Unidad 11

Recursividad y complejidad algorítmica

Programación

Índice

Introducción

Un programa tiene un punto de inicio en código para iniciarse en ejecución, usando el método main de la clase principal y posteriormente sigue un flujo lineal de llamadas entre métodos aplicando una lógica iterativa para realizar su funcionalidad. Existe otro concepto de desarrollo que puede facilitar el diseño de ciertos algoritmos, aplicando unos flujos de llamada diferente a lo que hasta ahora habíamos visto y que requiere una comprensión algorítmica de mayor complejidad. Este concepto, denominado recursividad, es lo que se verá en este tema.

Existe diferentes maneras de realizar la codificación de un algoritmo y obtener el mismo resultado. Siempre hay que pensar que aún que se llegue al mismo resultado, no todas las opciones son válidas, debido a que una solución puede ser más lenta que la otra o consuma más recursos. La complejidad algorítmica permite valorar el consumo que puede tener nuestro programa y buscar alternativas mejores.

Al finalizar esta unidad

- Conoceremos que es la recursividad y para que se emplea.
- Distinguiremos su clasificación
- Aplicaremos los diferentes beneficios de la recursividad.
- Entenderemos el concepto de complejidad algorítmica
- Aprenderemos a analizar el consumo de recursos para un algoritmo

11.1. Concepto de recursividad

La recursividad es una técnica que permite resolver determinados problemas de forma cíclica o circular, que de forma iterativa se tendría que duplicar instrucciones repetitivas abusando de estructuras de control for, while, etc.

Esta técnica es difícil de entender ya que no sigue un flujo lineal iterativo. Hay que dividir el problema en subproblemas más pequeños tantas veces como sea necesario hasta una condición de finalización. Se puede detectar rápidamente la técnica de la recursividad cuando se encuentra en el código un método que se llama a sí mismo.

11.2. Utilización de la recursividad

Hasta ahora un método iterativo para cumplir con su funcionalidad iba llamando a otros métodos de forma colaborativa para resolver el algoritmo, planteando desde la raíz del problema como iba a resolverse. Con la recursividad hay que plantear el problema desde la ramificación, es decir, dividir el problema en subproblemas más pequeños y plantear una condición de terminación para esos casos más pequeños, la condición de terminación se denomina **caso base**. En la siguiente imagen se puede ver un ejemplo.

Iterativo

```
public static void imprimirListaNumeros(int numeroMaximo) {

    int i=1;

    while(i <= numeroMaximo) {
        System.out.println("Contador: "+ i);
        i++;
    }

}
```

Recursivo

```
public static void imprimirListaNumerosRecursivo(int numeroMaximo) {

    if(numeroMaximo==1) {
        System.out.println("Contador: "+ numeroMaximo);
    } else {
        imprimirListaNumerosRecursivo(numeroMaximo-1);
        System.out.println("Contador: "+ numeroMaximo);
    }

}
```

Imagen 1. Ejemplo de método iterativo y recursivo

En el ejemplo tenemos un algoritmo para imprimir por pantalla una lista de números hasta un límite máximo. El **caso base** comprueba si ha llegado al mínimo, que es 1, de lo contrario se llama a si mismo pasando como parámetro el límite -1.

Es importante definir correctamente el caso base, como la llamada a sí mismo. Si no descomponemos correctamente, el comportamiento que tendrá será un bucle infinito a sí mismo.

Cualquier algoritmo resuelto con recursividad tiene su forma iterativa equivalente. Para empezar con los primeros pasos con la recursividad primero plantear el problema iterativo y luego buscar la solución aplicando la técnica de la recursividad.

11.3. Tipos de recursividad

Podemos distinguir diferentes tipos de recursividad según la forma de su llamada.

Recursividad simple

Solo existe una sola llamada a sí mismo en el cuerpo del método. Es el caso más sencillo y su descomposición es uniforme.

Iterativo

```
private static int fibonacci(int n) {

    int siguiente = 1, actual = 0, temporal = 0;

    for (int i = 1; i <= n; i++) {
        temporal = actual;
        actual = siguiente;
        siguiente = siguiente + temporal;
    }

    return actual;

}
```

Recursivo

```
private static int fibonacciRecursivo(int n) {
    if (n <= 2) {
        return 1;
    } else {
        return fibonacciRecursivo(n - 1) + fibonacciRecursivo(n - 2);
    }
}
```

Imagen 2. Ejemplo de recursividad simple para el cálculo del factorial de un número.

