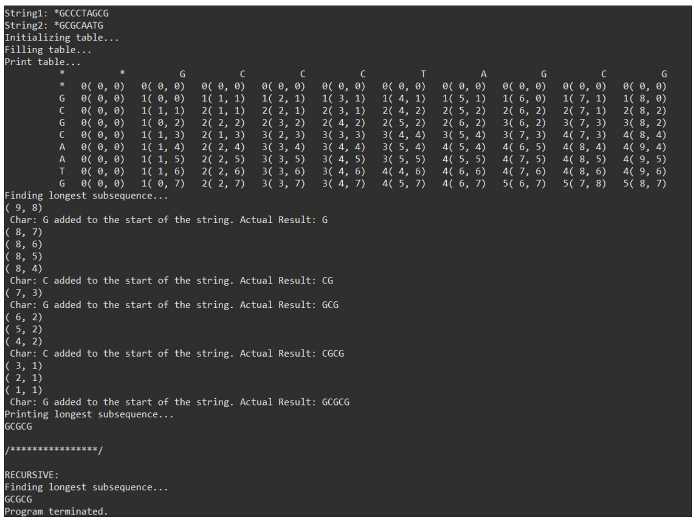
Algorithmics	Student information	Date	Number of session
	UO: 276903	31/3/21	5
	Surname: Garriga Suárez		
	Name: Carlos		

