Introducción a la Algoritmia (II): Costes

aprende.olimpiada-informatica.org/algoritmia-introduccion-2-costes

En el manual anterior hemos introducido el concepto de algoritmo y hemos planteado algunos problemas “algorítmicos” en abstracto. Aquí vamos a ser un poco más técnicos: explicaremos qué tipo de problemas se plantean en la OIE y qué determina la eficiencia de un algoritmo que se ejecuta en un ordenador.

En la OIE y en la IOI hay tres tipos de problemas:

- Problemas **clásicos** (en la IOI los llaman *batch problems*), en los que se pide escribir un programa que dada una entrada con los datos del problema produzca una salida con la solución del problema. El concursante envía su programa a un sistema llamado juez, que comprueba que el programa produzca la solución correcta ante una serie de casos privados. Estos son los problemas más frecuentes, sobre todo en la OIE.

- Problemas **interactivos** (según la IOI, *communication problems*). Este tipo de problemas se parece más a los que discutimos en el manual anterior. Se pide un programa que *interactúe* con el juez, normalmente haciendo un número limitado de preguntas o consultas para tratar de obtener información que resuelva el problema. Un ejemplo sería el problema de adivinar el número secreto preguntando si es mayor o menor que un número dado.

- Problemas de **solo salida** (*output-only*): en estos problemas al concursante se le dan los casos de prueba y él tiene que entregar las soluciones a esos casos, no un programa. Suelen ser problemas en los que hay que “experimentar” con las propiedades que tiene cada caso concreto y normalmente no se resuelven con algoritmos que dan la solución exacta en general, sino con “aproximaciones” que se adaptan a cada caso. Este tipo de problemas es bastante raro y no se discutirá aquí.

En los problemas clásicos e interactivos, el programa que envía el concursante se ejecuta en un servidor y los recursos que el programa puede consumir (tiempo y memoria) están limitados. En los problemas interactivos estos límites no suelen ser un problema y están para que no se pueda saturar el juez, porque la dificultad principal del problema está en hacer el mínimo número de consultas (hay algunos casos en los que calcular las consultas que hay que hacer de forma eficiente es un problema en sí, pero no es común). Sin embargo, en los problemas clásicos conseguir que el programa se ejecute dentro de esos límites de tiempo es el principal objetivo, y de hecho es el criterio más importante para evaluar el problema, ya que en la mayoría de problemas las puntuaciones parciales se otorgan dependiendo de la eficiencia del problema (en algunos casos también se dan puntos por resolver “casos particulares”).

En el ordenador, todas las operaciones básicas y accesos a memoria requieren un tiempo que depende de la frecuencia de ciclos del procesador. También, el almacenamiento de variables en memoria ocupa un espacio determinado. Para la mayoría de los casos en los que un programa trata una cantidad pequeña de datos (como por ejemplo, cuando comprobamos si da la solución correcta para casos en los que podemos calcular la solución a mano), el tiempo que se tarda en hacer los cálculos es imperceptible y el uso de memoria es insignificante. Sin embargo, muchos problemas son probados en el juez con casos grandes, en los que el programa puede llegar a tener que procesar millones de números. Para estos casos, es importante diseñar el algoritmo de forma que se minimice el número de operaciones posibles y, en concreto, tenemos que reducir cómo crece el número de operaciones con el tamaño de la entrada. Pondremos un ejemplo para ilustrar mejor la idea:

Ejemplo 1: Dado un entero n mayor que 1, determinar si n es primo.

Un número es primo si solo es divisible por 1 y por él mismo: a partir de esta definición, podemos idear este algoritmo sencillo, que prueba cada divisor entre 2 y $n-1$:

```
bool is_prime(int n) {
    for (int i=2; i < n; ++i) {
        if (n % i == 0) return false;
    }
    return true;
}
```

```
def is_prime(n):
    for i in range(1, n):
        if n%i == 0:
            return false
    return true
```

Vamos a contar las operaciones que se hacen en este algoritmo (en el “peor caso”, es decir, si n es primo): si miramos la cabecera del bucle, la comparación (i < n) se realiza $(n-1)$ veces y el incremento ++i se realiza $(n-2)$ veces. El cuerpo del bucle consiste en una operación módulo y una comparación con 0, que se ejecutan $(n-2)$ veces. En total, podríamos decir que se hacen $(n-1)+(n-2)+2 \cdot (n-2) = 4n-7$ operaciones. Contar operaciones de esta forma no es demasiado fiable: en primer lugar, no podemos suponer que operaciones distintas van a tardar el mismo tiempo (en concreto, la operación módulo es más costosa que un incremento). Además, como no sabemos cómo el compilador va a implementar las operaciones internamente (es posible que realice optimizaciones, que junte dos operaciones en una, etcétera) no podemos asegurar que nuestro conteo sobre el lenguaje en el que programamos se corresponda realmente con el número de operaciones que se realizan en el procesador. Por último, contar operaciones de esta forma para algoritmos más complicados es más difícil. En seguida aprenderemos que no vamos a tener que calcular el número de operaciones que realiza nuestro programa cada vez, sino que nos

va a bastar con saber de qué *orden* es el número de operaciones que va a hacer el programa. Pero antes de discutir eso, vamos a ver unas posibles optimizaciones del algoritmo que hemos pensado para determinar si un número es primo.

