

Asignatura

Programación



UNIVERSAE
Instituto Superior de FP

Asignatura

Programación

UNIDAD 11

Recursividad y complejidad
algorítmica



UNIVERSAE
Instituto Superior de FP

Recursividad

Técnica de programación

- Permite resolver determinados problemas
- No hay un flujo lineal iterativo
- El esquema del flujo de instrucciones es **cíclico o circular**
- Por ejemplo, Factorial, Fibonacci, etc.



Ventajas

- Simplifica el diseño para determinados problemas
- Evita el abuso de estructuras de control y bucles
- Es la única opción cuando hay solución de forma iterativa



Inconvenientes

- Uso de más memoria
- Sobrecarga de la pila de llamadas
- Lentitud





Estructura del procedimiento recursivo

Diseño

- El problema principal se descompone en **problemas más pequeños**
- La solución parte desde los subproblemas hacia el problema principal.
- Es necesario el uso de métodos o funciones.
- Debe existir una o más **llamadas sobre el mismo método o función**
- Debe haber siempre un **caso base** para terminar.
- Debe producirse una **variación de datos** sobre cada llamada

```
public static void imprimirListaNumerosRecursivo(int numeroMaximo) {  
  
    if(numeroMaximo>1) {  
        imprimirListaNumerosRecursivo(numeroMaximo-1);  
    }  
  
    System.out.println("Contador: "+ numeroMaximo);  
  
}
```

Recursividad (1) [Java Application]
Recursividad at localhost:57540
Thread [main] (Suspended (breakpoint at line 22 in Recursividad))
Recursividad.imprimirListaNumerosRecursivo(int) line: 22
Recursividad.imprimirListaNumerosRecursivo(int) line: 19
Recursividad.imprimirListaNumerosRecursivo(int) line: 19
Recursividad.imprimirListaNumerosRecursivo(int) line: 19
Recursividad.imprimirListaNumerosRecursivo(int) line: 19
Recursividad.main(String[]) line: 31

Console × Problems @ Java
<terminated> Recursividad (1) [Java Appli
imprimirListaNumeroRecursivo(5)
imprimirListaNumeroRecursivo(4)
imprimirListaNumeroRecursivo(3)
imprimirListaNumeroRecursivo(2)
imprimirListaNumeroRecursivo(1)
Contador: 1
Contador: 2
Contador: 3
Contador: 4
Contador: 5

```
método nombre (parámetros) {  
  
    // acciones antes de la siguiente llamada  
  
    // CASO BASE  
  
    // LLAMADA  
    // nombre (parámetros)  
  
    // acciones que se realizan después de la llamada  
  
}
```



Iterativo vs Recursivo

- No todos los problemas se pueden resolver iterativamente
- Un proceso recursivo puede tener la misma solución en forma iterativa
- ¿Ventaja o inconveniente?

Usar forma iterativa

- El código sea claro, sencillo y de complejidad básica
- No hay abuso de estructuras condicionales o iterativas.
- El límite de datos es considerable

```
private 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;  
}
```

Usar forma recursiva

- No hay solución mediante forma iterativa
- El código se simplifica
- El límite de datos es reducido.

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

Tipos de recursividad



Simple

- Existe una llama a si mismo dentro del mismo método

```
public double factorialRecursivo(int n) {
    if(n == 0)
        return 1;
    else
        return n * factorialRecursivo(n-1);
}
```



