CAPÍTULO III

Analizando algoritmos en búsqueda de eficiencia

Durante el desarrollo de este capítulo se utilizarán dos términos claves, “ejemplar” y “problema”, que se deben diferenciar correctamente para no cometer errores. Cuando se hace referencia a un ejemplar se refiere a un caso particular de un problema (por ejemplo, determinar si en número 5 es primo). En cambio, cuando se menciona un problema hace referencia a todos los casos particulares comprendidos bajo el mismo problema (por ejemplo, determinar los números enteros primos).

Un algoritmo debe funcionar correctamente en todos los ejemplares o casos del problema que resuelve. Para demostrar que un algoritmo es incorrecto basta con encontrar un ejemplar de dicho problema que no se puede resolver de manera correcta.

¿Qué significa que un algoritmo sea eficiente?

Normalmente, cuando hay que resolver un problema es posible que existan distintos algoritmos adecuados para este y, obviamente, se querrá elegir el mejor. Entonces, ¿cómo elegir entre varios algoritmos para el mismo problema?

Si solo hay que resolver un par de casos de un problema sencillo, seguramente no será de mayor importancia con qué algoritmo se resuelva, seguramente la elección se inclinará hacia uno que sea más rápido programar o a uno que ya esté desarrollado, sin preocuparnos por las propiedades teóricas. Pero si se tienen que resolver muchos casos o el problema es complejo, el proceso de selección del algoritmo deberá ser más cuidadoso y con criterio.

Existen dos enfoques para la selección de un algoritmo:

1. Empírico (o a posteriori) consiste en programar las técnicas competidoras e ir probándolas en distintos casos en la computadora.

2. Teórico (o a priori) consiste en determinar matemáticamente la cantidad de recursos necesarios para cada uno de los algoritmos, en función del tamaño de los casos considerados. Los recursos que más nos interesan son el tiempo de computación o procesamiento (este es el más importante) y el espacio de almacenamiento, principalmente entendida como la memoria principal de la computadora.

A lo largo del libro compararemos los algoritmos a través del enfoque teórico tomando como base sus tiempos de ejecución. En este sentido, se considera eficiente a un algoritmo en función a su velocidad de ejecución.

Hay que tener en cuenta que el tamaño de un ejemplar se corresponde formalmente con el número de bits que se requieren para representarlo en una computadora. Sin embargo, para que el análisis sea más claro y sencillo conviene ser menos formales y utilizar la palabra tamaño para indicar un entero que mida de alguna forma el número de componentes de un ejemplar. Por ejemplo cuando se habla de ordenamiento, normalmente se medirá el tamaño de un ejemplar por la cantidad de ítems que hay que ordenar, ignorando el espacio en bit que se requerirá para almacenar cada uno de estos ítems en la computadora.

El enfoque teórico tiene la ventaja de que no depende ni de la computadora que se esté utilizando (hardware), ni del lenguaje de programación (software), ni siquiera de las habilidades del programador (persona). Además se ahorra el tiempo que se habría invertido en el desarrollo de un algoritmo ineficiente, como el tiempo de computación que se habría desperdiciado para comprobarlo. Y lo más importante, permite medir la eficiencia de un algoritmo para todos los casos de un problema (es decir todos los tamaños).

A diferencia del teórico, el enfoque empírico muchas veces propone realizar las pruebas con ejemplar de tamaño chico y moderado, dado que el tiempo de computación requerido es elevado. Esto es una desventaja, suele suceder que los nuevos algoritmos comienzan a comportarse mejor que sus predecesores cuando aumenta el tamaño del ejemplar.

Como no existe una unidad de medida para expresar la eficiencia de un algoritmo, en su lugar se la expresa según el tiempo requerido por un algoritmo. Se puede decir entonces que el algoritmo para un problema requiere un tiempo del orden de $t(n)$ para una función dada t, si existe una constante positiva c y una implementación del algoritmo capaz de resolver todos los casos de tamaño n en un tiempo que no sea superior a $ct(n)$ unidades de tiempo (la unidad de tiempo es arbitraria, pueden ser años, minutos, días, horas, segundos, milisegundos, etc.).

