Laboratorio Nro. 1
Recursión

Sebastian Castaño Orozco
Universidad Eafit
Medellín, Colombia
scasta31@eafit.edu.co

Dennis Castrillón Sepúlveda
Universidad Eafit
Medellín, Colombia
dcastri9@eafit.edu.co

2) Ejercicios en línea

Se procede a explicar brevemente cada ejercicio de recursión 1 y 2 de la página CodingBat realizado.

2.1 Ejercicios Recursión 1

- BunnyEars2: El algoritmo devuelve la cantidad de orejas de conejo a partir de una posición dada. La condición de parada es que se haya llegado a la posición 0, en la que retorna un cero. Luego, el algoritmo se pregunta si la posición que está analizando corresponde a una posición par o impar, sumándole 3 o 2 respectivamente y haciendo recursión para la posición n-1 y así sucesivamente.

- SumDigits: Este algoritmo retorna la suma de los dígitos de un número. En primer lugar, verifica que el número ingresado sea mayor a cero, posteriormente, si el numero es menor a 10 (es digito), devuelve este número. Finalmente, si el número tiene 2 o mas cifras, realiza la recursión para sus decenas, centenas y así sucesivamente, hasta reducirse a unidades en donde retorna este valor.

- PowerN: El algoritmo retorna el resultado de elevar una base en un exponente n. En primera instancia, revisa que tanto la base como el exponente sean mayores o iguales a 1, evitando indeterminaciones. La condición de parada es que la recursión para n llegue a exponente 1, lo que retornaría la base ya que $x^1=x$. Posteriormente, el algoritmo hace recursión multiplicando la base por el resultado de la base elevada a la potencia n-1, ya que $x^n=x*x^{n-1}$.

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

- Fibonacci: El algoritmo retorna el valor del enesimo número de Fibonacci. En primer lugar, se revisa si la entrada es 0 o 1, retornando estos mismos valores dados por el valor de Fibonacci para sus 2 primeras posiciones, posteriormente hace recursión retornando el valor de Fibonacci en la posición n como la suma de los valores de Fibonacci para las posiciones n-1 y n-2, así F(n)=F(n-1)+F(n-2).

- Count7: El algoritmo retorna la cantidad de 7 de un número positivo. En primera instancia revisa que el número sea mayor a cero, posteriormente realiza la recursión cogiendo cada una de las cifras del número y verificando si estas son 7. En caso de que sean 7, retorna un 1 (true) o por el contrario retorna un cero (false), posteriormente se suman estos valores para cada cifra, dando la cantidad de 7 en el número analizado.

2.2 Ejercicios Recursión 2

- GroupSum: El algoritmo consiste en realizar dos recursividades, una donde se selecciona el número de la posición actual (restándoselo al target) y otro sin seleccionar el número (no se le resta a target). De esta forma se crean todos los subgrupos, donde la condición de parada es que target llegue a ser 0, si esto se cumple al final mostrara True, gracias a la operación lógica Or.

- GroupSum6: El algoritmo tiene como condición escoger para el subgrupo todos los números 6, por lo que se plantea que si hay un número 6 en la posición actual del arreglo se selecciona obligatoriamente (se le resta al target). Si no hay un número 6 en la posición actual del arreglo se realizan las dos recursividades normales del punto anterior, donde la condición de parada es que target llegue a ser 0, si esto se cumple al final mostrara True, gracias a la operación lógica Or.

- GroupSum5: El algoritmo tiene como condición escoger para el subgrupo todos los múltiplos de 5, por lo que se plantea que si hay un múltiplo de 5 en la posición actual del arreglo se selecciona obligatoriamente (se le resta al target), pero se tiene otra condición adicional de que si es múltiplo de 5 la posición actual y el siguiente número del arreglo es 1, nos saltamos esa posición para no tomarla. Si no hay un múltiplo de 5 en la posición actual del arreglo se realizan las dos recursividades normales del punto anterior, donde la condición de parada es que target llegue a ser 0, si esto se cumple al final mostrara True, gracias a la operación lógica Or.

