08 Feb 2021

Longest Common Subsequence

Longest common subsequence (LCS) problem is to find the longest subsequence common in all sequences. LCS problem comes in different fields of computer science. Brute force approach has exponential time complexity, and Dynamic programming approach solve it in polynomial time complexity. In this document, I explained Dynamic programming approach to solve this problem. To know more about use case of LCS and what the Dynamic programming is, please refer to Wikipedia.

VIKAS AWADHIYA

LinkedIn profile: https://www.linkedin.com/in/awadhiya-vikas

Introduction

To understand what longest common subsequence (LCS) is? Let's see how LCS is different from longest common substring. As name implies longest common substring have matching characters at continuous positions, but in LCS matching characters don't have to be in continuous positions but only requires to be in same order in all sequences, let's see it by these two sequences as an example,

abdcefg

cefadchfg

So longest common substring is "**cef**", as highlighted in fig1 below,

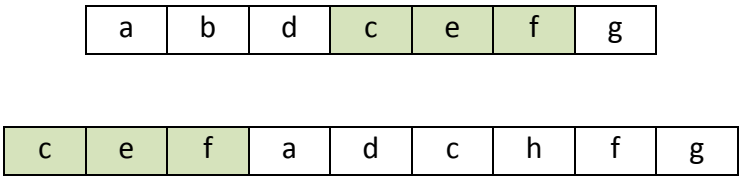


Fig 1

But LCS is "**adcfg**"**,** as highlighted in fig2 below,

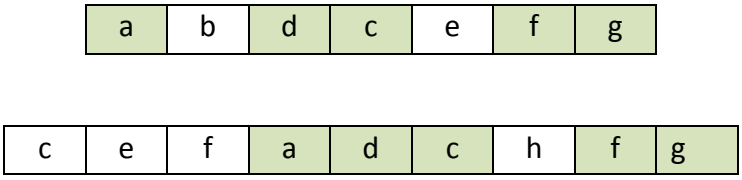


Fig 2

Above example explained what LCS is, now next step is to understand, brute force approach and dynamic programming approach to solve LCS problem.

Brute force approach

Brute force approach cannot be used directly because it has exponential time complexity, but knowledge of, how brute force approach works, is required to understand dynamic programming approach. Consider following sequences as an example,

abdabacfgih

adicabafgzh

Idea of brute force approach is, compare character by character,

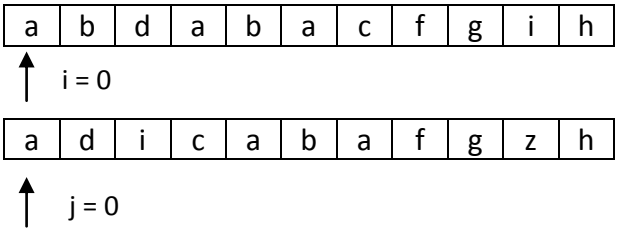


Fig 3

As showed in fig3 above, character by character comparison begins from index zero of both sequences, here **i** and **j** represent indexes of sequence1 and sequence2 respectively,

Step 1: **i = 0**, **j = 0**, match, both index **i** and **j** increments by one, and LCS = "**a**"

Step 2: **i = 1, j = 1**, mismatch '**b**' and '**d** ' are not equal, and only index **i** increment by one, no change in LCS,

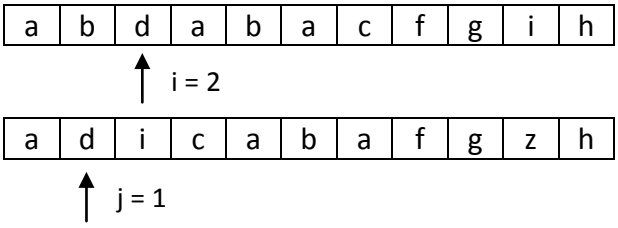


Fig 4

Step 3: **i = 2**, **j = 1**, match, both index **i** and **j** increments by one, and LCS = "**ad**", let's continue few more steps,

