ESTRUCTURA DE DATOS 1
Código ST0245

Laboratorio Nro. 1
Recursión

Alejandra Palacio Jaramillo
Universidad Eafit
Medellín, Colombia
apalacioj@eafit.edu.co

Valentina Moreno Ramírez
Universidad Eafit
Medellín, Colombia
vmorenor@eafit.edu.co

3) Simulacro de preguntas de sustentación de Proyectos

3.1 Complejidad:

a = n1+n2

$$T(a) = \begin{cases} c1, \ \ donde\ c1 \ = \ 5. \\ c2 + T(a-2), \ donde\ c2 \ = \ 10. \\ c3 + T(a-1) + T(a-1), \ donde\ c3 \ = \ 5. \end{cases}$$

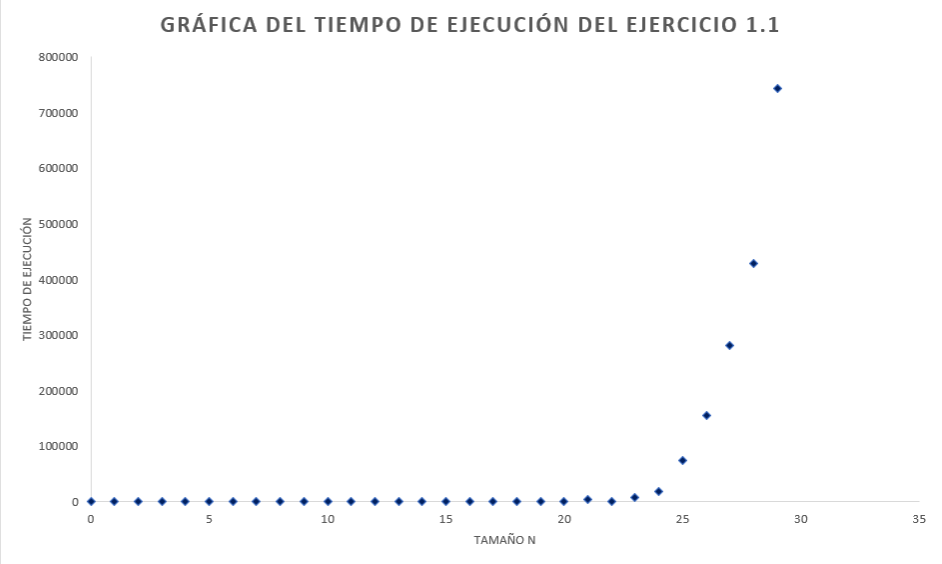
$T(a) = c_3 + T(a-1) + T(a-1)$
$T(a) = (c_3)(2^a - 1) + (c_1)(2^{a-1})$, donde c_1 es un parámetro arbitrario.
$T(a) = O((c_3)(2^a - 1) + (c_1)(2^{a-1}))$, aplicando notación O.
$T(a) = O((c_3)(2^a - 1))$, aplicando la regla de la suma.
$T(a) = O(c_3*2^a - c_3)$
$T(a) = O(c_3*2^a)$, aplicando regla de la suma.
$T(a) = O(2^a)$, aplicando regla del producto.

La complejidad asintótica para el peor de los casos del algoritmo que calcula la longitud de la subsecuencia común más larga es $O(2^a)$, en otras palabras, exponencial, donde a es igual a la suma de las longitudes de ambas cadenas de caracteres. En la siguiente gráfica, se evidencia este comportamiento.

3.2 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

n	T(n)
0	0
1	0
2	0
3	0
4	0
5	0
6	0
7	0
8	1
9	0
10	1
11	0
12	1
13	5
14	1
15	11
16	66
17	65
18	5
19	18
20	58
21	3499
22	208
23	8157
24	19157
25	74244
26	154642
27	280860
28	428144
29	743591



La tendencia de la función graficada con relación al tiempo de implementación del algoritmo que calcula la longitud de la subsecuencia más larga es exponencial, lo cual concuerda con la complejidad asintótica para el peor de los casos hallada en el ejercicio anterior. De acuerdo con la gráfica, se puede concluir que cuando el tamaño del problema es 30, el tiempo de ejecución del algoritmo es cercano a 743591 milisegundos, lo cual equivale a aproximadamente 12 minutos.

