Recursividad

Algoritmos y Programación III - UNPAZ

Recursividad

- **Definición.** Una función es recursiva si en algún momento del cuerpo de la función se llama a sí misma.

Recursividad

- **Definición.** Una función es recursiva si en algún momento del cuerpo de la función se llama a sí misma.

```
1  static int funcion(int n)
2  {
3      ...
4      int a = funcion(n-1);
5      // Hacemos algo con a
6      ...
7      return (...);
8  }
```

Recursividad

- **Definición.** Una función es recursiva si en algún momento del cuerpo de la función se llama a sí misma.

```
1  static int funcion(int n)
2  {
3      ...
4      int a = funcion(n-1);
5      // Hacemos algo con a
6      ...
7      return (...);
8  }
```

- Para el compilador y la ejecución del código, no hay diferencia entre llamar a otra función y a la misma función.

Factorial

- Muchas operaciones matemáticas se definen de manera recursiva. Una de ellas es el factorial (que se escribe $n!$):

$$\begin{aligned} 0! &= 1 \\ n! &= n \cdot (n-1)! \end{aligned}$$

Factorial

- Muchas operaciones matemáticas se definen de manera recursiva. Una de ellas es el factorial (que se escribe $n!$):

$$\begin{aligned} 0! &= 1 \\ n! &= n \cdot (n-1)! \end{aligned}$$

- En castellano esto se lee como *"El factorial de* 0 *es* 1*. El factorial de n es el producto de n por el factorial de* $n-1$*."*.

Factorial

- Muchas operaciones matemáticas se definen de manera recursiva. Una de ellas es el factorial (que se escribe $n!$):

$$\begin{aligned} 0! &= 1 \\ n! &= n \cdot (n-1)! \end{aligned}$$

- En castellano esto se lee como *"El factorial de* 0 *es* 1*. El factorial de n es el producto de n por el factorial de* $n-1$*."*.
- **Ejemplo:**

$$\begin{aligned} 4! &= 4 \cdot 3! \\ &= 4 \cdot (3 \cdot 2!) \\ &= 4 \cdot (3 \cdot (2 \times 1!)) \\ &= 4 \cdot (3 \cdot (2 \times (1 \times 0!))) \\ &= 4 \cdot (3 \cdot (2 \times (1 \times 1))) \\ &= 24 \end{aligned}$$

Factorial

- En Java, podemos escribir una función recursiva que implemente **directamente** esta definición:

```
1  public static int factorial(int n)
2  {
3    if (n == 0)
4      return 1;
5    else
6      return n * factorial(n-1);
7  }
```

Factorial

- Para analizar su comportamiento, la escribimos de la siguiente forma equivalente:
-

```
1   public static int factorial(int n)
2   {
3     if (n == 0)
4       return 1;
5     else
6     {
7       int recursion = factorial (n-1);
8       int resultado = n * recursion;
9       return resultado;
10    }
11  }
```

Factorial

Analicemos cómo calcularía el factorial de 3:

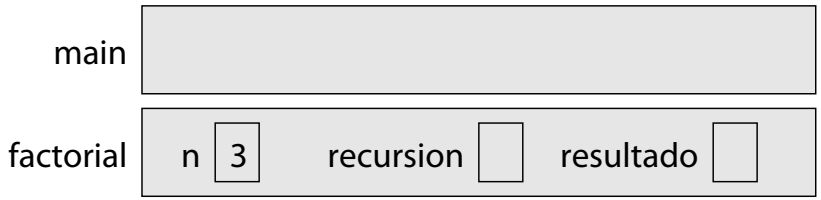
```
1  public static void main(String[] args)
2  {
3    factorial(3);
4  }
5
6  public static int factorial(int n)
7  {
8    if (n == 0)
9      return 1;
10   else
11   {
12     int recursion = factorial (n-1);
13     int resultado = n * recursion;
14     return resultado;
15   }
16 }
```

main

Factorial

Analicemos cómo calcularía el factorial de 3:

