ANÁLISIS DE ALGORITMOS

Contenido:

Análisis de Algoritmos

1. Introducción

Cuando se implementa un algoritmo es importante considerar su eficiencia, por lo tanto es necesario realizar un análisis que permita predecir el tiempo computacional que utiliza para resolver un problema, o los recursos que requiere, por ejemplo, la memoria (Cormen, 2001).

Para medir la eficiencia de un algoritmo podría considerarse la posibilidad de medir el tiempo de ejecución del programa en segundos y el espacio de memoria que ocupa en bytes, sin embargo estas opciones presentan una serie de desventajas (Franch, 1999):

- Son poco precisas, por ejemplo, si un programa ocupa 1 MB y tarda 10 s en ejecutarse, ¿Se puede decir si es eficiente o no?
- Son a posteriori, ya que debe contarse con el código ejecutable para estudiar su eficiencia.
- Dependen de diversos factores como el hardware subyacente, el sistema operativo, el compilador, el lenguaje o los datos de entrada y su configuración.

Para lograr la independencia de los factores antes mencionados, el estudio de la eficiencia de un algoritmo y sus estructuras de datos se debe enfocar en los datos de entrada del programa.

De aquí podemos deducir tres enfoques diferentes para el análisis de algoritmos (Brassard, 1996):

Enfoque Empírico: También llamado enfoque *a posteriori*, consiste en programar el algoritmo y probarlo para diferentes instancias del problema con ayuda de un computador.

Enfoque Teórico: También llamado *a priori*, consiste en determinar matemáticamente la cantidad de recursos utilizados por el algoritmo mediante una función del tamaño de las instancias consideradas.

Enfoque Híbrido: Es una mezcla de los enfoques teórico y empírico. La eficiencia del algoritmo es determinada de forma teórica y luego el programa es probado con cierta entrada en una máquina determinada.

Se estudiará el enfoque teórico. Entre las formas existentes para realizar teóricamente el análisis algorítmico podemos mencionar las dos más relevantes: El análisis temporal y el análisis espacial.

El análisis temporal se enfoca en el estudio del tiempo de ejecución de un algoritmo con respecto al tamaño de su entrada. El análisis espacial se enfoca en la cantidad de memoria utilizada para almacenar las estructuras de datos utilizadas por un algoritmo. Ambos tipos de análisis se estudiarán en las secciones posteriores.

2. Tamaño de la Entrada de un Algoritmo

Este va a depender del problema estudiado (Cormen, 2001), por ejemplo, si se realiza un ordenamiento de elementos la forma natural de medir los datos de entrada es la cantidad de elementos a ordenar; si

la entrada del algoritmo es un grafo, el tamaño de su entrada puede ser descrito por la cantidad de vértices y arcos del grafo; si se desea resolver el factorial de un número el tamaño de la entrada será la magnitud del valor al cual se le calculará el factorial.

3. Complejidad en Tiempo (Tiempo de Ejecución de un Algoritmo)

La complejidad en tiempo o tiempo de ejecución de un algoritmo se refiere a la rapidez con la cual el éste resuelve un determinado problema. Esta complejidad depende de varios factores (Aho, 1983):

a) Los datos de entrada.
b) La calidad del código creado por el compilador.
c) La naturaleza y velocidad de las instrucciones de la máquina utilizada para ejecutar el programa.
d) La complejidad en tiempo del algoritmo subyacente en el programa.

Los puntos b y c son dependientes de las características de *hardware* y *software* de la máquina en la cual se ejecute el algoritmo, por lo tanto no deben ser tomados en cuenta si se desea realizar un análisis objetivo del algoritmo, y por esta misma razón, la unidad de medida utilizada no será en función de segundos.

El enfoque para calcular el tiempo de ejecución de un algoritmo entonces será sobre los puntos a (sobre casos grandes de entrada) y d.

De acuerdo a lo anteriormente explicado, el tiempo de ejecución de un algoritmo con respecto a una entrada en particular, es el número de operaciones elementales o pasos ejecutados (Cormen, 2001).

Una operación elemental se define como un paso computacional cuyo costo está acotado superiormente por una constante independiente de los datos de entrada o del algoritmo utilizado, por ejemplo, operaciones aritméticas (adición, substracción, multiplicación y división), asignaciones, comparaciones y operaciones lógicas, entre otras (Alsuwaiyel, 2003).

Como se expresó anteriormente, la cantidad de tiempo que tarda una instrucción en ejecutarse depende de la máquina en la que se ejecute. Para lograr la independencia de la máquina se definirá, por conveniencia, que el tiempo de ejecución de una instrucción elemental es constante. De acuerdo a esto, el tiempo de ejecución de un algoritmo es la suma de los tiempos de ejecución de cada instrucción elemental ejecutada y se denota como $T(n)$, siendo n el tamaño de la entrada del algoritmo.

Formalmente:

> El tiempo de ejecución o complejidad en tiempo de un algoritmo es una función matemática, T: N $\rightarrow$ N, donde $T(n)$ es el número de pasos realizados por el algoritmo en función de una entrada n (Sipser, 2006).

Se pueden realizar diversos estudios sobre la complejidad en tiempo de un algoritmo, entre ellos podemos nombrar los siguientes:

a) Peor Caso: Corresponde al mayor tiempo de ejecución del algoritmo, entre todos los posibles tiempos de ejecución, para una entrada de tamaño *n*.

b) Mejor Caso: Corresponde al menor tiempo de ejecución del algoritmo, entre todos los posibles tiempos de ejecución, para una entrada de tamaño *n*.

c) Caso Promedio: Corresponde al tiempo promedio de ejecución del algoritmo, para una entrada de tamaño *n*.