Con base en el planteamiento anterior es claro que estimar el tiempo de ejecución del algoritmo para dos ADNs mitocondriales (300.000 caracteres) es bastante complejo para este algoritmo, pero lo que sí se puede afirmar es que cuando esta función tiende al infinito, el resultando también tiende hacia el infinito (números con muchísimas cifras).

3.3

De acuerdo con los dos puntos anteriores, se concluye que la complejidad del algoritmo del ejercicio 1.1 es inapropiada para encontrar la longitud de la subsecuencia común más larga con cadenas como las de los *datasets* porque estas tienen una gran cantidad de caracteres, lo cual implica más tiempo de ejecución, más *heap* y más *stack*. Como consecuencia de esto, el algoritmo resultaría ineficiente tanto en tiempo como en almacenamiento y, por tanto, se requiere otro tipo de programación.

3.4

GroupSum5: de acuerdo con el enunciado dado, para este algoritmo, cuando el inicio y la longitud del arreglo son iguales, se retorna que el objetivo es igual a cero, siendo este el caso base. Adicionalmente, otra condición de parada ocurre cuando el número en la posición que indica el índice módulo 5 es igual a cero, el índice es menor que el índice de la última posición del arreglo y el número en la posición siguiente a la actual es igual a 1. Cuando esta condición se cumple, el número en la posición siguiente a la actual será igual a cero, para así cumplir con una de las condiciones dadas en el ejercicio. Por último, como caso recursivo se tiene que cuando no se cumplen las dos condiciones anteriores, se

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

llama al método dos veces pasándole como índice la siguiente posición, el mismo arreglo, pero diferente valor objetivo. En un llamado, el objetivo será el objetivo actual menos el número en la posición que indica el índice y en el otro, será el valor objetivo. Cabe resaltar que, para cumplir las condiciones del ejercicio, en uno de estos llamados se toma en cuenta si el número es múltiplo de 5. Finalmente, estos dos llamados se unen con un *or* ya que, si se cumplen las condiciones en cualquiera de los dos llamados, el resultado será true y, de lo contrario, false.

3.5

Recursión 1:

- **Factorial:** el algoritmo permite calcular el factorial de un número dado, donde el caso base se da cuando este número es igual a cero, su factorial es igual a 1 (por definición de factorial). Por otro lado, cuando el número es diferente de cero, el caso recursivo irá haciendo la búsqueda y vuelta atrás hasta llegar a el caso base.

  Complejidad

  $$T(n) = \begin{cases} c1, donde\ c1 = 3 \\ c2 + T(n-1), donde\ c2 = 4 \end{cases}$$

  $T(n) = c_2 + T(n-1)$
  $T(n) = c_2*n + c_1$, donde c_1 es un parámetro arbitrario.
  $T(n) = O(c_2*n + c_1)$, aplicando notación O.
  $T(n) = O(c_2*n)$, aplicando regla de la suma.
  $T(n) = O(n)$, aplicando regla del producto.

  La complejidad asintótica para el peor de los casos es O(n), es decir, lineal, donde n es igual al número al cual se le quiere calcular el factorial.

- **Bunny Ears:** el algoritmo permite calcular, con una cantidad k de conejos dada por el usuario, la cantidad de orejas totales de todos los k conejos. Cuando no hay ningún conejo, es decir, cuando no hay alguna oreja por contar, se llega al caso base y al devolverse luego de la búsqueda, se sumará 2 unidades al valor retornado en cada caso.

  Complejidad

  $$T(n) = \begin{cases} c1, donde\ c1 = 3 \\ c2 + T(n-1), donde\ c2 = 4 \end{cases}$$

  $T(n) = c_2 + T(n-1)$
  $T(n) = c_2*n + c_1$, donde c_1 es un parámetro arbitrario.
  $T(n) = O(c_2*n + c_1)$, aplicando notación O.
  $T(n) = O(c_2*n)$, aplicando regla de la suma.
  $T(n) = O(n)$, aplicando regla del producto.

  La complejidad asintótica para el peor de los casos es O(n), es decir, lineal, donde n es igual al número de conejitos al cual se le quiere calcular el número total de orejas.

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:** Para este algoritmo, como caso base se tienen los dos primeros números de esta sucesión, esto es, cuando el valor de n es 0 o 1, teniendo como resultado este mismo valor. Para el caso recursivo, se tienen dos llamados recursivos los cuales, mediante la búsqueda y vuelta atrás, hallan el n de dos posiciones atrás y el n de una posición atrás, los cuales se suman y dan como resultado el Fibonacci n.

