman in the 1950s. In both contexts it refers to simplifying a complicated problem by breaking it down into simpler sub-problems in a recursive manner. While some decision problems cannot be taken apart this way, decisions that span several points in time do often break apart recursively. Likewise, in computer science, if a problem can be solved optimally by breaking it into sub-problems and then recursively finding the optimal solutions to the sub-problems, then it is said to have optimal substructure.

3.2.1 Applicability for DP

There are two key attributes that a problem must have in order for dynamic programming to be applicable. Those are

- Optimal Substructure
- Overlapping Sub-Problems

If a problem can be solved by combining optimal solutions to non-overlapping sub-problems, the strategy is called "divide and conquer" instead of dynamic programming. This is why merge sort and quick sort are not classified as dynamic programming problems.

3.2.2 Optimal Substructure

Optimal substructure means that the solution to a given optimization problem can be obtained by the combination of optimal solutions to its sub-problems. Such optimal substructures are usually described by means of recursion.
For example, given a graph G=(V,E), the shortest path p from a vertex u to a vertex v exhibits optimal substructure: take any intermediate vertex w on this shortest path p. If p is truly the shortest path, then it can be split into subpaths p1 from u to w and p2 from w to v such that these, in turn, are indeed the shortest paths between the corresponding vertices (by the simple cut-and-paste argument). Hence, one can easily formulate the solution for finding shortest paths in a recursive manner, which is what the Bellman–Ford algorithm or the Floyd–Warshall algorithm does.

3.2.3 Overlapping Sub-Problems

Overlapping sub-problems means that the space of sub-problems must be small, that is, any recursive algorithm solving the problem should solve the same sub-problems over and over, rather than generating new sub-problems.
For example, consider the recursive formulation for generating the Fibonacci series: $F_i = F_{i-1} + F_{i-2}$, with base case $F_1 = F_2 = 1$. Then $F_{43} = F_{42} + F_{41}$, and $F_{42} = F_{41} + F_{40}$. Now F_{41} is being solved in the recursive sub-trees of both F_{43} as well as F_{42}. Even though the total number of sub-problems is actually small (only 43 of them), we end up solving the same problems over and over if we adopt a naive recursive solution such as this. Dynamic programming takes account of this fact and solves each sub-problem only once. This can be achieved in either of two ways:

- **Top-down approach:** This is the direct fall-out of the recursive formulation of any problem. If the solution to any problem can be formulated recursively using the solution to its sub-problems, and if its sub-problems are overlapping, then one can easily memoize or store the solutions to the sub-problems in a table. Whenever we attempt to solve a new sub-problem, we first check the table to see if it is already solved. If a solution has been recorded, we can use it directly, otherwise we solve the sub-problem and add its solution to the table.

- **Bottom-up approach:** Once we formulate the solution to a problem recursively as in terms of its sub-problems, we can try reformulating the problem in a bottom-up fashion: try solving the sub-problems first and use their solutions to build-on and arrive at solutions to bigger sub-problems. This is also usually done in a tabular form by iteratively generating solutions to bigger and bigger sub-problems by using the solutions to small sub-problems. For example, if we already know the values of F_{41} and F_{40}, we can directly calculate the value of F_{42}.

Chapter 4

Solving LCS: Brute Force Technique

4.1 Solution

Suppose we are to find LCS of string X and Y. Then we can enumerate all subsequences of X, and check whether they are subsequences of Y and then take the longest one among them.

4.2 Time Complexity

Suppose X has length n and Y has length m.

- Checking = $\mathcal{O}(n)$ time per subsequence.
- 2^m subsequences of Y (each bit-vector of length m determines a distinct subsequence of Y).

Worst-case running time = $\mathcal{O}(n2^m)$ = **exponential time**.

4.3 Remarks

Although we can go with this algorithm for smaller values of m and n, we will not be able to solve this problem for strings having larger length i.e 100. So we need faster algorithms. Hence dynamic programming is in preference.

Chapter 5

Solving LCS: Dynamic Programming Approach

We already know the properties that are required for a problem to be eligible to be solved with dynamic programming . LCS exhibits both Optimal sub-structure and Overlapping sub-problems property. Hence we can use dynamic programming to solve it. Now we will discuss the method of solving LCS with dynamic programming in detail.

