Capítulo 4.

Complejidad temporal de algoritmos.

4.1. Tiempo de ejecución y tamaño de la entrada.

Se desea tener una medida de la duración del tiempo de ejecución de un algoritmo en función del tamaño de la entrada.

A través de llamados al sistema operativo se puede conocer el valor del reloj de tiempo real. Invocando al reloj, antes y después de realizar el algoritmo se tendrá una medida de la duración del tiempo de ejecución. Sin embargo esta medida es muy dependiente del hardware (memoria, reloj, procesador), del sistema operativo (multitarea, multiusuario) y puede variar significativamente dependiendo del computador, del compilador, y de la carga del sistema. Al disponer de sistemas con multiprocesadores o que la ejecución sea distribuida también afecta medir el tiempo con cronómetro.

Por la razón anterior como una medida del tiempo de ejecución, se considera **contar** las instrucciones del lenguaje de alto nivel que son necesarias realizar.

El tamaño de la entrada debe ser precisado con más detalle. Podría ser el número de bits que miden la información que el algoritmo procesa, pero en forma tradicional se considera el número de elementos o componentes básicas que son sometidos al proceso.

Por ejemplo si tenemos un arreglo de n componentes, y el algoritmo tiene por objetivo, sumar los valores de las componentes, o bien ordenar las componentes, se suele decir que n es el tamaño de la entrada. Independientemente si el arreglo es de enteros, o de estructuras.

4.2. Complejidad temporal. Definición.

Se denomina complejidad temporal a la función $T(n)$ que mide el número de instrucciones realizadas por el algoritmo para procesar los n elementos de entrada.
Cada instrucción tiene asociado un costo temporal.

Afecta al tiempo de ejecución el orden en que se procesen los elementos de entrada.
Podría considerarse que los valores de los n casos que se presentan como entrada son los correspondientes: a un caso típico, o a un caso promedio, o de peor caso. El peor caso es el más sencillo de definir (el que demore más para cualquier entrada), pero si se desea otros tipos de entrada habría que definir qué se considera típico, o la distribución de los valores en el caso promedio.

4.3. Tipos de funciones.

Las funciones de n pueden ser de diferente tipo:
Funciones constantes: $f(n) = 5$, o bien $g(n) = 10$.
Funciones logarítmicas: $f(n) = \log(n)$, o bien $g(n) = n\log(n)$
Funciones polinomiales: $f(n) = 2n^2$, o bien $g(n) = 8n^2 + 5n$
Funciones exponenciales: $f(n) = 2^n$, o bien $g(n) = 2^{5n}$.
O mezclas de las anteriores, o cualquier función de n en un caso general.

En general, a medida que aumenta n, las exponenciales son mayores que las polinomiales; a su vez éstas son mayores que las logarítmicas, que son mayores que las constantes.

4.4. Acotamiento de funciones.

Veremos algunas definiciones que permiten clasificar las funciones por su orden de magnitud. Interesa encontrar una cota superior de la complejidad temporal. Consideremos la siguiente definición preliminar de la función O mayúscula (big oh), con la intención de clasificar funciones polinomiales.

Se dice que $T(n)$ es $O(n^i)$, si existen c y k tales que: $T(k) <= c\,k^i$

Sea $T(n) = (n+1)^2$.
Entonces debemos encontrar c y k tales que: $(k+1)^2 <= ck^i$.

Si suponemos que i toma valor 3, la desigualdad anterior se cumple para $c= 4$ y $k>=1$
Si suponemos que i toma valor 3, la desigualdad anterior se cumple para $c= 2$ y $k>=1{,}4376$

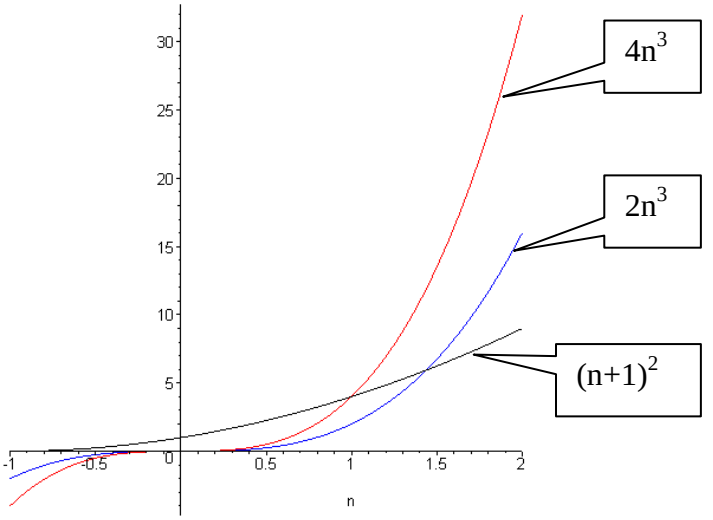

Figura 4.1. $T(n)$ es $O(n^3)$.

Se advierte que $T(n)$ queda acotada por arriba por $2n^3$ para $n>1.5$. Entonces $T(n)$ es $O(n^3)$.

Seguramente también es fácil encontrar soluciones si *i* es mayor que 3.

Interesa encontrar el *i* menor posible, que cumpla la definición.

Si suponemos que *i* toma valor 2, la desigualdad anterior se cumple para *c*= 4 y *k*>=1
Lo cual prueba que $T(n)$ es $O(n^2)$.
Relaciones que podemos observar en el siguiente diagrama.

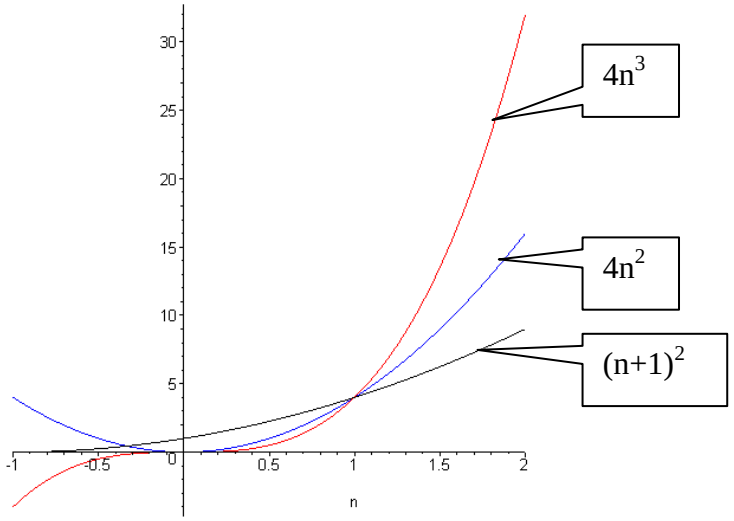

Figura 4.2. $T(n)$ también es $O(n^2)$.

Si para *i*=2, intentamos encontrar un *c* menor, por ejemplo 2, se tendrá que $T(n)$ queda acotada para *n*>2.41412:

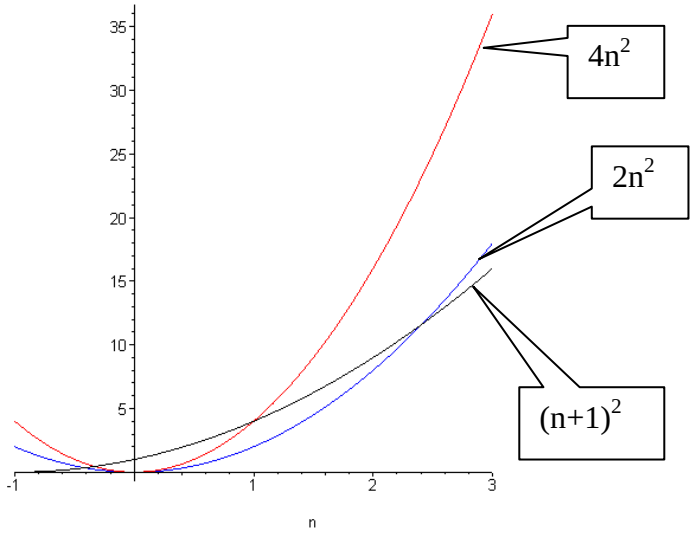

Figura 4.3. $T(n) = (n+1)^2$ es $O(n^2)$.

Si seguimos disminuyendo c, por ejemplo 1.1, $T(n)$ queda acotada para n>20,48.
Considerando los casos anteriores, una mejor definición de la función O es la siguiente:

Se dice que $T(n)$ es $O(n^i)$, si existen c y n_0 tales que: $T(n) <= c\ n^i$ con $n>=n_0$

Intentemos probar que: $T(n) = 3n^3 + 2\ n^2$ es $O(n^3)$

Debemos encontrar un c que cumpla: $3n^3 + 2\ n^2 <= cn^3$
Reemplazando c=5, en la relación anterior, se encuentra $n>=1$. Por lo tanto c=5 y n_0=1.
Debido a que existen c y n_0, $T(n)$ es $O(n^3)$
La generalización para otro tipo de funciones se logra, con la siguiente definición.

4.5. Función O.

Se dice que $T(n)$ es $O(\ f(n)\ )$, si existen c y n_0 tales que: $T(n) <= c\ f(n)$ con $n>=n_0$

Sin embargo se necesita un mejor concepto para acotar el orden o magnitud de una función. Esto considerando el primer ejemplo, en el que se podía decir que $T(n)$ era $O(n^3)$ y también que era $O(n^2)$.

4.6. Función Θ.

Una mejor definición para acotar funciones, es la función Θ que define simultáneamente cotas superior e inferior para $T(n)$.

Se dice que $T(n)$ es $\Theta(\ f(n)\ )$, si existen c_1, c_2 y n_0 tales que:

$$c_1\ f(n) <= T(n) <= c_2\ f(n) \text{ con } n>=n_0$$

Para el ejemplo anterior, c_1 = 3, y c_2 = 5, con n_0 =1.

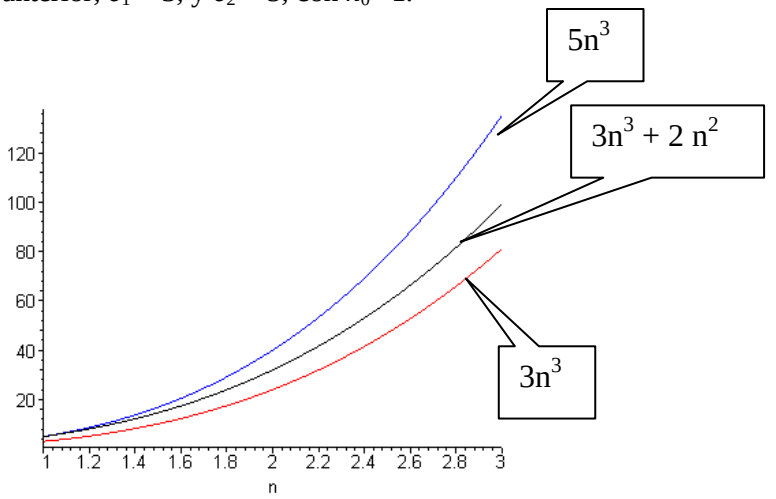

Figura 4.4. $T(n)= 3n^3 + 2\ n^2$ es $\Theta(n^3)$.