Esto mismo se puede definir formalmente:

$T(n) = O(f(n))$ si existe una constante c y un valor $n0$ tales que $T(n) <= c\, f(n)$ cuando $n > n0$

Existen ciertos órdenes que se repiten con tanta frecuencia que tienen su propia denominación, como se describe a continuación, y sus funciones de crecimiento se observan en la figura 1 para valores de tamaño de uno a diez –si bien el rango de valores es pequeño– es suficiente para apreciar la diferencia entre los distintos órdenes de complejidad y cómo varía el tiempo de ejecución en base al tamaño de la entrada:

1. en el orden de $O(c)$, o de tiempo constante;

2. en el orden de $O(\log n)$, o de tiempo logarítmico;

3. en el orden de $O(n)$, o de tiempo lineal;

4. en el orden de $O(n \log n)$, o de tiempo casi lineal;

5. en el orden de $O(n^2)$, o de tiempo cuadrático;

6. en el orden de $O(n^3)$, o de tiempo cúbico;

7. en el orden de $O(n^k)$, o de tiempo polinómico;

8. en el orden de $O(c^n)$, o de tiempo exponencial;

9. en el orden de $O(n!)$, o de tiempo factorial.

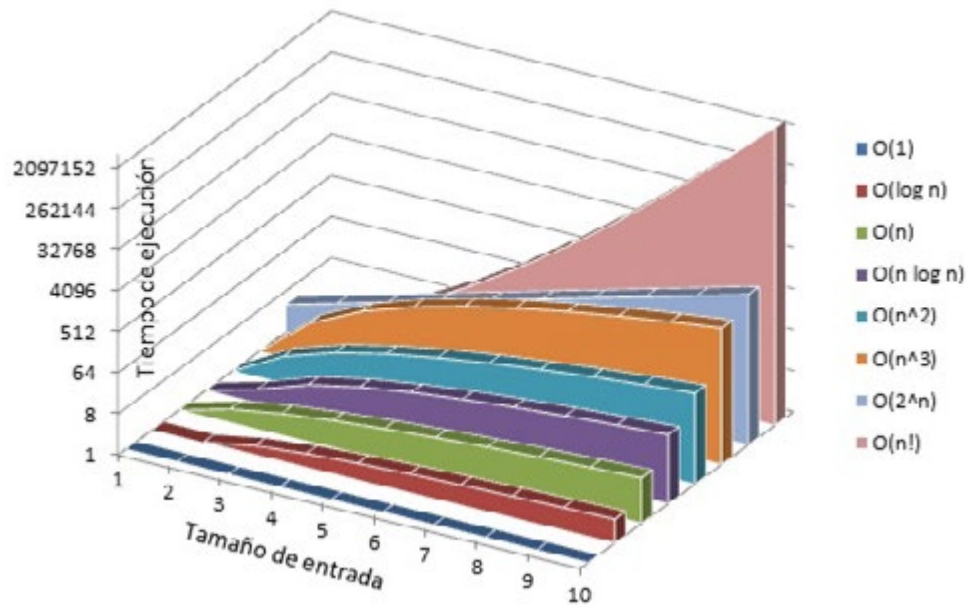

Figura 1. Gráfica de comparación de órdenes de crecimiento de algoritmos

Cuando se comparan dos algoritmos no deben olvidarse las constantes ocultas (o magnitud del orden) es decir un algoritmo en el orden de $O(n)$ es más eficiente que uno en el orden de $O(n2)$, pero si la magnitud del primero es "hora" y la del segundo es "segundos", el segundo algoritmo será más eficiente.

