ESTRUCTURA DE DATOS 1
Código ST0245

Laboratorio Nro. 1
Recursión

Alejandro Villada Toro
Universidad Eafit
Medellín, Colombia
avilladat@eafit.edu.co

Cristian Alzate Urrea
Universidad Eafit
Medellín, Colombia
calzateu@eafit.edu.co

3) Simulacro de preguntas de sustentación de Proyectos

3.1
Complejidad del ejercicio 1.1
La complejidad esta dada por la función T(n,m), donde n es la longitud de la primera de cadena de caracteres, mientras que m es la longitud de la segunda, C_n son cantidades constantes de operaciones.
C_1=7 C_2=12 C_3=4

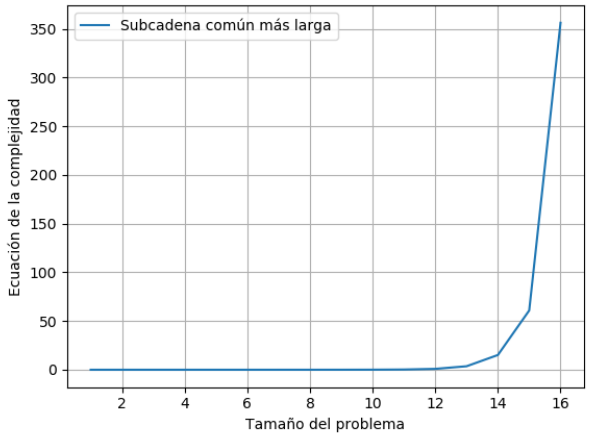
$$T(n, m) = \begin{cases} C_1 & \text{cuando } n=0 \text{ o } m=0 \\ C_2 + T(n-1, m-1) & \\ C_3 + T(n-1, m) + T(n, m-1) & \text{cuando } n>0 \text{ y } m>0 \end{cases}$$

La función recursiva es bastante difícil de encontrar con lo anterior, pero podemos expresar T en función de una sola variable así T(s) siendo *s* igual a la suma de las longitudes es decir *n+m.* De esta manera el peor caso nos queda $T(s) = C_3 + 2(T(s-1))$ y asi la función recursiva es:
$T(s) = C_2(2^s-1) + C_1 2^{s-1}$
Aplicando la notación O
$T(s)$ es $O(C_2(2^s-1) + C_1 2^{s-1})$
Aplicando reglas de suma y producto
$T(n)$ es $O(2^s)$

Y al reemplazar *s* tenemos que $T(n, m)$ es $O(2^{n+s})$ □

3.2
Nota: No fue posible tomar los tiempos con 20 tamaños diferentes, solo se pudo con 16.
Al tomar los tiempos para 16 tamaños diferentes del ejercicio 1.1, se obtiene la siguiente gráfica:

PhD. Mauricio Toro Bermúdez
Docente | Escuela de Ingeniería | Informática y Sistemas
Correo: mtorobe@eafit.edu.co | Oficina: Bloque 19 – 627
Tel: (+57) (4) 261 95 00 Ext. 9473



Como se observa claramente, se genera una función exponencial, por lo tanto, la complejidad del tiempo del problema está dada por una función del mismo tipo.

Una buena aproximación de la función de la complejidad del tiempo sería $O(2^n)$ siendo n la longitud de las cadenas; con esta fórmula tenemos el tiempo que tardaría con unas cadenas cuya longitud es de *300 000* es alrededor de $2^{300\,000}$, este valor es simplemente gigantesco.

3.3

Por lo que vimos en el punto anterior, no es para nada viable utilizar este algoritmo para calcular la longitud de la subcadena común mas larga entre dos cadenas cuya longitud es de alrededor de 300 000 caracteres. Concluimos, se debe implementar un mejor método para este cálculo.

