April 2025, India

Longest Common Subsequence

Tutorial

Tutorial By

Vikas Awadhiya

Tutorial

Version 2.0

Tutorial By

Vikas Awadhiya

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

Introduction

If two or more than two sequences contain same elements in the same order these elements as a combine are called common sub-sequence. These sequences may have more than one sub-sequence and the longest sub-sequence among these sub-sequences is called longest common sub-sequence.

Longest common sub-sequence problem is not associated with only strings rather a concept applicable to any type of sequences for example number sequences but strings are commonly used to explain it. So, let's consider two strings "abc" and "abc". These strings are equal because in both of them character 'a' is followed by character 'b' and character 'b' is followed by character 'c'. It means these characters are in same order in both of the strings.

Now consider two other strings, "a*-bc" and "*$a#bc". These strings also have characters 'a', character 'b' and character 'c' are in same order as highlighted in gray colored cells in fig 1.0 below.

a	*	-	b	c

$	a	#	b	c

Fig 1.0

In these strings the longest common subsequence (LCS) is "abc". Sub-sequence is different from substring because in sub-sequence only the respective orders of characters matter and character may not be immediately followed by other as show above in fig 1.0, character 'a' is not immediately followed by character 'b' in both of strings rather immediately followed by character '*' and character '#'. That's why the longest common substring "bc" is two characters long but longest common sub-sequence "abc" is three characters long.

Longest common subsequence problem is an entry level problem of dynamic programming like "Hello World" program for a programming language but definitely it is not as easy as "Hello World" program for beginners.

A problem can be solved by dynamic programming if it can be divided into sub-problems and these sub-problems can be solved into reclusively and are of overlapping nature. This problem introduces both of these aspects of dynamic programming.

The Problem

In a problem, longest common sub-sequence (LCS) is required to be found for given strings. Longest common sub-sequence problem is not limited to only two sequences and there could be more than two sequences but for explaining purpose only two sequences are considered. The answer of this problem may be not unique because it is possible that the two given strings contain more than one common sub-sequence of same length obviously the longest length and due to this may have multiple answers.

This problem can be solved by naïve approach like other problems but the dynamic programming approach is the practical/efficient approach for real world problems. So, let's see the naïve/brute-force approach first and it helps to understand why dynamic programming approach is efficient one.

Naïve Approach

Let's consider two strings str$_1$ = "**abccab**" and str$_2$ = "**accabb**". The naïve approach tries to find the longest common subsequence by comparing the characters of the strings with considering all possibilities as follows,

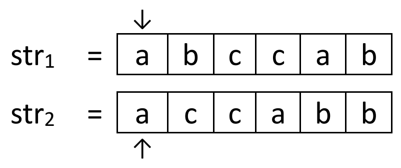

Fig 2.0

As show above in the fig 2.0 comparison begins from first character of both of the strings. It is a match, so the LCS = "a" and matching continues as show below in fig 3.0.

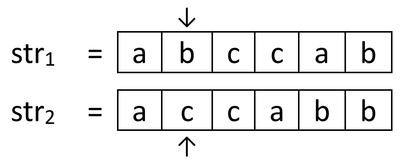

Fig 3.0

It is a mismatch, so no change in LCS and pointer moves to the next character in str$_2$ and matching continues as show below in fig 4.0.

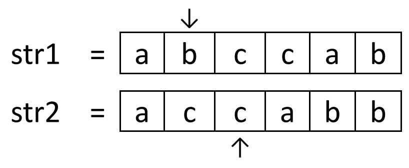

Fig 4.0

This is another mismatch and this continues until the pointer reaches to the second last character of the str$_2$ as show below in fig 5.0.

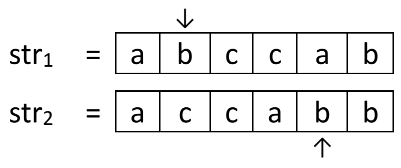

Fig 5.0

This is a match and LCS become LCS = “ab”. Pointers of both the sequences move and matching continues as show below in fig 6.0.

