METODOLOGIA DE PROGRAMACIÓN

PRINCIPIOS Y APLICACIONES

Título: Metodología de programación. Principios y aplicaciones

Autor: © Alejandro Rabasa Dolado
Laureano Santamaría Arana

I.S.B.N.: 84-8454-323-4
Depósito legal: A-984-2004

Edita: Editorial Club Universitario Telf.: 96 567 38 45
C/. Cottolengo, 25 - San Vicente (Alicante)
www.ecu.fm

Printed in Spain
Imprime: Imprenta Gamma Telf.: 965 67 19 87
C/. Cottolengo, 25 - San Vicente (Alicante)
www.gamma.fm
gamma@gamma.fm

Reservados todos los derechos. Ni la totalidad ni parte de este libro puede reproducirse o transmitirse por ningún procedimiento electrónico o mecánico, incluyendo fotocopia, grabación magnética o cualquier almacenamiento de información o sistema de reproducción, sin permiso previo y por escrito de los titulares del Copyright

INDICE

Prólogo 5

Tema 1: Cálculo de complejidades 7
1.1.- Introducción al análisis de la eficiencia de los algoritmos 8
1.2.- Noción de complejidad: Notación asintótica y órdenes 11
1.3.- Cotas de complejidad 14
1.4.- Cálculo de complejidades de algoritmos por instrucciones 16
1.5.- Ejercicios de cálculo de complejidad 17

Tema 2: Principios y tipos de recursividad 25
2.1.- Introducción a los algoritmos recursivos 26
2.2.- Etapas de Análisis y Composición 29
2.3.- Etapa de Verificación 30
2.4.- Tipos de recursividad 34
2.4.1.- Recursividad Anidada 34
2.4.2.- Recursividad Múltiple o en Cascada 35
2.4.3.- Recursividad Lineal 35
2.5.- Ejercicios de recursividad 37

Tema 3: Esquemas de transformación recursivo-iterativo 43
3.1.- Transformación a iterativo de funciones Recursivas Lineales Finales 44
3.2.- Transformación a iterativo de funciones Recursivas Lineales No-Finales 45
3.2.1.- Método de inversión funcional 45
3.2.2.- Método de inversión funcional mediante pila 48
3.2.3.- Método del reparentizado 52

Tema 4: Divide y Vencerás 59
4.1.- Introducción y esquema de "Divide y Vencerás" 60
4.2.- Ejemplo explicativo de "Divide y Vencerás": El producto a trozos. 61
4.3.- Ejercicios de "Divide y Vencerás" 64
4.4.- Cuándo aplicar "Divide y Vencerás" 68

Tema 5: Algoritmos Voraces 69
5.1.- Introducción y esquema de los Algoritmos Voraces 70
5.2.- Ejemplo explicativo de Algoritmo Voraz: La devolución del cambio. 71
5.3.- Ejercicios de Algoritmos Voraces 73

5.4.- Cuándo aplicar Algoritmos Voraces 87

Tema 6: Backtracking 89
6.1.- Introducción y esquema de Backtracking 90
6.2.- Ejemplo explicativo de Backtracking: Viajar en coche 95
6.3.- Ejercicios de Backtracking 97
6.4.- Cuándo aplicar Backtracking 104

Tema 7: Algo más sobre los esquemas de programación 105
7.1.- Otros esquemas: Ramificación y Poda. Programación Dinámica 106
7.1.1.- Ramificación y Poda 106
7.1.2.- Programación dinámica 113
7.2.- ¿Por qué usar esquemas de programación? 117
7.3.- Comparativa de los esquemas de programación 118

Anexo: Relación de ejercicios 121

Bibliografía 125

Prólogo

Este libro va dirigido a aquellas personas que se inician en el apasionante mundo de la programación. Todos los que nos enfrentamos a la tarea de resolver, programando, un determinado problema, nos hacemos con frecuencia las mismas preguntas: “¿Por dónde empiezo?”, “¿Hice una vez algo parecido?”. La verdad es que cada problema es un mundo, y en la mayoría de los casos, admite muy distintos planteamientos. Pero también es verdad que muchos problemas se resuelven bajo unas estructuras parecidas.

