

Resumen de programación 3

Tema 4. Análisis de algoritmos.

Índice:

4.1. Introducción	3
4.2. Análisis de las estructuras de control	3
4.2.1. Secuencias	3
4.2.2. Sentencia condicional (if)	4
4.2.3. Bucles “para” (desde)	4
4.2.4. Llamadas recursivas	7
4.2.5. Bucles “mientras” (while) y “repetir” (repeat)	9
4.3. Uso de un barómetro	10
4.5. Análisis del caso medio	11
4.6. Resolución de recurrencias	12

Bibliografía:

Se ha tomado apuntes de los libros:

- *Fundamentos de algoritmia*. G. Brassard y P. Bratley
- *Estructuras de Datos y Algoritmos*. R. Hernández

Este tema es el último de los pertenecientes a costes y de los que más ejercicios cortos han entrado en exámenes. Por ello, merece la pena pararse sobre todo en las fórmulas para hallar el coste de la recursividad. Nos saltamos el apartado 4.4, que trata de ejemplos adicionales, ya que los iremos incluyendo en el resto del tema.

4.1. Introducción.

El **objetivo principal** de este libro es enseñar a diseñar algoritmos eficientes. Para resolver el mismo problema con varios algoritmos es preciso decidir cuál de ellos es el más adecuado para la aplicación considerada, recordemos que eso lo hemos visto en el tema 1. Una herramienta esencial para este propósito es el análisis de los algoritmos, aunque no tendremos una fórmula mágica para hallar la eficiencia de los algoritmos.

Existen técnicas básicas que suelen resultar útiles, tales como saber la forma de enfrentarse a estructuras de control y a ecuaciones de recurrencia, que será lo que tratemos en este capítulo.

Añadiremos en este capítulo el análisis del bucle “if” para completarlo aun más.

4.2. Análisis de las estructuras de control.

El análisis de los algoritmos suele efectuarse desde dentro hacia fuera. Seguiremos estos **pasos**:

- En primer lugar, se determina el tiempo requerido por las instrucciones individuales (suele estar acotado por una constante).
- Después se combinan estos tiempos de acuerdo con las estructuras del programa.

En esta sección ofreceremos unos principios generales que resultan útiles en aquellos análisis relacionados con las estructuras de control de uso más frecuente, así como ejemplos de la aplicación de estos principios (para mientras,...).

4.2.1. Secuencias.

Sean P_1 y P_2 dos fragmentos de un algoritmo (instrucciones o subalgoritmos). Sean t_1 y t_2 los tiempos requeridos por P_1 y P_2 , respectivamente. Estos tiempos pueden depender de distintos parámetros tales como el tamaño del caso.

La **regla de la composición secuencial** dice que el tiempo necesario para calcular " $P_1; P_2$ ", esto es, primero P_1 y después P_2 , es simplemente $t_1 + t_2$. Por la regla del máximo este tiempo está en $\theta(\max(t_1, t_2))$, es decir, como vimos previamente el coste del algoritmo lo determinará el más ineficiente.

Ejemplo:

$$\left. \begin{array}{l} P_1 \rightarrow t_1(n) \\ P_2 \rightarrow t_2(n) \end{array} \right\} t(n) = t_1(n) + t_2(n)$$

siendo:

$t_1(n)$: Lo que cuesta la primera función.

$t_2(n)$: Lo que cuesta la segunda función.

El coste del algoritmo, siguiendo la regla del máximo será:

$$t(n) \in \theta \left(\max(t_1(n), t_2(n)) \right)$$

4.2.2. Sentencia condicional (if).

Este añadido es del autor. Tenemos la siguiente sentencia condicional:

```
si B entonces  $S_1$ 
si no  $S_2$ 
fsi
```

Tenemos estos costes:

```
si  $B \rightarrow \theta(f_0(n))$ 
si  $S_1 \rightarrow \theta(f_1(n))$ 
si  $S_2 \rightarrow \theta(f_2(n))$ 
```

La cota superior será: $O \left(\max(f_0(n), f_1(n), f_2(n)) \right)$ (camino máximo).

