# RECURSIVIDAD

En matemáticas se da el nombre de recursión a la técnica consistente de definir una **función en términos de sí misma**. Ejemplo: Factorial de un número, Serie Fibonacci, series y sucesiones en general.

Se llama recursividad a un proceso mediante el cual una **función se llama a sí misma de forma repetida**, hasta que se **satisface alguna condición determinada**. El proceso se utiliza para efectuar acciones repetidas en las que cada acción se determina mediante un resultado anterior. Se pueden escribir de esta forma muchos problemas iterativos.

Aunque se puede utilizar la recursividad como **una alternativa a la iteración**, una solución recursiva es, normalmente, menos eficiente en términos de tiempo de computadora que una solución iterativa, o, debe estar optimizada para que sea eficiente. Sin embargo, en muchas circunstancias, el uso de la recursión permite a los programadores especificar soluciones naturales, sencillas, que serían, en caso contrario, un poco más difíciles de resolver. Ej. backtracking, buscar la salida de un laberinto, árboles, fractales.

**Ejemplo** (en teoría):

```
metodo1 (...){
  ...
}

metodo2 (...){
  ...
  metodo1(...); //llamada al metodo1
  ...
}
```

```
metodo1(...){
  ...
  metodo1(...);
  ...
}
```

Cuando se hace una llamada a un método recursivo para resolver un problema, el método en realidad es capaz de resolver **sólo el caso más simple**, o caso base, por lo tanto, un método recursivo se compone de dos cosas:

- **Caso base:** determina el fin de las llamadas recursivas.
- **Paso recursivo (llamada recursiva):** es cuando el método llama a una nueva copia de sí mismo para trabajar en el problema más pequeño.

El paso recursivo se ejecuta mientras siga activa la llamada original al método (es decir, que no haya terminado su ejecución). Cada llamada recursiva al método se guarda **en memoria usando una estructura de datos tipo pila**.

Para que la recursividad termine en un momento dado, cada vez que el método se llama a sí mismo, **la secuencia de problemas debe converger en un caso base**.

Las métodos con llamadas recursivas **utilizan memoria extra en las llamadas**; existe un límite en las llamadas, que depende de la memoria de la computadora. En caso de superar este límite ocurre un **error de overflow**.

Cada llamada recursiva al método se almacena en una pila en donde se almacena el estado de cada llamada recursiva, cuando llega al caso base, lo que hace es vaciar la pila evaluando los resultados de cada llamado al método recursivo.

**Ejemplo (Función recursiva)**:

S(n) = S(n-1) + 2*n,

Como punto de partida, se puede afirmar que:

- Para n = 1, la función retorna 1, S(1) = 1 **(caso base)**.
- Para n = 2, se puede escribir, S(2) = S(2-1) + 2*2, que es igual a: S(2) = S(1) + 4 = 5.

El algoritmo que determina la suma de modo recursivo ha de tener presente una condición de salida o una condición de parada. Para este caso el caso base es S(1) = 1.