Recursividad múltiple

Es la más compleja de realizar. Existe más de una llamada a sí mismo en el cuerpo del método.

Iterativo

```
private static int fibonacci(int n) {

    int siguiente = 1, actual = 0, temporal = 0;

    for (int i = 1; i <= n; i++) {
        temporal = actual;
        actual = siguiente;
        siguiente = siguiente + temporal;
    }

    return actual;

}
```

Recursivo

```
private static int fibonacciRecursivo(int n) {
    if (n <= 2) {
        return 1;
    } else {
        return fibonacciRecursivo(n - 1) + fibonacciRecursivo(n - 2);
    }
}
```

Imagen 3. Ejemplo de recursividad múltiple para aplicar la función de Fibonacci.

PARA AMPLIAR CONOCIMIENTO...

Para ampliar el conocimiento sobre Fibonacci dirigiros al siguiente enlace. https://es.wikipedia.org/wiki/Sucesi%C3%B3n_de_Fibonacci#Funci%C3%B3n_generadora

Recursividad cruzada o indirecta

Existen más de dos métodos que tienen llamadas entre ellos. El método A en su cuerpo llama al método B y el método B en su cuerpo llama al método A. Tiene que haber más de un caso base en cada método.

Iterativo

```
private static int parImpar(int num) {
    return num%2;
}
```

Recursivo

```
private static int par(int num) {
    if (num == 0)
        return (1);
    return (impar(num - 1));
}

private static int impar(int num) {
    if (num == 0)
        return (0);
    return (par(num - 1));
}
```

Imagen 4. Ejemplo de recursividad cruzada o indirecta para determinar si un número es par o impar.

Recursividad anidada

Se produce cuando en los parámetros de la llamada se incluye otra llamada a sí mismo. Si tenemos el método A que devuelve un número y recibe como parámetro un número entero, la llamada seria metodoA(metodoA(n));

```
private static int ackermann(int n, int m) {
    if (n == 0)
        return (m + 1);
    else if (m == 0)
        return (ackermann(n - 1, 1));
    return (ackermann(n - 1, ackermann(n, m - 1)));
}
```

Imagen 5. Ejemplo de recursividad anidada para aplicar la función de Ackermann

PARA AMPLIAR CONOCIMIENTO...

Para ampliar el conocimiento sobre la función de Ackermann dirigiros al siguiente enlace. https://es.wikipedia.org/wiki/Funci%C3%B3n_de_Ackermann

11.4. Ventajas e inconvenientes

Existen más inconvenientes que ventajas. La recursividad necesita hacer uso de más memoria, aumenta la pila considerablemente al tener más llamadas a un método y por consecuencia el programa se vuelve más lento. Entonces, ¿cuáles son las ventajas? La recursividad se emplea para patrones concretos que de forma iterativa no se pueden llegar a plantear, hace más sencillo el algoritmo y evita el abuso de sentencias de control y bucles.

11.5. Complejidad algorítmica

Un algoritmo puede tener diferentes soluciones, acabamos de ver, que es posible tener un algoritmo diseñado de forma iterativa o recursiva. ¿Qué solución es la más correcta? Siempre que se diseñe una solución a un algoritmo, no solo basta con que el código sea lo más claro posible o se intente reutilizar todo el código posible, hay que priorizar los recursos que va a consumir nuestra solución.

La **complejidad algorítmica** estudia la cantidad de recursos que puede consumir una solución en base a los parámetros del tiempo que tarda y la cantidad de memoria que necesita. La forma de medir la complejidad algorítmica es mediante la notación Big-O (Notación Asintótica o Notación Landau) que mide como se va a comportar un algoritmo en función de los argumentos que se le pasen.

En el siguiente cuadro se puede ver las distintas ordenes de complejidad que existen.

