Laboratory practice No. 1: Recursion

Simón Álvarez Ospina
Universidad EAFIT
Medellín, Colombia
salvarezo1@eafit.edu.co

David Madrid Restrepo
Universidad EAFIT
Medellín, Colombia
dmadridr@eafit.edu.co

August 26, 2020

1 Report

i. For the code in 1.1 and 1.2 see [1].

ii. For the code in 2.1 and 2.2 see [2].

iii. *.

i. To calculate the complexity of the code made in 1.1, we have to work with two variables: n and m, which are the length of the first and second string, respectively. We start with the next two equations:

i. $T(n, m) = c_1 + T(n, m-1)$, c_1 constant.
ii. $T(n, 0) = c_2 + T(n-1, m)$, c_2 constant.

Because in the first equation n is constant in both sides, we can solve for m and get the following:

$T(n, m) = c_1 \cdot m + T(n, 0) := c_1 \cdot m + c_2 + T(n-1, m)$ (iii.)

In this new equation is evident that m is constant so we can repeat the procedure:

$T(n, m) = c_1 \cdot m \cdot n + c_2 \cdot n + c_3$, c_3 constant (iv.).

With this last equation, we can define the complexity as: $O(c_1 \cdot m \cdot n + c_2 \cdot n + c_3)$. By the addition rule, we'll get $O(c_1 \cdot n \cdot m + c_1 \cdot n)$ and, by the product rule, the complexity is $O(n \cdot m + n)$. Because $O(n \cdot m) < O(n(m+1)) < O(2 \cdot n \cdot m)$, the final complexity is $O(n \cdot m)$.

ii. *.

Python			
Strings' Length		Sum of Lengths	Time (s)
Length First String	Length Second String		
150	300	450	0.03778
300	600	900	0.11938
450	900	1350	0.23361
600	1200	1800	0.40085
750	1500	2250	0.68071
900	1800	2700	0.88860
1050	2100	3150	1.15334
1200	2400	3600	1.50448
1350	2700	4050	1.88600
1500	3000	4500	2.32000
1650	3300	4950	2.79851
1800	3600	5400	3.31123
1950	3900	5850	3.87271
2100	4200	6300	4.45454
2250	4500	6750	5.10074
2400	4800	7200	5.78328
2550	5100	7650	6.55773
2700	5400	8100	7.29444
2850	5700	8550	8.15138
3000	6000	9000	9.00145

Table 1: Time required to find the coincidences between two strings.

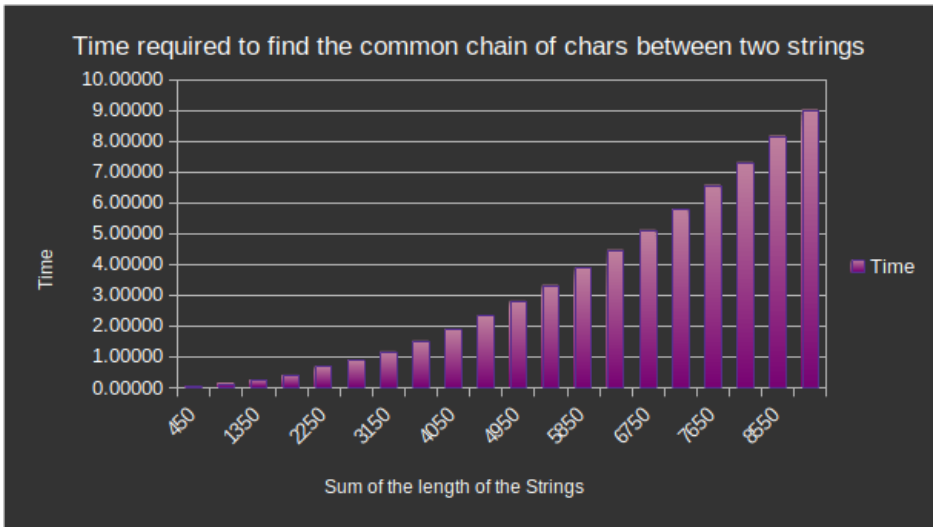
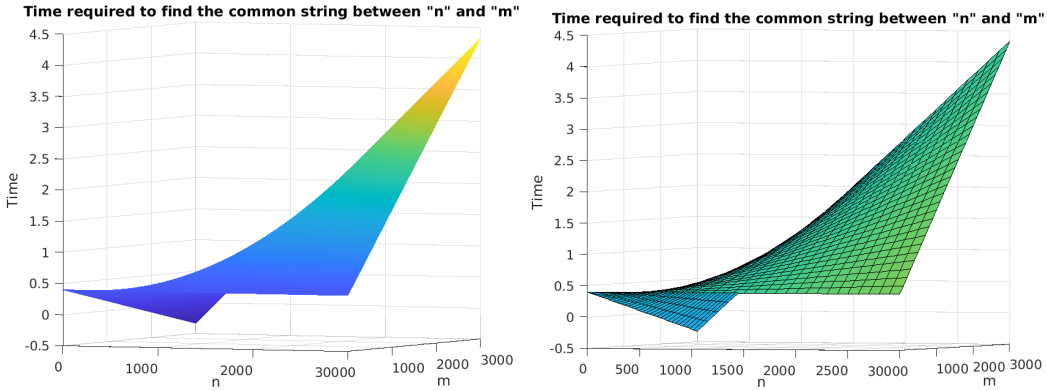


