MINISTRY OF EDUCATION AND SCIENCE OF UKRAINE
VN Karazin Kharkiv National University School of Mathematics and Computer Science
Department of Theoretical and Applied Informatics

Master's Thesis

IMPLEMENTATION OF THE LONGEST COMMON SUBSEQUENCE (LCS) ALGORITHM FOR STRING MATCHING

Final year Master's Program student,
group MCS-64
specialty - Computer Sciences and Information Technologies,
Author: Yang Ye
educational program: "Informatics"
Supervisor: Lyudmila Polyakova
Reviewer: Momot Myroslav
Adviser: Illia Ilin

Kharkiv, 2024

Longest Common Subsequence (LCS) means that you will be given two strings/patterns/sequences of objects. Among these two sequences/strings, you need to find the longest subsequence of elements in the same order that are present in both the strings and the patterns.

Example

For example, two lines are provided.

Let's assume that,

Pattern_1 = “RGBGARGA” Pattern_2 = “BGRARG”

- From template_1 you can create sequences of type "RGB", "RGGA", "RGAR". To create a sequence you need to maintain the relative position of each character in the string.
- From template_2 we can create sequences such as "BGR", "BRAG", "RARG". Sequences can be created provided that they preserve the relative position of the original string.

The term relative position means order.

For example, "BRG" is a valid sequence because in the original string "pattern_2" "B" appears first, then "R" and then "G". However, if the sequence were "RBRG", it would be invalid because "B" appears first in the original string (pattern_2).

R	G	B	G	A	R	G	A
B	G	R	A	R	G		

The LCS is “RARG” with length of 4

We have two options to find the longest common subsequence from the given two sequences or arrays.

- Naive method
- Dynamic Programming Solution: Longest Common Subsequence is also known as LCS.

The naive solution has a large time complexity and is not an optimal solution. Using a dynamic programming (DP) solution, we overcome the complexity problem.

Naive method

The naive method is a simple approach to the problem regardless of time complexity and other optimization factors.

The naive method consists of brute force, lots of loops and in most cases recursive methods.

The term "brute force" means trying all possible patterns to solve a given problem.

Example

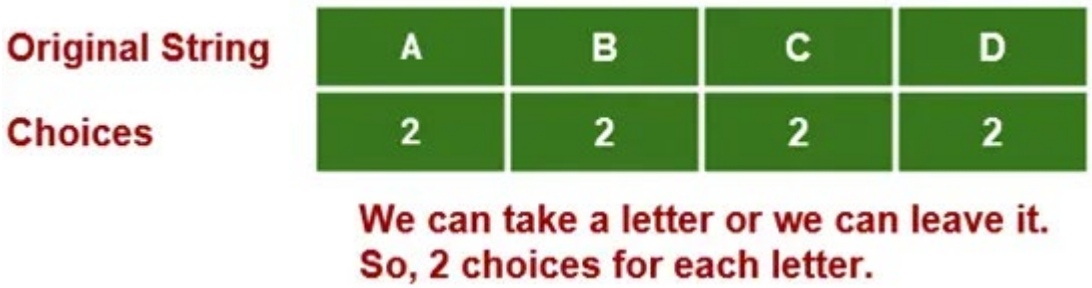
From the above example of pattern1 and pattern2, suppose pattern1 has length m and pattern2 has length n. To check all possible cases, we need to evaluate every possible subsequence of pattern1 with pattern2.

Here is a simple string of 4 letters "ABCD". For example, we need to create a sequence of "ABCD". Either we can take a character or not. This means that for each character we have two options to choose from.

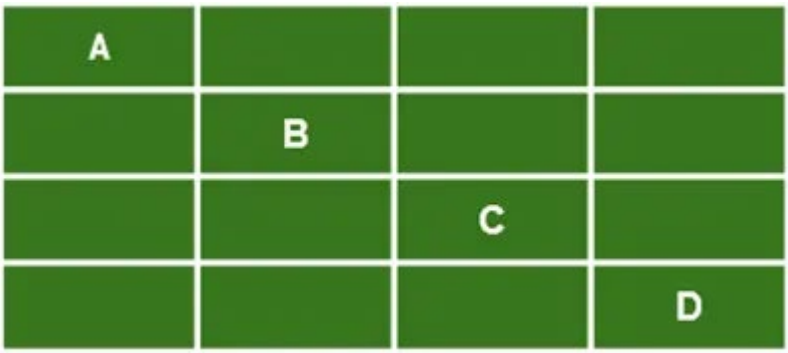
These include:

- The character will be added to the subsequence.
- The symbol will not be added to the subsequence.

Here the images show all the sequences we can make from the string "ABCD".



Sequence of 1 character:



Sequences of 2 characters:

A	B		
A		C	
A			D
	B	C	
	B		D
		C	D

Sequences of 3 characters:

A	B	C	
A	B		D
A		C	D
	B	C	D

The diagram above shows 14 sequences. If we do not take the letters, but essentially an empty string, then there will be 15 sequences in total. Moreover, the string "ABCD" is itself a sequence. So, there are 16 sequences in total.