Escuela de Ingeniería Informática
Universidad de Oviedo

Universidad de Oviedo
Universidá d'Uviéu
University of Oviedo

Activity 1. Validation Results

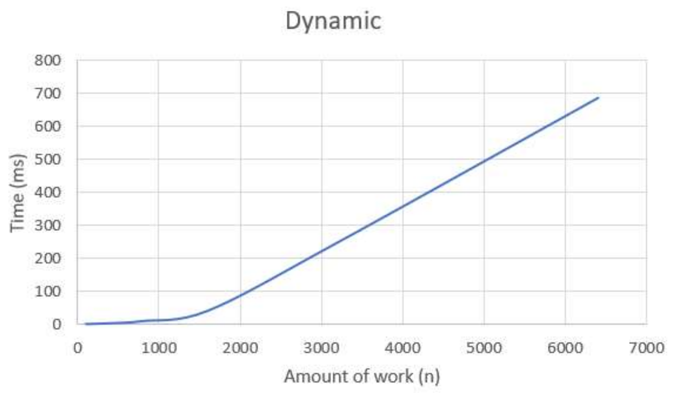


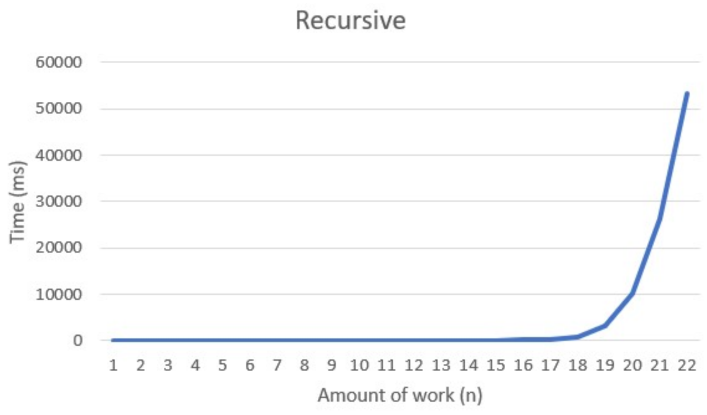
```
String1: *GCCCTAGCG
String2: *GCGCAATG
Initializing table...
Filling table...
Print table...
          *          *          G          C          C          C          T          A          G          C          G
          *   0( 0, 0)   0( 0, 0)   0( 0, 0)   0( 0, 0)   0( 0, 0)   0( 0, 0)   0( 0, 0)   0( 0, 0)   0( 0, 0)   0( 0, 0)
          G   0( 0, 0)   1( 0, 0)   1( 1, 1)   1( 2, 1)   1( 3, 1)   1( 4, 1)   1( 5, 1)   1( 6, 0)   1( 7, 1)   1( 8, 0)
          C   0( 0, 0)   1( 1, 1)   2( 1, 1)   2( 2, 1)   2( 3, 1)   2( 4, 2)   2( 5, 2)   2( 6, 2)   2( 7, 1)   2( 8, 2)
          G   0( 0, 0)   1( 0, 2)   2( 2, 2)   2( 3, 2)   2( 4, 2)   2( 5, 2)   2( 6, 2)   3( 6, 2)   3( 7, 3)   3( 8, 2)
          C   0( 0, 0)   1( 1, 3)   2( 1, 3)   3( 2, 3)   3( 3, 3)   3( 4, 4)   3( 5, 4)   3( 7, 3)   4( 7, 3)   4( 8, 4)
          A   0( 0, 0)   1( 1, 4)   2( 2, 4)   3( 3, 4)   3( 4, 4)   3( 5, 4)   4( 5, 4)   4( 6, 5)   4( 8, 4)   4( 9, 4)
          A   0( 0, 0)   1( 1, 5)   2( 2, 5)   3( 3, 5)   3( 4, 5)   3( 5, 5)   4( 5, 5)   4( 7, 5)   4( 8, 5)   4( 9, 5)
          T   0( 0, 0)   1( 1, 6)   2( 2, 6)   3( 3, 6)   3( 4, 6)   4( 4, 6)   4( 6, 6)   4( 7, 6)   4( 8, 6)   4( 9, 6)
          G   0( 0, 0)   1( 0, 7)   2( 2, 7)   3( 3, 7)   3( 4, 7)   4( 5, 7)   4( 6, 7)   5( 6, 7)   5( 7, 8)   5( 8, 7)
Finding longest subsequence...
( 9, 8)
 Char: G added to the start of the string. Actual Result: G
( 8, 7)
( 8, 6)
( 8, 5)
( 8, 4)
 Char: C added to the start of the string. Actual Result: CG
( 7, 3)
 Char: G added to the start of the string. Actual Result: GCG
( 6, 2)
( 5, 2)
( 4, 2)
 Char: C added to the start of the string. Actual Result: CGCG
( 3, 1)
( 2, 1)
( 1, 1)
 Char: G added to the start of the string. Actual Result: GCGCG
Printing longest subsequence...
GCGCG

/****************/

RECURSIVE:
Finding longest subsequence...
GCGCG
Program terminated.
```

Activity 2. Experimental Time Measurements





Algorithmics	Student information	Date	Number of session
	UO: 276903	31/3/21	5
	Surname: Garriga Suárez		
	Name: Carlos		

N	t_dynamic(ms)
100	0,8
200	1,7
400	3,5
800	9,9
1600	40,9
3200	248,9
6400	687,9

N	t_recursive(ms)
2	0,1
4	0,1
6	0,3
8	0,7
10	2,5
12	10,4
14	37,3
16	220,9
18	792,6
20	10249,6
22	53180,5

As we can observe in the graphs, we can see that for bigger values of n, dynamic programming is a much better option considering time. Recursive approach grows very fast for small values of n.

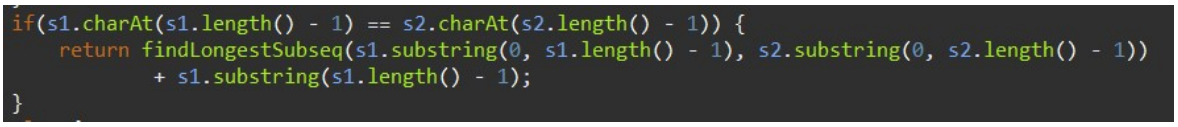
Activity 3. Answer to questions

- **Determine theoretically complexities (time, memory space and waste of stack) for both implementations, recursive (approximated) and using programming dynamic.**

The theoretical complexity for the recursive approach is something between O(n) and O(2^n).

