Programación 2

Lección 3. Introducción a la recursividad

1. **Definiciones recursivas**
   - ✓ Número natural y número entero
   - ✓ Múltiplos de 7
   - ✓ Secuencia de datos
   - ✓ Factorial de un número natural
   - ✓ Sucesión de Fibonacci
2. **Algoritmos recursivos**
   - ✓ Función factorial
   - ✓ Función fibonacci

1. Definiciones recursivas

Muchas estructuras matemáticas, estructuras de datos y algoritmos admiten sencillas definiciones recursivas. En esta lección se presenta una ilustrativa colección de ejemplos.

¿Qué es un número natural?

Base de la definición: El **0** es un número natural

Recurrencia: Si **n** es un número natural

entonces **n + 1** es también un número natural

De la definición anterior se deduce que el conjunto de los números naturales es { ?, ?, ?, … }

¿Qué es un número natural?

Base de la definición: El **0** es un número natural

Recurrencia: Si **n** es un número natural

entonces **n + 1** es también un número natural

De la definición anterior se deduce que el conjunto de los números naturales es { 0, 1, 2, 3, 4, ... }

¿Hay más formas de definir los naturales?

Dos definiciones recursivas de los números naturales

Base de la definición: El **0** es un número natural
Recurrencia: Si **n** es un número natural
entonces **n + 1** es también un número natural

De la definición anterior se deduce que el conjunto de los números naturales es { 0, 1, 2, 3, 4, ... }

¿Hay más formas de definir los naturales? Sí, por ejemplo:

Base de la definición: El **0** es un número natural
Base de la definición: El **1** es un número natural
Recurrencia: Si **n** es un número natural
entonces **n + 2** es también un número natural

Estas definiciones constan de **una base** (más o menos amplia) y de **una recurrencia** (más o menos amplia) apoyada en alguna **operación auxiliar**

Una definición de los números enteros

Base de la definición: El ... es un número entero

Recurrencia: Si ... es un número entero
entonces ... es también un número entero

De la definición planteada debe deducirse que el conjunto de los números enteros es { …, -4, -3, -2 , -1, 0, 1, 2, 3, 4, … }

Una definición recursiva de los números enteros

Base de la definición: El **0** es un número entero

Recurrencia: Si **n** es un número entero
entonces **n - 1** es también un número entero

Recurrencia: Si **n** es un número entero
entonces **n + 1** es también un número entero

De la definición propuesta se deduce que el conjunto de los números enteros es:

{ …, -4, -3, -2, -1, 0, 1, 2, 3, 4, … }

Otra definición recursiva de los números enteros

Base de la definición: El **-1** es un número entero

Base de la definición: El **1** es un número entero

Recurrencia: Si **m** y **n son** números enteros
entonces **m + n** es también un número entero

De la definición anterior se deduce que el conjunto de los números enteros es:

{ …, -4, -3, -2, -1, 0, 1, 2, 3, 4, … }

Una definición de los múltiplos de 7

Base: El ... es un número múltiplo de 7

Recurrencia: Si ... es un múltiplo de 7

entonces ... es también múltiplo de 7

De la definición planteada debe deducirse que el conjunto de los números múltiplos de 7 es :

{ …, -21, -14, -7, 0, 7, 14, 21, … }

Una definición recursiva de los múltiplos de 7

Base de la definición: El **0** es un múltiplo de 7

Recurrencia: Si **n** es un múltiplo de 7

entonces **n - 7** es también un múltiplo de 7

Recurrencia: Si **n** es un múltiplo de 7

entonces **n + 7** es también un múltiplo de 7

De la definición propuesta se deduce que el conjunto de los números múltiplos de 7 es:

{ …, -21, -14, -7, 0, 7, 14, 21, … }

Otra definición recursiva de los múltiplos de 7

Base: El **7** es un número múltiplo de 7

Base: El **-7** es un número múltiplo de 7

Recurrencia: Si **m** y **n** son múltiplos de 7
entonces **m + n** es también un múltiplo de 7

De la definición anterior se deduce que el conjunto de los números múltiplos de 7 es:

{ ... , -21, -14, -7, 0, 7, 14, 21, ... }

Definición recursiva de secuencia de datos de tipo T

Base: Un número nulo de datos de tipo *T* define la secuencia <> de datos de tipo *T*

Recurrencia: Si **s** es una secuencia de datos de tipo *T* entonces el resultado de concatenar tras **s** un dato de tipo *T* es también una secuencia de datos de tipo *T*

De la definición anterior se deduce que si *d1*, *d2*, *d3* y *d4* son datos de tipo *T* entonces también son secuencias de datos de tipo *T*:

- <>
- <d1> <d2> <d3> <d4> …
- <d1,d4> <d3,d2> <d2,d2> …
- <d1,d4,d1> <d2,d1,d3> …
- <d2,d2,d1,d3> <d4,d2,d1,d3> …
- <d2,d1,d3,d2,d2> <d2,d1,d3,d2,d1> …
- etc., etc.