So, you can create 2^4or 16 subsequences of "ABCD". Then a string of length m will have to be made up of a subsequence of 2^m.

For each subsequence, we need to check it against the entire pattern2. This will take 0(n) time. 0(n) stands for the complexity function, which calculates the time it takes to execute.

Thus, the overall time complexity becomes O(n*2^m**).**As an example, we saw above the values m=8 and n=5.

Here are the steps of the naive method:

Step 1) Take the sequence from template1.
Step 2) Match the sequence of step 1 with pattern 2.
Step 3) If it matches, save the subsequence.
Step 4) If there is more sequence left in Pattern 1, go to Step 1 again.
Step 5) Print the longest subsequence.

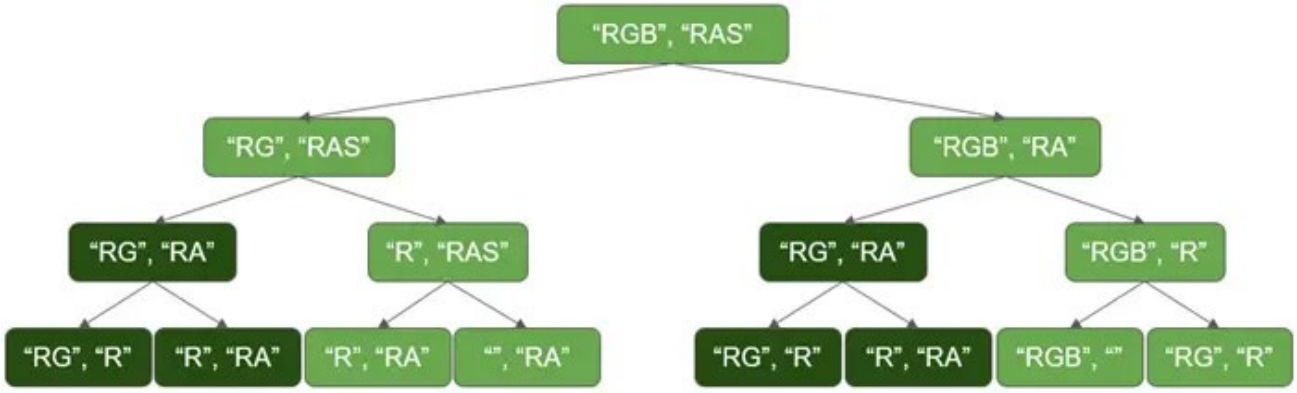
Optimal basis

The term "optimal substructure" means that the optimal (simple) solution can be found by solving subproblems. For example, in the example above, we have pattern1 and pattern2.

Step 1) Take the first two characters from each pattern.

Step 2) Take the third to fifth characters from each pattern.

Step 3) Continue in the same manner with the remaining characters.



Recursive structure of the LCS problem
We find the LCS on a substring (a string generated from the original string). Then we keep track of the length of the LCS of the substrings.

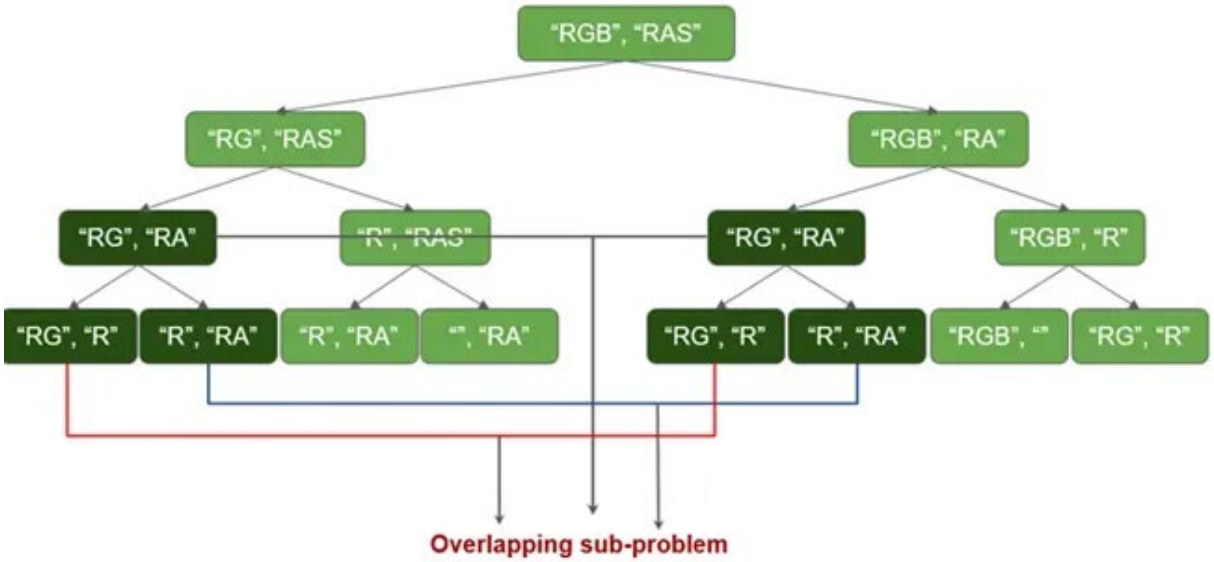
Here's another interesting property: overlapping subtasks. A task is said to have overlapping subtasks if the task formulation can be broken down into smaller subtasks and used multiple times in the program.