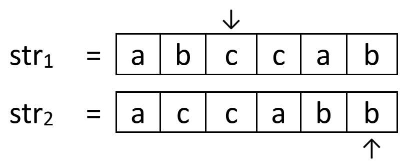

Fig 6.0

Now next three characters of str_1 cannot match because str_2 has only one character ‘b’ left and there is only one possible match if pointer reaches to the last character of str_1 as show below in fig 7.0.

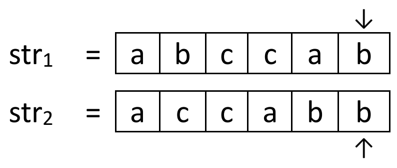

Fig 7.0

The LCS updated to LCS = “abb” but this is definitely not a longest common subsequence. If we reconsider the decision of not moving the pointer in str_1 when a mismatch happened at character ‘b’ and move the pointer to the next character in str_1 as follows then all remaining four characters do match as highlighted by gray colored cells in fig 8.0 below.

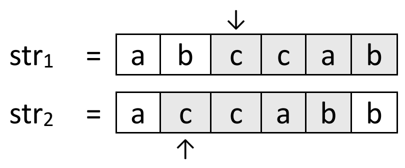

Fig 8.0

This clearly highlights the possibility that even not considering a single character drastically changes the outcome from LCS = “abb” to LCS = “accab”.

In naïve approach the longest common subsequence problem becomes a combinatory problem of $O(2^n)$ exponential time complexity. This cannot be a practical approach where all the possibilities have to be checked because in a real-world scenario, the sequences are unlikely to be as short as six characters as they are in this example. This $O(2^n)$ time complexity becomes impractically large for even moderate sized inputs.

The dynamic programming approach can solve this problem in $O(m \times n)$ time complexity. It is also called polynomial time complexity because if the size of both the sequences is n then it becomes $O(n^2)$ which is quadratic polynomial and it doesn’t grow exponentially.

Dynamic Programming Approach

The dynamic programming approach focuses on individual characters and obtains efficiency by storing the results evaluated in the previous steps to avoid revaluation.

Let’s reconsider the strings str_1 = “**abccab**” and str_2 = “**accabb**” as considered in naïve approach.

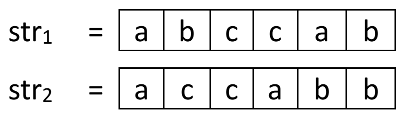

Fig 9.0

In this approach a character matches again all characters of other string and doesn’t stop when a match occurs rather it stores match information and match continues.

Let’s see character matching for individual characters and it starts with $str_2[0]$ as follows,

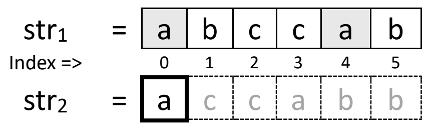

Fig 10.0

As highlighted by gray colored cells in fig 10.0 above, $str_2[0]$ matches at index 0 and index 4 in str_1 and after this character match LCS becomes LCS = “a” and LCS may update as matching proceeds.

Character matching for $str_2[1]$ is as follows,

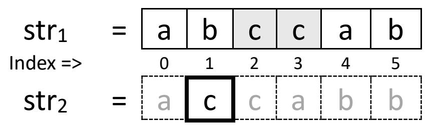

Fig 11.0

As highlighted by gray colored cells above in fig 11.0, $str_2[1]$ matches at index 2 and index 3 in str_1. Now onwards dynamic programming helps to find the LCS. The match result of previous character $str_2[0]$ and this character $str_2[1]$ are listed in fig 12.0 below on the left,