La definición de Θ encuentra una $f(n)$ que acota tipo sándwich a la función $T(n)$.

Cuando empleemos a lo largo del texto la notación O, realmente estamos refiriéndonos a la definición de Θ.

4.7. Costo unitario.

Aplicando la definición puede comprobarse que las funciones constantes tienen complejidad O(1).

Ejemplo. Sea $T(n) = 5$.
Se puede escribir: $c_1*1 <= 5 <= c_2*1$ con $n>=n_0$
Y es simple encontrar: $c_1 = 4$, $c_2 = 6$ y n_0 cualquiera.
Es decir $f(n) =1$. Con lo cual, se puede afirmar que : $T(n) =5$ es O(1) para todo n.

El tiempo de ejecución de un algoritmo que no depende significativamente del tamaño de la entrada es de complejidad O(1).

Una de las mayores simplificaciones para cálculos de complejidad es considerar que las acciones primitivas del lenguaje son de costo unitario.

Por ejemplo es usual considerar que el cálculo de una expresión, la asignación, son de complejidad O(1). Excepcionalmente, en comparaciones más detalladas de algoritmos se cuentan aparte las comparaciones, y los movimientos o copias de datos.

4.8. Regla de concatenación de acciones. Regla de sumas.

Se realiza la acción A, como la secuencia dos acciones A_1 y A_2 de complejidades temporales $T_1(n)$ y $T_2(n)$ respectivamente.

Teorema de sumas.

Si $T_1(n)$ es O(f(n)) y $T_2(n)$ es O(g(n)) entonces: A es de complejidad O(max($f(n)$, $g(n)$).

Demostración:
Por definición:
$T_1(n) <= c_1 f(n)$ para $n >= n_1$
$T_2(n) <= c_2 g(n)$ para $n >= n_2$
Sea $n_0 = \max(n_1, n_2)$

Para $n>=n_0$ se tiene: $T(n) = T_1(n) + T_2(n) <= c_1 f(n) + c_2 g(n)$ (1)

Caso a) Sea $f(n) > g(n)$
Entonces el lado derecho de la relación (1):
$c_1 f(n) + c_2 g(n) = (c_1 + c_2)f(n) - c_2(f(n) - g(n))$
Y como $c_2(f(n) - g(n))$ es positivo, se puede escribir:
$c_1 f(n) + c_2 g(n) < = (c_1 + c_2)f(n)$

Caso b) Sea $f(n) < g(n)$
Entonces el lado derecho de la relación (1):
$c_1 f(n) + c_2 g(n) = (c_1 + c_2)g(n) - c_1(g(n) - f(n))$
Y como $c_1(g(n) - f(n))$ es positivo, se puede escribir:
$c_1 f(n) + c_2 g(n) <= (c_1 + c_2)g(n)$

De los dos resultados anteriores, se obtiene la desigualdad:
$T(n) = T_1(n) + T_2(n) <= (c_1 + c_2) \max(f(n), g(n))$ para $n_0 = \max(n_1, n_2)$.
Aplicando la definición, de la función O, se reconoce que:

$$T(n) = O(\max(f(n), g(n))$$

Corolario.

Si se conoce que $f > g$, entonces: $O(f + g)$ es $O(f)$.

Ejemplo:
$O(n^2 + n) = O (n^2)$ para valores de n tales que $n^2 > n$.

4.9. Regla de productos.

Si $T_1(n)$ es $O(f(n))$ y $T_2(n)$ es $O(g(n))$ entonces: $T_1(n)T_2(n)$ es $O(f(n) g(n))$.

Demostración.
Por definición:
$T_1(n) <= c_1 f(n)$ para $n >= n_1$
$T_2(n) <= c_2 g(n)$ para $n >= n_2$
Sea $n_0 = \max(n_1, n_2)$

Para $n>=n_0$ se tiene: $T(n) = T_1(n) T_2(n) <= c_1c_2 f(n)g(n)$
Con $c=c_1c_2$ y aplicando la definición de la función $O(n)$ se logra que $T(n)$ es $O(f(n) g(n))$ para $n>=n_0$.

Ejemplos:
$O(3 n^2) = O (n^2)$ ya que 3 es O(1), y n^2 es $O(n^2)$.
La regla del producto también puede aplicarse en: $n*O(n) = O (n^2)$

Si c es una constante y n el tamaño de la entrada:
$O(c) = c*O(1) = O(1)$
$O(cn) = c*O(n) = O(n)$

4.10. Regla de alternativa.

if (c) a1; else a2;

En cálculos de peor caso se toma la complejidad de la acción de mayor orden. Luego se considera la regla de sumas para el cálculo de la condición y la acción.

Considerando de costo unitario el cálculo de la condición, la Figura 4.5 muestra la complejidad de la sentencia if.

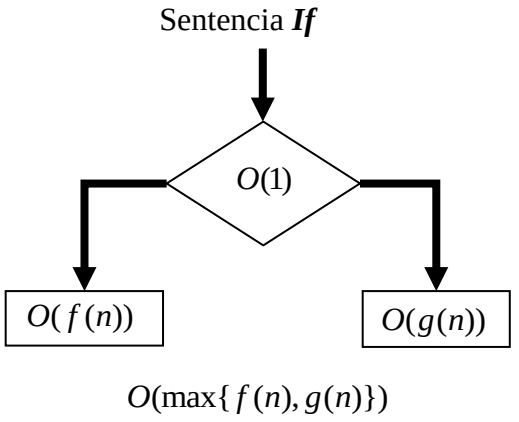

Figura 4.5. Costo de alternativa.

4.11. Regla de iteración.

for (i=0; i< n; i++) a;
Por regla de sumas se tiene *n* veces la complejidad temporal de la acción *a*.

Si la acción del bloque *a* es O(1) entonces el for es de complejidad *n**O(1) = O(*n*)
La Figura 4.6, considera costos unitarios para la inicialización, reinicio, y cálculo de la condición; la complejidad del bloque es O(*f*(*n*)); el número de veces es de complejidad O(*g*(*n*)).

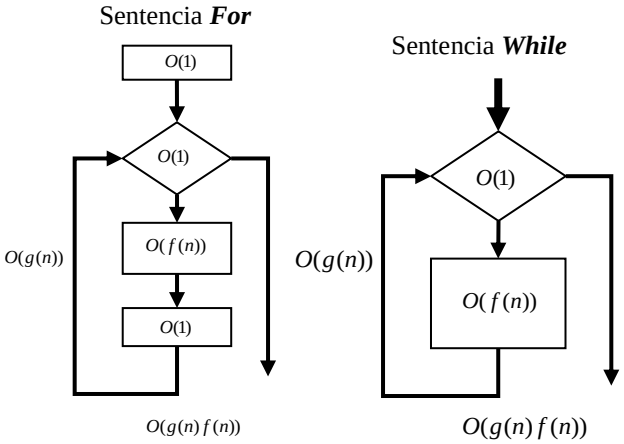

Figura 4.6. Costo de lazo for y while.

Ejemplo 4.1.

Se tienen tres for anidados:

```
for (i=1; i<=n-1; i++)
  for( j= i+1; j <=n; j++)
    for (k=1; k <=j; k++){ O(1) ;}
```

Calcular la complejidad del segmento.

La sumatoria más interna se realiza j veces.

$$\sum_{k=1}^{k=j} O(1) = jO(1) = O(j)$$

El segundo for realiza la sumatoria: $\sum_{j=i+1}^{j=n} j = (i+1)+(i+2)+...+(n-1)+n$

La cual puede componerse según: $\sum_{j=i+1}^{j=n} j = \sum_{j=1}^{j=n} j - \sum_{j=1}^{j=i} j$

Pero son conocidas las sumas de los enteros y los cuadrados de los números:

$$\sum_{j=1}^{j=n} j = \frac{n(n+1)}{2} = O(n^2)$$

$$\sum_{j=1}^{j=n} j^2 = \frac{n(n+1)(2n+1)}{6} = O(n^3)$$

Reemplazando la primera fórmula en las sumatorias del lado derecho, se logra:

$$\sum_{j=i+1}^{j=n} j = \sum_{j=1}^{j=n} j - \sum_{j=1}^{j=i} j = \frac{n(n+1)}{2} - \frac{i(i+1)}{2}$$

El tercer for realiza la sumatoria:

$$\sum_{i=1}^{i=n-1} \left(\frac{n(n+1)}{2} - \frac{i(i+1)}{2}\right) = \frac{n(n+1)}{2}(n-1) - \frac{1}{2}\sum_{i=1}^{i=n-1}(i^2+i)$$

Lo cual se logra extrayendo la parte que no depende de i fuera de la sumatoria. Y reemplazando las fórmulas de las sumatorias conocidas, con los índices correspondientes:

$$\sum_{i=1}^{i=n-1} \left(\frac{n(n+1)}{2} - \frac{i(i+1)}{2}\right) = \frac{n(n+1)(n-1)}{3} = \frac{n^3-n}{3} = O(n^3)$$

4.12. Algoritmos recursivos.

En varios casos de mucho interés se conoce la complejidad temporal de un algoritmo mediante una relación de recurrencia.

Sea:

$T(n) = T(n/2) + c$ con $T(1) = c$.

Es decir el algoritmo aplicado a una entrada de tamaño *n*, puede descomponerse en un problema de la mitad del tamaño. El costo de la descomposición es una constante de valor *c*.
Para resolver una relación de recurrencia es preciso conocer el costo para una entrada dada; en este caso para entrada unitaria el costo es la constante *c*.

Otra forma de visualizar el tipo de solución que da el algoritmo al problema anterior, es considerar que en cada pasada se descarta la mitad de los datos de entrada.

Si a partir del caso conocido se van evaluando casos más complejos, pueden obtenerse:

T(1)	= c	= c	= 0 + c
T(2)	= T(1) + c	= 2c	= c + c
T(4)	= T(2) + c	= 3c	= 2c + c
T(8)	= T(4) + c	= 4c	= 3c + c
T(16)	= T(8) + c	= 5c	= 4c + c

$T(2^i) = T(2^{i-1}) + c = (i+1)c = ic + c = c(i+1)$

Nótese que se han evaluado en números que son potencias de dos, para obtener con facilidad una expresión para el término general.
Con $2^i = n$, se tiene sacando logaritmo de dos en ambos lados:
$\log_2(2^i) = \log_2(n)$, pero $\log_2(2^i) = i \log_2(2) = i$. Resultando $i = \log_2(n)$.