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

- GroupSumClump: El algoritmo tiene como condición escoger o no para el subgrupo todos los números que sean iguales y adyacente, por lo que se utilizó un ciclo para determinar cuántas veces se repetía dicho número, ya con esta información se le podía restar al target o simplemente con nos saltábamos dichas posiciones. Si en el arreglo no hay números iguales adyacentes simplemente se realizaba las recursiones normales, donde la condición de parada es que target llegue a ser 0, si esto se cumple al final mostrara True, gracias a la operación lógica Or.
- GroupNOADJ: El algoritmo tiene como condición de que si se seleccionaba un número, el numero adyacente a este no podía ser seleccionado, por lo que se llevaba un control del target para verificar si cambiaba (si se le había restado o no el número), si el target cambiaba se saltaba la posición adyacente y sino simplemente se continuaba con normalidad, donde la condición de parada es que target llegue a ser 0, si esto se cumple al final mostrara True, gracias a la operación lógica Or.
- SplitArray: El algoritmo tiene como condición de retornar true o false dependiendo, de si un arreglo de números se puedan dividir en dos grupos y que su suma sean iguales, para realizar esto se plantea la condición de parada con un contador que no supere el tamaño del array donde se compara la suma de los dos grupos. Cuando no se cumple la condición de parada se realiza una operación lógica Or entre dos recursiones, en la primera se le suma a la variable suma1 el número de la posición actual y se deja igual suma2, en la segunda se deja suma1 igual y se le suma el número de la posición actual a suma2.

3) Simulacro de preguntas de sustentación de Proyectos

3.1 Complejidad asintótico del ejercicio 1.1

Se procede a calcular la complejidad asintótica para el algoritmo de la longitud de subsecuencia común mas larga entre dos cadenas de caracteres.

```
def long_sub(cadena1,cadena2):
        m=len(cadena1) # C1 = 1
        n=len(cadena2) # C2= 1
        if m==0 or n==0:
            cont=0 # C3 = 4
        else:
```

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

```
    if cadena1[m-1]==cadena2[n-1]: # C4 = 1
        cont=1+long_sub(cadena1[0:m-1],cadena2[0:n-1]) # T(m-1)+T(n-1) + C5
    else:
        cont= max(long_sub(cadena1[0:m-1],cadena2), # T(m-1)+T(n-1) + C6
                  long_sub(cadena1,cadena2[0:n-1]))

    return(cont) # C7 = 1
```

$$T(m,n) = T(m-1) + T(n-1) + C_1 + C_2 + C_3 + C_4 + C_5 + C_6 + C_7$$
$$T(m,n) = T(m-1) + T(n-1) + C$$
$$T(m,n) = C_1(2^m - 1) + C_2(2^n - 1)$$

3.2 Estimación de tiempo para subsecuencia común entre cadenas de ADN's mitocondriales

Se procede a tomar los tiempos desde n=1 hasta n=16, siendo n la longitud de la cadena de caracteres a analizar, generando cadenas de caracteres aleatorias de tamaño n y ejecutando el algoritmo. Posteriormente se grafican los tiempos.

