Introducción a la Algoritmia (III): Ejemplos de análisis de costes

aprende.olimpiada-informatica.org/algoritmia-introduccion-3-costes-ejemplos

En el manual anterior introdujimos la notación O grande para especificar la eficiencia de un algoritmo. La definición rigurosa de esta notación puede encontrarse aquí, pero para trabajar con ella sólo es necesario tener la idea de que representa una cota superior de cómo es el "crecimiento" del número de operaciones que realiza un programa en relación con el tamaño de la entrada.

Vamos a poner algunos ejemplos:

Ejemplo 1: Dado un texto de longitud n y una palabra de longitud m, determinar si el texto contiene esa palabra.

Un posible algoritmo para resolver este problema, implementado en C++, sería el siguiente:

```
bool buscar_palabra(string texto, string palabra) {
    int n = texto.length();
    int m = palabra.length();
        for (int i=0; i < n; ++i) {
        int j = 0;
        while (j < m and i+j < n and texto[i+j] == palabra[j]) {
            ++j;
        }
        if (j == m) {
            return true;
        }
    }
    return false;
}
```

```
def buscar_palabras(texto, palabra):
    n = len(texto)
    m = len(palabra)
    for i in range(n):
        j = 0
        while j < m and i+j < n and texto[i+j] == palabra[j]:
            j = j+1
        if j == m
            return true
    return false
```

Esta función lo que hace es ir recorriendo el texto carácter por carácter y comprobando si en ese carácter empieza la palabra que buscamos. Utilizamos un bucle de n iteraciones para recorrer el texto y dentro de ese bucle hay otro bucle que realiza hasta m iteraciones mientras va comparando los caracteres del texto con los caracteres de la palabra que buscamos. Por tanto, el número de operaciones total es $\mathcal{O}(nm)$. (En general, si

un bucle realiza $\mathcal{O}(A)$ iteraciones y el código del cuerpo del bucle tiene complejidad $\mathcal{O}(B)$, la complejidad total es $\mathcal{O}(AB)$. En el caso de un bucle dentro de otro, los números de iteraciones se multiplican).

En realidad, cuando este algoritmo procese textos “naturales”, el bucle interior se interrumpirá muchas veces bastante antes de llegar a m iteraciones, porque el principio de la palabra que buscamos generalmente no coincidirá con la mayoría de las palabras del texto. Sin embargo, en el contexto de la OIE es importante pensar en el peor caso al que se enfrentará el algoritmo y no sólo en el caso promedio. Si, por ejemplo, el texto es “AAAAAAAAAAAAAAAAAAAAAAAB” y buscamos la palabra “AAAAAAB”, el bucle interior se ejecutará siempre casi hasta el final, haciendo que en efecto el número de operaciones sea del orden de $n \cdot m$.

Podemos observar que el bucle interior no llegará a ejecutarse hasta el final cuando estemos buscando en los últimos $m-1$ caracteres. De hecho, como en esos caracteres no vamos a encontrar la palabra que buscamos, podemos sustituir la condición $i < n$ del bucle principal por $i < n-m$. Entonces, en total de iteraciones en el peor caso sería $(n-m)m$, y podríamos decir que el algoritmo es $\mathcal{O}((n-m)m)$... Pero es preferible seguir diciendo $\mathcal{O}(nm)$. Formalmente ambas opciones tienen sentido, pero a la hora de calificar la eficiencia de un algoritmo informalmente no se suelen incluir restas dentro de la notación O: queremos conocer una cota superior de cómo crece el coste temporal del algoritmo en función del tamaño de la entrada cuando el tamaño de la entrada es grande y $\mathcal{O}(nm)$ expresa esa información de forma concisa; el hecho de que si m es cercano a n entonces el algoritmo tarda menos tiempo no es una información tan relevante en este caso.

Nota: Hay un algoritmo (conocido como algoritmo de Knuth-Morris-Pratt) que soluciona el problema de buscar una palabra en un texto con complejidad temporal (en el peor caso) de $\mathcal{O}(n+m)$. No es un algoritmo muy complicado: invitamos al lector a buscarlo y a tratar de entender cómo funciona o incluso a intentar pensar por sí mismo cómo se podría hacer (si el lector quiere enfrentarse a un reto).

Ejemplo 2: Dado un vector de n enteros (positivos o negativos), encontrar la suma máxima que se puede conseguir sumando términos consecutivos.

Aquí vemos una posible solución al problema.

```
int suma_maxima = v[0];
for (int i = 0; i < n; ++i) {
    for (int j = i; j < n; ++j) {
        int suma_actual = 0;
        for (int k=i; k <= j; ++k) {
            suma_actual += v[k];
        }
        if (suma_actual > suma_maxima) {
            suma_maxima = suma_actual;
        }
    }
}
```

```
suma_maxima = v[0]
for i in range(n):
    for j in range(i, n):
        suma_actual = 0;
        for k in range(i, j+1):
            suma_actual += v[k]

        if suma_actual > suma_maxima:
            suma_maxima = suma_actual
```

Lo que hace este código es calcular la suma de todos los intervalos $[i, j]$ e ir almacenando la mayor de todas en una variable. Analizar la complejidad temporal de este algoritmo no es tan fácil porque ahora los bucles no realizan un número fijo de iteraciones cada vez. El número total de iteraciones que realizan los bucles es:

\[\sum_{i=0}^{n-1} \sum_{j=i}^{n-1} (j-i+1) \]

Utilizando las fórmulas $\sum_{i=0}^n i = \frac{n(n+1)}{2}$ y $\sum_{i=0}^n i^2 = \frac{n(n+1)(2n+1)}{6}$, podemos obtener que el número de iteraciones total es:

\[\sum_{i=0}^{n-1} \sum_{j=i}^{n-1} (j-i+1) = \sum_{i=0}^{n-1} \frac{(n-i)(n-i+1)}{2} = \frac{(n+1)(n+2)(n+3)}{6} \]

Como para la notación O grande sólo nos importa el término dominante, la complejidad del algoritmo es $\mathcal{O}(n^3)$. En general, debido a que $\sum_{i=0}^n i^k = \mathcal{O}(n^{k+1})$, cuando tenemos m bucles for anidados que iteran sobre variables con valores de 0 a n la complejidad suele ser $\mathcal{O}(n^m)$ aunque los valores en los que empieza o acaba la itera una variable estén definidos por variables anteriores, porque al sumar el número total de iteraciones vemos que sólo nos ahorramos un factor constante.

