

Resumen de programación 3

Tema 2. Algoritmia elemental.

Índice:

2.3. La eficiencia de los algoritmos	3
2.4. Análisis de “caso medio” y “caso peor”	4
2.5. ¿Qué es una operación elemental?	7
2.6. Más factores de tiempo	8

Bibliografía:

Se ha tomado apuntes del libro:

- *Fundamentos de algoritmia*. G. Brassard y P. Bratley

Nos hemos saltado los apartados 2.1 y 2.2, que corresponden con la introducción y problemas y ejemplares respectivamente, ya que lo siguiente es más propio del tema. Por ello, empezaremos por el 2.3. Por otro lado, comentar que este tema ha sido costoso el resumirlo debido a que el resto de temas serán más complicados de estudiar y conviene entender bien el concepto básico de la algoritmia, que junto con el tema 1 no tiene ejercicios.

2.3. La eficiencia de los algoritmos.

Cuando tengamos que resolver un problema es posible que estén disponibles varios algoritmos adecuados. Deseamos seleccionar el mejor posible, tal y como vimos en la introducción (tema 1). Para ello tendremos dos enfoques:

- El **enfoque empírico** (o **a posteriori**): Consiste en programar las técnicas comparadoras e ir probándolas en distintos casos con ayuda de una computadora.
- El **enfoque teórico** (o **a priori**): Es el que nosotros propugnamos en este libro. Consiste en determinar matemáticamente la *cantidad de recursos* necesarios para cada uno de los algoritmos como función del tamaño de los casos considerados. Nos interesarán en especial estos recursos:
 1. **Tiempo de computación**: Es el que usaremos a lo largo del libro, por tanto, será el más importante a tener en cuenta.
 2. **Espacio de almacenamiento**: Indica el espacio que ocupa un algoritmo. No lo emplearemos en este tema y posteriores.

Compararemos los algoritmos tomando como base sus tiempos de ejecución. Cuando hablemos de la **eficiencia** de un algoritmo querremos decir lo rápido que se ejecuta y en esto consistirá nuestro análisis del coste (temas 3 y 4).

Utilizaremos la palabra *tamaño* para indicar cualquier entero que mida de alguna forma el número de componentes de un ejemplar. Daremos algunas veces la eficiencia de nuestros algoritmos en términos del valor del ejemplar que estemos considerando en lugar de considerar su tamaño.

La **ventaja** de la aproximación teórica es que no depende ni de la computadora que se esté utilizando ni del lenguaje de programación ni siquiera de las habilidades del programador.

Si deseamos medir la eficiencia de un algoritmo en términos del tiempo que necesita para llegar a una respuesta, entonces no existe una opción tan evidente. Se puede expresar en *segundos*, al no disponer de una computadora estándar a la cual referir todas las medidas.

Los factores para determinar el tiempo necesario del algoritmo son:

1. **Implementación**: Depende de:
 - Plataforma.
 - Lenguaje.
 - Compilador.
2. **Tamaño del problema**: Un problema grande necesitará más tiempo.
3. **Contenido de los datos**: Cómo vienen ordenados los datos.

Al no poder referir todas las medidas en una computadora estándar tendremos este principio:

Principio de invarianza: Corresponde con el primer factor antes citado, que es la implementación.

Definición: Dos implementaciones distintas de un mismo algoritmo no diferirán en su eficiencia en más de una constante multiplicativa. Si dos implementaciones del mismo algoritmo necesitan $t_1(n)$ y $t_2(n)$ segundos, respectivamente, para resolver un caso de tamaño n , entonces siempre existen constantes positivas c y d , tales que $t_1(n) \leq c * t_2(n)$ y $t_2(n) \leq d * t_1(n)$ siempre que n sea suficientemente grande:

$$\begin{array}{lll} \text{Implementación 1} & t_1(n) & t_1(n) \leq c * t_2(n) \\ \text{Implementación 2} & t_2(n) & t_2(n) \leq d * t_1(n) \end{array}$$

El principio sigue siendo cierto sea cual fuere la computadora utilizada para implementar el algoritmo. Para un problema más grande va a tardar lo mismo en la misma proporción, salvo por una constante multiplicativa, denominada en estos casos constante oculta.