Step 4: **i = 3, j = 2**, mismatch '**i**' and '**a**' are not equal, only index **i** increment by one, and going in this way no match occurs until value of index **i** reach to **i = 9**, see fig5 below,

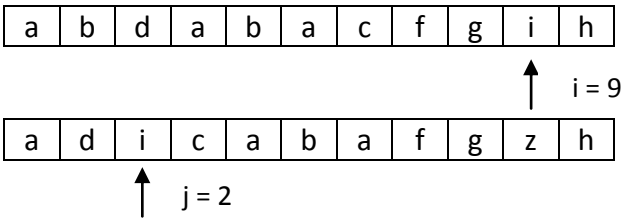


Fig 5

Step 5: **i = 9, j = 2**, match, both index **i** and **j** increment by one, and LCS = "**adi**"

Step 6: **i = 10, j = 3**, mismatch, current position of index **i** is on last character in sequence1. Character at index **j = 3** cannot be found because no chars left for comparison. Only index **j** increments by one,

Step 7 to 12: For index **j = 4, 5, 6, 7, 8, 9**, skipping because there is no match,

Step 13, **i = 10, j = 10**, match, both index **i** and **j** increment by one and LCS = "**adih**". After this comparison all characters in sequence2, are compared, and comparison is completed.

LCS = "**adih**", for comparison starts from index **j = 0**,

But "**adih**" may not be the answer, it is one of the possible subsequence, but there may be some other longer subsequence, which may start from non zero index and may not have characters from continuous positions, for example after step 3 if character at index 2 was not considered and index **j** would have incremented to **j = 3**, then LCS would have been to "**adcfgh**" because char '**c**' would have matched at index **i = 6**, and this subsequence is longer than "**adih**", see in fig 6 below,

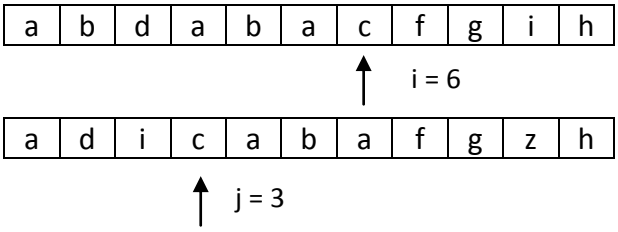


Fig 6

Steps of brute force approach indicate, until all possibilities, in terms of all indexes and combinations of characters are not evaluated, LCS cannot be evaluated. Covering all indexes and characters combination manually is not a correct way. Another fact is, these sequences are purposely kept small for explaining purpose but sequences could be of large size and a solution of exponential complexity is not a practical approach for long sequences of real world problems.

Dynamic programming approach

Brute force approach is an easy way to solve LCS but it is not an efficient approach. Dynamic programming can reduce time complexity from exponential to polynomial. One of the core ideology of dynamic programming is, results evaluated in previous steps should be used in next step and should not be evaluated again. Avoiding reevaluation of result brings time complexity down.

Brute force approach is going to be used again, but this time with small modification and using dynamic programming to avoiding reevaluation of results evaluated in previous steps. Consider same sequences again,

abdabacfgih

adicabafgzh

As mentioned above small modification in brute force approach, this time each char will be matched individually and matched characters shall not combine with other matched characters, it means match of every character in sequence2, will start from beginning of sequence1 (**i = 0** index) and not like what is done in previously, where next character match start, one index after the previous matched character index.