El peor caso: la alternativa más útil

Cuando se analiza un algoritmo se lo puede realizar desde tres enfoques útiles:

1. Mejor caso, poco útil dada la poca o escasa ocurrencia de dicho caso.

2. Caso medio, útil cuando los casos a resolver son muy volátiles e igualmente probables que ocurran (es decir equiprobables). Requiere de mucha información a priori respecto de los casos a resolver. Esto en general es difícil de conocer.

3. Peor caso, es adecuado para problemas críticos donde se debe conocer el tiempo requerido para el peor caso. No requiere información respecto de los casos a resolver.

A lo largo del libro se trabajará con el análisis del peor caso, salvo que se indique lo contrario, esta elección se debe a que este enfoque nos garantiza que en el peor de los casos nuestro algoritmo se ejecutará en ese tiempo –es decir su cota superior–, aunque esto no implica que siempre tendrá dicho tiempo, en algunas ocasiones puede ser mejor. Ahora veamos algunos términos básicos requeridos para comprender el resto del capítulo:

Operación elemental: es aquella que se puede acotar por una constante que depende solamente de la implementación de dicha instrucción en una computadora o lenguaje de programación (por ejemplo una suma, una multiplicación o una asignación). Ya que el tiempo exacto requerido por cada operación elemental no es importante, se simplifica diciendo que estas se pueden ejecutar a coste unitario.

Notación asintótica: se denomina de esta manera porque trata acerca del comportamiento de funciones en el límite, es decir para valores significativamente grandes de sus parámetros. Esta notación nos permite determinar que un algoritmo es preferible respecto a otro incluso en casos de tamaño moderado o grande. Se puede pensar que *n* representa el tamaño del ejemplar sobre el cual es preciso que se aplique un determinado algoritmo, y en *t*(n) representa la cantidad de un recurso dado que se invierte en ese ejemplar para la implementación de un algoritmo particular.

Considere ahora un algoritmo que requiere tres pasos para ser ejecutado, y que dichos pasos requieren un tiempo de *O*(n), *O*(n3) y *O*(n log n), por lo tanto el algoritmo requiere un tiempo de *O*(n + n3 + n log n).

Pero aunque el tiempo requerido por el algoritmo es lógicamente la suma de los tiempos requeridos por cada uno de sus pasos, el tiempo del algoritmo es el mayor de los tiempos requeridos para ejecutar cada uno de los pasos de dicho algoritmo.

O(n + n3 + n log n) = *O*(*máximo*(n, n3, n log n)) = *O*(n3)

Reglas elementales para el análisis de algoritmos

Cuando se dispone de distintos algoritmos para resolver un problema, es necesario definir cuál de ellos es el más eficiente. Una herramienta esencial para este propósito es el análisis de algoritmos. Este análisis suele efectuarse desde dentro del algoritmo hacia afuera, determinando el tiempo requerido por las instrucciones secuenciales y cómo se combinan estos tiempos con las estructuras de control que enlazan las distintas partes del algoritmo. A continuación se presentan las reglas básicas para el análisis de algoritmos:

Operaciones elementales: el coste computacional de una operación elemental es 1 (asignación, entrada/salida, operación aritmética que no desborde el tamaño de la variable utilizada). La mayoría de las instrucciones de un algoritmo son operaciones elementales que están en el orden de *O*(1).

Secuencia: en un fragmento de código, su coste computacional será la suma del coste individual de cada una de sus operaciones o fragmentos de código, aplicando la regla anterior. Es decir si se

cuenta con un fragmento de código compuesto de cinco operaciones elementales, el coste de dicho fragmento será $O(5)$.

Si se tiene dos fragmentos de códigos *F1* de coste α y *F2* de coste β que forman un programa, el coste de dicho programa será la suma de ambos fragmentos $O(\alpha + \beta)$.

Condicional *if-else* (decisión): si g es el coste de la evaluación de un condicional *if*, su valor es de $O(1)$ por cada condición que se deba evaluar (considerando siempre el peor caso, en el que se deba realizar todas las comparaciones dependiendo del operador lógico usado). A esto se le suma el coste de la rama de mayor valor, sea esta la verdadera o falsa (considerando las anidadas si corresponde), aplicando las reglas anteriores.

