Análisis de Algoritmos, Sem: 2018-2, 3CV1, Práctica 2, 1 de Septiembre de 2018

Práctica 2: Funciones Recursivas vs Iterativas

Reyes Valenzuela Alejandro, Gutiérrez Povedano Pablo.

Escuela Superior de Cómputo
Instituto Politécnico Nacional, México
*areyesv*11@*gmail.com*, *gpovedanop*@*gmail.com*

Resumen: En este documento se compará el rendimiento de dos algoritmos implementados de maneras distintas (utilizando recursividad e iteraciones) para determinar la complejidad temporal de los estos.

Palabras Clave: Recursividad, Iteración, Recurrencia.

1 Introducción

Existen diversas maneras de resolver un problema computacional, por ejemplo, si necesitamos obtener la potencia n de un número, podemos recurrir a una función iterativa, que multiplique n veces nuestro número hasta llegar a lo solicitado por el usuario.

Sin embargo, podemos solucionar este problema de manera recursiva, al multiplicar el número una vez por sí mismo pero a la potencia n-1 (llamando de nuevo a la misma función).

Como podemos observar ambos caminos nos dan solución a nuestro problema, sin embargo, puede ocurrir que un camino tenga mayor complejidad temporal que el otro, y visceversa, por lo que en esta práctica analizaremos situaciones en donde podemos utilizar ambas funciones y el costo que estas producen, para posteriormente compararlas y definir en qué casos conviene utilizar recursividad y en cuáles iteraciones.

2 Conceptos Básicos

Se llama recursividad a un proceso mediante el que una función se llama a sḿisma de forma repetida, hasta que se satisface alguna determinada condición.

El proceso se utiliza para operaciones repetidas en las que cada acción se determina mediante un resultado anterior. Se pueden escribir de esta forma muchos problemas iterativos. Se deben satisfacer dos condiciones para que se pueda resolver un problema recursivamente:

Primera condición: El problema se debe escribir en forma recursiva.

Segunda condición: La sentencia del problema debe incluir una condición de fin.

Para calcular la complejidad de una función recursiva se utiliza la recurrencia, la cual es definida como una ecuación o desigualdad que describe una función en términos de su valor para entradas más pequeñas.

A continuación mostraremos los dos algoritmos propuestos para esta práctica (Desarrollados de manera recursiva y de manera iterativa):

Encontrar el término n de la sucesión de Fibonacci (Iteративamente)

$Fibonacci(n)$:

```
int t1 = 0, t2 = 1, aux=0, i=0
for i ← 0 to i < n do
    aux = t1 + t2
    t1 = t2
    t2 = aux
return aux
```

Encontrar el término n de la sucesión de Fibonacci (Recursivamente)

$Fibonacci(n)$:

if i $= 1$ *or* i $= 0$ *then*
  $return\ 1$
else
  $return\ (Fibonacci(n-1) + Fibonacci(n-2))$

Hallar la suma de los primeros n cubos (Recursivamente)

Algoritmo $S(n)$:

if i $= 1$ *then*
  $return\ 1$
else
  $return\ (S(n-1) + (n * n * n))$

Hallar la suma de los primeros n cubos (Iterativamente)

Algoritmo $S(n)$:

int suma $= 0$, i
for i $\leftarrow 1$ to i $\leq$ n *do*
  $suma = suma + (n * n * n)$
$return\ suma$