We must distinguish two cases in the method:

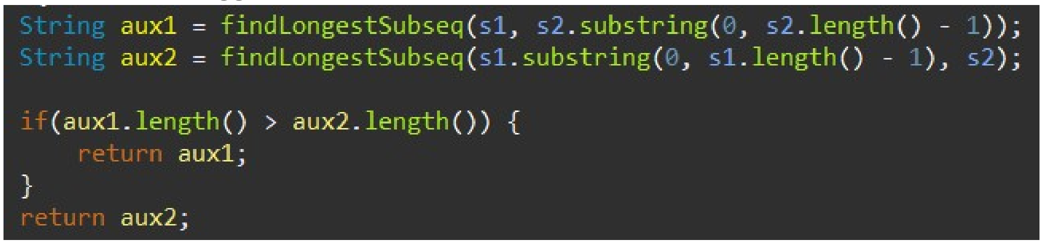
- When the last character of both strings is the same: what we do is to return a call to the method with both strings without considering the last character plus that character.



```
if(s1.charAt(s1.length() - 1) == s2.charAt(s2.length() - 1)) {
    return findLongestSubseq(s1.substring(0, s1.length() - 1), s2.substring(0, s2.length() - 1))
            + s1.substring(s1.length() - 1);
}
```

This is divide and conqueror by subtraction and we have a = 1, as we are calling the method once; b = 1 as we are reducing the size of the problem in one unit and k=0. In the end we have that the complexity is O(n).

- When the last character of both strings is not the same: what we do is to create two new strings by calling the method first with the first string and the second without the last character and then calling the method with the first string without the last character and the second string. Then we return the string which is bigger.



```
String aux1 = findLongestSubseq(s1, s2.substring(0, s2.length() - 1));
String aux2 = findLongestSubseq(s1.substring(0, s1.length() - 1), s2);

if(aux1.length() > aux2.length()) {
    return aux1;
}
return aux2;
```

Putting all this together we have that a = 2 as we need two calls to the method, b = 1 as we are reducing the problem in one unit and k = 0.
Then the complexity is something between O(n) and O(2^n).

The theoretical complexity for the dynamic programming approach would be O(m*n) as you are filling a matrix with two nested loops. Then the algorithm to find the LCS have a complexity at most the smallest of both strings. With this I mean that if a have a string m and another n, and m > n then the complexity of finding an LCS will be at most O(n).

- **Compute theoretical times and compare them with the experimental measurements.**

With dynamic programming the theoretical times would be calculated as follows. The time complexity is O(m*n). To facilitate calculus, I am going to assume that m = n. This led us to quadratic complexity.

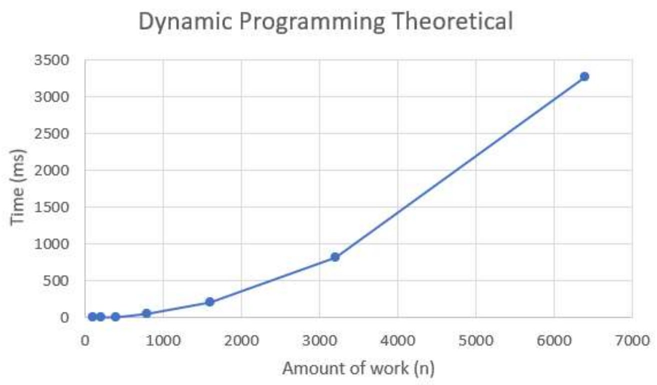
N2 = K * N1 ➔ t2 = (f(n2) / f(n1)) * t1 = ($(n2)^2$ / $(n1)^2$) * t1 = k^2 * t1

The obtained results are these:

N	Theoretical Time
100	0,8
200	3,2
400	12,8
800	51,2
1600	204,8

Algorithmics	Student information	Date	Number of session
	UO: 276903	31/3/21	5
	Surname: Garriga Suárez		
	Name: Carlos		

