Laboratorio Nro. 1: Recursión

Pablo Alberto Osorio Marulanda
Universidad Eafit
Medellín, Colombia
paosoriom@eafit.edu.co

Verónica Mendoza Iguarán
Universidad Eafit
Medellín, Colombia
vmendozai@eafit.edu.co

2)

2.1) Recursión 1:

Ejercicio 1: countAbc

El ejercicio comienza con una condición, donde evalúa que el string tenga más de 3 líneas, porque si no fuera así, entonces no se podría evaluar el string “abc” y ”aba”. Posteriormente se evalúan las primeras 3 letras de cada string, verificando si está o no los strings respectivos. En el caso de que si esté entonces se suma uno y se hace un llamado recursivo con el string, comenzando desde su posición número 2, pues si fuese otra posición se podrían ignorar posibles strings. Si esto no se cumple se llama recursivamente ignorando la primera posición del string.

Ejercicio 2: countHi2

El ejercicio comienza evaluando si el string, desde el principio es igual al que se busca (“hi”). Si esto no se cumple entonces verificamos si el string es mayor a 3(para de esta manera verificar si tiene x o no). Si el string comienza con “x” y tiene un “hi” contiguo entonces retornamos 0 y el string evaluado desde la tercera posición en el string. Si no se cumple lo anterior evaluamos la posibilidad de que solamente comience con “hi”, si este es el caso sumamos 1 y evaluamos lo restante del string. Si nada de esto se cumple evaluamos desde la siguiente posición respectiva del string evaluado.

Ejercicio 3: parenBit

En la primera línea del código se examina si el tamaño de la cadena es menor a dos (pues corresponde a los dos paréntesis), de ser así se retorna la cadena. Luego se verifica si el primer y el último carácter son paréntesis, si esto pasa se retorna toda la cadena, si solo el primer carácter es paréntesis y el último no, se hace el llamado recursivo desde el primer caracter hasta el penúltimo, buscando así el paréntesis final. Finalmente si el primer carácter no es un paréntesis se hace otro llamado recursivo desde el segundo caracter en adelante.

Ejercicio 4: nestParen

Este código verifica si la cadena de caracteres contiene pares de paréntesis.

La primera línea es la condición de parada, está retorna “true” si la cadena se encuentra vacía. En la segunda línea se verifica si el primer carácter es un paréntesis y si el último caracter también lo es, sí es así, se hace el llamado recursivo empezando desde el carácter inicial más uno y finalizando con el último carácter menos uno

(de esta manera la cadena va disminuyendo y se van formando pares de paréntesis). Finalmente, si no sucede lo anterior retorna "false".

Ejercicio 5: strDist

El ejercicio cuenta el número de strings que se encuentran entre dos "sub" previamente asignados. Se verifica primero si el string es mayor que sub, de esta manera "sub" podrá estar dentro del string. Si el string comienza por el "sub" entonces procedemos a verificar si también acaba con este. Si es así entonces retornamos la longitud del string. Si no termina con el sub, entonces hacemos un llamado recursivo con el string menos su último término. Si por otra parte, no comienza con el sub, entonces hacemos el llamado recursivo con el string eliminando su primer término.

2.2) Recursión 2:

Ejercicio 1: groupNoAdj

En la primera línea de este ejercicio está la condición de parada: si "start" es mayor al número de caracteres de "nums", "target" será 0. Luego se hace el llamado recursivo de manera que se busca ir restando cada número del subgrupo a "target" para que este llegue a 0, si se toma un número, el que está al lado no se tomará por lo que en el llamado recursivo a "start" se le suma dos. De esta manera, se suman los subgrupos correspondientes ignorando los que son contiguos.

Ejercicio 2: groupSum6

El ejercicio comienza evaluando, con el caso base, si la variable "start" es mayor a la longitud del arreglo. Si es así el llamado se acaba y retorna 0. Ahora, evaluamos si la posición del arreglo en "start" es 6 y allí dentro hacemos un llamado recursivo. De esta manera se generarán grupos que tengan el 6 si el arreglo contiene tal número, es, básicamente, generar una rama adicional con tal restricción. Si no se cumple lo anterior, el código se desarrolla como normalmente lo hace el groupSum, es decir, con dos llamados recursivos donde uno anexa un valor al subgrupo y el otro(llamado recursivo) no lo agrega(el número).