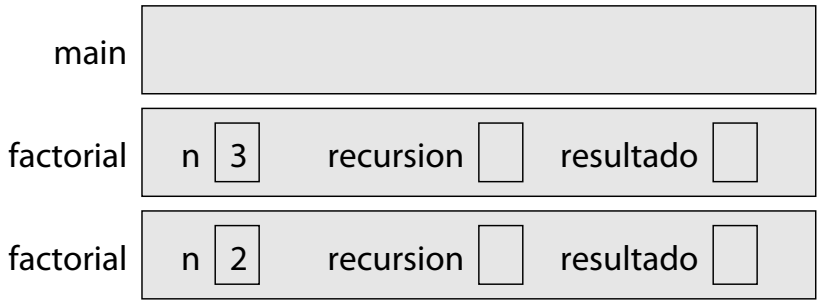
```
1  public static void main(String[] args)
2  {
3    factorial(3);
4  }
5
6  public static int factorial(int n)
7  {
8    if (n == 0)
9      return 1;
10   else
11   {
12     int recursion = factorial (n-1);
13     int resultado = n * recursion;
14     return resultado;
15   }
16 }
```



Factorial

Analicemos cómo calcularía el factorial de 3:

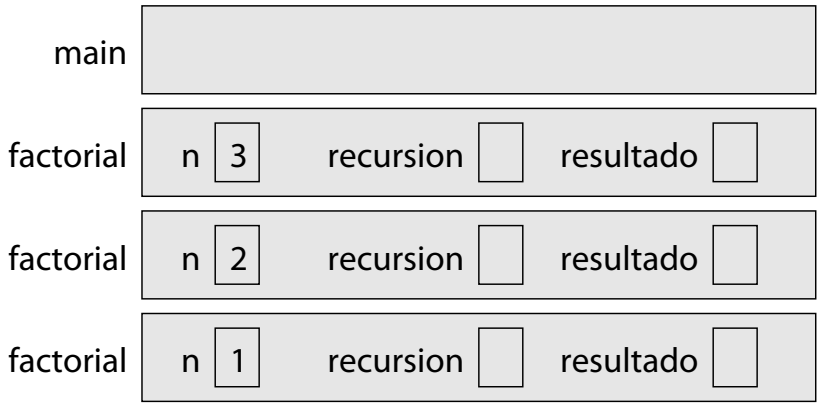
```
1  public static void main(String[] args)
2  {
3    factorial(3);
4  }
5
6  public static int factorial(int n)
7  {
8    if (n == 0)
9      return 1;
10   else
11   {
12     int recursion = factorial (n-1);
13     int resultado = n * recursion;
14     return resultado;
15   }
16 }
```



Factorial

Analicemos cómo calcularía el factorial de 3:

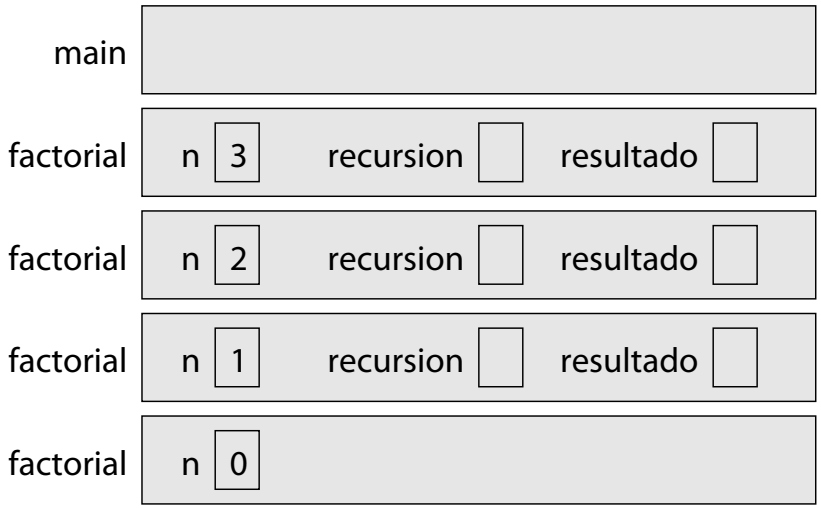
```
1  public static void main(String[] args)
2  {
3    factorial(3);
4  }
5
6  public static int factorial(int n)
7  {
8    if (n == 0)
9      return 1;
10   else
11   {
12     int recursion = factorial (n-1);
13     int resultado = n * recursion;
14     return resultado;
15   }
16 }
```



