# Diseño de algoritmos

*Jesús Bermúdez de Andrés. UPV-EHU*
*Guías para la solución de ejercicios: Análisis de algoritmos*

*Curso 2008-09*

1. Con un algoritmo de función de coste temporal $f(n) = n^3$ resolvemos problemas de tamaño $K$ en una hora. ¿Hasta qué tamaño podremos resolver, en el mismo tiempo, con una máquina 1000 veces más rápida? ¿Y si la función de coste fuese $f(n) = 2^n$?

   **Solución:**
   Supongamos que $f(n)$ representa el número de operaciones elementales que hace el algoritmo para entradas de tamaño $n$. Si consideramos que cada operación elemental necesita tiempo $t$ para realizarse, tenemos que $f(K)t$ es una hora. La máquina 1000 veces más rápida tarda tiempo $t/1000$ en realizar cada operación elemental. Así que, la solución la encontramos resolviendo la ecuación que iguala el tiempo utilizado por ambas máquinas: $f(K)t = f(x)t/1000$.

   Cuando $f(n) = n^3$ la respuesta es $x = 10\sqrt{K}$. Cuando $f(n) = 2^n$ la respuesta es $x = K + \lg_2 1000$.

2. Disponemos de dos algoritmos $A$ y $B$ para resolver el mismo problema, con implementaciones que realizan $8n^2$ y $64n \lg n$ operaciones elementales, para entradas de tamaño $n$, respectivamente. Determine para qué tamaños de la entrada, el algoritmo $A$ es más rápido que $B$.

   **Solución:**
   Para tamaños $n > 1$ y menores o iguales que el mayor número entero que satisface la inecuación $8n^2 < 64n \lg n$.

3. Determine para qué tamaños de entrada, en la misma máquina, es más rápido un algoritmo con función de coste $100n^2$ que otro con función de coste $2^n$.

   **Solución:**
   Hay que encontrar los valores naturales que satisfacen la inecuación $100n^2 < 2^n$. Es decir $\forall n \geq 15$.

4. Analice el siguiente algoritmo, definiendo la función de coste pertinente y especificando claramente qué es lo que se está considerando como tamaño de la entrada.

```
func Es_Equilibrado (V, inicio, fin) return integer
  for i in inicio .. fin loop
    izq ← 0;
    for k in inicio .. i − 1 loop izq ← izq+V(k); end loop;
    der ← 0;
    for k in i .. fin loop der ← der+V(k); end loop;
    if izq = der then return i;
  end loop;
  return 0;
```

**Solución:**
Determinamos como tamaño de la entrada el número de elementos de $V$ comprendidos entre los índices *inicio* y *fin*. Es decir $n = fin - inicio + 1$. Entonces la función de coste en tiempo queda definida por la siguiente suma:

$$\begin{aligned} f(n) &= \sum_{i=1}^{n} (\Theta(1) + \sum_{k=1}^{i-1} \Theta(1) + \sum_{k=i}^{n} \Theta(1)) \\ &= \sum_{i=1}^{n} (\Theta(1) + n\Theta(1)) \\ &= \Theta(n) + \Theta(n^2) \end{aligned}$$

Concluimos que Es_Equilibrado es $\Theta(n^2)$.

5. El siguiente algoritmo busca la primera aparición de un string $B(1..k)$ en el string $A(1..n)$; devuelve *true* y el índice de $A$ donde comienza $B$, si lo encuentra; y *false* en caso contrario. El valor $n - k + 1$ es la posición más a la derecha en $A$ donde podría comenzar $B$.

```
func StringSearch (A, B: String) return (boolean, natural)
  N ← A.length; K ← B.length; Inicio ← A.firstIndex;
  Encontrado ← false;
  Limite ← N-K+1;
  while not Encontrado and Inicio ≤ Limite loop
    I ← Inicio;
    J ← B.firstIndex;
    while J ≠ K+1 and then (A(i) = B(j)) loop
      I ← I+1; J ← J+1;
    end loop ;
    Encontrado ← (J=K+1);
    if not Encontrado then Inicio ← Inicio+1;
  end loop ;
  return (Encontrado, Inicio)
```

¿Cuántas veces se ejecuta la comparación $A(i) = B(j)$ en el peor caso? ¿Qué entradas dan lugar al peor caso? Obsérvese que el operador booleano *and then* hace que la comprobación sólo se realice cuando sea verdadera la condición J $\neq$ K+1.