4. Funciones Asintóticamente Equivalentes

Sean $f(n)$ y $g(n)$ dos funciones positivas para un n relativamente grande, se dice que son asintóticamente equivalentes o asintóticas si se cumple que:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1$$

En este caso se escribe $f(n) \sim g(n)$; esto significa que ambas funciones crecen de la misma forma cuando n es suficientemente grande. Por ejemplo, si $f(n) = 6n^3 + 2n^2 + 20n + 45$, $g(n) = 6n^3$ es asintóticamente equivalente a $f(n)$.

Si se toma en cuenta que $T(n)$ es el tiempo de ejecución de un algoritmo, entonces $g(n)$ es una función que mide el tiempo de ejecución asintótico del algoritmo.

5. Orden de Crecimiento

Al calcular $T(n)$ se realiza un conteo preciso del número de operaciones del algoritmo, tarea que puede resultar complicada y hasta imposible en muchas ocasiones, y brinda más información de la que realmente se necesita para analizar la eficiencia de un algoritmo para casos grandes. Por esta razón es mejor tomar en cuenta una abstracción de $T(n)$ llamada orden de crecimiento (Alsuwaiyel, 2003).

El orden de crecimiento o tasa de crecimiento de un algoritmo es una función $g(n)$, asintóticamente equivalente a $T(n)$, es decir, sólo toma en cuenta el término de mayor orden dentro de la función $T(n)$, descartando así sus términos de menor orden, en la que además también se descartan las constantes multiplicativas, todo esto bajo la premisa de que a mayor tamaño de n menor es la contribución tanto de los términos de menor orden como de las contantes en la función.

Por ejemplo, si $T(n) = 6n^3 + 2n^2 + 20n + 45$ su orden de crecimiento es de n^3.

6. Análisis Asintótico

Al realizar el estudio de la complejidad en tiempo de un determinado algoritmo se trata de simplificar la función $T(n)$ para obtener solamente su orden de crecimiento, el cual nos brinda información aproximada sobre el comportamiento del algoritmo para tamaños grandes de n.

> El análisis asintótico se enfoca en el estudio del orden de crecimiento de un algoritmo para tamaños de entrada relativamente grandes.

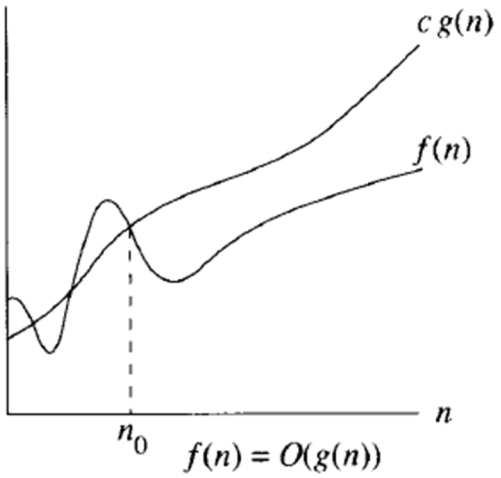
7. Notación Asintótica

Para formalizar la noción de complejidad en tiempo de un algoritmo se utiliza una notación matemática conveniente para comparar y analizar los tiempos de ejecución de los algoritmos sin la necesidad de hacer cálculos complicados. Esta notación se conoce como notación asintótica.

Existen varios tipos de notación asintótica entre los cuales podemos nombrar la notación *O* grande (O), la notación omega (Ω) y la notación theta (Θ).

7.1. Notación O Grande

La notación *O grande* nos permite encontrar una cota superior asintótica para el tiempo de ejecución de un algoritmo. Se utiliza en el estudio del peor caso.



Sean $f(n)$, $g(n)$: $\mathbb{N} \rightarrow \mathbb{R}^*$, se define formalmente como:

$$O(g(n)) = \{ f(n) \,/\, \exists c \in \mathbb{R}^+ \; \exists n_0 \in \mathbb{N} \; \forall n > n_0 : 0 \leq f(n) \leq c\, g(n) \}$$

De acuerdo a esta definición, $O(g(n))$ es el conjunto de todas aquellas funciones $f(n)$ que a partir de un tamaño de entrada considerable, n_0, se encuentran acotadas superiormente por una función $g(n)$ multiplicada por una constante c.

Cuando se afirma que el tiempo de ejecución de un algoritmo es de $O(g(n))$ esto quiere decir que no importa cual entrada particular de tamaño n se utilice, el tiempo para ese conjunto de entradas es como máximo $g(n)$.

Propiedades de la Notación O Grande

Sean $f(n)$, $f_2(n)$, $g(n)$, $g_2(n)$ y $h(n)$: $\mathbb{N} \rightarrow \mathbb{R}^*$ y c una constante, la notación *O grande* verifica las siguientes propiedades:

a) Reflexividad: $f(n) \in O(f(n))$.

b) Transitividad: $f(n) \in O(g(n)) \wedge g(n) \in O(h(n)) \Rightarrow f(n) \in O(h(n))$.

c) Regla de la eliminación de constantes: $\forall c \in R^+: O(c\,g(n)) = O(g(n))$.

d) Regla de la suma: $f(n) \in O(g(n)) \wedge f_2(n) \in O(g_2(n)) \Rightarrow f(n) + f_2(n) \in O(\ máx\{g(n), g_2(n)\}\ )$
Dado que $O(g(n)) + O(g_2(n)) = O(\ g(n) + g_2(n)\ ) = O(\ máx\{g(n), g_2(n)\}\ )$.