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

**Ejemplo (Factorial):**

n! = n * (n – 1) ! donde 0! = 1

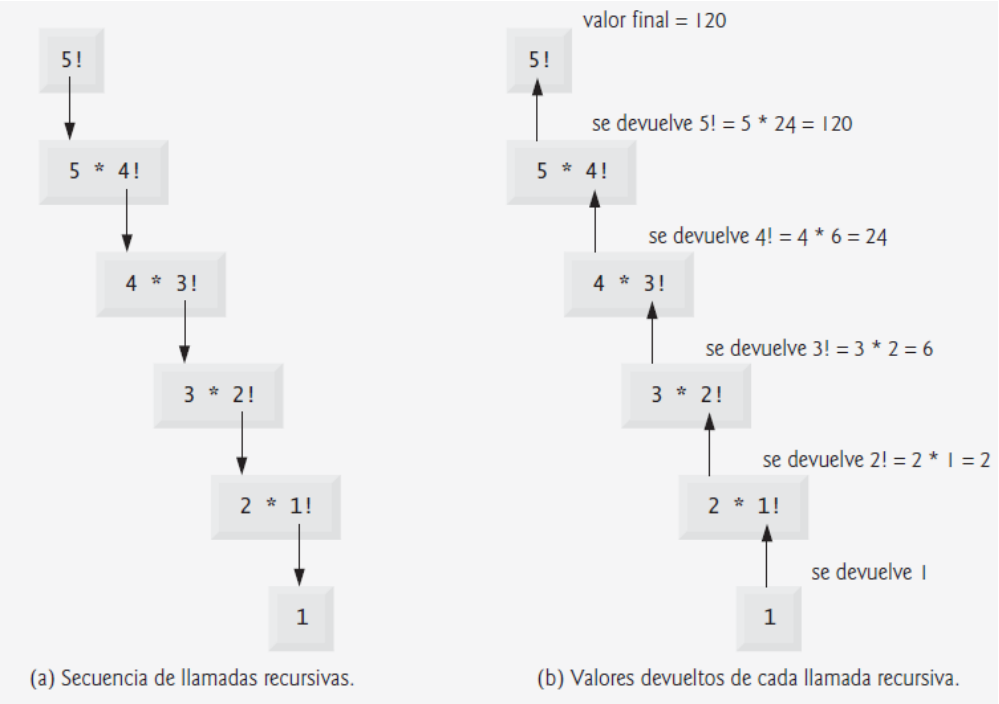


*Imagen 1. Pila recursiva factorial de 5*

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

```
}
```

En resumen, se considera un método recursivo:

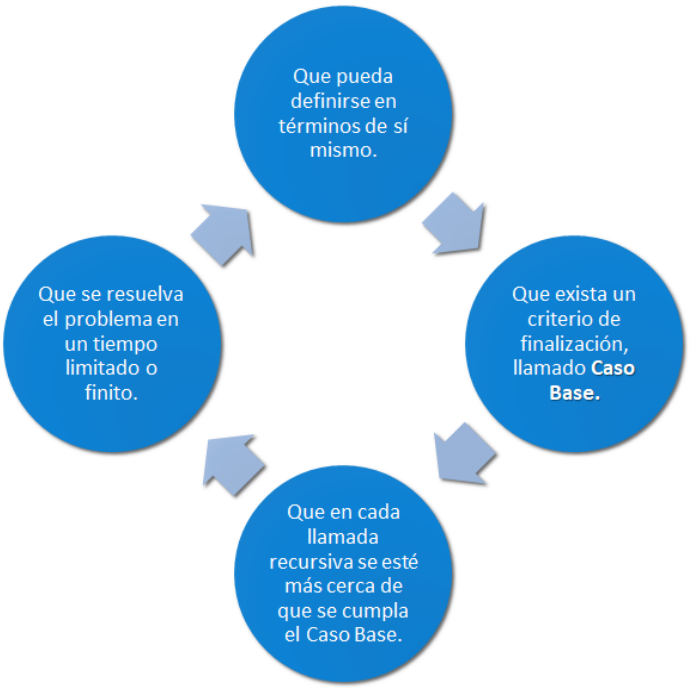


*Imagen 5. Resumen conceptos.*

**Ejercicios**

Escriba un método recursivo para:

1. Multiplicar dos números mediante sumas sucesivas.
2. División de un número entre otro mediante restas sucesivas.
3. Sumar todos los números impares hasta n. Donde n es un número que se pasa como parámetro.
4. Calcular y retornar cualquier potencia para cualquier número.
5. Imprimir en consola un arreglo de cualquier tipo.
6. Sumar todos los elementos de un array.
7. Devolver el menor elemento de un arreglo.
8. Contar cuántas veces está un elemento dentro de un array.
9. Imprimir en consola una matriz cuadrada de cualquier tipo.
10. Sumar todos los elementos de una matriz cuadrada.
11. Determinar si una palabra o frase es palíndroma o no. boolean.
12. Contar el número de vocales de una cadena: `int vocales(String cadena, int contador)`.

**Ejemplo (Fibonacci):**

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

La serie Fibonacci **empieza con 0 y 1**, y tiene la propiedad de que cada número siguiente de Fibonacci es la suma de los dos números Fibonacci anteriores. Esta serie ocurre en la naturaleza y describe una forma de espiral. La proporción de números de Fibonacci sucesivos converge en un valor constante de 1.618…, un número denominado **proporción dorada**, o media dorada.