**Solución:**
El string $B(1..k)$ puede comenzar en $A$ en $n - k + 1$ posiciones distintas, ocurriendo el caso peor cuando $B$ se compara, a partir de cada una de ellas, completamente (es decir, los $k$ caracteres de $B(1..k)$). Esto requerirá $(n-k+1)k$ comparaciones entre caracteres de $A$ y de $B$. Las entradas que dan lugar al peor caso son las que satisfacen el siguiente predicado:

$$\forall\, i, j (1 \leq i, j \leq n \rightarrow A(i) = A(j)) \wedge$$
$$\forall\, i, j (1 \leq i \leq k-1 \wedge 1 \leq j \leq n \rightarrow B(i) = A(j) \wedge B(k) \neq A(1))$$

Por ejemplo, $A(1 \ldots 7) = aaaaaaa$ y $B(1 \ldots 3) = aab$

6. El siguiente algoritmo realiza una búsqueda secuencial de un número $X$ en una tabla *T(I..F)*. Determine cuántas comparaciones, del número $X$ con un elemento de la tabla $T$, se realizan en el caso peor y en el caso medio.

```
func BUSQUEDA_SECUENCIAL (T: Tabla; I, F: integer; X: integer)
        return boolean
  Actual ← I
  while Actual ≤ F and then T(Actual) ≠ X loop
    Actual ← Actual+1
  end loop
  if Actual > F then return false
  else return true
```

**Solución:**
El caso peor ocurre cuando $X = T(F)$ o bien cuando $X \notin T(I..F)$. En ambos casos el número de comparaciones $T(\text{Actual}) \neq X$ es $f(n) = n$, siendo $n = F - I + 1$.

Para el análisis en el caso medio, vamos a suponer que todos los elementos de la tabla son distintos.

Primero analizaremos el algoritmo suponiendo que $X \in T(I..F)$; numeramos los índices $I..F$ con $1..n$, respectivamente, y suponemos que la probabilidad de que $X$ se encuentre en cada una de las posiciones $i : 1..n$ es la misma. Obsérvese que para cada posición $i : 1..n$, el número de comparaciones que se realizan es precisamente $i$. De manera que la función de coste, en este caso medio es:

$$M(n) = \sum_{i=1}^{n} \frac{1}{n}\, i = \frac{1}{n} \sum_{i=1}^{n} i = \frac{n+1}{2}$$

Ahora estudiemos el caso de que pueda ocurrir $X \notin T(1..n)$. Entonces tenemos $n+1$ casos posibles: Representaremos con $I_i$ $(1 \leq i \leq n)$ los casos en que $X$ está en la posición $i$; y con $I_{n+1}$ el caso en que $X \notin T(1..n)$.

Sea $q$ la probabilidad de que $X \in T(1..n)$ y supongamos que cada posición es igualmente probable. Entonces la probabilidad de cada caso es $p(I_i) = q\frac{1}{n}$ y $p(I_{n+1}) = 1 - q$. En los casos $I_i$ el número de comparaciones es $i$, y en el caso $I_{n+1}$ es $n$. Por lo tanto, la función de coste en este caso medio es:

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

Obsérvese que si sabemos que $X \in T(1..n)$, es decir $q = 1$, obtenemos el resultado anterior.

Concluimos que BUSQUEDA_SECUENCIAL es $\Theta$(n), siendo $n$ el número de elementos en la tabla de búsqueda, tanto en el caso peor como en promedio.

7. Un número natural $n \geq 1$ es *triangular* si es la suma de una sucesión ascendente no nula de naturales consecutivos que comienza en 1. Por tanto, los cinco primeros números triangulares son 1, $3 = 1+2$, $6 = 1+2+3$, $10 = 1+2+3+4$ y $15 = 1+2+3+4+5$.

   *a*) Escriba un algoritmo que, dado un entero positivo $n \geq 1$, decida si éste es un número triangular.

   *b*) Analice su algoritmo.

