Laboratory practice 1: Recursion

Santiago Isaza Cadavid
Universidad EAFIT
Medellín, Colombia
sisazac@eafit.edu.co

Hamilton Smith Gómez Osorio
Universidad EAFIT
Medellín, Colombia
hsgomezo@eafit.edu.co

August 19, 2018

1) *Project questions Simulation*

1.a. *Explain how does GroupSum5 works*

In this exercise the objective is to check if is possible to get a target (num) adding the numbers that are in an array. But there are restrictions: all the multiples of 5 in the array must be added. Procedure: the program receives an index, an array of integers and the target/objective. We evaluate if the index is equal or greater than the length of the array, if it is, if the target is zero then it returns "true" if not, then it returns "false". While it is not the case we keep evaluating every position in the array and checking if it is multiple of 5, if it is then we verify that the next number is or not, a number one. If it is, we make a recursive call with the index plus two units (to omit the number one) and subtracting the multiple already found from the objective; when it is not like that, the parameters that we give to the program are the same but the index is now index plus one. When the number in the position of the index is not multiple of five then we make a recursive call in which we intend to analyze the next position of the array without counting the previous visited number and other recursive call in which we do count the previous visited number in order to check in which option we could get the target.

1.b. *Calculate the complexity of the online exercises*

1.2.1 powerN

```
public int powerN(int base, int n) {
    int prod=1; // C
    if(base==1 —— n==0) return 1; // C
    else prod= base * powerN(base,n-1); // C*T(n-1)
    return prod; // C
    }
```

$$T(n) = c * T(n-1)$$

The complexity of this algorithm is O(log n)

1.2.2 SumDigits

```
public int SumDIgits(int n) { int sum=0;
    if(n==0)return 0; //C
    else sum= n%10 + sumDigits(n/10); // C + T(n-1)
    return sum; // C }
```

$$T(n) = T(n-1) + C_1 + C_2 + C_3$$
$$T(n) = c + (C_1 + C_2 + C_3) * n$$
$$T(n) = O(c + (C_1 + C_2 + C_3) * n)$$
$$T(n) = O((C_1 + C_2 + C_3) * n)$$
$$T(n) = O(n)$$

The complexity of this algorithm is O(n)

1.2.3 Triangle

```
public int triangle(int rows) {
    int block=0;
    if(rows==0)return 0; // C
    else block= rows + triangle(rows-1); // C + T(n-1)
    return block;
    }
```

$$T(n) = T(n-1) + C_1 + C_2$$
$$T(n) = c + (C_1 + C_2) * n$$
$$T(n) = O(c + (C_1 + C_2) * n)$$
$$T(n) = O((C_1 + C_2) * n)$$
$$T(n) = O(n)$$

The complexity of this algorithm is O(n)

1.2.4 count7

```
public int count7(int n) {
    int cont=0;
    if(n==0) return 0; // C
    else if(n%10==7) cont= 1 + count7(n/10); // C + T(n-1)
    else cont= count7(n/10); // T(n-1)
    return cont;
    }
```

$$T(n) = T(n-1) + T(n-1) + c$$

$$T(n) = 2^{n-1} + c$$
$$T(n) = O(2^{n-1} + c)$$
$$T(n) = O(2^{n-1})$$
$$T(n) = O(2^n)$$

The complexity of this algorithm is $O(2^n)$

1.2.5 bunnyEars

```
public int bunnyEars(int bunnies) {
    int ears=0;
    if(bunnies==0) return 0; // C
    else ears= 2+bunnyEars(bunnies-1); // C + T(n-1)
    return ears;
    }
```

$$T(n) = T(n-1) + C_1 + C_2$$
$$T(n) = c + (C_1 + C_2) * n$$
$$T(n) = O(c + (C_1 + C_2) * n)$$
$$T(n) = O((C_1 + C_2) * n)$$
$$T(n) = O(n)$$

The complexity of this algorithm is O(n)

1.2.6 groupSum6

```
public boolean groupSum6(int start, int[] nums, int target) {
    if(start¿=nums.length) return target==0; // C
    if(nums[start]==6) return groupSum6(start+1,nums,target-nums[start]); // C + T(n-1)
    return groupSum6(start+1,nums,target-nums[start]) || // C + T(n-1)
    groupSum6(start+1,nums,target); // C +T(n-1)
    }
```

$$T(n) = T(n-1) + T(n-1) + T(n-1) + C_1 + C_2 + C_3$$
$$T(n) = 3^{n-1} + C_1 + C_2 + C_3$$
$$T(n) = O(3^{n-1} + C_1 + C_2 + C_3)$$
$$T(n) = O(3^{n-1})$$
$$T(n) = O(3^n)$$

