Análisis de Algoritmos, Sem: 2019-1, 3CV1, Práctica 1, 29/08/2018

Práctica 2: Funciones Recursivas vs Iterativas.

Hernández Castellanos César Uriel, Aguilar Garcia Mauricio

Escuela Superior de Cómputo
Instituto Politécnico Nacional, México
*uuriel*12009*u*@*gmail.com*, *mauricio.aguilar.garcia*,90@*gmail.com*

Resumen: La práctica consiste en ver las diferencias entre algoritmos implementados de manera recursiva o iterativa así como su complejidad.

Palabras Clave: Algoritmo, Complejidad, Recursiva, Iterativa, Fibonacci y Suma.

1. Introducción

Los algoritmos, como ya sabemos, son un conjunto de pasos ordenados, finitos, los cuales tienen una entrada y una salida, y su función es resolver un problema. La resolución de un problema se puede dar de diferentes maneras, por lo cual existen diferentes algoritmos que realizan una misma función y a su vez, existen diferentes maneras de implementar un algoritmo, como por ejemplo de manera iterativa o recursiva.

Para el diseño recursivo es necesario establecer la función recursiva, correspondiente al algoritmo; porteriormente de diseña el caso directo, el cual contiene la condición de frontera o de paro, y el caso recursivo, en el cual se obtiene mediante la definición de la función recursiva. Para el diseño iterativo es necesario determinar el inicio de la iteración, la condición de continuación y las intrucciones que se realizan en cada iteración, aparte de las instrucciones (si son requeridas) al término de las iteraciones.[1][2]

En este reporte podremos notar las diferencias entre estas maneras de implementar los algoritmos, lo cual puede influir en la complejidad de un algoritmo de manera extraordinaria y como consecuencia la cantidad de tiempo necesaria para que el algoritmo cumpla su función.

La manera en que lo veremos es implementando el algoritmo de fibonacci para encontrar n números, de manera recursiva e iterativa, así como la suma de los primeros cubos de un número también de manera recursiva e iterativa.

2. Conceptos básicos

Definición de recursividad

Recurrencia, recursión o recursividad es la forma en la cual se especifica un proceso basado en su propia definición, siendo esta caracterÍstica discernible en términos de autorreferencialidad, autopoiesis, fractalidad, o, en otras palabras, construcción a partir de un mismo tipo. Con ánimo de una mayor precisión, y para evitar la aparente circularidad en esta definición, se formula el concepto de recursión de la siguiente manera: Un problema que pueda ser definido en función de su tamaño, sea este N, pueda ser dividido en instancias más pequeñas (N) del mismo problema y se conozca la solución explícita a las instancias más simples, lo que se conoce como casos base, se puede aplicar inducción sobre las llamadas más pequeñas y suponer que estas quedan resueltas.[3]

Figura 1: Recursividad

Iterativo

Cualidad o característica de un algoritmo, procedimiento o función de calcular un valor especificado repitiendo muchas veces un conjunto de instrucciones. Al igual que la recursividad se utiliza para implementar estructuras de repetición.

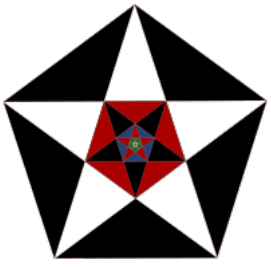

Figura 2: Pentágono iterativo

Sucesión de Fibonacci

La sucesión de Fibonacci es una serie de números que se obtiene por adición de lo dos números anteriores, obteniéndose la serie: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, etc. Esta serie se hizo famosa por describir las proporciones naturales de todas las cosas en el universo. Lo cierto es que hoy en día la aparición de esta serie ocurre de forma muy amplia y extensa en campos como las matemáticas, ciencias de la computación, biología o teoría de juegos

Figura 3: Sucesión de Fibonacci en la naturaleza

3. Experimentación y Resultados

Problema uno

Implementar la sucesión de Fibonacci mediante un algoritmo recursivo y mediante unalgoritmo iterativo.