3.4

GroupSum5

Dado un arreglo de enteros, el método *groupSum5* nos dice si es posible escoger un grupo de esos enteros, de manera que sumen un valor objetivo dado, con la condición de que si alguno de los elementos del arreglo es cinco, entonces debe ser escogido, excepto si el cinco esta seguido de un uno, en cuyo caso no deben ser escogidos; recibe los parámetros *nums,* que es un arreglo de enteros, *start* que es un entero y al principio siempre va a ser igual cero, y *target* que es un entero que representa el objetivo al que se debe llegar. El funcionamiento de la función es el siguiente: primero pregunta si *start* es mayor o igual que la longitud de *nums*, en caso de que si, retorna *true* si *target* es igual a cero, o *false* si pasa lo contrario; en caso de que no; existen cuatro posibilidades, la primera sucede si el elemento de *nums* en la posición *start* es múltiplo de 5 y además es el último elemento de nums, de este modo retorna un llamado recursivo de la función, con *start* mas uno, el mismo *nums,* y *target* menos el elemento de *nums* en la posición *start, groupSum5(start + 1, nums, target - nums[start].* La segunda sucede cuando el elemento en *start* de *nums* es múltiplo de cinco y el siguiente elemento es distinto de uno, así retorna un llamado recursivo de la función, con *start* más uno, el mismo *nums,* y *target* menos el elemento de *nums* en la posición *start, groupSum5(start + 1, nums, target - nums[start].* La tercera sucede solo cuando el elemento en *start* de *nums* es múltiplo de cinco, de esta manera retorna un llamado recursivo de la función, con *start* más dos, el mismo *nums,* y *target* menos el elemento de *nums* en la

PhD. Mauricio Toro Bermúdez
Docente | Escuela de Ingeniería | Informática y Sistemas
Correo: mtorobe@eafit.edu.co | Oficina: Bloque 19 – 627
Tel: (+57) (4) 261 95 00 Ext. 9473

posición *start, groupSum5(start + 2, nums, target-nums[start]).* El cuarto sucede cuando no se cumple ninguna de las condiciones anteriores, en este caso, retorna la disyunción entre dos llamados recursivos, el primer disyunto es el llamado de recursivo de la función con *start* más uno, el mismo *nums* y *target* menos el elemento en la posición *start* de *nums groupSum5(start + 1, nums, target - nums[start]);* el segundo disyunto es el llamado recursivo de la función con *start* más uno, el mismo *nums* y *target* sin ningún cambio *groupSum5(start + 1, nums, target).* El principio en el que se basa la función es que existe alguna manera de que la suma de un grupo de enteros sea igual a un valor objetivo, si y solo si, existe alguna manera de que el valor objetivo menos la suma de un grupo de enteros sea igual a cero.
target = a+b+c ←→ *target-a-b-c = 0.*

3.5
Ejercicios de Recursión 1 en CodingBat

BunnyEars

El método *BunnyEars* calcula la cantidad de orejas que hay en un número dado de conejos; recibe el parámetro *bunnies* que es un entero positivo y representa la cantidad de conejos a los que se le calculara el número de orejas. El método tiene el siguiente funcionamiento: primero verifica si la cantidad de conejos es menor o igual a 1, en el caso de que si, retorna la cantidad de conejos *bunnies* multiplicada por dos; en caso de que no, retorna el método con la cantidad de conejos menos uno y a esto se le suma dos *BunnyEars(bunnies-1) +2*. La idea en la que se basa el algoritmo es que el número de orejas de una cantidad de conejos dada es igual a dos más el número de orejas de la cantidad de conejos menos 1.

La complejidad del tiempo está dada por la función T(n), donde n es la cantidad de conejos y C_n son cantidades constantes de operaciones.
C_1=4 C_2=3
T(n) = |C_1 cuando n<=1
|C_2 + T(n-1) cuando n>1
La función de recursividad está dada por:
$T(n) = C_2n + C_1$
Aplicando la notación O
T(n) es $O(C_2n + C_1)$
Aplicando reglas de sumas y producto
T(n) es O(n)

BunnyEars2

El método *BunnyEars2* calcula la cantidad de orejas de un numero dado de conejos, con una particularidad, aquellos conejos que se encuentran en posiciones pares han sufrido una mutación y poseen tres orejas, por lo que la cantidad de orejas será mayor a lo común; al igual que *BunnyEars* recibe el parámetro *Bunnies* que es un numero entero positivo y representa el numero de conejos a los que se le calculará la cantidad de orejas. El método tiene un funcionamiento muy similar al de *BunnyEars,* con la diferencia de que a la hora de realizar el llamado recursivo comprueba si el conejo esta ubicado en una posición par o impar, en el caso de que sea par, suma 3 al llamado de la función *BunnyEars2(bunnies-1) +3;* en caso de que sea impar suma solo 2 *BunnyEars2(bunnies-1) +2.* El principio en el que se basa este algoritmo es el mismo que el de *BunnyEars.*

