Recursividad

Introducción

En este documento se intenta presentar de modo sencillo la recursividad como técnica muy útil para la resolución de determinados tipos de problemas.

Aproximación a la recursividad

Supongamos que queremos saber cuál es el valor más grande de un *array* como el siguiente:

```
int[] v = {21, 34, 12, 43, 8, 31, 20, 14};
```

Supongamos que disponemos de una clase llamada *ArrayUtils* que ofrece un método llamado mayor para hacer esto; entonces, solo hay que llamarlo:

```
int max = ArrayUtils.mayor(v);
```

Supongamos que somos nosotros quienes tenemos que escribir el código del método mayor para que otros lo usen. Un método para encontrar el valor máximo de un *array* no es muy complicado:

```
public static int mayor(int[] v) {
    int result = v[0];

    for (int i = 1; i < v.length; i++) {
        if (v[i] > result) {
            result = v[i];
        }
    }

    return result;
}
```

Hemos hecho la suposición razonable de que el *array* que nos pasan tiene al menos un elemento y aplicamos un algoritmo sencillo que inicializa el resultado con el primer elemento del *array* y luego va comparando con todos los demás, actualizando el resultado cada vez que encuentra un elemento mayor que el mayor encontrado hasta ese momento. Fácil, pero… es que le tenemos fobia a los bucles ¿no podría solucionarse el problema sin usar bucles?

Pensemos, …, hay un caso en el que no hacen falta bucles: cuando el *array* solo tienen un elemento.

```
public static int mayor(int[] v) {
    int result;

    if (v.length == 1) {
        result = v[0];
    } else {
        ¿…?
    }

    return result;
}
```

La cuestión es que, en general, hemos de suponer que el *array* tendrá más de un elemento. Pero, puestos a suponer, supongamos que existen dos métodos, *mayorInf* y *mayorSup*, que, dado un *array*, devuelven, respectivamente, el mayor elemento de la primera mitad del *array* y el mayor elemento de la segunda mitad del array; ello nos facilitaría muchísimo las cosas:

```
public static int mayor(int[] v) {
    int result;

    if (v.length == 1) {
        result = v[0];
    } else {
        int inf = mayorInf(v);
        int sup = mayorSup(v);

        if (inf > sup) {
           result = inf;
        } else {
           result = sup;
        }
    }

    return result;
}
```

¿Estamos haciendo trampas? ¿Nos hemos limitado a esconder los bucles en métodos auxiliares? Ya llegaremos a eso. Antes, fijémonos en que, en realidad, no nos hace falta tener dos métodos auxiliares diferentes; podría haber solo uno, con más parámetros que indicasen la zona del *array* en la que queremos buscar.

```
public static int mayor(int[] v) {
    int result;

    if (v.length == 1) {
        result = v[0];
    } else {
        int inf = mayorAux(v, 0, v.length / 2);
        int sup = mayorAux(v, v.length / 2, v.length);

        if (inf > sup) {
           result = inf;
        } else {
           result = sup;
        }
    }

    return result;
}
```

Claro que, si tenemos un método capaz de encontrar el valor más grande entre dos posiciones indicadas de un *array*, podríamos usarlo, sin más, para encontrar el valor más grande de todo el *array* (solo habría que hacer que las posiciones indicadas abarquen el *array* completo).

```
public static int mayor(int[] v) {
    return mayorAux(v, 0, v.length);
}
```

Radical, pero en esto no hay magia, habrá que escribir el código de *mayorAux*. Para hacerlo podemos aprovechar lo hecho previamente para *mayor*.

```
public static int mayorAux(int[] v, int ini, int fin) {
    int result;

    if (fin == ini + 1) { // un solo elemento
        result = v[ini];
    } else {
        int inf = mayorAux(v, ini, (fin + ini) / 2);
        int sup = mayorAux(v, (fin + ini) / 2, fin);

        if (inf > sup) {
           result = inf;
        } else {
           result = sup;
        }
    }

    return result;
}
```

Llegados a este punto, no hay trampa, el problema está completamente resuelto sin usar bucles. El método *mayorAux*, cuando el *array* tiene un solo elemento, lo devuelve, y cuando tiene más, se llama a sí mismo para calcular los mayores de cada mitad del *array* y elegir luego entre ellos. Si las mitades tienen más de un elemento, se repetirá el proceso de la misma manera para cada mitad del rango de búsqueda, tal como muestra la Ilustración 1, de tal manera, que siempre se acabará alcanzando cada uno de los elementos del *array*.

