	<p style="text-align: center;">UNIVERSIDAD EAFIT ESCUELA DE INGENIERÍA DEPARTAMENTO DE INFORMÁTICA Y SISTEMAS</p>	<p>Código: ST245</p>
		<p>Estructura de Datos 1</p>

Laboratorio Nro. 1: Recursión

Luis Javier Palacio Mesa

Universidad Eafit
Medellín, Colombia
ljpalaciom@eafit.edu.co

Santiago Castrillon Galvis

Universidad Eafit
Medellín, Colombia
scastrillg@eafit.edu.co

Kevyn Santiago Gómez Patiño

Universidad Eafit
Medellín, Colombia
ksgomezp@eafit.edu.co

2)

2.3

Expliquen con sus propias palabras cómo funciona el ejercicio GroupSum5

Tenemos tres parámetros: un entero start que nos sirve como índice para movernos en nums que es el arreglo y un target que es el número que nos sirve como objetivo. La condición de parada se da cuando el start ya superó al último elemento del arreglo y es allí donde retornamos si el objetivo se cumplió o no. Durante el proceso se hacen llamados recursivos en los que para elegir un número del arreglo se resta al target y así, al final esto significa que target es igual a cero cuando un subgrupo al sumarse cumple el objetivo.

Si no entra a la condición de parada, se pregunta si el número que se está revisando con start es divisible por 5, si es de esta manera hacemos una pregunta que nos ayudara a determinar si el número siguiente es igual a uno y de ser así aumentamos dos a el índice para que el próximo llamado recursivo no elija el uno. Si no es divisible por 5 entonces se hacen dos llamados usuales del groupSum común, un llamado que le reste a el target y otro que no.

DOCENTE MAURICIO TORO BERMÚDEZ

Teléfono: (+57) (4) 261 95 00 Ext. 9473. Oficina: 19 - 627 Correo:

mtorobe@eafit.edu.co

2.4

Calculen la complejidad de los Ejercicios en Línea de los numerales 2.1 y 2.2

```
1. public boolean strCopies(String str, String sub, int objetivo) {  
    if (str.startsWith(sub)) { // C1          objetivo--; //C2  
        }  
        if(str.length() == 0){ // C3  
    return objetivo == 0; //C4  
    }  
        return strCopies(str.substring(1), sub, objetivo); //T(n-1)  
    }  
T(n) = C1 + C2 + C3 + C4 si str.length() == 0 y str.startsWith(sub)  
T(n) = C1 + C2 + T(n-1), si str.length != 0  
T(n) = C1 + Cn  
T(n) es O(C1 + Cn)  
T(n) es O(Cn)  
T(n) es O(n)
```

```
2. public int strCount(String str, String sub) {  
    if (str.length() < sub.length()) { //C1  
        return 0; C2  
    }  
        if(str.startsWith(sub)) return 1 + strCount(str.substring(sub.length()),  
sub); //C3 T(n-1)  
        return strCount(str.substring(1), sub); // C4 + T(n-1)  
    }  
T(n) = C1 + C2 si str.length() < sub.length  
T(n) = C + T(n-1) de lo contrario
```

$T(n) = C1 + Cn$
 $T(n)$ es $O(C1 + Cn)$
 $T(n)$ es $O(Cn)$
 $T(n)$ es $O(n)$

```
3. public String parenBit(String str) { if(str.startsWith("("))
    return str.substring(0, str.indexOf(")") + 1); //C1 +C2    return
    parenBit(str.substring(1)); //C3 +T(n-1)
}
```

$T(n) = C1 + C2$ si $str.startsWith("(")$
 $T(n) = C + T(n-1)$ de lo contrario
 $T(n) = C1 + Cn$
 $T(n)$ es $O(C1 + Cn)$
 $T(n)$ es $O(Cn)$
 $T(n)$ es $O(n)$

```
4. public String stringClean(String str) { if(str.length() == 1)
    return str; // C1 + C2
    if(str.charAt(0) == str.charAt(1)) return stringClean(str.substring(1)); //C3 +
    T(n-1)
    return str.charAt(0) + stringClean(str.substring(1)); //C3 + T(n-1)
}
```