Reemplazando en el término general resulta: $T(n) = c(\log_2(n) + 1)$

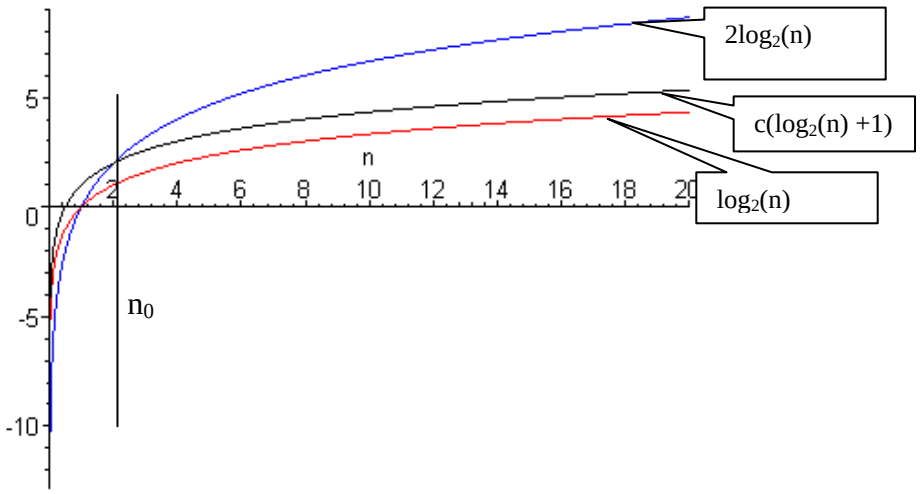

Figura 4.7. $T(n) = c(\log_2(n) + 1)$ es $O(\log_2(n))$

Las gráficas de la Figura 4.7, muestran que para $T(n)$ existen c_1 y c_2 que la acotan por encima y por debajo, para $n > 2{,}2$. Finalmente, se tiene que la solución de la ecuación de recurrencia es:

$$T(n) = O(\log_2(n))$$

Las gráficas comparan el costo lineal versus el logarítmico.

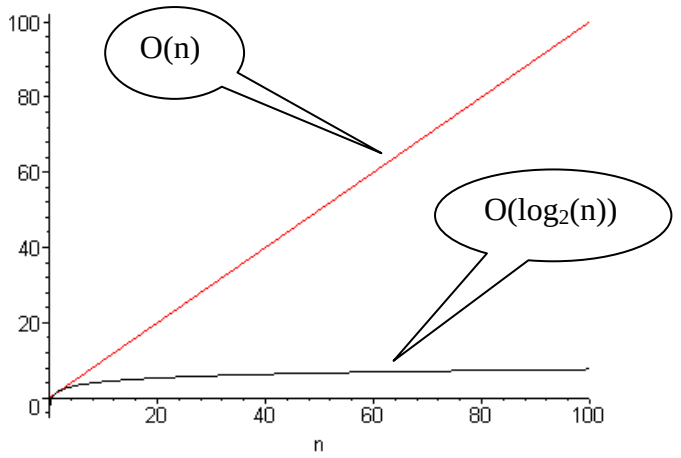

Figura 4.8 Costo lineal versus costo logarítmico.

Ejemplo 4.2. Evaluando la complejidad en función del tiempo.

Si $T(n)$ refleja el número de instrucciones de costo unitario que deben realizarse para resolver para n entradas, puede tenerse una medida en unidades de tiempo conociendo el valor aproximado de la duración de una instrucción.

Si O(1) es equivalente a 1 [μseg], se puede construir la siguiente tabla, en cada columna se tiene una complejidad temporal diferente:

n	$3n^2 + 7n$	n^2	$n \log_2 n$
100	0,03 [seg]	0,01 [seg]	6,6 [mseg]
10.000	5 [minutos]	1,7 [minutos]	133 [mseg]
100.000	8 [horas]	3 [horas]	1,66[seg]
1.000.000	35 [días]	12 [días]	6 [seg]

Figura 4.9 Costo temporal.

Usando teoremas sobre comparación de funciones, se tiene que: $O(3n^2 + 7n) = O(n^2)$.
La tabla muestra que las complejidades de los dos algoritmos cuadráticos son comparables en el orden de magnitud.

Ejemplo 4.3. Aplicación a un algoritmo sencillo.

Calcular la complejidad del segmento que obtiene el mínimo elemento de un arreglo.

```
min = A[0];
```

```
for(i=1; i<n; i++)
  if(A[i] < min) min = A[i];
```

Si la comparación y la asignación son de costo O(1), entonces el if, en peor caso, es de costo O(1) + O(1) = O(1).
El for se realiza (*n*-1) veces, su costo es: (*n*-1) O(1) = O(*n*-1) = O(*n*).
La concatenación de la asignación con el for es de costo: O(1) + O(*n*) = O(*n*)
Finalmente el segmento es de orden de complejidad O(*n*).

Ejemplo 4.4. Comparación de complejidad entre dos algoritmos.

Comparar dos algoritmos para calcular la suma de los primeros n enteros.
Algoritmo 1.

```
suma= 0;
for(i=1; i<=n; i++)   suma+=i;
```

Algoritmo 2.

```
Suma = n*(n+1)/2;
```

Algoritmo 1.
La primera asignación a la variable suma es O(1).
El for realiza una vez la asignación inicial y *n* veces: test de condición, suma, e incremento de variable de control; más un test de condición con el que se termina el for.
Para el for, entonces, se tiene: O(1) + *n**(O(1)+O(1) +O(1)) + O(1) = O(*n*)

El total es: O(1)+O(*n*) = O(*n*).

Algoritmo 2.
Costo de la suma, más costo de la multiplicación, más costo de la división. Es decir: O(1)+O(1)+O(1), lo cual resulta O(1).

Por lo tanto conviene emplear el algoritmo 2.

Ejemplo 4.5. Búsqueda en arreglos.

Un problema básico es buscar si un valor está presente en una de las componentes de un arreglo.

Con las siguientes definiciones:

```
typedef int Tipo;       /* tipo de item del arreglo */
typedef int Indice;     /* tipo del índice */
#define noencontrado  -1
#define verdadero  1
#define MaxEntradas  10

Tipo Arreglo[MaxEntradas];              //Define el arreglo en donde se busca
```

4.13. Búsqueda secuencial.

La búsqueda secuencial compara la clave con cada una de las componentes. La primera vez que encuentre un elemento del arreglo igual al valor buscado se detiene el proceso. No encuentra claves repetidas y no se requiere que el arreglo esté ordenado.
Si lo recorre en su totalidad, cuidando de no exceder los rangos del arreglo, y no lo encuentra debe indicarlo con algún valor específico de retorno. En el diseño se considera retornar el índice de la componente que cumple el criterio de búsqueda, se decide entonces que un retorno con valor -1 (ya que no es un índice válido), indicará que el valor buscado no fue hallado.

```
Indice BusquedaSecuencial(Tipo A[], Indice Inf, Indice Sup, Tipo Clave)
{    Indice i;

     for(i = Inf; i<=Sup; i++)
      if (A[i] ==  Clave) return(i);
      return (noencontrado) ;
}
```

La evaluación de la condición del if es O(1), también el retorno es O(1). El bloque que se repite es entonces O(1).

La iniciación del for es O(1). El test de la condición del for es O(1), también el incremento de i es O(1). El bloque se repite: (Sup-Inf +1) veces en peor caso.

La complejidad es:

$$O(1) + (Sup\text{-}Inf +1)*(O(1) +(O(1) + O(1)) +O(1))$$

Simplificando:

$$(Sup\text{-}Inf+1)\ O(1) = O(Sup\text{-}Inf+1)$$

La entrada, en este caso, es el número de componentes en las que se busca.

Si n= Sup-Inf+1, se tiene finalmente:

$$T(n) = O(n).$$

4.14. Búsqueda binaria (Binary Search)

Se requiere tener un arreglo ordenado en forma ascendente. El algoritmo está basado en ubicar, mediante la variable auxiliar *M*, la mitad del arreglo aproximadamente. Entonces o se lo encuentra justo en el medio; o en la mitad con índices menores que *M* si el valor buscado es menor que el de la componente ubicada en la mitad; o en la mitad con índices mayores que *M* si el valor buscado es mayor. El costo de encontrar la mitad es de costo constante. El ajustar los índices también es de costo constante, corresponde a los dos if then else anidados. Si se tienen *n* componentes, la complejidad puede describir según:

$$T(n) = T(n/2) + c$$

El costo, en un arreglo con una componente, es constante; es decir T(1) = O(1). La solución de esta ecuación de recurrencia es: $T(n) = O(\log(n))$.

```
int BusquedaBinaria(Tipo A[], Indice Inf, Indice Sup, Tipo Clave)
{    Indice M;

     while (verdadero)
    {
      M = (Inf + Sup)/2;
      if (Clave < A[M])
         Sup = M - 1;
      else if (Clave > A[M])
         Inf = M + 1;
      else   return M;
      if (Inf > Sup) return (noencontrado) ;
    }
}
```

0	
1	
2	
…	
….	
M-1	
M	
M+1	
….	
….	
n-3	
n-2	
n-1	

Figura 4.10 Búsqueda binaria.

Un aspecto importante del diseño de un bloque repetitivo es asegurar que éste termina. Al inicio se tiene que Sup> Inf. Cada vez que Sup disminuye, Inf no cambia; y también cuando Inf aumenta Sup no cambia. En ambos casos la condición Inf >Sup cada vez está más cercana a cumplirse.

Si el valor buscado es menor que el menor valor contenido en el arreglo, después de algunas iteraciones, no importando si *n* inicialmente es par o impar, se llega a que Inf, Sup y M apuntan a la primera componente del arreglo, etapa en la que se compara con la primera componente, produciendo Inf>Sup. Similar situación se produce cuando el valor buscado es mayor que la mayor componente del arreglo. Lo cual verifica que el algoritmo siempre termina en un número finito de pasos, y que trata bien el caso de que la búsqueda falle.

4.15. Sobre el costo O(1).

Las personas que conocen los detalles internos de un procesador saben que una suma demora menos que una multiplicación o división; esto si los números son enteros. En el caso de flotantes los costos son aún mayores que para enteros.

Veamos esto con más detalle. Si consideramos números de *n* bits, y un sumador organizado de tal manera que sume primero los bits menos significativos, luego la reserva de salida de éstos más los dos bits siguientes; y así sucesivamente. Si se considera O(1) el costo de la suma de un bit, entonces la suma de dos enteros de *n* bits será O(*n*). Esta estructura de un sumador con propagación ondulada de la reserva puede mejorarse con circuitos adicionales para generar reservas adelantadas.

Un algoritmo primitivo para efectuar multiplicaciones es mediante la suma repetitiva de uno de los operandos.
Por ejemplo el producto de dos números de 4 bits:

```
      0101*1100
 ------
   0000
  0000
 0101
0101
-------
0111100
```

Esto implica efectuar *n* sumas si los operandos son de *n* bits. Lo cual implica un costo O(n^2) para la multiplicación, considerando que cada suma es de costo O(*n*). Razón por la cual se suelen diseñar unidades de multiplicación, en hardware, con mejores algoritmos.

En un ambiente de microcontroladores o microprocesadores que no tengan implementada la operación multiplicación o división, pero que si tengan en su repertorio de acciones: sumas, restas y desplazamientos a la izquierda y derecha en un bit, se consideran esas instrucciones de costo O(1); nótese que en el ambiente anterior, la suma era O(*n*).

