Algorithmics	Student information	Date	Number of session
	UO: 270534		
	Surname: Iglesias Préstamo		
	Name: Ángel		

Escuela de Ingeniería Informática
Universidad de Oviedo

Universidad de Oviedo
Universidá d'Uviéu
University of Oviedo

Activity 1. [Validation Results]

First, I have to say that in my implementation, I found another possible sequence that is also valid. However, I know where we can decide the actual sequence to be print. The dynamic programming implementation leads us to the following string: **GCGCG**. Whereas the recursive implementation prints the following string: **GCCTG**. Which again is also valid and has the same length. This is due to the way I implement each algorithm and will be explained in more detail in the questions part.

Activity 2. [Experimental time measurements]

n	*t_dynamic*
100	1.8
200	2.7
400	3.8
800	9.0
1,600	37.1
3,200	122.3
6,400	405.9
12,800	1,523.0
	$O(m \cdot n)$

n	*t_recursive*
1	0.0
2	0.0
3	0.1
4	0.2
5	0.3
6	0.4
7	0.5
8	0.9
9	2.1
10	3.0
11	8.5
12	14.4
13	30.9
14	63.5
15	148.5
16	452.7
17	1,056.5
18	2,117.5
19	4,194.8
20	10,175.0
	$O(2^n)$

Activity 3. [Questions]

a. **Question 1.** *Determine theoretically complexities (time, memory space and waste of stack) for both implementations, recursive (approximated) and using programming dynamic.*

- In the case of programming dynamic, we may notice that we just have two loops, first we iterate over the rows: outer loop, and then we iterate over the columns. As the table may not be even, we will say that the complexity will not be $O(n^2)$, because they are nested loops, but $O(m \cdot n)$, where m is the number of rows, and n the number of columns.

- In the case of the recursive implementation is way easier finding the theoretical time complexities. We just need to apply the formulas studied in the D&C part. In the worst case: $a = 2,\ \ b = 1$ and $k = 0$; as the time complexity of the rest of the code is constant and everything is executed sequentially. More in more, we apply a D&C by subtraction technique, this means, we must apply the second formula: $O(n) = O\left(a^{\frac{n}{b}}\right) = O(2^n)$, as a is greater than 1.

b. **Question 2.** *Compute theoretical times and compare them with the experimental measurements.*

t_dynamic	*t_recursive*
$t_1 = 122.3;\ n_1 = 3{,}200\ and\ n_2 = 6{,}400$ $t_3 = 405.9;\ n_3 = 6{,}400\ and\ n_4 = 12{,}800$ • $t_2 = \left(\frac{6{,}400}{3{,}200}\right)^2 \cdot 122.3 = 489.2$ • $t_4 = \left(\frac{12{,}800}{6{,}400}\right)^2 \cdot 405.9 = 1{,}623.6$	$t_1 = 2{,}117.5;\ n_1 = 18\ and\ n_2 = 20$ $t_3 = 63.5;\ n_3 = 14\ and\ n_4 = 15$ • $t_2 = 2^{20-18} \cdot 2{,}117.5 = 2^2 \cdot 2{,}117.5 = 8{,}470$ • $t_4 = 2^{15-14} \cdot 63.5 = 2 \cdot 63.5 = 127$

Values	*t_dynamic*	*t_recursive*
$Actual_1$	405.9	10,175.0
$Expected_1$	489.2	8,470
$Actual_2$	1,523.0	148.5
$Expected_2$	1,623.6	127

c. **Question 3.** *Why large sequences cannot be processed with the recursive implementation? Explain why dynamic programming implementation raises an exception for large sequences.*

d. **Question 4.** *Find the code section that determines which subsequence is chosen, modify this code to verify that both solutions can be achieved.*

i. In the case of the recursive algorithm, if we change the following snippet:

```
String first = findLongestSubseq(s1, s2.substring(0, s2Length));
String second = findLongestSubseq(s1.substring(0, s1Length), s2);
```

And instead of such, we just write the next one:

```
String first = findLongestSubseq(s1.substring(0, s1Length), s2);
String second = findLongestSubseq(s1, s2.substring(0, s2Length));
```

We will get the exact same string as in the dynamic solution.

ii. In the case of the dynamic part, notice that not all number will hold, this is caused because there are some decisions that may be valid according to more than one criterion. This is, if the letter of both strings is the same, but also other surrounding position has the maximum value. What's interesting is that the solution is still valid.