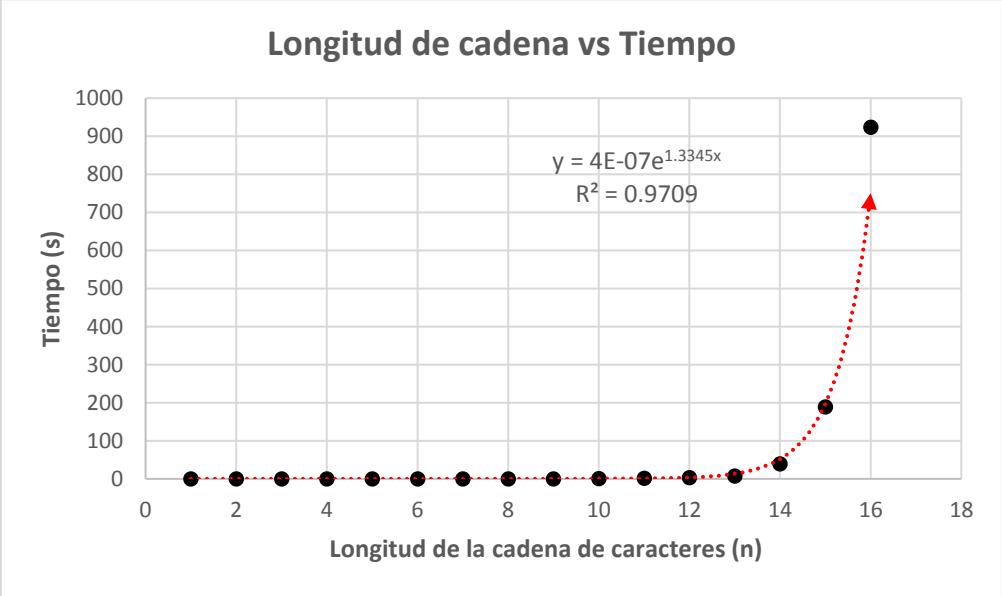

Figura 1. Grafica de la complejidad en el tiempo para ejercicio 1.1

Se obtiene la ecuación exponencial aproximada de la recta descrita por el tiempo tomado por el algoritmo dependiendo de la longitud de la cadena de caracteres de las entradas y se procede a estimar el tiempo necesario para encontrar la subcadena

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

común mas larga entre dos ADN's mitocondriales (cada cadena tiene 300000 caracteres).

$$Tiempo(s) = (4x10^{-7})e^{1.3345(n)}$$

$$Para\ n \cong 300000 \rightarrow Tiempo(s) = (4x10^{-7})e^{1.3345(300000)}$$

Cabe resaltar que este valor no se puede obtener debido al tamaño del número, por tanto, se procede a estimar el tiempo con el mayor valor n que puede procesar Python.

$$Para\ n \cong 531 \rightarrow Tiempo(s) = (4x10^{-7})e^{1.3345(531)} \cong 2.25x10^{301} \cong 7.12x10^{293} años$$

3.3 Interpretación complejidad del algoritmo

La complejidad del algoritmo del ejercicio 1.1 no es apropiada para calcular la subsecuencia común más larga entre ADN's mitocondriales ya que esta crece exponencialmente, por lo tanto, a medida que crecen las longitudes de caracteres a comparar, el tiempo que toma el algoritmo en ejecutarse para n cantidad de caracteres es mucho mayor al tiempo que requiere para ejecutarse para n-1 caracteres.

Incluso, estimar el tiempo que se demora el algoritmo en ejecutarse para cadenas de 300000 caracteres fue imposible debido a la capacidad de cálculo de calculadoras convencionales y Python, por lo que se estimó para cadenas de 531 caracteres (máximo valor que arrojó resultado) y se obtuvo que tardaría 7.12e+293, siendo este un valor descomunal. Por ende, se concluye que el algoritmo desarrollado no es apropiado para realizar el proceso con cadenas de ADN's mitocondriales.

3.4 Opcional: Funcionamiento algoritmo GroupSum5

```
1 def existe(start,nums,target):
2     if target==0:
3         return True
4     else:
5         if start<len(nums):
6             if nums[start]%5==0:
```

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

```
7            if nums[start+1]==1:
8               si= existe(start+2,nums,target-nums[start])
9               no=False
10           else:
11              si= existe(start+1,nums,target-nums[start])
12              no=False
13       else:
14           si= existe(start+1,nums,target-nums[start])
15           no= existe(start+1,nums,target)
16       si_no=si or no
17       return bool(si_no)
```

En primera instancia se define las variables que se usaran para determinar si existe un subgrupo con la condición de que se escojan todos los múltiplos de 5, y además si hay un 1 luego de un múltiplo de 5 no se puede seleccionar. En la línea 1 se usan las siguientes variables:

- Start: Es un contador que permite saber en qué punto del arreglo estamos ubicados.
- Nums: Es el arreglo de números del cual verificaremos si existe un subgrupo que cumpla la condición.
- Target: Es el objetivo que el subgrupo debe de cumplir.

En la línea 2 y 3 se define la condición de parada del ejercicio, donde si la variable del objetivo (target) llega a 0, se retorne True, indicando que existe un subgrupo que cumplió con la condición dada. Posteriormente si el objetivo (target) no es 0, en la línea 5 se plantea otra condición donde se verifica si el contador del arreglo (start) es menor a la longitud del arreglo para evitar errores.

De la línea 6 a la 12, se plantea la condición de que si el valor del arreglo en el que se está posicionado es múltiplo de 5 (%5==0), además posteriormente se verifica si el número que prosigue al número actual es 1, se salta la posición en la que se encuentra ese número (start+2), pero si el numero adyacente no es 1, continua no se salta dicha posición (start+1). Dentro de estas condiciones solo se plantea una recursión, que consiste en restarle a la variable objetivo (target) el número de la posición actual del arreglo (target-nums[start]), con el único cambio de saltarse o no la posición adyacente como se explicó anteriormente.