**Solución:**
Un algoritmo trivial para comprobar que un número natural positivo $n$ es triangular consiste en calcular sucesivamente los números triangulares y compararlos con $n$. En cada iteración, generaremos el siguiente número triangular $NT$. Si $NT = n$, habremos acabado con exito; si $NT > n$, habremos terminado con fracaso.

```
func EsTriangular (n: positivo) return boolean
  NT ← 1
  Ultimo_Sumando ← 1
  while NT < n loop
    Ultimo_Sumando ← Ultimo_Sumando + 1
    NT ← NT + Ultimo_Sumando
  end loop
  return (NT = n)
```

Las operaciones del bucle requieren tiempo constante y las iteraciones se realizan hasta que $NT$ es igual o mayor que $n$. Esto es, el número de veces que se realiza este bucle es igual al número de sumandos que tiene la sucesión

ascendente no nula de naturales consecutivos que comienza en 1 y cuya suma es

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

Obsérvese que si se ha salido del bucle porque $NT > n$, eso significa que en la iteración anterior, $NT$ aún era menor que $n$ y tras realizar una suma más ha sucedido $NT > n$; pero el orden del algoritmo sigue siendo el mismo.

Así pues, si despejamos $k$ de la ecuación $k^2 + k - 2n = 0$, obtenemos el orden del número de veces que se ejecuta el bucle: $\Theta(\sqrt{n})$, tanto si consideramos que hemos salido del bucle con éxito como con fracaso. Concluimos que EsTriangular $(n)$ es $\Theta(\sqrt{n})$.

El espacio de memoria extra empleado es $O(1)$, puesto que tan solo se han empleado dos variables locales.

8. Calcule la función de coste temporal de los siguientes algoritmos:

   *a*) **func** Serie ($n, m$: natural) **return** natural
   **if** $n \leq 1$ **then return** $m$
   **else return** $m$ + Serie($n - 1$, $2 \times m$)

   *b*) **func** Total ($n$: natural) **return** natural
   **if** $n \leq 1$ **then return** 1
   **else return** Total($n - 1$) + $2 \times$ Parcial($n - 1$)

   siendo

   **func** Parcial ($k$: natural) **return** natural
   **if** $k \leq 1$ **then return** 1
   **else return** $2 \times$ Parcial($k - 1$)

**Solución:**

*a*) Obsérvese que el tamaño del parámetro $m$ no es relevante, si consideramos elemental la operación $2 \times m$. Así que sólo aparece una llamada recursiva con el tamaño de la entrada disminuido en 1. Por lo tanto la función de coste $s(n)$ responde a la ecuación siguiente:

$$s(n) = \begin{cases} \Theta(1) & \text{si } n \leq 1 \\ s(n-1) + \Theta(1) & \text{si } n > 1 \end{cases}$$

Que resolviendo resulta $s(n) = \Theta(n)$.

*b*) Analicemos primero PARCIAL $(k)$. Observamos que se produce una sola llamada recursiva con PARCIAL $(k-1)$ y el resto son operaciones elementales. Así que la función de coste temporal $p(k)$ responde a la ecuación siguiente:

$$p(k) = \begin{cases} \Theta(1) & \text{si } k \leq 1 \\ p(k-1) + \Theta(1) & \text{si } k > 1 \end{cases}$$

Que resolviendo resulta $p(k) = \Theta(k)$.

Analicemos ahora TOTAL $(n)$. Se produce una sola llamada recursiva con TOTAL$(n-1)$. Una llamada al algoritmo PARCIAL$(n-1)$, que resulta de $\Theta$(n), según el análisis anterior. El resto son operaciones elementales. Así que la función de coste $t(n)$ responde a la ecuación siguiente:

$$t(n) = \begin{cases} \Theta(1) & \text{si } n \leq 1 \\ t(n-1) + \Theta(n) & \text{si } n > 1 \end{cases}$$

Que resolviendo resulta $t(n) = \Theta(n^2)$

9. Dado el algoritmo siguiente, que determina si una cadena $C$ es palíndromo:

```
func Pal (C, i, j) return booleano
  if i ≥ j then return verdadero
  elsif C(i) ≠ C(j) then return falso
  else return Pal(C, i + 1, j − 1)
```

Analice la evaluación de PAL$(C, 1, n)$ en el caso peor y en el caso medio, suponiendo equiprobabilidad de todas las entradas y siendo $\{a, b\}$ el alfabeto que forma las cadenas.