5.1 Stucture of LCS problem

The structure of LCS can be represented as below: Suppose $X = \langle x_1 x_2 ..x_m \rangle$, $Y = \langle y_1 y_2 ...y_n \rangle$ and $Z = \langle z_1 z_2 ...z_k \rangle$ is their longest commmon subsequence. Now we have:

1. If $x_m = y_n$ then $z_k = x_m = y_n$. So Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_n \neq y_m$ and $z_k \neq x_m$ then Z_k is an LCS of X_{m-1} and Y_n.
3. If $x_n \neq y_m$ and $z_k \neq y_n$ then Z_k is an LCS of X_m and Y_{n-1}.

Note: $X_{m-1} = \langle x_1 x_2 ..x_{m-1} \rangle$, $Y_{n-1} = \langle y_1 y_2 ...y_{n-1} \rangle$ and $Z_{k-1} = \langle z_1 z_2 ...z_{k-1} \rangle$.

5.2 Recursive Structure of the Subproblem

From the optimal substructure stated in the previous subsection, we know that, to find the LCS of X and Y, we just need to go through the following procedure:

- if $x_n = y_m$, find the LCS of X_{m-1} and Y_{n-1} and then append x_m (or y_n).
- if $x_n \neq y_m$, find the LCS of X_{m-1} and Y_n, and the LCS of X_m and Y_{n-1}, then pick the longer one.

We can see a overlapping subproblems attribute from this recursive structure. For example, when finding the LCS of X and Y, we may need to find the LCS of X and Y_{n-1} and the LCS of X_{m-1} and Y first; and these two both depend on one subproblem, to find the LCS of X_{m-1} and Y_{n-1}.

So let us build the recursive relations among the optimal values of the subproblems. Denote c[i,j] as the length of X_i and Y_j. When i=0 or j=0, the LCS of X_i and Y_j is an empty sequence, so c[i,j]=0, otherwise the recursive relation can be defined as:

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1, j-1] & \text{if } i, j > 0 \text{ and } x_i = y_j \\ max(c[i-1, j], c[i, j-1]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

5.3 Computing length of the LCS

Using the recursive formula defined above, we can easily contrive an algorithm to compute c[i,j], but the execution time will grow exponentially with the length of input. Since there are only $\Theta(m * n)$ subproblems in the subproblem space, we can use the bottom-up approach to improve efficiency.

The LCS_LENGTH(X,Y) algorithm takes $X = \langle x_1 x_2 ..x_m \rangle$ and $Y = \langle y_1 y_2 ...y_n \rangle$ as inputs and then outputs two matrices c[0..m, 0..n] and b[1..m, 1..n]. c[i,j] stores the length of LCS(X_i, Y_j) and b[i,j] stores where c[i,j] gets its value from (this will be explained later). At the end of the algorithm, the length of LCS(X,Y) will be stored at c[m,n].

```
LCS_LENGTH(X, Y)
 1  m := length[X]
 2  n := length[Y]
 3  for i := 1 to m do
 4      c[i, 0] := 0
 5  for j := 1 to n do
 6      c[0, j] := 0
 7  for i := 1 to m do
 8      for j := 1 to n do
 9          if x_i = y_j then
10              c[i, j] := c[i − 1, j − 1] + 1
11              b[i, j] :=′↖′
12          else if c[i − 1, j] ≥ c[i, j − 1] then
13              c[i, j] := c[i − 1, j]
14              b[i, j] :=′↑′
15          else
16              c[i, j] := c[i, j − 1]
17              b[i, j] :=′←′
18  return (c,b)
```

	j	0	1	2	3	4
i		y	D	B	P	T
0	x	0	0	0	0	0
1	A	0	0	0	0	0
2	D	0	1	1	1	1
3	A	0	1	1	1	1
4	P	0	1	1	2	2
5	T	0	1	1	2	3