La cota inferior será: $\Omega \left(\min(f_0(n), f_1(n), f_2(n)) \right)$ (camino mínimo).

4.2.3. Bucles “para” (desde).

Los bucles (lazos) “para” (desde) son los más fáciles de analizar. Considérese el bucle siguiente:

```
para  $i \leftarrow 1$  hasta  $m$  hacer  $P(i)$ 
```

Se nos dan varios **casos**:

- El tiempo $(t(n))$ requerido por $P(i)$ **no depende realmente de i** , aún cuando pudiera depender del tamaño del ejemplar o del ejemplar en sí. Es el caso más sencillo.

En este caso, el tiempo total requerido por el bucle es simplemente $l = m * t$ o bien $T(n) = m * t(n)$. Se harían m iteraciones y cada uno con el mismo coste.

Sería algo así como este bucle “mientras”:

```
 $i \leftarrow 1$ ;
mientras  $i < m$  hacer
   $P(i)$ ;
   $i \leftarrow i + 1$ ;
```

Asignaremos costes unitarios (operaciones elementales) a la comprobación $i < m$, a las instrucciones $i \leftarrow 1$ e $i \leftarrow i + 1$ y a las instrucciones de salto implícitas en el bucle “mientras”.

Se demuestra que este tiempo al hacer las operaciones está acotado superiormente por $m * t$, tal como habíamos escrito antes.

- b. El tiempo $(t(n))$ requerido por $P(i)$ **varía como función de i**. Si despreciamos el tiempo requerido por el bucle de control, para $m \geq 1$, entonces ese mismo bucle “para” requiere un tiempo que está dado por una suma $\sum_{i=1}^m t(i)$. Sería entonces $T(n) = \sum_{i=1}^m t(i, n)$ el coste del bucle.

Ejemplo: Analizamos el coste de este bucle “para”:

```
funcion fibiter(n)
  i ← 1; j ← 0;
  para k ← 1 hasta n hacer
    j ← i + j;
    i ← j - i;
  fpara
  devolver j
ffuncion
```

Se nos darán dos casos en función de los costes de las operaciones aritméticas:

1. **Las operaciones aritméticas se consideran como de coste unitario.** Las instrucciones de dentro del bucle “para” requieren un tiempo constante. Suponemos que el tiempo requerido por estas instrucciones está acotado superiormente por alguna constante c . El tiempo requerido por el bucle “para” está acotado superiormente por n veces esta constante: $n \cdot c$.

El algoritmo tiene coste $\theta(n)$.

2. **Las operaciones aritméticas no se consideran como de coste unitario.** Podemos llegar a ver como al paso del bucle se vuelve costosa una instrucción tal como $j \leftarrow i + j$, por haber operandos muy grandes.

Sea c una constante tal que este tiempo está acotado superiormente por $c \cdot k$ para todo $k \geq 1$. Si despreciamos el tiempo requerido por el control del bucle y las instrucciones que preceden al bucle concluiremos que el tiempo requerido por el algoritmo está acotado superiormente por:

$$\sum_{k=1}^n c \cdot k = c \cdot \sum_{k=1}^n k = c \cdot \frac{n \cdot (n+1)}{2} \in O(n^2)$$

Un razonamiento similar indica que este tiempo se encuentra en $\Omega(n^2)$ y se deduce, por tanto, que está en $\theta(n^2)$.

El análisis del bucle “para” que empiezan en un valor que no sea 1 o que avanzan con pasos mayores, debería resultar evidente.

Ejemplo: Se nos da el siguiente bucle:

```
para i ← 5 hasta m paso 2 hacer P(i)
```

Aquí, $P(i)$ se ejecuta $((m - 5) \div 2) + 1$ veces siempre que $m \geq 3$.

