Common Child

Isaac DeJager

March 15, 2023

Click here to view the HackerRank page

1 Problem Description

A string is said to be a child of another string if it can be formed by deleting 0 or more characters from the other string. Letters cannot be rearranged. Given two strings of equal length, what's the longest string that can be constructed such that it is a child of both?

Example

$s1 =$ "ABCD" $s2 =$ "ABDC"

These strings have two common children with maximum length 3, "ABC" and "ABD". They can be formed by eliminating either "D" or "C" from both strings. Returns 3.

Function Description

Complete the commonChild function in the editor below, commonChild has the following parameters:

- string s1: a string
- string s2: another string

Returns

- int: the length of the longest string which is a common child of the input strings

Input Format

There are two lines, each with a string $s1$ and $s2$.

Constraints

- $1 \leq |s1|, |s2| \leq 5000$ where $|s|$ means "the length of s"
- All characters are upper case in the range ascii[A-Z]

Sample Input 0
HARRY SALLY

Sample Output 0
2

Explanation 0
The longest string that can be formed by deleting zero or more characters from ”HARRY” and ”SALLY” is ”AY”, whose length is 2.

Sample Input 1
AA BB

Sample Output 1
0

Explanation 1
”AA” and ””BB” anve no characters in common and hence the output is 0.

Sample Input 2
SHINCHAN NOHARAAA

Sample Output 2
3

Explanation 2
”BD” is the longest child of the given strings and has length 2.

2 Solution

```
int commonChild(string s1, string s2) {

    //Create arrays that track the LCS between any pairs
    //of letters
    int arr[2][s2.length() + 1];
    memset(arr, 0, sizeof(arr));

    //Loop through every char in s1, compare to
    //every char in s2
    for (int i = 0; i < s1.length(); i++) {
        for ( int j = 1; j < s2.length() + 1; j++) {

            //if they equal then the LCS equals the LCS of
            //the strings minus that char + 1
            if (s1.at(i) == s2.at(j - 1)) {
                arr[(i + 1) % 2][j] = arr[i % 2][j - 1] + 1;
            }

            //otherwise it equals the LCS minus the char on
            //one of the strings
            else {
                arr[(i + 1) % 2][j] = arr[(i + 1) % 2][j - 1];
                if (arr[(i + 1) % 2][j] < arr[i % 2][j]) {
                    arr[(i + 1) % 2][j] = arr[i % 2][j];
                }
            }
        }
    }

    return arr[s1.length() % 2][s2.length()];

}
```

3 Analysis

This problem is essentially the Longest Common Subsequence problem simplified: given any two strings, return the longest common substring (LCS). Here though, we need only return the length of the LCS, not the LCS itself. While this doesn't reduce time complexity, it does reduce the space requirements. To understand the above solution, lets first look at a description to the solution to the Longest Common Subsequence problem.

Given the strings $s1$ and $s2$, create a table with $|s1| + 1$ rows and $|s2| + 1$ columns and fully initialize to 0. The rows and columns represent the corresponding characters of $s1$ and $s2$. As an example, consider the strings:
$s1$ = "HARRY"
$s2$ = "SALLY"
So the table, when initialized, looks like:

	-	**S**	**A**	**L**	**L**	**Y**
-	0	0	0	0	0	0
H	0	0	0	0	0	0
A	0	0	0	0	0	0
R	0	0	0	0	0	0
R	0	0	0	0	0	0
Y	0	0	0	0	0	0

The objective of this table is to calculate the length of the LCS between any two strings that lead up to the corresponding characters. For example, consider the element at [A, Y]. Whatever value is in this box represents the length of the LCS between strings:
$s1'$ = "HA"
$s2'$ = "SALLY"
The way that we calculate it can be easily derived from examining the length of the LCS of marginally smaller strings, specifically the length of strings without the characters "A" or "Y" respectively.
The contents of the table are calculated as so:

- Loop through each table element, left to right and up to down (starting at "H" v "S" first)
- If the characters are different, then examine the values of the table directly above and directly left. Chose the higher value and place that in the current table element
- If the characters are the same, then examine the value of the table at the upper diagonal left. Add one to it and place that in the current table element

Performing these actions on every table element populates the table like so:

	-	**S**	**A**	**L**	**L**	**Y**
-	0	0	0	0	0	0
H	0	0	0	0	0	0
A	0	0	1	1	1	1
R	0	0	1	1	1	1
R	0	0	1	1	1	1
Y	0	0	1	1	1	2

The value in the bottom right corner of the table is the length of the LCS. You can return the actual LCS by creating arrow pointers whenever you fill a table element, but for purposes of this problem we need only return the length.

The logic is that every time you add a new character to either substring, you can find the length of the LCS by looking at the length of the LCS before you added this character. For example, when looking to fill [A, A], you are looking for the LCS of substrings:

$s1' =$ "SA"

$s2' =$ "HA"

Since "A" = "A", you look to see the length of the LCS between substrings:

$s1'' =$ "S"

$s2'' =$ "H"

Since that is the top diagonal left. Add 1 to that value add you have the length of the LCS between $s1'$ and $s2'$.

Now to check the next box:

$s1' =$ "SAL"

$s2' =$ "HA"

Since "L" $\neq$ "A", we need to see which of the following substring pairs have a longer LCS:

$s1'a =$ "SA"

$s2'a =$ "HA"

OR

$s1'b =$ "SA"

$s2'b =$ "H"

Since the first pair has the longer LCS, then we can say that the length of the LCS between $s1'$ and $s2'$ is 1.

And the process continues. Every box is calculated recursively, since the easiest way to find the length of the LCS between any to substrings is too look at the length of the LCS between slightly smaller substrings.

And now, to apply this to the above solution. Since there is no need to return the actual LCS, only its length, we do not need to create a massive 2D array. We only need two rows, since each row of the full array is calculated by only looking at the row directly above it.

So, the solution begins by creating a 2D array arr with only 2 rows and $|s2|+1$ columns, fully initialized at 0. Then we enter a nested for loop to compare every character in $s1$ to $s2$. The outer loop runs from $i = 0$ to $i = |s1| - 1$, while the inner runs from $j = 1$ to $j = |s2|$. As we run through the loops, we are repeatedly rewriting the rows of arr. For $i = 0$, we write row 1 and leave row 0 as all 0s, just like in the full array. Every time a character matches, we just add 1 to the left diagonal (which is 0 at the start). And every time a character doesn't match, we find the larger between the element directly above and to the left and input the greater.

When $i = 1$, we rewrite row 0 and use modulos to treat row 1 as if it were below row 0. Then we repeat until we've cylced through every pair of letters. Once we're done, whichever row we finished on holds the length of the LCS at its final entry.

The time complexity of this algorithm is easily computed. Assume $|s1| = n$ and $|s2| = m$ where $n \geq m$. Then the time complexity is:

$$O(n * m) \leq O(n * n) = O(n^2)$$

If this algorithm were to return the LCS instead of just this length, then the space complexity would be a 2D array of dimensions $[n + 1,\ m + 1]$. Which, when accounting for the worst case scenario, turns into $O(n^2)$ space.

As it is however, since we use a 2D array with only 2 rows, the space complexity comes out to:

$$O(2 * m) \leq O(2 * n) = O(n)$$