LA COMPLEJIDAD DE LOS ALGORITMOS

1 INTRODUCCIÓN

En un sentido amplio, dado un problema y un dispositivo donde resolverlo, es necesario proporcionar un método preciso que lo resuelva, adecuado al dispositivo. A tal método lo denominamos algoritmo.

En el presente texto nos vamos a centrar en dos aspectos muy importantes de los algoritmos, como son su diseño y el estudio de su eficiencia. El primero se refiere a la búsqueda de métodos o procedimientos, secuencias finitas de instrucciones adecuadas al dispositivo que disponemos, que permitan resolver el problema. Por otra parte, el segundo nos permite medir de alguna forma el coste (en tiempo y recursos) que consume un algoritmo para encontrar la solución y nos ofrece la posibilidad de comparar distintos algoritmos que resuelven un mismo problema. Este capítulo está dedicado al segundo de estos aspectos: la eficiencia. En cuanto a las técnicas de diseño, que corresponden a los patrones fundamental es sobre los que se construyen los algoritmos que resuelven un gran número de problemas, se estudiarán en los siguientes capítulos.

2 EFICIENCIA Y COMPLEJIDAD

Una vez dispongamos de un algoritmo que funciona correctamente, es necesario definir criterios para medir su rendimiento o comportamiento. Estos criterios se centran principalmente en su simplicidad y en el uso eficiente de los recursos. A menudo se piensa que un algoritmo sencillo no es muy eficiente. Sin embargo, la sencillez es una característica muy interesante a la hora de diseñar un algoritmo, pues facilita su verificación, el estudio de su eficiencia y su mantenimiento. De ahí que muchas veces prime la simplicidad y legibilidad del código frente a alternativas más crípticas y eficientes del algoritmo. Este hecho se pondrá de manifiesto en varios de los ejemplos mostrados a lo largo de este libro, en donde profundizaremos más en este compromiso. Respecto al uso eficiente de los recursos, éste suele medirse en función de dos parámetros: **el espacio**, es decir, memoria que utiliza, y **el tiempo**, lo que tarda en ejecutarse. Ambos representan los costes que supone encontrar la solución al problema planteado mediante un algoritmo. Dichos parámetros van a servir además para comparar algoritmos entre sí, permitiendo determinar el más adecuado entre varios que solucionan un mismo problema. En este capítulo nos centraremos solamente en la eficiencia temporal. El tiempo de ejecución de un algoritmo va a depender de diversos factores como son: los **datos de entrada** que le suministremos, la **calidad del código** generado por el compilador para crear el programa objeto, la **naturaleza y rapidez de las instrucciones** máquina del procesador concreto que ejecute el programa, y la **complejidad intrínseca** del algoritmo. Hay dos estudios posibles sobre el tiempo:

1. Uno que proporciona una medida teórica (a priori), que consiste en obtener una función que acote (por arriba o por abajo) el tiempo de ejecución del algoritmo para unos valores de entrada dados.

2. Y otro que ofrece una medida real (a posteriori), consistente en medir el tiempo de ejecución del algoritmo para unos valores de entrada dados y en un ordenador concreto. Ambas medidas son importantes puesto que, si bien la primera nos ofrece estimaciones del comportamiento de los algoritmos de forma independiente del ordenador en donde serán implementados y sin necesidad de ejecutarlos, la segunda representa las medidas reales del comportamiento del algoritmo. Estas medidas son funciones temporales de los datos de entrada. Entendemos por tamaño de la entrada el número de componentes sobre los que se va a ejecutar el algoritmo. Por ejemplo, la dimensión del vector a ordenar o el tamaño de las matrices a multiplicar. La unidad de tiempo a la que debe hacer referencia estas medidas de eficiencia no puede ser expresada en segundos o en otra unidad de tiempo concreta, pues no existe un ordenador estándar al que puedan hacer referencia todas las medidas. Denotaremos por