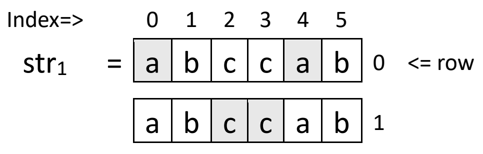

Fig 12.0

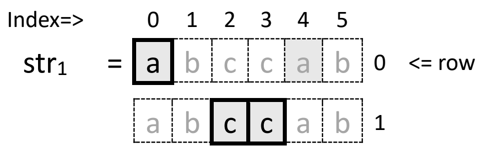

Fig 12.1

As highlighted above on the left in fig 12.0 row_1 represent match result of $str_2[0]$. The results with LCS perspective are shown in fig 12.1 above on the right, match positions of $str_2[1]$ in str_1, at index 2 or index 3 has a preceding match at index 0 in row_0 and these matches are highlighted by bold boundary cells. The row_0 represents search result of $str_2[0]$ in str_1. It means the characters $str_2[0]$ and $str_2[1]$ are in the same order in both of the strings. Because of it str$_2$[1] is appended in LCS and it becomes LCS = “ac”. This can be observed in fig 13.0 as shown below where bold boundary cells are in same order in both of the strings.

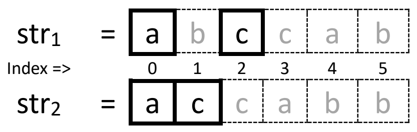

Fig 13.0

Character matching continues for str$_2$[2] and gray colored cells highlight the positions in str$_1$ where the match occurs.

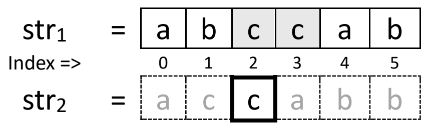

Fig 14.0

The match results of all matched characters so far are listed in fig 15.0 below on the left.

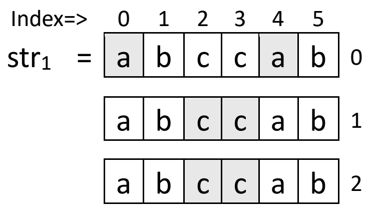

Fig 15.0

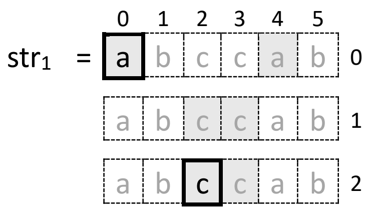

Fig 15.1

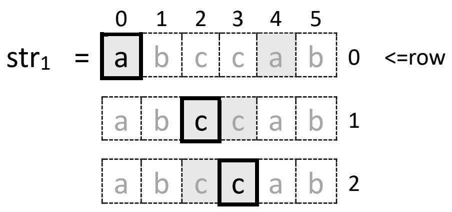

Fig 15.2

In fig 15.0 above on the left, row$_2$ represent match results of str$_2$[2]. Match at index 2 in str$_1$ as shown in row$_2$ in fig 15.1 above in the centre, has only one preceding match, highlighted by bold boundary cells and that is at row$_0$ which represent str$_2$[0] match result. In this position, possible LCS is “ac” but LCS is already 2 character long. The other match position is at index 3 in str$_1$ as shown in fig 15.2 above on the right, row$_1$ which represents match result of str$_2$[1] has a match occurrence at index 2 which is preceding position of index 3 and then with respect to this match there is another preceding match at index 0 in row$_0$ which represents match results of str$_2$[0] as highlighted by bold boundary cells. That means three characters are in same order in both of the string. This can be viewed fig 16.0 below where bold boundary cells are in same order in both of the strings. After character match of str$_2$[2] LCS becomes LCS = “acc”.

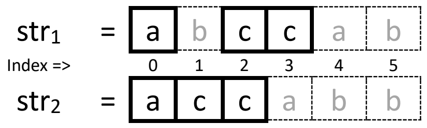

Fig 16.0

Character matching continues for character str$_2$[3] as follows,

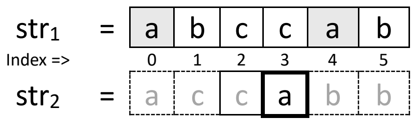

Fig 17.0

The match results of str$_2$[3] are highlighted by gray colored cells in fig 17.0 above.

The match results of all matched characters so far are listed in fig 18.0 below on the left.