Factorial

Analicemos cómo calcularía el factorial de 3:

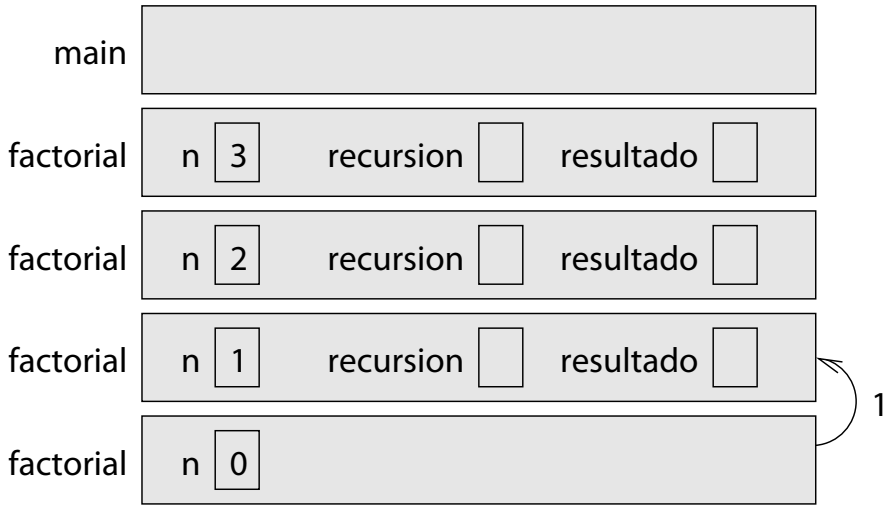
```
1  public static void main(String[] args)
2  {
3    factorial(3);
4  }
5
6  public static int factorial(int n)
7  {
8    if (n == 0)
9      return 1;
10   else
11   {
12     int recursion = factorial (n-1);
13     int resultado = n * recursion;
14     return resultado;
15   }
16 }
```



Factorial

Analicemos cómo calcularía el factorial de 3:

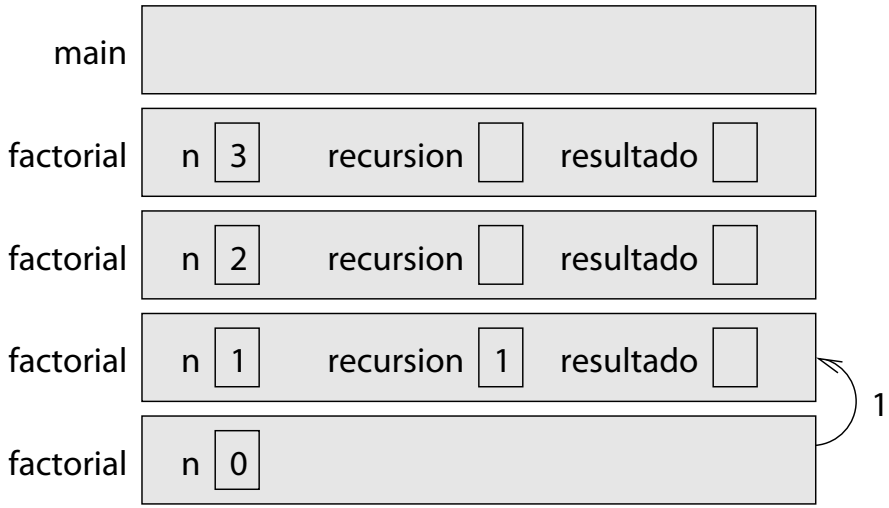
```
1  public static void main(String[] args)
2  {
3    factorial(3);
4  }
5
6  public static int factorial(int n)
7  {
8    if (n == 0)
9      return 1;
10   else
11   {
12     int recursion = factorial (n-1);
13     int resultado = n * recursion;
14     return resultado;
15   }
16 }
```