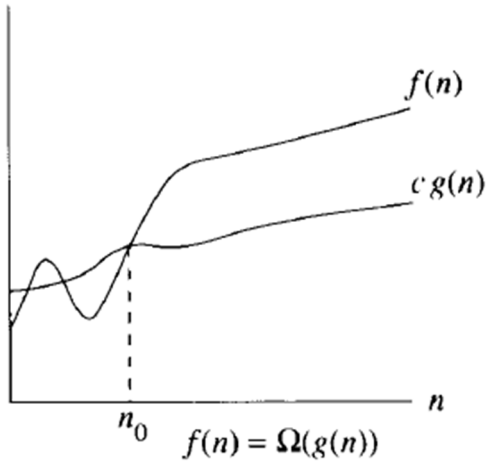
e) Regla del producto: $f(n) \in O(g(n)) \wedge f_2(n) \in O(g_2(n)) \Rightarrow f(n) \times f_2(n) \in O(\ g(n) \times g_2(n)\ )$
Dado que $O(g(n)) \times O(g_2(n)) = O(\ g(n) \times g_2(n)\ )$.

f) Regla de los logaritmos: $\forall a,b \in N: O(\log_a(n)) = O(\log_b(n))$.

g) Propiedad de linealidad: $\sum_{i=1}^{n} O(f(i)) = O(\sum_{i=1}^{n} f(i))$

7.2. Notación Ω

La notación Ω nos permite encontrar una cota inferior asintótica para el tiempo de ejecución de un algoritmo. Se utiliza en el estudio del mejor caso.



Sean $f(n)$, $g(n)$: $N \rightarrow R^*$, se define formalmente como:
$\Omega(g(n)) = \{ f(n) / \exists c \in R^+ \exists n_0 \in N\ \forall n > n_0: 0 \leq c\,g(n) \leq f(n) \}$

De acuerdo a esta definición, $\Omega(g(n))$ es el conjunto de todas aquellas funciones que a partir de un tamaño de entrada considerable, n_0, se encuentran acotadas inferiormente por una función $g(n)$ multiplicada por una constante c.

Cuando se afirma que el tiempo de ejecución de un algoritmo es de $\Omega(g(n))$ esto quiere decir que no importa cual entrada particular de tamaño n se utilice, el tiempo para ese conjunto de entradas es de por lo menos $g(n)$.

7.3. Notación Θ

La notación Θ nos permite encontrar una cota asintótica ajustada para el tiempo de ejecución de un algoritmo. Se utiliza para el análisis exacto de un algoritmo.

Sean $f(n)$, $g(n)$: $N \rightarrow R^*$, se define formalmente como:

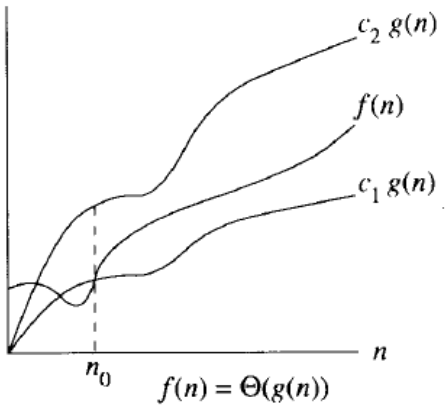
$$\Theta(g(n)) = \{ f(n) / \exists c_1 \in R^+ \exists c_2 \in R^+ \exists n_0 \in N\ \forall n > n_0 : 0 \leq c_1\,g(n) \leq f(n) \leq c_2\,g(n) \}$$

De acuerdo a esta definición, $\Theta(g(n))$ es el conjunto de todas aquellas funciones que a partir de un tamaño de entrada considerable, n_0, se encuentran acotadas inferiormente por la función $g(n)$

multiplicada por una constante c_1, y superiormente por la función $g(n)$ multiplicada por una constante c_2.

Teorema:
Para dos funciones cualesquiera $f(n)$ y $g(n)$:

$f(n) \in \Theta(g(n)) \Leftrightarrow f(n) \in O(g(n)) \cap \Omega(g(n))$



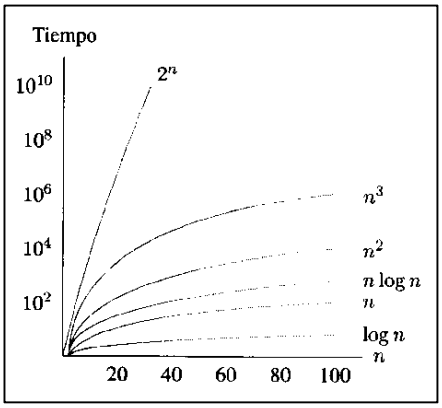
Por motivos de sencillez, cuando una función $f(n)$ pertenece al conjunto $O(g(n))$ en vez de colocar la notación de conjuntos $f(n) \in O(g(n))$ se puede colocar $f(n) = O(g(n))$, lo mismo se aplica para las notaciones Ω y Θ.

8. Jerarquía de Órdenes de Complejidad

El cálculo de la complejidad en tiempo de un algoritmo nos da una idea de la cantidad de tiempo aproximado que tardará el algoritmo para dar solución a un problema con un tamaño de entrada n. Al afirmar que la complejidad en tiempo de un algoritmo es de $O(g(n))$, se está afirmando que el tiempo aproximado para dar solución a una instancia del problema de tamaño n es $g(n)$.

A continuación se presenta una tabla con los órdenes de complejidad conocidos, organizados de menor a mayor.