PhD. Mauricio Toro Bermúdez
Docente | Escuela de Ingeniería | Informática y Sistemas
Correo: mtorobe@eafit.edu.co | Oficina: Bloque 19 – 627
Tel: (+57) (4) 261 95 00 Ext. 9473

La complejidad del tiempo está dada por la función T(n), donde n es la cantidad de conejos y C_n son cantidades constantes de operaciones.
C_1=4 C_2=6 C_3=3
T(n) = |C_1 cuando n<=1
|C_2 + T(n-1) cuando n>1 y n%2==0
|C_3 + T(n-1) cuando n>1

La función de recursividad está dada por:
T(n) = Cn + C_1
Aplicando la notación O
T(n) es O(C_2n + C_1)
Aplicando reglas de sumas y producto
T(n) es O(n)

CountHi
El método *counHi* calcula el numero de veces que aparece *"hi"* en minúsculas dentro de una cadena dada; recibe el parámetro *str* que es una cadena de caracteres a la cual se le realizara el cálculo. El método tiene el siguiente funcionamiento: primero pregunta si la longitud de la cadena de caracteres es menor o igual a uno, en el caso de que si, retorna cero, en el caso de que no, continua preguntándose si los últimos dos elementos de la cadena son *'h'* e *'i',* en el caso de que si, retorna el llamado recursivo con *str* sin su último elemento y esto se le suma uno *countHi(str.substring(0,str.length()-1))+1*; en el caso de que no, solamente retorna el llamado recursivo con *str* sin su último elemento *countHi(str.substring(0,str.length()-1).* El principio en el que se basa el algoritmo es en que, en el caso de que los últimos dos elementos de la cadena sean igual *"hi",* la cantidad de *"hi"* en una cadena es igual uno mas la cantidad de *"hi"* en la cadena menos sus últimos dos elementos; en caso contrario, seria la cantidad de *"hi"* en la cadena menos su ultimo elemento.

La complejidad del tiempo está dada por la función T(n), donde n es la longitud de la cadena y C_n son cantidades constantes de operaciones.
C_1=4 C_2=14 C_3=3
T(n) = |C_1 cuando n<=1
|C_2 + T(n-1) cuando n>1

La función de recursividad está dada por:
T(n) = C_2n + C_1
Aplicando la notación O
T(n) es O (C_2n + C_1)
Aplicando reglas de sumas y producto
T(n) es O(n)

CountX
El método *countX* calcula el número de veces que aparece *'x'* en minúscula dentro de una cadena dada; recibe el parámetro *str* que es una cadena de caracteres a la cual se le realizara el cálculo. El método tiene el siguiente funcionamiento: primero pregunta si la longitud de la cadena de caracteres es igual a cero, en el caso de que si, retorna cero, en el caso de que no, continua preguntándose si el último elemento de la cadena es *'x'*, en el caso de que si,

PhD. Mauricio Toro Bermúdez
Docente | Escuela de Ingeniería | Informática y Sistemas
Correo: mtorobe@eafit.edu.co | Oficina: Bloque 19 – 627
Tel: (+57) (4) 261 95 00 Ext. 9473

retorna el llamado recursivo con *str* sin su último elemento y esto se le suma uno *countX(str.substring(0,str.length()-1))+1*; en el caso de que no, solamente retorna el llamado recursivo con *str* sin su último elemento *countX(str.substring(0,str.length()-1).* El principio en el que se basa el algoritmo es en que, en el caso de que el ultimo elemento de la cadena sea igual *'x'*, la cantidad de *'x'* en la cadena es igual uno más la cantidad de *'x'* en la cadena menos su último elemento; en caso contrario, sería la cantidad de *'x'* en la cadena menos su último elemento.