En primer lugar, podemos pensar que si n no es par, entonces no es divisible por ningún número par, y no tenemos por qué comprobar esos números (si n es par, entonces no es primo a no ser que sea igual a 2). Consideremos este algoritmo:

```
bool is_prime(int n) {
    if (n % 2 == 0 and n > 2) return false;
    for (int i=3; i < n; i += 2) {
        if (n % i == 0) return false;
    }
    return true;
}
```

```
def is_prime(n):
    if n > 2 and n%2 == 0: return false
    for i in range(3, n, 2):
        if n%i == 0: return false
    return true
```

Este algoritmo realiza la mitad de iteraciones del bucle que el anterior, por lo tanto hace (aproximadamente) la mitad de operaciones y por lo tanto tarda la mitad de tiempo. Esto se podría considerar un avance formidable: reducir a la mitad el coste de tiempo es bastante impresionante, pero aún se puede hacer mejor. Estamos comprobando si n es divisible por números mayores que $\frac{n}{2}$, cosa que obviamente nunca ocurrirá. El siguiente algoritmo elimina esas comprobaciones innecesarias:

```
bool is_prime(int n) {
    if (n % 2 == 0 and n > 2) return false;
    for (int i=3; i <= n/2; i += 2) {
        if (n % i == 0) return false;
    }
    return true;
}
```

```
def is_prime(n):
    if n > 2 and n%2 == 0: return false
    for i in range(3, n/2+1, 2):
        if n%i == 0: return false
    return true
```

Ahora hemos vuelto a reducir una vez más a la mitad el número de operaciones, así que este algoritmo tarda una cuarta parte de lo que tardaba el original. Puede parecer una mejora muy impresionante, pero aún se puede hacer mucho mejor. Para ello, utilizaremos una propiedad: si n es compuesto, entonces tiene un divisor entre 2 y $\sqrt{n}$. Esto es sencillo de comprobar: si n se descompone en dos factores, si uno es mayor que $\sqrt{n}$, entonces el otro tiene que se ser menor que $\sqrt{n}$ (o bien los dos son iguales a $\sqrt{n}$). Podemos modificar el algoritmo inicial para aprovechar esta propiedad:

```
bool is_prime(int n) {
    for (int i=2; i*i <= n; ++i) {
        if (n % i == 0) return false;
    }
    return true;
}
```

```
def is_prime(n):
    if n < 2: return false
    i = 2
    while i*i <= n:
        if n%i == 0: return false
        i = i+1
    return true
```

Si se piensa detenidamente, esta optimización es significativamente distinta a las dos anteriores. Antes, habíamos reducido el número de operaciones en un factor constante, pero el *crecimiento* del número de operaciones seguía siendo el mismo, era un crecimiento "lineal" (si se duplicaba n, el número de operaciones también se duplicaba). Ahora, sin embargo, el número de operaciones es proporcional a la raíz cuadrada de n. Podemos ver que esto supone una gran diferencia para números muy grandes. Podríamos introducir en este algoritmo la optimización de no considerar números pares para reducir las operaciones a la mitad otra vez, pero si lo pensamos nos damos cuenta de que, para números grandes, la mayor parte de la "optimización" es debido al hecho de que sólo consideramos números hasta la raíz cuadrada de n.

Para verlo en términos más concretos, vemos el tiempo en milisegundos que tardan (en un procesador de PC "normal") los diferentes códigos que hemos escrito antes para determinar que diferentes números son primos. La primera fila es el código sin optimizaciones, la segunda es con las dos optimizaciones de paridad y mirar divisores hasta $\frac{n}{2}$ y la tercera mirando divisores hasta $\sqrt{n}$:

	n = 10000019	**n = 100000007**	**n = 1000000007**
A	42ms	326ms	3217ms
B	10ms	96ms	877ms
C	~0ms	~0ms	~0ms

Se ve claramente que el segundo programa es unas cuatro veces más rápido que el primero, pero claramente el ganador es el tercero: aunque lleguemos a valores muy altos de n, su tiempo de ejecución sigue siendo imperceptible. Para que el tiempo de ejecución llegara a acercarse a un segundo, tendríamos que dar un n primo del orden de 10^{18} (se sale del tipo int de C++, pero se puede hacer con el tipo long long), valor para el cual el programa B tardaría unos 30 años en ejecutarse.

Esta es la idea con la que nos tenemos que quedar: es mucho más importante optimizar el "crecimiento" del número de operaciones respecto a la entrada que mejorar el número de operaciones en un factor constante. A esto que llamamos "crecimiento" se le suele llamar **complejidad algorítmica** y se denota con la siguiente notación, que se llama "notación O grande": el algoritmo inicial y sus dos optimizaciones diríamos que tienen una complejidad $\mathcal{O}(n)$, porque el número de operaciones es proporcional a n, pero el algoritmo con la última optimización tiene una complejidad $\mathcal{O}(\sqrt{n})$. (La notación O grande no necesariamente denota una relación de proporcionalidad directa, pero el concepto es parecido. En el siguiente manual explicaremos con más detalle esta notación y exploraremos más ejemplos para conocerla mejor. Quien esté interesado en la definición matemáticamente formal de la notación O puede consultar la página de Wikipedia) En la OIE, un programa que tenga la complejidad algorítmica adecuada obtendrá todos los puntos aunque su "factor constante" no esté totalmente optimizado. Normalmente, hay suficiente tiempo como para que un programa entre dentro del límite aunque haga el doble de operaciones de las estrictamente necesarias. Sin embargo, en un programa que tenga una complejidad algorítmica no óptima el número de operaciones se disparará irremediablemente para casos grandes y no conseguirá pasar todos los casos dentro del límite.

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.