$T(n) = C1 + C2$ $str.length() == 1$
 $T(n) = C + T(n-1)$ de lo contrario
 $T(n) = C1 + Cn$
 $T(n)$ es $O(C1 + Cn)$
 $T(n)$ es $O(Cn)$
 $T(n)$ es $O(n)$

```
5. public int array11(int[] nums, int index) { int cont = 0; // C1
    if(nums.length == index) return 0; //C2 + C3    if(nums[index]
    == 11) cont++; //C4 + C5
    return cont + array11(nums,index+1); //T(n-1)
```

```
}  
T(n) = C1 + C2 + C3 si nums.length == index  
T(n) = C + T(n-1) de lo contrario  
T(n) = C1 + Cn  
T(n) es O(C1 + Cn)  
T(n) es O(Cn)  
T(n) es O(n)
```

```
6. public boolean groupSum6(int start, int[] nums, int target) {  
    if (start == nums.length) { // C1  
        return target == 0; // C2  
    }  
    return nums[start] == 6 ? groupSum6(start + 1, nums, target - 6) //C3 +  
T(n-1)  
        : groupSum6(start + 1, nums, target - nums[start])  
        || groupSum6(start + 1, nums, target); // C + T(n-1) + T(n-1)  
}  
T(n) = C1 + C2 start == nums.length  
T(n) = C + T(n-1) + T(n-1) de lo contrario (peor caso)  
T(n) = C*2^n + C'  
T(n) es O( C*2^n + C'  
T(n) es O( C*2^n )  
T(n) es O( 2^n )
```

```
7. public boolean groupSum5(int start, int[] nums, int target) {  
    if (start == nums.length) { // C1  
        return target == 0; // C2  
    }  
    if (nums[start] % 5 == 0) { //C3  
        if (start < nums.length - 1 && nums[start + 1] == 1) { //C4  
return groupSum5(start + 2, nums, target - nums[start]); // T(n-2)  
        }  
        return groupSum5(start + 1, nums, target - nums[start]); //T(n-1)
```

```
    }  
    return groupSum5(start + 1, nums, target - nums[start])  
|| groupSum5(start + 1, nums, target); T(n-1) + T(n-1)  
}
```

$T(n) = C1 + C2$ start == nums.length
 $T(n) = C + T(n-1) + T(n-1)$ de lo contrario (peor caso)
 $T(n) = C \cdot 2^n + C'$
 $T(n)$ es $O(C \cdot 2^n + C')$
 $T(n)$ es $O(C \cdot 2^n)$
 $T(n)$ es $O(2^n)$

```
8. public boolean splitArray(int[] nums) {  
    return auxiliarSplitArray(0, nums, 0, 0);  
}  
public boolean auxiliarSplitArray(int start, int[] nums, int suma1, int suma2)  
{  
    return nums.length == start ? suma1 == suma2 // C1 + C2  
        : auxiliarSplitArray(start + 1, nums, suma1 + nums[start], suma2)  
        || auxiliarSplitArray(start + 1, nums, suma1, suma2 + nums[start]);  
//T(n-1) + T(n-1)  
  
}
```

$T(n) = C1 + C2$ start == nums.length
 $T(n) = C + T(n-1) + T(n-1)$ de lo contrario (peor caso)
 $T(n) = C \cdot 2^n + C'$
 $T(n)$ es $O(C \cdot 2^n + C')$
 $T(n)$ es $O(C \cdot 2^n)$
 $T(n)$ es $O(2^n)$

```
9. public boolean split53(int[] nums) {  
    return splitAux(0,nums,0,0);  
}  
public boolean splitAux(int start,int [] nums, int sum1, int sum2){  
if(start == nums.length) return sum1 == sum2; // C1 + C2
```

```
if(nums[start] % 5 == 0){
    return splitAux(start + 1, nums, sum1 + nums[start], sum2); //C3 + T(n-1)
} else if(nums[start] % 3 == 0){
    return splitAux(start + 1, nums, sum1, sum2 + nums[start]); //C4 + T(n-1)
}
return splitAux(start + 1, nums, sum1 + nums[start], sum2) ||
splitAux(start + 1, nums, sum1, sum2 + nums[start]); // T(n-1) + T(n-1)

}
T(n) = C1 + C2 start == nums.length
T(n) = C + T(n-1) + T(n-1) de lo contrario (peor caso)
T(n) =  $C \cdot 2^n + C'$ 
T(n) es  $O(C \cdot 2^n + C')$ 
T(n) es  $O(C \cdot 2^n)$ 
T(n) es  $O(2^n)$ 
```