Orden	Complejidad
$O(1)$	Complejidad constante
$O(\log(n))$	Complejidad logarítmica
$O(n)$	Complejidad lineal
$O(n \log (n))$	Complejidad semi-lineal o cuasi-lineal
$O(n^2)$	Complejidad cuadrática
$O(n^3)$	Complejidad cúbica
$O(n^k)$, $k > 3$	Complejidad polinomial
$O(2^n)$	Complejidad exponencial
$O(n!)$	Complejidad factorial



9. Reglas para el Cálculo de la Eficiencia de un Algoritmo

Cuando se analiza en la práctica la eficiencia de un algoritmo, casi nunca se realizan cálculos muy detallados, estos son simplificados mediante la notación asintótica y sus propiedades.

A continuación se enumeran algunas reglas, basadas en la teoría de la notación asintótica, para el cálculo de la eficiencia de un algoritmo:

9.1. Declaraciones

Las declaraciones de constantes, variables y tipos no se toman en cuenta al realizar el análisis en tiempo de un algoritmo. Igualmente sucede con los comentarios.

9.2. Operaciones elementales

Las operaciones elementales tendrán un coste constante, $O(1)$. Se consideran operaciones elementales (siempre que no dependan de una función o un procedimiento) las siguientes:

- Asignación.
- Entrada/Salida.
- Operaciones aritméticas.
- Operaciones relacionales.
- Operaciones lógicas.
- Operadores de acceso a una estructura: Corchetes para acceder a una posición de un arreglo, selector de campo de un registro, acceso a un apuntador, entre otros.

9.3. Secuenciación

Para calcular el orden al cual pertenece una secuenciación se aplica la regla de la suma, por ejemplo, si la secuencia está formada por las operaciones S_1, S_2, …, S_m y sus respectivos tiempos de ejecución y ordenes son $T_1(n) \in O(g_1(n))$, $T_2(n) \in O(g_2(n))$, …, $T_m(n) \in O(g_m(n))$ entonces:

$$T_1(n) + T_2(n) + \ldots + T_m(n) \in O(máx\{ g_1(n), g_2(n), \ldots, g_m(n) \})$$

9.4. Estructuras Condicionales

Sean S_i secuencias de operaciones, *cond* la condición de la estructura, $T_i(n)$ y $T_{cond}(n)$ sus tiempos de ejecución, y $O(g_i(n))$ y $O(g_{cond}(n))$ los órdenes a los cuales pertenecen respectivamente ($i \in$ N y $1 \leq i \leq m$).

Condicional Simple

```
si (cond) entonces
   S1
fsi
```

El orden de una expresión condicional simple se calcula como el tiempo de ejecución de la condición *cond*más el tiempo de ejecución de la secuencia S_1, tomando en cuenta que esta se ejecuta: $T_{cond}(n) + T_1(n) \in O(g_{cond}(n)) + O(g_1(n))$. Ahora bien, si aplicamos la regla de la suma, tenemos que:

$$\begin{aligned} O(g_{cond}(n)) + O(g_1(n)) &= O(g_{cond}(n) + g_1(n)) \\ &= O(máx\{g_{cond}(n), g_1(n)\}) \end{aligned}$$

Por lo tanto:

$$T_{cond}(n) + T_1(n) \in O(máx\{g_{cond}(n), g_1(n)\})$$

Condicional Doble

```
si(cond) entonces
   S1
sino
   S2
fsi
```

El orden de una expresión condicional doble se calcula como el tiempo de ejecución de la condición *cond* más el tiempo de ejecución máximo entre las secuencias S_1 y S_2, tomando en cuenta que siempre se va a ejecutar la secuencia de mayor tiempo: $T_{cond}(n) + máx\{T_1(n), T_2(n)\} \in O(g_{cond}(n)) + O(máx\{g_1(n), g_2(n)\})$. Simplificando la expresión, tenemos que:

$$\begin{aligned} O(g_{cond}(n)) + O(\max\{g_1(n), g_2(n)\}) &= O(g_{cond}(n) + máx\{g_1(n), g_2(n)\}) \\ &= O(máx\{g_{cond}(n), g_1(n), g_2(n)\}) \end{aligned}$$

Por lo tanto:

$$T_{cond}(n) + máx\{T_1(n), T_2(n)\} \in O(máx\{g_{cond}(n), g_1(n), g_2(n)\})$$

Selección

```
selección (cond)
   val1:  S1
   val2:  S2
   …
   valm-1: Sm-1
   sino:  Sm
fselección
```

El orden de una expresión de selección se calcula como el tiempo de ejecución de la condición *cond* más el tiempo de ejecución máximo entre las secuencias S_i ($1 \le i \le m$), tomando en cuenta que siempre se va a ejecutar la secuencia de mayor tiempo: $T_{cond}(n) + máx\{T_1(n), T_2(n), \ldots, T_m(n)\} \in O(g_{cond}(n)) + O(máx\{g_1(n), g_2(n), \ldots, g_m(n)\})$. Simplificando la expresión, tenemos que:

$$\begin{aligned} O(g_{cond}(n)) + O(máx\{g_1(n), g_2(n), \ldots, g_m(n)\}) &= O(g_{cond}(n) + máx\{g_1(n), g_2(n), \ldots, g_m(n)\}) \\ &= O(máx\{g_{cond}(n), g_1(n), g_2(n), \ldots, g_m(n)\}) \end{aligned}$$

Por lo tanto:

$$T_{cond}(n) + máx\{T_1(n), T_2(n), \ldots, T_m(n)\} \in O(máx\{g_{cond}(n), g_1(n), g_2(n), \ldots, g_m(n)\})$$