Now table will be use to show match, and empty cell means mismatch, here **i** and **j** represent index of sequence1 and sequence2 respectively. Numbers in last row represent indexes of sequence1,

Step 1: **j = 0**,

a	b	d	a	b	a	c	f	g	i	h

a			a		a					
0	1	2	3	4	5	6	7	8	9	10

Fig 7

As highlighted in fig7 above, first char of sequence2 (which is '**a**') matches with character of sequence1 at indexes **i = 0, 3, 5**,

Step 2: **j = 1**,

		d								
0	1	2	3	4	5	6	7	8	9	10

Fig 8

'**d**' matches only at index **i = 2** of sequence1 as highlighted in fig8 above,

Step 3: **j = 2, 3**, result is written together with previous two match results,

a	b	d	a	b	a	c	f	g	i	h

j												
0	a			a		a						
1			d									
2										i		
3							c					
	0	1	2	3	4	5	6	7	8	9	10	**i**

Fig 9

Table showed in fig9 above will not be used in implementation directly, but it is an observation table. It brings following observations,

Characters of sequence2 are being compared from index **j = 0, 1, 2, 3….** and if these characters also appear in same index order in sequence1 then these characters becomes subsequence,

View table show in fig9 above from **j = 0** and **j** increase from top to bottom. Let's see steps again,

Step1: **j = 0**, 'a' matches at indexes **i** = 0, 3, 5,

Step2: **j = 1**, 'd' matches at index **i = 2**,

Character '**a**' is at index **j = 0** and '**d**' at **j = 1** in sequence2, and as show by highlighting in above table character '**a**' matches at **i = 0** and character '**d**' matches at **i = 2**, it means characters '**a**' and '**d**' appears in same order in both sequences and that is exactly the requirement of common subsequence. LCS is "**ad**" after **j = 1** is matched,

Step3: **j = 2**, match at index **i = 9**, as highlighted in above table. Characters '**a**', '**d**', and '**i**' are in same order, in both sequences. First '**a**' appears then '**d**' and then '**i**'. LCS becomes "**adi**".

Step4: **j = 3**, match at index **i = 6**, as highlighted in above table. Characters '**c**' and '**i**' are not in same order in sequence1 as they are in sequence2, in sequence2, '**i**' comes before '**c**' but in

sequence1 '**i**' comes after '**c**'. These two characters are not in same order and cannot be the part of same subsequence. After this step, there would be two LCS candidates, "**adi**" and "**adc**".

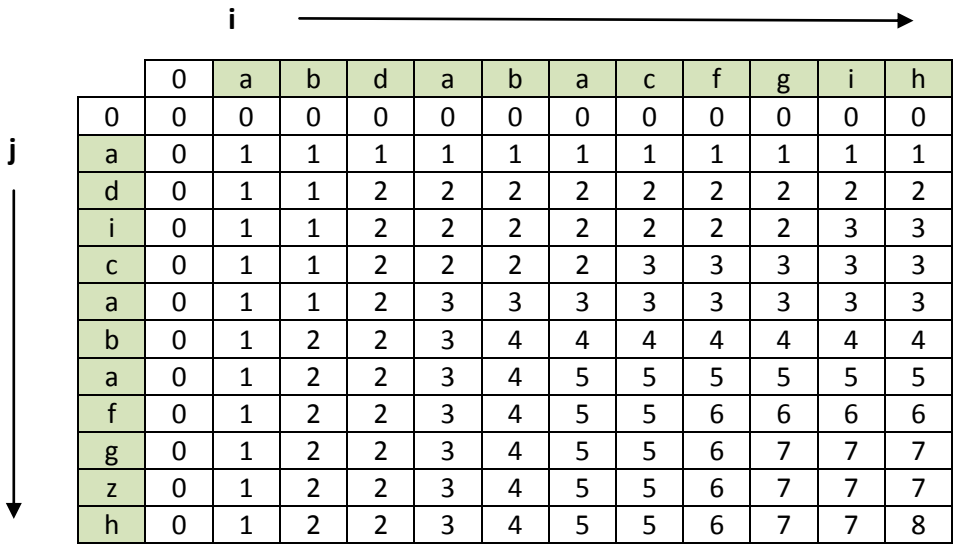
These steps can be repeated for all characters of sequence2, but is not required to show them all. The purpose of showing this table is to explain the core logic behind the dynamic programming approach. Dynamic programming approach matches each character individually and save result in table and then use table to find characters those are in same order rather than trying to cover all indexes with all possible combinations as done in brute force approach.