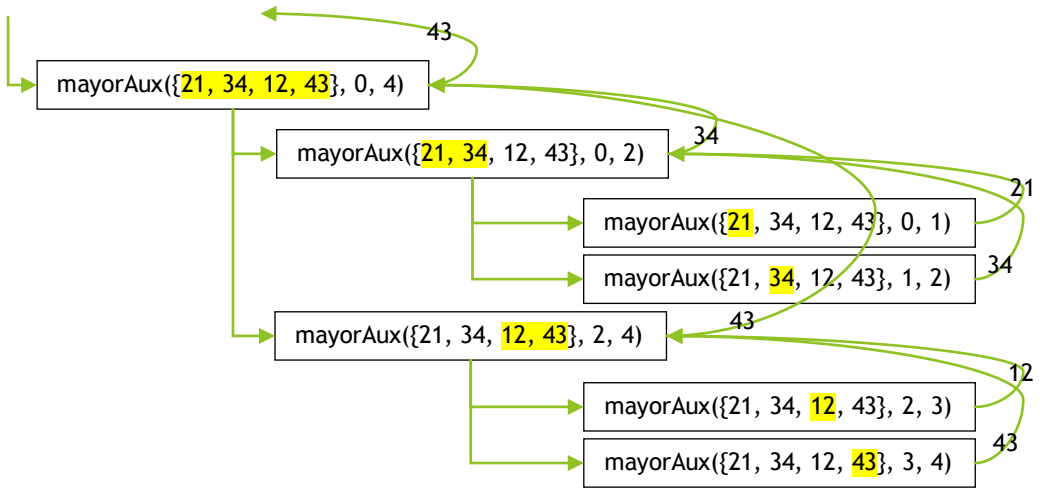

Ilustración 1

Caracterización de la recursividad

Estructura de una solución recursiva

La solución que hemos alcanzado para el método *mayorAux* es una solución recursiva, en oposición a la solución inicial del método *mayor* usando un bucle, que es una solución iterativa (iterar = repetir). Hay que aclarar que, aunque para llegar a la solución recursiva que queríamos hemos usado la excusa de desarrollar una solución sin bucles, la recursividad nada tiene que ver con que, haya o no haya bucles; perfectamente pueden haber bucles en un algoritmo recursivo, aunque, habitualmente, no dominan su estructura.

Para implementar un método recursivo es necesario partir de una formulación recursiva del problema que se quiere resolver. La mayoría de los problemas admiten más de una formulación; por ejemplo, la bien conocida función factorial puede definirse como:

1. Dado un número natural, n, la función factorial(n) toma el valor 1 cuando n vale 0, y el producto de todos los valores comprendidos entre 1 y n, ambos incluidos, cuando n tiene un valor mayor que cero.
2. Dado un número natural, n, la función factorial(n) toma el valor 1 cuando n vale 0, y el producto de n por factorial(n – 1), cuando n tiene un valor mayor que cero.

La diferencia fundamental entre ambas formulaciones es que en la segunda definición aparece una versión más pequeña (factorial (n – 1)) del problema que se está definiendo (factorial(n)). La primera formulación expresa que, en general, n! = $\prod_{i=1}^{n} i$ = 1 * …* (n – 2) * (n – 1) * n, lo que conduce a una repetición de multiplicaciones:

```
public static int factorial(int n) {
    int result = 1;

    for (int i = 1; i <= n; i++) {
        result = result * i;
    }

    return result;
}
```

La segunda expresa que n! = n * (n - 1)!, lo que conduce a una solución recursiva en la que el problema se resuelve, en parte, resolviendo una versión más pequeña del mismo:

```
public static int factorial(int n) {
    int result;

    if (n == 0) {
        result = 1;
    } else {
        result = n * factorial(n – 1);
    }

    return result;
}
```

El ejemplo que desarrollamos en la sección previa, también admite distintas formulaciones:

1. Dados una *array*, *v*, de elementos de tipo *int*, y dos valores, *ini*, *fin*, de tipo *int*, que representan un rango válido de índices de v, el mayor elemento de v en dicho rango es un elemento de v cuyo índice está comprendido entre *ini* y *fin*, y cuyo valor es mayor o igual que el de cualquier elemento de v cuyo índice esté comprendido entre *ini* y *fin*.
2. Dados una *array*, *v*, de elementos de tipo *int*, y dos valores, *ini*, *fin*, de tipo *int*, que representan un rango válido de índices de v, el mayor elemento de v en dicho rango es:
    a. **si el rango abarca un solo elemento:** ese elemento *v[ini]*
    b. **si el rango abarca más de un elemento:** el mayor elemento de la primera mitad del rango, o el mayor elemento de v en la segunda mitad del rango.
3. Dados una *array*, *v*, de elementos de tipo *int*, y dos valores, *ini*, *fin*, de tipo *int*, que representan un rango válido de índices de v, el mayor elemento de v en dicho rango es:
    a. **si el rango abarca un solo elemento:** ese elemento *v[ini]*
    b. **si el rango abarca más de un elemento:** *v[ini]*, o el mayor elemento de v en el rango que va desde *ini + 1* a *fin*.