2.4. Análisis de “caso medio” y “caso peor”

Analizaremos el coste de este algoritmo, que corresponde con el de la ordenación por inserción:

```

procedimiento insertar  $T([1..n])$ 
  para  $i \leftarrow 2$  hasta  $n$  hacer
     $x \leftarrow T[i]$ ;  $j \leftarrow i - 1$ ;
    mientras  $j > 0$  y  $x < T[j]$  hacer
       $T[j + 1] \leftarrow T[j]$ ;
       $j \leftarrow j - 1$ ;
    fmientras
       $T[j + 1] \leftarrow x$ 
  fpara
fprocedimiento

```

La nomenclatura empleada será:

\leftarrow : Asignaciones.

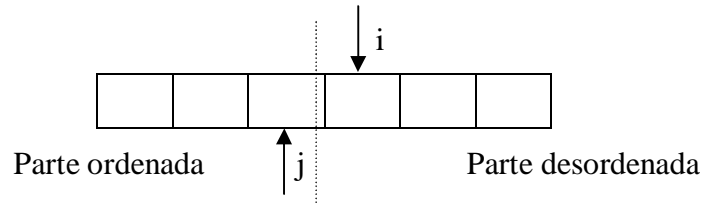
mientras .. fmientras, para .. fpara,...: Bucles mientras, para,...

Como norma general para analizar el coste realizaremos estos pasos:

1. **Análisis del funcionamiento:** Simularemos su funcionamiento, usando para ello ejemplos de distintos valores de vectores, ...
2. A continuación, **analizaremos el coste** propiamente dicho.

En nuestro ejemplo tendremos lo siguiente, aunque lo retomaremos en el resumen del tema 7, no obstante lo veremos en este apartado:

1. Análisis del funcionamiento:



La **idea básica** es insertar un elemento dado en el lugar que le corresponda de una parte ordenada del vector. En un primer paso, se considera que la parte ordenada está formada por el primer elemento del vector. Entonces, se elige el elemento de la posición 2 y se inserta en la parte ordenada, de manera que si debe estar en la posición anterior se desplaza el elemento de la posición 1 hacia la derecha, haciéndole sitio, y se inserta en la posición 1. Como puede observarse se requerirá de una variable temporal que almacene el elemento a insertar. Seguidamente, se considera la parte del vector $a[1] \dots a[2]$ ordenada, se elige el tercer elemento y se siguen las mismas opciones. Este método se repite hasta que el último elemento $a[n]$ haya sido tratado.

En resumen, el algoritmo puede describirse de la siguiente forma:

- Tomar un elemento en la posición i .
- Buscar en su lugar en las posiciones anteriores.
- Mover hacia la derecha los restantes.
- Insertarlo.

2. Análisis del coste:

El algoritmo de ordenación por inserción es:

```

procedimiento insertar  $T([1..n])$ 
(1)  para  $i \leftarrow 2$  hasta  $n$  hacer
(2)     $x \leftarrow T[i]; j \leftarrow i - 1;$ 
(3)    mientras  $j > 0$  y  $x < T[j]$  hacer
(4)      {  $T[j + 1] \leftarrow T[j]$ 
            $j \leftarrow j - 1$ 
      }
      fmientras
(5)     $T[j + 1] \leftarrow x$ 
      fpara
fprocedimiento
  
```

Tendremos en cuenta estos tiempos:

t_a : Asignación	t_c : Comparación	t_r : Resta
t_i : Incremento	t_v : Acceso al vector	

Las distintas sentencias del algoritmo anterior son:

$$(1) t_a + (n - 1) * t_i + n * t_c$$

siendo:

t_a : Asigna i el valor 2 ($i \leftarrow 2$).

t_i : Incrementa $n - 1$ veces (de 2 a n).

t_c : Compara de 2 a n , incluido n , que sale del bucle.

$$(2) (2 * t_a + t_v + t_r) * (n - 1)$$

siendo:

t_a : Significa que hay dos asignaciones ($x \leftarrow T[i]$ y $j \leftarrow i - 1$)

t_v : Indica acceso al vector ($T[i]$).

$$(3) (2 * t_c + t_v) * N(i, T)$$

$$(4) (2 * t_v + 2 * t_a + t_i + t_r) * N(i, T)$$

siendo:

$N(i, T)$: Número de iteraciones del bucle “mientras” en cada pasada del bucle “para”. Depende de i y del vector T , por lo que será variable.

El tiempo total del bucle “mientras” será la suma de (3) y (4):

$$(3) + (4) (3 * t_v + 2 * t_a + 2 * t_c + t_r + t_i) * N(i, T)$$

$$(5) (t_v + t_i + t_a) * (n - 1)$$

Hemos visto los tiempos que emplean. Luego analizaremos los casos peores y mejores, aunque el caso promedio ya veremos que no es tan fácil hallarlo, ya que requiere un cono cimienta *a priori* acerca de la distribución de los casos que hay que resolver. Esto suele ser un requisito poco realista.