Es decir el coste de un fragmento condicional es del orden de $O(g$ + *máximo*(rama verdadera, rama falsa)).

Ciclo *for*: en los ciclos con contador explícito (ciclo desde-hasta) que encierran un fragmento de código *F1* de coste α, el coste total será el producto del número de iteraciones del ciclo n por el coste del fragmento de código a ejecutar *F1*, si dentro de dicho fragmento de código no existe presencia de otro ciclo y trabajamos con un n, significativamente grande que varía en función del tamaño de entrada. El coste de dicho fragmento se considera como una constante. Y el mismo se considera despreciable o de $O(1)$ respecto al número de iteraciones del ciclo.

Es decir el coste total estará en el orden de $O(n * O(1))$ lo que significa que es de $O(n)$. Si dentro del *F1* tenemos la presencia de otro ciclo del mismo tipo, el coste total estará en el orden de $O(n * n * O(1))$ lo que significa que es de $O(n^2)$.

Ciclo *while*: en este tipo de ciclos se aplica la regla anterior, solo se debe tener en cuenta que a veces no se cuenta con una variable de control numérica, sino que depende del tamaño de las estructuras con las que se esté trabajando y su dimensión, o del tipo de actividad que se realice dentro de dicho ciclo.

Recursividad: el cálculo del coste total de un algoritmo recursivo no es tan sencillo como los casos que vimos previamente. Para realizar este cálculo existen dos técnicas comúnmente utilizadas: ecuaciones recurrentes y teoremas maestros. Para realizar la primera de ellas se busca una ecuación que elimine la recursividad para poder obtener el orden de dicha función, muchas veces esto no es una tarea sencilla. En cambio, el segundo utiliza funciones condicionales y condiciones de regularidad para realizar el cálculo del coste. Estas técnicas antes mencionadas son avanzadas y exceden los alcances de este libro por lo que solo se explicarán algunos ejemplos utilizando la técnica de ecuaciones recurrentes para determinar el orden de funciones recurrentes.

Ahora se analizará el siguiente ejemplo siguiendo las reglas previamente definidas para determinar el orden del algoritmo “numeros_pares_impares” que se observa en la fiegura 2, dicho algoritmo es iterativo.

```python
def numeros_pares_impares(numero):
    """Muestra los numeros pares e impares."""
    cont_imp = 0
    for i in range(1, numero+1):
        if(i % 2 == 0):
            print(numero, 'Es Par')
        else:
            print(numero, 'Es Impar')
            cont_imp += 1

    print('Cantidad de Números Impares:', cont_imp)
```

Figura 2. Código de algoritmo "numeros_pares_impares"

El análisis de un algoritmo se realiza desde adentro hacia afuera, entonces lo primero que determinamos es el orden del condicional (*if* (i % 2 == 0)) y sus dos ramas la verdadera y la falsa. Entonces según la regla del condicional:

1. solo tenemos una comparación en el *if* *O*(1),

2. la rama verdadera solo tiene una operación elemental *O*(1),

3. la rama falsa tiene dos operación elementales *O*(2).

El resultado del condicional es del orden *O*(3).

Continuamos el análisis con el ciclo *for* (de 1 hasta *n*), según la regla del punto ciclo *for*:

1. el orden del ciclo *for* es de *O*(n),

2. el orden del condicional es despreciable respecto al del ciclo *for* y se considera en el orden de *O*(1).

El resultado del ciclo *for* es del orden *O*(n *1) o sea *O*(n), las otras dos instrucciones restantes del algoritmo (la primera y la última) son operaciones elementales de *O*(1) y se desprecian respecto al orden del ciclo *for*. El resultado final es que el algoritmo "numeros_pares_impares" es del orden de *O*(n).