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

}
```

$T(n) = C1 + C2 \text{ start} \geq \text{nums.length}$
 $T(n) = C + T(n-1) + T(n-1)$ de lo contrario (peor caso)
 $T(n) = C \cdot 2^n + C'$
 $T(n)$ es $O(C \cdot 2^n + C')$
 $T(n)$ es $O(C \cdot 2^n)$
 $T(n)$ es $O(2^n)$

2.5

Expliquen con sus palabras las variables (qué es 'n', qué es 'm', etc.) del cálculo de complejidad del numeral anterior


Hemos visto diferentes variables que nos sirven a la hora de usar recursión. Todo depende del problema y de sus necesidades:

1. `str.length()`: La mejor manera que encontramos para hacer recursión con strings fue reduciendo la cadena en cada llamado gracias al substring. De esta manera en algún momento la cadena no tendría longitud y es ahí cuando se llega al caso base. Así, si llamamos **n** a la longitud del string, el problema se va reduciendo cada vez con $T(n-1)$
2. `start` y `nums.length()`: Puesto que modificar un arreglo puede ser engorroso, preferimos usar una variable que nos sirva de índice, así pues al sumarle uno a este índice lo que estamos diciendo es que ya nos faltan menos elementos por analizar, de modo que si **n** es la longitud del arreglo, el problema se va reduciendo cada vez con $T(n-1)$
- 3.`str`: En los problemas con string, esta era una cadena dada que había que analizar.
4. `sub`: En algunas ocasiones nos daban esta variable para ver si el string principal la contenía dentro con ciertas condiciones.
- 5.`target`: En los ejercicios de recursión este era un entero dado. Nuestro trabajo era verificar si algún subgrupo del arreglo a veces bajo algunas condiciones, al sumarse era igual al target.
6. `Sum1` y `Sum2`: Estas variables servían para llevar la suma de los subgrupos de un arreglo y si al final dos subgrupos sumaban lo mismo se retornaba un valor verdadero o falso de lo contrario.
7. `n`: En `strCopies` se preguntaba si un string aparecía **n** veces en otro, así que la **n** nos servía como contador.


3) Simulacro de preguntas de sustentación de Proyectos

3.1

N	R ArraySum	R ArrayMax	N	Fibonacci
100000	8	7	20	2

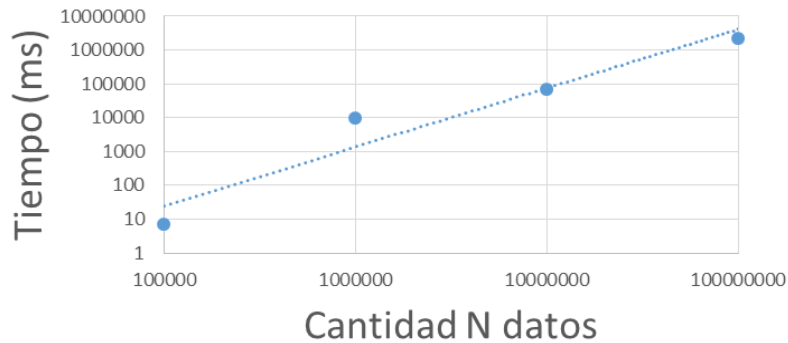
	UNIVERSIDAD EAFIT ESCUELA DE INGENIERÍA DEPARTAMENTO DE INFORMÁTICA Y SISTEMAS			Código: ST245
				Estructura de Datos 1

1000000	98	9459	30	29
10000000	1286	71915	40	3509
100000000	8403	2211830	50	330610

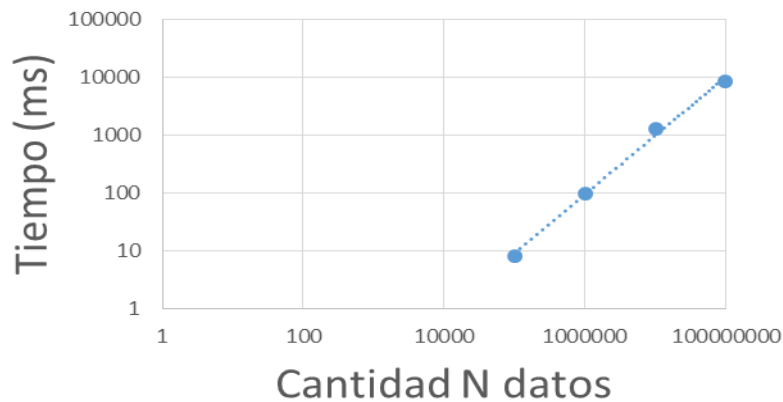
	UNIVERSIDAD EAFIT ESCUELA DE INGENIERÍA DEPARTAMENTO DE INFORMÁTICA Y SISTEMAS	Código: ST245
		Estructura de Datos 1

3.2

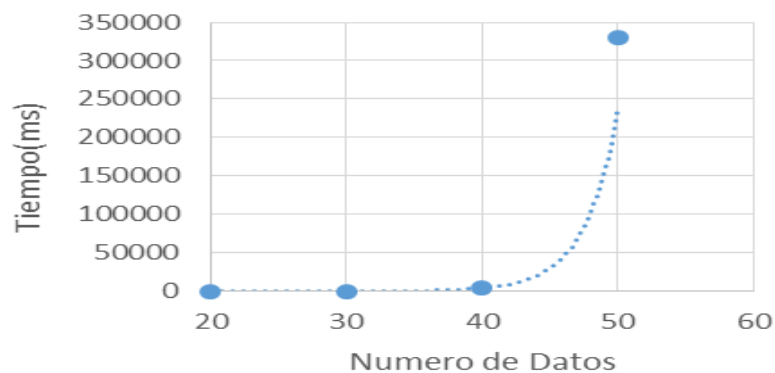
R ArrayMax



R ArraySum



Fibonacci



3.3

Concluimos que los datos teóricos respecto a nuestros datos experimentales son adecuados y con muy poco margen de error. Vemos que las gráficas de ArraySum y ArrayMax son lineales y Fibonacci es un función creciente no lineal, justo como la teoría indica.

3.4

El Stack Overflow en este caso se producía por que hacíamos llamados recursivos para muchos datos lo cual llenaba la pila y esta no podía seguir hasta que no le diéramos más espacio para poder ejecutar las cantidades de datos tan altas.

3.5

El valor más grande que logramos calcular fue de Fibonacci (50) ya que si poníamos un valor más se hubiese tomado un gran tiempo. Fibonacci de 1000000 tomaría mucho tiempo en acabar ya que su complejidad es $O(n^2)$.

3.6

Al hacer Fibonacci con ciclos podemos hacer número muy grandes puesto que la complejidad es $O(n)$.