The diagram below shows that the recursive algorithm called a function multiple times with the same parameter.

For example, let's look at a recursion tree.

In the dark field you can see overlapping subtasks. ("RG", "RA"), ("RG", "R") and others are named several times.

To optimize this process we have the Dynamic Programming (DP) approach.



RELATED ARTICLES

- **Circular Linked List: Advantages and Disadvantages**
- **Array in Data Structure: What are Arrays Operations [Examples]**
- **Heapsort Algorithm (with Python and C++ code)**
- **Tower of Hanoi Algorithm: Python, C++ Code**

Recursive Longest Message Sequence Method

The graph above represents a recursive method. Each recursive function has a base variant that allows you to break the recursion or start returning from the stack.

For this implementation we will use the base case. So the algorithm looks like this:

- If all elements before the last element are the same, increase the length by one and return the result.
- Pass two templates to the function and take the maximum return value.
- If one pattern has zero length, then we have no subsequence to compare. In this case, return 0. This is the basic case of recursion.

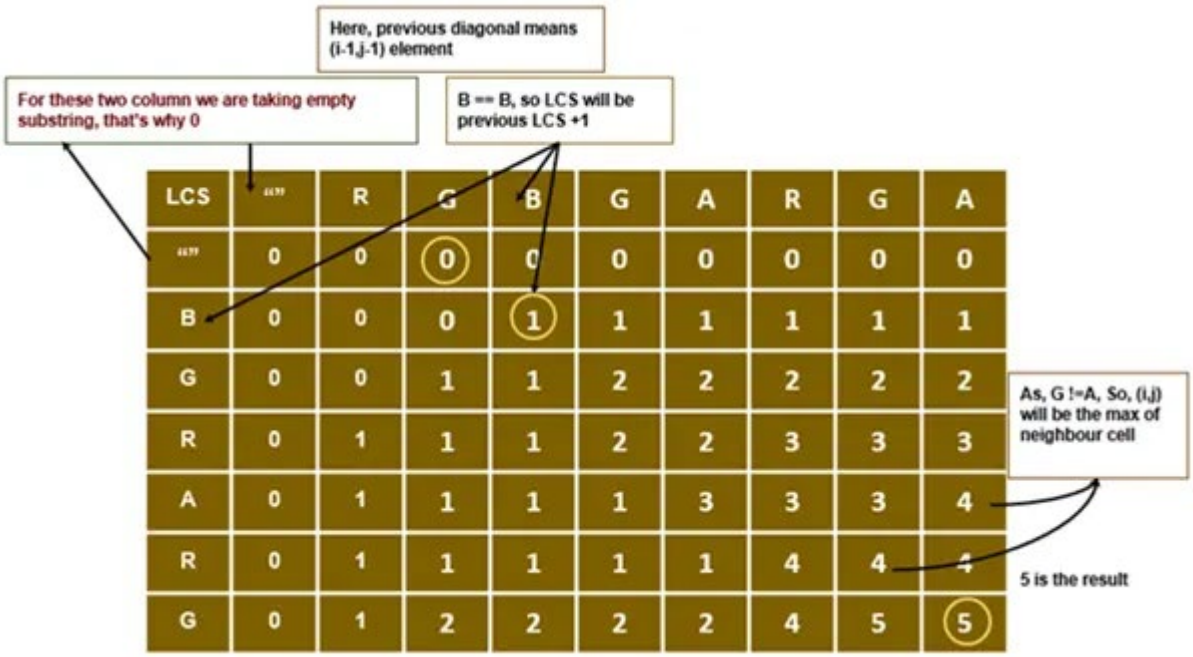
Longest Common Subsequence (LCS) Dynamic Programming Method

Dynamic programming means optimizing a simple recursive method. For example, if we see a graph of a recursive or naive approach, we see that there are several identical function calls. The dynamic programming method writes all the calculations into an array and uses it when needed.

We will use a two-dimensional array of size mxn, where m and n are the lengths of template2 and template1.

As for2D array, we can use List data structures in Python or Vector/Array data structures in C++.

Here is the LCS table which is used as a two-dimensional array data structure for the dynamic programming approach.