La mente humana tiende a buscar (y encontrar) semejanzas entre hechos nuevos y hechos ya conocidos. De la misma manera, un programador agradecerá tener asimilados una serie de planteamientos típicos de problemas que se repiten con cierta frecuencia, y así, ante un problema nuevo, poder compararlo con aquellos que en su día aprendió a resolver.

Bajo nuestro punto de vista, la Metodología de Programación no es sino el estudio de esas estructuras algorítmicas: bajo qué circunstancias se dan, a qué problemas se aplican y qué ventajas y dificultades se derivan de su uso.

El libro que tiene entre sus manos pretende ser breve y conciso. En él abordamos de manera directa, los esquemas de programación más habituales y los empleamos para resolver problemas clásicos de programación, que el lector podrá encontrar en cualquiera de las obras de la extensa bibliografía existente sobre la materia. Aquí trataremos de enunciarlos y explicarlos incluyendo trazas y variantes que los adaptan a los niveles que consideramos básicos, para alguien

que se está iniciando en la algoritmia. Haremos uso de un pseudocódigo muy parecido al lenguaje ANSI-C para transcripción de la mayor parte de los algoritmos.

Esperamos que nuestro trabajo pueda ser de utilidad: que este ejemplar termine lleno de anotaciones y referencias, y que dentro de algún tiempo, cuando el lector se enfrente a un problema de programación, pueda decir: "…venía algo parecido en aquel libro…".

Los autores.-

Tema 1: Cálculo de complejidades

En este tema se va a hacer una introducción al cálculo de complejidades. Para ello será necesario definir conceptos como la eficiencia espacial y temporal de los algoritmos, la notación asintótica y las cotas de complejidad. Se verá una serie de reglas y propiedades que guían el cálculo de la complejidad, y se propondrán ejercicios de ejemplo para facilitar la comprensión de los contenidos expuestos.

INDICE DEL TEMA

1.1.- Introducción al análisis de la eficiencia de los algoritmos
1.2.- Noción de complejidad: Notación asintótica y órdenes
1.3.- Cotas de complejidad
1.4.- Cálculo de complejidades de algoritmos por instrucciones
1.5.- Ejercicios de cálculo de complejidad

1.1.- Introducción al análisis de la eficiencia de los algoritmos

La eficiencia de los algoritmos puede ser contemplada desde diferentes puntos de vista y entre los más frecuentes encontramos como sinónimos de código eficiente los siguientes:

- La claridad y simplicidad del algoritmo. Con frecuencia, este es un aspecto que se valora mucho, pues un código claro y simple, hace fácil la fase de mantenimiento de los programas; pero no debemos confundirnos, la eficiencia en la fase de mantenimiento no es la eficiencia en sí del algoritmo.
- El mejor uso de los recursos ante unas mismas condiciones del problema. Por ejemplo el uso de la memoria, teniendo en cuenta que éste es un recurso típicamente crítico en el diseño de software. Este enfoque nos conduciría al concepto de eficiencia espacial.
- Menor tiempo de ejecución ante similares condiciones de partida. Es decir: diremos que un algoritmo A es más eficiente que otro B si ante unos mismos parámetros de entrada, A tarda menos tiempo en terminar que B. Este enfoque de la eficiencia nos conduce al concepto de eficiencia temporal.

A lo largo del tema, al hablar de código eficiente, nos estaremos refiriendo a la eficiencia temporal, la cual va a depender, principalmente, de dos factores:

1) Los casos de entrada al programa: Cantidad, tamaño y contenido de los parámetros de entrada. No se tarda lo mismo en sumar las componentes de un vector de 100 componentes que las de uno de 10. Ni puede tardarse lo mismo en sumar las componentes de un vector de 100 componentes enteras, que si estas 100 componentes fueran números reales de muy alta precisión.

2) El computador sobre el que se ejecuta el programa y el compilador con que se ha generado el ejecutable. Estos factores solemos entenderlos como no-dependientes del programador, puesto que no siempre está en sus manos poder elegir unos u otros. Además, no es el programa lo que es más o menos eficiente, sino el algoritmo en sí.

Por tanto, el diseño del algoritmo por el que nos decantemos será el factor decisivo de cara a obtener una mayor eficiencia temporal. Por eso deberemos

evitar iteraciones innecesarias, asignaciones redundantes, comprobaciones de condiciones lógicas mal planteadas; y en general todo aquello que pueda introducir retardos en la futura ejecución de nuestro algoritmo, independientemente de la máquina sobre la que vaya a correr.

