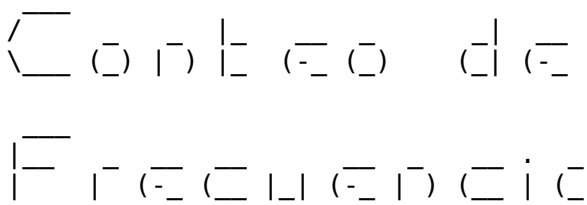
"Análisis de Algoritmos"
José Helo Guzmán



Contenido

▬▬▬▬ 1. Introducción.

Presentamos en esta sección algunos ejemplos de algoritmos con su respectivo conteo de frecuencias. Esta será la base para el análisis de algoritmos que haremos posteriormente.

▬▬▬▬ 2. Cómo Analizar un Algoritmo.

Veamos un ejemplo de un algoritmo:

```
sumar(n):=block
(  [a,b,x],
   a: n + 1,
   b: n + 2,
   x: 2*a + 3*b,
   return(x)
);
```

Este algoritmo o programa se parece mucho a algunos de los programas que ustedes conocen.

El algoritmo puede ser analizado desde muchas perspectivas. Por ejemplo:

• Tiempo de ejecución.

• Cuánto espacio necesita.

• Uso de recursos de red.

• Cantidad de poder de consumo.

Dependiendo de los requerimientos y del proyecto particular, se escoge el criterio para analizarlo. En este texto, vamos a estar principalmente interesados en el tiempo de ejecución y ocasionalmente mencionaremos algunos aspectos de la memoria.

Veamos el análisis del tiempo:

```
sumar(n):=block  --------------> T(n)=6
(  [a,b,x],
   a: n + 1,     --> 1 unidad
   b: n + 2,     --> 1 unidad
   x: 2*a + 3*b, --> 3 unidad
   return(x)     --> 1 unidad
);
```

Entonces T(n) = 6.

Observemos que cada instrucción toma 1 unidad de tiempo de ejecución. No vamos a analizar a un nivel inferior, como por ejemplo cuántas instrucciones en ensamblador toma cada instrucción del lenguaje original. Esto no es necesario para establecer el crecimiento en el tiempo que toma. Este algoritmo tarda siempre la misma cantidad de tiempo ejecutándose, es decir tiempo constante.

Se suele indicar:

T(n) = 6 = O(6) = O(6*1) = O(1)

Para el análisis de espacio tenemos las siguientes variables: n,a, b, x.

Por lo tanto el análisis de espacio es S(n) = 4.

▬▬▬ 3. Un Primer Ciclo.

Supongamos que tenemos el siguiente ciclo:

```
foo(n):=block
(
   for i:1 thru n do
   (
       print(i)
   )
);
```

Que es equivalente a:

```
foo(n):=block
(
   for i:1 thru n step 1 do
   (
       print(i)
   )
);
```

```
foo(n):=block
(
   for i:1 thru n next i+1 do
   (
       print(i)
   )
);
```

Veamos como funciona:

```
(%i) foo(5);
1
2
3
4
5
done
```

```
(%i) foo(10);
1
2
3
4
5
6
7
8
9
10
done
```

Este tipo de “loop” se puede expresar como:

$$\prod_{}^{n}$$

$$\sum_{i=1}^{n} 1 = n$$

Podemos ver que:

```
(%i) sum(1,i,1,n);
(%o) n
```

●● Y lo podemos analizar como:

```
foo(n):=block ---> T(n) = n
(
   for i:1 thru n do ---> n*1
   (
       print(i) ---> 1
   )
);
```

Este toma un tiempo

T(n) = n es O(n)

●● Para el siguiente ciclo:

```
foo(n):=block
(
   for i:n thru 1 step -1 do
   (
       print(i)
   )
);
```

Este ciclo es equivalente al anterior, excepto que hace el recorrido de forma inversa. De esta manera también toma un tiempo

T(n) = n es O(n)

▬▬▬ 4. Un Ciclo con Steps.

Para el siguiente ciclo:

```
foo(n):=block
(
   for i:1 thru n step 2 do
   (
```

```
       print(i)
   )
);
```

Tenemos que:

```
(%i) foo(6);
1
3
5
done
```

```
(%i) foo(12);
1
3
5
7
9
11
done
```

Al expresarlo como una sumatoria se tiene:

$$\sum_{i=1}^{n/2} 1 = n$$

```
(%i) sum(1,i,1,n/2);
(%o) n/2
```

Podemos analizar el ciclo como:

```
foo(n):=block ---> T(n) = n/2
(
   for i:1 thru n step 2 do --> (n/2)*1
   (
       print(i) ---> 1
   )
);
```

Este toma un tiempo

$T(n) = n/2$

$O(n/2) = O(\frac{1}{2}*n) = O(n)$

▬▬▬ 5. Ciclos Mezclados.