La complejidad del tiempo está dada por la función T(n), donde n es la longitud de la cadena y C_n son cantidades constantes de operaciones.
C_1=4 C_2= 9 C_3=3
T(n) = |C_1 cuando n==1
|C_2+ T(n-1) cuando n>1
La función de recursividad está dada por:
$T(n) = C_2n + C_1$
Aplicando la notación O
T(n) es $O(C_2n + C_1)$
Aplicando reglas de sumas y producto
T(n) es O(n)

Triangle
El método *Triangle* calcula el numero de cubos de un triangulo formado por cubos de un numero de filas dado; recibe el parámetro *rows* que es un entero positivo y representa la cantidad de filas del triángulo. El método tiene el siguiente funcionamiento: primero pregunta si la cantidad de filas es igual a cero, en ese caso retorna cero, si sucede lo contrario retorna la cantidad de filas más el llamado recursivo con el número de filas menos uno *triangle(rows-1)+rows*. El principio en el que se basa el algoritmo es que la cantidad de cubos de un triangulo de n filas es igual a n mas la cantidad de cubos de un triángulo en n-1 filas.

La complejidad del tiempo está dada por la función T(n), donde n es el número de filas del triángulo y C_n son cantidades constantes de operaciones.
C_1=3 C_2= 3
T(n) = |C_1 cuando n==0
|C_2 + T(n-1) cuando n>0

La función de recursividad está dada por:
$T(n) = C_2n + C_1$
Aplicando la notación O
T(n) es $O(C_2n + C_1)$
Aplicando reglas de sumas y producto
T(n) es O(n)

Ejercicios de Recursión 2 en CodingBat
GroupNoAdj
Dado un arreglo de enteros, el método *groupNoAdj* nos dice si es posible escoger un grupo de esos enteros, de manera que sumen un valor objetivo dado, con la condición de que si un entero dentro del arreglo es escogido el siguiente no debe ser escogido; recibe los

PhD. Mauricio Toro Bermúdez
Docente | Escuela de Ingeniería | Informática y Sistemas
Correo: mtorobe@eafit.edu.co | Oficina: Bloque 19 – 627
Tel: (+57) (4) 261 95 00 Ext. 9473

parámetros *nums,* que es un arreglo de enteros, *start* que es un entero y al principio siempre va a ser igual cero, y *target* que es un entero que representa el objetivo al que se debe llegar. El funcionamiento de la función es el siguiente: primero pregunta si *start* es mayor que la longitud de *nums* menos uno, en caso de que si, retorna *true* si *target* es igual a cero, o *false* si pasa lo contrario; en caso de que no, retorna la disyunción entre dos llamados recursivos, el primer disyunto es el llamado de recursivo de la función con *start* mas dos, el mismo arreglo de enteros *nums* y *target* menos el elemento en la posición *start* de nums *groupNoAdj(start + 2, nums, target - nums[start])*; el segundo disyunto es el llamado recursivo de la función con *start* mas uno, el mismo arreglo de enteros *nums* y *target* sin ningún cambio *groupNoAdj(start + 1, nums, target).* El principio en el que se basa la función es que existe alguna manera de que la suma de un grupo de enteros sea igual a un valor objetivo, si y solo si, existe alguna manera de que el valor objetivo menos la suma de un grupo de enteros sea igual a cero.
target = a+b+c ←→ *target-a-b-c = 0.*

La complejidad del tiempo está dada por la función T(n), donde n es la diferencia entre la longitud del arreglo de enteros y *start,* es decir, lo que le falta a start para ser igual a la longitud y C_n son cantidades constantes de operaciones.
C_1=6 C_2=6

$$T(n) = \begin{cases} C_1 & \text{cuando } n<=0 \\ C_2 + T(n-2) + T(n-1) & \text{cuando } n>0 \end{cases}$$

La función recursiva es bastante difícil de encontrar con lo anterior, pero una buena aproximación es $C_2 + 2(T(n-1))$, de esta manera la ecuación recursiva es:
$T(n) = C_2(2^n-1) + C_1 2^{n-1}$
Aplicando la notación O
$T(n)$ es $O(C_2(2^n-1) + C_1 2^{n-1})$
Aplicando reglas de suma y producto
$T(n)$ es $O(2^n)$