LCS	""	R	G	B	G	A	R	G	A
""	0	0	0	0	0	0	0	0	0
B	0	0	0	1	1	1	1	1	1
G	0	0	1	1	2	2	2	2	2
R	0	1	1	1	2	2	3	3	3
A	0	1	1	1	1	3	3	3	4
R	0	1	1	1	1	1	4	4	4
G	0	1	2	2	2	2	4	5	5

Let's discuss the logic we used here. Steps:

Step 1) If i or j is zero, we take the empty string from the given two strings and try to find common subsequences. However, since the substring we take is empty, the length of the subsequence is 0.

Step 2) If two characters match, we assign a value to index (i,j) by incrementing the previously calculated LCS that is present at index (i-1,j-1) (from the previous line).

Step 3) If it does not match, we take the maximum LCS of two adjacent indices. And thus we need to fill all the values in the 2D array.

Step 4) Finally, we return the value of the last cell of the two-dimensional array.

In essence, all values in a two-dimensional array contain the length of common subsequences. Among them, the last cell contains the length of the longest common subsequence.

Implementation in C++

Total

- The DP method has a lower time complexity; it is O(mn), where m and n are the length of the input string or array.
- DP is a faster approach than the recursive one, with a time complexity of $O(n*2)^{m}$**).**
- Dynamic programming (DP) is not memory optimized. We used a two-dimensional array of length m*n. So, the space complexity is (m*n).
- Recursive method: in the worst case, the maximum memory it occupies will be m+n, which is essentially the total length of the input string.
- Today's modern computer has enough memory to process this amount of memory.

There is an ongoing debate among programmers about whether it is necessary to know "algosics" for real work, or whether it is just some strange ritual for passing the interview funnel in a company a la FAANG (MANGA). Here in Karuna, in different teams, there are different opinions on this matter. For example, as a team leader of the Go team, I think that it would definitely not hurt to know some basic knowledge, but the main thing is that the person is good.

Opinions may differ, but I know one thing for sure: solving riddles can be very interesting. I once solved problems on hackerrank out of curiosity, and if to solve simple problems it is enough to guess how to sort the data or build a map (you don’t even need to know anything special), then for some it is quite problematic to come up with a solution.

One such problem is finding the longest common subsequence. A similar algorithm is used in real life, in programs like diff. I'll say right away: I couldn't solve the problem on my own in a reasonable amount of time (i.e. until I got tired of solving it) and looked up the algorithm on Wikipedia.

But never mind the algorithm, I became terribly curious about how the hell I had to reason to come to this decision myself. In the end, I decided to post these thoughts as an article on Habr.

Disclaimer: I'm definitely not an Olympiad participant or an algorithms guru, I'm just curious.

Task

On hackerrank the task sounded something like this:

Common child.

A string is called a child of another string if it can be formed by removing 0 or more characters from that string. The characters cannot be mixed.

Given two strings of equal length, find the length of the largest string that can be constructed so that it is a "child" of both strings.

For example, for the strings BCDEKF and CDEGKB, the greatest common child will be CDEK, i.e. 4 common symbols without changing their order.

In short, you need to write a body for such a function.

```
// only need to find the length
funccommonChild(s1string, s2string)int{
}
```

Less formally, it's like the result of looking at a diff of two files. A file is a sequence of elements (like lines of code), and diff looks for as many of the same elements as possible (the longest common subsequence), and then shows the rest as differences.

Head-on attempt

Books and articles about interview preparation often advise trying to solve the problem head-on first. So that it gives the correct answer at least for small input data, and then try to optimize the speed. It seems like this should show the interviewer that you understand the task at all and can at least do basic programming.

Here is the dumbest solution in the forehead: first find the beginning of the common child (more precisely - go through all the options where this beginning can be), and then, starting from this point, reduce the problem to the previous one. That is, recursion. Here it should be noted that the stack in Go is dynamic, so you usually don't have to worry about too much nesting of recursion, in other languages it could look more complicated.

```
funccommonChild(s1string, s2string)int{
varmaxLenint

fori := 0; i < len(s1); i++ {
forj := 0; j < len(s2); j++ {
ifs1[i] == s2[j] {
                next := commonChild(s1[i+1:], s2[j+1:])
```

```
                l := 1 + next
if l > maxLen {
                    maxLen = l
                }
            }
        }
    }

return maxLen
}
```

The solution is simple, works correctly on small examples, but already on a few miserable dozens of characters it freezes completely - the algorithm is not optimal (which is not surprising). If I understand correctly, it is O(2^n).

By the way, I asked ChatGPT what the algorithmic complexity is here, and without blinking an eye, he said with a smart look that it is O(n^2), because there is a cycle within a cycle. In short, the robot did not notice the recursion.

We are trying to optimize

If you solve a certain number of problems on Hackerrank, you will understand that this problem is often solved by storing intermediate calculations somewhere, i.e., roughly speaking, using some kind of cache. This is not rocket science: even if you have been sawing PHP programs all your life, you have definitely cached something at least a couple of times for speed. By the way, this is the main principle of dynamic programming - solve a subproblem once, remember the intermediate result, and then use it many times without unnecessary calculations.