  Complejidad:

  $$T(n) = \begin{cases} c1, & donde\ c1 = 3. \\ c2, & donde\ c2 = 3. \\ c3 + T(n-2) + T(n-1), & donde\ c3 = 3. \end{cases}$$

  $T(n) = c_3 + T(n-2) + T(n-1)$
  $T(n) = c_1*Fn + c_2*Ln - c_3$, donde Fn es el valor enésimo de Fibonacci, Ln es el enésimo número de Lucas y, c1 y c2 son parámetros arbitrarios.
  Como T(n) no queda en términos de n solamente, entonces se resolverá para T(n-1) + T(n-1) ya que esta complejidad representa mayor trabajo y, por tanto, una cota superior como la de la complejidad anterior.

  $T(n) = c_3 + T(n-1) + T(n-1)$
  $T(n) = (c_3)(2^n - 1) + (c_1)(2^{n-1})$, donde c_1 es un parámetro arbitrario.
  $T(n) = O((c_3)(2^n - 1) + (c_1)(2^{a-1}))$, aplicando notación O.
  $T(n) = O((c_3)(2^n - 1))$, aplicando la regla de la suma.
  $T(n) = O(c_3*2^n - c_3)$
  $T(n) = O(c_3*2^n)$, aplicando regla de la suma.
  $T(n) = O(2^n)$, aplicando regla del producto.

  La complejidad asintótica para el peor de los casos es $O(2^n)$, es decir, exponencial, donde n es el enésimo término de la serie de Fibonacci que se quiere calcular.

- **BunnyEars2:** este algoritmo es similar al de BunnyEars, con la diferencia de que habrá conejos pares (2 orejas y pata levantada) e impares (2 orejas). Se tiene el mismo caso base que en el ejercicio BunnyEars, pero habrá dos situaciones en las que se empleará la recursión; cuando el número de conejos es par, se suman 3 unidades al retorno de la vuelta atrás, en cambio, cuando es impar, se suman 2 unidades a este.

  Complejidad:

  $$T(n) = \begin{cases} c1, & donde\ c1 = 3. \\ c2 + T(n-1), & donde\ c2 = 6. \\ c3 + T(n-1), & donde\ c3 = 4. \end{cases}$$

  $T(n) = c_3 + T(n-1)$
  $T(n) = c_3*n + c_1$, donde c_1 es un parámetro arbitrario.
  $T(n) = O(c_3*n + c_1)$, aplicando notación O.
  $T(n) = O(c_3*n)$, aplicando regla de la suma.

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) = O(n), aplicando regla del producto.

La complejidad asintótica para el peor de los casos es O(n), es decir, lineal, donde n es el número de conejitos a los cuales se les quiere calcular el total de "orejitas"

- **CountX:** este algoritmo recibe una cadena de caracteres y retorna el número de veces que aparece 'x' en esta. Como caso base se tiene cuando la cadena es vacía, por ende, se retorna 0. Ahora, el caso recursivo está condicionado por: si el carácter en la posición 0 de la cadena de caracteres es igual a 'x' se retorna 1 más el llamado al método con la cadena sin el carácter que se acaba de evaluar. Por otro lado, si esta condición no se cumple, se llama el método de la forma que lo hicimos anteriormente sin sumarle 1.

  Complejidad:

$$T(n) = \begin{cases} c1, & donde\ c1 = 4. \\ c2 + T(n-1), & donde\ c2 = 6. \\ c3 + T(n-1), & donde\ c3 = 3. \end{cases}$$

  $T(n) = c_3 + T(n-1)$
  $T(n) = c_3 * n + c_1$, donde c_1 es un parámetro arbitrario.
  $T(n) = O(c_3 * n + c_1)$, aplicando notación O.
  $T(n) = O(c_3 * n)$, aplicando regla de la suma.
  T(n) = O(n), aplicando regla del producto.

  La complejidad asintótica para el peor de los casos es O(n), es decir, lineal, donde n es la longitud de la cadena de caracteres a la cual se le calculará el número de caracteres 'x' en ella.