Normalmente, consideraremos el **caso peor** del algoritmo, esto es, para cada tamaño de caso sólo consideraremos aquéllos en los cuales el algoritmo requiera más tiempo, a no ser que se indique lo contrario. Suele ser más difícil analizar el comportamiento **medio** del algoritmo que hacerlo en el caso peor.

2.5. ¿Qué es una operación elemental?

Corresponderá también como el principio de invarianza con el primer factor visto previamente, recordemos que era la **implementación**.

Definición: Una **operación elemental** es aquélla cuyo tiempo de ejecución puede ser acotada superiormente por una constante que sólo dependerá de la implementación particular usada: de la máquina, del lenguaje de programación, etc.

De esta manera, la constante no depende ni del tamaño ni de los parámetros del ejemplar que se esté considerando. Pueden ser operaciones tales como sumas, restas, multiplicaciones, divisiones, accesos a un vector, operaciones booleanas, comparaciones.

Veremos al igual que el anterior algoritmo esta parte en el tema 7, no obstante hay que hacer hincapié en la definición, porque se usa mucho y es importante.

Un **ejemplo:** Supongamos que cuando se analiza algún algoritmo, encontramos que para resolver un caso de un cierto tamaño se necesita efectuar a adiciones, m multiplicaciones y s instrucciones de asignación.

Supongamos también que se sabe que una suma nunca requiere más de t_a ms., que una multiplicación nunca requiere más de t_m ms. y que una asignación nunca requiere más de t_s ms., donde t_a , t_m y t_s son constantes que dependen de la máquina utilizada. El tiempo total T requerido por nuestro algoritmo estará acotado por:

$$T \leq a * t_a + m * t_m + s * t_s \leq \max(t_a, t_m, t_s) * (a + m + s)$$

siendo:

a, m, s : Variables según la implementación.

t_a, t_m, t_s : Constantes que cambiarán de una implementación a otra.

T estará acotado por un múltiplo constante del número de operaciones elementales (ejecutadas a coste unitario) que hay que ejecutar.

En **conclusión**, el primer factor del tiempo de ejecución visto anteriormente, que es la implementación **no** afectará mucho al coste del algoritmo.

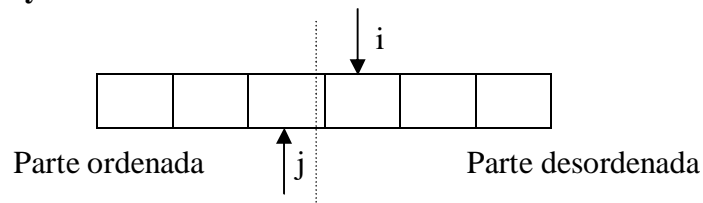
2.6. Más factores de tiempo

En cuanto a los otros **factores** del tiempo de algoritmo trataremos los dos que nos quedan los siguientes (añadido del autor, tomando apuntes de otra persona). Recordemos que hemos visto previamente la implementación (apartado 2.3):

- El contenido de los datos:

En nuestro ejemplo de la ordenación por inserción tendremos estos **casos**:

1. Vector ya ordenado:



Nunca se entrará en el bucle “mientras” (mientras $j > 0$ y $x < T[j]$) por no cumplirse la condición de $x < T[j]$. Por tanto, $N(i, T) = 0$.

Recordemos que $N(i, T)$ es el número de iteraciones del bucle “mientras” en cada pasada del bucle “para”.

2. Vector completamente desordenado (de mayor a menor):

Se recorrerán todos los elementos de la parte desordenada hasta encontrar su posición. En este caso, $N(i, T) = i$.

- Tamaño del problema:

Siguiendo el ejemplo anterior tendremos:

(1) Coste n

(2) Coste n

$$(3) + (4) \begin{cases} \text{- Caso mejor} & \text{Coste } 0 \\ \text{- Caso peor} & \sum_{i=2}^n i \end{cases}$$

(5) Coste n

Caso mejor: Vector ya ordenado. Tiene coste lineal (n).

Caso peor: Vector completamente desordenado.

$$n + \sum_{i=2}^n i \approx \text{coste } n^2 \quad \left(\sum_{i=2}^n i = \frac{n*(n+1)}{2} \approx n^2 \right)$$

Caso promedio: En general haremos el análisis para el caso peor. No es normal hacerlo para el caso promedio.