Para ciclos mezclados

```
foo(n):=block
(
   for i:1 thru n do
       for j:1 thru n do
           print(i,j)
);
```

Veamos como funciona:

```
(%i) foo(3);
1  1
1  2
1  3
2  1
2  2
2  3
3  1
3  2
3  3
(%o) done
```

```
(%i) foo(4);
1  1
1  2
1  3
1  4
2  1
2  2
2  3
2  4
3  1
3  2
3  3
3  4
4  1
4  2
4  3
4  4
(%o) done
```

Se puede representar como:

$$\sum_{i=1}^{n} \sum_{j=1}^{n} 1 = n^2$$

$$\sum^{n} n = n^2$$

$$\sum_{i=1}^{n}$$

Podemos ver que:

```
(%i) sum(
          sum(1,j,1,n),
          i,
          1,
          n);
(%o) n^2
```

Podemos analizar como:

```
foo(n):=block ---> T(n) = n^2
(
   for i:1 thru n do ---------------->n*n*1
       for j:1 thru n do  ---> n*1
           print(i,j) ---> 1
);
```

Entonces:

$T(n) = n^2 = O(n^2)$

▬▬▬ 6. Otros Ciclos Mezclados.

Tenemos el siguiente Loop.

```
foo(n):=block
(
   for i:1 thru n do
   (
       for j:1 thru n while (j <= i) do
       (
           print(i,j)
       ),
       print("-----")
   )
);
```

Al ejecutarlo se obtiene:

```
(%i) foo(4)
```

i	j
1	1

```
2  1
2  2
```

```
3  1
3  2
3  3
```

```
4  1
4  2
4  3
4  4
```

```
done
```

Expresado matemáticamente tenemos:

$$\sum_{i=1}^{n} \sum_{j=1}^{i} 1 = n^2$$

En maxima tenemos:

```
(%i) sum(1,j,1,i);
(%o) i
```

```
(%i) sum( sum(1,j,1,i),
          i,
          1,
          n);
(%o) (n^2 + n)/2
(%o) (n*(n+1))/2
```

Y así sucesivamente. Por lo tanto el tiempo de ejecución de:

$$1 + 2 + 3 + ... + n = \left(\frac{n*(n+1)}{2} \right)$$

O equivalentemente:

$T(n) = (1/2)*n*(n+1)$

$T(n) = (1/2)*(n^2 + n)$

Que es:

$O(n^2)$

▬▬▬ 7. Más Loops!!.

Tenemos el siguiente algoritmo.

Aquí lo escribimos como un “for”

```
foo(n):=block
(
   for i:1 thru n next i*2 do
   (
       print(i)
   )
);
```

Aquí tenemos el mismo algoritmo con un “while”.

```
foo(n):=block
(
   i:1,
   while i<=n do
   (
      print(i),
      i: i*2
   )
);
```

n = 10

i	
1	2^0
2	2^1
4	2^2
8	2^3
16	--> No se ejecuta.

n = 20

i	
1	2^0
2	2^1
4	2^2
8	2^3
16	2^4
32	--> No se ejecuta.

Cuántas veces se ejecuta este while?

Para probar esto tenemos:

$2^k \leq n$

$k \leq \log_2(n)$

Y este algoritmo toma tiempo $O(\log_2(n))$

Veamos la siguiente tabla.

Observación: para encontrar la mejor solución al logaritmo base 2 se utiliza la siguiente función:

```
log2(x):=float(radcan(log(x)/log(2)));
```

Podemos verificar este resultado de la siguiente manera:

n	while (i<n)	$\log_2(n)$
10	4	3.3219 ~ 4
20	5	4.3219 ~ 5
50	6	5.6439 ~ 6
64	7	6.0000 ~ 7
100	7	6.6439 ~ 7
500	9	8.9658 ~ 9

▬▬▬ 8. Siguen los Loops!!.

Veamos este otro ejemplo.