3 Experimentación y Resultados

A continuación mostramos capturas de ejecución de los programas

Fibonacci Iterativo

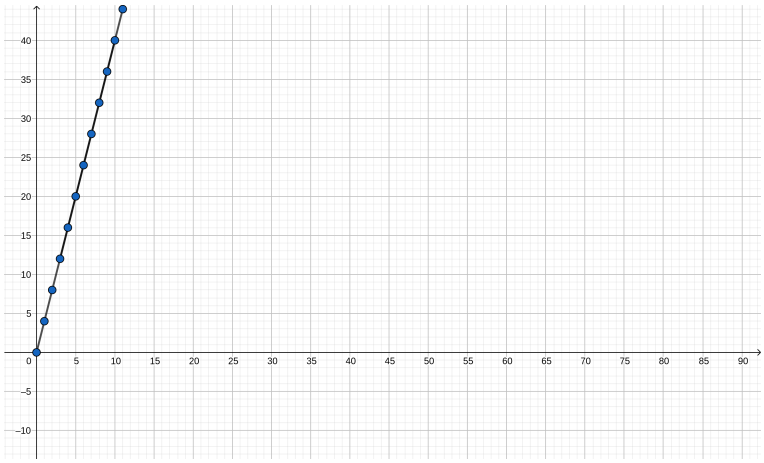
```
El término 4 de la sucesión de Fibonacci es:5
[alejandro@alejandro-pc ~]$ ./fib
Termino de la sucesión?
5
k=20
El término 5 de la sucesión de Fibonacci es:8
[alejandro@alejandro-pc ~]$ ./fib
Termino de la sucesión?
6
k=24
```

Fibonacci Recursivo

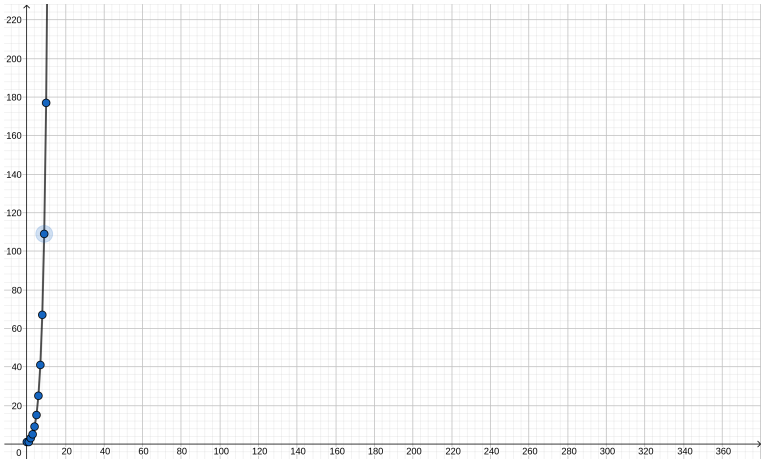
```
[alejandro@alejandro-pc ~]$ ./fibrec
Termino de la sucesión?
2
El término 2 de la sucesión de Fibonacci es:2
k=3
[alejandro@alejandro-pc ~]$ ./fibrec
Termino de la sucesión?
10
El término 10 de la sucesión de Fibonacci es:89
k=177
```

Podemos observar que la función recursiva regresa valores de k bastante altos, a comparación de la iterativa, esto se debe a que la recursiva se llama dos veces a sí mismo cuando no se llega a la condición de frontera. Podemos observar esto al graficar las funciones tras solicitar los primeros once términos de la sucesión.

Fibonacci Iterativo



Fibonacci Recursivo



Mientras que la gráfica del algoritmo iterativo corresponde a una recta, la del recursivo aumenta de manera exponencial, por lo que podemos decir experimentalmente que el algoritmo recursivo tiene un mayor orden de complejidad a comparación de el algoritmo iterativo.

A continuación demostraremos lo explicado anteriormente de manera formal:

Fibonacci Iterativo

int t1 = 0, t2 = 1, aux=0, i=0	C_1	1
for i ← 0 to < n *do*	C_2	$n+1$
aux = t1 + t2	C_3	n
t1 = t2	C_4	n
t2 = aux	C_5	n
return aux	C_6	1