Ejemplo completo de análisis de un algoritmo con bucle “para”. Para ello, analizaremos la **ordenación por selección**:

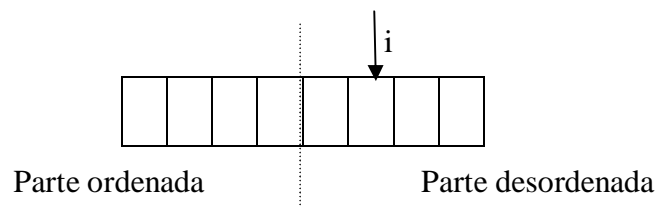
```

procedimiento seleccionar( $T(1..n)$ )
  para  $i \leftarrow 1$  hasta  $n - 1$  hacer
     $minj \leftarrow 1$ ;  $minx \leftarrow T[i]$ ;
    para  $j \leftarrow i + 1$  hasta  $n$  hacer
      si  $T[j] < minx$  entonces
         $minj \leftarrow j$ ;
         $minx \leftarrow T[j]$ ;
    fsi
  fpara
   $T[minj] \leftarrow T[i]$ ;  $T[i] \leftarrow minx$ ;
fpara
fprocedimiento
  
```

Seguiremos estos pasos:

1. Análisis de su funcionamiento:

Dentro del bucle para “exterior” tendremos uno “interior” que nunca sabremos si realiza las mismas instrucciones con el mismo coste. Estaremos en el caso b) de los vistos anteriormente.



La **idea básica** es seleccionar el menor elemento de una parte desordenada del vector y colocarlo en la posición del primer elemento no ordenado. En un primer paso, se recorre el vector hasta encontrar el elemento menor. Para ello se coloca el primer elemento en una variable temporal y se va comparando con los demás elementos del vector tal que si se encuentra uno menor se asigna a la variable temporal. Recorrido todo el vector, el elemento de la variable temporal (que será el menor) se intercambia con el de la primera posición (el primero escogido de la parte desordenada). Seguidamente, se considera únicamente la parte del vector no ordenado y se repite el proceso de búsqueda del menor, y así sucesivamente.

Se puede describir de la siguiente manera:

- Seleccionar el elemento menor de la parte del vector no ordenada.
- Colocarlo en la primera posición de la parte no ordenada del vector.

El coste es independiente de cómo vienen ordenados los datos, es decir, del contenido de los datos.

2. Análisis del coste:

Veremos el número de instrucciones y analizaremos el coste del algoritmo en el caso peor, como es habitual:

```
procedimiento seleccionar( $T(1..n)$ )
(1) para  $i \leftarrow 1$  hasta  $n - 1$  hacer
     $minj \leftarrow 1$ ;  $minx \leftarrow T[i]$ ;
(2)    para  $j \leftarrow i + 1$  hasta  $n$  hacer
(3)        { si  $T[j] < minx$  entonces
             $minj \leftarrow j$ ;
             $minx \leftarrow T[j]$ ;
            fsi
        fpara
     $T[minj] \leftarrow T[i]$ ;  $T[i] \leftarrow minx$ ;
fpara
fprocedimiento
```

Suponemos que las operaciones de suma, asignación,... son **elementales**, por tanto, no las consideraremos en el coste total del algoritmo. Pasamos a ver el número de instrucciones en cada paso:

(1) n iteraciones \approx coste n

(2) $(n - i)$ iteraciones

Tendremos que el bucle “para” interior y el “si” tendrán el siguiente número de instrucciones:

$$(2) + (3) \quad T(n) \approx \sum_{i=1}^n (n - i).$$

Resolviendo la sucesión, tendremos que

$$T(n) \approx \sum_{i=1}^n (n - i) = (n - 1) * (n - 2) * \dots * 1 \approx n^2.$$

Como conclusión, el coste en todos los casos es **$O(n^2)$** .

4.2.4. Llamadas recursivas.

El análisis de algoritmos recursivos suele ser sencillo. Una inspección sencilla del algoritmo suele dar lugar a una ecuación de recurrencia que imita el flujo de control dentro del algoritmo. Una vez que se ha obtenido la ecuación de recurrencia, se pueden aplicar las técnicas generales para resolverlas.

Ejemplo: Considérese el algoritmo recursivo siguiente:

```

funcion fibrec (n)
  si  $n < 2$  entonces devolver n
  si no devolver fibrec ( $n - 1$ ) + fibrec ( $n - 2$ )
  
```

Sea $T(n)$ el tiempo requerido por una llamada a fibrec (n):

- Si $n < 2$, el algoritmo devuelve simplemente n , lo cual requiere un tiempo constante a .
- En caso contrario, la mayor parte del trabajo se invierte en dos llamadas recursivas, que requieren un tiempo $T(n - 1)$ y $T(n - 2)$, respectivamente.