Table

Dynamic programming approach begins with creating table; this table is different from observation table showed above. Brute force approach and observation table were explained to make you understand, logic behind the rules of dynamic programming table creation process, without this knowledge it hard to understand table creation process.

In table both sequence are represented along row and columns and one extra row and column are use for zero entry. This table is different from observation table; in this table numbers will be written in cell to represent matched subsequence length rather matched character itself.

Following table will be created,



	0	a	b	d	a	b	a	c	f	g	i	h
0	0	0	0	0	0	0	0	0	0	0	0	0
a	0	1	1	1	1	1	1	1	1	1	1	1
d	0	1	1	2	2	2	2	2	2	2	2	2
i	0	1	1	2	2	2	2	2	2	2	3	3
c	0	1	1	2	2	2	2	3	3	3	3	3
a	0	1	1	2	3	3	3	3	3	3	3	3
b	0	1	2	2	3	4	4	4	4	4	4	4
a	0	1	2	2	3	4	5	5	5	5	5	5
f	0	1	2	2	3	4	5	5	6	6	6	6
g	0	1	2	2	3	4	5	5	6	7	7	7
z	0	1	2	2	3	4	5	5	6	7	7	7
h	0	1	2	2	3	4	5	5	6	7	7	8

Final table

Above table will be the end result of table creation process of dynamic programming approach. Table will be created through step by step character comparisons, after table creation, it will be travels in bottom-up manner to find LCS.

Table creation

Each row except first two rows represents a single character in sequence2 and each column except first two represent a single charter in sequence1. Content of row represents result of a single character comparison of sequence2 with all characters of sequence1 in respective cells.

Table will be completed once all characters of sequence2 are compared.

Step1: **j = 0**, here **j** represents index of sequence2,

	0	a	b	d	a	b	a	c	f	g	i	h
0	0	0	0	0	0	0	0	0	0	0	0	0
a	0											

Table 1

	0	a
0	0	0
a	0	+1

Table 2

	0	a	b	d	a	b	a	c	f	g	i	h
0	0	0	0	0	0	0	0	0	0	0	0	0
a	0	1										

Table 3

As showed above in table1 purple color boundary cell represents match, on match, rule1 applies,

Rule1: When match occurs, cell is updated with value of cell one above and one left to current cell plus one, it means,

Table[r, c] = Table[r – 1, c – 1] + 1, hear r and c represent row and column indexes respectively.

Table 2 shows rule1 in action with light blue boundary cell, and the result of applying rule1 is showed with orange boundary cell in table 3 above.

Logic behind the rule1 will be discussed after few steps but until that, just accept it and see what happen when match moves ahead, and '**a**' will be compared with '**b**' of sequence1,

	0	a	b	d	a	b	a	c	f	g	i	h
0	0	0	0	0	0	0	0	0	0	0	0	0
a	0	1										

Table 4

	0	a	b
0	0	0	0
a	0	1	

Table 5

	0	a	b	d	a	b	a	c	f	g	i	h
0	0	0	0	0	0	0	0	0	0	0	0	0
a	0	1	1									

Table 6

Character ‘a’ compared with ‘b’, mismatch, showed above in table 4 by red boundary cell. When mismatch occurs rule2 applies,

Rule2: When mismatch occurs, cell is update with max value between left cell and top cell, it means,

Table[r, c] = max (Table[r, c - 1], Table[r – 1, c])

Table 5 above shows rule2 in action where max value between light blue boundary cells will be considered, and result of rule2 is showed with orange boundary cell in table 6.

Again accept rule2 for now and will discuss its reasoning in coming steps.