Se hizo uso del lenguaje de programación Java, para lo que se definició una clase llamada SucesionDeFibonacci, la cual contiene únicamente dos atributos (número A y número B) de tipo entero que nos auxiliaran en la implementación del algoritmo iterativo de la sucesión de Fibonacci.

Se implementan tres métodos, imprimirFibonacciIterativo(), mostrarNFibonacci() e imprimirFibonacciRescursivo() a continuación se muestra su implementación.

```java
package practicados;

public class SucesionDeFibonacci {
    private int numA, numB;

    public SucesionDeFibonacci() {
        this.numA = 0;
        this.numB = 1;
    }

    public void imprimirFibonacciIterativa(int numeroDeTerminos) {
        System.out.print(numA + " ");
        for (int i = 2; i <= numeroDeTerminos; i++) {
            System.out.print(numB + " ");
            numB = numA + numB;
            numA = numB - numA;
        }
    }
    public int mostrarNFibonacci(int numeroDeTerminos) {
        if (numeroDeTerminos == 0 || numeroDeTerminos == 1) {
            return numeroDeTerminos;
        } else {
            return mostrarNFibonacci(numeroDeTerminos - 1) + mostrarNFibonacci(numeroDeTerminos - 2);
        }
    }
    public void imprimirFibonacciRecursivo(int numeroDeTerminos) {
        for (int i = 0; i < numeroDeTerminos; i++) {
            System.out.print(mostrarNFibonacci(i)+" ");
        }
    }

}
```

Figura 4: Implementación de la sucesión de Fibonacci

Mostrar los algoritmos con la notación vista en clase (pseudocódigo).

Algorithm 1 Algoritmo iterativo

```
1: procedure IMPRIMIRFIBONACCIINCREMENTAL(numeroDeTerminos)
2:
3:     numA ← 1
4:
5:     numB ← 1
6:
7:     mostrarNumA()
8:
9:     for i ← 2, i <= numeroDeTerminos do
10:
11:        mostrarNumB()
12:        numB ← numA + numB
13:        numA ← numB − numA
14:
15:    end for
16:
17: end procedure
```

Algorithm 2 Algoritmo recursivo

```
1: procedure IMPRIMIRFIBONACCIRECURSIVO(numeroDeTerminos)
2:
3:     for i ← 0, i < numeroDeTerminos do
4:
5:         mostrarNFibonacci(i)
6:
7:     end for
8:
9: end procedure
10:
11: procedure MOSTRARNFIBONACCI(n)
12:
13:    if n == 0 || n == 1 then
14:
15:        return n
16:
17:    else
18:
19:        return mostrarNFibonacci(n-1)+ mostrarNFibonacci(n-2)
20:
21:    end if
22:
23: end procedure
```

Demostrar formalmente que el algoritmo iterativo tiene orden lineal.

Algorithm 1 Algoritmo iterativo

```
1: procedure IMPRIMIRFIBONACCIINCREMENTAL(numeroDeTerminos)
2:
3:     numA ← 1
4:
5:     numB ← 1
6:
7:     mostrarNumA()
8:
9:     for i ← 2, i <= numeroDeTerminos do
10:
11:        mostrarNumB()
12:        numB ← numA + numB
13:        numA ← numB − numA
14:
15:    end for
16:
17: end procedure
```

$$
\begin{aligned}
&numA \leftarrow 1 && O(1)\\
&numB \leftarrow 1 && O(1)\\
&mostrarNumA() && O(1)\\
&for\ i \leftarrow 2; i < numeroDeTerminos + 1\ do\\
&\quad mostrarNumB()\\
&\quad numB \leftarrow numA + numB && O(n)\\
&\quad numB \leftarrow numA + numB\\
&end\ for\\
&\qquad O(n * 1 * 1 * 1) = O(n)\\
&\qquad \therefore S.F.I \in O(n)
\end{aligned}
$$

Figura 5: Demostración de la complejidad del algoritmo de la sucesión de Fibonacci iterativo

A partir de gráficas experimentalmente, proponer el orden de complejidad para el algoritmo recursivo e iterativo.