As a result, we can come to the following decision:

```
// here the structure that stores the lengths of the remaining lines, i.e. essentially the position, is used as a cache key.
type Key struct {
    len1 int
    len2 int
}
```

```
varcache =map[Key]int{}

funccommonChild(s1string, s2string)int{
   key := Key{len1: len(s1), len2: len(s2)}

   cached, exists := cache[key]
ifexists {
returncached
   }

varmaxLenint

fori := 0; i < len(s1); i++ {
forj := 0; j < len(s2); j++ {
ifs1[i] == s2[j] {
            next := commonChild(s1[i+1:], s2[j+1:])

            l := 1 + next
ifl > maxLen {
                maxLen = l
            }
        }
    }
   }

   cache[key] = maxLen
returnmaxLen
}
```

This solution works faster, but after some length it still freezes. That is, you need to cache it somehow smarter. But how?

Visualization

Smart books say that successful visualization is half the solution. But how do you visualize it, comparing each symbol with each other? When I was looking for a solution myself, I tried drawing two sequences side by side and connecting them with a million arrows, but it didn't lead to anything.

In fact, I am especially offended that the correct option did not come to mind during the solution process, because several years ago I myself wrote on Habr about visualizing SQL joins (Part 1,part 2). And in the second part I just drew the union of all the elements with all in the form of a rectangle (matrix).

In general, if you need to compare everything with everyone, draw a matrix. For example, for the strings BCDEKF and CDEGKB, draw the correct answer, i.e. the largest sequence. In this simple case, it is obvious:

```
  | B | C | D | E | K | F |
--|---|---|---|---|---|---|
C | | 1 | | | | |
--|---|---|---|---|---|---|
D | | | 2 | | | |
--|---|---|---|---|---|---|
E | | | | 3 | | |
--|---|---|---|---|---|---|
G | | | | | | |
--|---|---|---|---|---|---|
K | | | | | 4 | |
--|---|---|---|---|---|---|
B | | | | | | |
--------------------------
```

It is also useful to draw some less straightforward cases, such as the strings EFABCD and ABCDEF. There are two common subsequences: EF (marked with an asterisk) and ABCD, the second being larger.

```
  | E | F | A | B | C | D |
--|---|---|---|---|---|---|
A | | | 1 | | | |
--|---|---|---|---|---|---|
B | | | | 2 | | |
--|---|---|---|---|---|---|
C | | | | | 3 | |
--|---|---|---|---|---|---|
D | | | | | | 4 |
--|---|---|---|---|---|---|
E | 1*| | | | | |
--|---|---|---|---|---|---|
F | | 2*| | | | |
```

From these two pictures it is already clear in principle that the sequence always increases by shifting one (or more) cell to the right and one (or more) cell down. Which, if you think about it, is clear anyway, but with a picture it is much clearer.

Okay, this is also a simple example, we also need a case with repeating letters.

BAABA and BABAB

This example is good because there are a lot of combinations for such short strings, and there can be several subsequences of maximum length, not only BABA (the answer that first comes to mind), but also BAAB, for example.

```
  | B | A | A | B | A |
--|---|---|---|---|---|
B | * | | | * | |
--|---|---|---|---|---|
A | | * | * | | * |
--|---|---|---|---|---|
B | * | | | * | |
--|---|---|---|---|---|
A | | * | * | | * |
--|---|---|---|---|---|
B | * | | | * | |
----------------------
```

I marked the letter matches with asterisks, and it is already clear, in principle, that, if we take into account that the sequences are built with a mandatory shift to the right and down, then our BABA is visible to the naked eye. Other, smaller ones are also visible: B, BA, BAB, etc., which we discard.

Instead of asterisks, you can write some number that increases diagonally, and by the largest number we will eventually find the largest subsequence.

But there is a nuance: of course, the sequence does not always go strictly diagonally (one cell to the right and one down). Sometimes it shifts more, passes

through emptiness (i.e. through non-matching symbols). These non-matching symbols themselves do not change the length of the general subsequence, but the path passes through them. Therefore, for each empty cell, you can also write a number and an arrow that will point to the most advantageous path (to the maximum of the numbers from above or to the left).

As a result, we can come to the classic picture (taken from Wikipedia)

		A	B	C	D
	0	0	0	0	0
D	0	← 0	← 0	← 0	↖ 1
C	0	← 0	← 0	↖ 1	← 1
D	0	← 0	← 0	↑ 1	↖ 2
A	0	↖ 1	← 1	← 1	↑ 2

and the algorithm:

The table is filled in row by row, starting from the upper left corner DA. It is filled in as follows: if the symbols match (for example CC, DD, AA), then the number from the previous element along the diagonal (the nearest cell on the left from the top) is taken, with the addition of one. If they do not match, then the maximum

number from two options is simply copied: the adjacent left cell or the adjacent upper one. The arrow in the figure indicates the field from which the value was taken.

