

# Recursividad

M.S.C. Jacob Green

# Recursividad

- Es una forma de especificar un proceso en base a su propia definición.
- Es una estructura alternativa para realizar tareas repetitivas (ciclos).
- Se llama función recursiva a aquella que se llama así misma y consta de dos partes:
  - Caso base: ocasiona que la función deje de llamarse así misma.
  - Caso recursivo: implica volver a llamar a la función, utilizando parámetros cada vez más cercanos al caso base.

# Factorial

- El factorial de un número implica multiplicar todos los números desde 1 hasta n.
- Ejemplo:
  - $5! = 5 * 4 * 3 * 2 * 1 = 120$

# Factorial

- El factorial de un número implica multiplicar todos los números desde 1 hasta n.
- Ejemplo:
  - $5! = 5 * 4 * 3 * 2 * 1 = 120$
- La definición matemática de factorial es:
  - $n! = n * (n - 1)!$
  - Es decir,  $n! = n * (n-1) * (n-2) * (n-3) \dots * 1$

# Factorial

- El factorial de un número implica multiplicar todos los números desde 1 hasta n.
- Ejemplo:
  - $5! = 5 * 4 * 3 * 2 * 1 = 120$
- La definición matemática de factorial es:
  - $n! = n * (n - 1)!$
  - Es decir,  $n! = n * (n-1) * (n-2) * (n-3) \dots * 1$

```
int factorial(int n) {  
    int fact = 1;  
    while(n > 1) {  
        fact = fact * n;  
        n = n - 1;  
    }  
    return fact;  
}
```

# Factorial

```
int factorial(int n) {  
    int fact = 1;  
    while(n > 1) {  
        fact = fact * n;  
        n = n - 1;  
    }  
    return fact;  
}
```

Iteración	n	fact	n actualizada
1	5	$1 * 5 = 5$	$5 - 1 = 4$

# Factorial

```
int factorial(int n) {  
    int fact = 1;  
    while(n > 1) {  
        fact = fact * n;  
        n = n - 1;  
    }  
    return fact;  
}
```

Iteración	n	fact	n actualizada
1	5	1 * 5 = <b>5</b>	5 - 1 = <b>4</b>
2	<b>4</b>	<b>5</b> * 4 = 20	4 - 1 = 3

# Factorial

```
int factorial(int n) {  
    int fact = 1;  
    while(n > 1) {  
        fact = fact * n;  
        n = n - 1;  
    }  
    return fact;  
}
```

Iteración	n	fact	n actualizada
1	5	$1 * 5 = 5$	$5 - 1 = 4$
2	4	$5 * 4 = \mathbf{20}$	$4 - 1 = \mathbf{3}$
3	<b>3</b>	$\mathbf{20} * 3 = 60$	$3 - 1 = 2$



# Factorial

```
int factorial(int n) {  
    int fact = 1;  
    while(n > 1) {  
        fact = fact * n;  
        n = n - 1;  
    }  
    return fact;  
}
```

Iteración	n	fact	n actualizada
1	5	$1 * 5 = 5$	$5 - 1 = 4$
2	4	$5 * 4 = 20$	$4 - 1 = 3$
3	3	$20 * 3 = \mathbf{60}$	$3 - 1 = \mathbf{2}$
4	<b>2</b>	$\mathbf{60} * 2 = 120$	$2 - 1 = 1$

# Factorial

```
int factorial(int n) {  
    int fact = 1;  
    while(n > 1) {  
        fact = fact * n;  
        n = n - 1;  
    }  
    return fact;  
}
```

Iteración	n	fact	n actualizada
1	5	$1 * 5 = 5$	$5 - 1 = 4$
2	4	$5 * 4 = 20$	$4 - 1 = 3$
3	3	$20 * 3 = 60$	$3 - 1 = 2$
4	2	$60 * 2 = 120$	$2 - 1 = 1$
5	1	-----	-----

# Factorial

```
int factorial(int n) {  
    int fact = 1;  
    while(n > 1) {  
        fact = fact * n;  
        n = n - 1;  
    }  
    return fact;  
}
```

Iteración	n	fact	n actualizada
1	5	$1 * 5 = 5$	$5 - 1 = 4$
2	4	$5 * 4 = 20$	$4 - 1 = 3$
3	3	$20 * 3 = 60$	$3 - 1 = 2$
4	2	$60 * 2 = 120$	$2 - 1 = 1$
5	1	-----	-----

el factorial de **n = 5** es **120**

# Factorial

- También se puede definir de la siguiente manera:

$$n! = \begin{cases} 1 & \text{si, } n = 0 \\ (n-1)! * n & \text{si, } n \geq 1 \end{cases}$$