Entonces se puede decir que la función de complejidad de un algoritmo es una función $f: N \rightarrow N$ tal que $f(n)$ es la cantidad de operaciones que realiza el algoritmo en el peor caso cuando toma una entrada de tamaño *n*.

Algunas observaciones a tener en cuenta:

1. Se mide la cantidad de operaciones en lugar del tiempo total de ejecución (ya que el tiempo de ejecución es dependiente del hardware y no del algoritmo).

2. Interesa el peor caso del algoritmo.

3. La complejidad se mide en función del tamaño de la entrada y no de una entrada en particular.

Ahora resolveremos mediante ecuaciones recurrentes los ejemplos de las funciones factorial, Fibonacci y búsqueda binaria recursivas vistas en el capítulo anterior:

Para el caso de factorial recursivo la ecuación recursiva queda de la siguiente manera, tenemos la llamada recursiva con (n-1) y se suma una constante que es el caso base que contempla una operación elemental: $T(n) = T(n-1) + c$ con $T(0) = 1$.

Ahora resolveremos la ecuación de recurrencia sustituyéndola por su igualdad hasta llegar a una cierta $T(n0)$ conocida para calcular la cota superior:

$T(n) = T(n-1) + c$

$= T(n-2) + c + c$

$= T(n-3) + c + 2c$

$= T(n-k) + k*c$

Cuando $k = n-1$ tenemos que

$T(n) = T(1) + (n-1)*c$

$= 1 + (n-1)$

$= n$

Finalmente obtendremos que la función factorial recursivo es del orden de $T(n) = O(n)$.

Para el caso de Fibonacci recursivo la ecuación queda de la siguiente forma: tenemos las llamadas recursivas (n-1) y (n-2) y se suma constante dado que es el caso base, el mismo es una operación elemental: $T(n) = T(n-1) + T(n-2) + c$ con $T(0) = 0$ y $T(1) = 1$

Ahora debemos resolver la ecuación de recurrencia igual que en caso anterior:

$T(n) = T(n-1) + T(n-2) + c$

$= 2T(n-1) + c$

$= 2(2T(n-2) + c) + c$

$= 2^2T(n-2) + 2^2*c$

$= 2^kT(n-k) + 2^k*c$

Cuando k = n-1 tenemos que

$T(n) = 2^{n-1}T(0) + 2^{n-1}*c$

$= 1 + (n-1)$

$= 2^{n-1}$

Entonces podemos establecer que Fibonacci recursivo es del orden de $T(n) \cong 2^{n-1}$.

Finalmente veamos el caso de la búsqueda binaria recursiva. La ecuación recursiva para la misma quedaría de la siguiente forma: $T(n) = T(n/2^i) + c$ con $T(n<=1) = 1$ y la variable *i* que toma valores de 1 a log n como se demuestra a continuación.

Ahora pasemos a resolver la ecuación recursiva:

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

$= T(n/4) + c + c$

$= T(n/8) + c + 2c$

$= T(n/2^k) + k*c$

Podemos determinar entonces una ecuación que nos indique el valor de k, es decir, cuantas veces tenemos que particionar el vector hasta que solo quede un elemento para comparar:

$1 = n/2^k$

$1 * 2^k = n$

$2^k = n$

$\log_2 2^k = \log n$

$k \log_2 2 = \log n$

$k * 1 = \log n$

$k = \log n$

Como la ecuación $n/2^k = 1$ se resuelve en tiempo $k = \log_2 n$, nos queda que

$T(n) = T(n/2^k) + k*c$

$= T(1) + \log_2 n$

Entonces de esta manera obtenemos que la función búsqueda binaria recursiva es del orden de $T(n) \cong \log n$, dado que si n no es múltiplo de dos no obtendremos un numero entero como resultado.

Como puede observase, el cálculo de la complejidad de una función recursiva no es una actividad trivial. Puede realizarse de distintas manera pero no siempre de forma clara y sencilla –como en una función iterativa–. Es más bien una tarea bastante artesanal que requiere mucho criterio y experiencia por parte de quien la realice.