**Solución:**
Consideremos $n = j - i + 1$ el tamaño de la entrada. Estudiaremos, como operación característica, el número de comparaciones $C(i) \neq C(j)$. Es fácil ver que, en el caso peor, ese número viene dado por la función siguiente:

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

Que resolviendo resulta $f(n) = \lfloor \frac{n}{2} \rfloor = \Theta(n)$

En el caso medio, la equiprobabilidad de las entradas hace que la probabilidad de que los valores en los extremos sean distintos es $\frac{1}{2}$, ya que el alfabeto es {a,b}; y en ese caso el número de comparaciones que se realizan es 1. En caso de que los extremos sean iguales – cuya probabilidad es $\frac{1}{2}$ – el número de comparaciones es 1 más el número promedio de comparaciones para una cadena de tamaño $n-2$. Es decir, que la función de coste en el caso medio responde a la ecuación siguiente:

$$g(n) = \begin{cases} 0 & \text{si } n \leq 1 \\ \frac{1}{2} + \frac{1}{2}(1 + g(n-2)) & \text{si } n > 1 \end{cases}$$

Que resolviendo por expansión:

$$g(n) \;=\; 1+\frac{1}{2}g(n-2)=\ldots=\frac{1}{2^i}g(n-2i)+\sum_{k=0}^{i-1}\frac{1}{2^k}=\ldots$$

que cuando $i=\frac{n}{2}$ resulta $g(n)=2-\frac{2}{\sqrt{2}^n}=O(1)$.

Concluimos que PAL es $\Theta(n)$ en el caso peor, pero $O(1)$ en el caso medio.

10. Imagine un robot situado sobre unos raíles sin fin, que tiene la posibilidad de moverse un paso a la derecha o a la izquierda al ejecutarse la operación *posición* ← *posición* +1. La dirección del movimiento depende del valor del parámetro *sentido*. La operación *cambiar(sentido)* modifica el valor de *sentido* de *derecha* a *izquierda* y viceversa; y esos son los únicos valores del parámetro *sentido*. Además, el robot tiene la posibilidad de advertir la presencia de un determinado objeto (frente a la posición determinada por el parámetro *posición*) al ejecutarse la operación booleana *detecto_en(posición)*. El siguiente algoritmo mueve pendularmente al robot en busca del objeto citado, partiendo de un origen que marcamos con el valor 0.

```
límite ← 1
sentido ← derecha
loop
  posición ← 0
  while posición < límite loop
    posición ← posición +1
    if detecto_en(posición) then return (posición, sentido)
  end loop
  sentido ← cambiar(sentido)
  for i in 1..límite loop
    posición ← posición +1
  end loop
  límite ← 2 × límite
end loop
```

Asumiendo que el objeto será encontrado, analice el número de veces que se realizará la operación *posición* ← *posición* +1 en función de la distancia inicial del objeto al origen.

**Solución:**
Representemos con $n$ la distancia inicial del objeto al origen (ese es el tamaño de la entrada). Los bucles **while** y **for** son, respectivamente, de ida hasta *límite* y vuelta. En cada viaje de ida y vuelta, se realiza $2 \times$ *límite* veces la operación *posición* ← *posición* +1. En cada iteración del bucle externo **loop**, se duplica el valor de *límite*.

Obsérvese que debe existir un natural $k \geq 0$ tal que el valor $n$ satisfaga $2^{k-1} < n \leq 2^k$. Puesto que el valor inicial de *límite* es 1, en el peor caso $n = 2^k$ y

el número de veces que se realiza la operación *posición* ← *posición* +1 es el siguiente:

$$\begin{aligned} & 2 + (2 \times 2) + (2 \times 2^2) + \ldots + (2 \times 2^{k-1}) + 2^k \\ = \; & 2 \times (2^k - 1) + 2^k \\ = \; & 3 \times 2^k - 2 \end{aligned}$$

Concluimos que el número de veces que se realiza la operación *posición* ← *posición* +1 es $O(n)$.

11. Cuando los números tienen muchos dígitos, hay que cuestionarse si las operaciones aritméticas pueden considerarse operaciones elementales.