Table 5.1: table c

	j	0	1	2	3	4
i		y	D	B	P	T
0	x	0	0	0	0	0
1	A	0	↑	↑	↑	↑
2	D	0	↖	←	←	←
3	A	0	↑	↑	↑	↑
4	P	0	↑	↑	↖	←
5	T	0	↑	↑	↑	↖

Table 5.2: table b

Table 5.3: tables for strings X=⟨ADAPT⟩ and Y=⟨DBPT⟩

5.4 Construct the LCS

With the help of matirx b from LCS_LENGTH, we can construct the LCS of X and Y. Starting from b[m,n], we can navigate the matrix according to the direction of each 'arrow':

- when b[i,j] = $\nwarrow$, *it means* $x_i = y_i$ is an element of LCS(X_i, Y_j) i.e. LCS(X_i, Y_j) is LCS(X_{i-1},Y_{j-1}) appends x_i (or y_j)
- when b[i,j] = $\uparrow$, *it means* $LCS(X_i, Y_j)$ is the same as LCS(X_{i-1}, Y_j)
- when b[i,j] = $\leftarrow$, *it means* $LCS(X_i, Y_j)$ is the same as LCS(X_i,Y_{j-1})

```
PRINT-LCS(b, X, i, j)
1 if i = 0 or j = 0 then
2     return
3 if b[i, j] = "↖" then
4     PRINT-LCS(b, X, i − 1, j − 1)
5     print x_i
6 else if b[i, j] = "↑" then
7     PRINT-LCS(b, X, i − 1, j)
8 else
9     PRINT-LCS(b, X, i, j − 1)
```

E.g., for two sequences X=⟨ADAPT⟩ and Y=⟨DBPT⟩, the results of LCS_LENGTH() and PRINT-LCS() can be shown as:

	j	0	1	2	3	4
i		y	D	B	P	T
0	x	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↑ 0
2	D	0	↖ 1	← 1	← 1	← 1
3	A	0	↑ 1	↑ 1	↑ 1	↑ 1
4	P	0	↑ 1	↑ 1	↖ 2	← 2
5	T	0	↑ 1	↑ 1	↑ 2	↖ 3

Lets explain this diagram. Firstly, LCS_LENGTH() computes matrix c and matrix b from X and Y, and the cell at the ith row and jth column stores the value of c[i,j] and the arrow pointing to next entry of b. At c[5,4], number 3 stands for the length of LCS$\langle DPT \rangle$. To re-construct the LCS, we just need to follow the arrow from the lower right corner. Each $\nwarrow$ on the path denotes an element of the LCS.

So, according to the diagram, the procedure will finally print out: "DPT"

5.5 Complexity

- Each entity of the table can be computed in $\mathcal{O}(1)$ time
- There are $|X| * |Y|$ entities to be filled. So total time complexity is O(|X|·|Y|).
- Similarly total memory complexity is also O(|X|·|Y|)

Chapter 6

Improvement

Unfortunately, unless something is said explicitly about input alphabet, no known optimization on time complexity is possible.

But if the sample space is known then following optimizations can be applied,

- For problems with a bounded alphabet size, the Method of Four Russians can be used to reduce the running time of the dynamic programming algorithm by a logarithmic factor.
- For problems where S and T has no repeated alphabet, Longest common subsequence problem can be reduced to Longest increasing subsequence. Thus solving the problem in $O(n \cdot logn)$.

But memory optimization is certainly possible. In each state we only need last two rows. So we can keep last two rows using even, odd sequence. Thus giving memory complexity O(2·|Y|)

Chapter 7

Conclusion

Longest Common Subsequence or LCS is a illustration of power of dynamic programming in problem solving. Without using DP we would have a exponential time complexity that is impossible to solve with existing computational power.

However, this well known LCS problem is the basis of data comparison programs such as the diff utility, and has applications in bioinformatics. It is also widely used by revision control systems such as Git for reconciling multiple changes made to a revision-controlled collection of files. Even in medical science LCS is an important problem. Fast DNA Sequence Clustering can be done based on LCS.

Again more complicated problems have been made easier with idea of LCS problem such that finding LCS of K-sequences, Longest Increasing Subsequence, Finding if string is K-palindrome etc.