```
public static void fibonacci() {  
    System.out.println("Ingrese el numero hasta el que desea hacer la Sucesión de  
    Fibonacci");  
    double n = leerInt();  
    double num1 = 0;  
    double num2 = 1;  
    double numeroNuevo;  
  
    System.out.print(num1 + "," + num2);  
    for (int i = 1; i <= n - 2; i++) {  
        numeroNuevo = num1 + num2;  
        System.out.print(",");  
  
        System.out.print(numeroNuevo);  
        num1 = num2;        num2 =  
        numeroNuevo;  
    }  
    System.out.println();  
}
```

3.7

¿Qué concluyen sobre la complejidad de los problemas de CodingBat Recursion 1 con respecto a los de Recursion 2?

Puesto que en los ejercicios que hicimos de recursión 1 sólo hacíamos de a un llamado recursivo reduciendo el problema con $T(n-1)$, todos estos ejercicios tuvieron un orden de $O(n)$. Por otro lado, en recursión 2, escogiendo siempre el peor caso hacíamos dos llamados recursivos, y de esta manera obtuvimos un orden de $O(n^2)$.

4) Simulacro de Parcial

1. start +1, nums,target
2. a
3. int res = solucionar(n - a, a, b, c) + 1; res = Math.max(res, solucionar(n - b, a, b, c) + 1); res = Math. max(res, solucionar(n -c, a, b, c) + 1);
4. e