Recursión 2:

- **GroupSum6:** de acuerdo con el enunciado dado, en este algoritmo se tiene como caso base cuando el inicio (índice en el arreglo) es igual a la longitud del arreglo, se retornará que el objetivo es igual a cero. Para los casos recursivos, se tiene una condición previa: si la posición actual el arreglo (nums[start]) es igual a seis, se llamará al método para la posición siguiente del arreglo con el objetivo disminuido en 6. Si esta condición no se cumple, entonces se hará otro llamado recursivo para la siguiente posición, con el objetivo disminuido en el número de la posición actual (nums[start]) y otro también para la siguiente posición, pero con el valor objetivo inicial, creando así todas las posibilidades. El algoritmo retorna true si se cumplen las condiciones dadas, de lo contrario, false.

  Complejidad:

$$T(n) = \begin{cases} c1, donde\ c1 = 5 \\ c2 + T(n-1), donde\ c2 = 8 \\ c3 + T(n-1) + T(n-1), donde\ c3 = 8 \\ c4,\ donde\ c4 = 1 \end{cases}$$

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

$T(n) = c_3 + T(n-1) + T(n-1)$
$T(n) = (c_3)(2^n - 1) + (c_1)(2^{n-1})$, donde c_1 es un parámetro arbitrario.
$T(n) = O((c_3)(2^n - 1) + (c_1)(2^{n-1}))$, aplicando notación O.
$T(n) = O((c_3)(2^n - 1))$, aplicando la regla de la suma.
$T(n) = O(c_3*2^n - c_3)$
$T(n) = O(c_3*2^n)$, aplicando regla de la suma.
$T(n) = O(2^n)$, aplicando regla del producto.

La complejidad asintótica para el peor de los casos es $O(2^n)$, es decir, exponencial, donde n es la longitud del arreglo.

- **GroupNoAdj:** este ejercicio tiene una implementación similar al anterior, solo que en este caso la excepción será que cuando se escoja un número para realizar la suma, el número siguiente a este no podrá ser elegido. Como caso base se tiene el mismo que el del algoritmo anterior. Como casos recursivos se tiene uno similar al anterior: se realiza un llamado recursivo en el cual el start va de dos en dos, restandole al target el valor que se tiene en esa posición y se realiza otro llamado donde start va de uno en uno y el objetivo es el planteado al inicio, revisando así todas las posibilidades con las condiciones dadas en el ejercicio. El algoritmo retorna true si se cumplen las condiciones dadas (target = 0), de lo contrario, false.

  Complejidad:

  $$T(n) = \begin{cases} c1, donde\ c1 = 5 \\ c2 + T(n-2) + T(n-1), donde\ c2 = 8 \\ c3,\ donde\ c3 = 1 \end{cases}$$

  $T(n) = c_2 + T(n-2) + T(n-1)$

  Como esta complejidad es la misma que la complejidad para el algoritmo recursivo de Fibonacci (realizado anteriormente), entonces se tiene que la complejidad asintótica para el peor de los casos es $O(2^n)$, es decir, exponencial, donde n es la longitud del arreglo.

- **GroupSum5 (3.4 Opcional):** el algoritmo explicado para este ejercicio está en el punto 3.4.

  Complejidad:

  $$T(n) = \begin{cases} c1, donde\ c1 = 5 \\ c2, donde\ c2 = 5 \\ c3 + T(n-1) + T(n-1), donde\ c3\ = 12 \\ c4,\ donde\ c4 = 1 \end{cases}$$

  $T(n) = c_3 + T(n-1) + T(n-1)$
  $T(n) = (c_3)(2^n - 1) + (c_1)(2^{n-1})$, donde c_1 es un parámetro arbitrario.
  $T(n) = O((c_3)(2^n - 1) + (c_1)(2^{n-1}))$, aplicando notación O.
  $T(n) = O((c_3)(2^n - 1))$, aplicando la regla de la suma.

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

$T(n) = O(c_3*2^n - c_3)$
$T(n) = O(c_3*2^n)$, aplicando regla de la suma.
$T(n) = O(2^n)$, aplicando regla del producto.

La complejidad asintótica para el peor de los casos es $O(2^n)$, es decir, exponencial, donde n es la longitud del arreglo.