Factorial

Analicemos cómo calcularía el factorial de 3:

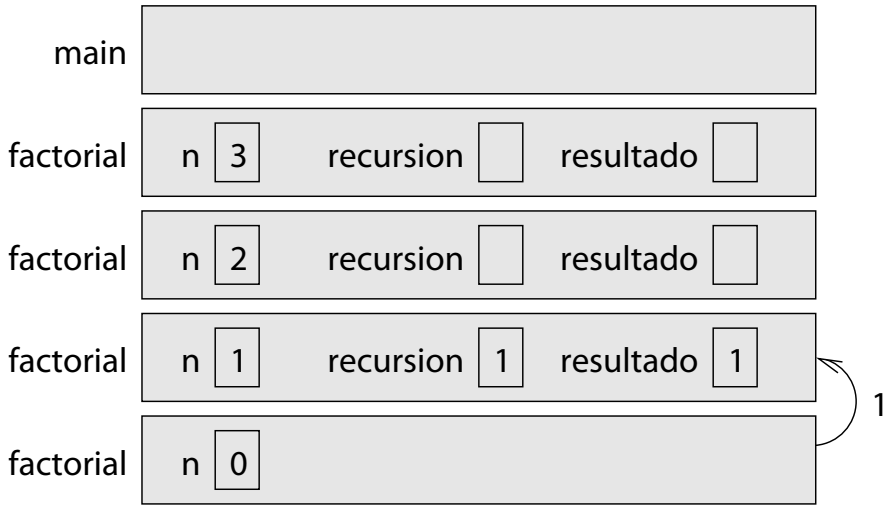
```
1  public static void main(String[] args)
2  {
3    factorial(3);
4  }
5
6  public static int factorial(int n)
7  {
8    if (n == 0)
9      return 1;
10   else
11   {
12     int recursion = factorial (n-1);
13     int resultado = n * recursion;
14     return resultado;
15   }
16 }
```



Factorial

Analicemos cómo calcularía el factorial de 3:

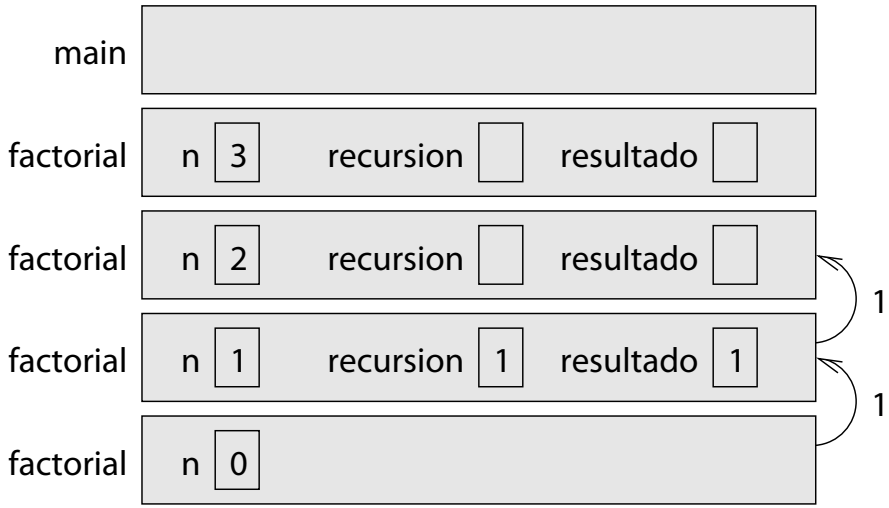
```
1  public static void main(String[] args)
2  {
3    factorial(3);
4  }
5
6  public static int factorial(int n)
7  {
8    if (n == 0)
9      return 1;
10   else
11   {
12     int recursion = factorial (n-1);
13     int resultado = n * recursion;
14     return resultado;
15   }
16 }
```