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

Entre la línea 13 y 15, se desarrolla las instrucciones para cuando el valor que se revisa actualmente del arreglo no es múltiplo de 5, se plantean dos recursiones, una consiste en restarle al target el número actual y otra que no se lo resta. Por último, en las líneas 16 y 17, se realiza la operación lógica de Or para ir subiendo en el árbol si algún subgrupo es True.

En pocas palabras el algoritmo recursivo planteado, crea el árbol, pero en los puntos donde existe un múltiplo de 5 obliga si o si a seleccionarlo y además si el numero siguiente es 1 tampoco se permite seleccionarlo, y si algún subgrupo cumple, se muestra el True comparándolo con una operación lógica (Or).

3.5 Complejidad de algoritmos de ejercicios en línea (CodingBat)

Se procede a obtener la complejidad de los algoritmos de los ejercicios en línea, procesando las ecuaciones extraídas de los algoritmos a través de Wolfram Alpha, obteniendo las ecuaciones de recurrencia.

3.5.1 Ejercicios Recursión 1

Ejercicio 1: BunnyEars2

```
def BunnyEars2(n):
    if n==0:
        return 0 # C_1 = 2
    else:

        if (n%2)==0: # C_2 = 2
            return 3+BunnyEars2(n-1) # T(n-1) + 1
        else:
            return 2+BunnyEars2(n-1) # T(n-1) + 1
```

$$T(n) = T(n-1) + C = Cn + C_1$$

Ejercicio 2: SumDigits

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

```
def SumDigits(n):
    if (n<=0):
        return 'No es posible, ingrese valores enteros mayores a cero.' # C_1 = 2
    if n<10:
        return n # C_2 = 2
    else:
        return SumDigits(n%10)+SumDigits(int(n/10)) # C_3 = 3
```

$$T(n) = C$$

Ejercicio 3: PowerN

```
def powerN(base,n):
    if base<1 or n<1: # C_1 = 4
        return "No es posible, ingrese valores enteros iguales o superiores a 1."
    if n==1:
        return base # C_2 = 2
    else:
        return base*powerN(base,n-1) # C_3 = T(n-1) + C
```

$$T(n) = T(n-1) + C = Cn + C_1$$

Ejercicio 4: Fibonacci

```
def fibonacci(n):
    if n==0:
        return 0 # C_1 = 2
    if n==1:
        return 1 # C_2 = 2
    else:
        return fibonacci (n-2)+ fibonacci(n-1) # T(n-2)+T(n-1) + C
```

$$T(n) = T(n-2) + T(n-1) + C = C_1F_n + C_2L_n - C$$

$$dónde\ F_n\ es\ el\ número\ n\ de\ Fibonacci\ y\ L_n\ es\ el\ número\ n\ de\ Lucas$$

Ejercicio 5: Count7

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

```
def count7(n):
    if (n<=0):
        return 'No es posible, ingrese valores enteros mayores a cero.' # C_1 = 1
    if n<10:
        if n==7:
            return 1
        else:
            return 0  # C_2 = 4
    else:
        return count7(n%10)+count7(int(n/10)) # C_3 = 3
```

$$T(n) = C$$

3.5.2 Ejercicios Recursión 2

Ejercicio1: GroupSum

```
def existe(start,nums,target):
    if target==0:
        return True # T(n) = c1 = 3
    else:
        if start<len(nums):
            si= existe(start+1,nums,target-nums[start])
            no= existe(start+1,nums,target)
            si_no=si or no
            return bool(si_no)

            # T(n) = c2 + T( n - 1 ) + T( n - 1 ) , donde c2 = 9
```

$$\mathrm{T(n)} = \mathrm{c2}\,(2^n - 1) + \mathrm{c1}\,2^{(\mathrm{n}-1)}$$

Ejercicio2: GroupSum6

```
def existe(start,nums,target):
    if target==0 and start==len(nums):
        return True # T(n) = c1 = 6
    else:
```