También empezamos a vislumbrar la necesidad de calcular cuánto va a tardar en terminar nuestro algoritmo, pero no tan sólo en términos absolutos (12 ó 15 segundos), sino que además debemos ser capaces de expresar dicho tiempo en función de los casos de entrada.

Veamos como ejemplo de cálculo de tiempos por instrucciones la resolución del Ejercicio 1.1.

Ejercicio 1.1

Cálculo de tiempo de ejecución de ordenación de un vector por el método de selección cuyo algoritmo en pseudocódigo se muestra a continuación:
Para calcular el tiempo T, de ejecución de este algoritmo definimos las constantes:
ta: tiempo de asignación de enteros
tc: tiempo de comparación de enteros
ti: tiempo de incrementar un entero
te: tiempo de acceso a un elemento, del vector.
Notar que con la definición de dichas constantes, hacemos que el cálculo del tiempo sea independiente de la máquina sobre la que se ejecuta.

T=(1)+(n-1)((2)+(3)+(4mc|4pc)+(5))

```
(1)para (i=1;i<=n-1;i++)
{
(2)    pmin=i; /*guarda la pos. Del minimo*/
(3)    para (j=i+1;j<=n;j++)
       {
(4)          si  (a[j]<a[pmin]) pmin=j;
       }
(5)    intercambiar(a[i],a[pmin])
}
```

Es decir, el tiempo del algoritmo, T, es lo que se tarda el la ejecución de la línea (1), más lo que tarda la línea (2), la (3), la (4) y la (5), que se ejecutan todas ellas (n-1) veces (por estar dentro de un bucle *for* que se ejecutará (n-1) veces).
Además la línea (4) tiene un mejor caso, 4mc (si no se cumple la condición, no se hace nada) y un peor caso, 4pc (si se cumple la condición, hace una asignación).

Veamos ahora, cuál es el tiempo de ejecución de cada línea en función de las constantes definidas previamente.
(1): ta + (n-1)*ti + n*tc.
(n-1): porque va desde 1 hasta n-1. Se da una comparación más que incrementos
(2): (n-1 veces) ta.
(3):para cada i (n-1 veces): ta+(n-i)*ti+(n-i+1)*tc
(n-i): porque va desde i+1 hasta n.
(4): mejor caso: (mc) (n-1 veces): (n-i)(2*te+tc). FALSE: sólo compara
peor caso: (pc) (n-1 veces)**:** (n-i)(2*te+tc) + (n-i)*ta. TRUE: Compara y asigna
(5): (n-1 veces): (2*te+3*ta)

¿Cómo afectan los diferentes factores al cálculo del tiempo de ejecución T?
- El tamaño de los datos de entrada (Valor de "n")
- El contenido de los datos de entrada: Tmin y Tmax. Nos centraremos en peor caso (Tmax)
- El código generado por el compilador y el ordenador empleado. Sólo afecta a los "t" (ctes)
El factor determinante, para problemas "suficientemente" grandes, es el tamaño de los datos de entrada.

Desarrollando la ecuación, T=(1)+(n-1)((2)+(3)+(4mc|**4pc**)+(5))
y fijándonos en "n": saldría un polinomio de orden cuadrático.
$T \in O(n^2)$

Generalizando, podemos concluir que el tiempo de ejecución de un programa está en función del tamaño (y contenido) de los datos de entrada: T=f(n).

Fin Ejercicio 1.1

1.2.- Noción de complejidad: Notación asintótica y órdenes

Recurrimos al término **complejidad** f(n), para expresar **la eficiencia temporal** (a veces, espacial) de un algoritmo **en función del tamaño n de sus datos** de entrada; y cómo varía su tiempo de ejecución a medida que varía también el tamaño mencionado n de la entrada, T=f(n).

Calcular el tiempo en función de n, y no hacerlo en términos absolutos (p.e. 12 segundos) nos permite evaluar la complejidad (temporal) de un algoritmo independientemente de la máquina dónde se ejecute.