The complexity of this algorithm is $O(3^n)$

1.2.7 groupNoAdj

```
public boolean groupNoAdj(int start, int[] nums, int target) {
    if(start¿=nums.length) return target==0; // C
    return groupNoAdj(start+1,nums,target) || // C + T(n-1)
    groupNoAdj(start+2,nums,target-nums[start]); // T(n-2)
    }
```

$$T(n) = T(n-1) + T(n-2) + C_1 + C_2$$

$$T(n) = T(n-1) + T(n-1) + C_1 + C_2$$

$$T(n) = 2^{n-1} + C_1 + C_2 +$$

$$T(n) = O(2^{n-1} + C_1 + C_2)$$

$$T(n) = O(2^{n-1})$$

$$T(n) = O(2^n)$$

The complexity of this algorithm is $O(2^n)$

1.2.8 groupSum5

Taken from: http://gregorulm.com/codingbat-java-recursion-2/

```
    public boolean groupSum5(int start, int[] nums, int target)
    if(start>=nums.length) return target==0; // C
    if(nums[start]%5==0)
    if(start¡nums.length-1& &nums[start+1]==1)
    return groupSum5(start+2,nums,target-nums [start]); //C + T(n-2)
    return groupSum5(start+1,nums,target-nums[start]);
    // C + T(n-1)
    return groupSum5(start+1,nums,target) || // C + T(n-1)
    groupSum5(start+1,nums,target-nums[start]);
    //T(n-1)
```

$$T(n) = T(n-1) + T(n-1) + T(n-2) + C_1 + C_2 + C_3$$

$$T(n) = T(n-1) + T(n-1) + T(n-1) + C_1 + C_2 + C_3$$

$$T(n) = 3^{n-1} + C_1 + C_2 + C_3$$

$$T(n) = O(3^{n-1} + C_1 + C_2 + C_3)$$

$$T(n) = O(3^{n-1})$$

$$T(n) = O(3^n)$$

The complexity of this algorithm is $O(3^n)$

1.2.9 splitArray

Taken from: http://gregorulm.com/codingbat-java-recursion-2/

```
public boolean splitArray(int[] nums) {
return theRealSplit(nums,0,0,0);
}
public boolean theRealSplit(int [] nums, int start, int first, int second){
if(start==nums.length) {
return first==second; // C
}else{
return theRealSplit(nums,start+1, first+nums[start],second) || // C + T(n-1)
theRealSplit(nums,start+1,first,second+nums[start]);
// C + T(n-1)
}
}
```

$$T(n) = T(n-1) + T(n-1) + C_1 + C_2$$

$$T(n) = 2^{n-1} + C_1 + C_2$$

$$T(n) = O(2^{n-1} + C_1 + C_2)$$

$$T(n) = O(2^{n-1})$$

$$T(n) = O(2^n)$$

The complexity of this algorithm is $O(2^n)$

1.2.10 groupSumClump

```
public boolean groupSumClump(int start, int[] nums, int target) {
if(start¿=nums.length) return target==0; //C
int sum=0;
int i;
for(i=start;i<nums.length;i++){
if(nums[i]==nums[start]){
sum+ =nums[start]; // C + C
}else{
break;
}
}
return groupSumClump(i,nums,target) || //C + T(n-1) groupSumClump(i,nums,target-
sum); // C + T(n-1) }
```

$$T(n) = T(n-1) + T(n-1) + C_1 + C_2 + C_3 + C_4$$

$$T(n) = 2^{n-1} + C_1 + C_2 + C_3 + C_4$$

$$T(n) = O(2^{n-1} + C_1 + C_2 + C_3 + C_4)$$

$$T(n) = O(2^{n-1})$$

$$T(n) = O(2^n)$$

The complexity of this algorithm is $O(2^n)$

1.c. Explain what the variable n means in the previous exercises

1.3.1 powerN

The complexity of this algorithm is O(log n). The variable n means the power required for the user.

1.3.2 sumDigits

The complexity of this algorithm is O(n). The variable n means the number which the algorithm sums its digits.

1.3.3 Triangle

The complexity of this algorithm is O(n). The variable n means the number of rows that the triangle has.

1.3.4 Count7

The complexity of this algorithm is $O(2^n)$ The variable n means the number which the algorithm counts how many sevens it has.

1.3.5 bunnyEars

The complexity of this algorithm is O(n). The variable n means the number of bunnies.

1.3.6 groupNoAdj

The complexity of this algorithm is $O(2^n)$. The variable n means the positions that haven't been visited yet, is related with the size of the array.

1.3.7 groupSum5

The complexity of this algorithm is $O(3^n)$. The variable n means the positions that haven't been visited yet, is related with the size of the array.

1.3.8 splitArray

The complexity of this algorithm is $O(2^N)$. The variable n means a position in the array, where the algorithm evaluates according to the constraints.

1.3.9 groupSumClump

The complexity of this algorithm is $O(2^n)$. The variable n means the positions that haven't been visited yet, is related with the size of the array.

1.d. What did you learn about Stack Overflow? Why does this happen?

Thanks to this error we learned that a good base case (stopping condition) is necessary in order to avoid an infinite loop, and to get the expected answer in a recursive algorithm. The Stack Overflow error happens due to a excessive use of memory. It happens when the algorithm makes a bad recursive call.

1.e. What is the greatest number you could get with the Fibonnacci algorithm? Why? Why can't it work with Fibonnacci one million?