Una definición recursiva del factorial de un número, n!

Base de la definición: El factorial de 0 es igual a 1, es decir, 0! = 1

Recurrencia: Para n > 0 se define el factorial de n del modo siguiente n! = n x (n-1)!

Deducir, a partir la definición anterior, los valores de:

- 0! = ?
- 1! = ?
- 2! = ?
- 3! = ?
- 4! = ?
- 5! = ?
- Etc., etc.

Una definición recursiva del factorial de un número, n!

Base de la definición: El factorial de 0 es igual a 1, es decir, 0! = 1

Recurrencia: Para n > 0 se define el factorial de n del modo siguiente n! = n x (n-1)!

De la definición anterior se deduce que:

- 0! = 1 // Se deduce de la base
- 1! = 1 x 0! = 1 // Se deduce de la recurrencia (y de la base)
- 2! = 2 x 1! = 2 // Se deduce de la recurrencia (y de la base)
- 3! = 3 x 2! = 6 // Se deduce de la recurrencia (y de la base)
- 4! = 4 x 3! = 24 // Se deduce de la recurrencia (y de la base)
- 5! = 5 x 4! = 120 // Se deduce de la recurrencia (y de la base)
- Etc., etc., etc.

Una definición recursiva de la sucesión de Fibonacci

Base de la definición: El 0 es primer elemento de la sucesión de Fibonacci

Base de la definición: El 1 es segundo elemento de la sucesión de Fibonacci

Recurrencia: Para i > 2 el i-ésimo elemento de la sucesión de Fibonacci es igual a la suma de los dos elementos que le preceden en dicha sucesión

Deducir, a partir de la definición anterior, los naturales que integran la sucesión de Fibonacci:

{ ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, … }

Una definición recursiva de la sucesión de Fibonacci

Base de la definición: El 0 es primer elemento de la sucesión de Fibonacci

Base de la definición: El 1 es segundo elemento de la sucesión de Fibonacci

Recurrencia: Para i > 2 el i-ésimo elemento de la sucesión de Fibonacci es igual a la suma de los dos elementos que le preceden en dicha sucesión

De la definición anterior se deduce que los naturales que integran la sucesión de Fibonacci son los siguientes:

{ 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, … }

2. Algoritmos recursivos

Función que devuelve el factorial de un natural

```
// Pre: n ≥ 0
// Post: factorial(n) = (∏α∈[1,n].α)
int factorial (const int n) {
   if (n == 0) {
      return 1;                          // Solución base
   }
   else {
      // n - 1 ≥ 0
      // factorial(n-1) = (∏α∈[1,n-1].α)
      return n * factorial(n-1);      // Solución recurrente
   }
}
```

Otra función recursiva que calcula el factorial de un natural

```
// Pre: n ≥ 0
// Post: resultado = (∏α∈[1,n].α)
void factorial (const int n, int& resultado) {
   if (n == 0) {
      resultado = 1;                    // Solución base
   }
   else {
      // n - 1 ≥ 0
      factorial(n-1, resultado);        // Solución recurrente
      // resultado = (∏α∈[1,n-1].α)
      resultado = n * resultado;
   }
}
```

Una función recursiva que calcula cualquiera de los elementos de una sucesión de Fibonacci

```
// Pre: n ≥ 1
// Post: (n = 1 ∨ n = 2 → fibonacci(n) = n - 1) ∧
//       (n > 2 → fibonacci(n) =
//                        fibonacci(n-2) + fibonacci(n-1))
int fibonacci (const int n) {
   if (n <= 2) {                    // Solución base
      // n = 1 ∨ n = 2
      return n - 1;
   }
   else  {                          // Solución recurrente
      // n > 2
      return fibonacci(n-2) + fibonacci(n-1);
   }
}
```

```
// Pre: n ≥ 1
// Post: (n = 1 ∨ n = 2 → fibonacci(n) = n - 1) ∧
//       (n > 2 → fibonacci(n) =
//                        fibonacci(n-2) + fibonacci(n-1))
int fibonacci (const int n) {
   if (n <= 2) {                  // Solución base
      // n = 1 ∨ n = 2
      return n - 1;
   }
   else  {                        // Solución recurrente
      // n > 2
      return fibonacci(n-2) + fibonacci(n-1);
   }
}
```

Una duda: ¿La eficiencia de la función anterior es satisfactoria?