Laboratorio Nro. 1: Recursión

Santiago Escobar Mejía
Universidad Eafit
Medellín, Colombia
sescobarm@eafit.edu.co

Sebastián Giraldo Gómez
Universidad Eafit
Medellín, Colombia
sgiraldog@eafit.edu.co

Luisa María Vásquez
Universidad Eafit
Medellín, Colombia
lmvasquez@eafit.edu.co

2) Ejercicios en línea sin documentación HTML en GitHub

2.3) Explicación groupSum5

La solución al ejercicio es:

```
public boolean groupSum5(int start, int[] nums, int target) {
    if(start>=nums.length){
        return target==0;
    }else{
        if(nums[start]%5==0){
            if(nums[start+1]==1){
                return groupSum5(start+2,nums,target-nums[start]);
            }else{
                return groupSum5(start+1,nums,target-nums[start]);
            }
        }else{
            return
groupSum5(start+1,nums,target)||groupSum5(start+1,nums,target-nums[start]);
        }
    }
}
```

El ejercicio recibe un parámetro de control “start”, un arreglo con los valores y un número “target” a el cual se le quiere buscar un subconjunto que sume dicho número.

El método evalúa si el elemento en la posición “start” es un múltiplo de 5, si lo es evalúa si el siguiente es un 1, si lo es le resta a el “target” el número 5 y le suma 2 a el “start” con el fin de evitar evaluar el 1; si no es múltiplo de 5 el método hace un llamado recursivo restando el número en “start” y otro sin restarlo.

2.4) Complejidad ejercicios Recursión 1 y 2

Recursión 1

```
public int triangle(int rows) {                                  // complejidad del
algoritmo  O(N)
      if (rows == 0) {                                            // (1)
         return 0;
      } else {                                                    // (1)
         return rows + triangle(rows - 1);                       // triangle(rows -1)
+ 1
      }
   }
```

```
public String noX(String str) {        // complejidad del algoritmo O(N)
      if (str.length() == 0) {
         return "";
      } else {
         if (str.charAt(0) == 'x') {
            return noX(str.substring(1));                          // T(n-1)
         } else {                                                 //(1)
            return str.charAt(0) + "" + noX(str.substring(1));    // T(n-1)
         }
      }
   }
```

DOCENTE MAURICIO TORO BERMÚDEZ
Teléfono: (+57) (4) 261 95 00 Ext. 9473. Oficina: 19 - 627
Correo: mtorobe@eafit.edu.co

```
  public int powerN(int base, int n) {                    //Complejidad del algoritmo O(N)
      if (n == 0) {                                        //(1)
         return 1;
      } else {                                             //(1)
         return base * powerN(base, n - 1);               //  Cambio T(n-1) + C se ejecuta N
      }
   }
```

```
  public String changePi(String str) {
                                                          // Complejidad del metodo O(N)
      if (str.length() == 0) {                                    // (1)
         return "";
      } else if (str.length() == 1) {                             //(1)
         return str.charAt(0) + "";
      } else {    //(1)
         if ((str.charAt(0) + "" + str.charAt(1)).equals("pi")) {    //(1)
            return "3.14" + changePi(str.substring(2));
         } else {
            return str.charAt(0) + changePi(str.substring(1));    // T(N - 1), en su peor ejecución.
         }
      }
   }
```

```
  public int countHi(String str) {
                                                          // Complejidad del metodo O(N)
      if (str.length() == 1 || str.length() == 0) {
         return 0;
      } else {
```

DOCENTE MAURICIO TORO BERMÚDEZ
Teléfono: (+57) (4) 261 95 00 Ext. 9473. Oficina: 19 - 627
Correo: mtorobe@eafit.edu.co

```
        if ((str.charAt(0) + "" + str.charAt(1)).equals("hi")) {
            return 1 + countHi(str.substring(1));                    //T(n-1)
        } else {
            return 0 + countHi(str.substring(1));                    //T(n-1)
        }
    }
}
```

