# UE_6: Problemas de recorrido

## Definición.

Llamamos problemas de recorrido a todos aquellos para los que se necesita recorrer un dominio (una estructura de datos) para su resolución. Es decir, o no existe un algoritmo recogido en una fórmula, que lo convierta en un problema de solución directa, o bien, no queremos utilizarlo para redactar la solución. El algoritmo requiere conocer el dominio y establecer ciertos cálculos.
El problema del cálculo del factorial de un número natural está en esta categoría de problemas. No disponemos de una fórmula, si bien, de su definición emana una regla que nos da la pista para calcularlo. Dicha regla establece que hay que realizar multiplicaciones sucesivas entre los números que existen en el rango de naturales que va desde el 1 hasta el propio número.
Para calcular el 3! multiplicamos todos los números que hay en el intervalo [1,3] ==> 3! = 1.2.3 = 6

En todos los problemas de recorrido aparecen estos dos aspectos: el dominio de recorrido y el cálculo (la operación que se realiza con los elementos de dicho dominio).

El dominio hay que saber recorrerlo, hay que "entrar", "visitar" todos los elementos sin dejarse ninguno y "salir". Esto se consigue cuando contestamos a las siguientes preguntas:

1. ¿Con qué elemento empiezo?
2. ¿Cómo avanzo de uno en otro?
3. ¿Con qué elemento acabo?

En el problema del factorial se recorre el intervalo cerrado de naturales entre 1 y el propio número ([1,n]); he empezado en el "1", he avanzado sumando 1 a cada número y he acabado en el "n". Atención, también podría haber empezado en el "n", avanzado restando 1 a cada número y acabado en el "1" (3! = 3.2.1 = 6).

El cálculo siempre implica a una operación. Como hay que efectuarla sucesivas veces, necesitamos que tenga dos propiedades: que la operación sea asociativa y que tenga elemento neutro. Asociativa para poder recombinar varias aplicaciones de ella sin perder congruencia y el elemento neutro para garantizar un valor sensato cuando el dominio no tiene elementos (en el instante de la primera aplicación de la operación).
En el problema del factorial, la operación de recombinación es la multiplicación, que tiene elemento neutro (el 1) y la propiedad asociativa ( 1*(2*3) = ((1*2)*3 ).

## Especificación.

Para poder especificar problemas de recorrido necesitamos una notación que exprese los dos aspectos. Lo más fácil de especificar es la operación, pues en la mayoría de los casos se tratará de una operación aritmético-lógica. En otro caso podremos definir dicha operación como un problema más (un subproblema) con su correspondiente especificación FUNCIÓN-PRE-POST. En matemáticas se usan "cuantificadores" para expresar un recorrido sobre un dominio. Son macro operadores que actúan sobre muchos operandos (no, como los habituales que o son monarios -un operando- o binarios –dos operandos). Estos cuantificadores son operadores "m_arios" – actúan sobre "m" operandos. Cada cuantificador está asociado a una operación. El sumatorio (∑) a la suma, el productorio (∏) a la multiplicación, el existencial (□) a la disyunción, etc.

El problema del factorial lo podemos especificar usando el productorio `n! = ∏ α ∈ [1,n] • α`
Lo que hemos escrito se lee así: el factorial de n es lo mismo que multiplicar todos los números de [1,n]. La letra griega α representa a los números del intervalo. Se dice que es la variable ligada al cuantificador y su nombre es indiferente. Hay un símbolo que representa la pertenencia de un elemento a un conjunto (el ∈) y otro (el •) que separa el dominio de recorrido del problema, de la propiedad o expresión que define a lo que se va a multiplicar.

En general puede ocurrir que no se aplique la operación a "todos" los elementos del dominio, sino que se descarten algunos de ellos (se filtren). En ese caso, la notación incorpora ese filtro como una función sobre los elementos y se añade entre el

dominio y la propiedad sepatrándolos con el símbolo "|". Por ejemplo, el problema de la suma de los números impares de entre los "n" primeros, se especificaría: `sumaImparesN(n) = ∑ α ϵ [1,n] | EsImpar(α) • α`

En general, toda expresión cuantificada se puede expresar como:

**ζ α ϵ |Đ | φ(α) • ρ(α)**

Cuyos elementos son:

- El cuantificador ( ζ ) que lleva oculta su operación asociada.
- Su variable ligada ( α ).
- El dominio de actuación de la variable ( |Đ ).
- Un filtro (opcional) sobre los elementos del dominio ( φ ).
- Una propiedad sobre los elementos ( ρ ).