T(n) el tiempo de ejecución de un algoritmo para una entrada de tamaño n. Teóricamente T(n) debe indicar el número de instrucciones ejecutadas por un ordenador idealizado. Debemos buscar por tanto medidas simples y abstractas, independientes del ordenador a utilizar. Para ello es necesario acotar de alguna forma la diferencia que se puede producir entre distintas implementaciones de un mismo algoritmo, ya sea del mismo código ejecutado por dos máquinas de distinta velocidad, como de dos códigos que implementen el mismo método. Esta diferencia es la que acota el siguiente principio:

A la hora de medir el tiempo, siempre lo haremos en función del número de operaciones elementales

que realiza dicho algoritmo, entendiendo por operaciones elementales (en adelante OE) aquellas que el ordenador realiza en tiempo acotado por una constante. Así, consideraremos OE las operaciones aritméticas básicas, asignaciones a variables de tipo predefinido por el compilador, los saltos (llamadas a funciones y procedimientos, retorno desde ellos, etc.), las comparaciones lógicas y el acceso a estructuras indexadas básicas, como son los vectores y matrices. Cada una de ellas contabilizará como 1 OE. Resumiendo, el tiempo de ejecución de un algoritmo va a ser una función que mide el número de operaciones elementales que realiza el algoritmo para un tamaño de entrada dado. En general, es posible realizar el estudio de la complejidad de un algoritmo sólo en base a un conjunto reducido de sentencias, aquellas que caracterizan que el algoritmo sea lento o rápido en el sentido que nos interesa. También es posible distinguir entre los tiempos de ejecución de las diferentes operaciones elementales, lo cual es necesario a veces por las características específicas del ordenador (por ejemplo, se podría considerar que las operaciones + y ÷ presentan complejidades diferentes debido a su implementación). Sin embargo, en este texto tendremos en cuenta, a menos que se indique lo contrario, todas las operaciones elementales del lenguaje, y supondremos que sus tiempos de ejecución son todos iguales. Para hacer un estudio del tiempo de ejecución de un algoritmo para los tres casos citados comenzaremos con un ejemplo concreto. Supongamos entonces que disponemos de la definición de los siguientes tipos y constantes:

```
CONST n =...; (* num. maximo de elementos de un vector *);

TYPE vector = ARRAY [1..n] OF INTEGER;
```

y de un algoritmo cuya implementación en Modula-2 es:

```
PROCEDURE Buscar(VAR a:vector; c:INTEGER): CARDINAL;
    VAR j: CARDINAL;
    BEGIN j:=1;                                  (* 1 *)
        WHILE (a[j]<c) AND (j<n) DO              (* 2 *)
            j:=j+1                               (* 3 *)
        END;                                     (* 4 *)
        IF a[j]=c THEN                           (* 5 *)
        RETURN j                                 (* 6 *)
        ELSE RETURN 0                            (* 7 *)
        END                                      (* 8 *)
    END  Buscar;
```

Para determinar el tiempo de ejecución, calcularemos primero el número de operaciones elementales (OE) que se realizan:

– En la línea (1) se ejecuta 1 OE (una asignación).

– En la línea (2) se efectúa la condición del bucle, con un total de 4 OE (dos comparaciones, un acceso al vector, y un AND).

– La línea (3) está compuesta por un incremento y una asignación (2 OE).

– La línea (5) está formada por una condición y un acceso al vector (2 OE).

– La línea (6) contiene un RETURN (1 OE) si la condición se cumple.

– La línea (7) contiene un RETURN (1 OE), cuando la condición del IF anterior es falsa.