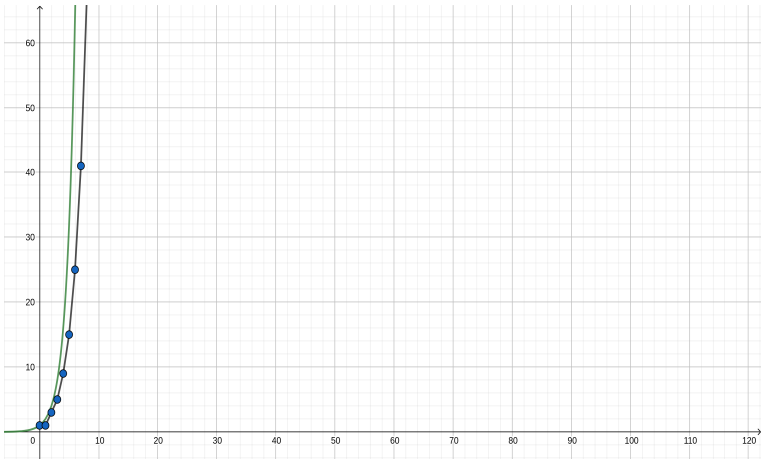
El tiempo en función de n está dado por:

$$T(n) = C_1 + C_2{*}(n+1) + C_3{*}n + C_4{*}n + C_5{*}n + C_6$$

Agrupando los términos semejantes:

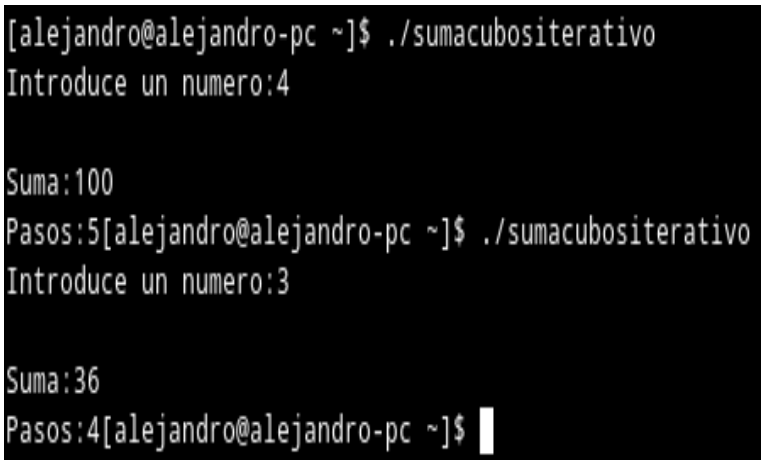
$$T(n) = n(C_2{+}C_3{+}C_4{+}C_5) + (C_1 + C_2 + C_6)$$

Al ser n el término de mayor exponente (en este caso 1), podemos concluir que el algoritmo iterativo tiene órden lineal. Ahora acotaremos la función recursiva con la función 2^n (Gráfica verde).

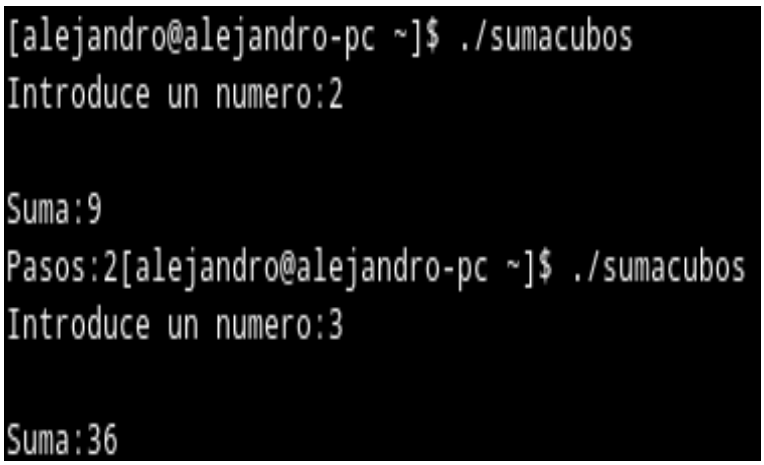


Como podemos observar, la función 2^n acota a la función recursiva, por lo que podemos decir que su orden de complejidad es exponencial (2^n).