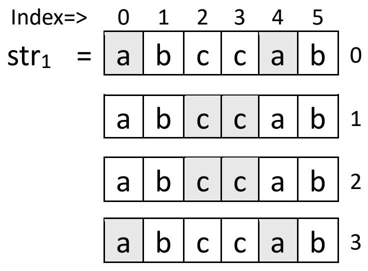

Fig 18.0

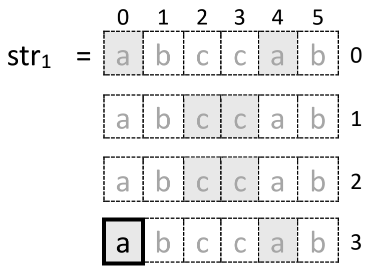

Fig 18.1

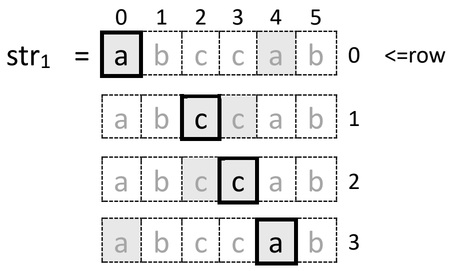

Fig 18.2

In fig 18.0 above on the left, row_3 represent match results of character $str_2[3]$. Match at index 0 in str_1 as shown in row_3 in fig 18.1 above in the centre, has no preceding match and due to this doesn't affect LCS and which is already three characters long. The other match position is at index 4 in str_1 as shown in row_3 in fig 18.2 above on the right has a preceding match at index 3 in row_2 which represents match results of $str_2[2]$ that further has preceding match at index 2 as shown in row_1 which also has a preceding match at index 0 at row_0 in fig 18.2 highlighted by bold boundary cells. It means match of $str_2[3]$ at index 4 can be appended in LCS or in other words all four characters are in same order in both strings. This can be observed in fig 19.0 below where bold boundary cells are in same order in both of the strings.

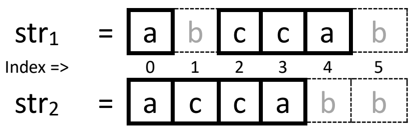

Fig 19.0

After character match of $str_2[3]$ LCS becomes LCS = "acca", four character long.

Similarly, the match results of remaining two characters $str_2[4]$ and $str_2[5]$ can also be evaluated and after evaluation LCS becomes LCS = "accab" and the process ends.

To find the longest common sub-sequence between two strings dynamic programming technique make $m \times n$ comparisons as already seen, n characters of str_2 compared with m characters of str_1. The space it requires is also $m \times n$ to store the characters match results of n characters of str_2 with m characters of str_1.

The dynamic programming approach by using $m \times n$ space can solve the problem in $O(m \times n)$ time complexity as compared to unrealistic exponential time complexity $O(2^n)$ of naïve approach.

This section explained theoretical and logical perspective or to say the idea behind the dynamic programming approach but $m \times n$ table creation is a crucial perspective which is remained to explain to solve this problem. Next section explains it.

Table Creation

A m × n table creation is crucial to find the longest common sub-sequence. This table contains the information in terms of length of LCS found so far and it gets updated with each character match.

Let's consider two string str_1 = "**abdabacfgih**" and str_2 = "**adicabafgzh**" and table for these are shown below in fig 20.0.

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

Fig 20.0

The LCS can be found by traversing the table in bottom-up manner after table crated or in other words when the character matching completes.

The rows and columns represent str_2 and str_1 respectively. This table is not of order m × n rather it has one extra row and one extra column. The reason behind extra row and column is, on character match or mismatch current cell get its value by looking at the entry of previous row and the pervious column and this extra row and column allows to access previous row and col at character matching for first character.

To construct the table, it is first required to understand what happens when a match or mismatch occurs.

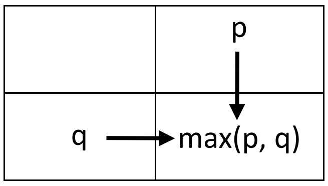

Fig 21.0

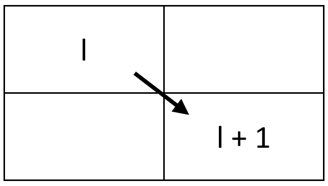

Fig 21.1

The match and mismatch are handled differently in table creation. The giant cells shown in fig 21.0 above on the left represents the mismatch scenario and, on a mismatch, the value assigned to current cell is the maximum value among the previous cell (left col) and the cell above of it (same col in previous row). The reason behind it is that a mismatch doesn't affect the LCS and the previous value of LCS must be carried forward but there are two possibilities, the first one is that this character doesn't affect LCS at all and due to this LCS value (in terms of length) must be equal to the value what is was when character matching was performed at same index for the previous character (above cell). The second possibility is that this character has a mismatch here but may has a match at some previous index (previous cell). Among both of these possibilities the one which forming the longest subsequence is begin considered.