9.5. Estructuras Iterativas

El orden de cualquier estructura iterativa se calcula como el tiempo de ejecución de la condición de parada del ciclo (la cual generalmente el de $O(1)$) más la sumatoria, sobre el número de iteraciones del ciclo, del tiempo de ejecución de la secuencia S: $T_{cond}(n) + \sum_{\#iter} T(n)$. Si $T_{cond}(n) \in O(g_{cond}(n))$ y $T(n) \in O(g(n))$ entonces $T_{cond}(n) + \sum_{\#iter} T(n) \in O(g_{cond}(n)) + \sum_{\#iter} O(g(n))$, simplificando la expresión tenemos que:

$$\begin{aligned} O(g_{cond}(n)) + \sum_{\#iter} O(g(n)) &= O\left(g_{cond}(n)\right) + O\left(\sum_{\#iter} g(n)\right) \\ &= O\left(g_{cond}(n) + \sum_{\#iter} g(n)\right) \\ &= O\left(máx\{g_{cond}(n), \sum_{\#iter} g(n)\}\right) \end{aligned}$$

Por lo tanto:

$$T_{cond}(n) + \sum_{\#iter} T(n) \in O\left(máx\{g_{cond}(n), \sum_{\#iter} g(n)\}\right)$$

A continuación estudiaremos el caso particular del ciclo para.

Ciclo Para

```
para i ← inicio hasta fin hacer
   S
fpara
```

En el ciclo para en particular, se conoce el número de iteraciones a realizar, por lo tanto la sumatoria quedaría como sigue:

$$\sum_{i=inicio}^{fin} T(n) = \sum_{i=1}^{fin-inicio+1} T(n) = \sum_{i=0}^{fin-inicio} T(n)$$

De esta manera se pueden utilizar las sumas notables:

$\sum_{i=1}^{n} c = cn$	$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$	$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$
$\sum_{i=1}^{n} i^3 = \left(\frac{n(n+1)}{2}\right)^2$	$\sum_{i=1}^{n} i^k \approx \frac{n^{k+1}}{\lvert k+1 \rvert}, k \neq -1$	$\sum_{i=1}^{n} (2i+1) = n^2$
$\sum_{i=1}^{n} 2i = n(n+1)$	$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$	$\sum_{i=0}^{n} c^i = \frac{c^{n+1} - 1}{c - 1}$

10. Análisis de Complejidad en Tiempo de Algoritmos Iterativos

A continuación se muestran varios ejemplos que demuestran cómo se realiza el análisis asintótico de algoritmos iterativos.

10.1. Factorial de un Número

```
01  algoritmo factorial
02  var
03        entero: n, fact
04  inicio
05        escribir (“Factorial de un número”)
06        escribir (“Introduzca un número entero: ”)
07        leer (n)
08        fact ← 1
09        para i ← 2 hasta n en 1 hacer
10              fact ← fact * i
11        fpara
12        escribir (“El factorial de ”, n, “ es: ”, fact)
13  fin
```

Para calcular la complejidad en tiempo del algoritmo del factorial no se toman en cuenta las líneas 1 – 4 y 13 ya que son las líneas de inicio y fin del algoritmo más las líneas de las declaraciones. Se puede ver el algoritmo del factorial entonces como una secuenciación de las instrucciones 5 – 9 y 12. Nótese que el ciclo (línea 9) se considera una instrucción, y debe ser analizado aparte. De acuerdo a lo explicado anteriormente tenemos que el tiempo del factorial es:

$T_{fact}(n) = T_5(n) + T_6(n) + T_7(n) + T_8(n) + T_9(n) + T_{12}(n)$

- *Analizando las líneas 5 – 9 y 12:* Estas líneas representan operaciones elementales.

  $T_5(n) \in O(1)$
  $T_6(n) \in O(1)$
  $T_7(n) \in O(1)$
  $T_8(n) \in O(1)$
  $T_{12}(n) \in O(1)$

- *Analizando el ciclo de la línea 9:* Este ciclo tiene implícitamente la condición de parada $i \leq n$ y una sola instrucción interna en la línea 10, por lo tanto:

  $T_9(n) = T_{cond}(n) + \sum_{i=1}^{n} T_{10}(n)$

  $T_{cond}(n)$, $T_{10}(n) \in O(1)$ porque son operaciones elementales. Entonces tenemos:

$O(1) + \sum_{i=1}^{n} O(1)$	=	$O(1) + O(\sum_{i=1}^{n} 1)$	Propiedad de linealidad
	=	$O(1) + O(n)$	Suma notable
	=	$O(1 + n)$	Regla de la suma
	=	$O(máx\{1, n\})$	Regla de la suma
	=	$O(n)$	Regla de la suma

  Por lo tanto $T_9(n) \in O(n)$

Finalmente, dada la regla de la suma $T_{fact}(n) \in O(máx\{1, 1, 1, 1, n, 1\}) \Rightarrow T_{fact}(n) \in O(n)$
De acuerdo al resultado el tiempo del factorial está en $O(n)$, por lo tanto es de orden lineal.

10.2. Fibonacci

```
01  func fibonacci(entero: n): entero
02  var
03        entero: fn, ant, act, i
04  inicio
05        si (n = 0 ∨ n = 1) entonces
06              retornar (n)
07        sino
08              ant ← 0
09              act ← 1
10              para i ← 2 hasta n en 1 hacer
11                    fn ← ant + act
12                    ant ← act
13                    act ← fn
14              fpara
15              retornar(fn)
16        fsi
17  ffunc
```

Como se puede ver $T_{fib}(n) = T_5(n)$, ya que el cuerpo de la función está formado únicamente por un condicional doble.