As showed for two compares, same can be done for all remain comparison of ‘**a**’ and the result would be as showed in table 7 below,

	0	a	b	d	a	b	a	c	f	g	i	h
0	0	0	0	0	0	0	0	0	0	0	0	0
a	0	1	1	1	1	1	1	1	1	1	1	1

Table7

Step2: **j = 1**, '**d**' would be compared first with '**a**' and then with '**b**', and both would be mismatch, due to mismatch, rule2 will be applied as explained above. After that when '**d**' would be compared with '**d**' and match will occurs as show in purple boundary cell in table 8 below,

	0	a	b	d	a	b	a	c	f	g	i	h
0	0	0	0	0	0	0	0	0	0	0	0	0
a	0	1	1	1	1	1	1	1	1	1	1	1
d	0	1	1									

Table 8

	0	a	b	d
0	0	0	0	0
a	0	1	1	1
d	0	1	1	+1

Table 9

	0	a	b	d	a	b	a	c	f	g	i	h
0	0	0	0	0	0	0	0	0	0	0	0	0
a	0	1	1	1	1	1	1	1	1	1	1	1
d	0	1	1	2								

Table 10

As discussed earlier, because of match, rule1 will be applied and showed in table 8, 9, 10.

Reasoning of Rule1: Reasoning is related to the observation table discussion earlier, if you remember step2 of observation table, where match of '**d**' occurred, and by checking observation table, we saw all match indexes of '**a**' against match index of '**d**', and found that, '**a**' and '**d**' are in same order in both sequences.

Rule1 is applying same logic but only difference is, this time information is not kept in form of match character itself but in term of count. The meaning of count here is numbers of characters are already in same order, in both sequences.

After rule1 is applied value update to 2 as show with orange cell in table 10. Cell value 2 indicates there are two characters are in same order in both sequences.

Next thing to understand is reasoning behind rule2.

Reasoning of Rule2: Rule1 update match information, but in case of mismatch, rule2 make sure that match information gathered in previous steps should propagate in table, that's why on mismatch, left and top cell is views to find if there was any previous match of current character or match of previous character respectively. It may also happen that a character only present in sequence2, in this case it will not be matched against any characters of sequence1 and all entries of that row will be the copy of previous row, for example '**z**' is only present in sequenc2, see its row entries highlighted as light blue boundary row in table 11 below,

	0	a	b	d	a	b	a	c	f	g	i	h
0	0	0	0	0	0	0	0	0	0	0	0	0
a	0	1	1	1	1	1	1	1	1	1	1	1
d	0	1	1	2	2	2	2	2	2	2	2	2
i	0	1	1	2	2	2	2	2	2	2	3	3
c	0	1	1	2	2	2	2	3	3	3	3	3
a	0	1	1	2	3	3	3	3	3	3	3	3
b	0	1	2	2	3	4	4	4	4	4	4	4
a	0	1	2	2	3	4	5	5	5	5	5	5
f	0	1	2	2	3	4	5	5	6	6	6	6
g	0	1	2	2	3	4	5	5	6	7	7	7
z	0	1	2	2	3	4	5	5	6	7	7	7
h	0	1	2	2	3	4	5	5	6	7	7	8

Table 11

Comparisons for all characters of sequence2 can be done in same way and not required to demo explicitly. After all comparisons table will be completed.

One last point remains to discuss about table is, why it has one extra row and column of all zero entries? The answer you might have understood, that both rule1 and rule2 require checking match value of previous match but for first character, there is no previous character whose match value can be checked and that's why this extra row and column provide with all zero entries, so that rule1 and rule2 can work correctly.

Table bottom-up parse

Once table is created, LCS can be found by reading/parsing table in bottom-up manner. Table is required to be read / pared to find all matches positions. Value of last cell of table is maximum possible length of LCS common in both sequences. There could more than one subsequences of maximum length are possible among sequences and it doesn't have to be unique. See table 12 below, its last cell highlighted, parse will start from last cell,