Introducimos la **notación asintótica** para contemplar el comportamiento de la función f(n) **en sus límites**, es decir, **para valores de n suficientemente grandes**, de manera que podamos hacer simplificaciones, por ejemplo obviar valores constantes tales como tiempos de llamada a una función si ésta se produce "pocas" veces, es decir, si el número de llamadas no está en función de n.

Sea una función $f(n) \in f : N \rightarrow R^{\geq 0}$

La función *t(n)* es del orden de *f(n)*, $t(n) \in O(f(n))$

Si se cumple $\exists c \geq 0 / t(n) \leq cf(n)$, para valores de n, a partir de un cierto umbral, $n >= n_0$

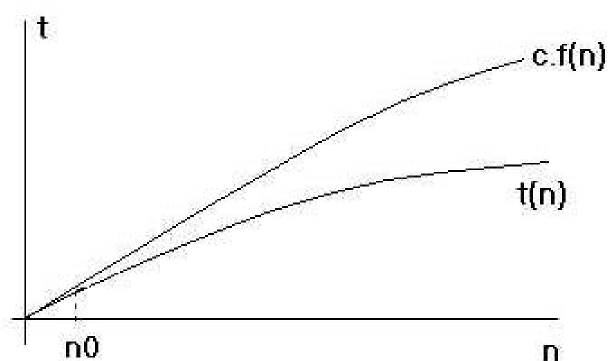


Fig.1.1: t(n)∈O(f(n))

Los órdenes de complejidad son conjuntos de funciones (espacios).

A continuación se citan algunas **propiedades** que serán útiles para el cálculo de complejidades, según describe R. Peña en *Diseño de Programas. Formalismo y Abstracción*:

1.- $f(n) \in O(g(n)),\ g(n) \in O(h(n)),\ \Rightarrow\ f(n) \in O(h(n))$

2.- $O(f(n)) = O(g(n)) \Leftrightarrow f(n) \in O(g(n)),\ g(n) \in O(f(n))$

3.- $O(f(n)) \subset O(g(n)) \Leftrightarrow f(n) \in O(g(n)),\ g(n) \notin O(f(n))$

4.- Regla de la suma (para secuencias de instrucciones)

$O(f(n)+g(n)) = O(max(f(n), g(n)))$

5.- Regla del producto (para bucles)

$O(f(n)) \bullet O(g(n)) = O(f(n) \bullet g(n))$

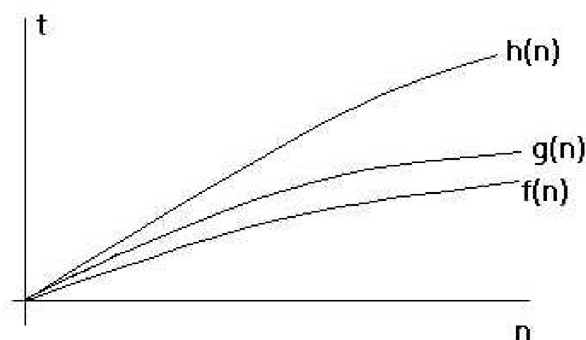


Fig. 1.2: Tres funciones sobre las que calcular órdenes de complejidad

Ésta es la **jerarquía** de los órdenes de complejidad:

$$O(1) \subset O(\log \ n) \subset O(n) \subset O(n^2) \subset ... \subset O(n^a) \subset ... \subset O(2^n) \subset O(n!)$$

En la tabla y gráficas siguientes (Fig. 1.3, Fig. 1.4 y Fig. 1.5) se muestran valores y representaciones gráficas de las funciones más significativas para facilitar la comprensión de las relaciones entre los distintos órdenes.

Notar cómo a partir de n=10 la función $t=2^n$ está siempre por encima de $t=n^a$.

Notar también cómo se confirman el resto de órdenes detallados en la jerarquía citada anteriormente.

n	**$t=2^n$** t=2 exp n	**$t=n^a$** t=n exp a (a=3)	**$t=n^2$** t=n exp 2	**t=n** t=n	**$t=lg_{10}$ n** t=lg n
1	2	1	1	1	0
2	4	8	4	2	0,30103
3	8	27	9	3	0,47712125
4	16	64	16	4	0,60205999
5	32	125	25	5	0,69897
6	64	216	36	6	0,77815125
7	128	343	49	7	0,84509804
8	256	512	64	8	0,90308999
9	512	729	81	9	0,95424251
10	1024	1000	100	10	1
11	2048	1331	121	11	1,04139269
12	4096	1728	144	12	1,07918125
13	8192	2197	169	13	1,11394335
14	16384	2744	196	14	1,14612804
15	32768	3375	225	15	1,17609126
16	65536	4096	256	16	1,20411998
17	131072	4913	289	17	1,23044892
18	262144	5832	324	18	1,25527251
19	524288	6859	361	19	1,2787536
20	1048576	8000	400	20	1,30103