GroupSum6
Dado un arreglo de enteros, el método *groupSum6* nos dice si es posible escoger un grupo de esos enteros, de manera que sumen un valor objetivo dado, con la condición de que si alguno de los elementos del arreglo es seis entonces debe ser escogido; recibe los parámetros *nums,* que es un arreglo de enteros, *start* que es un entero y al principio siempre va a ser igual cero, y *target* que es un entero que representa el objetivo al que se debe llegar. El funcionamiento de la función es el siguiente: primero pregunta si *start* es mayor o igual que la longitud de *nums*, en caso de que si, retorna *true* si *target* es igual a cero, o *false* si pasa lo contrario; en caso de que no, existen dos posibilidades, la primera que el elemento en la posición *start* de *nums* es seis, lo que lleva a retornar un llamado recursivo de la función con *start* más uno, el mismo nums y target menos seis. La segunda es si el elemento analizado es distinto de seis, en ese caso, retorna la disyunción entre dos llamados recursivos, el primer disyunto es el llamado de recursivo de la función con *start* más uno, el mismo *nums* y *target* menos el elemento en la posición *start* de *nums groupSum6(start + 1, nums, target - nums[start])*; el segundo disyunto es el llamado recursivo de la función con *start* más uno, el mismo *nums* y *target* sin ningún cambio *groupSum6(start + 1, nums, target).* El principio en el que se basa la función es que existe alguna manera de que la suma de un grupo de enteros sea igual a un valor objetivo, si y solo si, existe alguna manera de que el valor objetivo menos la suma de un grupo de enteros sea igual a cero.

PhD. Mauricio Toro Bermúdez
Docente | Escuela de Ingeniería | Informática y Sistemas
Correo: mtorobe@eafit.edu.co | Oficina: Bloque 19 – 627
Tel: (+57) (4) 261 95 00 Ext. 9473

target = a+b+c ←→ *target-a-b-c = 0.*

La complejidad del tiempo está dada por la función T(n), donde n es la diferencia entre la longitud del arreglo de enteros y *start,* es decir, lo que le falta a start para ser igual a la longitud y C_n son cantidades constantes de operaciones.

C_1=5 C_2=6 C_3=6

$T(n) = \begin{cases} C_1 & \text{cuando } n<=0 \\ C_2 + T(n-1) & \\ C_3 + T(n-1) + T(n-1) & \text{cuando } n>0 \end{cases}$

La función recursiva está dada por:

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

Aplicando la notación O

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

Aplicando reglas de suma y producto

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

SplitArray

Dado un arreglo de enteros, el método *SplitArray* nos dice si es posible dividir el arreglo en dos grupos, de tal manera que la suma de los elementos de cada grupo sea la misma; recibe los parámetros *nums* que es un arreglo de enteros, *s1* que es un entero donde se ira guardando la información del primer grupo, al principio siempre es igual a cero, *s2* que es un entero donde se ira guardando la información del segundo grupo, al principio siempre es igual a cero; y *start* es un entero que nos servirá para movernos a través del arreglo, al principio siempre es igual a cero. El funcionamiento del método es el siguiente: primero pregunta si *start* es igual a la longitud de *nums,* en caso de que si, retorna *true* si *s1* igual *s2*, o *false* si pasa lo contrario; en caso de que no, retorna la disyunción de dos llamados recursivos, el primer disyunto es un llamado recursivo de la función con el mismo *nums*, *s1* más el elemento de *nums* en la posición *start,* el mismo *s2,* y *start* más uno *split(nums, s1 + nums[start], s2, start + 1),* el segundo disyunto es un llamado recursivo de la función con el mismo *nums,* el mismo *s1, s2* más el elemento de *nums* en la posición *start,* y *start* más uno *split(nums, s1, s2 + nums[start], start + 1).* El principio en el que se basa el algoritmo es que para saber si es posible dividirlo en dos y que cada grupo sumen lo mismo, se deben comprobar todas las combinaciones posibles, en este orden de ideas debemos añadirle a *s1* y a *s2* elementos de *nums* a cada uno por separado y cuando hayamos añadido todos los elementos ya sea a *s1,* a *s2* o a ambos, verificamos si son iguales.