Mostrar mediante gráficas que el algoritmo para calcular la sucesión de Fibonacci en forma iterativa tiene complejidad lineal

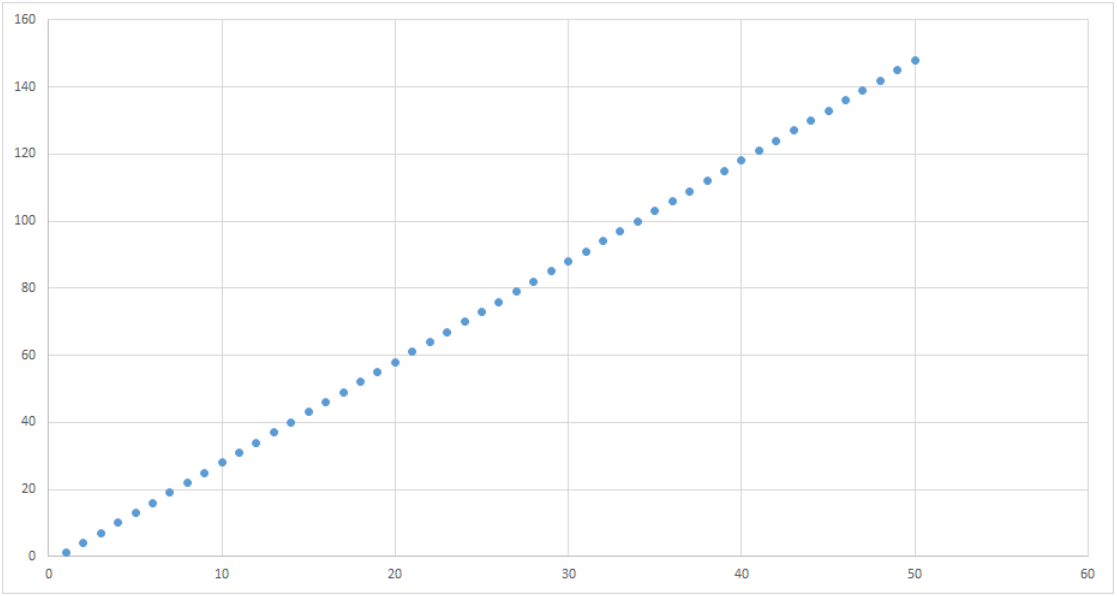

Figura 6: Gráfica del algoritmo que genera la serie de Fibonacci de manera iterativa