# Factorial

- También se puede definir de la siguiente manera:

$$n! = \begin{cases} 1 & \text{si, } n = 0 \quad \leftarrow \text{ caso base} \\ (n-1)! * n & \text{si, } n \geq 1 \quad \leftarrow \text{ caso recursivo} \end{cases}$$

# Factorial

- También se puede definir de la siguiente manera:

$$n! = \begin{cases} 1 & \text{si, } n = 0 \quad \leftarrow \text{ caso base} \\ (n-1)! * n & \text{si, } n \geq 1 \quad \leftarrow \text{ caso recursivo} \end{cases}$$

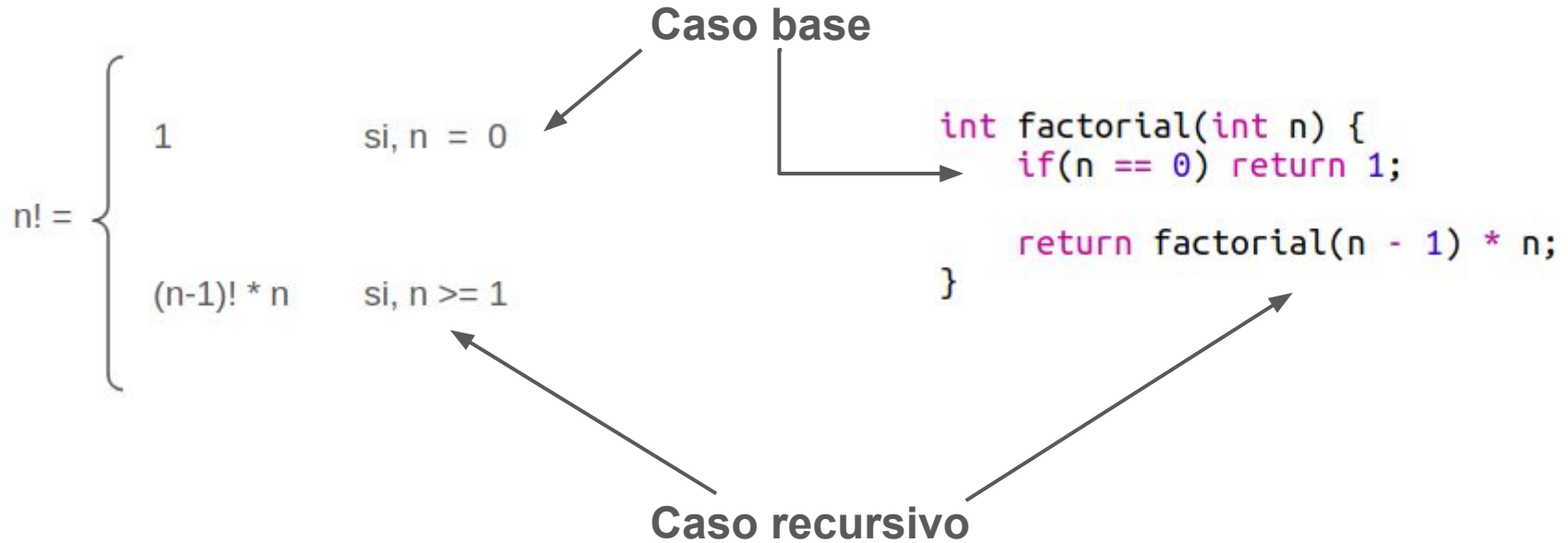
**Definición recursiva!!!**

# Factorial

$$n! = \begin{cases} 1 & \text{si, } n = 0 \\ (n-1)! * n & \text{si, } n \geq 1 \end{cases}$$

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

# Factorial





# Factorial

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

Llamada	n	valor parcial	return
1	5	?	factorial(4) * 5

# Factorial

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

Llamada	n	valor parcial	return
1	5	?	factorial(4) * 5
2	4	?	factorial(3) * 4

# Factorial

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

Llamada	n	valor parcial	return
1	5	?	factorial(4) * 5
2	4	?	factorial(3) * 4
3	3	?	factorial(2) * 3