Tenemos este “for”:

```
foo(n):=block
(
   p:0,
   for i:1 step 1 while p<=n do
   (
      print(i,p),
      p: p+i
   )
);
```

Que podríamos escribirlo así:

```
foo(n):=block
(
   p:0,
   i:1,
   while p<=n do
   (
      print(i,p),
      p: p+i,
```

```
      i: i+1
   )
);
```

Analicemos el comportamiento del loop con un corrida.

n = 10

i	p	p:p+i
1	0	1
2	1	3
3	3	6
4	6	10
5	10	15
6	15	--------> No se ejecuta

n = 20

i	p	p:p+i
1	0	1
2	1	3
3	3	6
4	6	10
5	10	15
6	15	21
7	21	--------> No se ejecuta

En general podemos ver como se desarrolla “p”:

i	p
1	0+1 = 1
2	0+1+2 = 3
3	0+1+2+3 = 6
4	0+1+2+3+4 = 10
5	0+1+2+3+4+5 = 15
⋮	
k	0+1+2+3+4+...+k <= n

Observemos que:

$$1 + 2 + 3 + ... + k \;<=\; n$$

$$\left(\frac{k*(k+1)}{2} \right) \;<=\; n$$

$$\left(\frac{k^2 + k}{2} \right) <= n$$

$$k^2 + k <= 2*n$$

$$k^2 + k - 2*n = 0$$

●● Al resolver esta ecuación de segundo grado se obtiene:

$$k1 = + \left(\frac{-1 + \sqrt{(1 + 8*n)}}{2} \right)$$

$$k2 = - \left(\frac{-1 + \sqrt{(1 + 8*n)}}{2} \right)$$

Podemos tomar cualquier raíz para el análisis. Tomemos la raíz positiva. Tenemos entonces que este valor es una solución para “k”, de manera que se cumpla el algoritmo anterior.

$$\left(\frac{-1 + \sqrt{(1 + 8*n)}}{2} \right)$$

Y podemos buscar una cota superior:

$$\left(\frac{-1 + \sqrt{(1 + 8*n)}}{2} \right)$$

$$<= -1 + \sqrt{(1 + 8*n)}$$

$$<= -1 + \sqrt{(8*n + 8*n)}$$

$$<= -1 + \sqrt{(16*n)}$$

$$<= -1 + \sqrt{(2^4*n)}$$

$$<= -1 + 2^2*\sqrt{(n)}$$

<= -1 + 4*√(n)

es O(√(n))

●● Con un poquito de experiencia se suele decir:

$$\left(\frac{k^2 + k}{2} \right) <= n$$

Se tiene que:

$k^2 <= n$

$k <= \sqrt{n}$

y por lo tanto es O(√n).

●● Verifiquemos este resultado:

n	Iter(n)	1.4*sqrt(n)
10	5	4.4272 ~ 5
50	10	9.8995 ~ 10
100	14	14.000 ~ 14
1000	45	44.272 ~ 45

▬▬▬▬ 9. Y Todavía siguen Más !!.

Tenemos los siguiente algoritmos.

Estos loops se ejecutan uno después del otro.

```
foo(n):=block
(
   /* T1 */
   for i:1 thru n do
   (
      print(i)
   ),

   /* T2 */
   for j:1 thru n do
   (
      print(j)
```

```
   )
);
```

El primero T1(n) = n.

El segundo es T2(n) = n.

T(n) = T1(n) + T2(n)

T(n) = n + n

T(n) = 2*n

Entonces el tiempo total es: T(n) = 2*n = O(n)

▬▬▬ 10. Comentario sobre While.

Cada uno de estos ciclos se puede implementar como un ciclo con “while”, el tiempo de ejecución sería el mismo. Veamos un ejemplo.

Dado el siguiente ciclo con un for:

```
cicloFor(n):=block
(
   for i:1 thru n step 1 do
       print(i)
);
```

Podemos ver que su funcionamiento y su eficiencia, es equivalente a este ciclo escrito con while.

```
cicloWhile(n):=block
(
   i:1,
   while i <= 10 do
   (
      print(i),
      i: i+1
   )
);
```

Podemos decir lo mismo de cualquier otra instrucción de iteración que se utilice en un programa.

▬▬▬ 11. Comentario sobre el If.

Supongamos que tenemos la siguiente operación.

```
foo(n):=block
(
   if (x = 0) then
      T(n) = 4 = O(1)
   elseif (x > 0) then
      T(n) = 5*n + 2 = O(n)
   else
      T(n) = 2*n² + 1 = O(n²)
);
```

Para el análisis del pero caso debemos suponer que el tiempo de ejecución de esta instrucción de “if” está dado por $O(n^2)$

▬▬▬ 12. Resumen de los Tiempos de ejecución.