Sea $h(n)$ el trabajo implicado en esta suma y en este control, es decir, el tiempo requerido por una llamada a fibrec(n) ignorando los tiempos invertidos dentro de las dos llamadas recursivas. Por definición de $T(n)$ y de $h(n)$, obtenemos la siguiente recurrencia:

$$\begin{cases} a & \text{Si } n = 0 \text{ ó } n = 1 \text{ (} n < 2 \text{)} \\ T(n) = T(n - 1) + T(n - 2) + h(n) & \text{En caso contrario} \end{cases}$$

Trataremos, por tanto, estos casos:

- **Si contamos las sumas con coste unitario**, $h(n)$ está acotado por una constante y la ecuación de recurrencia es similar a la ya encontrada antes. Tenemos que $T(n) \in \theta(f(n))$, razonando de manera similar para la cota inferior. Concluimos, entonces, que fibrec (n) requiere un tiempo exponencial en n .
- **Si no se cuentan las adiciones con un coste unitario**, $h(n)$ ya no queda acotado por una constante. $h(n)$ está dominado por la adición de f_{n-1} y f_{n-2} para n suficientemente grande. La adición requiere un tiempo $h(n) \in \theta(f(n))$.

Deducimos que el resultado es el mismo independientemente de si $h(n)$ es constante o lineal: $T(n) \in \theta(f(n))$. La única diferencia es la *constante multiplicativa oculta* en la notación θ .

4.2.5. Bucles “mientras” (while) y “repetir” (repeat).

Para analizar estos bucles no existe una forma evidente a priori de saber cuántas veces tendremos que pasar por el bucle. Podremos emplear dos **técnicas**:

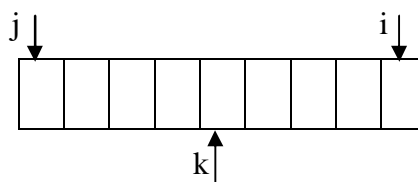
- Es la técnica estándar. Es hallar una función de las variables implicadas cuyo valor se decremente en cada pasada.
- Para determinar el número de veces que se repite el bucle necesitamos conocer mejor la forma en que disminuye el valor de esta función. Para analizar el bucle “mientras” de manera alternativa consiste en tratarlo como un algoritmo recursivo.

Ejemplo: Estudiaremos con detalle el algoritmo de búsqueda binaria.

```
funcion busq_binaria( $T[1..n], x$ )  
   $i \leftarrow 1; j \leftarrow n;$   
  mientras  $i < j$  hacer  
     $k \leftarrow (i + j) \div 2;$   
    caso_de  $x < T[k]: j \leftarrow k - 1;$   
            $x = T[k]: i, j \leftarrow k; \quad \{devolver\ k\}$   
            $x > T[k]: i \leftarrow k + 1;$   
  fcaso  
fmientras  
dev  $i;$   
ffuncion
```

Analizaremos el algoritmo siguiendo los pasos anteriores:

1. Análisis del funcionamiento:



La **idea básica** que subyace a la búsqueda binaria es comparar x con el elemento k que está en la posición media de T . El objetivo de la búsqueda binaria es hallar un elemento x de un vector $T[1..n]$ que está ordenado de modo no decreciente (sólo podremos aplicar la búsqueda binaria si este vector está ordenado así). Supongamos por sencillez que está garantizado que x aparece al menos una vez en T . Se nos pide buscar un entero i tal que $1 \leq i \leq n$ y $T[i] = x$.

Como el elemento k está en el centro del vector se dan tres casos:

- Elemento x está a la izquierda de k .
- Elemento x coincide con k .
- Elemento x está a la derecha de k .

Mediante sucesivas divisiones por 2 llegaremos hasta que $i = j$. En el primer caso, en el que x sea menor que la mitad moveremos el puntero j a la mitad izquierda. En el último caso, moveremos el puntero i . El caso intermedio, sería ya la solución, encontrando el elemento x .