Orden	Nombre	Descripción
O(1)	Constante	Es la más rápida. El tiempo es contante y no varía según el tamaño de los datos.
O(log n)	Logarítmica	Divide el problema en partes mas pequeñas, simplificando su resolución. Normalmente se aplica en recursividad. Por ejemplo la búsqueda dicotómica.
O(n)	Líneal	Depende del tamaño de los datos. Se suele encontrar en bucles simples de N iteraciones, por ejemplo algoritmos de búsqueda lineal, cálculo de factorial.
O(n log n)	Cuasi lineal	Es similar a la orden logaritmica. Divide el problema en partes más pequeñas pero es necesario la unión de cada parte para llegar a la solución final. Ejemplos, ordenación rápida (Quicksort) o la ordenación por mezcla (merge sort)
O(n2)	Cuadrática	Se usa para recorrer todos los elementos de un conjunto. Aparece cuando hay bucles anidados de 2 niveles. Un bucle dentro de otro, por ejemplo recorrer un array bidimensional (matriz)

Orden	Nombre	Descripción
O(n3)	Cúbica	Igual que la cuadrática. Aparecen bucles anidados de 3 niveles. Por ejemplo recorrer un array multidimensional
O(na)	Polinominal	Igual que las anteriores pero los niveles son mayores de 2. La orden cúbica también seria polinominal.
O(an)	Exponencial	A mayor datos de uso peor se comporta. Se suele dar en la recursividad multiple. Por ejemplo, el algoritmo fibonnaci en forma de recursividad.
O(n!)	Factorial	Es la peor orden. Solo se aplica a algoritmos mediante fuerza bruta.

Imagen 6. Tabla de tipos de complejidad

Los distintos tipos de orden están ordenados según el coste que consumen de recursos. El orden de menor a mayor es:

O(1) < O(Log n) < O(n) < O(n Log n) < O(n2) < O(n3) < O(na) < O(an) < O(n!)

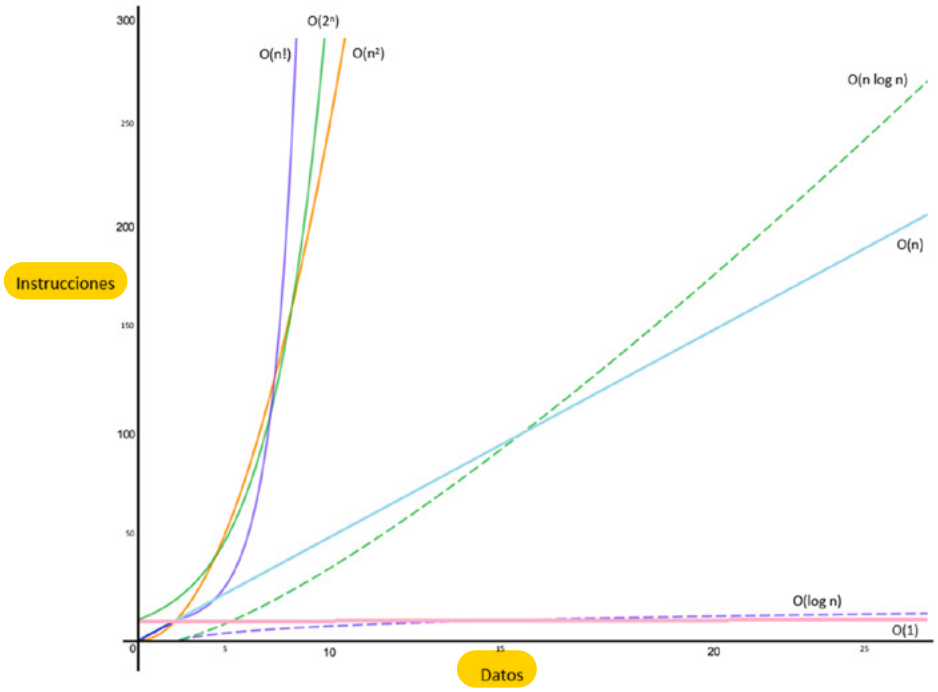

Imagen 7. Gráfico del orden de consumo según los datos de entrada y las instrucciones

Para determinar qué orden corresponde al desarrollo de un algoritmo hay que analizar el tipo de instrucciones (sentencias de control, bucles, anidamientos, recursividad, etc) y la cantidad de datos que hay que abordar. Siempre se estudiará el mejor de los casos y el peor posible.