Recursión 2

```
public boolean groupNoAdj(int start, int[] nums, int target) {
    if (start >= nums.length) {                                      // (1)
        return target == 0;
    } else {                                                         //(1)
        return groupNoAdj(start + 1, nums, target) ||
       groupNoAdj(start + 2, nums, target - nums[start]);
                                                                     // 2 recursiones
una con cambio de T(n-1) y la otra con cambio de T(n-2)
    }
                                                                     // por lo que en
su peor caso seria con cambio T(n-1), eso deja una complejidad de
  }
                                                                       // a = 2
recursiones , T(n-1) ,donde a es mayor a 1 por lo que 2T(n-1)+C seria una
complejidad de  O(2^N)
```

```
public boolean groupSum5(int start, int[] nums, int target) {
```

// la complejidad del algoritmo es de **O(2^N),** con a = 2 en el pero de los casos, y cambio de T(N-1) da como complejidad 2*T(n-1)+c con complejidad O(2^N)

```
    if (start >= nums.length) {                                    //(1)
        return target == 0;
    } else {                                                       //(1)
        if (nums[start] % 5 == 0) {                                //(1)
            if (start + 1 < nums.length && nums[start + 1] == 1) {  //(1)
                return groupSum5(start + 2, nums, target - nums[start]);   // N/8
            } else {                                                        // (1)
                return groupSum5(start + 1, nums, target - nums[start]);   //N/4
            }
        } else {                                                           //(1)
            return groupSum5(start + 1, nums, target) ||
           groupSum5(start + 1, nums, target - nums[start]);               // 2(N/2)
        }
    }
}
```

```
public boolean splitArray(int[] nums) {                          // complejidad del metodo (1)
    return aiuda(0, nums, 0, 0);                                 //(1)
}

private boolean aiuda(int ini, int a[], int sum1, int sum2) {    // O(2^N)
    if (ini >= a.length) {                                       // (1)
        return sum1 == sum2;
    } else {                                                     //(1)
        return aiuda(ini + 1, a, sum1 + a[ini], sum2) ||
aiuda(ini + 1, a, sum1, sum2 + a[ini]);                          // la complejidad del algoritmo es de O(2^N), con a = 2 en el peor de los casos donde a es el numero de
```

recursiones ejecutadas en el peor caso , y cambio de T(N-1) da como complejidad 2*T(n-1)+c con complejidad **O(2^N)**

```
        }
    }
```

```
public boolean splitOdd10(int[] nums) {              // complejidad del metodo (1)
        return aiuda(0, nums, 0, 0);                 //(1)
    }

    private boolean aiuda2(int ini, int a[], int sum1, int sum2) {       // O(2^N)
        if (ini >= a.length) {                                           //(1)
            return sum1 % 10 == 0 && sum2 % 2 != 0;
        } else {                                                                  // (1)
            return aiuda(ini + 1, a, sum1 + a[ini], sum2) ||
 aiuda(ini + 1, a, sum1, sum2 + a[ini]);
                                                                        // la complejidad del
```

algoritmo es de **O(2^N)**, con a = 2 en el pero de los casos donde a es el numero de recurciones ejecutaas en el peor caso , y cambio de T(N-1) da como complejidad 2*T(n-1)+c con complejidad O(2^N)

```
        }
    }
```

```
  public boolean split53 (int[] nums) {                  // complejidad del metodo (1)
        return aiuda(0, nums, 0, 0);                      //(1)
    }

    private boolean aiuda3(int ini, int a[], int sum1, int sum2) {
                                                                        // la complejidad del
```

algoritmo es de O(2^N), con a = 2 en el pero de los casos, y cambio de T(N-1) da como complejidad 2*T(n-1)+c con complejidad **O(2^N)**