	0	a	b	d	a	b	a	c	f	g	i	h
0	0	0	0	0	0	0	0	0	0	0	0	0
a	0	1	1	1	1	1	1	1	1	1	1	1
d	0	1	1	2	2	2	2	2	2	2	2	2
i	0	1	1	2	2	2	2	2	2	2	3	3
c	0	1	1	2	2	2	2	3	3	3	3	3
a	0	1	1	2	3	3	3	3	3	3	3	3
b	0	1	2	2	3	4	4	4	4	4	4	4
a	0	1	2	2	3	4	5	5	5	5	5	5
f	0	1	2	2	3	4	5	5	6	6	6	6
g	0	1	2	2	3	4	5	5	6	7	7	7
z	0	1	2	2	3	4	5	5	6	7	7	7
h	0	1	2	2	3	4	5	5	6	7	7	8

Table 12

LCS length

As discussed, table have to parsed to find all match positions, as show above in table 12, match count is 8, but how to identify match position in table? Answer is, if you remember, rule1 update cell with value of previous match count plus one,

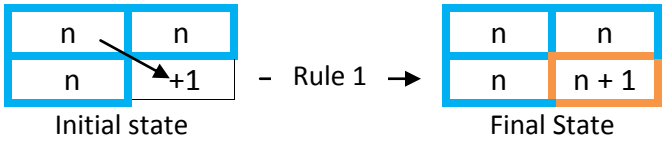


Fig 10

As show in fig 10, final state indicates match position and parse will find all match positions. Parse always begins from last cell and moves up. Yellow background cell represent path, parse follows to find match and orange cell represent match position. Following are steps,

Step 1:

	0	a	b	d	a	b	a	c	f	g	i	h
0	0	0	0	0	0	0	0	0	0	0	0	0
a	0	1	1	1	1	1	1	1	1	1	1	1
d	0	1	1	2	2	2	2	2	2	2	2	2
i	0	1	1	2	2	2	2	2	2	2	3	3
c	0	1	1	2	2	2	2	3	3	3	3	3
a	0	1	1	2	3	3	3	3	3	3	3	3
b	0	1	2	2	3	4	4	4	4	4	4	4
a	0	1	2	2	3	4	5	5	5	5	5	5
f	0	1	2	2	3	4	5	5	6	6	6	6
g	0	1	2	2	3	4	5	5	6	7	7	7
z	0	1	2	2	3	4	5	5	6	7	7	7
h	0	1	2	2	3	4	5	5	6	7	7	8

Table 13

As showed in table 13 above, last cell is match position and match character is '**h**'. Special point about the parse is, it provides subsequence in reverse order, which means character found at step 1 is last character of subsequence, LCS = "**h**".

After match found, parse have to move, but where it should move when match position found? Here observation 1 will help to move ahead,

Observation 1: A character in sequence can be included as match once, which means current row has contributed in subsequence and parse should move to row one above current row, and about the cell position, it should be one left to the match position in row one above current row. In simple words, if cell **Table[r, c]** is match position then parse should move to cell position **Table[r – 1, c – 1]**, as show in table 14 below,

Step 2:

	0	a	b	d	a	b	a	c	f	g	i	h
0	0	0	0	0	0	0	0	0	0	0	0	0
a	0	1	1	1	1	1	1	1	1	1	1	1
d	0	1	1	2	2	2	2	2	2	2	2	2
i	0	1	1	2	2	2	2	2	2	2	3	3
c	0	1	1	2	2	2	2	3	3	3	3	3
a	0	1	1	2	3	3	3	3	3	3	3	3
b	0	1	2	2	3	4	4	4	4	4	4	4
a	0	1	2	2	3	4	5	5	5	5	5	5
f	0	1	2	2	3	4	5	5	6	6	6	6
g	0	1	2	2	3	4	5	5	6	7	7	7
z	0	1	2	2	3	4	5	5	6	7	7	7
h	0	1	2	2	3	4	5	5	6	7	7	8

Table14

As shows above in table 14, current position highlighted as yellow cell, is not a match position, now path have to decide how it should move when current position is not match position? Here observation 2 will help to find next move,

Observation 2: If current cell is not matching position and value of cell just one above of it has same value, It means current value is propagated from above row and not a result of match occurred in current row and due to this path can directly move to one cell above.