La primera formulación es no recursiva. La segunda se apoya en dos versiones del problema original aplicadas a casos cuyo tamaño es la mitad del original, y conduce a la implementación recursiva mostrada en la sección anterior. La tercera se apoya en una versión del problema original que tiene un elemento menos, y conduce a una solución recursiva diferente:

```
public static int mayorAux(int[] v, int ini, int fin) {
    int result;

    if (fin == ini + 1) { // un solo elemento
        result = v[ini];
    } else {
        int sup = mayorAux(v, ini + 1, fin);

        if (v[ini] > sup) {
           result = v[ini];
        } else {
           result = sup;
        }
    }

    return result;
}
```

Todas las formulaciones recursivas mostradas tienen en común dos cosas:

1. Al descomponer el problema en subproblemas más sencillos, algunos de estos subproblemas, o todos, son versiones más pequeñas del mismo problema. Esto implica que el problema tiene un tamaño, que se puede descomponer en subproblemas de tamaño menor, y que, en consecuencia… El tamaño es una característica intrínseca de cada problema, que hay que descubrir antes de afrontar una solución recursiva: en problemas de *arrays* como los mostrados, puede ser el tamaño del *array*, o el número de elementos tratar del *array*; en problemas numéricos como el factorial, puede ser directamente un valor pasado; en problemas de ristras, puede ser la longitud de las mismas. A veces, pueden ser una combinación de propiedades de los parámetros que se manejan.
2. Existe una versión del problema que tiene el menor tamaño posible, que, por tanto, ya no se puede descomponer en subproblemas del mismo tipo, y que tiene una solución diferente, que se considera trivial al no requerir más recursividad.

Ello da lugar a que la estructura típica de un algoritmo recursivo esté dominada por una estructura de selección que permite discriminar, examinando las condiciones que determinan el tamaño del problema, si estamos en el caso más pequeño (caso base), que no requiere recursividad, o estamos en el caso general, cuya solución requiere tres pasos:

1. Descomponer el problema en subproblemas.
2. Resolver los subproblemas por separado, efectuando llamadas recursivas para aquellos que son versiones más pequeñas del original.
3. Combinar las soluciones de los subproblemas para componer la solución del problema original.

Tipos de recursividad

Para el problema de saber cuál es el valor mayor de un array hemos presentado dos soluciones recursivas, una que requiere dos llamadas, y otra que requiere solo una. Esto marca una diferencia en la forma en que la solución evoluciona que nos permite hablar de dos tipos de recursividad:

- Lineal, que es la que ocurre cuando cada llamada a un método recursivo puede desencadenar, como mucho una nueva llamada.
- No lineal, que es la que ocurre cuando una llamada a un método recursivo puede desencadenar múltiples llamadas, diversificando así los caminos de la ejecución.

También cabe distinguir entre la recursividad directa, que es la que ocurre en todos estos ejemplos, donde un método "se llama a sí mismo" de la indirecta, que ocurriría si un método llama a otro diferente que luego lo llama a él (o, en general, si se vuelve a llamar al método original a través de una cadena de llamadas a otros métodos diferentes).

Inmersión

Otro asunto en que debemos fijarnos es que, inicialmente, el problema que queríamos resolver era saber cuál era el elemento más grande de un *array*, pero el problema que hemos resuelto realmente es saber cuál es elemento más grande de una zona de un *array* delimitada por dos índices, que no es exactamente lo mismo, sino que es un problema más general.

Si hubiésemos querido mantener el problema original, que tenía como único parámetro el *array*, habría que haber pasado a cada llamada recursiva un nuevo *array*, creado previamente copiando la parte del array original que correspondiese, lo cual, al menos en Java, no es demasiado eficiente. En lugar de ello, hemos buscado un problema más general, aumentando los parámetros, de tal manera que el problema que queremos resolver es un caso particular de ese problema general, pero el problema más general se puede resolver con más facilidad o eficiencia.

Esta acción de resolver un problema más general para tener la solución de uno más particular es lo que se conoce como inmersión recursiva (“sumergimos” el problema particular dentro del general que nos es más adecuado resolver).