Consideremos, en este último contexto, dos algoritmos para multiplicar:

4.17.1. Algoritmos de multiplicación.

```
/*  retorna  m*n   */
unsigned int multipliquelineal(unsigned int m, unsigned int n)
{   unsigned int r=0;
    while ( n>0 )
      {  r+=m; n--; }
    return(r);
}
```

El bloque se repite *n* veces, y está constituido por un test de condición, una suma y un incremento. Todas operaciones que pueden traducirse a instrucciones de máquina, y se consideran de costo O(1). Entonces esta multiplicación es de costo O(*n*) o lineal.

```
/* retorna  m*n */
unsigned int multipliquelog(unsigned int m, unsigned int n)
{  unsigned int r=0;

   while ( n>0 )
   {  if(n&1) r+=m;
      m*=2; n/=2;
   }
   return(r);
}
```

En cada ciclo del while se divide por dos el valor de *n*.
La multiplicación por dos, equivale a un corrimiento a la izquierda en una posición; la división por dos equivale a un corrimiento a la derecha en una posición; el test del bit menos significativo se realiza con una operación and; acumular el producto parcial en la local *r*, se efectúa con una suma. Todas estas operaciones, en este ambiente se consideran de costo constante: O(1).

El algoritmo puede describirse según:

$$T(n) = T(n/2) + O(1) \quad \text{con} \quad T(1) = O(1).$$

Resultado obtenido antes, como solución de una ecuación de recurrencia, con costo logarítmico: $O(\log_2 n)$; concluyendo que la segunda rutina, es mucho mejor que la primera. Debe notarse que en este ejemplo se emplea el valor de uno de los operandos como el tamaño de la entrada.

Dependiendo de lo que consideremos como costo unitario, un algoritmo puede tener costos muy diferentes.

4.17.2. Algoritmos de división.

Consideremos un algoritmo primitivo de división, basado en restas sucesivas.

```
//Retorna cuociente q y resto r.    n = q*d + r.  El resto r se pasa por referencia.
unsigned int dividelineal(unsigned int n, unsigned int d, unsigned int *r)
{   unsigned int q=0;

    while (n>=d) { n-=d; q++; }
    *r= n;     //escribe el resto
    return(q);
}
```

En su peor caso, con denominador unitario, y *n* el máximo representable, se tiene costo: O(*n*).

El siguiente algoritmo ha sido tradicionalmente usado en procesadores que no tengan implementada una unidad de multiplicación de enteros en hardware.

```
//Retorna cuociente q y resto r. Con n = q*d + r .
unsigned int dividelog(unsigned int n, unsigned int d, unsigned int *r)
{   unsigned int dd=d, q=0;
    *r = n;
     while (dd<=n) dd*=2;
     while (dd > d)
          {  dd/=2; q= q*2;
             if (dd<=*r) {*r-=dd; q++;}
          }
     return(q);
}
```

El primer while duplica el denominador hasta que sea mayor que el numerador.
Por ejemplo para operandos de 8 bits, el mayor representable será: 11111111 en binario, lo cual equivale a 255. Si *d* es 1, el while (en su peor caso) genera los siguientes valores para *dd*: 2, 4, 8, 16, 32, 64 128, 256. Es decir se realiza 8 veces. Lo cual también se puede calcular mediante $\log_2(256) = \log_2(2^8) = 8$.

En su peor caso con denominador unitario y con el máximo valor para *n*, se tienen $\log_2 n$ repeticiones.

El segundo while divide por 2 el tamaño de *dd*, cada vez que se realiza el lazo, su costo es: $O(\log_2 dd)$.
Sumando estas complejidades, en un peor caso, se tendrá que $O(\log_2 n)$. Con *n* el mayor número representable.

Conclusión: En los algoritmos que estudiaremos, en este curso, es importante tener claridad acerca del significado del costo O(1).

4.18. Complejidad *n*log(*n*).

Existen algoritmos que pueden describirse por la siguiente relación de recurrencia:

$$T(n) = 2T(n/2) + cn, \text{ con } T(1) = c.$$

En cada iteración el problema puede descomponerse en dos subproblemas similares pero de la mitad del tamaño. El costo de la descomposición es proporcional a *n*.

La fórmula anterior es un ejemplo de los algoritmos basados en el paradigma: *Dividir para vencer*. Origina los métodos de solución basados en particiones.

La solución de la ecuación de recurrencia es : $T(n) = cn(\log_2(n)+1)$

La cual puede calcularse en forma similar al caso de costo logarítmico.

T(1) = c
T(2) = 2*T(1) + c*2 = 4c = 2c *2
T(4) = 2*T(2) + c*4 = 12c = 4c *3
T(8) = 2*T(4) + c*8 = 32c = 8c *4
T(16) = 2*T(8) + c*16 = 80c = 16c *5

$T(2^i) = 2*T(2^{i-1}) + c*2^i$ = $= 2^i\ c\ *(i+1)$

Con $2^i = n$, se tiene $i = \log_2(n)$, sacando logaritmo de dos en ambos lados.

Reemplazando en: $T(2^i)$ $= 2^i\ c\ *(i+1)$

Se obtiene, finalmente:

$$T(n) = n\ c\ *(\log_2(n) + 1)$$

La complejidad temporal de T(n) es $\Theta(\ n \log(n)\ )$ para n>=2.
Por el teorema de los productos, no importa la base de los logaritmos.

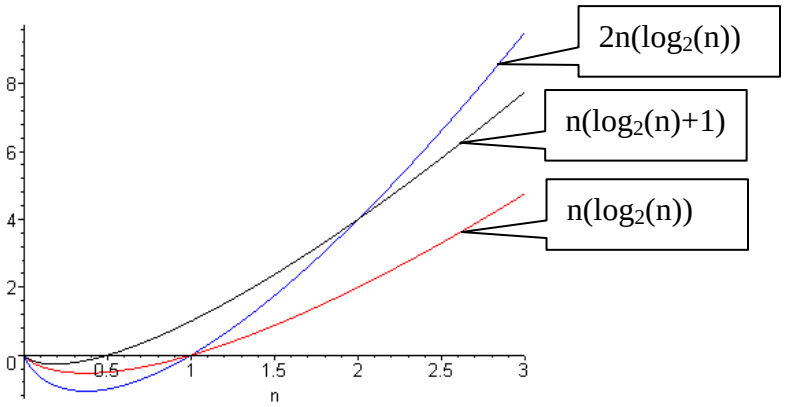

Figura 4.11 Complejidad $\Theta(\ n \log(n)\ )$.

La gráfica a continuación compara la complejidad cuadrática con la n*log(n).

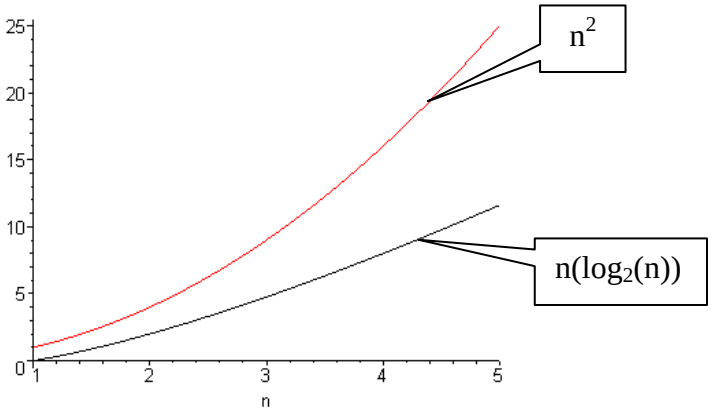

Figura 4.12 Θ(*n* log(*n*)) versus cuadrática.

La cuadrática crece mucho más rápidamente que la *n**log(*n*).

La *n**log(*n*) crece mucho más rápidamente que la lineal, lo que se muestra a continuación:

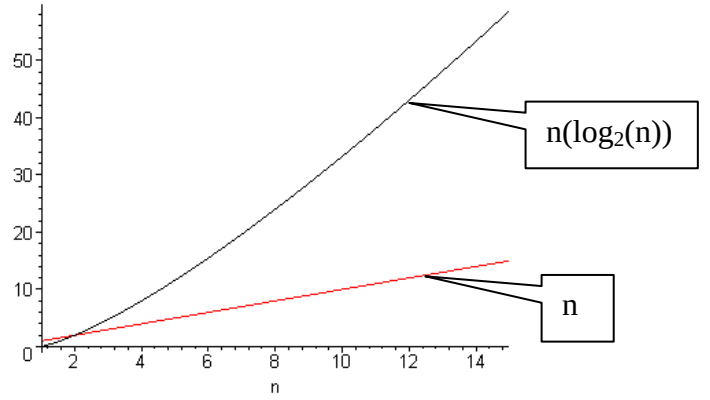

Figura 4.13 Θ(*n* log(*n*)) versus lineal.

Para mayores valores de n, se aprecian mejor las diferencias.

4.19. Comparación entre complejidades típicas.

La siguiente gráfica compara cuatro complejidades usuales.

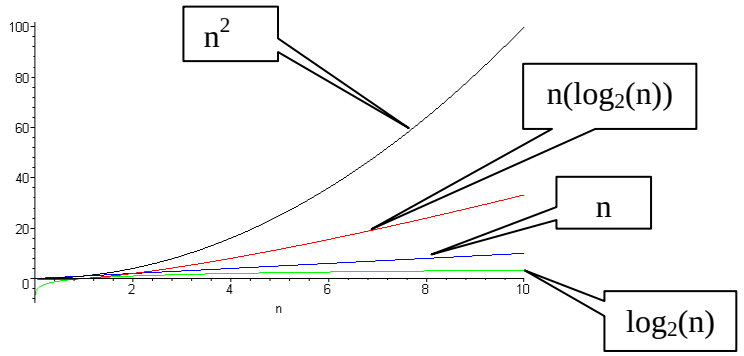

Figura 4.14 Comparación entre cuatro tipos de complejidades.

Cuando un programador principiante encuentra que el primer algoritmo que se le ocurrió, para resolver un problema, es O(n^2), es posible que le sorprenda la existencia de un algoritmo (que estudiaremos en este texto) de complejidad O($n$log(n)).

Lo mismo puede decirse de primeros intentos de diseño de algoritmos que conducen a uno de costo O(n), que pueden ser planteados con complejidad O(log(n)).

4.20. Estudio adicional.

En los textos de referencia existen, normalmente al inicio, capítulos dedicados al cálculo de complejidades temporales, y al acotamiento del orden de crecimiento. Donde se dan métodos para acotar funciones o para resolver relaciones de recurrencia.

4.21. Solución de ecuaciones de recurrencia.

4.21.1. Recurrencias homogéneas.

Son del tipo:

$$a_0T(n)+a_1T(n-1)+a_2T(n-2)+\ldots+a_kT(n-k)=0$$

Donde los a_i son coeficientes reales y k un número natural entre 1 y n.