Figure 1: Graphic of Table 1



(a) (b)

Figure 2: Three-dimensional graph of $T(n, m)$

With the purpose of finding the time required to obtain the common chain of chars between two strings, we use three dates of the table to find out which are the approximate values of c_1, c_2 and c_3. For this, we use the function of the number of operations demonstrated before in 3.1:

i. $T(3000, 6000) = c_1 \cdot 18 \cdot 10^6 + c_2 \cdot 3000 + c_3 = 9s$
ii. $T(2700, 5400) = c_1 \cdot 1458 \cdot 10^4 + c_2 \cdot 2700 + c_3 = 7.3s$
iii. $T(2250, 4500) = c_1 \cdot 10125 \cdot 10^3 + c_2 \cdot 2250 + c_3 = 5.1s$

This is easier to visualize on a matrix, in this way:

$$\left[\begin{array}{ccc|c} 18000000 & 3000 & 1 & 9 \\ 14580000 & 2700 & 1 & 7.3 \\ 10125000 & 2250 & 1 & 5.1 \end{array}\right] \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} \longrightarrow \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} \frac{7}{13500000} \\ \frac{-11}{45000} \\ \frac{2}{5} \end{bmatrix}$$

With the formula in (iv), if we replace c_1, c_2, c_3* by its values in the matrix, we obtain the following:

$$T(300000, 300000) = \frac{7}{13500000} \cdot 300000^2 + \frac{-11}{45000} \cdot 300000 + \frac{2}{5} = \frac{698906}{15} s$$

With these results we conclude that it will take 12 hours 56 minutes and 34 seconds to calculate the coincidences between the two strings.

Note: The programming language used in the algorithm was Python, so the time required will be more than if we would used another programming language like Java.

iii. The program took 12 hours with 56 minutes and 34 seconds to find the matches between both DNA strands, which is undoubtedly time consuming, considering that it could be drastically reduced by changing the test environment (the programming language, in this case Python), because there are others low level programming languages that could get the same result in less time than the given in the previous test.

iv. GroupSum5 is a recursive algorithm that given an array of integers, an initial value and a target, try to find if there is a subset of integers that when they are added give as result the value of target. This exercise also has extra cases that must be included, as if the array has numbers multiples of 5, they must be part of the sum. Also, if the number multiple of five is previous to a number 1, this won't be included in the sum.

v. *.

i. The complexity of each exercise in Recursion 1 [3] is the same for all of them, the only thing that changes is what n means:

i. **count8:** In this exercise, n means the amount of digits that the number has.
ii. **array6:** Here, n is the length of the array.
iii. **array11:** The same as array6.
iv. **count11:** n is the length of the string.

*In this report we did not focus on finding the actual values of c_1, c_2, c_3, but to approximate the amount of time needed to find the coincidences between the first string n and the second one m.

v. **nestParen:** The same as count11.

The function that finds the time required to the worst case of the exercises above, is:

$T(n) = c_1 + T(n-1) := c_1 \cdot n + c_2$, $c_1 \wedge c_2$ constants *.

With this function, we can define the complexity of each algorithm as $O(c_1 \cdot n + c_2)$ and, by the addition rule and product rule, is $O(n)$.

ii. The functions that define the required time to the exercises be executed in recursion 2 [4] are a little more complicated. Here, we will concentrate in two different groups, but the complexity of both of them will be the same, as we'll demonstrate later.

i. **groupSum6:** In this exercise, n means the length of the array.
ii. **groupSum5:** The meaning of n is the same as the previous one.
iii. **splitArray:** Here, n is the length of the array.
iv. **splitOdd:** The same as splitArray.
v. **split53:** The same as splitArray too.