Obsérvese cómo no se contabiliza la copia del vector a la pila de ejecución del programa, pues se pasa por referencia y no por valor (está declarado como un argumento VAR, aunque no se modifique dentro de la función). En caso de pasarlo por valor, necesitaríamos tener en cuenta el coste que esto supone (un incremento de n OE). Con esto:

• En el caso mejor para el algoritmo, se efectuará la línea (1) y de la línea (2) sólo la primera mitad de la condición, que supone 2 OE (suponemos que las expresiones se evalúan de izquierda a derecha, y con "cortocircuito", es decir, una expresión lógica deja de ser evaluada en el momento que se conoce su valor, aunque no hayan sido evaluados todos sus términos). Tras ellas la función acaba ejecutando las líneas (5) a (7). En consecuencia, T(n)=1+2+3=6.

Reglas generales para el cálculo del número de OE

La siguiente lista presenta un conjunto de reglas generales para el cálculo del número de OE, siempre considerando el peor caso. Estas reglas definen el número de OE de cada estructura básica del lenguaje, por lo que el número de OE de un algoritmo puede hacerse por inducción sobre ellas.

• Vamos a considerar que el tiempo de una OE es, por definición, de orden 1. La constante c que menciona el Principio de Invarianza dependerá de la implementación particular, pero nosotros supondremos que vale 1.

• El tiempo de ejecución de una secuencia consecutiva de instrucciones se calcula sumando los tiempos de ejecución de cada una de las instrucciones.

• El tiempo de ejecución de la sentencia " IF C THEN S1 ELSE S2 END;" es T =T (C) + max{T (S 1),T (S 2)}.

• El tiempo de ejecución de un bucle de sentencias "WHILE C DO S END;" es

T =T (C) + (nº iteraciones)*(T (S) +T (C)). Obsérvese que tanto T (C) como T (S) pueden variar en cada iteración, y por tanto habrá que tenerlo en cuenta para su cálculo.

• Para calcular el tiempo de ejecución del resto de sentencias iterativas (FOR, REPEAT , LOOP) basta expresarlas como un bucle WHILE.

• El tiempo de ejecución de las llamadas a procedimientos recursivos va a dar lugar a ecuaciones en recurrencia, que veremos posteriormente.

ALGORITMO DE PRIM.

Pseudo código del algoritmo:

La idea básica consiste en añadir, en cada paso, una arista de peso mínimo a un árbol previamente construido. Más explícitamente:

Paso 1. Se elige un vértice u de G y se considera el árbol S = {u}

Paso 2. Se considera la arista e de mínimo peso que une un vértice de S y un vértice que no es de S, y se hace S = S + e

Paso 3. Si el nº de aristas de T es n-1 el algoritmo termina. En caso contrario se vuelve al paso 2

Análisis de complejidad

· **Primera aproximación**

En el paso 2, si S tiene k vértices hay n-k vértices en V-S. Por tanto, necesitamos hallar la arista de mínimo peso entre k(n-k) aristas. Como $k(n-k)<(n-1)2$, el coste resulta $O(n^2)$. Pero el bucle del paso 2 se repite n-1 veces luego la complejidad es $O(n^3)$

· **Segunda aproximación**

Una buena estructura de datos mejora la complejidad. Consideremos las listas S y Z = V-S.

A cada vértice de Z le etiquetamos inicialmente así:

$t(z)=w(uz)$ si existe la arista uz,

$t(z)=\infty$ si no existe

Ahora en el paso 2, se elige el vértice z de Z con etiqueta mínima, se halla $v \in S$ tal que $t(z)= w(vz)$, se añade z a S y se actualizan las etiquetas de los vértices de Z así:

$t(x):=\min\{t(x), w(zx)\}$

Ahora, en cada paso el mínimo se calcula entre n-k etiquetas. Y así el coste total es

$(n-1)+(n-2)+\ldots+1=n(n-1)/2 \in O(n^2)$

ALGORITMO DE KRUSKAL

Pseudo código del algoritmo:

La idea básica consiste en elegir sucesivamente las aristas de mínimo peso sin formar ciclos.

Paso 1. Se elige la arista de mínimo peso e y se considera S={e}.

Paso 2. Sea e' la arista de mínimo peso tal que e'$\notin$S y S + e' es un grafo acíclico. Se hace S=S+e'.