Ejercicio: Encuentra una solución a este problema de complejidad $\mathcal{O}(n)$.

Ejemplo 3: Dado un vector de n números en orden creciente, determinar si un número dado pertenece al vector.

La solución a este problema es la idea de búsqueda dicotómica que explicamos en el primer manual introductorio. Una implementación posible sería la siguiente:

```
int numero; // el número que buscamos
bool encontrado = false;
int l = 0;
int r = n-1;
while (r > l and not encontrado) {
    int m = (l + r)/2;
    if (v[m] > numero) {
        r = m-1;
  }
  else if (v[m] == numero) {
    encontrado = true;
  }
  else {
    l = m+1;
  }
}
if (v[l] == numero) {
  encontrado = true;
}
```

```
numero = 0 #el número que buscamos
encontrado = false
l = 0;
r = n-1;

while r > l and not encontrado:
    m = (l + r)//2
    if v[m] > numero:
        r = m-1
    elif v[m] == numero:
        encontrado = true
    else:
        l = m+1

if v[l] == numero:
  encontrado = true
```

Aquí, siempre sabemos que el número tiene que estar en el intervalo $[l, r]$, que vamos disminuyendo a la mitad de longitud en cada iteración. La complejidad de este algoritmo es $\mathcal{O}(\log n)$.

Ejemplo 4: (Clasificatorio OIE 2018) ¿Cuántas palabras de longitud n con las letras a y b no contienen la palabra abbaaaba?

Una forma de resolver este problema es generar todas las palabras recursivamente (si no se conoce este concepto, véase: Manual de recursividad; también Manual de búsqueda completa y backtracking) y comprobar si cada palabra contiene abbaaaba o no. Hay un total de 2^n palabras y la comprobación de cada palabra se hace en $\mathcal{O}(n)$, así que la complejidad total es $\mathcal{O}(n \cdot 2^n)$.

A parte de contar lo que se tarda una vez se tienen generadas las palabras, también habría que contar la complejidad asociada a la generación de las palabras con el método recursivo. Pero el resultado sigue siendo $\mathcal{O}(2^n)$: si nos imaginamos el árbol generado por la recursividad, el número de llamadas a la función recursiva es $1 + 2 + 4 + \ldots + 2^n = 2^{n+1}-1 = \mathcal{O}(2^n)$.

Ejercicio: (para quienes conozcan los conceptos de recursividad y backtracking y quieran practicar con ellos) Es posible mejorar la complejidad de la solución a $\mathcal{O}(2^n)$ modificando la recursividad para no generar palabras no válidas en lugar de comprobar la validez de las palabras una vez se han generado. Intentad implementar una solución así.

Nota: La solución que obtenía la puntuación máxima de este problema tiene complejidad $\mathcal{O}(\log n)$, utilizando métodos más avanzados.

Ejemplo 5: Dado un vector de números, ordenar el vector ascendentemente.

Aquí hay una posible solución a este problema.

```
for (int i = 0; i < n; ++i) {
  for (int j = i+1; j < n; ++j) {
    if(v[i] > v[j]) swap(v[i], v[j]);
  }
}
```

```
for i in range(n):
  for j in range(i+1, n):
    if v[i] > v[j]:
        swap(v[i], v[j])
```

Este código en la primera iteración del bucle exterior pone el elemento mínimo en la posición v[0] (porque recorre todo el vector y si encuentra alguno menor que el que hay en la posición v[0] lo mueve allí), en la segunda pone el segundo elemento mínimo en la posición v[1], en la tercera el tercero menor en la v[2], y así hasta que el vector queda ordenado. La complejidad total es $\mathcal{O}(n^2)$, ya que tiene dos bucles anidados con variables que iteran hasta n (como hemos visto en el ejemplo 3, el hecho de que la variable j empiece desde i+1 no afecta a la complejidad total: podemos calcular que el número total de iteraciones es $(n-1) + (n-2) + \ldots + 1 = \frac{n(n-1)}{2} = \mathcal{O}(n^2)$). Este código se puede optimizar reduciendo la cantidad de operaciones swap si en cada iteración primero se calcula el mínimo en el intervalo $[i, n-1]$ y después se hace un swap para colocarlo en la posición i, pero eso no cambia la complejidad total.

Este es un problema de algoritmia clásico. La solución que se muestra en el código se conoce como Selection Sort, pero hay muchas más. Y las hay más eficientes, de hecho: en el siguiente manual explicaremos una solución cuya complejidad es $\mathcal{O}(n \log n)$. Lo más interesante de todo, sin embargo, es que es posible demostrar de forma

relativamente sencilla que, para el problema de ordenación en su forma más general, $\mathcal{O}(n \log n)$ es la mejor complejidad que se puede obtener para un algoritmo que resuelva el problema. Lo explicaremos en el siguiente manual.

Añadir nuevo comentario

Acerca de formatos de texto

Texto sin formato

- No se permiten etiquetas HTML.
- Saltos automáticos de líneas y de párrafos.
- Las direcciones de correos electrónicos y páginas web se convierten en enlaces automáticamente.