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

```
    if start<len(nums):
        if nums[start]==6:
            si= existe(start+1,nums,target-nums[start])
            no=False
            # T(n) = c2 + T( n - 1 ) , donde c2 = 4
        else:
            si= existe(start+1,nums,target-nums[start])
            no= existe(start+1,nums,target)
            # T(n) = c3 + T( n - 1 ) + T( n - 1 ) , donde c3 = 4
        si_no=si or no
        return bool(si_no)
        # T(n) = c4 + T( n - 1 ) + T( n - 1 ) , donde c4 = c3 + 6=10
```

$$T(n) = c4\,(2^n - 1) + c1\,2^{(n-1)}$$

Ejercicio3: GroupSum5

```
def existe(start,nums,target):
    if target==0:
        return True # T(n) = c1 = 6
    else:
        if start<len(nums):
            if nums[start]%5==0:
                if nums[start+1]==1:
                    si= existe(start+2,nums,target-nums[start])
                    no=False
                else:
                    si= existe(start+1,nums,target-nums[start])
                    no=False
                # T(n) = c2 + T( n - 1 ) , donde c2 = 9
            else:
                si= existe(start+1,nums,target-nums[start])
                no= existe(start+1,nums,target)
                # T(n) = c3 + T( n - 1 ) + T( n - 1 ) , donde c3 = 4
            si_no=si or no
            return bool(si_no)
            # T(n) = c4 + T( n - 1 ) + T( n - 1 ) , donde c4 = c3+6=10
```

$$T(n) = c4\,(2^n - 1) + c1\,2^{(n-1)}$$

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

Ejercicio4: GroupSumClump

```
def existe(start,nums,target):
    if target==0:
        return True # T(n) = c1 = 6
    else:
        if start<len(nums)-1:
            cont=0
            if nums[start+cont]==nums[start+1+cont]:
                while nums[start+cont]==nums[start+1+cont]:
                    cont=cont+1
                    if len(nums)==start+cont+1:
                        break
                cont=cont+1
                si= existe(start+1+cont,nums,target-nums[start]*cont)
                no= existe(start+1+cont,nums,target)
                # T(n) = c2*n + T( n - 1 ) + T( n - 1 ) + c3  , donde c2 = 12 y c3=12
            else:
                si= existe(start+1,nums,target-nums[start])
                no= existe(start+1,nums,target)
                # T(n) = c4 + T( n - 1 ) + T( n - 1 ) , donde c4 = 4
            si_no=si or no
            return bool(si_no)
            # T(n) = c2*n + T( n - 1 ) + T( n - 1 ) + c5  , donde c2 = 12 y c5= c3+ 7 =19
```

$$T(n) = c2\left(-n + 2^{(n+1)} - 2\right) + c5\,2^{(n-1)} + c1\,2^{(n-1)}$$

Ejercicio5: GroupNoAdj

```
def existe(start,nums,target,targetAnterior):
    if target==0:
        return True # T(n) = c1 = 6
    else:
        if start+1<len(nums):
            if target==targetAnterior:
                si= existe(start+2,nums,target-nums[start],target)
                no= False
```

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

```
        else:
            si= existe(start+1,nums,target-nums[start],target)
            no= existe(start+1,nums,target,target)
            # T(n) = c2 + T( n - 1 ) + T( n - 1 ) , donde c2 = 8
        si_no=si or no
        return bool(si_no)
        # T(n) = c3 + T( n - 1 ) + T( n - 1 ) , donde c3 = c2 + 7= 15
```

$$T(n) = c3\ (2^n - 1) + c1\ 2^{(n-1)}$$

Ejercicio6: SplitArray

```
def splitArray(nums):
  return recArray(nums, 0, 0, 0)

def recArray ( nums, index, sum1, sum2 ):
    if ( index >= len(nums) ):
        return sum1 == sum2 # T(n)=c1=5
    else:
        value = nums[index]
        return (recArray(nums, index + 1, sum1 + value, sum2)
                or recArray(nums, index + 1, sum1, sum2 + value))
        # T(n)=T(n-1)+T(n-1)+c2, dodne c2=5
print(splitArray([5,2,7,3]))
```

$$T(n) = c2\ (2^n - 1) + c1\ 2^{(n-1)}$$

3.6 Explicación variables del cálculo de complejidad de los ejercicios en línea (CodingBat)

3.6.1 Ejercicios Recursión 1

Para los ejercicios de recursión 1, se parte de una estructura similar, en la que solo se cuenta con una entrada al algoritmo recursivo.

