# **Laboratory practice No. 1: Recursion**

|  |  |
| --- | --- |
| **Julian David Ramirez Lopera**  Universidad Eafit  Medellín, Colombia  jdramirezl@eafit.edu.co | **Andres Salazar**  Universidad Eafit  Medellín, Colombia  asalaza5@eafit.edu.co |

**3) Practice for final project defense presentation**

**3.1 The Longest Common Substring algorithm takes two strings and finds the LCS. It achieves this by comparing each character (two pointers), starting from the back, and generating three different recursions: 1. if the characters are the same, we move both the pointers back by one and 2. if they don’t match the algorithm returns the max between moving the pointer of the first string one position and moving the pointer of the second string one position back as well. This ends when either pointer gets to the begging of their respective string. The algorithm complexity is O(2^k), K being the length of the longest string as in the worst case when there are no matches the pointers need to move through the longest string.**

**3.2 The algorithm takes approximately 0 ms to run with 300.000 characters.**

**3.3 Even though the algorithm seems fast at first, an exponential time complexity becomes slow when comparing millions of characters, an approach with loops is better.**

**3.4 GroupSum5 works as follows: It the array, an index and the target value. It first checks if we have either arrived at the target value (which returns true) or if we have passed through all the array and we didn’t get to the value (Which returns false). If neither case happened then two things can happen: 1. the number at the position is a multiple of 5 and the next one is a one, so we add the multiple of five and we skip two positions so we ignore the one, or 2. we have a multiple of five not followed by a one or we have a normal number in which case we make two recursive calls: One adding the number to the target and other not adding the number.**

**3.5**

**Recursion 1:**

* **count7.java: This algorithm determines the number of occurrences of a digit in each number. And although it implements recursion, its worst-case time and space complexity are of O(N) and O(N) respectively. This is because the number of operations is always going to be the length of the number (# of digits) and the depth of the recursion will go as far as the length of number (# of digits).**
* **count8.java: Like count7, this problem enquires for the number of digits in a number that are equal to 8. Nonetheless, it requires to count every 8 that has an 8 to its left twice. This essentially only replaces a O(1) operation for another. Which in turn leaves the overall time and space complexity to remain at O(N).**
* **countx.java: This algorithm performs a linear scan over a string in a recursive way. For this reason, its time and space complexity are of O(N).**
* **SumDigits.java: This code goes through the given number digit by digit with no repetitions making it O(n) in both time and space.**
* **triangle.java: This algorithm performs the sum of numbers up to point. Consequently, its time and space complexities remain at O(N).**

**Recursion 2:**

* **groupNoAdj.java: This algorithm branches into two recursive calls for each time it is called. By subtracting the current value of the index of the nums array. It is possible to establish whether it is possible to reach a sum once the target reaches zero. Because this algorithm does two recursive operations for each call, the overall time complexity turns to be O(2^n), with a worst-case time complexity of O(n) which represents the height of the recursion call stack.**
* **groupSum5.java: This algorithm specifies that all numbers divisible by 5 must be added to the sum in order to reach the target, and that if a 1 follows any of these numbers, it must not be included in the sum. Although it can be argued that these conditions could reduce the average space and time complexity, the worst-case scenario remains to be O(2^k) time and O(n).**
* **GroupSumClump.java: This algorithm specifies that groups of consecutive equal numbers must be summed together to reach the target. Similarly, this involves optimizations in average runtimes. However, the overall worst-case time and space complexity remains at O(2^k) time and O(n).**
* **split53.java: This algorithm requests to determine whether it is possible to divide an array such that elements divisible by 3 are in one group, elements divisible by 5 and not divisible by 3 are in another and for any other element, it can be in their group. This turns out to be a O(2^k) time and O(n). Space complexity here stands for the case where all the elements belong to one of the specified groups.**
* **splitOdd10.java: This algorithm branches into two recursive calls for each time it is called. Thus, time complexity turns out to be O(2^k) and space complexity O(n).**

**3.6 The explanation for N and M for each of the problems are in their respective explanations.**

***4) Practice for midterms***

* 1. *A – C - A*
  2. *Lineas 9 a 12 no se agrega nada.*
  3. *B*
  4. *C*
  5. *A – B*
  6. *A*
  7. sumaAux(n, i+2) - sumaAux(n, i+1)

***5) Recommended reading (optional)***

Mapa conceptual

**6)** **Team work and gradual progress (optional)**

* 1. Meeting minutes

***6.2*** History of changes of the code

***6.3*** History of changes of the report