Fig. 1.3: Tabla de valores para distintas funciones

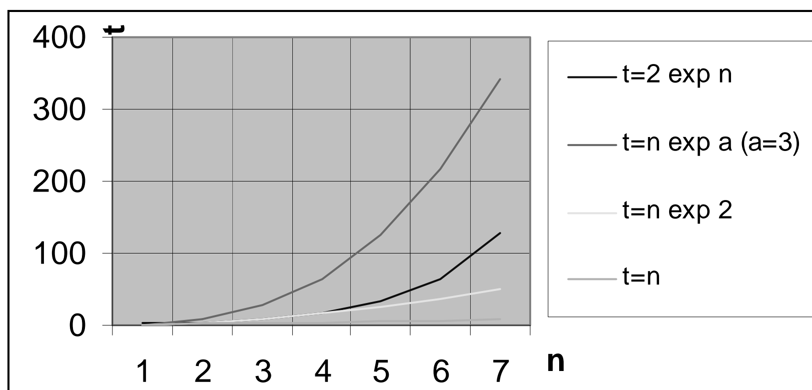


Fig. 1.4: Representación gráfica de las funciones $t=2^n$; $t=n^a$; $t=n^2$ y t=n

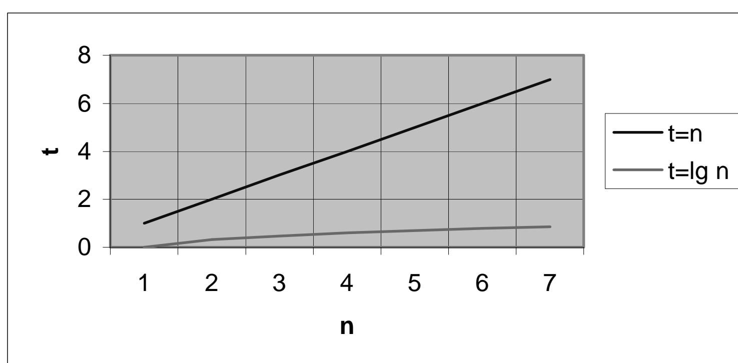


Fig. 1.5: Representación de las funciones t=n y $t=lg_{10}$ n

1.3.- Cotas de complejidad

En este apartado vamos a introducir los conceptos de cota superior, cota inferior y orden exacto.

O(f(n)) es la **cota superior** del tiempo de ejecución T(n):

$$f : N \to R^{\geq 0}$$

$$O(f(n)) = \{g : N \to R^{\geq 0} \mid \exists c \in R^{>0}, n_0 \in N. \forall n \geq n_0 . g(n) \leq cf(n)\}$$

Es decir, el conjunto de funciones que están por debajo de f(n). Así, O(f(n)) acota superiormente el problema.

Ω(f(n)) es la **cota inferior** del tiempo de ejecución T(n):

$$f : N \rightarrow R^{\geq 0}$$
$$\Omega(f(n)) = \{ g : N \rightarrow R^{\geq 0} \mid \exists c \in R^{>0}, n_0 \in N. \forall n \geq n_0 . g(n) \geq cf(n) \}$$

Es decir, el conjunto de funciones que están por arriba de f(n). Así, Ω(f(n)) acota inferiormente el problema.

O(f(n)) y Ω(f(n)) no dependen de que el análisis realizado sea para el caso peor, promedio o mejor. Son simplemente conjuntos de funciones que siempre están por encima o por debajo de T(n). (tiempo de ejecución del algoritmo en función de n).

Si evaluamos el algoritmo en el **peor caso** (que es donde debemos centrarnos siempre que calculemos una complejidad):

O(f(n)) es la cota superior del tiempo de ejecución.

Ω(f(n)) es la cota inferior del tiempo de ejecución. (Lo cual no significa que sea el tiempo de ejecución en el mejor caso).