Y que se lee:

> "Efectúa sucesivamente (con la operación asociada a ζ) la propiedad ρ(α) de los elementos α del dominio Đ que cumplan el filtro φ(α)"

Y todo problema de recorrido se puede especificar en función de una expresión cuantificada.

## Taxonomía de problemas de recorrido.

Hay cuatro grandes grupos de problemas:

- Acumulación: Con operaciones aritméticas como la '+' y el '*'. Son problemas en donde el resultado esperado es del tipo de la propiedad que se efectúa, la ρ (α).
- Búsqueda: Con operaciones lógicas como la '/\' y la '\/'. Son problemas en donde el resultado es un valor lógico ("buscar si se cumple algo"), o un valor del dominio ("buscar qué elemento cumple algo"), o un número natural ("buscar la posición del elemento que cumple algo"). La ρ (α) es éso que se debe cumplir.
- Extremales: Con operaciones de comparación como '<' y '>'. Son problemas en donde el resultado es un valor del tipo de ρ (α) ("la máxima o mínima cuestión"), o un valor del dominio ("qué elemento hace máxima o mínima una cuestión"), o un número natural ("la posición del elemento que hace máxima o mínima una cuestión"). La ρ (α) es éso que se maximiza o minimiza.
- Construcción: Con operaciones de construcción de dominios. Son problemas en donde el resultado es un dominio como el de partida, pero filtrado (la ρ (α) es el filtro) o es de otro tipo (la ρ (α) hace de función de transformación de tipos).

Y varios subgrupos para cada uno de ellos. En la tabla siguiente se muestran las propiedades de los más representativos (de los cabezas de familia):

| Clase Problema | Cuantificador | Operación | Elemento neutro | Propiedades |
|---|---|---|---|---|
| Acumulación | Sumatorio (SUM) | Suma ('+') | 0 | Conmutativa, Asociativa |
| Acumulación | Productorio (MUL) | Multiplicación ('*') | 1 | Conmutativa, Asociativa |
| Búsqueda | ExisteAlgún(ALGÚN) | Disyunción ('\/') | Falso | Conmutativa, Asociativa |
| Búsqueda | ParaTodo (TODOS) | Conjunción ('/\') | Cierto | Conmutativa, Asociativa |
| Extremales | Máximo (MAX) | "Mayor2 (x,y)" | - | Asociativa |
| Extremales | Mínimo (MIN) | "Menor2 (x,y)" | - | Asociativa |
| Construcción | Filtrar (FIL) | "Construir (x,\|Đ)" | "Vacío" | - |
| Construcción | Aplicar (MAP) | "Construir (x,\|Đ)" | "Vacío" | - |

## Codificación.

Para programar algoritmos sobre problemas de recorrido, los lenguajes de programación disponen de dos mecanismos de cómputo: la recursividad y los bucles.

**1.-** La *recursividad* es un mecanismo de definición implícita donde en la definición, cabe lo definido. Aparentemente, dicho mecanismo está en contra de lo que se entiende por una definición bien hecha, pero no es así porque existe la manera en que la recursividad bien empleada define completamente un mecanismo. La clave está en dejar explicitadas las condiciones por las que se deja de aplicar la propia definición recursiva y en su lugar, lo que se aplica es una acción o un axioma o valor conocidos; completando así el proceso de actuación del mecanismo.En el ejemplo del factorial.

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \cdot (n-1)! & \text{si } n > 0 \end{cases}$$

En esta definición hay muchas cosas sumergidas que conviene reflotar:

Es una definición (función con dominio en |N) en dos trozos. Cada trozo se especifica con una regla. Cada regla tiene dos partes: un predicado que indica para qué valores del dominio de definición debe activarse dicha regla; y una expresión que representa al valor de la definición factorial que se debe aplicar si se satisface su predicado. Una regla aplica la propia definición (la recursividad) y la otra, no. Esta definición propone un método de cálculo para el factorial de cualquier número. Ejemplo: 3! se calcula aplicando tres veces consecutivas la segunda regla y una vez la primera.

3! = (2ª regla) = 3 · 2!
  2! = (2ª regla) = 2 · 1!
    1! = (2ª regla) = 1 · 0!
      0! = (1ª regla) = 1  ahora se pueden completar las aplicaciones pendientes dando marcha atrás.
    1! = 1 · 0! = 1 · 1 = 1
  2! = 2 · 1! = 2 · 1 = 2
3! = 3 · 2! = 3 · 2 = 6