Suma de Cubos Iterativo

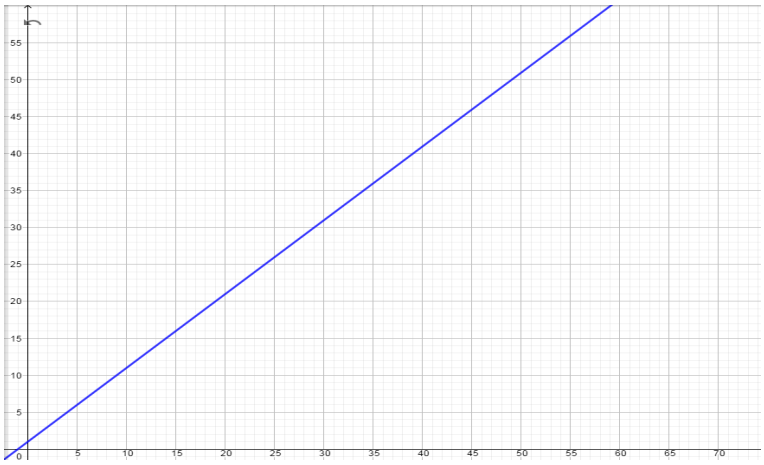


Suma de Cubos Recursivo

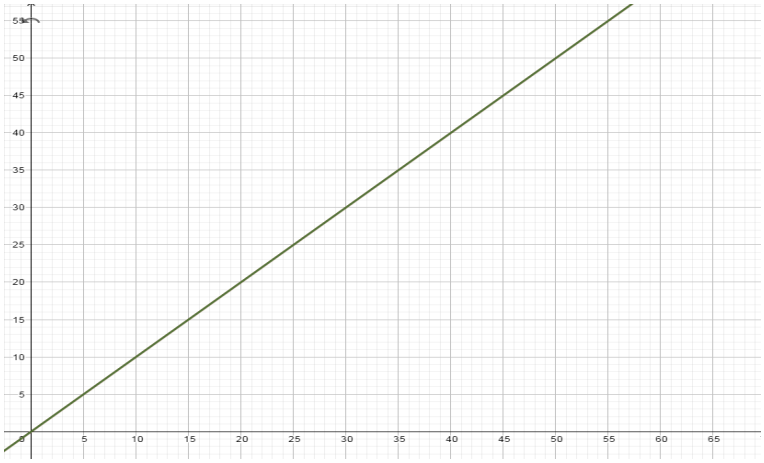


Podemos observar que nuestras funciones obtienen valores de manera lineal, por lo que podríamos deducir que ambos algorítmos tienen órden lineal, para sustentar esto, se procederá a comprobar esto de manera gráfica y de manera formal.

Suma de Cubos Iterativo



Suma de Cubos Recursivo



Podemos observar que generan la misma gráfica de una línea recta, por lo que gráficamente decimos que es lineal. Ahora procederemos a comprobar esto mediante la definición.

Suma de Cubos Iterativo

int suma = 0, i	C_1	1
for i $\leftarrow$ 1 to i $\leq$ n *do*	C_2	$n+1$
$suma = suma + (n * n * n)$	C_3	n
return suma	C_4	1

Calculando $T(n)$:

$$T(n) = C_1 + C_2{*}(n+1) + C_3{*}n + C_4$$

Agrupando términos semejantes:

$$T(n) = n(C_2{+}C_3) + (C_1 + C_2 + C_4)$$

Suma de Cubos Recursivo

Algoritmo $S(n)$:

```
if i = 1 then
   return 1
else
   return (S(n − 1) + (n ∗ n ∗ n))
```

Definimos nuestra condición de frontera:

$$T(n) = \begin{cases} 1 & si \quad n = 1 \\ \\ T(n-1) + 1 & si \quad n > 1 \end{cases}$$

Por lo que tenemos que:

$$\begin{aligned} T(n) &= T(n \text{ - } 1) + 1 \\ &= T(n \text{ - } 1 \text{ - } 1) + 1 + 1 \\ &= T(n \text{ - } 2) + 2 \\ &= T(n \text{ - } 2 \text{ - } 1) + 2 + 1 \\ &= T(n \text{ - } 3) + 3 \\ &= ... \\ &= T(n \text{ - } i) + \text{i} \end{aligned}$$