DOCENTE MAURICIO TORO BERMÚDEZ
Teléfono: (+57) (4) 261 95 00 Ext. 9473. Oficina: 19 - 627
Correo: mtorobe@eafit.edu.co

```
        if (ini >= a.length) {                                  //(1)
            return sum1 == sum2;
        } else {                                                 //(1)
            if (a[ini] % 3 == 0) {                              //(1)
                return aiuda(ini + 1, a, sum1 + a[ini], sum2);  //N/3
            } else if (a[ini] % 5 == 0) {//(1)
                return aiuda(ini + 1, a, sum1, sum2 + a[ini]);  //N/3
            } else {                                            //(1)
                return aiuda(ini + 1, a, sum1 + a[ini], sum2) ||
                 aiuda(ini + 1, a, sum1, sum2 + a[ini]);
                                                                 // peor caso
complejidad 2(N/3) pero con margen de cambio de 2*T(n-1)
            }
        }
    }
```

2.5) Expliquen con sus palabras las variables (qué es 'n', qué es 'm', etc.) del cálculo de complejidad del ejercicio 2.4

En la notación O(n) , "n" se utiliza como una variable de referencia que indica la complejidad mediante el número y tipo de ejecuciones, un ejemplo de esto puede ser un algoritmo de que diga los números de n hasta 1, este algoritmo se ejecutaría n-1 veces.

Pero además de la n existen otros valores como:

a* T(n **op c**) + **C**

-a: Que indica el número de llamadas recursivas que se generan a partir de cada llamada recursiva.

-op c: Expresa cómo decrece el problema:

-Aritméticamente (restando una unidad) -1.

-Geométricamente (dividiendo) /2.

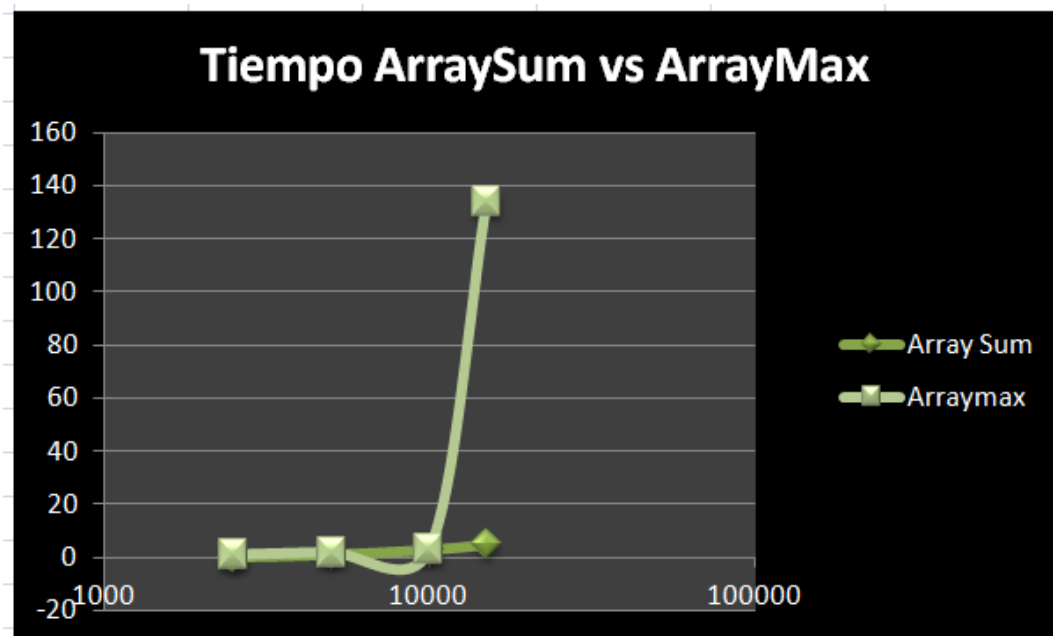
-**Constante:** Indica el coste de acciones realizadas por el método sin considerar la llamada recursiva.