Ejercicio 3: groupSumClump

Se tiene el caso base general de los groupSum. Se declara una variable "i" que será utilizada en el ciclo y como comienzo para los siguientes llamados recursivos. Ahora, se declara un ciclo con el propósito de sumar, para cada llamado, los números del arreglo que sean iguales a la primera posición y sean contiguos. Después de esto, se generan dos llamados recursivos, ambos comenzando donde acabó la "i" del ciclo anterior, de manera que no se repitan los "grupos de suma". Generamos dos "ramas" (dos llamados recursivos), una en donde se reste la suma (es decir, un grupo que la incluya), y otra donde no lo haga, es decir, el "target" se deje igual.

Tomado de:

Miranda, Alfredo (2013) github.com/mirandaio

https://github.com/mirandaio/codingbat/blob/master/java/recursion-2/groupSumClump.java

Ejercicio 4:splitArray

Para la solución de este ejercicio se deberá hacer un método auxiliar que reciba parámetros que necesitamos para ejecutar el group sum corriente. Dado que son dos subgrupos los que necesitamos formar, entonces se deberán hacer dos llamados recursivos. El caso base es el mismo de siempre, es decir, se ejecuta cuando el start (variable que irá aumentado y visitando las posiciones del arreglo) sea igual al tamaño del arreglo. Los dos

subgrupos tienen una remarcada diferencia. No podemos hacer lo que se hacía en el groupSum(ignorar un elemento creando subgrupos), sino hacer operaciones diferentes con cada target(suma y resta) para así crear dos subgrupos que, aunque diferentes, tienen en cuenta todos los elementos.

Ejercicio 5: split53

El ejercicio se ejecuta de manera similar al anterior (splitArray), de manera que, creamos un método auxiliar con los parámetros necesarios, La condición base será la misma de los group sum. Para poder ejecutar los subgrupos con las restricciones creamos dos condicionales que se ejecutan cuando la restricción es válida, sin embargo, como son subgrupos diferentes, entonces en uno de ellos sumamos el valor en el arreglo, mientras que en el otro lo restamos. De esta manera garantizamos que se crean subgrupos disjuntos. Dado que podría no cumplirse ninguna de las restricciones proponemos, con otros dos llamados recursivos, la creación de los subgrupos que normalmente conformarían las ramas respectivas.

3) Simulacro de preguntas de sustentación de Proyectos

1. El ejercicio de groupSum5 comienza verificando si el start es mayor o igual a la longitud del arreglo, pues dado que esta variable irá aumentando y evaluando cada posición, entonces el caso base dependerá de ella.

Se evaluará primero si el módulo 5 de la posición del arreglo que se está visitando es igual a 0, es decir, es múltiplo de tal número. Posteriormente, si la posición respectiva del arreglo no es la última, entonces se entrará a evaluar si su contigua es igual a 1. De ser así, entonces se toma la posición actual y no la siguiente, es por ello que se suma "+2" al start. Luego, si la posición sigue siendo múltiplo de 5 (o es la última o su siguiente no es uno) se evalúa normalmente con otro llamado recursivo, teniendo en cuenta que el valor múltiplo quedará siempre como un subgrupo válido igual al target buscado.

Ahora, si tal valor (el que se está visitando en el arreglo) no es múltiplo de 5, se ejecuta el groupSum como normalmente se hace, es decir, sacando dos ramas donde se tiene en cuenta una posición y otra donde no, formando subgrupos sin restricciones.

2. **Complejidad de los ejercicios en línea:**

2.1.

countAbc:

$T(n) = c_4 + T(n-1)$

$T(n)=c_4n+c_1$

$T(n)$ es $O(c_4n+c_1)$ por definición de O

$T(n)$ es $O(c_4n)$ por Regla de suma

$T(n)$ es $O(n)$ por Regla del producto

countHi2:
T(n) = c_8 + T(n-1)
T(n)=c_8n+c_1
T(n) es O(c_8n+c_1) por definición de O
T(n) es O(c_8n) por Regla de suma
T(n) es O(n) por Regla del producto

parenBit:
T(n) = c_5 + T(n-1)
T(n)=c_5n+c_1
T(n) es O(c_5n+c_1) por definición de O
T(n) es O(c_5n) por Regla de suma
T(n) es O(n) por Regla del producto

nestParen:
T(n) = c_5 + T(n-1)
T(n)=c_5n+c_1
T(n) es O(c_5n+c_1) por definición de O
T(n) es O(c_5n) por Regla de suma
T(n) es O(n) por Regla del producto

strDist:
T(n) = c_6 + T(n-1)
T(n)=c_6n+c_1
T(n) es O(c_6n+c_1) por definición de O
T(n) es O(c_6n) por Regla de suma
T(n) es O(n) por Regla del producto

2.2.