The giant fig 12.1 above on the right represents the match scenario and as shown above, on a match value of current cell is assigned the value of diagonal cell (previous cell in previous row) plus one because if a match occurs it is appened to the already known LCS.

Now let's see the character matching for $str_2[0]$ as follows,

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

Fig 22.0

$Str_2[0]$ matches with $str_1[0]$ and that's why the value assigned to table[0][0] is the value of diagonal cell (previous col in previous row) plus one. Here 1 represents the length of LCS after this match.

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

Fig 23.0

$Str_2[0]$ doesn't match with $str_1[1]$ and that's why the value assigned to table[0][1] is the max(table[0][0], value of above cell (same col in previous row)) = 1. Similarly, the value of entire row_0 is evaluated as follows,

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

Fig 24.0

Matching continues for $str_2[1]$ as follows,

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

Fig 25.0

Str_2[1] doesn’t match at first two characters but match happens at str_1[2]and this character can be appended to the LCS. That’s why the value assigned to table[1][2] = table[0][1] + 1.

Matching continues for Str_2[2] and it matches only at index 9 in str_1 as follows,

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

Fig 26.0

Matching continues for str_2[3] and it matches only at index 6 in str_1 as follows,

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

Fig 27.0

Similarly, the remaining characters of str2 can be matched and the remaining rows populates as shown below in fig 28.0

			0	1	2	3	4	5	6	7	8	9	10
		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
0	a	0	1	1	1	1	1	1	1	1	1	1	1
1	d	0	1	1	2	2	2	2	2	2	2	2	2
2	i	0	1	1	2	2	2	2	2	2	2	3	3
3	c	0	1	1	2	2	2	2	3	3	3	3	3
4	a	0	1	1	2	3	3	3	3	3	3	3	3
5	b	0	1	2	2	3	4	4	4	4	4	4	4
6	a	0	1	2	2	3	4	5	5	5	5	5	5
7	f	0	1	2	2	3	4	5	5	6	6	6	6
8	g	0	1	2	2	3	4	5	5	6	7	7	7
9	z	0	1	2	2	3	4	5	5	6	7	7	7
10	h	0	1	2	2	3	4	5	5	6	7	7	8

Fig 28.0

Once the table is created the longest common sub-sequence can be found by observing the table by traversing in bottom-up manner. This is the final step to find the LCS and next section discusses it.

Bottom-Up Traversal Of The Table

To find the longest common sub-sequence table is required to be travers in bottom-up approach but the length of LCS can be found by looking at rightmost cell of the table.

6	a	0	1	2	2	3	4	5	5	5	5	5	5
7	f	0	1	2	2	3	4	5	5	6	6	6	6
8	g	0	1	2	2	3	4	5	5	6	7	7	7
9	z	0	1	2	2	3	4	5	5	6	7	7	7
10	h	0	1	2	2	3	4	5	5	6	7	7	8

Fig 29.0

As highlighted by gray colored cell above in fig 29.0 the rightmost cell, table[10][10] have value 8 It means possible LCS among these two strings is eight characters long.

The traversal in bottom-up approach begins from rightmost cell in this case from table[10][10] as show below in fig 30.0 below on the left and rightmost cell highlighted by gray colred cell with bold boundary.