Ahora veremos algunos ejemplos y que orden le corresponde.

```
private static void cadenaInvertida(String cadena) {

    String cadenaOriginal = cadena;
    StringBuilder strb = new StringBuilder(cadenaOriginal);
    String cadenaInvertida = strb.reverse().toString();

}
```

Ejemplo 6

Imagen 8. Ejemplo de algoritmo para invertir una cadena de caracteres.

En el ejemplo de la imagen 6 se puede ver que hay 3 instrucciones, no hay estructuras condicionales ni bucles. Va a tener una orden constante porque siempre se van a ejecutar las 3 instrucciones independientemente se use una vez, con lo cual, se determina O(1).

```
private static void recorrerArray(Object[] lista) {

    for (int i=0;i<lista.length;i++) {
        System.out.println(lista[i]);
    }

}
```

Ejemplo 7

Imagen 9. Ejemplo de algoritmo para recorrer un array.

En el ejemplo de la imagen 7 tenemos un bucle que recorre todos los elementos de un array y realiza una sola instrucción por cada elemento. El peor caso de todos es N instrucciones, N puede ser 1, 10, 1000, o infinidad de elementos. Con lo cual su orden es incremental según la cantidad de datos que reciba, con lo cual, se determina O(n).

```
private static void recorrerArrayInversaNoLineal(Object[] lista) {

    int n = lista.length;

    while(n>0) {
        System.out.println(lista[n]);
        n /= 2;
    }

}
```

Ejemplo 8

Imagen 10. Ejemplo de algoritmo para recorrer un array desde el final en delante de forma no lineal.

En el ejemplo de la imagen 8 tenemos un algoritmo que recorre un array al revés para mostrar elementos en posiciones divididas. En comparación con el ejemplo anterior, en este caso, nunca va a recorrer el array completo, vemos que el índice por cada iteración los divide en 2, con lo cual, podemos determinar que le corresponde O(log n).

Cuando tenemos un algoritmo muy grande, que contienen diferentes instrucciones, diferentes entradas de datos e interacciones con muchas llamadas a métodos se dividirá el algoritmo por partes sencillas y obtendremos su orden de complejidad y luego se ira uniendo cada parte sumando su complejidad. La orden de complejidad resultante tiene que ser la parte del algoritmo que más recurso consume y la suma de las inferiores.

Existen muchas reglas y funciones para el cálculo de ordenes de complejidad que requiere conocimientos matemáticos avanzados. Para no entrar en profundidad en conceptos matemáticos a continuación se muestran unas reglas básicas que pueden ayudar para encontrar rápidamente que orden de complejidad corresponde a un algoritmo:

- **Las sentencias sencillas** de 1 o 2 líneas como las de asignación, entrada/salida, evaluación de operaciones aritméticas, las que trabajan sobre un tamaño de datos constante,etc. siempre será O(1).
- **En estructuras condicionales (if, if/else)**. La complejidad siempre será la mayor de cualquiera de sus partes condicionales.
- **En bucles (for, for each, while).** Si los elementos del bucle son fijos y las instrucciones dentro del bucle son sencillas, la orden sera O(1). De lo contrario si los elementos del bucle no son fijos, y existen N elementos, será O(n).
- **Bucles anidados.** Si tenemos más de un bucle dentro de otro ya partiremos de una orden de O(n2) a O(na).
- **Para un conjunto de instrucciones.** La orden de complejidad será la suma de todas sus partes.
- Cuando hay llamadas a **procedimientos, funciones o métodos**, la orden de complejidad se determina según el contenido que tienen dentro.
- **En recursividad.** La complejidad del algoritmo viene dada por el caso base y de los recurrentes.

Una vez entendido que es la complejidad algorítmica, cobra real importancia realizar un buen diseño de nuestro programa, no solo que estéticamente sea legible y fácil de entender, sino que hay que preocuparse por buscar una solución sencilla con instrucciones que no sean complejas, para minimizar al máximo el consumo de recursos y que nuestra orden de complejidad sea la más baja posible.

www.universae.com