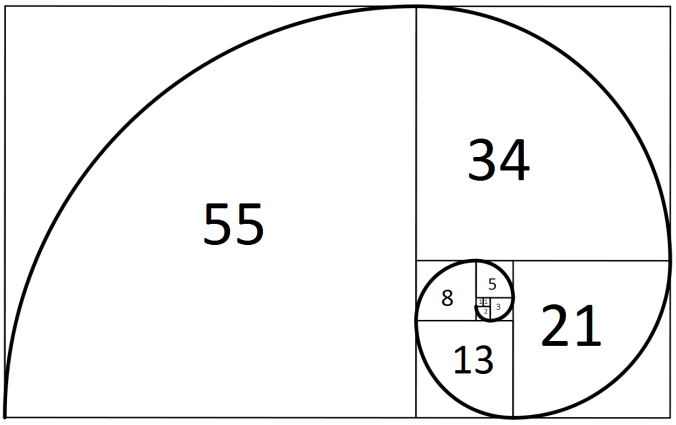


*Imagen 2. Espiral Fibonacci*


*Imagen 3. Espiral en la vida real*

**Código:**

```
public int fibonacci(int n) {

	int ant1 = 0, ant2 = 1, fibonacci = n;

	for (int i = 0; i < n-1; i++) {
		fibonacci = ant1 + ant2;
		ant1 = ant2;
		ant2 = fibonacci;
	}

	return fibonacci;
}
```

Recursivo:

```
public int fibonacci(int n) {

	if(n==1 || n==0) {
		return n;
	}else {
		return fibonacci(n-2) + fibonacci(n-1);
	}
}
```

**Problema:** Complejidad en tiempo de ejecución.

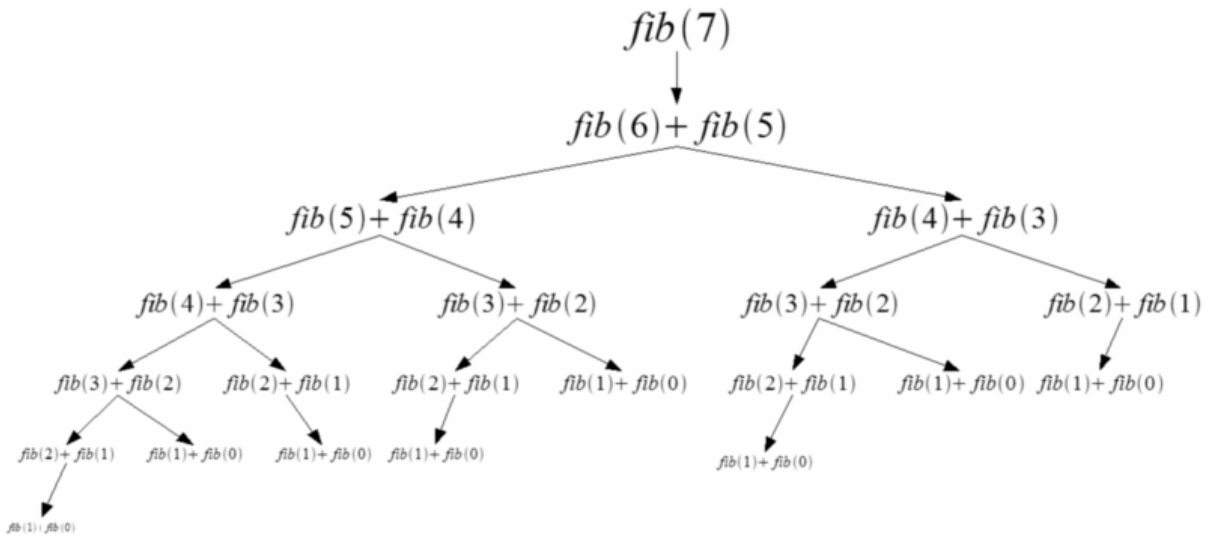


*Imagen 4. Árbol de llamadas recursivas.*

Cada invocación del método fibonacci que no coincide con uno de los casos base (0 o 1) produce dos llamadas recursivas más al método fibonacci. Por lo tanto, este conjunto de llamadas recursivas **se sale rápidamente de control**.

Para calcular el valor 20 de Fibonacci se requieren **21.891** llamadas al método fibonacci; para calcular el valor 30 de Fibonacci se requieren **2.692.537** llamadas. A medida que se trate de calcular valores más grandes de Fibonacci se requiere un aumento considerable en tiempo de cálculo y en el número de llamadas al método fibonacci. Por ejemplo, el valor 31 de Fibonacci requiere **4.356.617** llamadas, y el valor 32 de Fibonacci requiere **7.049.155** llamadas.

**Mejora al algoritmo de fibonacci:**

```
public int fibonacciMejorado(int numero){
	if(numero==0 || numero==1){
		return numero;
	}else{
		return fibonacciAux(numero, 2, 0, 1);
	}
}
```

```
public int fibonacciAux(int n, int i, int a, int b) {
	if(n==i){
		return a + b;
	}else{
		return fibonacciAux(n, i+1, b, a+b);
	}
}
```

**Diferencia de tiempo entre ambos algoritmos:**

fibonacci(50) -> **75.557025827 segundos**
fibonacciMejorado(50) -> **6.414E-6 segundos**