- *Analizando $T_{fib}(n)$:*

  $T_{fib}(n) = T_{cond}(n) + máx\{T_6(n), T_8(n) + T_9(n) + T_{10}(n) + T_{15}(n)\}$
  - $T_{cond}(n), T_6(n), T_8(n), T_9(n), T_{15}(n) \in O(1)$ ya que son operaciones elementales.
  - *Analizando $T_{10}(n)$:*

    $T_{10}(n) = T_{cond2}(n) + \sum_{i=1}^{n-1}(T_{11}(n) + T_{12}(n) + T_{13}(n))$

    $T_{cond2}(n), T_{11}(n), T_{12}(n), T_{13}(n) \in O(1)$ ya que son operaciones elementales, por lo tanto la sumatoria queda como:

    $$\begin{aligned} O(1) + \sum_{i=1}^{n-1}(O(1) + O(1) + O(1)) &= O(1) + \sum_{i=1}^{n-1} O(1) && \text{Regla de la suma} \\ &= O(1) + O(\sum_{i=1}^{n-1} 1) && \text{Propiedad de linealidad} \\ &= O(1) + O(n-1) && \text{Suma notable} \\ &= O(1) + O(n) && \text{Regla de la suma} \\ &= O(máx\{1, n\}) && \text{Regla de la suma} \\ &= O(n) && \text{Regla de la suma} \end{aligned}$$

    $T_{10}(n) \in O(n)$.

  Finalmente:
  $T_{fib}(n) \in O(máx\{1, 1, 1, 1, n, 1\}) \Rightarrow T_{fib}(n) \in O(n)$

La función de Fibonacci de n está en $O(n)$.

11. Análisis de Complejidad en Tiempo de Algoritmos Recursivos

El análisis de complejidad en tiempo de un algoritmo recursivo se basa en el cálculo de recurrencias. Una recurrencia es una ecuación o inecuación que describe una función en términos de sí misma para entradas más pequeñas.

Los pasos generales a seguir para el cálculo de recurrencias son:

a) Identificar el caso a analizar (mejor caso, peor caso, caso promedio).
b) Plantear la recurrencia asociada al algoritmo.
c) Aplicar un método de solución de la recurrencia. Existen diferentes métodos para la solución de recurrencias. Entre ellos se pueden mencionar:
   - Método de expansión, también llamado método de iteración.
   - Método de acotación.
   - Método de sustitución.
   - Método del árbol recursivo.
   - Método maestro.
   - Método de la ecuación característica.
   - Método de la función generatriz.
d) Expresar el resultado obtenido en notación asintótica.

La selección del método para resolver la recurrencia depende en gran parte de la forma de la recurrencia obtenida. En esta oportunidad se estudiarán solo dos de estos métodos: El método de expansión y el método de acotación.

11.1. Método de Expansión

La idea básica de este método es expandir (iterar) la recurrencia y expresarla como una sumatoria de términos dependientes únicamente de n y las condiciones iniciales (Cormen, 2001), hasta percibir un patrón particular a partir del cual sea posible inferir la solución con un $T(n_0)$ conocido.

Este método habitualmente solo funciona para ecuaciones de recurrencia con una sola llamada recursiva.

Generalmente este método requiere mucha álgebra, sin embargo, la clave se encuentra en enfocarse en dos parámetros: (1) El número de veces que la recurrencia debe ser iterada hasta alcanzar la condición de parada, y (2) la suma de términos generada en cada nivel del proceso iterativo.

Ejemplo:

A continuación se realizará el análisis del peor caso de la función recursiva del factorial de n.

```
01 func factorial(entero: n): entero
02 inicio
03       si(n = 0 ∨ n = 1) entonces
04             retornar(1)
05       sino
06             retornar(n * factorial(n-1))
07       fsi
08 ffunc
```

- *Ecuación de Recurrencia Asociada:*

$$T(n) = \begin{cases} 1 & \text{, si } n = 0 \text{ o } n = 1 \\ T(n-1) + 1 & \text{, si } n > 1 \end{cases}$$

- *Método de Expansión:*

$$\begin{aligned} T(n) &= T(n-1) + 1 = (T(n-2) + 1) + 1 \\ &= T(n-2) + 2 = (T(n-3) + 1) + 2 \\ &= T(n-3) + 3 = (T(n-4) + 1) + 3 \\ &= T(n-4) + 4 = (T(n-5) + 1) + 4 \\ &\dots \\ &= T(n-k) + k \end{aligned}$$

  Reducción al caso base:

  En este caso la expansión termina cuando $n - k = 1$, en ese momento el valor de $k = n - 1$.

  Sustituyendo:

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

- *Notación Asintótica:* De acuerdo al resultado obtenido mediante el método de expansión se concluye que $T(n) \in O(n)$.

11.2. Método de Acotación

Este método se utiliza para resolver ecuaciones recurrentes con varias llamadas recursivas. Consiste en buscar una cota para la ecuación recurrente, de forma tal en que se pueda convertir en una ecuación con una única llamada recursiva, la cual se puede resolver por el método de expansión.