As a result, in the lower right corner there will be an answer: the maximum length of the common child. And by the arrows you can go to the beginning and restore the subsequence itself, not only the length (writing out the matching symbols).

```
funccommonChild(s1string, s2string)int{
   N := len(s1)
   matrix := make([][]int, N+1)
fori :=rangematrix {
      matrix[i] = make([]int, N+1)
   }

fori := 0; i < N; i++ {
forj := 0; j < N; j++ {
ifs1[i] == s2[j] {
            matrix[i+1][j+1] = matrix[i][j] + 1
         }else{
            max := matrix[i][j+1]
ifmatrix[i+1][j] > max {
               max = matrix[i+1][j]
            }

            matrix[i+1][j+1] = max
         }
      }
   }
returnmatrix[N][N]
}
```

In the problem statement on hackerrank, the lengths of the lines are the same, but there is no particular difference, it can be easily reworked for a more general case.

The complexity of such an algorithm is obviously O(N^2), because there is no recursion, nothing. Just a cycle within a cycle.

Given two strings, **s1** and **s2**, the task is to find the length of the Longest Common Subsequence. If there is no **common subsequence**, return 0.

A **subsequence** is a string generated from the original string by deleting 0 or more characters and without changing the relative order of the remaining characters. For example , subsequences of "ABC" are "", "A", "B", "C", "AB", "AC", "BC" and "ABC".

In general a string of length **n** has 2^n subsequences.

Examples:

Input: *s1 = "ABC", s2 = "ACD"*
Output: *2*
Explanation: *The longest subsequence which is present in both strings is "AC".*

Input: *s1 = "AGGTAB", s2 = "GXTXAYB"*
Output: *4*
Explanation: *The longest common subsequence is "GTAB".*

Input: *s1 = "ABC", s2 = "CBA"*
Output: *1*
Explanation: *There are three longest common subsequences of length 1, "A", "B" and "C".*

[Naive Approach] Using Recursion – O(2 ^ min(m, n)) Time and O(min(m, n)) Space

*The idea is to compare the last characters of **s1** and **s2**. While comparing the strings **s1** and **s2** two cases arise:*

1. ***Match** : Make the recursion call for the remaining strings (strings of lengths **m-1** and **n-1**) and add 1 to result.*
2. ***Do not Match** : Make two recursive calls. First for lengths **m-1** and **n**, and second for **m** and **n-1**. Take the maximum of two results.*

***Base case** : If any of the strings become empty, we return 0.*

For example, consider the input strings s1 = "ABX" and s2 = "ACX".

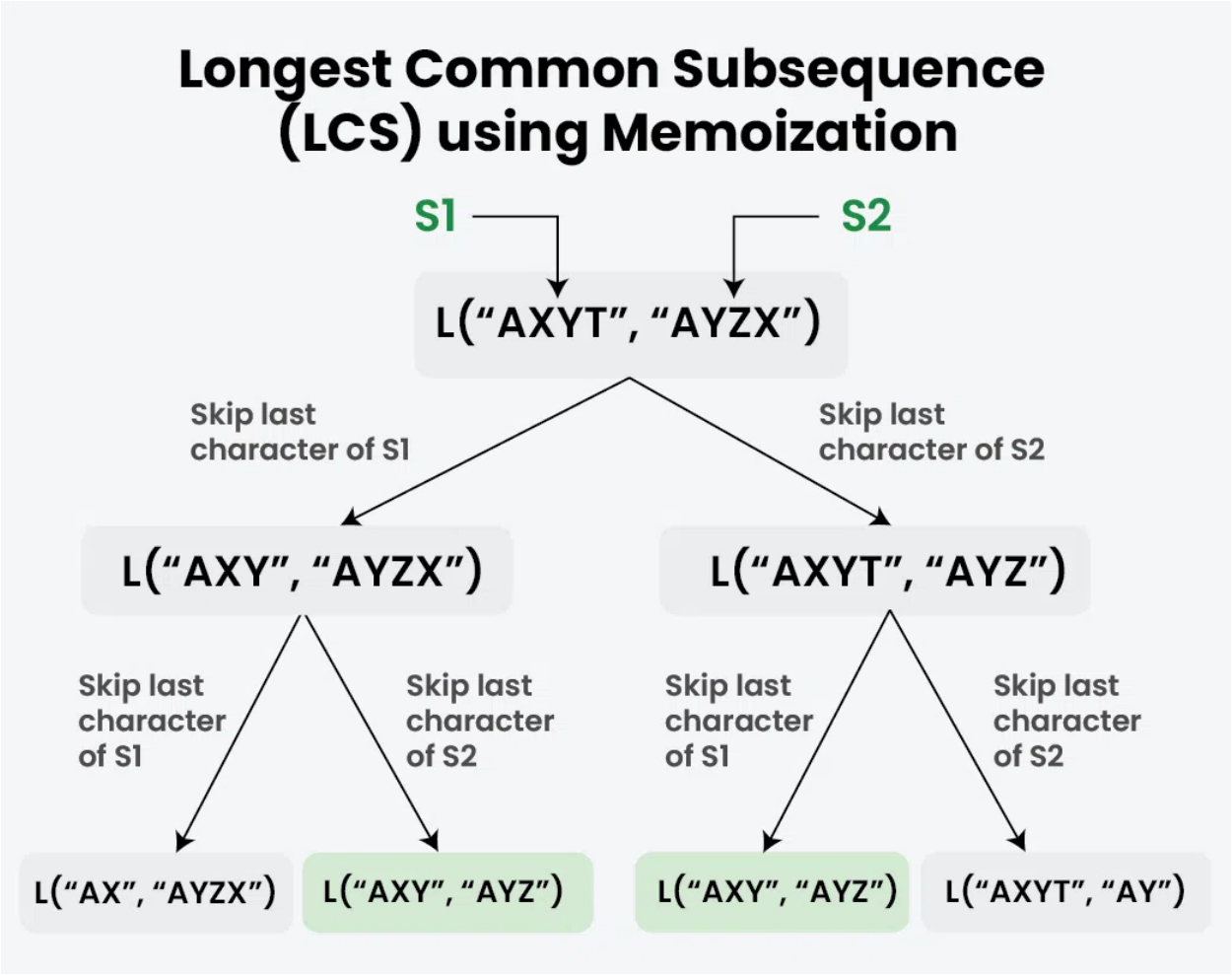
LCS("ABX", "ACX") = 1 + LCS("AB", "AC") [Last Characters Match]