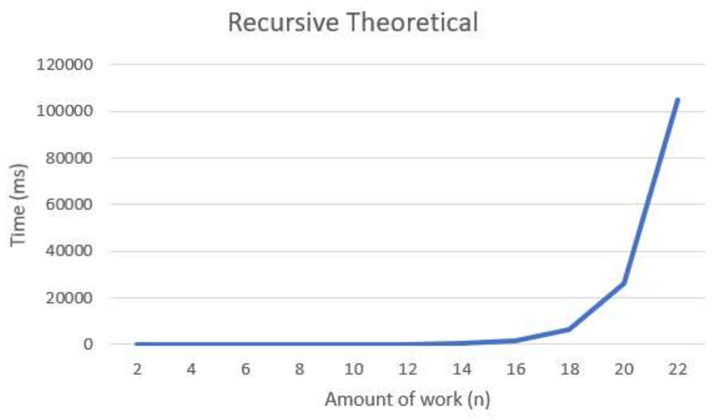
3200	819,2
6400	3276,8



With recursive approach we have a complexity between O(n) and O(2^n). Assuming O(2^n)

N2 = K * N1 ➔ t2 = (f(n2) / f(n1)) * t1 = (2^{n2}/ 2^{n1}) * t1 = 2^{n2-n1} * t1

N	Theoretical Time
2	0,1
4	0,4
6	1,6
8	6,4
10	25,6
12	102,4
14	409,6
16	1638,4
18	6553,6
20	26214,4
22	104857,6

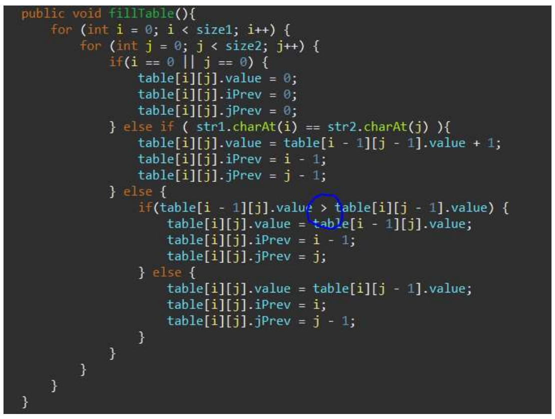


Algorithmics	Student information	Date	Number of session
	UO: 276903	31/3/21	5
	Surname: Garriga Suárez		
	Name: Carlos		

- **Why large sequences cannot be processed with the recursive implementation? Explain why dynamic programming implementation raises and exception for large sequences.**

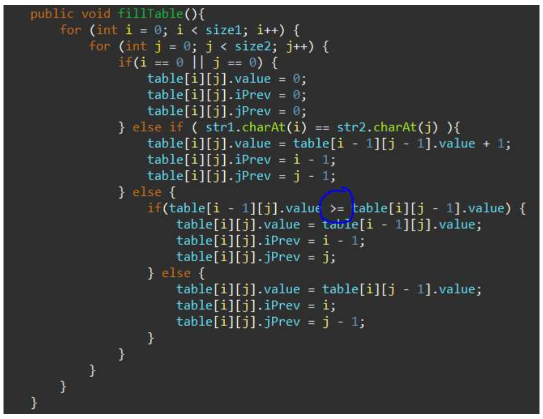
Large sequences cannot be processed with recursive implementation because the stack is not big enough to store all the recursive calls to the method.
Dynamic programming raises an exception with big values as when you are initializing the matrix there is not enough space to store all the values on the table.

- **The amount of possible LCS can be more than one, e. g. GCCCTAGCG and GCGCAATG has two GCGCG and GCCAG. Find the code section that determines which subsequence is chosen, modify this code to verify that both solutions can be achieved.**