Factorial

Analicemos cómo calcularía el factorial de 3:

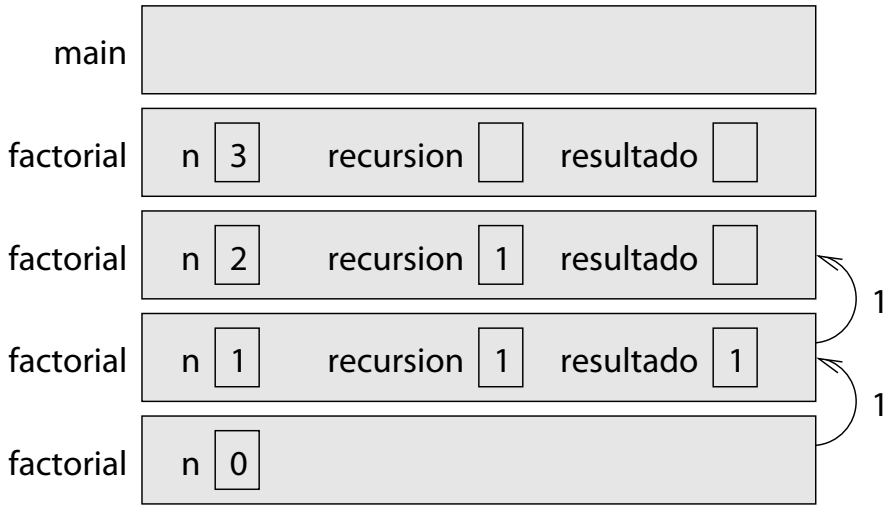
```
1  public static void main(String[] args)
2  {
3    factorial(3);
4  }
5
6  public static int factorial(int n)
7  {
8    if (n == 0)
9      return 1;
10   else
11   {
12     int recursion = factorial (n-1);
13     int resultado = n * recursion;
14     return resultado;
15   }
16 }
```



Factorial

Analicemos cómo calcularía el factorial de 3:

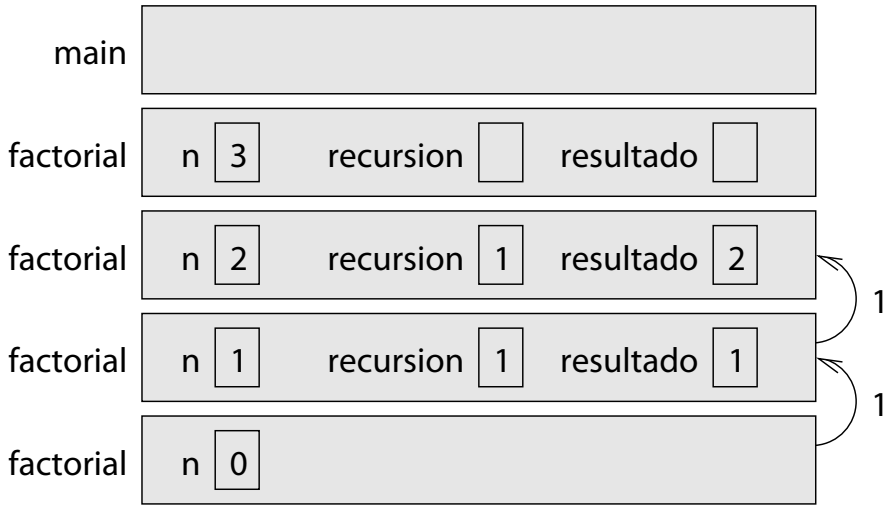
```
1  public static void main(String[] args)
2  {
3    factorial(3);
4  }
5
6  public static int factorial(int n)
7  {
8    if (n == 0)
9      return 1;
10   else
11   {
12     int recursion = factorial (n-1);
13     int resultado = n * recursion;
14     return resultado;
15   }
16 }
```