En los lenguajes de programación se permiten las funciones recursivas (las que se invocan a sí mismas) . Las distintas reglas se codifican con una sentencia alternativa (un IF-THEN-ELSE), donde las condiciones son las guardas de las reglas y los cálculos se realizan en las alternativas dentro de sentencias RETURN. El dominio de recorrido debe estar representado por uno o varios parámetros formales. En cada llamada se realiza la recombinación con un elemento del dominio y cada llamada se hace con un elemento menos, para garantizar que se alcanza el último elemento, se selecciona la alternativa que no tiene llamada recursiva y el proceso termina.

```
FUNCION factorial (Natural n) ----> Natural
PRE: cierto
POST1: resultado = n!
POST2: resultado = MUL j IN [1,n] . j
POST3:
                  | 1                      si n = 0
  factorial (n) = |
                  | n * factorial (n-1)    si n > 0
```

Que en lenguaje Java sería:

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

**2.-** Los *bucle*s son construcciones sintácticas que permiten la repetición de un bloque de sentencias. La repetición se controla con una condición, que establece cuando se acaba de recorrer el dominio (cuando se llega al último elemento). La recombinación se realiza en el bloque del bucle sobre una variable. Hay dos modelos de bucles, los que llevan la condición al principio (MIENTRAS-HACER) y los que la llevan al final (HACER-MIENTRAS). El programador debe garantizar la existencia de, al menos, dos variables: una para el recorrido del dominio (implicada en la condición del bucle) y otra para la recombinación de los cálculos. De igual manera, debe garantizar que dichas variables estén iniciadas antes de entrar en el bucle: una con el primer elemento del dominio y la otra con el elemento neutro de la operación. De manera esquemática siempre hay 2 variables y 4 fases:

Variables:

1. *elemento*: la que contendrá al elemento actual del dominio.
2. *resultado*: la que contendrá el valor calculado por la recombinación.

Fases:

1. *Iniciación*: de elemento (el primero del dominio) y resultado (el elemento neutro de la operación de recombinación).
2. *Condición*: expresión lógica sobre lo que queda del dominio.
3. *Recombinación*: lo que se hace en resultado con los elementos del dominio.
4. *Modificación*: del elemento actual del dominio para pasar al siguiente.

**Bucle** MIENTRAS-HACER

```
while (<<expresión_booleana>>)
  <<Bloque de sentencias>>;
```

Funcionamiento: Se evalúa la expresión booleana. Si el resultado es falso, se ignora el bloque y se sale del bucle. Si el resultado es cierto, se ejecuta el bloque y vuelve a evaluarse la expresión booleana.

Esquema

```
+---------------------+
| Iniciación;         |
+---------------------+
| while  (Condición)  |
+---+-----------------+
    | Recombinación;  |
    +-----------------+
    | Modificación;   |
    +-----------------+
```

**Bucle** HACER-MIENTRAS

```
do
   <<Bloque de sentencias>>;
while (<<expresión_booleana>>);
```

Funcionamiento: Se ejecuta el bloque Y se evalúa la expresión booleana. Si el resultado es falso, se sale del bucle. Si el resultado es cierto, vuelve a ejecutarse el bloque y la expresión booleana.

Esquema

```
+---------------------+
| Iniciación;         |
+---------------------+
| do                  |
+---------------------+
    | Recombinación;  |
    +-----------------+
    | Modificación;   |
+---+-----------------+
| while (Condición);  |
+---+-----------------+
```

La diferencia entre los dos estriba en que el bucle HACER-MIENTRAS ejecuta el bloque al menos una vez frente al MIENTRAS-HACER que podría no hacerlo ninguna.

También suele haber otro tipo de bucle que declara, inicia y controla variables intrínsecas. El bucle FOR. Recomendado cuando se conozca el número de elementos del dominio (cuando sea un intervalo de naturales).

**Bucle** FOR

```
for (<<declaración/iniciación>>;<<expresión_booleana>>;<<modificación>>)
  <<Bloque de sentencias>>;
```

Funcionamiento: Como el del bucle MIENTRAS-HACER.

Esquema

```
+-----------------------------------------+
| Iniciación; (solo del resultado)        |
+-----------------------------------------+
| for (Iniciación; Condición; Modificación) |
+---+-------------------------------------+
    | Recombinación;                      |
    +-------------------------------------+
```

Ejemplo de utilización de los tres bucles para el ejemplo del factorial:

```
int factorial (int n)
{
  int resultado = 1;
  int i = 1;
  while (i <= n)
  {
    resultado = resultado * i;
    i = i + 1;
  }
  return resultado;
}
```

```
int factorial (int n)
{
  int resultado = 1;
  int i = 1;
  do
  {
    resultado = resultado * i;
    i = i + 1;
  }
  while (i <= n)
  return resultado;
}
```

```
int factorial (int n)
{
  int resultado = 1;
  for (int i = 1; i <= n; i++)
    resultado = resultado * i;
  return resultado;
}
```