Si se reemplaza $T(n)=x^n$, resulta la ecuación:

$$a_0x^n+a_1x^{n-1}+a_2x^{n-2}+\ldots+a_kx^{n-k}=0$$

Factorizando:

$$(a_0x^k+a_1x^{k-1}+a_2x^{k-2}+\ldots+a_k)x^{n-k}=0$$

Se tiene entonces la ecuación característica:

$$a_0x^k+a_1x^{k-1}+a_2x^{k-2}+\ldots+a_k=0$$

4.21.1.1. Raíces diferentes.

Si las k raíces resultan distintas: $x_1, x_2, ..., x_k$, la solución de la ecuación de recurrencia es una combinación lineal de las soluciones. Donde los c_i se determinan a partir de las condiciones iniciales.

$$T(n) = \sum_{i=1}^{i=k} c_i x_i^n$$

Para resolverla se requieren k condiciones iniciales: $T(0),\ T(1),\ T(2),...,\ T(k-1)$

Ejemplo 4.6.

La ecuación de recurrencia para cálculos de complejidad en árboles AVL, queda dada por la ecuación de recurrencia de segundo orden, de Fibonacci, con $n \geq 2$:

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

Para resolverla es necesario conocer los valores iniciales: $T(0) = 0,\ T(1) = 1$.

Con el reemplazo: $T(n) = x^n$, resulta: $x^2 = x + 1$, ecuación de segundo grado con solución: $x_{1,2} = \frac{1 \pm \sqrt{5}}{2}$. Entonces la solución de la ecuación de recurrencia es:

$$T(n) = c_1 (\frac{1+\sqrt{5}}{2})^n + c_2 (\frac{1-\sqrt{5}}{2})^n$$

La que evaluada en n=0 y n=1, permite calcular las constantes, resultando:

$$T(n) = \frac{1}{\sqrt{5}} (\frac{1+\sqrt{5}}{2})^n - \frac{1}{\sqrt{5}} (\frac{1-\sqrt{5}}{2})^n$$

Realizando los cálculos con el procesador Maple, se obtiene:

```
> S3:= rsolve( { T(n)=T(n-1)+T(n-2), T(0) = 0,T(1)=1}, T(n)):
> evalf(S3);
```

$$.4472135952\ \ 1.618033988^{\,n} - .4472135956\ \ (-.6180339886)^n$$

Puede comprobarse que el segundo término es una serie alternada que tiende rápidamente a cero; y es menor que 0,2 para n>2. Graficando los valores absolutos del segundo término, mediante:

```
> plot(abs(-.4472135956*(-.6180339886)^n),n=0..10,thickness=2);
```

Se obtiene el gráfico:

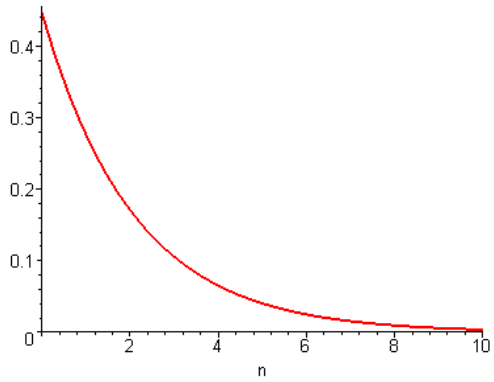

Figura 4.15 Acotamiento serie alternada

Para el primer término, se obtiene el crecimiento de T(n), mediante:

```
> plot(.4472135952*1.618033988^n,n=0..10,thickness=2);
```

Donde $\Phi = \frac{1+\sqrt{5}}{2} \approx 1,618033..$ se denomina razón áurea.

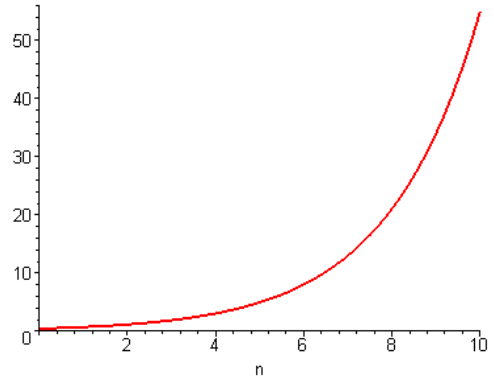

Figura 4.16 Crecimiento exponencial

Un gráfico de la función y dos adicionales que la acotan, se logra con:

```
> plot([.4472135952*1.618033988^n,.1*1.618^n,1.618^n],n=20..30,
thickness=2,color=[black,red,blue]);
```

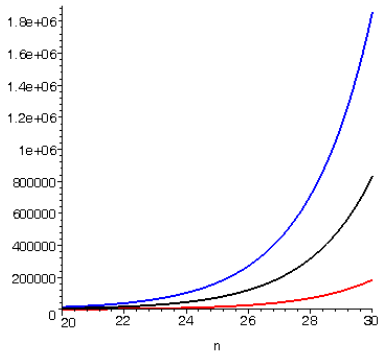

Figura 4.17 Orden de complejidad de recurrencia Fibonacci

Lo que permite establecer que:

$$T(n) = \Theta(\Phi^n)$$

4.21.1.2. Raíces múltiples.

En caso de tener una raíz múltiple de orden *m*, conservando el grado *k* de la ecuación de recurrencia, se tiene la ecuación característica:

$$(x - x_1)^m (x - x_2) + \ldots + (x - x_{k-m+1}) = 0$$

Si tuviéramos que contar los elementos de la secuencia: s_3, s_4, s_5, podemos realizar el cálculo según: (5-3 +1) = 3. Del mismo modo, podemos contar los elementos desde: $s_2, s_3, s_{4,}\ldots, s_{k-m+1}$, según: $(k-m+1)-2+1 = k-m$, lo cual muestra que la ecuación característica tiene k raíces en total.

La cual tiene como solución general a:

$$T(n) = \sum_{i=1}^{i=m} c_i n^{i-1} x_1^n + \sum_{i=m+1}^{i=k} c_i x_{i-m+1}^n$$

La primera sumatoria introduce *m* constantes en un polinomio de grado (*m*-1) en *n*.

$$\sum_{i=1}^{i=m} c_i n^{i-1} x_1^n = (c_1 n^0 + c_2 n^1 + \ldots + c_m n^{m-1}) x_1^n$$

La forma polinomial se debe a que si x_1^n es solución de la ecuación de recurrencia, entonces nx_1^n también será solución. Reemplazando x_1^n en la ecuación de recurrencia, debe cumplirse:

$$a_0 x_1^n + a_1 x_1^{n-1} + a_2 x_1^{n-2} + \ldots + a_k x_1^{n-k} = 0$$

La que derivada, respecto de x_1, resulta:

$$\frac{a_0 n x_1^n + a_1(n-1)x_1^{n-1} + \ldots + a_k(n-k)x_1^{n-k}}{x_1} = 0$$

Si x_1 no es cero, debe cumplirse:

$$a_0 n x_1^n + a_1(n-1)x_1^{n-1} + a_2(n-2)x_1^{n-2} + \ldots + a_k(n-k)x_1^{n-k} = 0$$

Lo que comprueba que nx_1^n también es solución de la ecuación de recurrencia.

Similar demostración puede realizarse para comprobar que $n^i x_1^n$ es solución, con $i < m$

La segunda sumatoria es una combinación lineal de las (*m*-*k*) raíces diferentes restantes.

Ejemplo 4.7.

Para la ecuación, con $n \geq 3$:

$$T(n) = 5T(n-1) - 8T(n-2) + 4T(n-3)$$

con condiciones iniciales $T(0) = 0,\ T(1) = 2,\ T(2) = 8$ para *k* = 0, 1, 2.

La ecuación característica resulta:

$$x^3 - 5x^2 + 8x - 4 = 0$$

Una gráfica del polinomio, muestra que tiene una raíz en x=1.

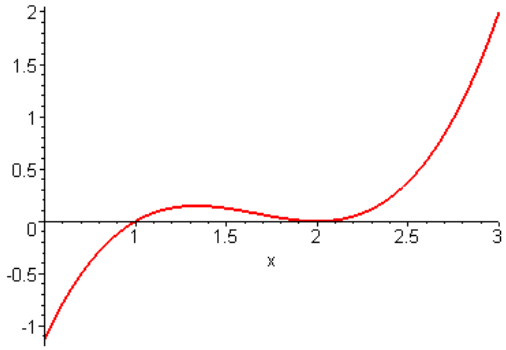

Figura 4.17.a Raíces de polinomio cúbico.

```
> plot(x^3-5*x^2+8*x-4,x=0.5..3);
```

Dividiendo el polinomio cúbico por (*x*-1) se obtiene:

$$\frac{x^3-5x^2+8x-4}{(x-1)} = x^2-4x+4 = (x-2)^2$$

La ecuación característica tiene una raíz de multiplicidad dos:

$$x^3-5x^2+8x-4 = (x-2)^2(x-1) = 0$$

Entonces la solución general es:

$$T(n) = (c_1 n^0 + c_2 n^1)2^n + c_3 1^n$$

Evaluando *T(n)* en las condiciones iniciales obtenemos, tres ecuaciones:

$$T(0) = (c_1 + c_2 0)2^0 + c_3 = 0$$
$$T(1) = (c_1 + c_2)2^1 + c_3 = 2$$
$$T(2) = (c_1 + c_2 2)2^2 + c_3 = 8$$

Las que permiten obtener: $c_1 = 0,\ c_2 = 1,\ c_3 = 0$.

```
> solve({c1+c3=0,2*c1+2*c2+c3=2,4*c1+8*c2+c3=8},{c1,c2,c3});
```

Reemplazando las constantes, la solución de la recurrencia, resulta:

$$T(n) = n2^n$$

Empleando Maple:

```
> S1:=rsolve({T(n)= 5*T(n-1)-8*T(n-2)+4*T(n-3),
              T(0)=0,T(1)=2,T(2)=8},T(n)):
> simplify(factor(S1));
```

$$2^n\, n$$

Puede graficarse la función y sus cotas, mediante:

```
> plot([S1,2*S1,0.5*S1],n=4..9,color=[black,blue,red]);
```

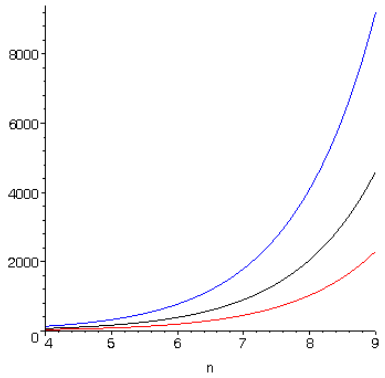

Figura 4.18 Cotas Ejemplo 4.7

Entonces la complejidad de $T(n)$, resulta:

$$T(n) = \Theta(n2^n)$$

Este crecimiento exponencial, no polinomial, es característico de determinado tipo de problemas que son clasificados como NP. Algunos de estos problemas no pueden ser resueltos por las computadoras actuales; a éstos se los denomina NPC por NP completo.

4.21.2. Recurrencias no homogéneas.

Veremos algunos tipos de ecuaciones de recurrencia no homogéneas que tienen solución conocida.

4.21.2.1. Excitación potencia de n.

Cuando la excitación, el lado derecho de la ecuación de recurrencia, es una potencia de *n*; con *b* un número real, se tiene:

$$a_0T(n) + a_1T(n-1) + a_2T(n-2) + \ldots + a_kT(n-k) = b^n$$

Puede intentarse, mediante una manipulación algebraica, la transformación a una ecuación homogénea. Esto no siempre es sencillo.

