


Algorithmics	Student information	Date	Number of session
	UO: 276244	28/03/21	5
	Surname: Beltran Diaz	 Escuela de Ingeniería Informática Universidad de Oviedo	
	Name: Martin		



Activity 1. VALIDATION RESULTS

For the sequences GCCCTAGC - GCGCAATG I obtain two different results from the dynamic version and the recursive one, these are: GCGCG for dynamic and GCCTG for the recursive one. As you can see both have the same length and are correct answers.

Let's check another case: GCATGCAT – GAATTCAG. For the dynamic version we obtain solution GATCA, and for the recursive version we obtain the same solution.

We both examples I want to show that apart from the internal work of the two algorithms, there could exist more than one valid solution for the same LCS problem.

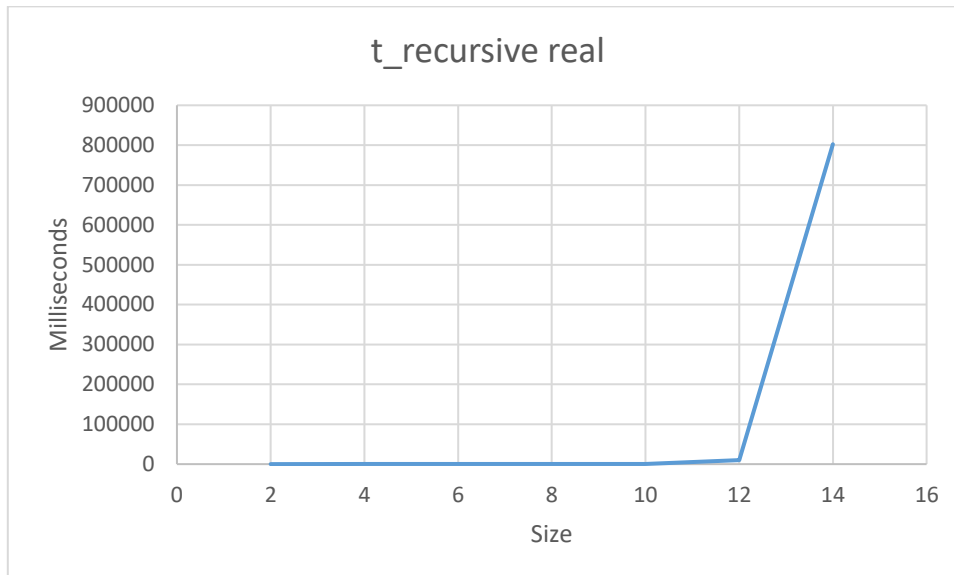
Activity 2. EXPERIMENTAL TIME MEASUREMENTS

In this point I will discuss only the experimental complexities and times obtained, the comparison with theoretical times will be performed in Activity 3.

- RECURSIVE VERSION:

n	t_recursive
2	0,4
4	1,2
6	7,8
8	55,1
10	579,9
12	9686,1
14	802295,6

Algorithmics	Student information	Date	Number of session
	UO: 276244	28/03/21	5
	Surname: Beltran Diaz		
	Name: Martin		



The recursive version implements D&C by subtraction. We consider $b=1$, since in each call we subtract one character from each string; $a = 3$, because there are three recursive calls to itself; and $k=0$, since the rest of the code has complexity $O(1)$, that is n^0 . Take into account that `longest()` method has complexity $O(1)$. With $a=3$, $b=1$ and $k=0$, as $a>1$, then we consider the complexity of this recursive algorithm as $O(a^{(n/b)}) = O(3^n)$.

```

public String findLongestSubseq(String s1, String s2){
    //Base case
    if(s1.length() == 0 || s2.length() == 0) {
        return "";
    }
    else {
        char c1 = s1.charAt(s1.length()-1);
        char c2 = s2.charAt(s2.length()-1);

        String s1prime = s1.substring(0, s1.length()-1); //S1 without rightmost element
        String s2prime = s2.substring(0, s2.length()-1); //S2 without rightmost element

        String l1 = findLongestSubseq(s1prime, s2);
        String l2 = findLongestSubseq(s1, s2prime);
        String l3 = findLongestSubseq(s1prime, s2prime);

        int longest = -1;
        if(c1 == c2) {
            l3 = l3 + c2;
            longest = longest(l1,l2,l3);
        }
        else {
            longest = longest(l1,l2,l3);
        }

        if(longest == 1) return l1;
        else if(longest == 2) return l2;
        else if(longest == 3) return l3;
        else throw new RuntimeException();
    }
}

```

Algorithmics	Student information	Date	Number of session
	UO: 276244	28/03/21	5
	Surname: Beltran Diaz		
	Name: Martin		

- **DYNAMIC VERSION:**

The times measured were for both the fillTable() and findLongestSubseq() methods, but I will study their complexities separately and then add them.