Para resolverlo definiremos d como el número de posiciones donde podrá ir el elemento:

$$d = j - i + 1$$

En el paso inicial, consideramos las n posiciones:

$$\hat{d} = \left(\frac{i+j}{2} - 1\right) - i + 1 = \frac{j-i}{2} \approx \frac{d}{2}.$$

Hemos sustituido j por su mitad, es decir, $j = k - 1 = \frac{i+j}{2}$ para el caso $x < T[k]$ (elemento en la mitad izquierda).

El significado de las variables hasta el momento es el siguiente:

d : Representa los valores de $j - i + 1$ antes del bucle “mientras”.

\hat{d} : Representa el número de iteraciones tras finalizar el bucle, antes de la siguiente iteración.

Para el caso $x > T[k]$ (elemento en la mitad derecha), tenemos que $k = \frac{i+j}{2}$. Sustituyendo, como vimos previamente:

$$\hat{d} = j - \left(\frac{i+j}{2} - 1\right) + 1 = \frac{j-i}{2} \approx \frac{d}{2}.$$

El coste será $O(\log(n))$, porque siempre se divide por 2 para buscar el elemento.

4.3. Uso de un barómetro.

Definición: Una instrucción barómetro es aquella que se ejecuta por lo menos con tanta frecuencia como cualquier otra instrucción del algoritmo.

Siempre que el tiempo requerido por cada instrucción esté acotado por una constante, el tiempo requerido por el algoritmo completo es del orden exacto del número de veces que se ejecuta la instrucción barómetro.

Ejemplo: analizaremos el siguiente algoritmo:

```
funcion fibiter(n)
   $i \leftarrow 1; j \leftarrow 0;$ 
  para  $k \leftarrow 1$  hasta  $n$  hacer
     $j \leftarrow i + j;$ 
     $i \leftarrow j - i;$ 
  fpara
  devolver j
ffuncion
```

Podremos considerar que la instrucción “ $j \leftarrow i + j$ ” se puede tomar como un barómetro, por ser ejecutada un número de veces igual a n . Se ve entonces que el algoritmo requiere un tiempo que está en $\theta(n)$.

Cuando un algoritmo contiene varios bucles anidados, toda instrucción del bucle más interno puede utilizarse en general como barómetro. Sin embargo, hay que hacer esto con cuidado, porque hay casos en los que es preciso tener en consideración el control implícito del bucle. Esto sucede cuando algunos de los bucles se ejecutan *cero veces*.

4.5. Análisis del caso medio.

Requiere suponer **a priori** una distribución de probabilidad para los casos en que se pedirá que resuelva nuestro algoritmo.

4.6. Resolución de recurrencias.

Tendremos dos tipos:

- Reducción por sustracción:

La ecuación de la recurrencia es la siguiente:

$$T(n) = \begin{cases} c * n^k & \text{si } 0 \leq n < b \\ a * T(n - b) + c * n^k & \text{si } n \geq b \end{cases}$$

La resolución de la ecuación de recurrencia es:

$$T(n) = \begin{cases} \theta(n^k) & \text{si } a < 1 \\ \theta(n^{k+1}) & \text{si } a = 1 \\ \theta(a^{n \text{ div } b}) & \text{si } a > 1 \end{cases}$$

- Reducción por división:

La ecuación de la recurrencia es la siguiente:

$$T(n) = \begin{cases} c * n^k & \text{si } 1 \leq n < b \\ a * T(n/b) + c * n^k & \text{si } n \geq b \end{cases}$$

La resolución de la ecuación de recurrencia es:

$$T(n) = \begin{cases} \theta(n^k) & \text{si } a < b^k \\ \theta(n^k * \log(n)) & \text{si } a = b^k \\ \theta(n^{\log_b a}) & \text{si } a > b^k \end{cases}$$

siendo:

a: Número de llamadas recursivas.

b: Reducción del problema en cada llamada.

$c * n^k$: Todas aquellas operaciones que hacen falta además de las de recursividad.