# Factorial

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

Llamada	n	valor parcial	return
1	5	?	factorial(4) * 5
2	4	?	factorial(3) * 4
3	3	?	factorial(2) * 3
4	2	?	factorial(1) * 2

# Factorial

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

Llamada	n	valor parcial	return
1	5	?	factorial(4) * 5
2	4	?	factorial(3) * 4
3	3	?	factorial(2) * 3
4	2	?	factorial(1) * 2
5	1	?	factorial(0) * 1

# Factorial

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

Llamada	n	valor parcial	return
1	5	?	factorial(4) * 5
2	4	?	factorial(3) * 4
3	3	?	factorial(2) * 3
4	2	?	factorial(1) * 2
5	1	?	factorial(0) * 1
6	0	1	1

# Factorial

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

Llamada	n	valor parcial	return
1	5	?	factorial(4) * 5
2	4	?	factorial(3) * 4
3	3	?	factorial(2) * 3
4	2	?	factorial(1) * 2
5	1	<b>1 * 1 = 1</b>	<b>factorial(0) * 1</b>
6	0	<b>1</b>	<b>1</b>

# Factorial

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

Llamada	n	valor parcial	return
1	5	?	factorial(4) * 5
2	4	?	factorial(3) * 4
3	3	?	factorial(2) * 3
4	2	<b>1 * 2 = 2</b>	<b>factorial(1) * 2</b>
5	1	<b>1 * 1 = 1</b>	factorial(0) * 1
6	0	1	1



# Factorial

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

Llamada	n	valor parcial	return
1	5	?	factorial(4) * 5
2	4	?	factorial(3) * 4
3	3	<b>2 * 3 = 6</b>	<b>factorial(2) * 3</b>
4	2	<b>1 * 2 = 2</b>	factorial(1) * 2
5	1	<b>1 * 1 = 1</b>	factorial(0) * 1
6	0	1	1

# Factorial

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

Llamada	n	valor parcial	return
1	5	?	factorial(4) * 5
2	4	<b>6</b> * 4 = 24	<b>factorial(3)</b> * 4
3	3	2 * 3 = <b>6</b>	factorial(2) * 3
4	2	1 * 2 = 2	factorial(1) * 2
5	1	1 * 1 = 1	factorial(0) * 1
6	0	1	1

# Factorial

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

Llamada	n	valor parcial	return
1	5	<b>24</b> * 5 = 120	<b>factorial(4)</b> * 5
2	4	6 * 4 = <b>24</b>	factorial(3) * 4
3	3	2 * 3 = 6	factorial(2) * 3
4	2	1 * 2 = 2	factorial(1) * 2
5	1	1 * 1 = 1	factorial(0) * 1
6	0	1	1

# Factorial

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

El factorial de **n = 5** es **120**

Llamada	n	valor parcial	return
1	5	$24 * 5 = 120$	<code>factorial(4) * 5</code>
2	4	$6 * 4 = 24$	<code>factorial(3) * 4</code>
3	3	$2 * 3 = 6$	<code>factorial(2) * 3</code>
4	2	$1 * 2 = 2$	<code>factorial(1) * 2</code>
5	1	$1 * 1 = 1$	<code>factorial(0) * 1</code>
6	0	1	1

# Fibonacci

- Es una serie donde cada número es calculado como la suma de los dos anteriores.
- 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

<b>n</b>	<b>0</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>	<b>6</b>	<b>7</b>	<b>8</b>
<b>Valor</b>	0	1	1	2	3	5	8	13	21

# Fibonacci

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

<b>n</b>	<b>0</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>	<b>6</b>	<b>7</b>	<b>8</b>
<b>Valor</b>	0	1	1	2	3	5	8	13	21

```
int n = 4;  
int resultado = 0;  
int anterior1 = 1; int anterior2 = 0;  
for(int i=0; i<n; i++) {  
    resultado = anterior1 + anterior2;  
    anterior1 = anterior2;  
    anterior2 = resultado;  
}  
cout<<resultado<<endl;
```

# Fibonacci

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

n	0	1	2	3	4	5	6	7	8
Valor	0	1	1	2	3	5	8	13	21

a          b          c

```
int n = 8;
int a =0; int b = 1; int c;
for(int i=0; i<=n; i++) {
    c = a + b
    a = b
    b = c
}
cout<<c<<endl;
```

```
int fibonacci(int n) {
    if(n == 0 || n == 1) return n;

    int a =0; int b = 1; int c;

    for(int i=0; i<=n-2; i++) {
        c = a + b;
        a = b;
        b = c;
    }
    return c;
}
```