groupNoAdj:
T(n) = T(n-2) + T(n-1) + c_2
T(n) = (2^n-1) + (2^n-2) + c_2
T(n) es O((2^n-1) + (2^n-2) +O(c_2), por definición de O
T(n) es O(2^(n-1)) + O(2^(n-2)), por Regla de la suma
T(n) es O(2^(n-1)), por Regla de la suma
T(n) es O(2^n), por Regla del Producto

groupSum6:
T(n) = T(n-1) + T(n-1) + T(n-1) + c_2
T(n) = c_2 + 3*T(n-1)
T(n) = c_2*3^(n-1) + ½ c_2*(3^n-1)

T(n) es O(c_2*3^(n-1) + ½ c_2*(3^n - 1)), por definición de O
T(n) es O(3^(n-1) + (3^n – 1)), por Regla del producto
T(n) es O(3^n – 1), por Regla de la suma
T(n) es O(3^n), por Regla del Producto

groupSumClump:
T(n) = c_2 + c_3*n + T(n-1) + T(n-1)
T(n) = c_2*(2^n - 1) - c_3*(n+2) + (c_1+4*c_3)*2^(n-1)
T(n) es O(c_2*(2^n - 1) - c_3*(n+2) + (c_1+4*c_3)*2^(n-1)), por definición de O
T(n) es O((2^n - 1) - (n+2) + (c_1+4*c_3)*2^(n-1)), por regla del producto
T(n) es O((2^n - 1) - (n+2) + (4*c_3)*2^(n-1)), por regla de la suma
T(n) es O((2^n - 1) - (n+2) + 2^(n-1)), por regla del producto
T(n) es O(2^n – 1), por regla de la suma
T(n) es O(2^n), por Regla del Producto

splitArray:
T(n) = T(n-1) + T(n-1) + c_2
T(n) = 2*T(n-1) + c_2
T(n) = c_1*2^(n-1) + c_2*(2^n-1)
T(n) es O(c_1*2^(n-1) + c_2*(2^n-1)), por definición de O
T(n) es O(2^(n-1) + 2^n-1), por Regla del Producto
T(n) es O(2^n-1), por Regla de la suma
T(n) es O(2^n), por Regla del Producto

split53:
T(n) = T(n-1) + T(n-1) + T(n-1) + T(n-1) + c_2
T(n) = 4*T(n-1) + c_2
T(n) = c_1*4^(n-1) + 1/3 c_2*(4^n - 1)
T(n) es O(c_1*4^(n-1) + 1/3 c_2*(4^n - 1)), por definición de O
T(n) es O(4^(n-1) + 4^n-1), por Regla del Producto
T(n) es O(4^n-1), por Regla de la suma
T(n) es O(4^n), por Regla del Producto

3. **countAbc:** La variable "n" representa la cantidad de posiciones en el arreglo de caracteres que aún no han sido procesadas. Esto implica que n representa, a su vez, la cantidad de iteraciones faltantes.

   countHi2: La variable "n" representa las posiciones en el string que aún no han sido visitadas, es decir, que aún no han pasado por un ciclo recursivo

   parenBit: Para este caso, la "n" simboliza cada uno de los caracteres en el string. De esta manera, representa las casillas faltantes en el arreglo de caracteres, es decir, las que aún no han sido visitadas por los llamados recursivos.

nestParen: Este es un caso particular, ya que la "n" representa la cantidad de paréntesis dividido 2 o, lo que es lo mismo decir, cada par de paréntesis simétrico o un paréntesis asimétrico, de manera que, adrede, representa el número de llamados recursivos que se van a hacer.

strDist: La variable "n" para este problema representa, al igual que los anteriores, la cantidad de caracteres del string que aún faltan por procesar, es decir, la cantidad de posiciones en el arreglo de caracteres que aún no han sido procesados por algún llamado recursivo.

groupNoAdj ; groupSum6; groupSumClump
Para estos caso, la variable "n" representa las posiciones en el arreglo que aún no han sido sumadas, es decir, aquellas que no han sido visitadas a través de la variable "start". De esta manera se eligen los subgrupos respectivos a la suma representada en el target.

splitArray ; splitArray 53
La variable "n" para el este caso representa una posición en el arreglo. Cada posición n es evaluada según las restricciones respectivas, y ya que el programa tiene una complejidad exponencial, entonces el arreglo es recorrido dos veces por iteración para las "n" respectivas faltantes en cada caso.