3) Simulacro de preguntas de sustentación de Proyectos

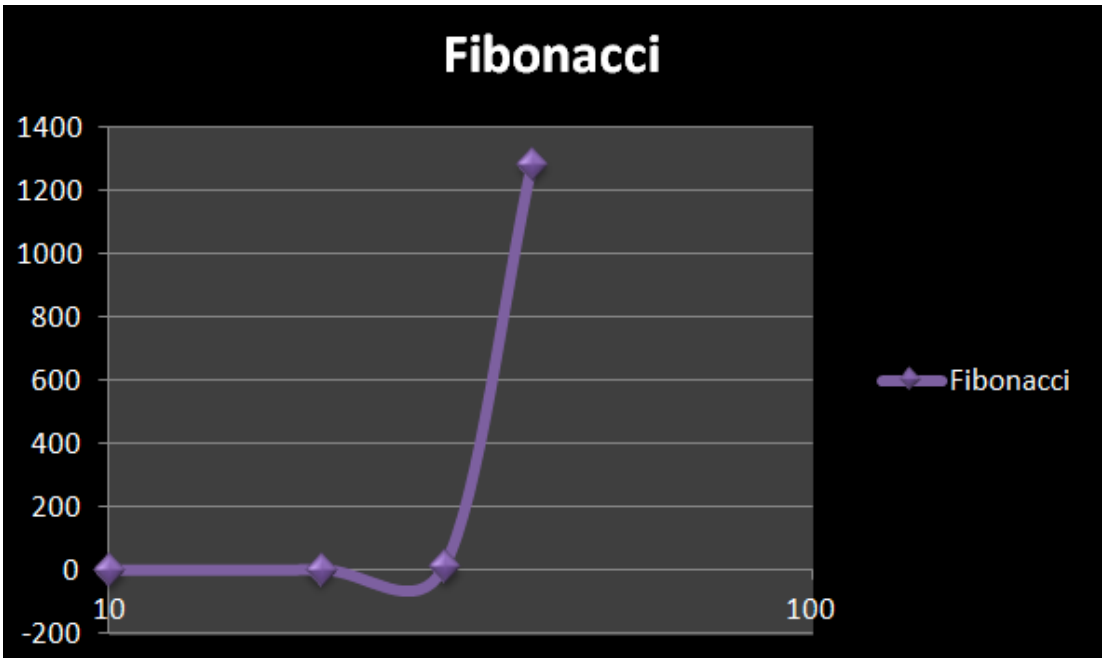
3.1 y 3.2)

	N=2500	N=5000	N=10000	N=15000
R ArraySum	0	1	3	5
R Array maximun	1	2	3	134

	N=10	N=20	N=30	N=40
Fibonacci	0	1	12	1284



DOCENTE MAURICIO TORO BERMÚDEZ
Teléfono: (+57) (4) 261 95 00 Ext. 9473. Oficina: 19 - 627
Correo: mtorobe@eafit.edu.co



3.3) ¿Qué concluyen respecto a los tiempos obtenidos en el laboratorio y los resultados teóricos?

Después de comparar los resultados obtenidos con las gráficas de complejidad de la notación O(n) podemos concluir lo siguiente de cada uno de los métodos:

- Fibonacci: La grafica de los tiempos de ejecución de este metodo se asemeja a la de la complejidad exponencial, es decir a la gráfica de O(2^n).
- ArraySum: La grafica de los tiempos de ejecución de este metodo se asemeja a la de la complejidad lineal, es decir a la gráfica de O(n).
- ArrayMax: La grafica de los tiempos de ejecución de este metodo se asemeja a la de la complejidad exponencial, es decir a la gráfica de O(n^2).

Debido a esto, el método que tiene la complejidad más baja y que consume menos recursos es el ArraySum, con su complejidad lineal de O(n).

Además también podemos concluir que hay factores que afectan la relación de los datos obtenidos con los teóricos, como lo puede ser lentitud de la máquina en un lapso de tiempo, llamados a otros métodos como “Math.max ó Math.random”, entre otros.

3.4) ¿Qué aprendieron sobre Stack Overflow?

El Stack Overflow(desbordamiento de pila) es un problema que ocurre debido a una llamada recursiva incorrecta, en otras palabras, un proceso llama a otro subproceso que sigue llamando más subprocesos sin parar, debido a que no tiene una condición
de parada correcta, lo que genera un desbordamiento en el Stack (pila) al sobrepasar el tope de memoria reservada para este.