Ejemplo:

A continuación se realizará el análisis de complejidad en tiempo de la función de Fibonacci.

```
01 func fibonacci(entero: n): entero
02 inicio
03       si(n = 0 ∨ n = 1) entonces
04             retornar(n)
05       sino
06             retornar(fibonacci(n-1) + fibonacci(n-2))
07       fsi
08 ffunc
```

- *Ecuación de Recurrencia Asociada:*

$$T(n) = \begin{cases} 1 & \text{, si } n = 0 \text{ o } n = 1 \\ T(n-1) + T(n-2) + 1 & \text{, si } n > 1 \end{cases}$$

- *Método de Acotación:*

$T(n-2) + T(n-2) + 1 \quad \leq \quad T(n-1) + T(n-2) + 1 \quad \leq \quad T(n-1) + T(n-1) + 1$

Por lo tanto obtenemos dos nuevas ecuaciones de recurrencia que acotan de forma inferior y superior a la ecuación de recurrencia original:

$$Sup(n) = \begin{cases} 1 & \text{, si } n = 0 \text{ o } n = 1 \\ 2Sup(n-1) + 1 & \text{, si } n > 1 \end{cases}$$

$$Inf(n) = \begin{cases} 1 & \text{, si } n = 0 \text{ o } n = 1 \\ 2Inf(n-2) + 1 & \text{, si } n > 1 \end{cases}$$

Luego, ambas ecuaciones deben resolverse por el método de expansión. El resultado final nos dirá en que intervalo se encuentra el orden del algoritmo.

$$\begin{aligned} Sup(n) &= 2Sup(n-1) + 1 = 2(2Sup(n-2) + 1) + 1 \\ &= 4Sup(n-2) + 3 = 4(2\text{Sup}(n-3) + 1) + 3 \\ &= 8Sup(n-3) + 7 = 8(2Sup(n-4) + 1) + 7 \\ &\ldots \\ &= 2^k Sup(n-k) + (2^k - 1) \end{aligned}$$

Reducción al caso base:

En este caso la expansión termina cuando $n - k = 1$, en ese momento el valor de $k = n - 1$.

Sustituyendo:

$$\begin{aligned} &= 2^{n-1} Sup(1) + (2^{n-1} - 1) \\ &= 2^{n-1} + 2^{n-1} - 1 \\ &= 2(2^{n-1}) - 1 \\ &= 2^n - 1 \end{aligned}$$

Por lo tanto $Sup(n) \in O(2^n)$.

$$
\begin{aligned}
Inf(n) &= 2Inf(n-2)+1 = 2(2Inf(n-4)+1)+1 \\
&= 4Inf(n-4)+3 = 4(2Inf(n-6)+1)+3 \\
&= 8Inf(n-6)+7 = 8(2Inf(n-8)+1)+7 \\
&\ldots \\
&= 2^k Inf(n-2k)+(2^k-1)
\end{aligned}
$$

Reducción al caso base:

En este caso la expansión termina cuando $n - 2k = 1$, en ese momento el valor de $k = (n-1)/2$.

Sustituyendo:

$$
\begin{aligned}
&= 2^{(n-1)/2} Inf(1) + (2^{(n-1)/2} - 1) \\
&= 2^{(n-1)/2} + 2^{(n-1)/2} - 1 \\
&= 2(2^{(n-1)/2}) - 1 \\
&= 2^{(n+1)/2} - 1 \\
&= 2^{n/2}\sqrt{2} - 1
\end{aligned}
$$

Por lo tanto $Inf(n) \in O(2^n)$.

- *Notación Asintótica:* De acuerdo al resultado obtenido mediante el método de expansión se concluye que $T(n) \in O(2^n)$, es decir, es de orden exponencial.

12. Complejidad en Espacio

La complejidad en espacio o complejidad espacial de un algoritmo se refiere a la cantidad de memoria utilizada por el mismo (Ravikumar, 1996). Formalmente:

> La complejidad espacial de un algoritmo se puede definir como una función E: N $\rightarrow$ N, donde $E(n)$ representa el número máximo de unidades de memoria que el algoritmo utiliza para almacenar una entrada del tamaño n (Sipser, 2006).

El decir que la complejidad espacial de un algoritmo es $E(n)$ es lo mismo que decir que el algoritmo se ejecuta en un espacio $E(n)$.

13. Análisis Espacial

> El análisis espacial es el estudio del orden de crecimiento de la función de complejidad en espacio de un algoritmo.

El análisis espacial tuvo su auge cuando se comenzaron a usar computadoras, ya que el espacio (tanto interno como externo) era limitado. Actualmente se puede ver que el análisis espacial se he dejado de aplicar en la mayoría de los casos, por esta razón se pueden ver en el mercado actual aplicaciones que requieren gran cantidad de megabytes e incluso gigabytes de memoria para funcionar, sin embargo, con la introducción de nuevos dispositivos tales como Smartphones, PDAs, tablet PCs, entre otros, los cuales cuentan con una cantidad limitada de memoria, se he venido recuperando esta tendencia (McConnell, 2008).

Al igual que sucede al realizar el análisis temporal de un algoritmo, se toma el término de mayor orden de $E(n)$ y se descartan sus constantes para conseguir una función $g(n)$ más sencilla que sea una cota asintótica de $E(n)$, obteniendo así que el algoritmo corre en un espacio de $O(g(n))$.

Este tipo de análisis debe ser, al igual que el análisis temporal, independiente del lenguaje de programación y de la máquina en la cual se vaya a codificar el algoritmo, por lo tanto, no se utilizarán unidades concretas de medida, por ejemplo Bytes, en cambio, se asumirá que cada tipo de dato primitivo necesita cierta cantidad de unidades de espacio (palabras) para ser almacenado:

Tipos Primitivos

lógico	1 palabra
caracter	1 palabra
entero	2 palabras
real	4 palabras

Tipos Definidos por el Usuario

enumerado	2 palabras
intervalo	Depende del tipo primitivo sobre el cual se defina

Tipos de Datos Estructurados

cadena	$long(c)$ palabras
arreglo	$(max_1 - min_1 + 1) \times (max_2 - min_2 + 1) \times \ldots \times (max_n - min_n + 1) \times (Costo(tipo))$ + 3 palabras
registro	$\sum_{i=1}^{n} Costo(tipo_i)$ palabras

Se debe tomar en cuenta en el caso de las funciones y procedimientos que los parámetros formales ocupan espacio en memoria también. En el caso de las funciones recursivas se debe tomar en cuenta la cantidad de llamadas hechas para calcular cuantas veces se almacena cada variable en la pila de memoria, y este valor se multiplica por la cantidad de memoria que usa el algoritmo. Se obviará el espacio utilizado por las direcciones de retorno a la llamada anterior.