			0	1	2	3	4	5	6	7	8	9	10
		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
0	a	0	1	1	1	1	1	1	1	1	1	1	1
1	d	0	1	1	2	2	2	2	2	2	2	2	2
2	i	0	1	1	2	2	2	2	2	2	2	3	3
3	c	0	1	1	2	2	2	2	3	3	3	3	3
4	a	0	1	1	2	3	3	3	3	3	3	3	3
5	b	0	1	2	2	3	4	4	4	4	4	4	4
6	a	0	1	2	2	3	4	5	5	5	5	5	5
7	f	0	1	2	2	3	4	5	5	6	6	6	6
8	g	0	1	2	2	3	4	5	5	6	7	7	7
9	z	0	1	2	2	3	4	5	5	6	7	7	7
10	h	0	1	2	2	3	4	5	5	6	7	7	8

Fig 30.0

			0	1	2	3	4	5	6	7	8	9	10
		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
0	a	0	1	1	1	1	1	1	1	1	1	1	1
1	d	0	1	1	2	2	2	2	2	2	2	2	2
2	i	0	1	1	2	2	2	2	2	2	2	3	3
3	c	0	1	1	2	2	2	2	3	3	3	3	3
4	a	0	1	1	2	3	3	3	3	3	3	3	3
5	b	0	1	2	2	3	4	4	4	4	4	4	4
6	a	0	1	2	2	3	4	5	5	5	5	5	5
7	f	0	1	2	2	3	4	5	5	6	6	6	6
8	g	0	1	2	2	3	4	5	5	6	7	7	7
9	z	0	1	2	2	3	4	5	5	6	7	7	7
10	h	0	1	2	2	3	4	5	5	6	7	7	8

Fig 30.1

The cell table[10][10] has value 8 which indicated there is a match occurred because 8 ≠ max(table[10][9], table[9][10]) rather its value is greater than cell above of it and previous cell. Because it is a match and character at str$_2$[10] which is 'h' is included in LCS due to this cell is highlighted by the bold boundary. LCS becomes LCS = "h".

Traversal moves to diagonal cell which is row$_9$ and col$_9$ as shown above in fig 30.1 above on the right. The value of table[9][10] = 7 = max(table[9][8], table[8][9]) and it is a mismatch condition. Traversal has to move to another cell and it should move to the cell having max value but in this case both, above cell and previous cell have same value and due to this traversal can moves to the any cell, so let's move to the above cell which is table[8][9] as shown in fig 31.0 below on the left.

			0	1	2	3	4	5	6	7	8	9	10
		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
0	a	0	1	1	1	1	1	1	1	1	1	1	1
1	d	0	1	1	2	2	2	2	2	2	2	2	2
2	i	0	1	1	2	2	2	2	2	2	2	3	3
3	c	0	1	1	2	2	2	2	3	3	3	3	3
4	a	0	1	1	2	3	3	3	3	3	3	3	3
5	b	0	1	2	2	3	4	4	4	4	4	4	4
6	a	0	1	2	2	3	4	5	5	5	5	5	5
7	f	0	1	2	2	3	4	5	5	6	6	6	6
8	g	0	1	2	2	3	4	5	5	6	7	7	7
9	z	0	1	2	2	3	4	5	5	6	7	7	7
10	h	0	1	2	2	3	4	5	5	6	7	7	**8**

Fig 31.0

			0	1	2	3	4	5	6	7	8	9	10
		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
0	a	0	1	1	1	1	1	1	1	1	1	1	1
1	d	0	1	1	2	2	2	2	2	2	2	2	2
2	i	0	1	1	2	2	2	2	2	2	2	3	3
3	c	0	1	1	2	2	2	2	3	3	3	3	3
4	a	0	1	1	2	3	3	3	3	3	3	3	3
5	b	0	1	2	2	3	4	4	4	4	4	4	4
6	a	0	1	2	2	3	4	5	5	5	5	5	5
7	f	0	1	2	2	3	4	5	5	6	6	6	6
8	g	0	1	2	2	3	4	5	5	6	**7**	7	7
9	z	0	1	2	2	3	4	5	5	6	7	7	7
10	h	0	1	2	2	3	4	5	5	6	7	7	**8**

Fig 31.1

The value at table[8][9] is 7 as shown in fig 31.0 above on the left and it indicate a mismatch. Its value is equals to the value of previous cell as 7 = max(table[7][9], table[8][8]). Because of mismatch traversal moves to the previous cell which has greater value as show in fig 31.1 above on the right.