LCS("AB", "AC") = max(LCS("A", "AC") , LCS("AB", "A")) [Last Characters Do Not Match]

LCS("A", "AC") = max(LCS("", "AC") , LCS("A", "A")) = max(0, 1 + LCS("", "")) = 1

LCS("AB", "A") = max(LCS("A", "A") , LCS("AB", "")) = max(1 + LCS("", "", 0)) = 1

*So overall result is 1 + 1 = **2***


Longest Common Subsequence (LCS) using Memoization
S1
S2
L("AXYT", "AYZX")
Skip last character of S1
Skip last character of S2
L("AXY", "AYZX")
L("AXYT", "AYZ")
Skip last character of S1
Skip last character of S2
Skip last character of S1
Skip last character of S2
L("AX", "AYZX")
L("AXY", "AYZ")
L("AXY", "AYZ")
L("AXYT", "AY")

Implementation of the algorithm in the languageC++

```
#include <iostream>
#include <vector>
using namespace std;

// Returns length of LCS for s1[0..m-1], s2[0..n-1]
int lcs(string &s1, string &s2, int m, int n, vector<vector<int>> &memo) {

    // Base Case
    if (m == 0 || n == 0)
        return 0;

    // Already exists in the memo table
    if (memo[m][n] != -1)
        return memo[m][n];

    // Match
    if (s1[m - 1] == s2[n - 1])
        return memo[m][n] = 1 + lcs(s1, s2, m - 1, n - 1, memo);

    // Do not match
    return memo[m][n] = max(lcs(s1, s2, m, n - 1, memo), lcs(s1, s2, m - 1, n,
memo));
}

int main() {
    string s1 = "AGGTAB";
    string s2 = "GXTXAYB";

    int m = s1.length();
    int n = s2.length();
    vector<vector<int>> memo(m + 1, vector<int>(n + 1, -1));
    cout << lcs(s1, s2, m, n, memo) << endl;
    return 0;
}
```

Implementation of the algorithm in the languageC

```c
#include <stdio.h>
#include <string.h>

// Define a maximum size for the strings
#define MAX 1000

// Function to find the maximum of two integers
int max(int a, int b) {
    return (a > b) ? a : b;
}

// Returns length of LCS for s1[0..m-1], s2[0..n-1]
int lcs(const char *s1, const char *s2, int m, int n, int memo[MAX][MAX]) {

    // Base Case
    if (m == 0 || n == 0) {
        return 0;
    }

    // Already exists in the memo table
    if (memo[m][n] != -1) {
        return memo[m][n];
    }

    // Match
    if (s1[m - 1] == s2[n - 1]) {
        return memo[m][n] = 1 + lcs(s1, s2, m - 1, n - 1, memo);
    }

    // Do not match
    return memo[m][n] = max(lcs(s1, s2, m, n - 1, memo), lcs(s1, s2, m - 1, n,
memo));
}
int main() {
    const char *s1 = "AGGTAB";
    const char *s2 = "GXTXAYB";

    int m = strlen(s1);
    int n = strlen(s2);

    // Create memo table with fixed size
    int memo[MAX][MAX];
    for (int i = 0; i <= m; i++) {
        for (int j = 0; j <= n; j++) {
            // Initialize memo table with -1
            memo[i][j] = -1;
        }
    }

    printf("%d\n", lcs(s1, s2, m, n, memo));

    return 0;
}
```

