Laboratory practice No. 1: Recursion

Agustín Rico Piedrahita
Universidad Eafit
Medellín, Colombia
aricop@eafit.edu.co

Santiago Cano
Universidad Eafit
Medellín, Colombia
scanof@eafit.edu.co

2)

2.3

The algorithm of the problem GroupSum5 works on the following way: the parameters are *start*, *nums*, *target*. *Start* works as a counter of the array's index or number of the array the algorithm is working on, and as we want the algorithm to work on the whole array, it will always start as *0*. *Nums* is the array given. *Target* is the target sum. The case base is reached when *start=nums.length*, because at that point the whole array would be evaluated. But when *start* hasn't reached the array's length, it will evaluate if the position *start* of the array is a multiple of 5; in that case, that number must be chosen (to do this, there's not going to be two options for the algorithm), and if the next number is a '1', then the '1' must not be chosen (to do this, the algorithm will skip 2 positions). If the number is not a multiple of five, then the algorithm will choose or not the number on the *start* position on the array. In order to be possible to reach the base case, each recursive call will sum +1 to *start*, and to be capable of determine if *target* was equalized, as each chosen number is going to subtract *target*, when the base case is reached, the algorithm will evaluate if target is equal to 0 or not; if it is, will return *true*. If not, will return *false*.

2.4

```
public int bunnyEars2(int bunnies) {
if(bunnies == 0) { //C1
return 0; //C2
} else if (bunnies%2 != 0) { //C3 + n/2-1
return 2 + bunnyEars2(bunnies - 1); //C4+ n/2-1
} else { //C5+ n/2 -1
return 3 + bunnyEars2(bunnies - 1); C6 + n/2-1
} // C7 + n/2-1
}
```

Complexity:

In the best case:

T(n) = C1 + C2

In the worst case is linear

T(n)= C3+ C4+ C5+ C6+ C7+ 4(n/2 -1)

T(n)= 2n-4

T(n)= n

```
public int triangle(int rows) { with n = rows
if(rows <= 1) { // C1
return rows; // C2
} else { // C3 + n-1
return rows + triangle(rows - 1); //C4+ n-1
} //C5+ n-1
}
```

Complexity:

T(n) = C1+C2 For n<=1

T(n) = C3+ C4+ C5 + 3(n-1) On the contrary

T(n) = 3n -3

T(n) = n

The complexity is O(n): linear.

```
public int sumDigits(int n) {
if(n < 10) { //C1
return n; //C2
} else { //C3 + log n (base 10)
return n%10 + sumDigits(n/10); //C4 + log n (base 10)
} //C5+ log n (base 10)
}
```

Complexity:

T(n)= C1+ C2 Para n < 10

T(n)=C3 + C4 + C5 + 3 (log n(base 10)) On the contrary

T(n) = 3(log n(base 10))

T(n) = log n (base 10)

The complexity is O(log n (base 10)): Logarithmic.

```
public int count7(int n) {
if(n < 10 && n == 7) { //C1
return 1; //C2
} else if(n < 10 && n != 7) { //C3
```

```
return 0; //C4
}else { //C5
if(n%10 == 7) { //C6 + log n (base 10)
return 1 + count7(n/10); //C7+ log n (base 10)
} else { //C8+ log n (base 10)
return count7(n/10); //C9+ log n (base 10)
} //C10+ log n (base 10)
} //C11
}
```

Complexity:

T(n) = C1 + C2 With n<10 y n =7

T(n) =C3 +C4 With n<10 y n!=7

T(n)= C6+ C7+ C8+ C9+ C10 + 5 log n (base 10) On the contrary

T(n)= 5 log n (base 10)

T(n) = log n (base 10)

The complexity is es O(log n (base 10)): Logarithmic.

```
public int count8(int n) {
if(n < 10 && n == 8) { //C1
return 1; //C2
} else if(n < 10 && n != 8) { //C3
return 0; //C4
} else { //C5
if(n%100 == 88) { //C6
return 2 + count8(n/10); //C7
}else if(n%10 == 8) { //C8
return 1 + count8(n/10); //C9
}else { //C10
return count8(n/10); //C11
} //C12
} //C13

private boolean groupSum6(int start, int[] nums, int target) {
if(start >= nums.length) { //C1
return target == 0; //C2
```

```
} else { //C3
if(nums[start] == 6) { //C4
return groupSum6(start + 1, nums, target - nums[start]); //C5 + T(n-1)
} else { //C6
return groupSum6(start + 1, nums, target) ||
groupSum6(start + 1, nums, target - nums[start]); //C7 + 2T(n-1)
}
}
}
```

Complexity:

T(n) = C1+ C2 if start>= nums.length

T(n)= C7+ 2T(n-1) On the contrary

T(n) = C*2^n + C'

T(n) = 2^n

O(2^n)

The complexity is exponential.