# Fibonacci

n	0	1	2	3	4	5	6	7	8
Valor	0	1	1	2	3	5	8	13	21

- Si  $n == 0$  ||  $n == 1$  entonces resultado es  $n$
- Si  $n \geq 2$  entonces el resultado es  $f(n-1) + f(n-2)$

```
int fibonacci(int n) {  
    if(n == 0 || n == 1) return n;  
  
    int a = 0; int b = 1; int c;  
  
    for(int i=0; i<=n-2; i++) {  
        c = a + b;  
        a = b;  
        b = c;  
    }  
    return c;  
}
```

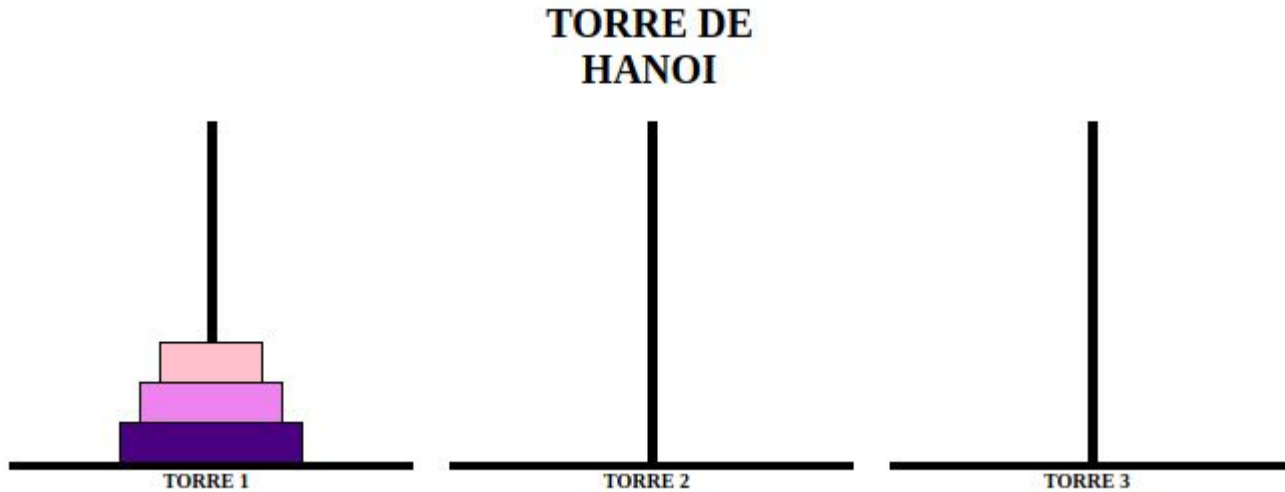
```
int fibonacci(int n) {  
    if(n == 0 || n == 1) return n;  
  
    return fibonacci(n-1) + fibonacci(n-2);  
}
```



# Divide y vencerás

- Es una técnica que consiste en transformar (dividir) un problema de tamaño  $n$  en problemas más pequeños, de tamaño menor que  $n$  pero similares al problema original.
- Al resolver los subproblemas y combinar las soluciones, se logra resolver el problema completo de una manera más fácil (vencerás).

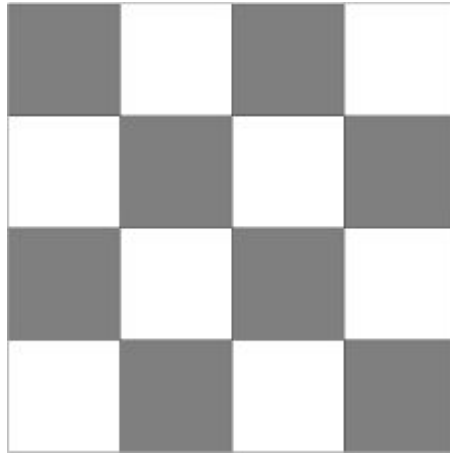
# Torres de Hanoi



# Backtracking

- Existen problemas que requieren probar de manera sistemática todas las posibilidades que pueden existir para encontrar una solución.
- Los algoritmos de Backtracking (vuelta atrás) hacen uso recursividad para probar cada una de las posibilidades que nos pueden llevar a la solución.
- Este método realiza una búsqueda exhaustiva y sistemática, de una posible solución al problema planteado.

# N-Reinas



# N-Reinas

