Cómo desarrollar una solución recursiva

Introducción

La recursividad es una técnica de solución de problemas muy útil en según qué situaciones. En este documento se muestra cómo plantear diversas soluciones recursivas a un problema. Concretamente, nos planteamos, como ejemplo, desarrollar un método que calcule la potencia a^n, siendo *a* un número real y *n* un número natural, usando operaciones aritméticas básicas.

Lo primero que haremos será establecer la interfaz del método a desarrollar, ya que los datos de entrada del problema serán los que nos ayuden a determinar cómo se calcula su tamaño y, a partir de ahí, distinguir entre el caso base y el caso general:

```
/**
 * @param a - base
 * @param n - exponente
 * @return a elevado a n
 */
public static float pow(float a, int n)
```

El método, tal como establece el enunciado, requiere dos parámetros, uno de tipo *float* (real, también podría haber sido *double* si se requiere más precisión) y otro de tipo *int* (los números naturales son el subconjunto de los enteros no negativos). El resultado será de tipo *float*, ya que será también un número *real*. La interfaz del método, por sí misma solo establece la forma de usarlo, y no implica qué tipo de solución se va a implementar, pudiendo existir múltiples opciones, tanto recursivas como no recursivas (iterativas).

Desarrollo de una solución simple

El primer paso para desarrollar cualquier solución es analizar cuidadosamente el problema. Cuando lo que busquemos sea una solución recursiva, ese análisis debe hacerse teniendo en mente encontrar, lo primero, la forma de determinar el tamaño del problema.

El cálculo de la potencia tiene dos datos de entrada: *a*, que es un número real, y *n*, que es un número natural. Tanto los números reales como los naturales son conjuntos y ordenables; pero mientras que

todo número real tiene siempre otro más pequeño que él, en el conjunto de los números naturales hay uno que es el más pequeño de todos: el cero. Ello nos hace plantearnos la hipótesis de que sea *n* el que determine el tamaño del problema.

Por otra parte, el enunciado planteaba resolver el problema usando operaciones aritméticas básicas, y la potencia puede calcularse como una secuencia de multiplicaciones. Teniendo todo esto en cuenta, nos planteamos varios ejemplos, empezando con el valor más pequeño posible de *n*, e incrementándolo sucesivamente:

$a^0 = 1$
$a^1 = a = a * 1 = a * a^0$
$a^2 = a * a = a * a^1$
$a^3 = a * a * a = a * (a * a) = a * a^2$
$a^4 = a * a * a * a = a * (a * a * a) = a * a^3$
$a^5 = a * a * a * a * a = a * (a * a * a * a) = a * a^4$
$a^6 = a * a * a * a * a * a = a * (a * a * a * a * a) = a * a^5$

En cada caso intentamos replantear la solución para expresarla en términos de una solución ya desarrollada para un valor de n más pequeño (a^6 lo expresamos en función de a^5, a^5 en función de a^4, a^4 en función de a^3 etc.). Generalizando, para cualquier valor de *n* tendríamos que:

$$a^n = a * a^{n-1}$$

El único caso para el que esta generalización no es aplicable es, como era de esperar, para el valor de *n* más pequeño, 0, ya que no existe un *n-1* en función del cual expresar la solución. Por tanto, ya tenemos los datos necesarios para una solución recursiva:

1. El tamaño del problema (*n*)
2. El caso base (*n* igual a cero), con su solución específica (1)
3. La solución del caso general en función de casos más pequeños del mismo problema ($a^n = a * a^{n-1}$, para todo $n > 0$)

Podemos implementar el método con estos datos:

```
public static float pow(float a, int n) {
    if (n == 0) { // Caso base
        return 1;
    } else { // Caso general
        return a * pow(a, n - 1);
    }
}
```