//////

```
public boolean groupNoAdj(int[] nums, int target) {
return groupNoAdj(0, nums, target);
}
private boolean groupNoAdj(int start, int[] nums, int target) {
if(start >= nums.length) { //C1
return target == 0; //C2
} else { //C3
return groupNoAdj(start + 1, nums, target) ||
groupNoAdj(start + 2, nums, target - nums[start]); //C4 + 2T(n-1)
}
}
```

T(n) = C1+C2 si start>= nums.length

T(n)= C4+ 2T(n-1) On the contrary

T(n) = C*2^n + C'

T(n) = 2^n

O(2^n)

The complexity is exponential.

```
///////
public boolean groupSum5(int[] nums, int target) {
return groupSum5(0, nums, target);
}
private boolean groupSum5(int start, int[] nums, int target) {
if(start >= nums.length) { //C1
return target == 0; //C2
} else { //C3
if(nums[start] % 5 == 0 && start < nums.length - 1) { //C4
if(nums[start + 1] == 1) { //C5
return groupSum5(start + 2, nums, target - nums[start]); //C6 + T(n-2)
} else { //C7
return groupSum5(start + 1, nums, target - nums[start]); //C8 + T(n-1)
}
} else { //C9
return groupSum5(start + 1, nums, target - nums[start]) ||
groupSum5(start + 1, nums, target); //C10 + 2T(n-1)
}
}
}
```

Complexity:

T(n) = C1+ C2 if start>=nums.length

T(n) = C10 + 2T(n-1)

T(n) = C*2^n + C'

T(n) = 2^n

O(2^n)

The complexity is exponential.

```
public boolean groupSumClump(int [] nums, int target) {
return groupSumClump(0, nums, target);
}
private boolean groupSumClump(int start, int[] nums, int target) {
if(start >= nums.length) { //C1
return target == 0; //C2
} else { //C3
```

```
int countAdj = 1; //C4
int lastAdjIndex = 0; //C5
for(int i = start; i < nums.length - 1; i++) { //C6 * n
if(nums[i] == nums[i + 1]) { //C7 * n
countAdj++; //C8 * n
lastAdjIndex = i + 1; //C9 * n
}
}
int firstAdjIndex = lastAdjIndex - countAdj + 1; //C10
if(countAdj > 1 && start == firstAdjIndex) { //C11
int adjSum = nums[lastAdjIndex] * countAdj; //C12
return groupSumClump(start + countAdj, nums, target - adjSum) ||
groupSumClump(start + countAdj, nums, target); //C13
} else { //C14
return groupSumClump(start + 1, nums, target - nums[start]) ||
groupSumClump(start + 1, nums, target); //C15 + 2T(n-1)
}
}
}
```

T(n) = C1 + C2 si start>= nums.length

T(n)= C15+ 2T(n-1)

T(n) = C*2^n + C'

T(n) = 2^n

O(2^n)

The complexity is exponential.

```
public boolean splitArray(int[] nums) {
return splitArray(0, nums, 0, 0);
}
private boolean splitArray(int start, int [] nums, int sum1, int sum2) {
if(start == nums.length) { //C1
return sum1 == sum2; //C2
} else { //C3
return splitArray(start + 1, nums, sum1 + nums[start], sum2) ||
splitArray(start + 1, nums, sum1, sum2 + nums[start]); //C4 + 2T(n-1)
```

PROFESSOR MAURICIO TORO BERMÚDEZ
Phone: (+57) (4) 261 95 00 Ext. 9473. Office: 19 - 627
E-mail: mtorobe@eafit.edu.co

}

}

Complexity:

T(n) = C1 + C2 if start== nums.length

T(n)= C4 +2T(n-1) On the contrary

T(n) = C*2^n + C'

T(n) = 2^n

O(2^n)

3) Practice for final project defense presentation

1. The Stack Overflow it's an exception that occurs when the base case of a recursive method it's not well written or has some problem, then the stack, which is where the variables are stored, will eventually fill up. Stack Overflow can also occur because the variables of our methods are too big to be stored.
2. The method can't be executed with a million because the number it would return would be too big to be stored in an *int*, and to be stored on the stack. It wouldn't even work using *BigInteger*.
3. By increasing the heap and the memory capacity of the stack. It'd be useful, as well, to run the method as BigInteger
4. The problems taken from Recursion-1, they all have linear or logarithmic complexity; the ones from Recursion-2 have exponential complexity; but those are algorithms much more complex, so it's understandable that they take more time and memory to be completed, and it's not due to a lack of efficiency from the programmer.

4) Practice for midterms

1. *start + 1, nums, target*
2. *a*
3. **…**
4. *e*
5. ***5.1:*** *return n; n-1; n-2* ***5.2:*** *b*
6. ***6.1:*** *sumaAux(n, i+2)* ***6.2:*** *sumaAux(n, i+1)*
7. ***7.1:*** *S, i+1, t* ***7.2:*** *S, i+1, t-S[i]*

PROFESSOR MAURICIO TORO BERMÚDEZ
Phone: (+57) (4) 261 95 00 Ext. 9473. Office: 19 - 627
E-mail: mtorobe@eafit.edu.co