Múltiple

- Existe mas de una llama a si mismo dentro del mismo método

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



Cruzada o indirecta

- Hay una llamada entrelazada entre dos métodos

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

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



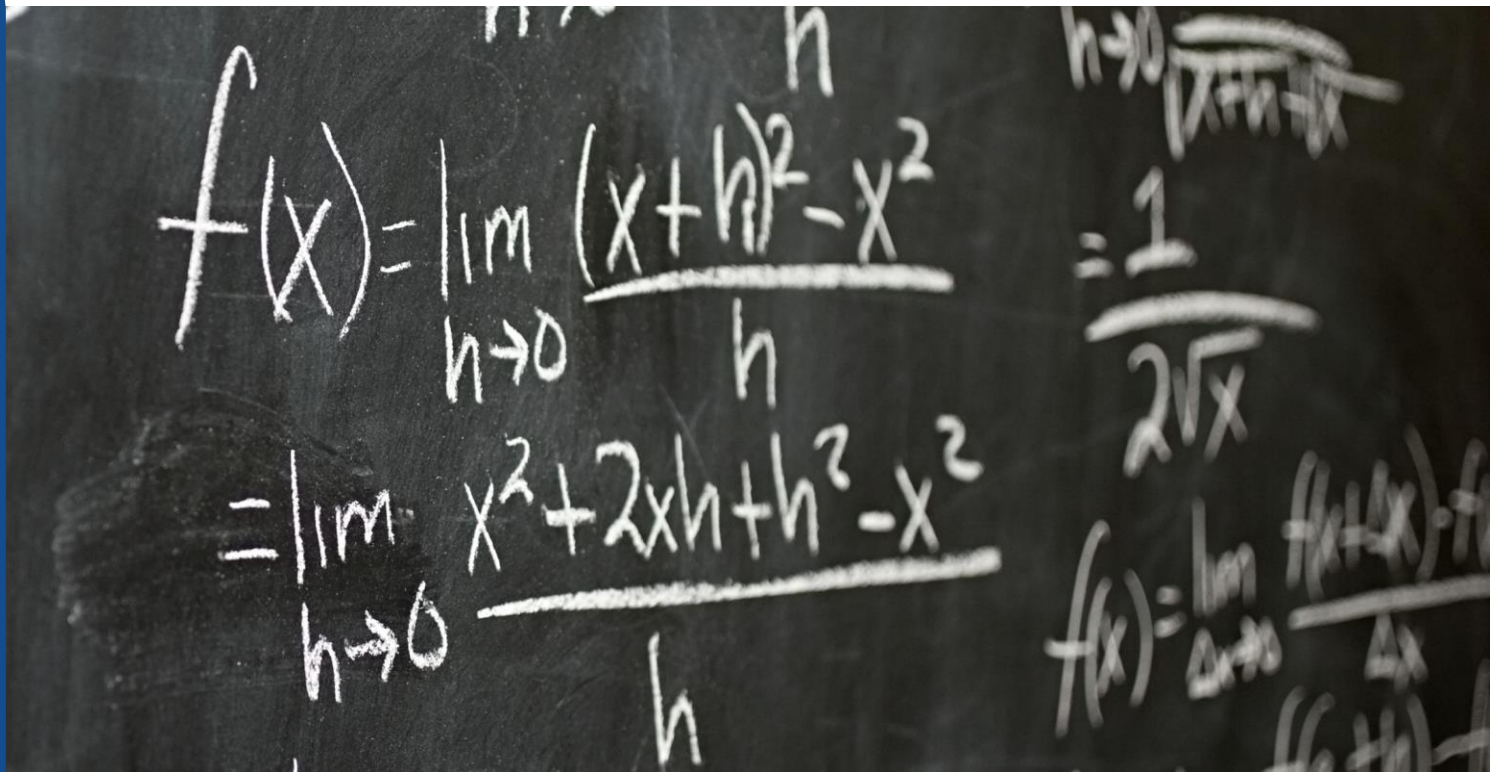
Anidada

- Aparece como parámetro de la llamada, otra llamada.

```
public 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)));
}
```


Complejidad algorítmica

- ¿Cómo determinar que una solución es mejor que otra?
- Estudiar la cantidad de recursos de una aplicación según el tiempo y la cantidad de memoria necesaria.
- Medición Big-O
 - **Notación Asintótica**
 - **Notación Landau**



Handwritten mathematical derivation on a chalkboard:

$$f(x) = \lim_{h \rightarrow 0} \frac{(x+h)^2 - x^2}{h}$$
$$= \lim_{h \rightarrow 0} \frac{x^2 + 2xh + h^2 - x^2}{h}$$
$$= \lim_{h \rightarrow 0} \frac{2xh + h^2}{h}$$
$$= \lim_{h \rightarrow 0} (2x + h) = 2x$$

Other visible text on the right side of the chalkboard includes:

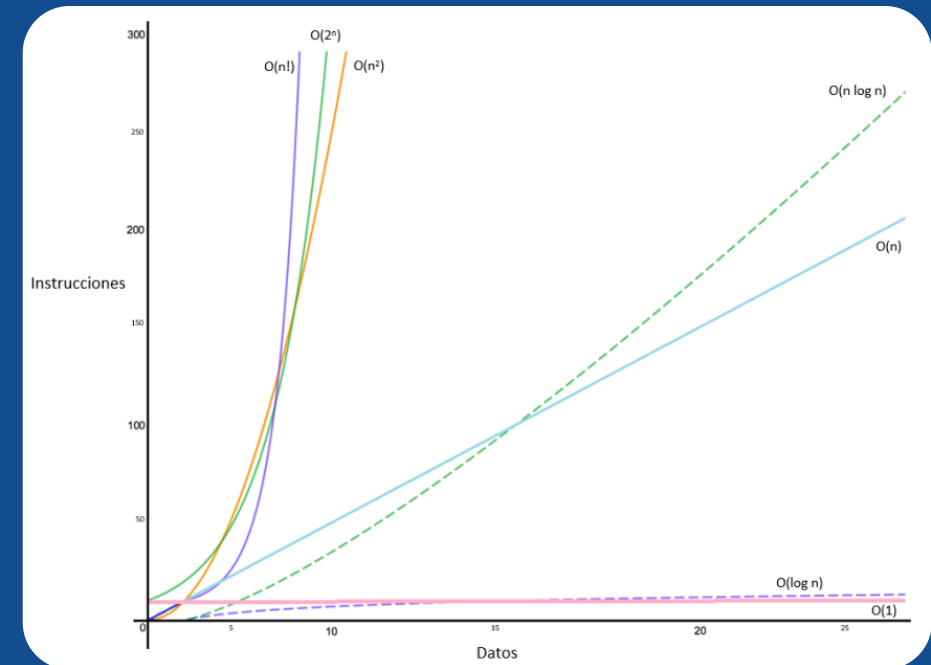
$$h \rightarrow 0 \frac{1}{\sqrt{x+h} - \sqrt{x}} = \frac{1}{2\sqrt{x}}$$
$$f(x) = \lim_{\Delta x \rightarrow 0} \frac{f(x+\Delta x) - f(x)}{\Delta x}$$



Ordenes de complejidad

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 logarítmica. 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)
$O(n^2)$	Cuadrática	Se usa 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)
$O(n^3)$	Cúbica	Igual que la cuadrática. Aparecen bucles anidados de 3 niveles. Por ejemplo recorrer un array multidimensional
$O(n^a)$	Polinomial	Igual que las anteriores pero los niveles son mayores de 2. La orden cúbica también sería polinomial.
$O(a^n)$	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.

$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(n^a) < O(a^n) < O(n!)$



Ejemplos de cálculo de complejidad



```
Long start = System.currentTimeMillis();
while(n>0) {
    n /= 2;
}
Long end = System.currentTimeMillis();
System.out.println("Tiempo gastado: " + ((end - start)*0.001));
```

```
Long start = System.currentTimeMillis();
long suma = 0;
for (long i = 0; i < n; i++) {
    suma += i;
}
Long end = System.currentTimeMillis();
System.out.println("Tiempo gastado: " + ((end - start)*0.001));
```

```
Long start = System.currentTimeMillis();
long suma = 0;
for (long i = 0; i < n; i++) {
    for (long j = 0; j < m; j++) {
        suma += j;
    }
}
Long end = System.currentTimeMillis();
System.out.println("Tiempo gastado: " + ((end - start)*0.001));
```

DATOS/ TIEMPO	1.000	10.000	100.000	1.000.000	10.000.000	100.000.000
Opción 1	0,0	0,0	0,0	0,0	0,0	0,0
Opción 2	0,0	0,0	0,001	0,002	0,005	0,02600
Opción 3	0,002	0,029	2,325	230,299	53039,62	Indeterminado

	Orden
Opción 1	$O(\log n)$
Opción 2	$O(n)$
Opción 3	$O(n^2)$



Resumen

1. Recursividad
2. Estructura del procedimiento recursivo
3. Iterativo vs Recursivo
4. Tipos de recursividad
5. Complejidad algorítmica
6. Ordenes de complejidad
7. Ejemplos de cálculo de complejidad

The background is a solid blue color with a subtle, large-scale grid pattern. Overlaid on this are numerous small, light blue arrows pointing in various directions, creating a sense of movement and flow. The text is centered in the middle of the image.

UNIVERSAE

— CHANGE YOUR WAY —