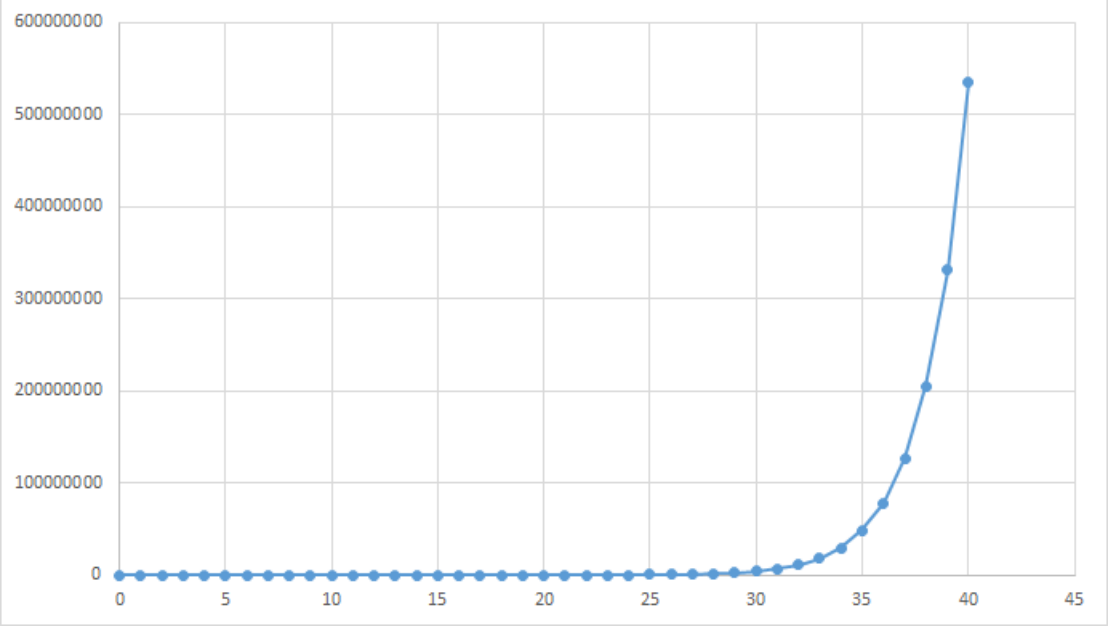

Figura 7: Gráfica del algoritmo que genera la serie de Fibonacci de manera recursiva

```
Generando sucesión de Fibonacci de manera recursiva...
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181
Generando sucesión de Fibonacci de manera iterativa...
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181

BUILD SUCCESSFUL (total time: 0 seconds)
```

Figura 8: Salida del algoritmo que genera la sucesión de Fibonacci como recursivo e iterativo

Problema dos

Implementar el siguiente algoritmo:

```
Algoritmo SumaDeCubos(n)
Input: Un entero positivo n
Output: La suma de los primeros n cubos
    if n = i return 1
    else return S(n-1) + n*n*n
```

i) A partir de las gráficas, proponga una función $g(n)$ tal que $T(n) \in O(g(n))$ con $T(n)$ el tiempo computacional del algoitmo
ii) Implemente un algoritmo de manera iterativa y apartir de gráficas, proponga una función $g(n)$ tal que $T(n) \in O(g(n))$ con $T(n)$ el tiempo computacional del algoritmo en el caso iterativo
iii) Calcule el tiempo computacional del algoritmo recursivo e iterativo de manera formal. Compare resultados

La gráfica nos muestra que el algoritmo tiene orden de complejidad lineal, es decir, $T(n) \in \Theta(n)$

Para el algoritmo iterativo utilizamos el siguiente código:

```
Algoritmo SumaDeCubos(n)
  Input: Un entero positivo n
  Output: La suma de los primeros n cubos
          acumulador = 0;                    \\C1 = 1
    while n                              \\C2 = n
       acumulador = acumulador + n*n*n    \\C3 sum(2,n,1)
       n = n - 1;
    return acumulador
```

Donde $sum(a, b, c)$ representa a la sumatoria desde a hasta b de c

El cual nos dió los siguientes valores:

Con estos valores podemos decir que el Algoritmo de SumaDeCubos Iterativo tiene orden lineal,
es decir, $T(n) = 2n \in \Theta(n)$
Para calcular de manera formal la complejidad del Algoritmo de SumaDeCubos Iterativo utilizaremos las constantes C_n