The value at table[8][8] indicates a match because table[8][8] = table[7][7] + 1 and because of match cell is highlighted by bold boundary and character $str_2[7]$ is appended in LCS as shown in the fig 31.1 above on the right. LCS becomes LCS = "hg". Traversal moves to the previous diagonal cell as shown in fig 32.0 below on the left.

			0	1	2	3	4	5	6	7	8	9	10
		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
0	a	0	1	1	1	1	1	1	1	1	1	1	1
1	d	0	1	1	2	2	2	2	2	2	2	2	2
2	i	0	1	1	2	2	2	2	2	2	2	3	3
3	c	0	1	1	2	2	2	2	3	3	3	3	3
4	a	0	1	1	2	3	3	3	3	3	3	3	3
5	b	0	1	2	2	3	4	4	4	4	4	4	4
6	a	0	1	2	2	3	4	5	5	5	5	5	5
7	f	0	1	2	2	3	4	5	5	**6**	6	6	6
8	g	0	1	2	2	3	4	5	5	6	**7**	7	7
9	z	0	1	2	2	3	4	5	5	6	7	7	7
10	h	0	1	2	2	3	4	5	5	6	7	7	**8**

Fig 32.0

			0	1	2	3	4	5	6	7	8	9	10
		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
0	a	0	1	1	1	1	1	1	1	1	1	1	1
1	d	0	1	1	2	2	2	2	2	2	2	2	2
2	i	0	1	1	2	2	2	2	2	2	2	3	3
3	c	0	1	1	2	2	2	2	3	3	3	3	3
4	a	0	1	1	2	3	3	3	3	3	3	3	3
5	b	0	1	2	2	3	4	4	4	4	4	4	4
6	a	0	1	2	2	3	4	5	**5**	5	5	5	5
7	f	0	1	2	2	3	4	5	5	**6**	6	6	6
8	g	0	1	2	2	3	4	5	5	6	**7**	7	7
9	z	0	1	2	2	3	4	5	5	6	7	7	7
10	h	0	1	2	2	3	4	5	5	6	7	7	**8**

Fig 32.1

The current cell table[7][7] also represent a match occurrence having value 6 = table[6][6] + 1 and due to this highlighted by bold boundary in fig 32.0 above on the left.

Because of match character str$_2$[7] is append in LCS and it becomes LCS = “hgf” and traversal moves to previous diagonal cell table[6][6] as shown in fig 32.1 above on the right.

Traversal stops when It moves beyond the 0th row and at this point LCS is completely evaluated. Final sate of traversal is shown below in fig 33.0.

			0	1	2	3	4	5	6	7	8	9	10
		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
0	a	0	**1**	1	1	1	1	1	1	1	1	1	1
1	d	0	1	1	**2**	2	2	2	2	2	2	2	2
2	i	0	1	1	2	2	2	2	2	2	2	3	3
3	c	0	1	1	2	2	2	2	3	3	3	3	3
4	a	0	1	1	2	**3**	3	3	3	3	3	3	3
5	b	0	1	2	2	3	**4**	4	4	4	4	4	4
6	a	0	1	2	2	3	4	**5**	5	5	5	5	5
7	f	0	1	2	2	3	4	5	5	**6**	6	6	6
8	g	0	1	2	2	3	4	5	5	6	**7**	7	7
9	z	0	1	2	2	3	4	5	5	6	7	7	7
10	h	0	1	2	2	3	4	5	5	6	7	7	**8**

Fig 33.0

All the gray colored cells with bold boundary are the match positions and the respective characters are appended in LCS. The final LCS becomes LCS = “hgfabada” but it is not the answer.

Because it is a bottom-up traversal begins from rightmost cell and moves upward and due to this the LCS it evaluates must be reversed.

In this case evaluated LCS after reversal becomes LCS = “adabafgh” and that is the answer.

Implementation

The C++ implementation of the longest common subsequence is provided under the MIT License. It also contains the “main.cpp” file to demonstrate its usage.

Code available under MIT License at: https://github.com/vikasawadhiya/Longest-Common-Subsequence

April 2025, India