Definiremos el **orden exacto** de f(n): Θ(f(n))

$$\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$$

Es decir, el orden exacto de f(n) es el conjunto de funciones que son cota superior y también son cota inferior de f(n).

Si una función g acota superior e inferiormente al problema, diremos que g $\in$ Θ(f(n))

1.4.- Cálculo de complejidades de algoritmos por instrucciones

Dada la definición de orden exacto, ya podemos centrarnos sobre él para enunciar una serie de reglas basadas en las propiedades genéricas que se vieron en el apartado 1.2.

Regla 1: Asignación: $\Theta(1)$ Valor constante. (Si no depende de n)

Regla 2: Secuencia (P1,P2...Pm):

$$\Theta(f_1(n)) + \Theta(f_2(n)) + ... + \Theta(f_m(n)) = \Theta(max(f_1(n), f_2(n),..., f_m(n)))$$

Regla 3: Condicional (Si B entonces S1, sino S2 fin_Si)

$$O(max(f_B(n), f_1(n), f_2(n)))$$

Regla 4: Iterativa *mientras* (mientras B hacer S fin_mientras)

$$O(f_{B,S}(n) \bullet f_{it}(n)) = ... = \Theta(n^2) .$$

Siendo los costes respectivos de evaluar B y ejecutar S

$f_{B,S}\, f_{it}$ los costes respectivos de evaluar B y ejecutar S y del total de iteraciones en función de n.

La iterativa *repetir* se calcula de manera análoga, con la ventaja de que está mucho más definida y no tenemos que suponer $O(f_{it}) \in O(n)$, pues es cierto en cualquier caso.

Notar la diferencia con la iterativa *mientras* en la que no se sabe a priori cuántas veces se ejecutará el bucle, y además hay que analizar si dicho número de veces puede expresarse o no en función de n.

Regla 5: Llamadas encadenadas a procedimientos.

Mientras no se trate de llamadas recursivas, deberemos calcular siguiendo el criterio “de dentro hacia fuera”, es decir calculando primero la complejidad de los procedimientos más internos (que no llaman a otro) y concatenándolas

debidamente con las complejidades de los procedimientos que los llaman. Así sucesivamente hasta la llamada principal.
Cuando hay recursividad, T(n) se suele expresar en función de T(m). Siendo m<n.

1.5.- Ejercicios de cálculo de complejidad

Ejercicio 1.2

Ordenación de un vector por el método de inserción.
El algoritmo de dicho método es:

```
Insercion (T[1..n]:entero)
 para (i=2; i<=n; i++)
 { x=T[i];
   j=i-1;
   mientras (j>0 Y x<T[j])
   { T[j+1]=T[j]; j-- }
   T[j+1]=x
 }
```

Centrándonos en el peor caso en el análisis de la sentencia *mientras*: Siempre x<T[j] y j va disminuyendo de uno en uno desde i-1 hasta 0, i veces.
Como mucho se repetirá para cada valor de j, entre 0 e (i-1) –es decir: i veces-. Ocurriendo que i toma valores entre 2 y n, la comprobación se efectúa:

$\Sigma_{i=2}^{n} i = \frac{n(n+1)}{2} - 1$ veces. Esta es la fórmula de la suma de la serie: 1+2+3+4+...+n quitándole el primer elemento, pues i empieza tomando valores en 2.

Dicha expresión desarrollada nos conduce a un polinomio de segundo grado y por tanto $\in \Theta(n^2)$

Fin Ejercicio 1.2

Ejercicio 1.3

Obtención del elemento máximo en un vector.

```
Maximo_Vec (A[1..n]:entero, & nMax: entero, & j: entero)

entero i;
nMax=A[1];
j=1;
para (i=2; i<=n-1; i++)
{
si A[i]>nMax
   {
    nMax=A[i];
    j=i
   }
}
```

nMax devolverá el número máximo encontrado y j devolverá su índice.
Por ejemplo, dado el vector A: 2-5-7-3-4-1, nMax=7
j=3 (si empezamos indexando por el 1).

Mejor caso: MAX en primer elemento del array: j=1. La sentencia *si* se da menos veces.
Peor caso: MAX el último elemento del array: j=n. La sentencia *si* se hace (n-1) veces.
Centrándonos en el peor caso, como lo de dentro del *si* tiene complejidad constante (no en función de n), podemos confirmar que $\in \Theta(n)$.