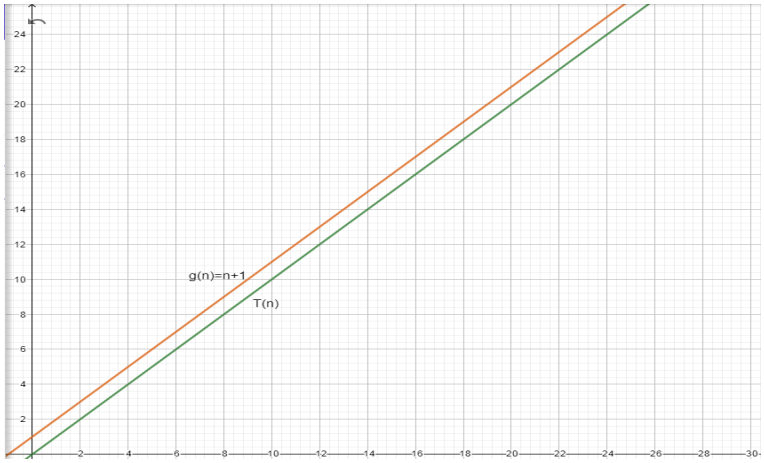
Donde $n - i = 1$; por lo que $i = n - 1$, posteriormente sustituimos:

$$\begin{aligned} &= T(1) + n \text{ - } 1 \\ &= 1 + n \text{ - } 1 \\ &= n \end{aligned}$$

Por lo que podemos concluir que $T(n)$ tiene órden lineal. Ahora acotaremos las funciones para demostrar que $T(n)\ \epsilon\ O(g(n))$ en ambas implementaciones.

Suma de Cubos Iterativo

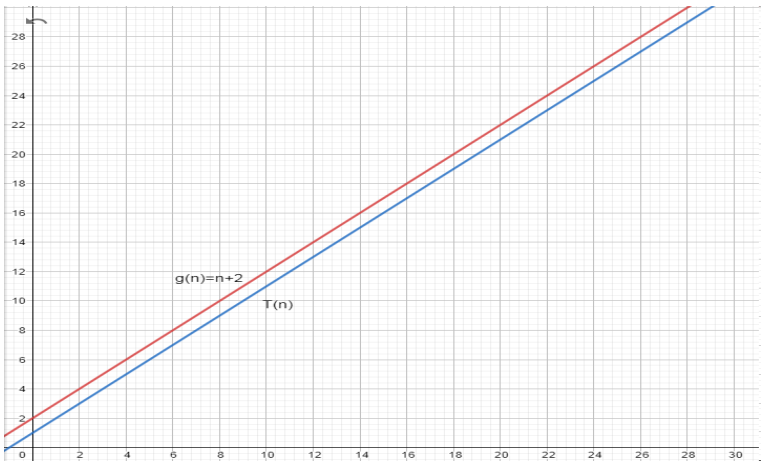
Para esta función proponemos a $g(n) = n + 1$



Como podemos ver $g(n)$ acota a $T(n)$, por lo que podemos decir que $T(n)$ ϵ $O(h(n))$.

Suma de Cubos Recursivo

Para esta función proponemos a $g(n) = n + 2$



Como podemos ver $g(n)$ también acota a $T(n)$, por lo que podemos decir que $T(n)$ ϵ $O(h(n))$.

4 Conclusiones

Conclusión General: Pudimos ver que a pesar de tener el mismo objetivo, las implementaciones provocaron que en ciertos casos el tiempo computacional sea mayor, a pesar de contar con menos instrucciones, situaciones como esta nos lleva a tener que considerar mejor la implementación del algoritmo al momento de proponerlo para solucionar un problema.

Conclusión Gutiérrez Povedano: Con el análisis de los algoritmos de la práctica implementados de forma iterativa y recursiva podemos notar que el orden de complejidad de algunos algoritmos no cambia como en el ejercicio 2 pero en el ejercicio 1 la complejidad pasa de ser lineal a exponencial.