Ejemplo 4.8.

Sea la relación de recurrencia no homogénea, con $n \geq 1$:

$$T(n) - 2T(n-1) = 3^n$$

Con condición inicial: $T(0) = 0$

Si se plantea, la relación, en (n+1), se obtiene:

$$T(n+1)-2T(n)=3^{n+1}$$

Multiplicando por 3, la ecuación original, y restándolas, se logra la homogénea:

$$T(n+1)-5T(n)+6T(n-1)=0$$

Con ecuación característica:

$$x^2-5x+6=0$$

Con soluciones: $x_1=3,\ x_2=2$. La solución general es:

$$T(n)=c_1 3^n+c_2 2^n$$

En la ecuación original, se puede calcular: $T(1)=2T(0)+3^1=3$

Evaluando las constantes, mediante las ecuaciones:

$$T(0)=c_1 3^0+c_2 2^0=0$$
$$T(1)=c_1 3^1+c_2 2^1=3$$

Se obtienen: $c_1=3,\ c_2=-3$

Finalmente:

$$T(n)=3\cdot 3^n-3\cdot 2^n$$

En Maple:

```
> S4:= rsolve( { T(n)-2*T(n-1) =3^n, T(0) = 0, T(1)=3}, T(n));
```

$$S4 := -3\,2^n+3\,3^n$$

Pueden encontrarse dos funciones que acoten, por encima y por debajo, a la función, mediante:

```
> plot([S4,1*3^n,2*3^n],n=1..5,thickness=2,color=[black,red,blue]);
```

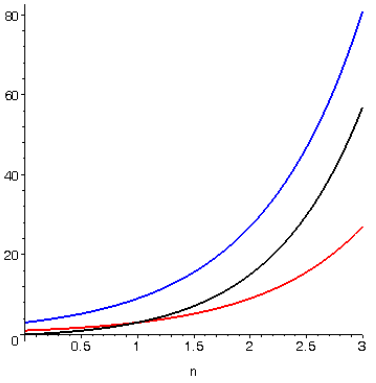

Figura 4.19 Cotas Ejemplo 4.8

Entonces, se concluye que para n>1: $T(n) = \Theta(3^n)$

4.21.2.2. Excitación polinómica.

$$a_0 T(n) + a_1 T(n-1) + a_2 T(n-2) + ... + a_k T(n-k) = p(n)$$

Donde $p(n)$ es un polinomio de grado d.

Se intenta llegar a una ecuación característica del tipo:

$$(a_0 x^k + a_1 x^{k-1} + a_2 x^{k-2} + ... + a_k)(x-1)^{d+1} = 0$$

La cual puede tratarse como una ecuación homogénea con raíces múltiples. La raíz múltiple estará asociada a un polinomio en *n*, y ya conocemos un método general para resolverlas.

Ejemplo 4.9.

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

El polinomio es de grado 1, por lo tanto d=1.
La ecuación característica resulta:

$$(x-2)(x-1)^2 = 0$$

Con solución general, para raíz simple en 2 y una múltiple doble en 1:

$$T(n) = c_1 2^n + (c_2 + c_3 n)1^n$$

Con T(0) =0, resultan: T(1)=1, T(2)=4; y pueden calcularse las constantes:

$$c_1 = 2,\ c_2 = -2,\ c_3 = -1$$

Finalmente, la solución resulta:

$$T(n) = 2 \bullet 2^n - n - 2 = \Theta(2^n)$$

La ecuación característica, en forma de polinomio, resulta:

$$x^3 - 4x^2 + 5x - 2 = 0$$

Lo cual equivale a la ecuación homogénea:

$$T(n) = 4T(n-1) - 5T(n-2) + 2T(n-3)$$

Comparando con la ecuación original, debería cumplirse:

$$2T(n-1)-5T(n-2)+2T(n-3)=n$$

Entonces para transformar en una ecuación homogénea se nos debería haber ocurrido derivar la ecuación homogénea anterior a partir de la original no homogénea. Para esto es preciso encontrar una expresión para n que dependa de $T(n-1), T(n-2)$ y $T(n-3)$.

Empleando la ecuación original se obtienen para: $T(n-1)$ y $T(n-2)$:

$$T(n-1)=2T(n-2)+n-1$$
$$T(n-2)=2T(n-3)+n-2$$

Eliminando éstas en la ecuación anterior se comprueba la igualdad del lado izquierdo con n.

En Maple, basta escribir:

```
> S3:=rsolve( { T(n) = 2*T(n-1)+n , T(0) = 0}, T(n));
```

$$S3 := 2\,2^n - 2 - n$$

Para verificar el orden del crecimiento, pueden dibujarse:

```
> plot([S3,2^n,3*2^n],n=2..6,thickness=2,color=[black,red,blue]);
```

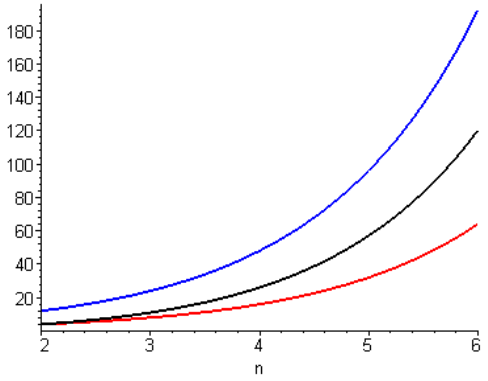

Figura 4.20 Cotas Ejemplo 4.9

Que muestra que $T(n)=\Theta(2^n)$ para $n>2$.

4.21.2.3. Método de los coeficientes indeterminados.

Permiten resolver ecuaciones de recurrencia con la forma:

$$a_0T(n)+a_1T(n-1)+a_2T(n-2)+\ldots+a_kT(n-k)=b^n p(n)$$

Donde p(n) es un polinomio de grado d.

Están basadas en descomponer la solución en sus partes homogénea y particular:

$$T(n) = T_h(n) + T_p(n)$$

Donde $T_h(n)$ es la solución homogénea, con excitación cero; y $T_p(n)$ es la solución particular.

$$T_p(n) = p_d(n) n^m b^n$$

Donde $p_d(n)$ es un polinomio, de orden d, con coeficientes que serán determinados; *m* es la multiplicidad de la raíz *b* en la ecuación característica. Notar que con excitación solamente de tipo polinomio, *m* es la multiplicidad de la raíz 1 en la ecuación característica (con *b*=1).

Se resolverán los ejemplos anteriores, usando este método.

Ejemplo 4.10.

Para:

$$T(n) - 2T(n-1) = 3^n$$

Con condición inicial: $T(0) = 0$

Se tiene la ecuación homogénea:

$$T_h(n) - 2T_h(n-1) = 0$$

Reemplazando $T_h(n) = x^n$ se obtiene la ecuación: $x^n - 2x^{n-1} = 0$

Entonces, la ecuación característica es: $x - 2 = 0$, resultando:

$$T_h(n) = c2^n$$

Como 3 no es raíz de la ecuación homogénea, se tendrá que *m* es cero; además la excitación no contiene un polinomio, entonces podemos escoger, la solución particular:

$$T_p(n) = a \cdot 3^n$$

Con a el coeficiente de un polinomio de grado 0, que deberá determinarse:

Reemplazando en la ecuación de recurrencia:

$$a \cdot 3^n - 2(a \cdot 3^{n-1}) = 3^n$$

Arreglando, resulta:

$$(a - \frac{2a}{3}) \cdot 3^n = 3^n$$

De donde resulta que:

$$a - \frac{2a}{3} = 1$$

Obteniéndose:

$$a = 3$$

La solución general es:

$$T(n) = T_h(n) + T_p(n) = c2^n + 3 \cdot 3^n$$

Para evaluar la constante c, se tiene:

$$T(0) = c2^0 + 3 \cdot 3^0 = 0$$

La que permite calcular $c = -3$
Obteniéndose igual solución que la anterior, determinada en el Ejemplo 4.8.

Ejemplo 4.11.

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

Con condición inicial: $T(0) = 0$

La solución homogénea, resulta: $T_h(n) = c2^n$

Como 1 no es solución de la ecuación homogénea, se tendrá que *m* es cero; además *b* es uno, por lo tanto el polinomio *p* debe ser de grado uno. Tenemos entonces la siguiente solución particular:

$$T_p(n) = p_1(n) = an + b$$

Que al ser reemplazada en la ecuación de recurrencia, permite obtener:

$$(a \cdot n + b) - 2 \cdot (a(n-1) + b) = n$$

Arreglando, para determinar coeficientes:

$$(-a) \cdot n + (2a - b) = n$$

De la cual se pueden plantear:

$$-a = 1$$

$$2a - b = 0$$

Entonces: $T_p(n) = an + b = -n - 2$

La solución general: $T(n) = c2^n - n - 2$

La constante se calcula de: $T(0) = c2^0 - 0 - 2 = 0$, obteniéndose igual solución que la anterior, determinada en el Ejemplo 4.9.

Ejemplo 4.12.

$$T(n)-2T(n-1)=2^n$$

Con condición inicial: $T(0)=0$

Como *b* es igual a la raíz de la ecuación homogénea, se tendrá con *d=0, m=1, b=2*, que la solución particular resulta:

$$T_p(n)=p_d(n)n^m b^n=a\cdot n\cdot 2^n$$

Reemplazando en la relación de recurrencia, se obtiene:

$$a\cdot n\cdot 2^n-2(a\cdot(n-1)2^{n-1})=2^n$$

El coeficiente debe cumplir: $a\cdot n-a\cdot(n-1)=1$, resultando $a=1$.

La solución general: $T(n)=c2^n+n2^n$, evaluada en 0:

$$T(0)=c\cdot 2^0+0\cdot 2^0=0$$

Como $c=0$, la solución es:

$$T(n)=n2^n=\Theta(n2^n)$$

En Maple, se obtiene igual solución:

```
> S4:= rsolve( { T(n)-2*T(n-1) =2^n, T(0) = 0}, T(n));
```

$$S4 := -2^n + (n+1)\,2^n$$

```
> simplify(S4);
```

$$2^n\,n$$

4.21.2.4. Método cuando n es potencia de dos.

Ejemplo 4.13.

$$T(n)=4T(n/2)+n$$

Con condición inicial: $T(1)=1$

Nótese que *n* debe ser 2 o mayor. Para *n=2*, puede calcularse: $T(2)=4T(1)+2=6$; por esta razón la condición inicial se da con *n*=1.