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

En primer lugar, los ejercicios 2 y 5 poseen una complejidad constante, en la que n representa un valor numérico entero de entrada, pero la complejidad es la misma ya que la recursión se hace sin utilizar valores previos de n.

$$T(n) = C \;\; Ejercicio\ 2$$
$$T(n) = C\ Ejercicio\ 5$$

En segundo lugar, los ejercicios 1 y 3 poseen una complejidad constante pero que depende de n, a continuación, se explica que es n para cada ejercicio.

$$T(n) = T(n-1) + C = Cn + C_1 \;\; Ejercicios\ 1\ y\ 3$$

Ejercicio 1: BunnyEars2

```
def BunnyEars2(n):
```

En este caso, el valor de n corresponde a la cantidad de iteraciones previas que ha realizado el algoritmo, el cuál depende directamente del tamaño del valor entero positivo ingresado al algoritmo. Explícitamente, n corresponde al número de orejas de conejo en la posición (n) que se desea conocer.

Ejercicio 3: PowerN

```
def powerN(base,n):
```

En este caso, el valor de n corresponde al exponente al que se desea elevar la base, al hacer recursión, la complejidad del algoritmo depende directamente de que tan grande sea este exponente ya que el algoritmo se devuelve buscando la solución para n-1 y multiplicándole la base una vez más, obteniendo el valor para n.

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

Finalmente, el ejercicio 4, corresponde al ejercicio de Fibonacci, el cual posee una complejidad exponencial ya que recurre a las iteraciones n-1 y n-2. A continuación la ecuación de complejidad del algoritmo.

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

$$dónde\ F_n\ es\ el\ número\ n\ de\ Fibonacci\ y\ L_n\ es\ el\ número\ n\ de\ Lucas$$

Para este ejercicio, el valor de n corresponde al enésimo número de Fibonacci.

3.6.2 Ejercicios Recursión 2

Ya que los 5 ejercicios parten de la misma estructura, la definición de n en todos los ejercicios se realiza de igual manera, por lo que se plantea una definición global para estos.

$$\mathrm{T(n)} = \mathrm{c2}\ (2^n - 1) + \mathrm{c1}\ 2^{(n-1)}\ Ejercicio\ 1$$

$$\mathrm{T(n)} = \mathrm{c4}\ (2^n - 1) + \mathrm{c1}\ 2^{(n-1)}\ Ejercicio\ 2$$

$$\mathrm{T(n)} = \mathrm{c4}\ (2^n - 1) + \mathrm{c1}\ 2^{(n-1)}\ Ejercicio\ 3$$

$$\mathrm{T(n)} = \mathrm{c2}\ \left(-n + 2^{(n+1)} - 2\right) + \mathrm{c5}\ 2^{(n-1)} + \mathrm{c1}\ 2^{(n-1)}\ Ejercicio\ 4$$

$$\mathrm{T(n)} = \mathrm{c3}\ (2^n - 1) + \mathrm{c1}\ 2^{(n-1)}\ Ejercicio\ 5$$

$$\mathrm{T(n)} = \mathrm{c2}\ (2^n - 1) + \mathrm{c1}\ 2^{(n-1)}\ Ejercicio\ 6$$

A partir de los datos ingresados del usuario se determina la equivalencia de la variable n en la ecuación de complejidad del algoritmo.

def existe(start,nums,target):

Donde:

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

- nums: Es el arreglo de números o el grupo de números que el usuario ingresa para comparar
- target: Es el objetivo al cual se quiere llegar, es decir la condición que brinda el usuario para que el subgrupo sea el adecuado

Teniendo en cuenta la información anterior, la variable n correspondería a la longitud o tamaño del arreglo, es decir si el arreglo nums este compuesto por [2,5,16], el tamaño de ese arreglo es 3, y a su vez la variable n es 3.

4) Simulacro de Parcial

4.1

1. a
2. c
3. a

4.2

1. a
2.

a. Verdadero

b. Falso

c. Verdadero

4.3 b

4.4 return lucas(n-2) + lucas(n-1)

4.4.1 c

4.5

1. a

2. b

4.4.1 a

4.6

4.6.1 SumaAux(n, i+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

4.6.2 SumaAux(n, i+1)

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