- **SliptArray:** para este algoritmo, se crea un método privado el cuál recibe un int inicio que servirá como índice en el arreglo, el arreglo dado en ese método, un int suma1 y un int suma2, los cuales será las variables que almacenarán las sumas que tiene de condición el ejercicio. Como caso base se tiene que si inicio es igual a la longitud del arreglo y suma1 es igual suma2, se retorna true, cumpliendo así las condiciones del ejercicio. Como casos recursivos, se hacen dos llamados al método con las mismas indicaciones para inicio y el arreglo que recibe, pero diferentes para las sumas. En un método, se le suma a suma1 el número en la posición inicio, mientras en el otro se le suma a suma2 el número en la posición inicio. Si al final las sumas dan diferentes, se retorna false.

  Complejidad:

$$T(n) = \begin{cases} c1, \ \ donde\ c1 = 6. \\ c2 + T(n-1) + T(n-1), \ donde\ c2 = 10. \\ c3, \ donde\ c3 = 1. \end{cases}$$

$T(n) = c_2 + T(n-1) + T(n-1)$
$T(n) = (c_2)(2^n - 1) + (c_1)(2^{n-1})$, donde c_1 es un parámetro arbitrario.
$T(n) = O((c_2)(2^n - 1) + (c_1)(2^{n-1}))$, aplicando notación O.
$T(n) = O((c_2)(2^n - 1))$, aplicando la regla de la suma.
$T(n) = O(c_2*2^n - c_2)$
$T(n) = O(c_2*2^n)$, aplicando regla de la suma.
$T(n) = O(2^n)$, aplicando regla del producto.

La complejidad asintótica para el peor de los casos del algoritmo que calcula la longitud de la subsecuencia común más larga es $O(2^n)$, en otras palabras, exponencial, donde n es igual a la longitud del arreglo dado.

- **Split53:** en este ejercicio, también se crea un método auxiliar que recibe los mismos parámetros que el método creado en el ejercicio anterior. Además, también se tiene el mismo caso base. Para cumplir con las condiciones del enunciado, en primer lugar, se evalúan dos condiciones y se llama dos veces al método: si el número en una posición determinada es múltiplo de 5, se agrega a suma1, de lo contrario, si el número en esa posición es múltiplo de 3 y no de 5, se agrega a suma2. Por último, si no se cumple ninguna de estas dos condiciones, se hacen dos llamados al método unidos por un or, así como en el ejercicio anterior, para obtener todas las posibles combinaciones de estos números y mirar si se cumplen las condiciones y el caso base. Retorna true si se cumplen y si no, false.

  Complejidad:

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

$$T(n) = \begin{cases} c1, donde\ c1 = 5 \\ c2\ + T(n-1), donde\ c2 = 9 \\ c3 + T(n-1), donde\ c3 = 13 \\ c4 + T(n-1) + T(n-1), donde\ c4\ = 9 \\ c5,\ donde\ c5 = 1 \end{cases}$$

T(n) = c_4 + T(n-1) + T(n-1)
T(n) = (c_4)(2^n – 1) + (c_1)(2^{n-1}), donde c_1 es un parámetro arbitrario.
T(n) = O((c_4)(2^n – 1) + (c_1)(2^{n-1})), aplicando notación O.
T(n) = O((c_4)(2^n – 1)), aplicando la regla de la suma.
T(n) = O(c_4*2^n – c_4)
T(n) = O(c_4*2^n), aplicando regla de la suma.
T(n) = O(2^n), aplicando regla del producto.

La complejidad asintótica para el peor de los casos es O(2^n), es decir, exponencial, donde n es la longitud del arreglo.

3.6 En cada caso, se definió el significado de la variable n y la equivalencia de las constantes.

4) Simulacro de Parcial

4.1

4.1.1 A
4.1.2 C
4.1.3 A

4.2

4.2.1 B
4.2.2 La A, B y C son verdaderas.

4.3

4.3.1 B

4.4

4.4.1 C

4.5

1 A
2 B
4.5.1 C

4.6

4.6.1 sumaAux(n.substring(i+2, n.length()), i);
4.6.2 sumaAux(n.substring(i+1, n.length()), i));

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

4.8(opcional)

Línea 3 = if(T < 0) return 0;
Línea 4 = if(T==0) return 1;
Linea 8 = f1 + f2 + f3

4.8(opcional) C

4.9(opcional) B

4.10(opcional)

```
Public static int lucas(int n){
if(n == 0) return 2;
if(n == 1) return 1;
return lucas(n-2) + lucas(n-1);
```

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