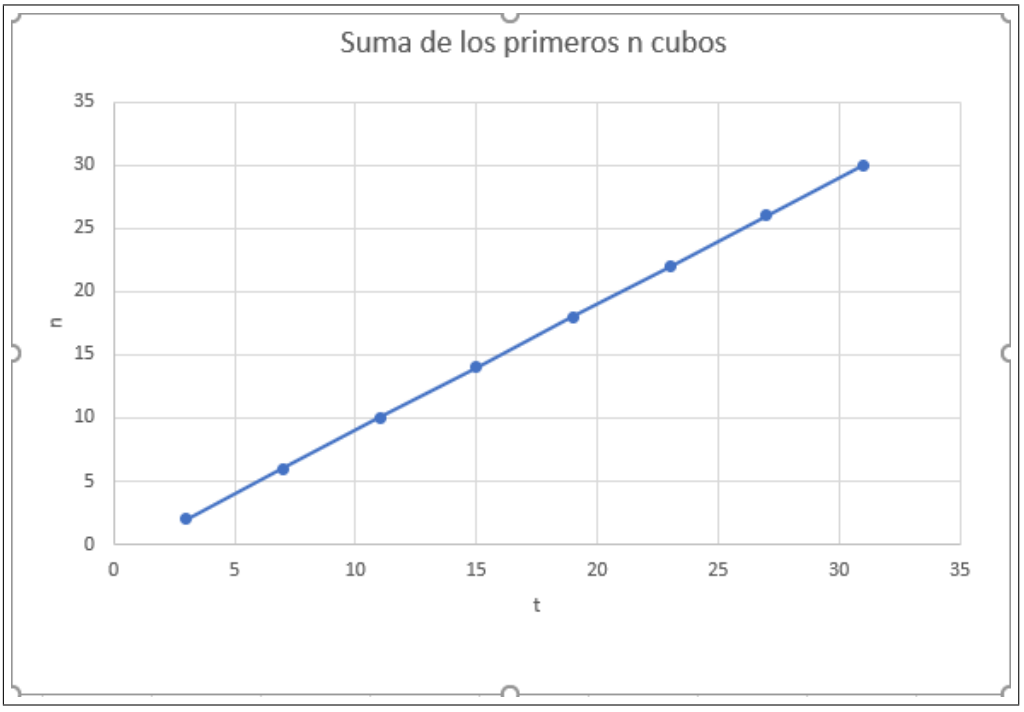

Figura 9: Gráfica del Algoritmo de SumaDeCubos Recursivo

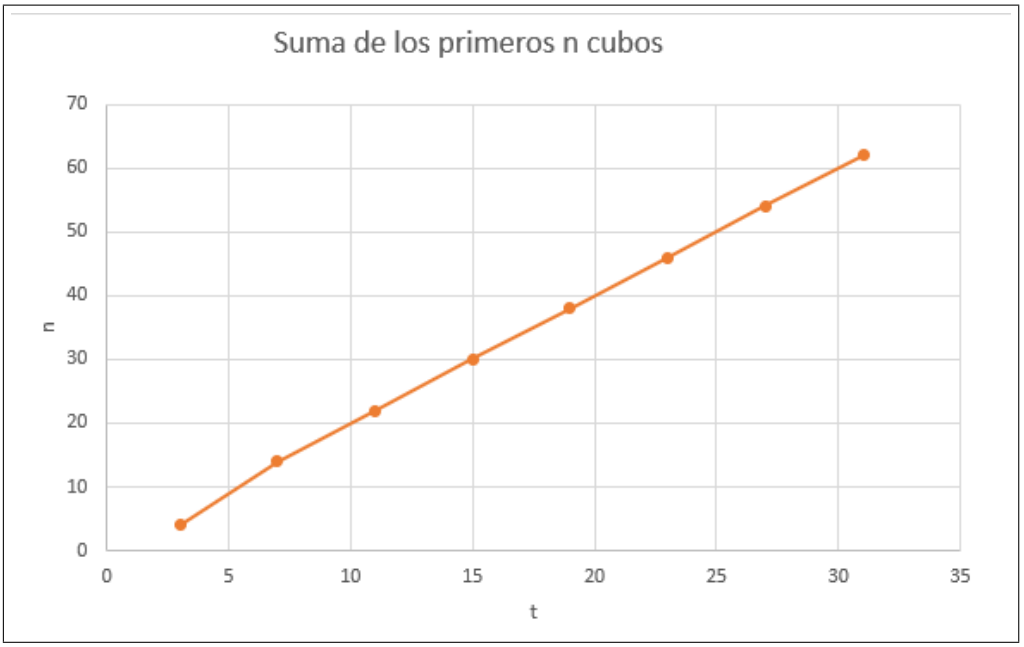

Figura 10: Gráfica del Algoritmo de SumaDeCubos Iterativo

$$T(n) = C1 + C2n + \sum_{i=2}^{n} C3$$

$$T(n) = C1 + C2n + C3(n-1)$$

$$T(n) = (C2 + C3)n + C1 - C3$$

Aquí el mejor y el peor de los casos es el mismo

Si consideramos el valor de cada constante como 1, tendremos

$$T(n) = 2n$$

Entonces $T(n)$ tiene la forma: $an + b$ y decimos que $T(n)\ \epsilon\ \theta(n)$