A continuación presentamos los tiempos de ejecución para diferentes loops, utilizamos una notación similar a un lenguaje de programación muy conocido.

Tabla Tiempos.	for(inicio, while, next)
O(n)	for(i:1, i<=n, i:i+1)
O(n)	for(i:1, i<=n, i:i+2)
O(n)	for(i:n, i>=1, i:i-1)
$O(\log_2(n))$	for(i:1, i<=n, i:i*2)
$O(\log_2(n))$	for(i:n, i>=1, i:i/2)
$O(\log_3(n))$	for(i:1, i<=n, i:i*3)
$O(\log_3(n))$	for(i:n, i>=1, i:i/3)

▬▬▬ 13. Algoritmo para Sumar un Vector.

Se presenta un vector con los subíndices para accesarlo. El vector inicia en la posición 1. Supongamos que tenemos el siguiente vector.

```
vec: [ 8, 3, 9, 7, 2]
       1  2  3  4  5
```

Y tenemos un algoritmo que suma los elementos de un vector:

```
sumar(vec,n):=block
(
   s:0,

   for i:1 thru n do
   (
      s: s + vec[i],
      print(s)
   ),
   return(s)
);
```

Al analizar cuánto tiempo tarda tenemos que:

```
sumar(vec,n):=block  ---> T(n) = 2*n + 2
(
   s:0,              --------> 1

   for i:1 thru n do --------> 2*n
   (
      s: s + vec[i], ---> 1
      print(s)       ---> 1
   ),
   return(s)         --------> 1
);
```

Por lo tanto T(n) = 2*n + 2

Si T(n) = 2*n+2 = O(n) que es la función mayor.

▬▬▬▬ 14. Algoritmo para Sumar Matrices.

Se tienen las siguientes matrices cuadradas:

```
A: [ [1, 2, 3],
     [4, 5, 6],
     [7, 8, 9]  ];
```

```
B: [ [11,12,13],
     [14,15,16],
     [17,18,19] ];
```

Al sumar las matrices se obtiene:

```
A + B : [ [12,  14,  16],
          [18,  20,  22],
          [24,  26,  28] ];
```

Veamos un algoritmo para la suma de matrices cuadradas, punto a punto:

```
sumarMat(A,B,n):=block
(
   for i:1 thru n do
   (
       for j:1 thru n do
       (
           A[i][j]: A[i][j] + B[i][j]
       )
   ),
   return(A)
);
```

Al analizar el algoritmo se obtiene:

```
sumarMat(A,B,n):=block
(
   for i:1 thru n do  ------------------------> n^2
   (
       for j:1 thru n do  ---------------> n
       (
          A[i][j]: A[i][j] + B[i][j] -> 1
       )
   ),
   return(A)   ------------------------------> 1
);
```

$T(n) = 2*n^2 + 2 = O(n^2)$

▬▬▬ 15. Resumen.

• Para un ciclo con “for” se puede establecer su tiempo de ejecución.

• Cualquier otro ciclo, ya sea con la forma de “while”, “repeat”, “do” se pueden estudiar de la misma forma que el “for”.

• Estos ciclos tienen formas características, con las que se puede saber su tiempo de ejecución.

▬▬▬ 16. Ejercicios.

●● Ejercicio 1.

Pruebe cada uno de los ejemplos en un lenguaje de programación.
Implemente la impresión para contar el número de veces que ocurren los ciclos.

●● Ejercicio 2.

Se tiene el siguiente algoritmo:

```
p:0;

i:1;
while i <= n do
(
   print(i,p),
   p:p+1,
   i:i*2
);

j:1,
while j <= p do
(
   print(j)
);
```

Indique si es correcto suponer que el tiempo de ejecución es:

O(log(log n))

●● Ejercicio 3.

Se tiene el siguiente algoritmo:

```
i:0;
while i <= n do
(
   j:1,
   while j <= n do
   (
      print(i,j)
      j:j*2
   ),
   i:i+1
);
```

Indique si es correcto suponer que el tiempo de ejecución es:

O(n * log n)

●● Ejercicio 4.

Tenemos el siguiente código.

```
while m <> n do
(
   if m>n then
      m: m-n
   else
```

```
      n: n-m
);
```

Para diferentes valores de “m” y “n”, encuentre el tiempo de ejecución del algoritmo.

Sugerencia: Pruebe con
m=16, n=2.
m=30, n=3.
m=50, n=5.