Now, to understand the number of operations required, we're going to split the previous five exercises in two groups, this because two of them have the same formula and the other three have another.

i. **Function of groupSum6 and groupSum5:** The number of operations required to these two exercises is the following:

$T(n) = c_1 + 2 \cdot T(n-1) := c_1(2^n - 1) + c_2 \cdot 2^{n-1}$, $c_1 \wedge c_2$ constants.

ii. **Function of splitArray, splitOdd and split53:** There are a number of differences in these three cases because we used loops, which makes $c_m \cdot n + c_k$ operations (k and m are just indices), and it will be taken in account in the function:

$T(n) = c_1 \cdot n + c_{2.1} + c_{2.2} + 2 \cdot T(n-1) = c_2 + c_1 \cdot n + 2 \cdot T(n-1) := c_1(2^{n+1} - n - 2) + c_2(2^n - 1) + c_3 \cdot 2^{n-1}$.

Now, we can define the complexity using these two functions. For the first case, complexity is defined as $O(c_1(2^n - 1) + c_2 \cdot 2^{n-1})$ that, by addition rule and

*The values of c_1 and c_2 can change depending of the exercise

product rule, is equal to $O(2^n)$.
For the second case, the complexity is defined as $O(c_1(2^{n+1}-n-2)+c_2(2^n-1)+c_3\cdot 2^{n-1})$, which is equivalent to $O(c_4\cdot 2^n - c_1\cdot n + c_5 + c_2\cdot 2^n - c_2 + c_3)$. This, because of the addition rule, is the same as $O(2^n(c_2+c_4)+c_6\cdot n)$, but, by the product rule, we're going to get $O(2^n+n)$. Now, we'll use the fact that $2^n > n$ for all n. With this, we can use the next inequality: $O(2^n) < O(2^n+1) < O(2^n+2^n) := O(2\cdot 2^n)$. Because the complexity of the right hand side is the same as $O(2^n)$ by the product rule, we can say that $O(2^n+n) \equiv O(2^n)$. Thus, we conclude that the final complexity is $O(2^n)$.

With these two results, is evident that both of them are the same, so the complexity for all the exercises in Recursion 2 are equivalent.

2 Midterm Exam

- Midterm simulation[5]:

  i. *.
  - **i.** $s.substring(i, n)$
  - **ii.** $true$
  - **iii.** $solve(t, s.substring(i), n - i)$

  ii. *.
  - **i.** $true$
  - **ii.** If $f(n)$ is $O(g(n))$ and $g(n)$ is $O(h(n))$, then $f(n)$ is $O(h(n))$. For any base $b > 0$, $log_b(n)$ is $O(ln(n))$.

  iii. $T(n, m) = c \cdot n \cdot m^2$.

  iv. $T(n) = T(n-1) + T(n-2) + c$, that is $O(2^n)$.

  v. *.
  - **i.** $true$.
  - **ii.** $s.charAt(0) == (s.charAt(s.length() - 1))$.

  vi. *.

  vii. The sum of the elements in the array a and its complexity is $O(n)$.

  viii. *.

  ix. *.
  - **i.** $sumaAux(n, i + 2)$
  - **ii.** $sumaAux(n, i + 1)$.

Professor Mauricio Toro Bermudez
Phone: (+57)(4)2619500 Ext. 9473. Office: 19 – 627
E-mail: mtorobe@eafit.edu.co

3 SubsequenceORDERED

Due to a problem in the interpretation of the problem 1.1, we had to redo the points of 3.1 and 3.2 [6], so... Here we go!

i. In this problem, is obvious that the worse problem is when the two strings (n, m, respectively) have the same length (This because we try to not finish the algorithm earlier), so $T(n, m)$ (number of operations) can depend only of one variable (whichever); we'll work with n. $T(n)$ is defined as follows:

$$T(n) = c_1 + 2 \cdot T(n-1) := c_1 \cdot (2^n - 1) + c_2 \cdot 2^{n-1}.$$

Due to the reflexive rule, we define the complexity of $T(n)$ as $O(c_1 \cdot (2^n - 1) + c_2 \cdot 2^{n-1}$. By the addition, we get $O(c_1 \cdot 2^n + c_4 \cdot 2^n)$ and, by the product rule, this is equivalent to $O(2^n)$.

ii. *.