4. El Stack Overflow es un error recurrente cuando se trabaja con recursión. Este error aparece cuando una recursión es, básicamente, infinita, es decir, no tiene condición de parada o esta falla. De tal manera, la recursión se llama una y otra vez en métodos posteriores y nunca finaliza. De manera que, cada una de las recursiones a las que se le asigna memoria en la pila crece, y, como no tenemos memoria infinita, pues, en efecto, colisiona l pila y sale tal error.
5. El valor más grande que se pudo calcular fue el Fibonacci de 46 (1.836.311.903), a partir de este número en adelante el programa arroja valores negativos pues el tamaño máximo que admite java para un entero es 2.147.483.647 y este fue superado.
   Se puede notar que a medida que "n" aumenta, el cálculo tarda más en realizarse pues la función recursiva hace numerosos llamados; por ejemplo para n = 6 se hacen 24 llamados, para n = 5, 14 llamados, para n = 4, son 8 llamados; con estos valores y una gráfica de dispersión se puede deducir un valor aproximado para el número de llamados con n = 1.000.000, serian 12856951143068 * 10^12 llamados, esto tomaría demasiado tiempo de ejecución siendo así imposible para java calcularlo.

6. Una posible opción para calcular el Fibonacci de números grandes es evitando los numerosos llamados recursivos, representando los resultados en un arreglo donde cada dígito es una posición, y almacenando en una variable las dos últimas posiciones del arreglo para calcular el Fibonacci siguiente.
7. A partir de la complejidad encontrada para los ejercicios en línea de condingBat de recursión1 y recursión2, se puede concluir que estos últimos tienen un mayor índice de complejidad, dado que todos ellos pertenecen a la familia de curvas de O(2^n). Esto también es observable al comparar la estructura de los códigos que componen a ambos ejercicios. Los primeros se componen, por lo general de un llamado recursivo, y si son dos están juntos en el mismo programa, entonces lo estarán dentro de un condicional que permite que solo se cumpla uno por recursión. De esta manera el número de iteraciones está linealmente relacionado con la variable "n". En el segundo caso, para cada programa de debe hacer

más de un llamado recursivo, de manera que el número de iteraciones está relacionado exponencialmente con la variable "n"

4) Simulacro de Parcial

1. start+1,nums,target

2. Opción b

3.
3.1(n-a,a,b,c)
3.2. (res,solucionar(n-b,a,b,c)+1)
3.3.(res,solucionar(n-c,a,b,c))

4. Opción e

5.
5.1.
linea 2: return n;
linea 3: n-1
linea 4: n-2
5.2. Opción b

6.
6.1. sumaAux(n,i+2);
6.2. sumaAux(n,i+1);

7.
7.1. S, i+1, t-S[i]
7.2. S, i+1, t

8.
8.1. return 0;
8.2. ni+nj;

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

6) Trabajo en Equipo y Progreso Gradual (Opcional)

a) Actas de reunión:

Trabajo en equipo y progreso grupal				
Integrante	**Fecha**	**Hecho**	**Haciendo**	**Por hacer**
Pablo	31/07/2018 Hora de inicio: 3pm	· Creación del algoritmo Fibonacci · Solucionando ejercicios en línea (Recursión 1)	· Resolver ejercicios en línea (Recursión 1-2)	· Implementar Fibonacci con números grandes
Verónica	31/07/2018 Hora de inicio: 3pm	· Desarrollo de ejercicios en línea (Recursión 1-2)	· Resolver ejercicios en línea (Recursión 2)	
Verónica	7/08/2018 Hora de inicio: 3pm	· Implementación de Fibonacci con valores grandes. · Cómo resolver Fibonacci con valores grandes		
Pablo	7/08/2018 Hora de inicio: 3pm	· Elaboración de ejercicios en línea (Recursión 2)		· Explicación del funcionamiento de GroupSum5
Verónica	11/08/2018 Hora de inicio: 12pm	· Solución a ejercicios simulacro de parcial (1, 6, 7 y 8)	· Ejercicios simulacro de parcial	· Explicación del funcionamiento de GroupSum5

Pablo	11/08/2018 Hora de inicio: 12pm	· Explicación del funcionamiento de GroupSum5 · Interpretación y desarrollo sobre Stack Overflow		· Cálculo de complejidad de ejercicios en línea
Verónica	18/08/2018 Hora de inicio: 12pm	· Cálculo de complejidad de ejercicios Recursión 2		
Pablo	18/08/2018 Hora de inicio: 12pm	· Cálculo de complejidad de ejercicios Recursión 1 · Explicación de las variables del cálculo de complejidad · Conclusión de la complejidad de recursión 1 y 2		
Pablo y Verónica	Hora de inicio: 12pm	· Solución a ejercicios simulacro de parcial(2, 3, 4 y 5)		

Nota: No se anexa historial de gitHub, debido a que se subió el código hasta que estuviese acabado completamente.(Igual con GoogleDocs)