Factorial

Analicemos cómo calcularía el factorial de 3:

```
1  public static void main(String[] args)
2  {
3    factorial(3);
4  }
5
6  public static int factorial(int n)
7  {
8    if (n == 0)
9      return 1;
10   else
11   {
12     int recursion = factorial (n-1);
13     int resultado = n * recursion;
14     return resultado;
15   }
16 }
```



Factorial

Analicemos cómo calcularía el factorial de 3:

```
1  public static void main(String[] args)
2  {
3    factorial(3);
4  }
5
6  public static int factorial(int n)
7  {
8    if (n == 0)
9      return 1;
10   else
11   {
12     int recursion = factorial (n-1);
13     int resultado = n * recursion;
14     return resultado;
15   }
16 }
```

Factorial

Analicemos cómo calcularía el factorial de 3:

```
1  public static void main(String[] args)
2  {
3    factorial(3);
4  }
5
6  public static int factorial(int n)
7  {
8    if (n == 0)
9      return 1;
10   else
11   {
12     int recursion = factorial (n-1);
13     int resultado = n * recursion;
14     return resultado;
15   }
16 }
```

Factorial

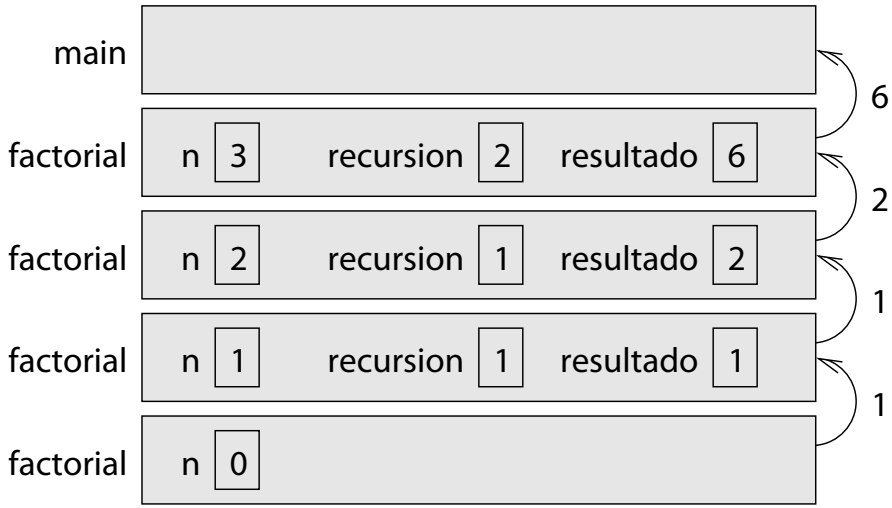
Analicemos cómo calcularía el factorial de 3:

```
1  public static void main(String[] args)
2  {
3    factorial(3);
4  }
5
6  public static int factorial(int n)
7  {
8    if (n == 0)
9      return 1;
10   else
11   {
12     int recursion = factorial (n-1);
13     int resultado = n * recursion;
14     return resultado;
15   }
16 }
```

Factorial

Analicemos cómo calcularía el factorial de 3:

```
1  public static void main(String[] args)
2  {
3    factorial(3);
4  }
5
6  public static int factorial(int n)
7  {
8    if (n == 0)
9      return 1;
10   else
11   {
12     int recursion = factorial (n-1);
13     int resultado = n * recursion;
14     return resultado;
15   }
16 }
```



Características de las funciones recursivas

- Hay un caso en el que la función se resuelve (trivialmente) sin necesidad de llamarse a sí misma. Este caso se denomina caso base.

Características de las funciones recursivas

- Hay un caso en el que la función se resuelve (trivialmente) sin necesidad de llamarse a sí misma. Este caso se denomina caso base.
- La llamada recursiva se realiza con un parámetro "más chico" que el que recibe.