La complejidad del tiempo está dada por la función T(n), donde n es la diferencia entre la longitud del arreglo de enteros y *start,* es decir, lo que le falta a start para ser igual a la longitud y C_n son cantidades constantes de operaciones.

C_1=5 C_2=8

$T(n) = \begin{cases} C_1 & \text{cuando } n=0 \\ C_2 + T(n-1) + T(n-1) & \text{cuando } n>0 \end{cases}$

La función recursiva está dada por:

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

Aplicando la notación O

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

Aplicando reglas de suma y producto

PhD. Mauricio Toro Bermúdez
Docente | Escuela de Ingeniería | Informática y Sistemas
Correo: mtorobe@eafit.edu.co | Oficina: Bloque 19 – 627
Tel: (+57) (4) 261 95 00 Ext. 9473

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

SplitOdd10

Dado un arreglo de enteros, el método *SplitOdd10* nos dice si es posible dividir el arreglo en dos grupos, de tal manera que la suma de los elementos del primer grupo sea múltiplo de diez y la suma de los elementos del segundo grupo sea impar; recibe los parámetros *nums* que es un arreglo de enteros, *s1* que es un entero donde se ira guardando la información del primer grupo, al principio siempre es igual a cero, *s2* que es un entero donde se ira guardando la información del segundo grupo, al principio siempre es igual a cero; y *start* es un entero que nos servirá para movernos a través del arreglo, al principio siempre es igual a cero. El funcionamiento del método es el siguiente: primero pregunta si *start* es igual a la longitud de *nums,* en caso de que si, retorna *true* si el residuo de *s1* divido diez es igual a cero y el residuo de *s2* dividido dos es diferente de cero, o *false* si pasa lo contrario; en caso de que no, retorna la disyunción de dos llamados recursivos, el primer disyunto es un llamado recursivo de la función con el mismo *nums*, *s1* más el elemento de *nums* en la posición *start,* el mismo *s2,* y *start* más uno *split(nums, s1 + nums[start], s2, start + 1),* el segundo disyunto es un llamado recursivo de la función con el mismo *nums,* el mismo *s1, s2* más el elemento de *nums* en la posición *start,* y *start* más uno *split(nums, s1, s2 + nums[start], start + 1).* El principio en el que se basa el algoritmo es que para saber si es posible dividirlo en dos y que cada grupo cumpla con las condiciones dadas, se deben comprobar todas las combinaciones posibles, en este orden de ideas debemos añadirle a *s1* y a *s2* elementos de *nums* a cada uno por separado y cuando hayamos añadido todos los elementos ya sea a *s1,* a *s2* o a ambos, verificamos si *s1* es múltiplo de diez y *s2* es impar.

La complejidad del tiempo está dada por la función T(n), donde n es la diferencia entre la longitud del arreglo de enteros y *start,* es decir, lo que le falta a start para ser igual a la longitud y C_n son cantidades constantes de operaciones.

C_1=9 C_2=8

$T(n) = \begin{cases} C_1 & \text{cuando } n=0 \\ C_2 + T(n-1) + T(n-1) & \text{cuando } n>0 \end{cases}$

La función recursiva está dada por:

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

Aplicando la notación O

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

Aplicando reglas de suma y producto

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

Split53

Dado un arreglo de enteros, el método *split53* nos dice si es posible dividir el arreglo en dos grupos, de tal manera que la suma de los elementos de cada grupo sea la misma, con la condición de que todos los múltiplos de cinco deben estar en el primer grupo y todos los múltiplos de tres que no sean múltiplos de cinco deben estar en el segundo; recibe los parámetros *nums* que es un arreglo de enteros, *s1* que es un entero donde se ira guardando la información del primer grupo, al principio siempre es igual a cero, *s2* que es un entero donde se ira guardando la información del segundo grupo, al principio siempre es igual a cero; y *start* es un entero que nos servirá para movernos a través del arreglo, al principio siempre es igual a cero. El funcionamiento del método es el siguiente: primero pregunta si