Podría plantearse una ligera mejora de este algoritmo si tenemos en cuenta que, cuando *n* vale 1, la llamada recursiva que se produce con *n* igual a *0* no aporta nada a la solución, ya que da lugar a una multiplicación por 1 que nos deja con el valor de *a*, que ya teníamos. Podemos ahorrar esa última llamada recursiva si añadimos *n == 1* como otra condición de base; a costa de incluir una segunda pregunta en el método. Para minimizar el coste que añade esa pregunta, podemos reordenar el tratamiento de los casos, jugando con que la probabilidad de que *n* sea mayor que 1 es, en circunstancias normales, significativamente mayor que la de que no lo sea.

```
public static float pow(float a, int n) {
    if (n > 1) { // Caso general
        return a * pow(a, n - 1);
    } else if (n == 1) { // Caso base
        return a;
    } else{ // Caso base n == 0
        return 1;
    }
}
```

De esta manera, en el caso general se sigue haciendo una sola pregunta, como en la primera versión; cuando *n* sea igual a 1, se ejecuta una segunda pregunta que sustituye a la llamada recursiva de la primera versión; y solo cuando n sea cero en la primera llamada se llegarán a realizar dos preguntas (si *n* no es igual a cero en la primera llamada, no llegará a serlo, ya que se terminará en el caso base *n == 1*).

Desarrollo de una solución equilibrada

La solución mostrada en el apartado anterior muestra una recursividad lineal en la que, en el caso general, cada problema se descompone en un subproblema ligeramente más pequeño:

$$a^6 = a * a * a * a * a * a = a * (a * a * a * a * a) = a * a^5$$

Generalmente, vale la pena indagar la posibilidad de obtener soluciones no lineales en las que cada problema se descomponga en dos o más subproblemas de tamaño similar y mucho menor que el original; subproblemas que, por tanto, convergen más rápidamente hacia el caso base y que, incluso, podrían ser paralelizables, si son independientes entre sí.

En el caso que nos ocupa, podemos observar que la solución consiste en una secuencia de multiplicaciones, de las que la solución recursiva separa una, para obtener un subproblema con un exponente un grado menor.

En lugar de hacer eso, podemos probar a dividir la secuencia de multiplicaciones por la mitad:

$$a^6 = a * a * a * a * a * a = (a * a * a) * (a * a * a) = a^3 * a^3$$

Igualmente, separamos una multiplicación, pero para dividir la secuencia de multiplicaciones en dos partes iguales, que coinciden con el resultado, para un tamaño del problema, exactamente la mitad que el original. Podemos comprobar esta aproximación para diferentes tamaños del problema:

$$a^5 = a * a * a * a * a = (a * a) * (a * a * a) = a^2 * a^3$$
$$a^4 = a * a * a * a = (a * a) * (a * a) = a^2 * a^2$$
$$a^3 = a * a * a = (a) * (a * a) = a^1 * a^2$$

Comprobamos que cuando *n* es par obtenemos dos subproblemas que son exactamente de la mitad del tamaño que el original, pero cuando *n* es impar, al no ser su mitad un número entero, obtenemos un subproblema cuyo tamaño es igual al redondeo superior[1] de la mitad del original ($\lceil n/2 \rceil$), y otro, cuyo tamaño es igual al redondeo inferior de dicha mitad ($\lfloor n/2 \rfloor$).

Para trasladar esta forma de solucionar el problema al método, basta con modificar el caso general:

```
public static float pow(float a, int n) {
    if (n > 1) { // Caso general
        return pow(a, n / 2) * pow(a, (n + 1) / 2);
    } else if (n == 1) { // Caso base
        return a;
    } else{ // Caso base n == 0
        return 1;
    }
}
```

Desarrollo de una solución optimizada

La solución "equilibrada" no lineal desarrollada en el apartado anterior no resulta, sin embargo, eficiente en comparación con la solución lineal simple. Es cierto que, en cada paso el tamaño del problema se divide por 2, frente a la solución lineal en que solo disminuye en 1; pero, al mismo tiempo, el número de subproblemas también se multiplica por 2, lo que da lugar a un crecimiento exponencial de las llamadas recursivas. La Ilustración 1 muestra cómo, para *n* igual a 11, la solución no lineal realiza un total de 21 llamadas, incluida la inicial, mientras que la solución lineal simple realizaría solo 11. La mayoría de esas llamadas son para calcular casos que también son calculados por otras llamadas.