First, for the fillTable() method we can see first two non-nested for-loops, from 0 to the size of the strings, so both have complexity $O(n)$. Then we find two nested loops from 0 to n each one, and inside, in both cases of the if-statement, an invocation of longest() method, with complexity $O(1)$. So the overall complexity of these nested loops would be $O(n^2)+O(1)+O(1)$, so $O(n^2)$.

```

public void fillTable(){
    // TODO: fill dynamic programming table with a cell (value, iPrev and jPrev) for each entry
    for(int i = 0; i < size1; i++) {
        table[i][0].value=0;
        table[i][0].iPrev = 0;
        table[i][0].jPrev = 0;
    }
    for(int j = 0; j<size2;j++) {
        table[0][j].value = 0;
        table[0][j].iPrev = 0;
        table[0][j].jPrev = 0;
    }
    int longe = 0;
    for(int i = 1; i<size1;i++) {
        for(int j = 1; j<size2;j++) {
            if(str1.charAt(i)==str2.charAt(j)) {
                longe=longest(table[i-1][j].value, table[i][j-1].value, table[i-1][j-1].value +1);
                table[i][j].value = longe;
                table[i][j].iPrev = i-1;
                table[i][j].jPrev = j-1;
            }
            else {
                longe = longest(table[i-1][j].value, table[i][j-1].value, table[i-1][j-1].value);
                table[i][j].value = longe;
                if(longe == table[i][j-1].value) {
                    table[i][j].iPrev = i;
                    table[i][j].jPrev = j-1;
                }else if(longe == table[i-1][j].value) {
                    table[i][j].iPrev = i-1;
                    table[i][j].jPrev = j;
                }
            }
        }
    }
}

```

For the method findLongestSubseq() I only have a while-loop from the right bottom corner to the left upper corner. At most this loop will be n^2 , but it hardly will reach this complexity.

Algorithmics	Student information	Date	Number of session
	UO: 276244	28/03/21	5
	Surname: Beltran Diaz		
	Name: Martin		

```

public void findLongestSubseq(boolean v){
    // TODO: After the table is filled, from table last element traces the MSC found
    int iCurrent = size1-1;
    int jCurrent = size2-1;

    int iPrev = table[iCurrent][jCurrent].iPrev;
    int jPrev = table[iCurrent][jCurrent].jPrev;

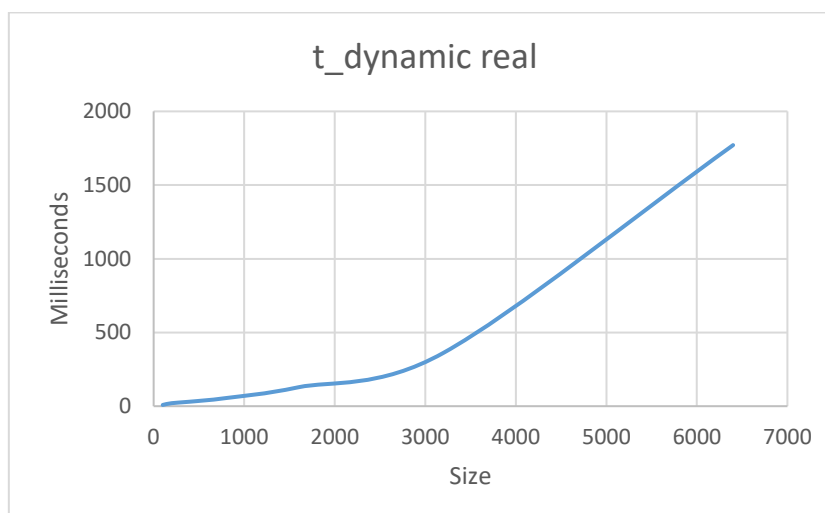
    while(iCurrent != 0 && jCurrent != 0) {
        if(iCurrent == iPrev+1 && jCurrent == jPrev+1)
            {result = str1.charAt(iCurrent) + result;}

        iCurrent = iPrev;
        jCurrent = jPrev;
        iPrev = table[iCurrent][jCurrent].iPrev;
        jPrev = table[iCurrent][jCurrent].jPrev;
    }
}

```

So the overall complexity of filling the table and finding the longest subsequent would be $O(n^2) + O(n^2)$, that is $O(n^2)$.

n	t_dynamic
100	8,2
200	20,5
400	30,4
800	55,2
1600	128,5
3200	361,4
6400	1771



Algorithmics	Student information	Date	Number of session
	UO: 276244	28/03/21	5
	Surname: Beltran Diaz		
	Name: Martin		

Activity 3. QUESTIONS

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

Theoretical complexities have already been explained in Activity 2.

In the recursive version the waste of stack would be $O(n)$, and the height of the tree of calls is N , because in each call we only reduce the strings by 1 char.