Java	
Length Strings	Time (s)
1	0
2	0
3	0
4	0
5	0
6	0
7	0
8	0
9	0
10	0.001
11	0.001
12	0.005
13	0.019
14	0.066
15	0.256
16	0.944
17	3.801
18	14.628

Table 2: Time required to find the coincidences in order between two strings with the same length.

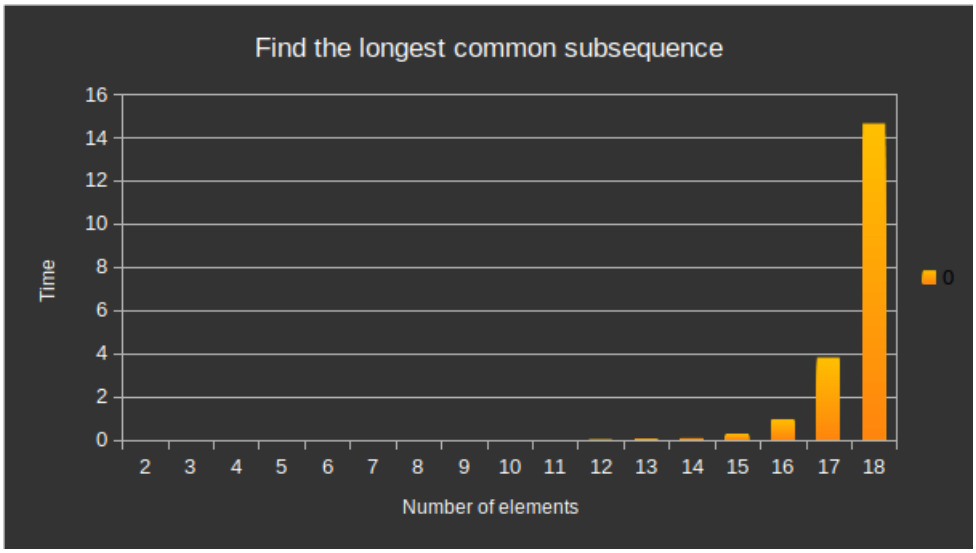


Figure 3: Graph of table 2.

With these dates, we can find the values of c_1, c_2 and c_3, which are: $c_1 = 20996983/655360000, c_2 = (-5113)/80000$ and $c_3 = (-4583562377)/655360000$ [†]. Now, we can predict the time required when n and m are equal to 300000 and it is: 10^90305s which is enough time to not make a conversion in centuries.

iii. Obviously, with this amount of time required, this isn't a good algorithm to work with strings that have a length of 300000 chars on them.

[†]By the same reason that before, here we don't try to find the actual values of the three constants, just make an approximation. Also, the values that we are working with, are small enough to don't have a behaviour well-defined.

References

[1] S. Álvarez and D. Madrid. (). For code 1.1 and 1.2, [Online]. Available: https://github.com/dmadridr/ST0245-002/tree/master/laboratorios/lab01/codigo.

[2] ——, (). For code 2.1 and 2.2, [Online]. Available: https://github.com/dmadridr/ST0245-002/tree/master/laboratorios/lab01/ejercicioEnLinea.

[3] Codingbat. (). Recursion 1 algorithms, [Online]. Available: https://github.com/dmadridr/ST0245-002/tree/master/laboratorios/lab01/ejercicioEnLinea/Recursion%201.

[4] ——, (). Recursion 2 algorithms, [Online]. Available: https://github.com/dmadridr/ST0245-002/tree/master/laboratorios/lab01/ejercicioEnLinea/Recursion%202.

[5] M. Toro, "Laboratorio Nro. 1 Recursión.", version 19., pp. 12–22, 2019. [Online]. Available: https://github.com/mauriciotoro/ST0245-Eafit/blob/master/laboratorios/lab01/ED1-Laboratorio1%20Vr%2019.0.pdf.

[6] (), [Online]. Available: https://github.com/dmadridr/ST0245-002/blob/master/laboratorios/lab01/codigo/Java%20Language/LongSubseqORDERED.java.

Professor Mauricio Toro Bermudez
Phone: (+57)(4)2619500 Ext. 9473. Office: 19 – 627
E-mail: mtorobe@eafit.edu.co