Implementation of the algorithm in the languageC#

```
// C# implementation of Top-Down DP of LCS problem
using System;
class GfG {

    // Returns length of LCS for s1[0..m-1], s2[0..n-1]
    static int lcs(string s1, string s2, int m,
                           int n, int[, ] memo) {
        // Base Case
        if (m == 0 || n == 0)
            return 0;

        // Already exists in the memo table
        if (memo[m, n] != -1)
            return memo[m, n];

        // Match
        if (s1[m - 1] == s2[n - 1]) {
            return memo[m, n]
                = 1 + lcs(s1, s2, m - 1, n - 1, memo);
        }

        // Do not match
        return memo[m, n]
            = Math.Max(lcs(s1, s2, m, n - 1, memo),
                       lcs(s1, s2, m - 1, n, memo));
    }

    public static void Main() {
        string s1 = "AGGTAB";
        string s2 = "GXTXAYB";

        int m = s1.Length;
        int n = s2.Length;
        int[, ] memo = new int[m + 1, n + 1];

        // Initialize memo array with -1
        for (int i = 0; i <= m; i++) {
            for (int j = 0; j <= n; j++) {
                memo[i, j] = -1;
            }
        }

        Console.WriteLine(lcs(s1, s2, m, n, memo));
    }
}
```

Implementation of the algorithm in the languageJava

```java
import java.util.Arrays;

class GfG {

    // Returns length of LCS for s1[0..m-1], s2[0..n-1]
    static int lcs(String s1, String s2, int m, int n,
                   int[][] memo) {
        // Base Case
        if (m == 0 || n == 0)
            return 0;

        // Already exists in the memo table
        if (memo[m][n] != -1)
            return memo[m][n];

        // Match
        if (s1.charAt(m - 1) == s2.charAt(n - 1)) {
            return memo[m][n]
                = 1 + lcs(s1, s2, m - 1, n - 1, memo);
        }

        // Do not match
        return memo[m][n]
            = Math.max(lcs(s1, s2, m, n - 1, memo),
                       lcs(s1, s2, m - 1, n, memo));
    }

    public static void main(String[] args) {
        String s1 = "AGGTAB";
        String s2 = "GXTXAYB";

        int m = s1.length();
        int n = s2.length();
        int[][] memo = new int[m + 1][n + 1];

        // Initialize the memo table with -1
        for (int i = 0; i <= m; i++) {
            Arrays.fill(memo[i], -1);
        }

        System.out.println(lcs(s1, s2, m, n, memo));
    }
}
```

Python

```python
# Python implementation of Top-Down DP of LCS problem

# Returns length of LCS for s1[0..m-1], s2[0..n-1]
def lcs(s1, s2, m, n, memo):
    # Base Case
    if m == 0 or n == 0:
        return 0

    # Already exists in the memo table
    if memo[m][n] != -1:
        return memo[m][n]

    # Match
    if s1[m - 1] == s2[n - 1]:
        memo[m][n] = 1 + lcs(s1, s2, m - 1, n - 1, memo)
        return memo[m][n]

    # Do not match
    memo[m][n] = max(lcs(s1, s2, m, n - 1, memo),
                     lcs(s1, s2, m - 1, n, memo))
    return memo[m][n]

if __name__ == "__main__":
    s1 = "AGGTAB"
    s2 = "GXTXAYB"

    m = len(s1)
    n = len(s2)
    memo = [[-1 for _ in range(n + 1)] for _ in range(m + 1)]
    print(lcs(s1, s2, m, n, memo))
```

Implementation of the algorithm in the languageJavaScript

```javascript
// A Top-Down DP implementation of LCS problem

// Returns length of LCS for s1[0..m-1], s2[0..n-1]
function lcs(s1, s2, m, n, memo) {
    // Base Case
    if (m === 0 || n === 0)
        return 0;

    // Already exists in the memo table
    if (memo[m][n] !== -1)
        return memo[m][n];

    // Match
    if (s1[m - 1] === s2[n - 1]) {
        memo[m][n] = 1 + lcs(s1, s2, m - 1, n - 1, memo);
        return memo[m][n];
    }

    // Do not match
    memo[m][n] = Math.max(lcs(s1, s2, m, n - 1, memo),
                          lcs(s1, s2, m - 1, n, memo));
    return memo[m][n];
}

// driver code
const s1 = "AGGTAB";
const s2 = "GS1TS1AS2B";

const m = s1.length;
const n = s2.length;
const memo = Array.from({length : m + 1},
                        () => Array(n + 1).fill(-1));

console.log(lcs(s1, s2, m, n, memo));
```

Summary

I was quite capable of solving this problem on my own, there is nothing complicated about it if I had drawn the right picture right away. But how to learn to guess such things, especially with the timer on, is a mystery to me.

If you could recommend in the comments some good article or book on how to think in order to easily come up with solutions to algorithmic problems, I would be very grateful.