Fin Ejercicio 1.3

Ejercicio 1.4

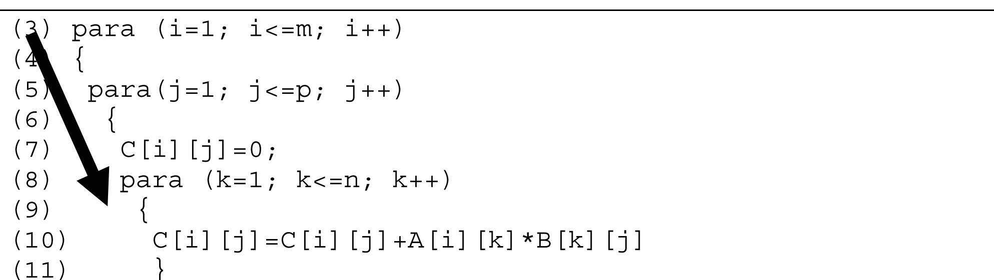
Cálculo de complejidad del producto de matrices, desarrollado por instrucciones.

```
(1) Prod_Mat (A[m][n]:real; B[n][p]:real; & C[m][p]:real)
(2) i, j, k: entero
(3) para (i=1; i<=m; i++)
(4) {
(5)  para(j=1; j<=p; j++)
(6)   {
(7)    C[i][j]=0;
(8)    para (k=1; k<=n; k++)
(9)     {
(10)     C[i][j]=C[i][j]+A[i][k]*B[k][j]
(11)     }
(12)  }
(13) }
```

La línea (3) se ejecutará m+1 veces; la (5) se ejecutará m(n+1); la (7), mn; la (8), mn(n+1) y la (10), mnn.

Si sumamos todo ello, obtenemos la expresión:
(m+1)+m(p+1)+mp+mp(n+1)+mpn = m+1+mp+m+mp+mpn+mp+mpn.
Si consideramos $m \approx n \approx p$, concluimos $T(n) \in \Theta(n^3)$

A partir de éste y de ejemplos anteriores, podemos observar que en los algoritmos con bucles anidados (de complejidad en el orden de n), la complejidad final es del orden del producto de tantas n, como bucles anidados haya de dicho orden.


```
(3) para (i=1; i<=m; i++)
(4) {
(5)  para(j=1; j<=p; j++)
(6)   {
(7)    C[i][j]=0;
(8)    para (k=1; k<=n; k++)
(9)     {
(10)     C[i][j]=C[i][j]+A[i][k]*B[k][j]
(11)     }
```

Fin Ejercicio 1.4

Ejercicio 1.5

Cálculo de complejidad de la traspuesta de una matriz por análisis de anidamiento de bucles

```
Trasp_Mat (& A[n][n]:entero)

i, j: entero;
para (i=1; i<=n-1; i++)
{
 para (j=i+1; j<=n; j++)
  {
   intercambio(A[i][j], A[j][i])
  }
}
```

En este caso nos encontramos con dos bucles anidados, ambos del orden de n, por tanto podemos concluir que $T(n) \in O(n^2)$

Fin Ejercicio 1.5

Ejercicio 1.6

Cálculo de la complejidad en la búsqueda binaria o dicotómica de un elemento en un vector ordenado crecientemente. (Talla del problema variable).

El algoritmo busca el número x en el vector V de n componentes entre sus componentes prim y ult. Si lo encuentra devuelve su posición, m, si no estaba, devuelve –1.

```
(1)    Busca_Bin    (V[n]:entero,    x:entero,    prim:entero,
ult:entero): entero
(2) i, j, m: enteros
(3) i=prim;
(4) j=ult;
(5)repetir
(6)  m=(i+j)/2;
(7)  si (V[m]>x) j=m-1
(8)  sino i=m+1
(9) hasta (i>=j O V[m]==x)
(10)si (V[m]==x) devuelve m
(11)sino devuelve -1
```