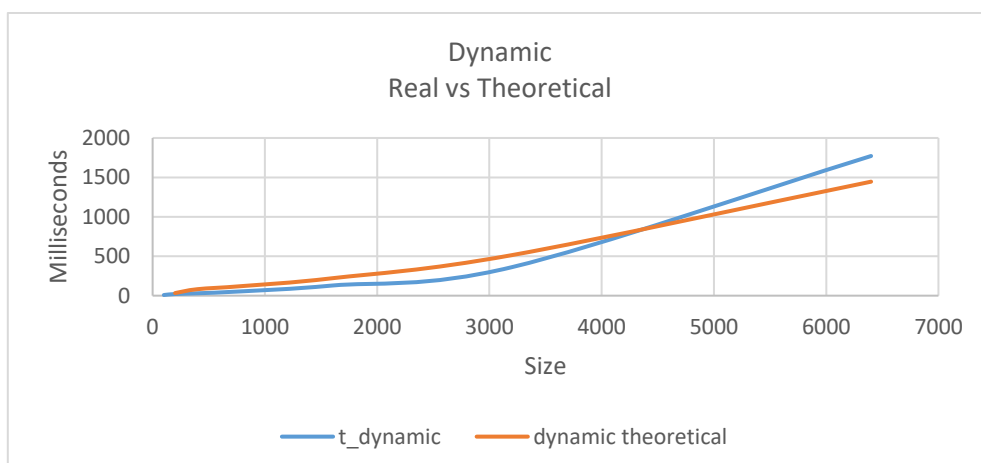
So the waste of stack is $O(n)$ and the memory consumption is $O(n * n) = O(n^2)$.

In the dynamic version, the memory consumption is directly proportional to the size of the matrix of all the values. That is, from my point of view, its memory consumption complexity is $O(n^2)$: n by n because the size of the matrix is `sizeFirstSequence * sizeSecondSequence`.

B) Compute theoretical times and compare them with the experimental measurements.

For the dynamic version, using the formula $f(n_2)/f(n_1) = t_2/t_1$ I obtain the following results:

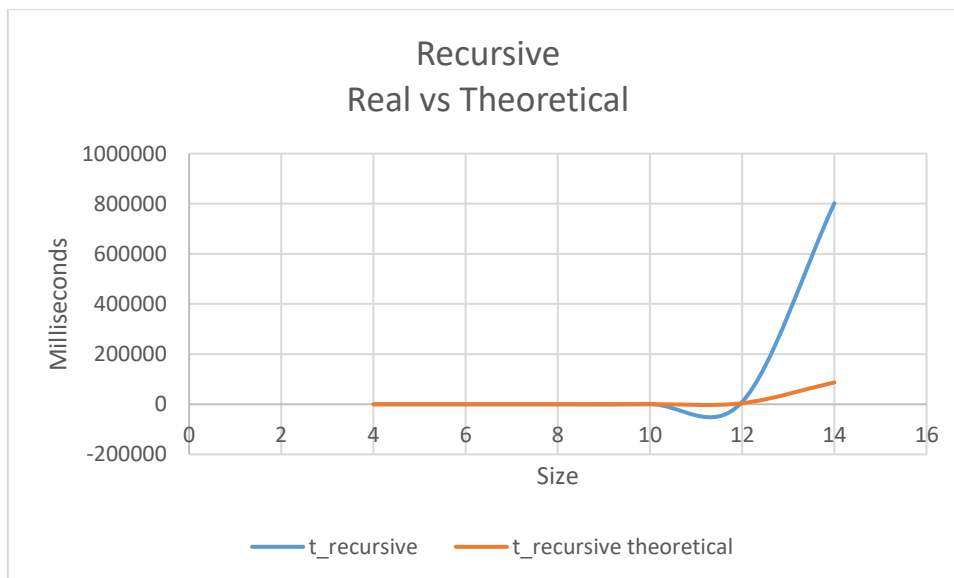
n	t_dynamic	dynamic theoretical
100	8,2	
200	20,5	32,8
400	30,4	82
800	55,2	121,6
1600	128,5	220,8
3200	361,4	514
6400	1771	1445,6



Algorithmics	Student information	Date	Number of session
	UO: 276244	28/03/21	5
	Surname: Beltran Diaz		
	Name: Martin		

For the recursive version I obtain the following results (using the same formula):

n	t_recursive	t_recursive theoretical
2	0,4	
4	1,2	3,6
6	7,8	10,8
8	55,1	70,2
10	476,9	495,9
12	9686,1	4292,1
14	802295,6	87174,9



C) Why large sequences cannot be processed with the recursive implementation? Explain why dynamic programming implementation raises an exception for large sequences.

As it is a recursive implementation, at some point it will throw a StackOverflow exception. On the other hand the dynamic version throws an OutOfMemoryException, because when using really long sequences, for example, of 1000 characters and 500 characters, the algorithm will need a matrix of 1000*500, and that is a lot of memory.

Algorithmics	Student information	Date	Number of session
	UO: 276244	28/03/21	5
	Surname: Beltran Diaz		
	Name: Martin		

D) 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.

The part in the code where I decide this is here:

```

else {
    longe = longest(table[i-1][j].value, table[i][j-1].value, table[i-1][j-1].value);
    table[i][j].value = longe;
    if(longe == table[i][j-1].value) {
        table[i][j].iPrev = i;
        table[i][j].jPrev = j-1;
    }else if(longe == table[i-1][j].value) {
        table[i][j].iPrev = i-1;
        table[i][j].jPrev = j;
    }
}

```

If the condition were swapped with the one in the else-if statement, another chain will be formed, since from two equal values in the table, between the upper one or left one, we will choose whichever is in the first if statement.