Figura 11: Calculando la complejidad del Algoritmo de SumaDeCubos Iterativo

La definición dice:

$$\theta\big(g(n)\big) = \{f(n) \mid \exists_n\, c1, c2\ > 0\ y\ n_0 > 0\ \ \ni\ \ 0 \leq c1g(n) \leq f(n) \leq c2g(n)\ \forall_{n \geq n_0}\}$$

Se tiene

$$f(n) = 2n\ \leq c2g(n)$$

$$f(n) = 2n\ \leq 4n\ \ \forall_{n \geq 1}$$

Por otro lado

$$f(n) = 2n\ \geq c1g(n)$$

$$f(n) = 2n\ \geq n\ \forall_{n \geq 1}$$

Luego tomando $c1 = 4, c2 = 1\ y\ n_0 = 1$, se tiene

$$c1g(n) \leq f(n) \leq c2g(n)$$

$$n \leq 2n \leq 4n \quad \therefore \quad \boldsymbol{f(n) = 2n\ \epsilon\ \theta(n)}$$

Figura 12: Demostrando que la complejidad del Algoritmo de SumaDeCubos Iterativo es de orden lineal

Ahora tenemos que demostrar que lo anterior es cierto

Para calcular el tiempo computacional de manera formal para el Algoritmo de SumaDeCubos Recursivo, tenemos que apreciar que para n=0, el algoritmo no hace nada y cuando se manda a llamar ella misma junto a la suma del n-ésimo cubo, tiene un coste de 1.

$$M(n) = \begin{cases} 0; & n = 0 \\ M(n-1) + 1 & n \geq 1 \end{cases}$$

Hacemos uso del método de sustitución hacia atrás para calcular la complejidad

i	Xi
0	$M(n)$
1	$M(n-1)+1$
2	$M(n-2)+2$
3	$M(n-3)+3$
i	$M(n-i)+i$

Ahora sustituyendo;

$$M(n-(n)) + (n)$$
$$M(0) + n = n$$

Por lo tanto $\boldsymbol{M(n) \epsilon \theta(n)}$

Condición de paro $n = 0$, entonces

$$n - i = 0$$
$$i = n$$

Figura 13: Calculando de manera formal la complejidad del Algoritmo de SumaDeCubos Recursivo

Para concluir se conocio que cada los dos tipos de algortimos son lineales pero el recursivo tiende a tener un tiempo menor como se ve en la siguiente grafica el color azul y gris, representan el recursivo y la acotación de este respectivamente y la naranja y amarrillo marcan el iterativo y su acotación respectivamente.