Conclusión Reyes Valenzuela: Pude observar que no hubo complicaciones para programar los algoritmos, ya que estos ya contaban con un pseudocódigo definido, esto nos ayudo a poder identificar la complejidad de ambos algoritmos de una manera más rápida, en donde a pesar de dar solución al mismo problema, el tiempo cambió de manera drástica.

5 Anexo

Calcular el orden de complejidad de algoritmo de la burbuja (BubbleSort).

$BubbleSort(A)$

for i $\leftarrow$ 1 to i $\leq$ A.length $-$ 1
  for j = A.length *downto* j $\leq$ i + 1 *do*
    if A[j] $<$ A[j $-$ 1] *then*
    exchange A[j] *with* A[j $-$ 1]

Ahora calcularemos la complejidad de este algoritmo, para eso asignamos las constantes de tiempo y definimos el número de veces que se ejecutarán las instrucciones

for i $\leftarrow$ 1 to i $\leq$ A.length $-$ 1	C_1	$n+1$
for j = A.length *downto* j $\leq$ i + 1 *do*	C_2	$\sum_{i=0}^{n-1} t_i$
if A[j] $<$ A[j $-$ 1] *then*	C_3	$\sum_{i=0}^{n-1} t_i$ - 1
exchange A[j] *with* A[j $-$ 1]	C_4	$\sum_{i=0}^{n-1} t_i - 1$

Calcularemos el valor de las sumatorias:

i	t_i
1	n-1
2	n-2
3	n-3
4	n-4
5	n-5

Por lo que tenemos que $t_i = n - i$

Ahora, el algoritmo está definido por $T(n)$, quién a su vez está conformado por:

$$T(n) = C_1{*}(n+1) + C_2{*}\sum_{i=0}^{n-1} t_i + C_3{*}\sum_{i=0}^{n-1}(t_i - 1) + C_4{*}\sum_{i=0}^{n-1}(t_i - 1)$$

Sustituyendo:

$$T(n) = C_1{*}(n+1) + C_2{*}\sum_{i=0}^{n-1}(n-i) + C_3{*}\sum_{i=0}^{n-1}(n-i-1) + C_4{*}\sum_{i=0}^{n-1}(n-i-1)$$

Como $\sum_{i=0}^{n-1}(n-i-1) = \frac{1}{2}(n-1)n$ y $\sum_{i=0}^{n-1}(n-i) = \frac{1}{2}(n+1)n$, entonces $T(n)$ queda de la siguiente forma:

$$T(n) = C_1{*}n + C_1 + C_2{*}\tfrac{1}{2}(n+1)n + C_3{*}\tfrac{1}{2}(n-1)n + C_4{*}\tfrac{1}{2}(n-1)n$$

Desarrollando y agrupando términos semejantes (las constantes al multiplicarse dan como resultado otra constante, por lo que podemos representarlas en una sola):

$$T(n) = C_1{*}n + C_1 + C_2{*}\tfrac{n^2}{2} + C_2{*}\tfrac{n}{2} + C_3{*}\tfrac{n^2}{2} - C_3{*}\tfrac{n}{2} + C_4{*}\tfrac{n^2}{2} - C_4{*}\tfrac{n}{2}$$

$$T(n) = n^2(C_2 + C_3 + C_4) + n(C_2 - C_3 - C_4) + (C_1)$$

Observamos que nuestro algoritmo en el peor de los casos se torna cuadrático, por lo que decimos que $BubbleSort \in O(n^2)$.

6 Bibliografía

Lcc.uma.es. (2018). [online] Available at: http://www.lcc.uma.es/ alvarezp/pm/recursividad.pdf [Accessed 4 Sep. 2018].

Ccc.inaoep.mx. (2018). [online] Available at: https://ccc.inaoep.mx/ jagonzalez/ADA/Recurrencias.pdf [Accessed 6 Sep. 2018].