Un ejemplo más claro para esto es cuando tenemos un método con llamados recursivos, como este:

```
public int factorial(int n){
  return n*factorial(n-1);
 }
```

Este método nos genera un Stack Overflow, debido a que no tiene una condición de parada que haga que deje de llamarse a si mismo. La solución a este problema es escribir la condición de parada de dicho método:

```
public int factorial(int n){

 if(n<=1){
       return 1;
 }
 return n*factorial(n-1);
 }
```

3.5) Cuál es el valor más grande qué pudo calcular para Fibonnacci? ¿Por qué? ¿Por qué no se puede ejecutar Fibonacci con 1 millón?

El valor que máximo que se puede calcular es muy variable debido a la capacidad y configuración de la máquina y entorno que ejecuta el método, por lo que es difícil decir un número exacto en el que se deja de poder calcular.
Esto pasa debido a que usando la recursión para calcular el Fibonacci se hacen muchos sub-problemas a la vez, por ejemplo con el Fibonacci(3) se debe hacer el Fibonacci (2) y el Fibonacci(1) , y para el Fibonacci(2) se debe hacer el Fibonacci(1) y el Fibonacci(0).

Por lo que a medida que el problema se vaya haciendo cada vez más grande va a consumir más recursos, va a tener más ejecuciones y va a demorar más

tiempo, lo que poco a poco va impidiendo su ejecución.

3.6) ¿Cómo se puede hacer para calcular el Fibonacci de valores grandes?

La mejor forma de calcular el Fibonacci no solo en valores grandes es usar la programación dinámica, la cual busca almacenar el resultado en sub-problemas para que no se deba calcular de nuevo.

```
public int fibDP(int x) {
                int fib[] = new int[x + 1];
                fib[0] = 0;
                fib[1] = 1;
                for (int i = 2; i < x + 1; i++) {
                        fib[i] = fib[i - 1] + fib[i - 2];
                }
                return fib[x]; }
```

De esta forma pasa de **$O(2^n)$ a O(n)**

Esto pasa debido a que usando la recursión para calcular el Fibonacci se hacen muchos sub-problemas a la vez, por ejemplo con el Fibonacci(3) se debe hacer el Fibonacci (2) y el Fibonacci(1) , y para el Fibonacci(2) se debe hacer el Fibonacci(1) y el Fibonacci(0).

En cambio al usar este método de programación, cada resultado se almacena y una vez necesitado se accede a él, teniendo así menos ejecuciones.

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

Al comparar la complejidad de los métodos de Recursión 1 y Recursión 2, la diferencia más notable es que los métodos de Recursión 1 tienen complejidad O(n) y los de Recursión 2 tiene complejidad O(n^2). Debido a esto surge la pregunta, ¿Por qué se da ese cambio de complejidad?

La respuesta a esto es simple, los métodos de Recursión 1 en el peor de los casos tienen un solo llamado recursivo, por ejemplo:

```
return rows + triangle(rows - 1);
```

En cambio, en los métodos de Recursión 2, en el peor de los casos tienen dos llamados recursivos, pueden tener distintos parámetros, pero al fin y al cabo llegan a la misma complejidad O(n^2), por ejemplo:

return groupSum5(start + 1, nums, target) || groupSum5(start + 1, nums, target - nums[start]);

Ahí radica la principal diferencia de complejidad, en la cantidad de llamados recursivos que se deben realizar en el peor de los casos respectivamente.

4) Simulacro de Parcial

1) ¿Qué parámetros colocaría en el llamado recursivo de la línea 4 para que el programa funcione?

R//= start + 1, nums, target

2) ¿Cuál ecuación de recurrencia describe el comportamiento del algoritmo anterior para el peor de los casos?

R//= a) T(n)=T(n/2)+C

3)

3.1) Complete el espacio de la línea 04 :

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

3.2) Complete los espacios de la línea 05 :

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

3.3) Complete los espacios de la línea 06 :

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

4) ¿Qué calcula el algoritmo desconocido y cuál es la complejidad asintótica en el peor de los casos del algoritmo desconocido?

e) La suma de los elementos del arreglo a y es O(n)