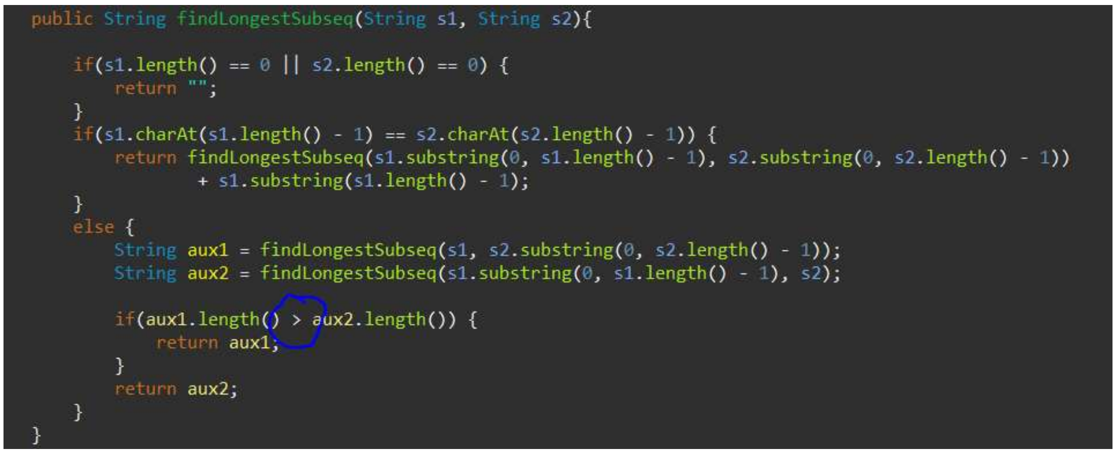
```
public void fillTable(){
    for (int i = 0; i < size1; i++) {
        for (int j = 0; j < size2; j++) {
            if(i == 0 || j == 0) {
                table[i][j].value = 0;
                table[i][j].iPrev = 0;
                table[i][j].jPrev = 0;
            } else if ( str1.charAt(i) == str2.charAt(j) ){
                table[i][j].value = table[i - 1][j - 1].value + 1;
                table[i][j].iPrev = i - 1;
                table[i][j].jPrev = j - 1;
            } else {
                if(table[i - 1][j].value > table[i][j - 1].value) {
                    table[i][j].value = table[i - 1][j].value;
                    table[i][j].iPrev = i - 1;
                    table[i][j].jPrev = j;
                } else {
                    table[i][j].value = table[i][j - 1].value;
                    table[i][j].iPrev = i;
                    table[i][j].jPrev = j - 1;
                }
            }
        }
    }
}
```

➔



```
public void fillTable(){
    for (int i = 0; i < size1; i++) {
        for (int j = 0; j < size2; j++) {
            if(i == 0 || j == 0) {
                table[i][j].value = 0;
                table[i][j].iPrev = 0;
                table[i][j].jPrev = 0;
            } else if ( str1.charAt(i) == str2.charAt(j) ){
                table[i][j].value = table[i - 1][j - 1].value + 1;
                table[i][j].iPrev = i - 1;
                table[i][j].jPrev = j - 1;
            } else {
                if(table[i - 1][j].value >= table[i][j - 1].value) {
                    table[i][j].value = table[i - 1][j].value;
                    table[i][j].iPrev = i - 1;
                    table[i][j].jPrev = j;
                } else {
                    table[i][j].value = table[i][j - 1].value;
                    table[i][j].iPrev = i;
                    table[i][j].jPrev = j - 1;
                }
            }
        }
    }
}
```

The code for the dynamic approach is in the method that initializes the table and the only thing you need to change is just a condition. It happens the same as I will explain for the recursive approach.



```
public String findLongestSubseq(String s1, String s2){

    if(s1.length() == 0 || s2.length() == 0) {
        return "";
    }
    if(s1.charAt(s1.length() - 1) == s2.charAt(s2.length() - 1)) {
        return findLongestSubseq(s1.substring(0, s1.length() - 1), s2.substring(0, s2.length() - 1))
                + s1.substring(s1.length() - 1);
    }
    else {
        String aux1 = findLongestSubseq(s1, s2.substring(0, s2.length() - 1));
        String aux2 = findLongestSubseq(s1.substring(0, s1.length() - 1), s2);

        if(aux1.length() > aux2.length()) {
            return aux1;
        }
        return aux2;
    }
}
```

The code for recursive approach is inside the method itself and it checks if one string is greater than the other. In the case they are equal the method will choose the as default aux2, but both are alright.