Si *n* es una potencia de dos, se tiene que: $n=2^k$, entonces reemplazando en la ecuación de recurrencia:

$$T(2^k)=4T(2^{k-1})+2^k$$

Con el siguiente cambio de variable: $U(k) = T(2^k)$

Se obtiene, la ecuación de recurrencia:

$$U(k) = 4U(k-1) + 2^k$$

Ecuación que podemos resolver, obteniéndose:

$$U(k) = c \cdot 4^k - 2^k$$

Arreglando, y cambiando la variable *U*, se obtiene:

$$T(2^k) = c \cdot (2^k)^2 - 2^k$$

Expresando en términos de *n*:

$$T(n) = c \cdot n^2 - n$$

La cual evaluada en *n*=1, permite calcular *c*.

$$T(1) = c \cdot 1^2 - 1 = 1$$

Finalmente:

$$T(n) = 2 \cdot n^2 - n = \Theta(n^2)$$

En Maple:

```
> S6:= rsolve( { T(n)-4*T(n/2) =n, T(1) = 1}, T(n));
```

$$S6 := n\,(2\,n - 1)$$

La determinación del orden de complejidad se logra con:

```
> plot([S6,1*n^2,2*n^2],n=2..8,thickness=2,color=[black,red,blue]);
```

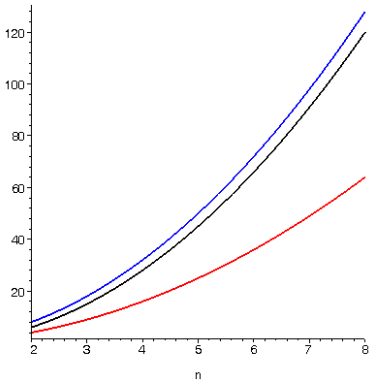

Figura 4.21 Cotas Ejemplo 4.13

Ejemplo 4.14.

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

Con condición inicial: $T(1)=1$

Nótese que *n* debe ser 2 o mayor. Para *n=2*, se tiene: $T(2)=2T(1)+2$. Por esta razón la condición inicial se da con *n*=1.

Se tiene que: $n=2^k$, entonces:

$$T(2^k)=2T(2^{k-1})+2^k$$

Con el siguiente cambio de variable: $U(k)=T(2^k)$

Se obtiene, la ecuación de recurrencia:

$$U(k)=2U(k-1)+2^k$$

Ecuación que podemos resolver, obteniéndose:

$$U(k)=c\cdot 2^k+k\cdot 2^k=T(2^k)$$

Expresando en términos de *n*, y empleando $k=\log_2(n)$:

$$T(n)=c\cdot n+n\cdot\log_2(n)$$

La cual evaluada en *n*=1, permite calcular que *c* es uno.

$$T(1)=c\cdot 1+1\cdot 0=1$$

Finalmente:

$$T(n)=n+n\cdot\log_2(n)=\Theta(n\cdot\log_2(n))$$

En Maple:

```
> S7:= rsolve( { T(n)-2*T(n/2) =n, T(1) = 1}, T(n));
```

$$S7 := n+\frac{n\ln(n)}{\ln(2)}$$

La determinación del orden de complejidad se logra con:

```
> plot([S7,n*ln(n)/ln(2),2*n*ln(n)/ln(2)],n=2..8,thickness=2,
        color=[black,red,blue]);
```

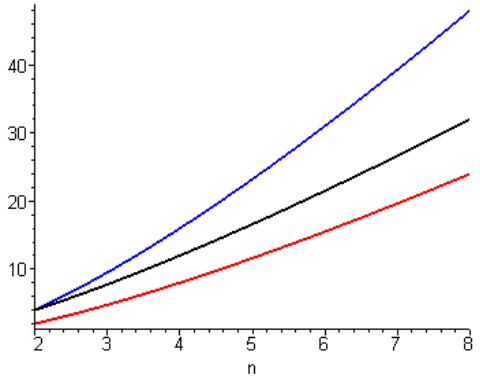

Figura 4.22 Cotas Ejemplo 4.14

4.22. Cálculos de complejidad a partir del código.

Los cálculos de complejidad pueden efectuarse, independientemente de la función que el algoritmo realiza. Se analiza un ejemplo, que muestra los diferentes costos basándose en criterios de cuenta de instrucciones.

Si bien no es necesario conocer la función realizada por el algoritmo para efectuar el cálculo de la complejidad temporal, la siguiente función implementa un algoritmo de ordenamiento conocido como burbuja. Opera sobre un arreglo de enteros de n posiciones.

```
void Alg1(int a[], int n)
{
  int i, j, temp;
  for (i=0; i<n-1; i++)              //1
     for (j=n-1; j>=i+1; j--)        //2
       if(a[j-1] > a[j])             //3
         { temp=a[j-1];              //4
           a[j-1]=a[j];              //5
           a[j]=temp;                //6
         }
}
```

4.22.1. Cálculo de complejidad basada en operaciones elementales.

Se calculan las operaciones elementales asociadas a cada línea considerando como operación elemental, a: comparaciones, asignaciones, sumas, restas, y acceso a componentes de un vector.
Línea 1: se ejecuta una asignación de inicio; una resta y una comparación de salida; una resta, una comparación y una suma por cada una de las iteraciones del lazo.
Línea 2: se ejecutan una resta y asignación de inicio y una suma más una comparación de salida; una suma y una comparación más una resta por cada una de las iteraciones.

Línea 3: se efectúa la condición, con 4 O(1): una diferencia, dos accesos a un vector, y una comparación.
Las líneas 4 a 6, es una acción compuesta que sólo se ejecuta si se cumple la condición de la línea 3, y se realiza un total de 9 O(1): 3, 4 y 2 O(1) respectivamente.

En el peor caso se efectúa siempre el bloque asociado al if. Tenemos entonces que el lazo interno se realiza:

$$T_1(n) = 2 \cdot O(1) + \sum_{j=i+1}^{n-1} (2+4+9+1) \cdot O(1) + 2 \cdot O(1)$$

La sumatoria se realiza: el número final menos el inicial, más uno:

$$(n-1)-(i+1)+1 = (n-i-1)$$

Entonces, en peor caso, el lazo interno tiene un costo:

$$T_1(n) = (4+16 \cdot (n-i-1)) \cdot O(1)$$

El lazo externo, tiene un costo de:

$$T(n) = 1 \cdot O(1) + \sum_{i=0}^{n-2} (2 + (4+16 \cdot (n-i-1)) + 1) \cdot O(1) + 2 \cdot O(1)$$

Arreglando, y considerando que los términos que no dependen de i, se suman *(n-1)* veces, se obtiene:

$$T(n) = \left(3 + (7+16n-16)(n-1) - 16 \sum_{i=0}^{n-2} i \right) \cdot O(1)$$

La suma de los primeros *(n-2)* términos, puede plantearse:

$$\sum_{i=0}^{n-2} i = \sum_{i=1}^{n-2} i = \frac{(n-2)(n-1)}{2}$$

Reemplazando en la ecuación anterior, se obtiene:

$$T(n) = (16n^2 - 25n + 12 - 8(n-2)(n-1))O(1)$$
$$T(n) = (8n^2 - n - 4)O(1)$$

Finalmente:

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

4.22.2. Cálculo de complejidad basada en instrucciones del lenguaje de alto nivel.

Cada instrucción del lenguaje se considera de costo unitario.

Se considera que las líneas 4, 5 y 6 son O(1), la acción compuesta tiene costo: O(1)+O(1)+O(1), es decir 3O(1).
El peor caso es que se realice siempre el bloque del if, y si se considera que éste es de costo O(1), entonces, las líneas 3, 4, 5 y 6 son 4O(1).

El for interno tiene un costo: $(n-i-1)\cdot 4\cdot O(1)$

Para el for externo, se tiene:

$$T(n)=\sum_{i=0}^{n-2}4(n-i-1)\cdot O(1)=\left(4(n-1)(n-2)-4\frac{(n-2)(n-1)}{2}\right)O(1)$$

Resultando:

$$T(n)=(2n^2-6n+4)\cdot O(1)=O(n^2)$$

Debe notarse que el costo unitario, del cálculo basado en contar instrucciones de alto nivel, es diferente del realizado contando operaciones elementales.

Puede comprobarse que los costos, de los dos procedimientos de cálculo, difieren en una constante:

$$R(n)=\frac{(8n^2-n-4)}{(2n^2-6n+4)}$$

Aplicando la regla de L'Hôpital, para calcular el límite, se tiene:

$$\lim_{n\to\infty}R(n)=4$$

La gráfica siguiente ilustra la relación anterior:

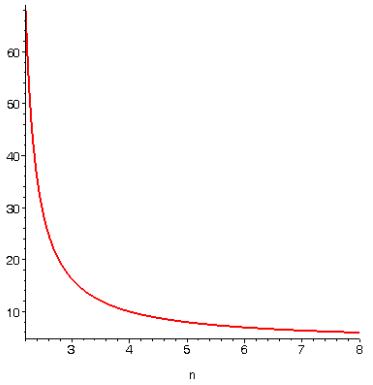

Figura 4.23. Razón entre complejidades

Se obtiene, mediante:

```
> e:=(8*n^2 -n-4)/(2*n^2-6*n+4);
```

$$e := \frac{8\,n^2 - n - 4}{2\,n^2 - 6\,n + 4}$$

```
> limit(e,n=infinity);
```

$$4$$

```
> plot(e,n=2.2..8,thickness=2);
```

4.22.3. Cálculo de complejidad basada en instrucciones del lenguaje de máquina.

La compilación de la función en C, genera el listado de las instrucciones en assembler, para el microcontrolador MSP430; el que se muestra más adelante.

Una aproximación para efectuar la cuenta es considerar que todas las instrucciones tienen igual costo.

Al listado se le han agregado comentarios para hacerlo más legible, y del listado pueden obtenerse las siguientes cuentas:

Antes de realizar i=0, se han ejecutado 9 instrucciones assembler, cuestión que no se contempla en los cálculos anteriores; tampoco se contemplan las 6 instrucciones que se realizan para salir de la función; corresponden a armar y desarmar el frame de la función. También aparecen las tres instrucciones necesarias para pasar los argumentos e invocar a la función.

El desarrollo del if resulta con 11 instrucciones. La acción compuesta dentro del if, se logra con 21 instrucciones. Reinicio de los for con 3 instrucciones; evaluación de las condiciones de los for en 4 instrucciones. Inicio del primer for con una instrucción, del segundo for con 3.