Supongamos el vector V= 2-12-20-35-36-40; y que el elemento a buscar es x= 12.
El algoritmo empieza llamándose con prim=1 y ult=6.
En (6) se calcula el elemento medio. En (7) si el medio es mayor que el buscado, rebaja el tope superior hasta el medio menos 1. En (8) si el medio es mayor aumenta el tope inferior hasta el medio mas 1.
Notar como de una iteración a otra va variando la talla del problema:

Paso: k	Tamaño del problema	Expresión equivalente
k=0 (Inicio)	n	$n/2^0$
k=1	n/2	$n/2^1$
k=2	n/4	$n/2^2$
k	...	**$n/2^k$**

En el peor caso, máximo de iteraciones, llegamos al tamaño del vector: 1. (En los casos de que x esté en el primer elemento, x esté en el último o x no esté)

Si hacemos $n/2^k=1$,
despejamos: $k=\log_2 (n)$ (o sea, el número de iteraciones en *repetir*)

La complejidad a trozos del algoritmo sería:

Inicializaciones: $\Theta(1)$;

repetir: $\Theta(\log_2 n)$;

Comprobaciones finales: $\Theta(1)$;

Al tratarse de una secuencia: $\Theta(1) + \Theta(\log_2 n) + \Theta(1)=\max (\Theta(1), \Theta(\log_2 n), \Theta(1)) = \Theta(\log_2 n)$.
$\forall n>2$.

<u>Fin Ejercicio 1.6</u>

Ejercicio 1.7

Cálculo de tiempos de un algoritmo que dibuja en pantalla un cuadrado de lado n (valor introducido por el usuario).

Implementación en C.

Por ejemplo: si n=4	Para los cálculos supondremos:
* * * *	tc: tiempo de comparación de enteros=0
* *	te: tiempo de escritura en pantalla
* *	tl: tiempo de lectura desde teclado
* * * *	ti: tiempo de incrementar una variable entera

Código	Tiempo
`int main()`	
`{`	
`(1) printf("Introduce un núm. mayor que 0");`	t(1)=te
`(2) scanf ("%d", &n);`	t(2)=tl
`(3) if (n==1)`	MEJOR CASO (if)
`printf ("*\n");`	t(3)=te
`else`	PEOR CASO
`{`	
`(4) for(j=0; j<n; j++)`	t(4)=n.ti
`(5) printf ("*");`	t(5)=n.te
`(6) printf("\n");`	t(6)=te (una sóla vez por estar fuera del bucle)
`(7) for (i=1; i<n-1; i++)`	
`{`	
`(8) printf("*");`	t(7)=(n-2).ti
`(9) for (j=1; j<n-1; j++)`	t(8)=(n-2).te
`(10) printf (" ");`	t(9)=(n-2).(n-2).ti
`}`	t(10)=(n-2).(n-2).te
`(11) for (j=0; j<n; j++)`	
`(12) printf("*");`	t(11)=t(4)=n.ti
`(13) printf("\n");`	t(12)=t(5)=n.te
`}`	t(13)=t(6)=te

Desde las líneas de código (4) a (6), se dibuja la primera fila de '*'. Entre las líneas de código (7) y (10) se dibujan desde la 2º a la penúltima fila. Con las líneas de código (11) a (13) –iguales que (4) a (6)- se dibuja la última fila de '*'.

Análisis por casos (Obligado por mejor/peor caso):

Mejor Caso: T(n)=t(1)+t(2)+t(3)

Peor Caso: T(n)=t(1)+t(2)+t(4)+t(5)+...+t(13).

Nos centramos en la expresión del Peor Caso.

Se deberá desarrollar la fórmula, y agrupar en función de n, pudiendo despreciar las constantes lineales.

<u>Fin Ejercicio 1.7</u>

Tema 2: Principios y tipos de recursividad

Vamos a introducir el concepto de recursividad, comentando los principios en los que se basa este planteamiento. Se estudiarán las etapas de Análisis y Composición con las correspondientes divisiones en casos para realizar planteamientos recursivos correctos. También se tratará la Verificación del diseño de programas recursivos para comprobar que los algoritmos propuestos terminan en algún momento. Por último haremos una clasificación de los tipos de recursividad con los que vamos a trabajar.

INDICE DEL TEMA

2.1.- Introducción a los algoritmos recursivos

2.2.- Etapas de Análisis y Composición

2.3.- Etapa de Verificación

2.4.- Tipos de recursividad

2.5.- Ejercicios de recursividad