	0	a	b	d	a	b	a	c	f	g	i	h
0	0	0	0	0	0	0	0	0	0	0	0	0
a	0	1	1	1	1	1	1	1	1	1	1	1
d	0	1	1	2	2	2	2	2	2	2	2	2
i	0	1	1	2	2	2	2	2	2	2	3	3
c	0	1	1	2	2	2	2	3	3	3	3	3
a	0	1	1	2	3	3	3	3	3	3	3	3
b	0	1	2	2	3	4	4	4	4	4	4	4
a	0	1	2	2	3	4	5	5	5	5	5	5
f	0	1	2	2	3	4	5	5	6	6	6	6
g	0	1	2	2	3	4	5	5	6	7	7	7
z	0	1	2	2	3	4	5	5	6	7	7	7
h	0	1	2	2	3	4	5	5	6	7	7	8

Table 15

As per the observation2, parse will move to one cell above as showed in table 15,

Step 3: Again current position is not match position and value one above the current cell is less than current cell, which means observation 1 and 2 are applicable, and observation 3 will help,

Observation 3: If current cell is not matching position and value of left cell is equal to current value and cell one above to current cell has value less than current value, it means, matched occurred in somewhere in left position of current row and due to this parse should moves to one cell left.

As per the observation 3, parse will move to one cell left of current cell, as show in table 16 below,

	0	a	b	d	a	b	a	c	f	g	i	h
0	0	0	0	0	0	0	0	0	0	0	0	0
a	0	1	1	1	1	1	1	1	1	1	1	1
d	0	1	1	2	2	2	2	2	2	2	2	2
i	0	1	1	2	2	2	2	2	2	2	3	3
c	0	1	1	2	2	2	2	3	3	3	3	3
a	0	1	1	2	3	3	3	3	3	3	3	3
b	0	1	2	2	3	4	4	4	4	4	4	4
a	0	1	2	2	3	4	5	5	5	5	5	5
f	0	1	2	2	3	4	5	5	6	6	6	6
g	0	1	2	2	3	4	5	5	6	7	7	7
z	0	1	2	2	3	4	5	5	6	7	7	7
h	0	1	2	2	3	4	5	5	6	7	7	8

Table 16

Step 4: current cell is match position, so subsequence will update to LCS = "**gh**", and path will move to according to rule1 as showed in table 17 below,

	0	a	b	d	a	b	a	c	f	g	i	h
0	0	0	0	0	0	0	0	0	0	0	0	0
a	0	1	1	1	1	1	1	1	1	1	1	1
d	0	1	1	2	2	2	2	2	2	2	2	2
i	0	1	1	2	2	2	2	2	2	2	3	3
c	0	1	1	2	2	2	2	3	3	3	3	3
a	0	1	1	2	3	3	3	3	3	3	3	3
b	0	1	2	2	3	4	4	4	4	4	4	4
a	0	1	2	2	3	4	5	5	5	5	5	5
f	0	1	2	2	3	4	5	5	6	6	6	6
g	0	1	2	2	3	4	5	5	6	7	7	7
z	0	1	2	2	3	4	5	5	6	7	7	7
h	0	1	2	2	3	4	5	5	6	7	7	8

Table 17

As explained in above steps, path will move in same way and not required to show each steps explicitly. Complete path is showed below in table 18,

	0	a	b	d	a	b	a	c	f	g	i	h
0	0	0	0	0	0	0	0	0	0	0	0	0
a	0	1	1	1	1	1	1	1	1	1	1	1
d	0	1	1	2	2	2	2	2	2	2	2	2
i	0	1	1	2	2	2	2	2	2	2	3	3
c	0	1	1	2	2	2	2	3	3	3	3	3
a	0	1	1	2	3	3	3	3	3	3	3	3
b	0	1	2	2	3	4	4	4	4	4	4	4
a	0	1	2	2	3	4	5	5	5	5	5	5
f	0	1	2	2	3	4	5	5	6	6	6	6
g	0	1	2	2	3	4	5	5	6	7	7	7
z	0	1	2	2	3	4	5	5	6	7	7	7
h	0	1	2	2	3	4	5	5	6	7	7	8

Table 18

Parse end when position reaches to cell have zero value (row of all zero entries) as showed above.