Paso 3. Si S tiene n-1 aristas, el algoritmo termina. En caso contrario se vuelve al paso 2.

Demostración de la validez del algoritmo

En primer lugar se observa que el algoritmo construye un árbol generador. Sea T el árbol construido por el algoritmo de Kruskal y supongamos que no es de peso mínimo. Sea S un árbol generador mínimo que contenga el mayor segmento inicial de la lista de aristas elegidas para T. Llamamos e a la primera arista elegida para T que no está en S. Añadiendo la arista e a S se crea un ciclo C. Este ciclo debe contener una arista e* que no es de T, pues T no contiene ningún ciclo. Consideramos el grafo S* = S + e - e*. Este grafo es un árbol generador. Además cuando el algoritmo elige la arista e, tanto e como e* están disponibles luego $w(e) \leq w(e^*)$. Por tanto S* es un árbol generador con $w(S^*) \leq w(S)$, es decir, es un árbol generador mínimo y contiene una arista más de T que el árbol S (la arista e), en contradicción con la elección de S.

Análisis de la complejidad

El coste del primer paso, ordenación de las aristas según su peso, es $O(|E|\log|E|)$, es decir, $O(|E|\log(|V|))$. En el segundo paso se debe detectar si la arista elegida forma un ciclo con las previamente elegidas. ¿Cómo? Basta tener marcada la componente conexa a que pertenece cada vértice (inicialmente habrá |V| componentes, una por vértice). Si se elige una arista con extremos de la misma etiqueta se está formando un ciclo y si tienen distinta etiqueta no hay ciclo y la elección es correcta. En este caso se deben actualizar las etiquetas para mantener siempre una por cada componente conexa. El coste total de este paso es $O(|E|)$ por la comprobación de la corrección más $O((|V|)^2)$ por las actualizaciones, pues en el peor caso en cada uno de los |V| pasos se deben actualizar las etiquetas de los |V| vértices.

La complejidad total del algoritmo de Kruskal es por tanto $O((|V|)^2+|E|\log(|V|))$

ALGORITMO DE ORDENACIÓN

a. Ordenación por inserción

Pseudocódigo del algoritmo:

La idea del algoritmo es la más intuitiva (la que se haría para ordenar una baraja):

- **Se trabaja con dos listas: A y B, la primera inicialmente vacía, y la segunda con la lista a ordenar B(i).**
- **Se asigna B(1) a A(1).**
- **Para i= 2 hasta n**
  - **Se elige B(i) y se compara con los elementos A(j) (j=1,i-1), para insertarlo en el lugar que corresponde.**
- **Fin de bucle i**

Código MATLAB

```
% procedimiento para ordenar una lista A por inserción
clc
clear all
n=input('Este programita ordena de menor a mayor una lista de
números aleatorios de tamaño n. Introduzca el valor del tamaño de
la lista: ')
A=rand(1,n)
S=size(A);
 for j=2:S(2)
    k=A(j);
    i=j-1;
    while ((i>0) & (A(i)>k))
        A(i+1)=A(i);
        i=i-1;
    end
    A(i+1)=k;
 end
B=A
```

Análisis de la complejidad

El coste mayor tiene lugar en el bucle, en cada paso se realizan tantas comparaciones como elementos tenga la lista A, es decir:

$$1 + 2 + \dots (n-1) = O(n^2)$$

b. Ordenación por mezcla

Pseudocódigo del algoritmo:

La estructura del algoritmo es la siguiente:

- **Se DIVIDE la lista de n elementos en dos sublistas de tamaño n/2.**
- **Se ORDEN cada sublista de manera recursiva.**
- **Se MEZCLAN las dos sublistas.**

Análisis de la complejidad

La complejidad de la mezcla es lineal (n comparaciones, por lo que se puede escribir esta relación de recurrencia en el tiempo de cómputo:

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

Como n sólo puede ser dividido a la mitad $\log_2(n)$ veces, el costo es O(nlog(n))