Ejemplos:

- Suma de 3 números:

```
01 func suma(entero: a, b, c): entero
02 inicio
03       retornar(a+b+c)
04 ffunc
```

Al calcular la complejidad en espacio de la función suma, tenemos:

$E(n) = 2 + 2 + 2 = 6$ (Las variables a, b y c ocupan dos unidades cada una)
$E(n) \in O(1)$ (Complejidad constante)

- Suma de un vector de números:

```
01 const
02       N ← ?
03 tipo
04       vector = arreglo [1..N] de real
05 func suma_vector(vector: v; entero: cant): real
06 var
07       entero: i
08       real: sum
09 inicio
10       sum ← 0.0
11       para i ← 1 hasta cant hacer
12             sum ← sum + v[i]
13       fpara
14       retornar(sum)
15 ffunc
```

Para calcular la complejidad en espacio de la función suma_vector, tenemos:

$E(n) = 4n + 11$ ($4n$+3 espacio por el vector y su descriptor y 8 por las variables cant, sum e i)
$E(n) \in O(n)$ (Complejidad lineal)

En este caso el espacio requerido por el vector depende del tamaño del problema, simplemente se generalizará n = N.

- Suma de un vector de números (versión recursiva)

Al calcular la complejidad en espacio de suma_vector_rec, tenemos:

$E(n) = n(4n + 5)$ ($4n$+3 espacio por el vector y dos unidades por la variable cant, por n llamadas)
$E(n) \in O(n^2)$ (Complejidad cuadrática)

```
01 const
02       N ← ?
03 tipo
04       vector = arreglo [1..N] de real
05 func suma_vector_rec(vector: v; entero: cant): real
06 inicio
07       si(n = 1) entonces
08             retornar(v[1])
09       sino
10             retornar(v[cant] + suma_vector_rec(v, cant-1))
11       fsi
12 ffunc
```

Si el vector fuera pasado por referencia (que es lo que sucede por defecto en el lenguaje de programación C), la complejidad en espacio de la función se convierte en lineal, ya que en cada llamada se pasaría solamente el descriptor del vector:

$E(n) = 6n + 3$ (Dos unidades por la variable cant, por n llamadas, más $4n+3$ del vector que se guarda una sola vez en memoria)

$E(n) \in O(n)$ (Complejidad lineal)

14. Equilibrio Tiempo/Memoria

```
01 algoritmo intercambio
02 inicio
03       var
04             entero: x, y, aux
05       x ← 5
06       y ← 7
07       aux ← x
08       x ← y
09       y ← aux
10       escribir(x, " ", y)
11 fin
```

```
algoritmo intercambio
inicio
      var
            entero: x, y
      x ← 5
      y ← 7
      x ← x + y
      y ← x - y
      x ← x - y
      escribir(x, " ", y)
fin
```

Los algoritmos anteriores realizan ambos el intercambio de dos números enteros *x* y *y*. El primero de ellos utiliza una variable auxiliar *aux* para poder realizar el intercambio, por lo cual ocupa más memoria que el segundo que no la utiliza, sin embargo, las operaciones utilizadas en el primer algoritmo son solo asignaciones, mientras que en el segundo, además de las asignaciones, se utilizan 3 operaciones aritméticas por lo cual tardaría más tiempo en ejecutarse.

Dependiendo del tipo de algoritmo, hay que buscar un equilibrio entre el tiempo y la memoria que debe utilizar ya que al requerir mejores tiempos se necesitará más memoria y viceversa (Martínez, 2003).

15. Referencias

Aho, A., Hopcroft, J. y Ullman, J. (1983). Data Structures and Algorithms. Massachusetts, USA: Addison-Wesley.

Alsuwaiyel, M. H. (2003). Algorithms. Design Techniques and Analysis. New Jersey, USA: World Scientific Publishing.

Brassard, G. y Batley, P. (1996). Fundamentals of Algorithmics. New Jersey, USA: Prentice-Hall.

Cormen, T., Leiserson, C., Rivest, R. y Stein Clifford. (2001). Introduction to Algorithms (Segunda Edición). Massachusetts, USA: McGraw-Hill.

Franch, X. (1999). Estructuras de datos. Especificación, Diseño e Implementación (Tercera Edición). Barcelona, España: Editions UPC.

Martínez, A., Rosquete, D. (2009). NASPI: Una Notación Estándar para Programación Imperativa. Télématique.

Martínez, F. y Martín, G. (2003). Introducción a la programación Estructurada en C. Valencia, España: Els Authors.

McConnell, J. J. (2008). Analysis of Algorithms. An Active Learning Aproach (Segunda Edición). Ontario, Canada: Jones and Bartlett Publishers. Inc.

Puntambekar, A. A. (2008). Analysis and Design of Algorithms (Primera Edición). Pune, India: Technical Publications Pune.

Ravikumar, C. P. (1996). Parallel Methods for VLSI Design. New, Jersey, USA: Ablex Publishing Corporation.

Sipser, M. (2006). Introduction to the Theory of Computation (Segunda Edición). Massachusetts, USA: Thomson Course Technology.