We made two tests, one with the page visualgo, in which the program only calculated twenty-six numbers, then when we implemented it in Java and it returned values until the forty-sixth number. The program didn't returned more values because it was a recursive algorithm, this type of algorithms create a pile of cases to analyze that do not conclude until it arrives to the case base and, we it tries to do it with very large numbers, the time and memory that the machine needs to get the job done is a lot, so this program doesn't have a very optimal behavior.

1.f. What can you do to calculate bigger Fibonacci 's values?

One of the solutions could be changing the type of data that stores this values. For example, in our code we used int type but if we used a long int type we would get bigger values. Other solution is using a big data structure, for example a dictionary in Python and add the values to it, but this a solution proposed by Functional Programming.

1.g. What do you conclude about the complexity of CodingBat's Recursion 1 with respect Recursion 2?

Recursion 2 exercises are more complex that the exercises of Recursion 1. We can see this because Recursion 2's complexity belongs to the order 2^n, where they grow exponentially, while Recursion 1's complexity is much more simple, because they grow on a lineal way. Other reason why Recursion 2 is more complex is because the number of recursive calls. Recursion 2 problems realize a decision tree, based on backtracking, to solve problems, doing this means more memory use and more recursive calls. Some of them having even tree recursive calls, while Recursion 1 problems only needed 1.

2) Midterm Simulation

2.a. Exercise 1

start+1, nums, target

2.b. Exercise 2

The recurrence equation that describes the algorithm's behavior is: a) T(n/2)+C

2.c. Exercise 3

2.3.1 Line 4

n - a , a , b , c

Line 5

res , solucionar(n-b,a,b,c) + 1

2.3.2 Line 6

res , solucionar(n-c,a,b,c) + 1

2.d. Exercise 4

The algorithm calculates: e) The sum of the elements that are in the array a and its complexity is O(n)

2.e. Exercise 5

2.5.1 Complete the lines

Line 2: return n Line 3: n-1 Line 4: n-2

2.5.2 How many instructions does this algorithm executes in the worst case scenario?

b) T(n)= T(n-1)+T(n-2)+C

2.f. Exercise 6

2.6.1 Line 10

0;

2.g. Line 12

sumaAux(n.substring(i+i),i+1)

2.h. Exercise 7

2.8.1 Line 9

s , i+1 , t;

2.8.2 Line 10

s , i+1 , t-s[i];

2.i. Exercise 8

2.9.1 Complete line 9

return 0;

2.9.2 Complete line 13

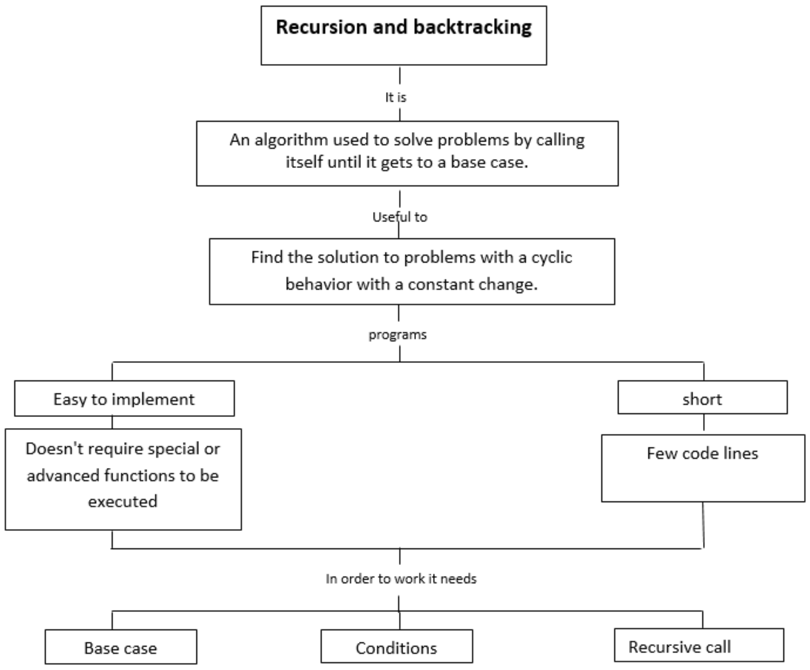
ni + nj

3) Recommended reading

3.a. Summary

Recursion and backtracking Recursion refers to those algorithms that call themselves in order to execute cyclic functions, but these functions change with the time, they start with an initial value and changing this value until they get to the base case to return a expected solution. Recursion is very used to solve problems because it is short and easy to program, you only need to establish the stopping condition and the change in the recursive call. On the other hand is responsible for solving what we can call stacked problems, which look for a base value and they are responsible for passing the results of the equations stored in memory to get the solution. This is the reason why solving problems with recursive algorithms take time and they are not very effective for problems that need many recursive calls because the algorithm wouldn't be optimal.

Figure 1: Recommended reading


Recursion and backtracking
It is
An algorithm used to solve problems by calling itself until it gets to a base case.
Useful to
Find the solution to problems with a cyclic behavior with a constant change.
programs
Easy to implement
Doesn't require special or advanced functions to be executed
short
Few code lines
In order to work it needs
Base case
Conditions
Recursive call