Características de las funciones recursivas

- Hay un caso en el que la función se resuelve (trivialmente) sin necesidad de llamarse a sí misma. Este caso se denomina caso base.
- La llamada recursiva se realiza con un parámetro "más chico" que el que recibe.
- ¿Qué sucede si la llamada recursiva se realiza con un parámetro "más grande" que el recibido?

```
1   public static int factorial(int n)
2   {
3     if (n == 0)
4       return 1;
5     else
6       return n * factorial(n+1);
7   }
```

Características de las funciones recursivas

- Cuando hablamos de un parámetro “más chico”, no hablamos particularmente del valor. Lo importante es que la llamada recursiva se acerque al caso base.

Características de las funciones recursivas

- Cuando hablamos de un parámetro "más chico", no hablamos particularmente del valor. Lo importante es que la llamada recursiva se acerque al caso base.

```
1   public static int funcion(int n)
2   {
3     if (n == 100)
4       return 1;
5     else
6       return 1 + funcion(n+1);
7   }
```

Características de las funciones recursivas

- Cuando hablamos de un parámetro "más chico", no hablamos particularmente del valor. Lo importante es que la llamada recursiva se acerque al caso base.

```
1   public static int funcion(int n)
2   {
3     if (n == 100)
4       return 1;
5     else
6       return 1 + funcion(n+1);
7   }
```

- Notar que esta función no termina si se ejecuta con $n > 100$. Decimos que en este caso la función se indefine.

Sucesión de Fibonacci

- **Definición.** La sucesión de Fibonacci se define recursivamente del siguiente modo:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2}, \text{ para } n \geq 2 \end{aligned}$$

Sucesión de Fibonacci

- **Definición.** La sucesión de Fibonacci se define recursivamente del siguiente modo:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2}, \text{ para } n \geq 2 \end{aligned}$$

- Los primeros términos de la sucesión son 0, 1, 1, 2, 3, 5, 8, 13, 21.

Sucesión de Fibonacci

- Intentemos implementar una función recursiva que dado un entero n calcule el n-ésimo término de la sucesión...

Sucesión de Fibonacci

- Intentemos implementar una función recursiva que dado un entero n calcule el n-ésimo término de la sucesión...
-

```
1  static int fib(int n)
2  {
3     if( n == 0 )
4        return 0;
5     else if( n == 1 )
6        return 1;
7     else
8        return fib(n-1) + fib(n-2);
9  }
```

Sucesión de Fibonacci

- Una versión más compacta ...

```
1  static int fib(int n)
2  {
3     return n < 2 ? n : fib(n-1) + fib(n-2);
4  }
```

Conclusión

- Utilizando recursividad podemos resolver las mismas funciones que se pueden implementar con ciclos.

Conclusión

- Utilizando recursividad podemos resolver las mismas funciones que se pueden implementar con ciclos.
- En algunos casos, la recursividad permite escribir código más sencillo de entender.

Conclusión

- Utilizando recursividad podemos resolver las mismas funciones que se pueden implementar con ciclos.
- En algunos casos, la recursividad permite escribir código más sencillo de entender.
- ¿Cómo programamos una función recursiva?

Conclusión

- Utilizando recursividad podemos resolver las mismas funciones que se pueden implementar con ciclos.
- En algunos casos, la recursividad permite escribir código más sencillo de entender.
- ¿Cómo programamos una función recursiva?
  1. Tenemos que evaluar si estamos en el caso base, y si es así, devolvemos el valor apropiado.

Conclusión

- Utilizando recursividad podemos resolver las mismas funciones que se pueden implementar con ciclos.
- En algunos casos, la recursividad permite escribir código más sencillo de entender.
- ¿Cómo programamos una función recursiva?
  1. Tenemos que evaluar si estamos en el caso base, y si es así, devolvemos el valor apropiado.
  2. Para el resto de los casos, llamamos a la función con un valor más cercano al caso base y con lo que devuelve calculamos el resultado para el caso actual.