Entonces el for interno, tiene un costo:

$$T_1(n) = 3 \cdot O(1) + \sum_{j=i+1}^{n-1} (4+11+21+3) \cdot O(1) + 4 \cdot O(1)$$

Arreglando, y realizando la sumatoria:

$$T_1(n) = 7 \cdot O(1) + 39 \sum_{j=i+1}^{n-1} O(1) = (7 + 39(n-i-1)) \cdot O(1) = (39n - 32 - 39i) \cdot O(1)$$

El primer for, tiene costo:

$$T_2(n) = 9 \cdot O(1) + 1 \cdot O(1) + \sum_{i=0}^{n-2} (4 + (39n - 32 - 39i) + 3) \cdot O(1) + 4 \cdot O(1) + 6 \cdot O(1)$$

Simplificando:

$$T_2(n) = 20 \cdot O(1) + \sum_{i=0}^{n-2} (39n - 25 - 39i) \cdot O(1)$$

Extrayendo el término que no depende de *i* de la sumatoria:

$$T_2(n) = 20 \cdot O(1) + (39n - 25)(n-1) \cdot O(1) - 39 \sum_{i=0}^{n-2} i \cdot O(1)$$

Realizando la sumatoria:

$$T_2(n) = (39n^2 - 64n + 45) \cdot O(1) - 39 \frac{(n-2)(n-1)}{2} O(1)$$

Simplificando, se obtiene:

$$T_2(n) = \frac{39n^2 - 11n + 12}{2} \cdot O(1)$$

Existen tres adicionales, requeridas para pasarle valores a los argumentos e invocar a la función:

$$T(n) = \frac{(39n^2 - 11n + 18)}{2} \cdot O(1)$$

```
 Alg1(a, N);
 0025EE   3E400500     mov.w   #0x5, R14          ;copia valor constante N en R14
 0025F2   3C400011     mov.w   #0x1100,R12        ;copia valor de la dirección de a en R12
 0025F6   B0126E25     call    #Alg1              ;Invocación de la función

void Alg1(int *a, int n)
{
Alg1:
 00256E   0A12         push.w  R10                ;formación del frame
 002570   0B12         push.w  R11                ;salva registros
 002572   0812         push.w  R8
 002574   0912         push.w  R9
 002576   0D4C         mov.w   R12,R13            ;argumento a en R12, copia en R13
 002578   0C4E         mov.w   R14,R12            ;argumento n en R14, copia en R12
  for (i=0; i<n-1; i++)        //1
 00257A   0A43         clr.w   R10                ;inicio de i
 00257C   293C         jmp     0x25D0             ;salto a condición de primer for
     if(a[j-1] > a[j])    //3
 00257E   0F4B         mov.w   R11,R15            ;en R11 almacena j
 002580   0F5F         rla.w   R15                ;cada entero ocupa 2 bytes. j*2
 002582   0E4D         mov.w   R13,R14            ;a pasa a R14
 002584   0E5F         add.w   R15,R14            ;en R14 apunta a a[j]
 002586   084B         mov.w   R11,R8             ;mueve j a R8
 002588   3853         add.w   #0xFFFF,R8         ;R8=j-1
 00258A   0858         rla.w   R8                 ;(j-1)*2 en bytes
 00258C   0F4D         mov.w   R13,R15
 00258E   0F58         add.w   R8,R15             ;en R15 forma a[j-1]
```

```
002590   AE9F0000 cmp.w   @R15,0x0(R14)        ;compara
002594   1734     jge     0x25C4               ;salta a fin del if
      { temp=a[j-1];    //4
002596   0F4B     mov.w   R11,R15
002598   3F53     add.w   #0xFFFF,R15
00259A   0F5F     rla.w   R15                  ;R15=2(j-1)
00259C   0E4D     mov.w   R13,R14
00259E   0E5F     add.w   R15,R14              ;R14 apunta a a[j-1]
0025A0   284E     mov.w   @R14,R8              ;temp =R8=*(a+2(j-1))
       a[j-1]=a[j];    //5
0025A2   0F4B     mov.w   R11,R15
0025A4   0F5F     rla.w   R15
0025A6   0E4D     mov.w   R13,R14
0025A8   0E5F     add.w   R15,R14              ;R14 apunta a a[j]
0025AA   094B     mov.w   R11,R9
0025AC   3953     add.w   #0xFFFF,R9
0025AE   0959     rla.w   R9
0025B0   0F4D     mov.w   R13,R15
0025B2   0F59     add.w   R9,R15               ;R15 apunta a a[j-1]
0025B4   AF4E0000 mov.w   @R14,0x0(R15)       ;asignación
       a[j]=temp;      //6
0025B8   0E4B     mov.w   R11,R14
0025BA   0E5E     rla.w   R14
0025BC   0F4D     mov.w   R13,R15
0025BE   0F5E     add.w   R14,R15
0025C0   8F480000 mov.w   R8,0x0(R15)
   for (j=n-1; j>=i+1; j--)  //2
0025C4   3B53     add.w   #0xFFFF,R11          ;reinicio segundo for
   for (j=n-1; j>=i+1; j--)  //2
0025C6   0F4A     mov.w   R10,R15              ;mueve j al registro R15
                                               ;evalúa condición segundo for
0025C8   1F53     inc.w   R15                  ; incrementa i
0025CA   0B9F     cmp.w   R15,R11              ;compara j (en R10) con i+1 (en R15)
0025CC   D837     jge     0x257E               ;entra al bloque del segundo for
 for (i=0;i<n-1; i++)      //1                 ;sale del segundo for
0025CE   1A53     inc.w   R10                  ;reinicio primer for
 for (i=0; i<n-1; i++)     //1
0025D0   0F4C     mov.w   R12,R15              ;trae operando n
0025D2   3F53     add.w   #0xFFFF,R15          ;resta 1 a n
0025D4   0A9F     cmp.w   R15,R10              ;comparación
0025D6   0334     jge     0x25DE               ;salida del primer for
   for (j=n-1; j>=i+1; j--)  //2
0025D8   0B4C     mov.w   R12,R11              ;inicia j
0025DA   3B53     add.w   #0xFFFF,R11          ;resta 1 a j
0025DC   F43F     jmp     0x25C6               ;test de la condición del segundo for
}
0025DE   30407046     br      #0x4670          ;termina Alg1

004670   3941     pop.w   R9                   ;desarma frame
```

```
004672  3841    pop.w   R8              ;restaura registros
004674  3B41    pop.w   R11
004676  3A41    pop.w   R10
004678  3041    ret                     ;retorno de Alg1
```

Un cálculo más exacto consiste en contar los ciclos de reloj necesarios para ejecutar las instrucciones. Esta información se obtiene en el manual del procesador en que se esté compilando. Esto debido a que, dependiendo de la estructura del procesador, algunas instrucciones demoran más que otras. Por ejemplo, la instrucción: mov.w @R14,0x0(R15) demora más en su ejecución que las instrucciones ret o pop.

Otros aspectos que muestra el código assembler, es la aritmética de punteros y las instrucciones relativas a registros para indireccionar; además del uso de registros y saltos.

El conteo de las instrucciones dependerá de la calidad del compilador que se esté empleando, y si dispone o no de niveles de optimización. También dependerá del procesador y su repertorio de instrucciones, y del sistema operativo que se esté empleando. Por estas razones, en cálculos de complejidad, no suelen contarse las instrucciones de un procesador, sino las del lenguaje de alto nivel o del pseudo código.

El orden de complejidad, que resulta de contar las instrucciones assembler, también es $\Theta(n^2)$.

Puede compararse el costo obtenido mediante contar instrucciones assembler, con la cuenta de operaciones elementales, mediante:

$$R(n) = \frac{(39n^2 - 11n + 18)}{\frac{2}{(8n^2 - n - 4)}}$$

$$\lim_{n\to\infty} R(n) = 2$$

Se observa que $R(n)$ tiende rápidamente a la constante dos.

```
> e2:=2*(39*n^2-11*n+18)/(8*n^2 -n-4);
```

$$e2 := \frac{2\,(39\,n^2 - 11\,n + 18)}{8\,n^2 - n - 4}$$

```
> limit(e2,n=infinity);
```

$$\frac{39}{4}$$

```
> plot(e2,n=1..6,thickness=2);
```

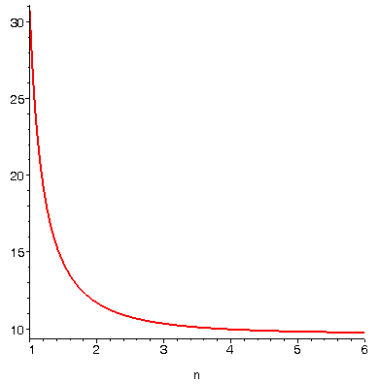

Figura 4.24 Razón constante entre complejidades.

Se concluye de este ejemplo, que una metodología razonablemente útil para calcular la complejidad de un algoritmo es contar las instrucciones del lenguaje de alto nivel o del pseudo código.

4.23. Resumen.

Se puede cuantificar el orden de crecimiento empleando la notación Θ.

El orden de crecimiento es el principal factor que determina el tiempo de ejecución de un algoritmo.

Si se reduce el orden de crecimiento se logran enormes reducciones en el tiempo de ejecución.

Problemas resueltos.

P4.1

Determinar la solución de la siguiente relación de recurrencia:
T(n) = T(n/2) + n, con T(1)= 2.

Si es necesario puede emplear que la suma de la siguiente progresión geométrica es:

$$\sum_{i=1}^{n} 2^i = 2^{(n+1)} - 2$$

¿Cuál es el orden de complejidad?

Solución.
T(2) = T(1) + 2 = 4 = 2^2
T(4) = T(2) + 4 = 8 = 2^3
T(8) = T(4) + 8 = 16 = 2^4
T(16) = T(8) + 16 = 32 = 2^5

$T(2^i) = T(2^{i-1}) + 2^i = 2^{i+1} = 2*2^i$
Con $2^i = n$, se obtiene la solución:

T(n) = 2*n
Entonces: T(n) es Θ(n), para todo n.

Ya que existen c1, c2 y n_0 tales que:

$$c1\ f(n) <= T(n) <= c2\ f(n) \text{ con } n>=n_0$$

Con c1=1, c2=3 y n_0 =0.

$$1n <= 2n <= 3n \text{ con } n>=0$$

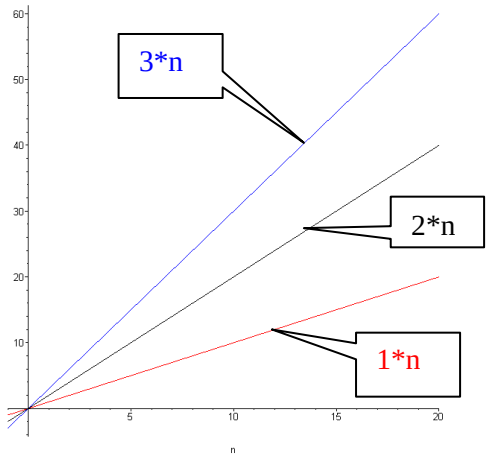

Figura P4.1 Cotas de T(n).

Ejercicios propuestos.

E4.1.

Dado un número n, encontrar a y b tales que: a*a+b*b = n*n, con a, b y n enteros mayores que cero.
Determinar la complejidad temporal y su orden de crecimiento.

E4.2.

Calcular la complejidad de: T(n) = T(n-1) + n con T(0)=0.

E4.3.

Calcular la complejidad:

$$T(n) = 3T(n-1) + 4T(n-2) \quad \text{si } n>1;\ T(0) = 0;\ T(1) = 1.$$

Sol. $T(n)=O(4_n)$

E4.4.

Calcular la complejidad:

$$T(n) = 2T(n-1) - (n+5)3^n \quad \text{si } n>0;\ T(0) = 0.$$

Sol. $T(n) = 9\cdot 2^n - 9\cdot 3^n - 3n3^n$ con orden de complejidad: $\Theta(n3^n)$.

E4.5.

Calcular la complejidad:

$$T(n) = 4T(n/2) + n^2 \quad \text{si } n>4,\ n \text{ potencia de 2};\ T(1) = 1;\ T(2) = 8.$$

Sol. $T(n) = n^2 + n^2 \log n$ $\Theta(n^2 \log n)$.

Índice general.

Índice de figuras.