PhD. Mauricio Toro Bermúdez
Docente | Escuela de Ingeniería | Informática y Sistemas
Correo: mtorobe@eafit.edu.co | Oficina: Bloque 19 – 627
Tel: (+57) (4) 261 95 00 Ext. 9473

start es igual a la longitud de *nums,* en caso de que si, retorna *true* si *s1* igual *s2*, o *false* si pasa lo contrario; en caso de que no, existen tres posibilidades, la primera sucede si el residuo del elemento de *nums* en la posición *start* dividido cinco es igual cero, y retorna el llamado recursivo con el mismo *nums*, *s1* mas el elemento de *nums* en *start,* el mismo s2, y *start* más uno *split(nums, s1 + nums[start], s2 start + 1)*. La segunda sucede si el residuo del elemento de *nums* en la posición *start* dividido tres es igual cero, y retorna el llamado recursivo con el mismo *nums*, el mismo *s1*, *s2* más el elemento de *nums* en *start,* y *start* más uno *split(nums, s1, s2 + nums[start], start + 1).* La tercera sucede si no se cumplen ninguna de las anteriores y retorna la disyunción de dos llamados recursivos, el primer disyunto es un llamado recursivo de la función con el mismo *nums*, *s1* más el elemento de *nums* en la posición *start,* el mismo *s2,* y *start* más uno *split(nums, s1 + nums[start], s2, start + 1),* el segundo disyunto es un llamado recursivo de la función con el mismo *nums,* el mismo *s1, s2* más el elemento de *nums* en la posición *start,* y *start* más uno *split(nums, s1, s2 + nums[start], start + 1).* El principio en el que se basa el algoritmo es que para saber si es posible dividirlo en dos y que cada grupo sumen lo mismo, se deben comprobar todas las combinaciones posibles, en este orden de ideas debemos añadirle a *s1* y a *s2* elementos de *nums* a cada uno por separado y cuando hayamos añadido todos los elementos ya sea a *s1,* a *s2* o a ambos, verificamos si son iguales.

La complejidad del tiempo está dada por la función T(n), donde n es la diferencia entre la longitud del arreglo de enteros y *start,* es decir, lo que le falta a start para ser igual a la longitud y C_n son cantidades constantes de operaciones.

C_1=5 C_2=8 C_3=8 C_4=8

$$T(n) = \begin{cases} C_1 & \text{cuando } n=0 \\ C_2 + T(n-1) & \\ C_3 + T(n-1) & \\ C_4 + T(n-1) + T(n-1) & \text{cuando } n>0 \end{cases}$$

La función recursiva está dada por:

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

Aplicando la notación O

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

Aplicando reglas de suma y producto

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

3.6

Las variables n y m son aquellas con las que se representa y entiende el tamaño del problema, es decir, son las variables que aumentan o disminuyen la complejidad del algoritmo ya sea en tiempo o en memoria.

4) Simulacro de Parcial

4.1

4.1.1 A
4.1.2 C
4.1.3 A

4.2

PhD. Mauricio Toro Bermúdez
Docente | Escuela de Ingeniería | Informática y Sistemas
Correo: mtorobe@eafit.edu.co | Oficina: Bloque 19 – 627
Tel: (+57) (4) 261 95 00 Ext. 9473

4.2.1 B
4.2.2 A y C

4.3 B

4.4

4.4.1 C

4.5

4.5.1 A
4.5.2 B

4.6 [OPC]

4.6.1 A

4.7 [OPC]

4.7.1 E

4.8 [OPC]

4.8.1 No sabemos definir n en este problema, por lo tanto, no lo pudimos solucionar

4.6

4.6.1 sumaAux (n, i+2)
4.6.2 sumaAux (n, i+1)

4.7 [OPC]

4.7.1 S, i + 1, t - S[i]
4.7.2 S, i + 1, t - S[i]

4.8 [OPC]

4.8.1 C

4.9 [OPC]

4.9.1 B

4.10 [OPC]

4.10.1 C

5) Lectura recomendada (opcional)

Mapa conceptual

6) Trabajo en Equipo y Progreso Gradual (Opcional)

6.1 *Actas de reunión*
6.2 *El reporte de cambios en el código*
6.3 *El reporte de cambios del informe de laboratorio*

PhD. Mauricio Toro Bermúdez
Docente | Escuela de Ingeniería | Informática y Sistemas
Correo: mtorobe@eafit.edu.co | Oficina: Bloque 19 – 627
Tel: (+57) (4) 261 95 00 Ext. 9473