    *a*) Consideremos los algoritmos de suma y multiplicación, realizados con lápiz y papel, aprendidos en la escuela. Denominamos *elemental* a la multiplicación, respectivamente suma, de dos dígitos decimales. Justifique que el algoritmo típico que usa para multiplicar, con lápiz y papel, dos números enteros $A$ y $B$ realiza $\Theta(mn)$ multiplicaciones elementales y $\Theta(mn)$ sumas elementales, siendo $m$ y $n$ el número de dígitos de $A$ y $B$ respectivamente.

    *b*) Las operaciones $A \times 10$, $\lfloor B/10 \rfloor$ y $B$ *mod* 10 pueden considerarse de $O(1)$ (consisten, respectivamente, en añadir, eliminar y seleccionar una cifra decimal; por lo tanto, para realizarlas no necesitamos multiplicaciones ni sumas).

    Analice el número de multiplicaciones elementales, y separadamente el número de sumas elementales, que se realizan con el siguiente algoritmo de multiplicación. Los símbolos $\oplus$ y $\otimes$ representan, respectivamente, las operaciones de suma y multiplicación de números naturales realizadas con el algoritmo típico considerado en el apartado anterior. Fíjese que las operaciones $A \times 10$, $\otimes$ y MULTIPLICAR se realizan con algoritmos diferentes.

    **func** MULTIPLICAR (A,B: natural) **return** natural
      **if** B=0 **then return** 0
      **else return** A$\otimes$($B$ *mod* 10) $\oplus$ MULTIPLICAR($A \times 10$, $\lfloor B/10 \rfloor$)

**Solución:**

*a*) En el algoritmo típico de multiplicación, cada dígito de $A$ se multiplica con cada dígito de $B$; por lo tanto se realizan $\Theta(mn)$ multiplicaciones elementales.

Al disponer los resultados de las multiplicaciones elementales se forma una tabla que tiene $\Theta(m+n)$ columnas, y supongamos que $n$ filas (asumiendo que $m \geq n$ y que disponemos la multiplicación para que así sea). Así pués el número de sumas elementales es de $\Theta(n(m+n)) = \Theta(mn + n^2)$, que con la hipótesis de $m \geq n$ es igual a $\Theta(mn)$.

*b*) Primero analizamos el número de multiplicaciones, resolviendo la recurrencia siguiente:
$f(m,n)$ = número de multiplicaciones elementales realizadas por MULTIPLICAR$(A,B)$ cuando $A$ tiene $m$ dígitos y $B$ tiene $n$.

$$\begin{aligned} f(m,1) &= m \\ f(m,n) &= m + f(m+1,n-1) \;\; si\; n>1 \end{aligned}$$

Resolviendo por expansión:

$$\begin{aligned} f(m,n) &= m + f(m+1,n-1) \\ &= m + (m+1) + f(m+2,n-2) = \ldots \\ &= m + (m+1) + \ldots + (m+(n-2)) + f(m+n-1,1) \\ &= m + (m+1) + \ldots + (m+(n-2)) + (m+(n-1)) \\ &= mn + (1+2+3+...+n-1) \\ &= mn + \Theta(n^2) = \Theta(mn) \;\; \text{(asumiendo } \; m \geq n) \end{aligned}$$

Ahora analizamos el número de sumas elementales. Obsérvese que si $A$ tiene $m$ dígitos y $B$ tiene $n>1$, podemos decir que MULTIPLICAR$(A\times 10, \lfloor B/10 \rfloor)$ tiene $\Theta(m+n)$ dígitos, pero A$\otimes(B \; mod \; 10)$ sólo tiene $\Theta(m)$; así que al sumarlos se realizan $\Theta(m)$ sumas elementales. Por otro lado, si $n=1$ podemos decir que el número de sumas elementales es 1. Por lo tanto, el orden de la función
$g(m,n)$ = número de sumas elementales realizadas por MULTIPLICAR$(A,B)$ cuando $A$ tiene $m$ dígitos y $B$ tiene $n$.
es el de la recurrencia

$$\begin{aligned} g(m,1) &= 1 \\ g(m,n) &= \Theta(m) + g(m+1,n-1) \;\; si\; n>1 \end{aligned}$$

que se resuelve igual que la antes.