Finally longest common subsequence is of length 8 and it is,

LCS = “**adabafgh**”

Implementation

C++ code is available under MIT licenses at: https://github.com/vikasawadhiya/Longest-Common-Subsequence

Here code is included only for completion of document.

lcs.hpp

```
/*******************************************************************************
** MIT License
**
** Copyright (c) 2021 VIKAS AWADHIYA
**
** Permission is hereby granted, free of charge, to any person obtaining a copy
** of this software and associated documentation files (the "Software"), to deal
** in the Software without restriction, including without limitation the rights
** to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
** copies of the Software, and to permit persons to whom the Software is
** furnished to do so, subject to the following conditions:

** The above copyright notice and this permission notice shall be included in all
** copies or substantial portions of the Software.

** THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
** IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
** FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
** AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
** LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
** OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
** SOFTWARE.
*******************************************************************************/

#ifndef LCS_HPP
#define LCS_HPP

#include <string>

namespace lcs{

std::string subsequence(const std::string& sequence1, const std::string&
sequence2);

}

#endif // LCS_HPP
```

lcs.cpp

```
/*******************************************************************************
** MIT License
**
** Copyright (c) 2021 VIKAS AWADHIYA
**
** Permission is hereby granted, free of charge, to any person obtaining a copy
** of this software and associated documentation files (the "Software"), to deal
** in the Software without restriction, including without limitation the rights
** to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
** copies of the Software, and to permit persons to whom the Software is
** furnished to do so, subject to the following conditions:

** The above copyright notice and this permission notice shall be included in all
** copies or substantial portions of the Software.

** THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
** IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
** FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
** AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
** LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
** OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
** SOFTWARE.
*******************************************************************************/


#include "lcs.hpp"
#include <vector>

namespace  {

std::string parseTable(const
std::vector<std::vector<std::string::size_type>>& table, const std::string&
sequence2){

    std::string subSeq(table.back().back(), ' ');
    std::string::reverse_iterator sIt = subSeq.rbegin();

    std::string::size_type rowIndex = table.size() - 1;
    std::string::size_type colIndex = table.front().size() - 1;

    while(rowIndex > 0){

        if(table[rowIndex - 1][colIndex] < table[rowIndex][colIndex] &&
                table[rowIndex][colIndex - 1] < table[rowIndex][colIndex]){

            *sIt = sequence2[rowIndex - 1];
            ++sIt;

            --rowIndex;
            --colIndex;
        }
        else{
            if(table[rowIndex][colIndex] == table[rowIndex - 1][colIndex]){

                --rowIndex;
```

```
            }
            else{
                --colIndex;
            }
        }
    }

    return subSeq;
}

}

std::string lcs::subsequence(const std::string &sequence1, const std::string
&sequence2){

    std::vector<std::vector<std::string::size_type>> table(
                sequence2.size() + 1,
std::vector<std::string::size_type>(sequence1.size() + 1, 0));

    for(std::string::size_type j = 0, seq2Size = sequence2.size(); j <
seq2Size; ++j){

        for(std::string::size_type i = 0, seq1Size = sequence1.size(); i <
seq1Size; ++i){

            if(sequence1[i] == sequence2[j]){

                table[j + 1][i + 1] = table[j][i] + 1;
            }
            else{
                table[j + 1][i + 1] = std::max(table[j][i + 1], table[j +
1][i]);
            }
        }
    }

    return parseTable(table, sequence2);
}
```

Summary

Goal of this document is to explain longest common subsequence and how to solve it using dynamic programming. This document doesn't cover time and space complexity analysis of dynamic programming approach but that can be studied as second step once you understand it.