[1] El redondeo superior de un número real es el número entero superior más próximo $\lceil 3{,}5 \rceil = 4$. El redondeo inferior es el número entero inferior más próximo ($\lfloor 3{,}5 \rfloor = 3$).

En la Ilustración 1, 11 de las llamadas calculan a^1, 4 calculan a^2, y 3 calculan a^3. Solo a^{11}, a^5 y a^6 se calculan una sola vez durante el proceso.

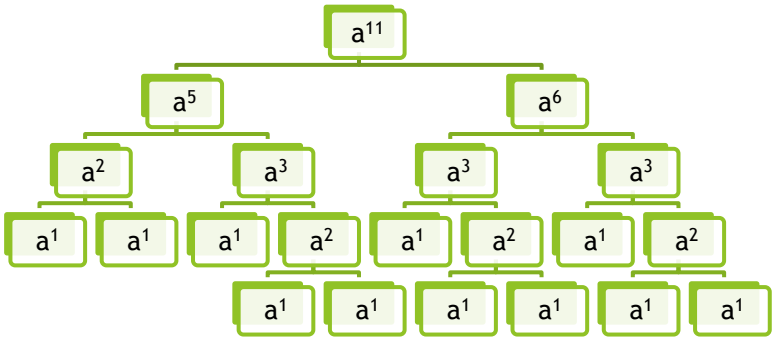

Ilustración 1

Si nos fijamos en lo que ocurre en el caso general:

```
return pow(a, n / 2) * pow(a, (n + 1) / 2);
```

nos damos cuenta de que, cuando *n* es par, *n/2* y *(n + 1) / 2* tienen el mismo valor, con lo que podríamos evitar la segunda llamada simplemente almacenando el valor de la primera en una variable y reutilizándolo:

```
int pown_2 = pow(a, n / 2);
return pown_2 * pown_2;
```

Cuando *n* es impar, el valor de *(n + 1) / 2* es 1 más que el de *n/2*, con lo que habría que realizar una multiplicación más:

```
int pown_2 = pow(a, n / 2);
float result = pown_2 * pown_2;

if (n % 2 != 0) {
    result = result * a;
}

return result;
```

Con esta modificación el número de llamadas se reduce drásticamente, como muestra la Ilustración 2, al evitar hacer llamadas repetidas para calcular el mismo paso. Formalmente, la solución lineal simple inicial requiere siempre n llamadas; la solución equilibrada, no lineal, puede requerir hasta *2 * n + 1* llamadas; y la solución optimizada final, que también es lineal, requiere, en el peor caso $\lfloor(\log_2 n) + 1\rfloor$.

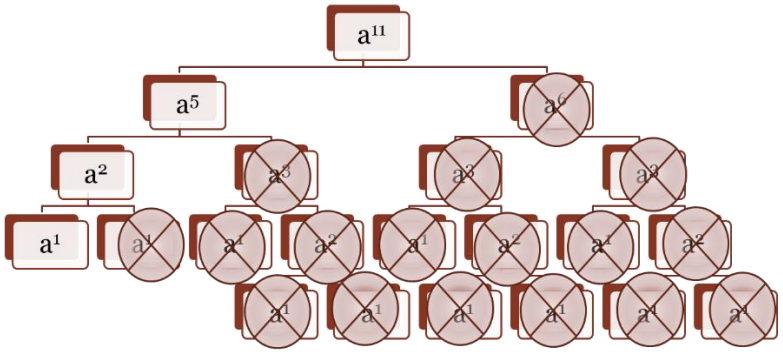

Ilustración 2

En este ejemplo, hemos alcanzado una solución lineal que llega a ser mejor que la inicial, pero no debemos perder de vista que el esfuerzo invertido en la solución equilibrada no ha sido en vano, ya que es la que ha servido de punto de partida para la optimización final, que no puede derivarse de la solución inicial.

La moraleja de este ejemplo, válida, haya o no recursividad, es que, en la medida de lo posible, no debemos conformarnos con la primera solución que se nos ocurra para un problema, sino que conviene estudiar sus implicaciones y buscar posibles alternativas, para quedarnos con la que sea mejor en el contexto en que se va a aplicar.