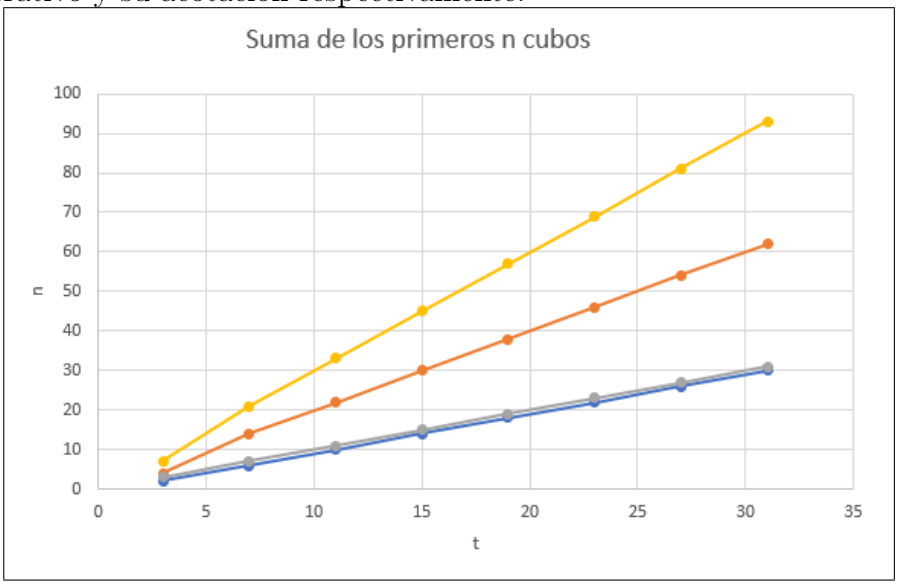

Figura 14: Resultado de ambas funciones conjuntas y sus acotaciones

4. Conclusiones

Conclusión general

En esta práctica pudimos verificar que hay algoritmos en los que en su versión iterativa y recursiva tienen diferente complejidad, mientras que hay algoritmos que en ambas versiones tienen la misma complejidad. Esto nos obliga en cierta manera a siempre realizar el análisis de los algoritmos para estar seguros que realmente escogimos la mejor versión, ya sea en su forma iterativa o recursiva.

Aguilar Garcia Mauricio

En esta práctica me pude dar cuenta de que las complejidades de los algoritmos pueden o no cambiar dependiendo si se realizan de manera iterativa o recursiva, como consecuencia de ésto es necesario realizar siempre el análisis para determinar qué método es el más adecuado para el algoritmo que se desea implementar.

Hernández Castellanos César Uriel

Los algoritmos recursivos e iterativos pueden tener complejidad diferente, mientras que en otras ocasiones ambos tendrán el mismo orden de complejidad, para calcular el orden de complejidad de ls algoritmos iterativos es necesario calcularlo linea por linea o por bloques, en cambio los algoritmos recursivos se le calcula el orden de complejidad estableciendo una ecuación de recurrencia de la cual obtendremos su orden de complejidad.

5. Anexo

5.1. Bubble Sort

Dado el algoritmo de ordenamiento Bubble Sort, cáculamos su complejidad, de la siguiente manera.
Sea A.*length* = n,

BubbleSort(A)		
for $i \leftarrow 1$ to $i \leq A.length - 1$ do	c_1	$n+1$
for $j \leftarrow A$.length downto $j \geq i+1$ do	c_2	$\sum_{i=0}^{n} t_i$
if A[j] $\leq$ A[j-*1*] then	c_3	$\sum_{i=0}^{n} (t_i - 1)$
exchange A[j *with* A[k]	c_4	$\sum_{i=0}^{n} (t_i - 1)$

Sea t_i la cantidad de veces que se ejecuta el for interno.
El peor caso se presenta cuando A está ordenado de manera decreciente.
Se tiene que

i	t_i
1	n–1
2	n–2
3	n–3
4	n–4
i	n–i

$\Rightarrow t_i = n - i$

$$T(n) = c_1 n + c_2(n-1) + c_3 \sum_{i=0}^{n-1} t_i + c_4 \sum_{i=0}^{n-1} (t_i - 1) + c_5 \sum_{i=0}^{n-1} (t_i - 1) + c_6(n-1)$$

sustituyendo t_i

$$T(n) = c_1 n + c_2(n-1) + c_3 \sum_{i=0}^{n-1} (n-i) + c_4 \sum_{i=0}^{n-1} (n-i-1) + c_5 \sum_{i=0}^{n-1} (n-i-1) + c_6(n-1)$$

simplificando y renombrando las constantes, se tiene
$T(n) = an^2 = bn + c = O(n^2)$
Cuando el arreglo esta ordenado de manera decreciente, Bubble Sort $\in O(n^2)$.
En cambio cuando el arreglo esta ordenado de manera ascendente, Bubble Sort $\in \Omega(n)$.

6. Bibliografía

[1]Brassard, Gilles; Bratley, Paul (1997). Fundamentos de Algoritmia. Madrid: PRENTICE HALL.

[2] Introduction to Algorithms, Second Edition by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein