# Resumen de recorridos (primer parcial)

La estructura de recorridos permite asegurar que una repetición condicional va a finalizar exitosamente, cumpliendo la tarea esperada. Los recorridos son útiles en los casos en los que hay que procesar múltiples elementos iguales, todos de forma similar.

Una vez identificamos que necesitamos un recorrido, debemos identificar:

- Esquema de recorrido
- Elementos a recorrer
- Subtarea del recorrido

Con estas tres cosas, plantear el código del recorrido se vuelve trivial. Veremos primero de forma genérica las partes y luego veremos ejemplos concretos.

## Esquema de recorrido

El esquema es la forma que va a tener el código, y determina cuántas subtareas se necesitan, dónde voy a ubicar cada subtarea, y de qué forma estas interactúan con el recorrido en general. El esquema dependerá de lo que se quiere realizar.

Un esquema de recorrido genérico sería:

```
Iniciar recorrido
while (hay que seguir recorriendo) {
    Procesar elemento actual
    Avanzar al siguiente elemento
}
Finalizar recorrido
```

Este esquema genérico implica contar con 5 subtareas, y colocarlas en determinados lugares del código (ver ejemplos para más información).

En la materia vemos 5 esquemas de recorridos en la primer parte de la materia:

- de transformación
- de búsqueda (sabiendo que el elemento buscado existe)
- de búsqueda (sin saber si el elemento buscado existe)
- de acumulación
- de máximo-mínimo

Muchos problemas pueden resolverse aplicando un esquema determinado, aunque no siempre el esquema calza tal cual, a veces se deben realizar ligeras modificaciones al mismo. Estas modificaciones deben ser realizadas a conciencia según el problema concreto a resolver.

Más aún, hay recorridos que no calzan en ninguno de los esquemas vistos propiamente en la materia.

### Esquema de transformación

Los recorridos de transformación garantizan que se realiza una tarea (que suele implicar un cambio de estado) para cada elemento a recorrer. El esquema quedaría:

```
Ir al primer elemento
while (quedan elementos por recorrer) {
    Transformar con/sobre el elemento actual
    Avanzar al siguiente elemento
}
Transformar con/sobre el elemento actual
```

Generalmente (pero no siempre) los elementos a procesar son filas, columnas o celdas y procesar cada elemento consiste en realizar una transformación sobre el mismo, es decir, modificando el tablero. La transformación implica así una subtarea “Transformar con/sobre el elemento actual”, que es la encargada de realizar la transformación sobre el elemento. Pero también pueden ser que los elementos sean valores como

colores o direcciones, en cuyo caso la transformación implica usar ese valor para hacer un transformación en el tablero.

La transformación puede implicar cambiar un elemento de una forma bajo ciertas condiciones, y dejarlo igual bajo otras. Es decir, la subtarea "Transformar con/sobre el elemento actual" puede implicar internamente una alternativa condicional. En esos casos, hay que tener cuidado de no confundir la condición de la transformación con la condición de corte del recorrido.

Problemas que suelen implicar una transformación incluyen enunciados con la forma:

- **Hacer X con cada elemento**

Donde "Hacer" puede ser una frase que comienza con verbos cualquiera que impliquen un cambio de estado, como "Poner una bolita roja", "Sacar todas las bolitas", etc. Ejemplos concretos podrían ser:

- Poner una bolita de color Rojo en cada celda de la fila actual.
- Poner una bolita Azul en cada celda del tablero que tenga al menos dos Verdes.
- Poner 10 bolitas de cada color.
- Poner una bolita de color Negro en cada celda lindante de la actual.
- Recolectar una banana en cada árbol del tablero en el que no hayan tarántulas.
- Destruir cada base enemiga en el mapa.

## Esquema de búsqueda (sabiendo que el elemento buscado existe)

Los recorridos de búsqueda permiten terminar el recorrido cuando se encuentra el elemento buscado. Para cada elemento, solo basta preguntar si es el elemento buscado o no, algo que se realiza en la condición del while.

```
Ir al primer elemento
while (not el elemento actual es el buscado) {
    Avanzar al siguiente elemento
}
Finalizar recorrido
```

La búsqueda implica la subtarea "el elemento actual es el buscado" que consiste en una función que describe un booleano, verdadero en caso de que el elemento actual sea el que se busca, falso en caso contrario.

Problemas que suelen implicar una búsqueda sabiendo que existe el elemento incluyen enunciados con formas del estilo:

- **Posicionarse en el elemento que cumple X**
- **Encontrar el elemento para el cual se cumple Y.**

Ejemplos concretos podrían ser:

- Posicionar el cabezal en la celda del tablero que tenga exactamente 3 bolitas de color Azul.
- Encontrar el color del cual hay bolitas en la celda actual.
- Encontrar la dirección a la cual hay menor distancia hacia el borde.
- Posicionar el cabezal sobre el fantasma.
- Encontrar la dirección hacia la cual no hay obstáculos.

**En este esquema se puede suponer que hay un elemento que cumple la condición**, de lo contrario no se podría cumplir la tarea.

## Esquema de búsqueda (sin saber si el elemento buscado existe)

En este caso, no se sabe si el elemento existe o no, por lo tanto, el recorrido como esquema no hace suposiciones.

```
Ir al primer elemento
while (quedan elementos por recorrer && not el elemento actual es el buscado) {
    Avanzar al siguiente elemento
}
Finalizar recorrido
```

Problemas que suelen implicar una búsqueda sin saber si existe el elemento incluyen enunciados con la forma:

- **Indicar si hay un elemento que cumpla X.**
- **Posicionarse en un elemento que cumpla X o en un lugar por defecto si no existe.**

Ejemplos concretos podrían ser:

- Indicar si hay una celda que tenga más de 20 bolitas.
- Indicar si hay una dirección hacia la cual el personaje no se puede mover.
- Indicar si hay un enemigo de algún color en la celda.
- Ubicar el cabezal en la primera celda vacía del tablero o en la esquina Norte-Este si no hay ninguna.

Notar que el orden de las condiciones en el while, en ocasiones, puede ser importante, ya que se hace uso de circuito corto. Se recomienda siempre tener primero la condición que implica mirar si quedan elementos por recorrer.

## Esquema de acumulación

El esquema de acumulación se utiliza cuando se desea calcular un valor total (un número) a partir de cada uno de los elementos que se recorren. Este tipo de recorrido se apoya en una variable que actúa de acumulador.

```
acumulador := valor inicial
Ir al primer elemento
while (quedan elementos por recorrer) {
    acumulador := acumulador ⊕ valor del elemento actual
    Avanzar al siguiente elemento
}
return ( acumulador ⊕ valor del elemento actual )
```

Es importante tener en cuenta que, en general, estos recorridos los vamos a realizar en funciones. No es que no pueda usarse un acumulador en un procedimiento, es viable, y hay escenarios en donde puede ser necesario. Sin embargo, lo más común es que hagamos una función independiente, y sí no lo hacemos, probablemente estemos incurriendo en una mala separación en subtareas.

Problemas que suelen implicar una acumulación incluyen cálculos de totales, y se expresan con la forma

- **Calcular el total de X para todos los elementos**

Ejemplos concretos podrían ser:

- Calcular el total de bolitas del tablero
- Calcular el promedio de bolitas rojas en cada celda del tablero
- Contar el total de puntos que pueden juntarse en el nivel.
- Calcular la cantidad de celdas vecinas que tienen una célula viva.
- Calcular la multiplicatoria de todas las bolitas azules de cada celda del tablero.

Notar que la acumulación implica determinar 3 cosas:

- La operación que se desea realizar (expresada con el símbolo ⊕ en el código), la cual, en la mayoría de los casos, suele ser una suma, pero hay escenarios en donde puede ser una multiplicación u otra operación.
- El "valor inicial", este suele ser un número, que actúa de valor neutro de la operación, por ej. en el caso de la suma, cero, en el caso de multiplicación, uno.
- El "valor del elemento actual", que implica determinar un número para el elemento actual. A veces este número es constante, a veces es una subtaréa compleja, que describe un número, y que puede implicar el uso de alternativa condicional en su funcionamiento, devolviendo el neutro de la operación en algunos casos.

Una subtarea muy útil en este tipo de recorridos suele ser ***unoSi_CeroSino***, que puede ser utilizada, por ejemplo, para contar la cantidad de elementos que cumplen cierta condición. El esquema en este caso pasaría a ser el siguiente:

```
acumulador := 0
Ir al primer elemento
while (quedan elementos por recorrer) {
  acumulador := acumulador + unoSi_CeroSino(condición sobre el elemento actual)
  Avanzar al siguiente elemento
}
return ( acumulador + unoSi_CeroSino(condición sobre el elemento actual) )
```

Si se acomodan las subtareas en el esquema, se puede obtener el mismo efecto, considerando el caso de borde al comienzo, en lugar de al final, lo cual suele dar como resultado una estructura más compacta y fácil de leer.

Para ello hay que tener en consideración primero ir siempre al elemento, antes de inicializar la variable, así como pasar al siguiente, antes de acumular dentro del while.

```
Ir al primer elemento
acumulador := valor del elemento actual
while (quedan elementos por recorrer) {
    Avanzar al siguiente elemento
    acumulador := acumulador ⊕ valor del elemento actual
}
return ( acumulador )
```

## Esquema de máximo-mínimo

El esquema de máximo-mínimo sirve para encontrar el elemento más grande o más chico entre todos. También hace uso de una variable como apoyo, pero notar que la forma en la que se actualiza, es distinta, ya que se pisa el valor completamente, en lugar de ir acumulando.

```
Ir al primer elemento
elMejorPorAhora := elemento actual
while (quedan elementos por recorrer) {
    Avanzar al siguiente elemento
  elMejorPorAhora := el mejor entre el elemento actual y elMejorPorAhora
}
return(elMejorPorAhora)
```

Notar que la estructura es similar a la acumulación que planteamos en el segundo caso, con la variable comenzando con el primer elemento como el valor más grande o más chico encontrado, al que llamamos “elMejorPorAhora”. Por esto, y por propiamente el problema de determinar un elemento más grande o más chico, es que debe haber al menos un elemento para procesar.

Problemas que suelen implicar un máximo o un mínimo incluyen enunciados con la forma:

- **Determinar el elemento más grande/chico bajo el criterio X**

Ejemplos concretos podrían ser:

- Determinar la cantidad más grande de bolitas en una misma celda del tablero.
- Determinar el color del cual hay más bolitas en el tablero.
- Determinar la dirección hacia la cual hay menor distancia al borde desde la celda actual.
- Determinar el insecto que más afecta a la plantación.
- Determinar la celda que menos puntos otorga.

Notar que muchas veces, estos enunciados implican además hacer algo con el elemento, lo cual hace que se confundan con búsquedas (por ej. posicionarse en la celda que tiene más bolitas) o con acumulaciones (calcular el color del cual hay más bolitas)

La subtarea a plantear es “el mejors entre el elemento actual y “elMejorPorAhora”, una subtarea que espera como argumento el elemento actual y el elemento que hasta el momento se determinó como el más grande o más chico, y que describe uno de los dos, según un criterio determinado. Esto implica el uso de alternativa condicional de expresiones en dicha subtarea. En los casos de números, en algunas ocasiones se puede hacer uso de las funciones de biblioteca **elMayorEntre_Y_** o **elMenorEntre_Y_**.

## Elementos a recorrer

Determinar apropiadamente los elementos que se están recorriendo es fundamental para poder completar adecuadamente el esquema de recorrido, ya que es el que permite completar 3 de las subtareas necesarias para el recorrido. En particular, poder "Ir al primer elemento", saber si "quedan elementos para recorrer" y poder "Avanzar al siguiente elemento" implica tener en claro qué se está recorriendo.

Entonces sí planteamos el esquema genérico, vemos que hay 3 subtareas claves a determinar:

```
Ir al primer elemento
while (quedan elementos para recorrer) {
    Procesar elemento actual
    Avanzar al siguiente elemento
}
Procesar elemento actual
```

Dependiendo de que se esté recorriendo, esas subtareas tomarán formas concretas. Algunos elementos comunes que solemos recorrer incluyen:

- Celdas de la fila/columna
- Filas/Columnas del tablero
- Celdas del tablero
- Direcciones
- Colores

Aunque se pueden recorrer muchas otras cosas. A continuación veremos la forma que toma cada una de esas subtareas dependiendo de los elementos que se recorran:

### Sobre celdas de la fila/columna

En este caso, basta con primitivas del lenguaje, por lo que es uno de los recorridos más simples.

**Ir al primer elemento:**

```
IrAlBorde(...)
```

Donde la dirección puede ser Sur/Norte si recorremos celdas de una columna, o Este/Oeste si son celdas de una columna.

**quedan elementos para recorrer:**

```
puedeMover(...)
```

Donde la dirección debe ser la opuesta a la usada en ir al primer elemento.

**Ir al siguiente elemento:**

```
Mover(...)
```

Donde la dirección debe ser la opuesta a la usada en ir al primer elemento.

### Sobre filas/columnas del tablero

En este caso son las mismas primitivas que antes, pero, en este caso, se debe pensar en el elemento a procesar como la totalidad de la fila/columna. En general, ese procesado suele implicar a su vez, un recorrido sobre las celdas de la fila/columna.

Un problema común es plantear un recorrido sobre filas/columnas con una subtarea que procesa celdas de la fila/columna en lugar de plantear un recorrido sobre celdas del tablero.

**Ir al primer elemento:**

```
IrAlBorde(...)
```

Donde la dirección puede ser Sur/Norte si recorremos filas, o Este/Oeste si son columnas.

**quedan elementos para recorrer:**

```
puedeMover(...)
```

**Ir al siguiente elemento:**

```
Mover(...)
```

Donde la dirección para estas dos últimas debe ser la opuesta a la usada en ir al primer elemento.

## Sobre celdas del tablero

En este caso utilizamos las subtareas de biblioteca planteadas:

**Ir al primer elemento:**

```
IrAlPrimeraCeldaEnUnRecorridoAl_YAl_(...)
```

**quedan elementos para recorrer:**

```
haySiguienteCeldaEnUnRecorridoAl_YAl_(...)
```

**Ir al siguiente elemento:**

```
IrAlSiguienteCeldaEnUnRecorridoAl_YAl_(...)
```

Donde las direcciónes pueden ser cualquiera siempre y cuando no sean opuestas ni iguales, y siempre se usan las mismas direcciones en todas las subtareas.

## Sobre direcciones

En este caso utilizamos una variable que contiene el elemento actual. La variable va a ir cambiando al elemento siguiente en cada repetición del while.

**Ir al primer elemento:**

```
direcciónActual := minDir()
```

**quedan elementos para recorrer:**

```
direcciónActual /= maxDir()
```

**Ir al siguiente elemento:**

```
direcciónActual := siguiente(direcciónActual)
```

### Sobre colores

Similar al anterior. La gracia de estos recorridos consiste en usar una variable, ya que lo que se recorre no implica un cambio de estado, sino que son valores, entidades abstractas, ergo, la única forma de recorrer es "recordando el valor actual".

**Ir al primer elemento:**

```
colorActual := minColor()
```

**quedan elementos para recorrer:**

```
colorActual /= maxColor()
```

**Ir al siguiente elemento:**

```
colorActual := siguiente(colorActual)
```

## Subtarea del recorrido

Como podemos apreciar, con los pasos anteriores vamos a tener la forma del código, y varias de las subtareas del recorrido (las que permiten moverse de un elemento al siguiente), pero el último paso es el que cambia según el problema concreto, ya que tiene que ver con cómo procesamos un elemento en particular.

Un recorrido consiste en hacer algo muchas veces. La subtarea es ese algo que se hace muchas veces (y al recorrer una colección, ese algo que se hace con cada elemento de la colección a recorrer). En un recorrido de transformación la subtarea nos dice qué transformación se aplica sobre cada elemento. En uno de búsqueda, qué pregunta le hacemos a cada elemento para saber si es el que buscamos. En uno de acumulación, cuánto acumulamos por cada elemento. En un de mínimo/máximo, qué maximizamos (o qué propiedad de cada elemento debemos comparar con el que es el mejor hasta ahora para determinar cuál es el mejor entre ellos).

Notar que la subtarea del recorrido **NO ES LO MISMO** que el objetivo del recorrido. En general el objetivo es lo que se obtiene al procesar TODA la colección, mientras que la subtarea es el procesamiento sobre UN elemento de la colección. En una transformación sobre el tablero el objetivo es aplicar la transformación sobre TODO el tablero pero la subtarea es aplicarla sobre UNA celda. En un búsqueda el objetivo es saber si alguno de TODOS los elementos cumple una propiedad (o encontrar al que la cumple) pero la subtarea es saber si UN elemento la cumple. En una acumulación el objetivo es obtener el total de lo que aportan TODOS los elementos pero la subtarea es obtener lo que aporta UN elemento. En una maximización el objetivo es obtener el mejor entre TODOS los elementos pero la subtarea es obtener el mejor entre UN elemento y el que es el mejor hasta el momento. Cuando escribimos la observación de un recorrido, lo que nos importa es la subtarea y NO el objetivo del recorrido, ya que es la subtarea la que nos dice cómo completar la parte del código que falta tras haber armado el esquema a partir de la clase de recorrido y de la colección a recorrer.

En el esquema genérico, esto implicaría pensar las subtareas "Procesar elemento actual", así como "Finalizar recorrido, en términos del objetivo concreto a lograr. Pero podemos pensar también cómo se modifican algunos esquemas concretos.

Veamos el ejemplo de lo que cambia en una transformación. En este caso, finalizar el recorrido suele implicar únicamente el caso de borde (aunque no siempre es así), y la subtarea que debe modificarse en este caso es "Transformar con/sobre el elemento actual". Esta subtarea es un procedimiento que modifica el tablero de alguna forma.

```
Ir al primer elemento
while (quedan elementos para recorrer) {
    Transformar con/sobre el elemento actual
    Avanzar al siguiente elemento
}
Transformar con/sobre el elemento actual
```

Esta subtarea va a cambiar en función de la necesidad del problema, por ejemplo, puede implicar "poner una bolita  roja" (para un recorrido de transformación sobre celdas de la fila) o "sacar todas las bolitas de la celda actual" (en estos de un recorrido de transformación sobre celdas del tablero).

En las acumulaciones, en cambio, la subtarea concreta a pensar es aquella que utilizamos para acumular, es decir "valor del elemento actual". Esta será una subtarea que utiliza el elemento actual para obtener un número. En este caso podemos por ejemplo, pensar en que la subtarea debe dar "el número de bolitas rojas en la celda actual" (en el caso de una acumulación sobre celdas del tablero) o podrían ser cosas más complejas, como "la cantidad de bolitas rojas que hay desde la celda actual hasta el borde en la dirección actual" (para un recorrido de acumulación sobre direcciones).

```
Ir al primer elemento
acumulador := valor del elemento actual
while (quedan elementos por recorrer) {
    Avanzar al siguiente elemento
    acumulador := acumulador ⊕ valor del elemento actual
}
return ( acumulador )
```

En el esquema de máximo o mínimo, la subtarea a pensar es aquella que implica determinar con cuál de los dos elementos quedarse al momento de compararlos. Esta subtarea suele luego reducirse internamente a un choose, que, dados dos elementos, describe uno de ellos (el mejor para el criterio elegido). Ya mencionamos los casos de las funciones de biblioteca **`maximoEntre_Y_`** y **`mínimoEntre_Y_`**.

```
Ir al primer elemento
elMejorPorAhora := elemento actual
while (quedan elementos por recorrer) {
    Avanzar al siguiente elemento
  elMejorPorAhora := el mejor entre el elemento actual y elMejorPorAhora
}
return(elMejorPorAhora)
```

Finalmente, en esquemas que implican búsqueda, lo que se debe pensar es la subtarea que determina si el elemento actual es o no el elemento buscado. Es decir, una subtarea que usa el elemento actual para indicar un booleano.

```
Ir al primer elemento
while (not el elemento actual es el buscado) {
    Avanzar al siguiente elemento
}
Finalizar recorrido
```

## Ejemplos menos abstractos

A continuación veremos ejemplos de recorridos aplicando esquemas a los distintos elementos a recorrer. En todos los casos dejaremos la subtarea que corresponde al procesado como algo abstracto, que deberá pensarse en términos del problema:

### Recorrido de transformación sobre celdas de la fila

```
IrAlBorde(Oeste)
while (puedeMover(Este)) {
    ProcesarCeldaActual()
    Mover(Este)
}
ProcesarCeldaActual()
```

Recorrido de transformación sobre filas

```
IrAlBorde(Norte)
while (puedeMover(Sur)) {
    ProcesarFilaActual()
    Mover(Sur)
}
ProcesarCeldaActual()
```

Recorrido de transformación sobre celdas del tablero

```
IrAlInicioEnUnRecorridoAl_YAl_(Este, Norte)
while (haySiguienteCeldaEnUnRecorridoAl_YAl_(Este, Norte)) {
    ProcesarCeldaActual()
    IrASiguienteCeldaEnUnRecorridoAl_YAl_(Este, Norte)
}
ProcesarCeldaActual()
```

Recorrido de transformación sobre direcciones

```
direcciónActual := minDir()
while (direcciónActual /= maxDir()) {
    ProcesarParaDirección_(direcciónActual)
    direcciónActual := siguiente(direcciónActual)
}
ProcesarParaDirección_(direcciónActual)
```

Recorrido de transformación sobre colores

```
colorActual := minColor()
while (colorActual /= maxColor()) {
    ProcesarParaColor_(colorActual)
    colorActual := siguiente(colorActual)
}
ProcesarParaColor_(colorActual)
```

Recorrido de búsqueda (sabiendo que está el elemento) sobre celdas de la fila

```
IrAlBorde(Oeste)
while (not esLaCeldaBuscada()) {
    Mover(Este)
}
```

Recorrido de búsqueda (sabiendo que está el elemento) sobre filas

```
IrAlBorde(Norte)
while (not esLaFilaBuscada()) {
    Mover(Sur)
}
```

Recorrido de búsqueda (sabiendo que está el elemento) sobre celdas del tablero

```
IrAlInicioEnUnRecorridoAl_YAl_(Este, Norte)
while (not esLaCeldaBuscada()) {
    IrASiguienteCeldaEnUnRecorridoAl_YAl_(Este, Norte)
}
```

Recorrido de búsqueda (sabiendo que está el elemento) sobre direcciones

```
direcciónActual := minDir()
while (not esLaDirección_LaBuscada(direcciónActual)) {
    direcciónActual := siguiente(direcciónActual)
}
return ( direcciónActual )
```

Recorrido de búsqueda (sabiendo que está el elemento) sobre colores

```
colorActual := minColor()
while (not esElColor_LaBuscada(direcciónActual)) {
    colorActual := siguiente(colorActual)
}
return ( colorActual )
```

Recorrido de búsqueda (sin saber si está el elemento) sobre celdas de la fila

```
IrAlBorde(Oeste)
while (puedeMover(Este) && not esLaCeldaBuscada()) {
    Mover(Este)
}
```

Recorrido de búsqueda (sin saber si está el elemento) sobre filas

```
IrAlBorde(Norte)
while (puedeMover(Sur) && not esLaFilaBuscada()) {
    Mover(Sur)
}
```

Recorrido de búsqueda (sin saber si está el elemento) sobre celdas del tablero

```
IrAlInicioEnUnRecorridoAl_YAl_(Este, Norte)
while (haySiguienteCeldaEnUnRecorridoAl_YAl_(Este, Norte)
            && not esLaCeldaBuscada()) {
    IrASiguienteCeldaEnUnRecorridoAl_YAl_(Este, Norte)
}
```

Recorrido de búsqueda (sin saber si está el elemento) sobre direcciones

```
direcciónActual := minDir()
while (direcciónActual /= maxDir()
            && not esLaDirección_LaBuscada(direcciónActual)) {
    direcciónActual := siguiente(direcciónActual)
}
return ( esLaDirección_LaBuscada(direcciónActual) )
```

Recorrido de búsqueda (sin saber si está el elemento) sobre colores

```
colorActual := minColor()
while (colorActual /= maxColor()
            && not esElColor_ElBuscado(colorActual)) {
    colorActual := siguiente(colorActual)
}
return ( esElColor_ElBuscado(colorActual) )
```

Recorrido de acumulación sobre celdas de la fila

```
IrAlBorde(Oeste)
acumulador := valorDeCeldaActual()
while (puedeMover(Este)) {
    Mover(Este)
    acumulador := acumulador ⊕ valorDeCeldaActual()
}
return ( acumulador )
```

Recorrido de acumulación sobre filas

```
IrAlBorde(Norte)
acumulador := valorDeFilaActual()
while (puedeMover(Sur)) {
    Mover(Sur)
    acumulador := acumulador ⊕ valorDeFilaActual()
}
return ( acumulador )
```

Recorrido de acumulación sobre celdas del tablero

```
IrAlInicioEnUnRecorridoAl_YAl_(Este, Norte)
acumulador := valorDeCeldaActual()
while (haySiguienteCeldaEnUnRecorridoAl_YAl_(Este, Norte)) {
    IrASiguienteCeldaEnUnRecorridoAl_YAl_(Este, Norte)
    acumulador := acumulador ⊕ valorDeCeldaActual()
}
return ( acumulador )
```

Recorrido de acumulación sobre direcciones

```
direcciónActual := minDir()
acumulador := valorDeDirección_(direcciónActual)
while (direcciónActual /= maxDir()) {
    direcciónActual := siguiente(direcciónActual)
    acumulador := acumulador ⊕ valorDeDirección_(direcciónActual)
}
return ( acumulador )
```

Recorrido de acumulación sobre colores

```
colorActual := minColor()
acumulador := valorDeColor_(colorActual)
while (colorActual /= maxColor()) {
    colorActual := siguiente(colorActual)
    acumulador := acumulador ⊕ valorDeColor_(colorActual)
}
return ( acumulador )
```

Recorrido de máximo-mínimo sobre celdas de la fila

```
IrAlBorde(Oeste)
```

```
elMejorPorAhora := valorDeCeldaActual()
while (puedeMover(Este)) {
    Mover(Este)
    elMejorPorAhora := elMejorEntre_Y_(valorDeCeldaActual(), elMejorPorAhora)
}
return ( elMejorPorAhora )
```

### Recorrido de máximo-mínimo sobre filas

```
IrAlBorde(Norte)
elMejorPorAhora := valorDeFilaActual()
while (puedeMover(Sur)) {
    Mover(Sur)
    elMejorPorAhora := elMejorEntre_Y_(valorDeFilaActual(), elMejorPorAhora)
}
return ( elMejorPorAhora )
```

### Recorrido de máximo-mínimo sobre celdas del tablero

```
IrAlInicioEnUnRecorridoAl_YAl_(Este, Norte)
elMejorPorAhora := valorDeCeldaActual()
while (haySiguienteCeldaEnUnRecorridoAl_YAl_(Este, Norte)) {
    IrASiguienteCeldaEnUnRecorridoAl_YAl_(Este, Norte)
    elMejorPorAhora := elMejorEntre_Y_(valorDeCeldaActual(), elMejorPorAhora)
}
return ( elMejorPorAhora )
```

### Recorrido de máximo-mínimo sobre direcciones

```
direcciónActual := minDir()
elMejorPorAhora := direcciónActual
while (direcciónActual /= maxDir()) {
    direcciónActual := siguiente(direcciónActual)
    elMejorPorAhora := elMejorEntre_Y_(direcciónActual, elMejorPorAhora)
}
return ( elMejorPorAhora )
```

### Recorrido de máximo-mínimo sobre colores

```
colorActual := minColor()
elMejorPorAhora := colorActual
while (colorActual /= maxColor()) {
    colorActual := siguiente(colorActual)
    elMejorPorAhora := elMejorEntre_Y_(colorActual, elMejorPorAhora)
}
return ( elMejorPorAhora )
```

## Conclusiones:

Vimos un montón de esquemas de recorridos distintos. Muchos problemas se pueden resolver aplicando directamente el esquema, casi que copiando y pegando el código. En todos los casos, debo primero identificar qué quiero un recorrido, luego identificar el esquema y los elementos sobre los que voy a recorrer. Incluso teniendo eso, siempre debo pensar la subtarea concreta, por lo que no es tan fácil como parece. **No olvidar renombrar las variables para que se ajusten al problema planteado** (en particular en los esquemas de acumulación y de mínimo/máximo).

Los esquemas son una guía muy útil, pero no son regla fija. Pueden haber problemas que no calcen en ninguno de los esquemas, y que combinen partes de diferentes esquemas, ej. búsqueda con procesamiento, o búsqueda con acumulación.

Los elementos a recorrer tampoco son exclusivamente los que mencionamos, puedo recorrer muchas otras cosas, como otros tipos de datos enumerativos, números (sí están acotados), celdas específicas (ej. celdas vacías), o cosas que están modeladas en celdas (ej. motoqueros).

En todos los casos, saber los esquemas y los elementos conocidos a recorrer ayuda a plantear los recorridos necesarios para solucionar otros problemas.

# Resumen de recorridos (segundo parcial)

Partimos del conocimiento de recorridos que tenemos hasta el primer parcial. Ya conocemos varios esquemas, y hemos recorrido diversos elementos.

Para el segundo parcial, se suma poder recorrer sobre elementos enumerativos (por ej. días de la semana, meses del año, etc.), lo cual, es idéntico a recorrer direcciones o colores, con la salvedad de que se debe primero realizar las subtareas que determinen cuál es el primer elemento, cuál es el último y como se obtiene el siguiente.

La parte más interesante tiene que ver con recorrer listas. Para estas, surge una nueva herramienta que nos permite recorrerlas cómodamente, **la repetición indexada**. Además las listas tienen algunas características que las hacen diferentes en términos de recorridos.

Por ejemplo, en las listas, no hay que procesar un elemento como caso borde ya que al recorrer con repetición indexada está garantizado haber recorrido todos los elementos, y que no falte ninguno. Esto también aplica a recorrer con repetición condicional, ya que contamos con la expresión "lista vacía" que nos permite reconocer cuando se nos acabaron los elementos sin incurrir en un error (como salirnos del tablero en el caso de las celdas o volver a considerar el primer elemento en el caso de los tipos enumerativos).

Por otro lado, al usar la repetición indexada para recorrer no es necesario plantear las subtareas correspondientes a "Ir al primer elemento", "quedan elementos por recorrer" y "Pasar al siguiente elemento" (aunque esto sí es necesario cuando usamos la repetición condicional, pero en principio se utilizan las primitivas de listas), lo cual simplificará los esquemas realizados.

En la materia vemos 6 esquemas de recorridos en la segunda parte de la materia:

- de transformación
- de filtro
- de acumulación
- de máximo-mínimo
- de búsqueda (sabiendo que el elemento buscado existe)
- de búsqueda (sin saber si el elemento buscado existe)

Los primeros cuatro, además, pueden considerarse recorridos de totalización, ya que en todos se está buscando un total (acumulando en una variable, que puede contener un número, un elemento o una lista).

A continuación mencionaremos los distintos esquemas. Para ello, los expresaremos como funciones, donde se recibirá por parámetro la lista cuyos elementos se deben recorrer bajo el nombre de "lista" (aunque no necesariamente es siempre una lista pasada por parámetro la que debe ser recorrida). También mencionaremos como identificar cada esquema y las subtareas a pensar.

## Recorrido de transformación

La transformación en una lista implica que el resultado a generar es una nueva lista, donde para cada elemento en la lista original, hay un elemento en la lista generada que le corresponde.

```
function lista_Transformada(lista) {
    listaResultadoAlMomento := []
    foreach elemento in lista {
                    listaResultadoAlMomento  :=  listaResultadoAlMomento  ++
[elemento_Transformado(elemento)]
    }
    return (listaResultadoAlMomento)
}
```

En una transformación, está garantizado por estructura del recorrido (ya que siempre hacemos ++ con una lista singular) que la longitud de la lista transformada, va a ser la misma que la de la lista original, es decir:

- **longitudDe_(lista_Transformada(lista)) == longitudDe_(lista)**

Si la lista original está vacía, la lista transformada será vacía, por lo que el recorrido no falla en caso de una lista vacía.

La lista a transformar puede ser una lista cualquiera, digamos que es de tipo “Lista de A”, siendo “A” un tipo cualquiera. La lista que se devuelve tendrá el tipo “Lista de B”, siendo “B” el tipo de la función `elemento_Transformado` (que podría ser el mismo que “A” en algunas ocasiones).

Así, este esquema requiere de una subtarea, que realice dicha transformación. La subtarea **elemento_Transformado** es la subtarea, tal que si le doy un elemento del tipo A, describe un elemento del tipo B (O al menos un valor diferente del tipo A), es decir, transformándolo. Tener en cuenta que, a pesar de llamarlo así, la lista original no es transformada sino que lo que se hace es crear una lista nueva con elementos nuevos.

Problemas que caen en la categoría de transformaciones suelen tener enunciados de la forma:

- **Dada una lista de A, describir una lista donde para cada A, se lo transformó en un B.**

Ejemplos concretos:

- Dada una lista de números, describir la lista donde a cada número se lo elevó al cuadrado.

```
function losNúmeros_ElevadosAlCuadrado(números) {
    // PARÁMETROS:
    // * números : Lista de Número
    // TIPO: Lista de Número
    cuadradosAlMomento := []
    foreach número in números {
        cuadradosAlMomento := cuadradosAlMomento ++
            [número ^ 2] // No necesito una subtarea, alcanza con ^
    }
    return (cuadradosAlMomento)
}
```

- Dada una lista de Personas, describir una lista con los nombres de las personas.

```
function nombreDePersonas_(personas) {
    // PARÁMETROS:
    // * personas : Lista de Persona
    // TIPO: Lista de String
    nombresAlMomento := []
    foreach persona in personas {
        nombresAlMomento := nombresAlMomento ++
            [nombre(persona)] // Alcanza a veces con la observadora

    }
    return (nombresAlMomento)
}
```

- Dada una lista de Deportes, describir la lista que contiene la cantidad de jugadores que compiten en cada deporte.

```
function jugadoresCompitiendoEnDeportes_(deportes) {
    // PARÁMETROS:
    // * deportes : Lista de Deporte
    // TIPO: Lista de Número
    competidoresPorDeporteAlMomento := []
    foreach deporte in deportes {
        competidoresPorDeporteAlMomento := competidoresPorDeporteAlMomento ++
            [cantidadCompitiendoEnDeporte_(deporte)]
            // A veces necesito una subtarea más compleja
    }
    return (competidoresPorDeporteAlMomento)
}
```

Puede darse el caso en donde la transformación es condicional, es decir, algunos elementos se transforman, mientras que otros permanecen igual. Esto solo puede ocurrir si A y B son el mismo tipo. Un ejemplo:

- Dada una lista de Autos, describir la lista de Autos donde se pintó de rojo a los autos azules.

```
function autos_ConAzulesPintadosARojo(autos) {
    // PARÁMETROS:
    // * autos : Lista de Auto
    // TIPO: Lista de Auto
    autosPintadosAlMomento := []
    foreach auto in autos {
        autosPintadosAlMomento := autosPintadosAlMomento ++
            [auto_PintadoARojoSiEsAzul(auto)]
            // A veces necesito una subtarea más compleja
    }
    return (autosPintadosAlMomento)
}

function auto_PintadoARojoSiEsAzul(auto) {
    // PARÁMETROS:
    // * auto : Auto
    // TIPO: Auto
    return (choose
        auto_PintadoDeRojo(auto) when (color(auto) == Azul)
        auto otherwise
    )
}

function auto_PintadoDeRojo(auto) {
    // PARÁMETROS:
    // * auto : Auto
    // TIPO: Auto
    return (Auto(auto | color <- Rojo))
}
```

## Recorrido de filtro

El filtrado en una lista implica que el resultado a generar es una nueva lista, donde van a estar algunos (pero potencialmente no todos) los elementos de la lista original, sin modificaciones. El esquema hace uso de **singular_Si_** como forma de agregar condicionalmente un elemento a la lista de resultados. Es decir, no siempre se agrega el elemento a la lista, depende de una condición.

```
function lista_Filtrada(lista) {
    listaResultadoAlMomento := []
    foreach elemento in lista {
        listaResultadoAlMomento := listaResultadoAlMomento ++
                        singular_Si_(elemento, debeAgregarse_(elemento))
    }
    return (listaResultadoAlMomento)
}
```

En un filtro, la longitud de la lista filtrada, va a ser la misma o menor que la de la lista original, es decir:

- **longitudDe_(lista_Filtrada(lista)) <= longitudDe_(lista)**

Es decir, nada impide que el resultado no sea vacío, incluso si la lista original no lo era. Sin embargo, si la lista original estaba vacía, también lo estará el resultado, pero el recorrido sigue funcionando bien.

La lista a filtrar puede ser una lista cualquiera, digamos que es de tipo “Lista de A”, siendo “A” un tipo cualquiera. La lista que se devuelve tendrá el tipo “Lista de A”, ya que los elementos no se transforman (más aún, los elementos en la lista resultante serán elementos que ya estaban en la lista original).

Así, este esquema requiere de una subtarea **debeAgregarse_** que dado un elemento del tipo A, describe un Booleano, que indica si el elemento debe ser agregado a la lista.

Problemas que caen en la categoría de filtros suelen tener enunciados de la forma:

- **Dada una lista de A, describir la lista de los A que cumplen X.**

Ejemplos concretos:

- Dada una lista de números, describir la lista de los números pares

```
function paresEn_(números) {
    // PARÁMETROS:
    // * números : Lista de Número
    // TIPO: Lista de Número
    paresAlMomento := []
    foreach número in números {
        paresAlMomento := paresAlMomento ++
                          singular_Si_(número, esPar_(número))
    }
    return (paresAlMomento)
}
```

- Dada una lista de Personas, describir las personas mayores de edad.

```
function mayoresDeEdadEn_(personas) {
    // PARÁMETROS:
    // * personas : Lista de Persona
    // TIPO: Lista de Persona
    mayoresDeEdadAlMomento := []
    foreach persona in personas {
        mayoresDeEdadAlMomento := mayoresDeEdadAlMomento ++
                           singular_Si_(persona, esMayorDeEdad_(persona))
    }
    return (mayoresDeEdadAlMomento)
}
```

- Dada una lista de Deportes, describir los deportes que sean acuáticos

```
function deportesDe_QueSonAcuáticos(deportes) {
    // PARÁMETROS:
    // * deportes : Lista de Deporte
    // TIPO: Lista de Deporte
    deportesAcuáticosAlMomento := []
    foreach deporte in deportes {
        deportesAcuáticosAlMomento := deportesAcuáticosAlMomento ++
                singular_Si_(deporte, esDeporte_DeTipo_(deporte, Acuático))
    }
    return (deportesAcuáticosAlMomento)
}
```

## Recorrido de acumulación

Hablamos de acumulación en una lista cuando el resultado implica calcular un número a partir de los diversos elementos de la lista.

```
function acumuladoEn_(lista) {
    acumuladoAlMomento := valorInicial
    foreach elemento in lista {
        acumuladoAlMomento := acumuladoAlMomento ⊕ valorDe_(elemento)
```

```
    }
    return (acumuladoAlMomento)
}
```

Nuevamente, al igual que en las acumulaciones de recorridos sobre tablero, la operación puede ser una suma, una multiplicación, u otra. Por lo que hay que determinar en este caso, dos cosas, la operación, y el **valor inicial**, que suele ser el neutro de la operación a realizar.

La lista sobre la cual acumular no tiene por que ser de números, sino que puede ser de cualquier tipo, es decir "Lista de A", siendo "A" un tipo cualquiera. Lo que se devuelve en este caso es un Número.

Este esquema requiere de una subtarea **valorDe_** que dado un elemento del tipo A, describe un Número, siendo dicho número el que vamos a acumular. A veces, la subtarea es trivial e innecesaria (por ejemplo, si ya estoy recorriendo números, tal vez consiste en usar directamente el elemento) o si estoy contando la cantidad de elementos, todo elemento vale uno. En otras ocasiones puede consistir en subtareas más complejas, incluso algunas que impliquen otros recorridos. En algunos escenarios, puede ser útil la subtarea **unoSi_CeroSino**.

Problemas que caen en la categoría de acumulaciones suelen tener enunciados de la forma:

- **Dada una lista de A, describir el total de X de los elementos de dicha lista.**

Ejemplos concretos:

- Dada una lista de números, describir la suma de los mismos

```
function sumatoriaDe_(números) {
    // PARÁMETROS:
    // * números : Lista de Número
    // TIPO: Número
    sumaAlMomento := 0
    foreach número in números {
        sumaAlMomento := sumaAlMomento + número
                        // Un caso donde se usa directo el elemento
    }
    return (sumaAlMomento)
}
```

- Dada una lista de cualquier cosa, determinar la longitud de la lista

```
function longitudDe_(lista) {
    // PARÁMETROS:
    // * lista : Lista de "Cualquiera"
    // TIPO: Número
    longitudAlMomento := 0
    foreach elemento in lista {
        longitudAlMomento := longitudAlMomento + 1
                        // Un caso donde se usa siempre uno
    }
    return (longitudAlMomento)
}
```

- Dada una lista de Personas, determinar la suma de edades de todas ellas

```
function edadTotalDe_(personas) {
    // PARÁMETROS:
    // * personas : Lista de Persona
    // TIPO: Número
    sumaDeEdadesAlMomento := 0
    foreach persona in personas {
        sumaDeEdadesAlMomento := sumaDeEdadesAlMomento + edad(persona)
    }
    return (sumaDeEdadesAlMomento)
}
```

## Recorrido de máximo-mínimo

Hablamos de máximo o mínimo en una lista cuando lo que se busca es un elemento de la lista que sea más grande o más pequeño que todo el resto, para algún criterio cualquiera de lo que significa ser grande o pequeño. Así, planteamos nuevamente la idea de "elMejorPorAhora" (el más grande, el más largo, el más chico, el más joven, etc.).

```
function elMejorEn_(lista) {
    elMejorPorAhora := primero(lista)
    foreach elemento in sinElPrimero(lista) {
        elMejorPorAhora := elMejorEntre_Y_(elemento, elMejorPorAhora)
    }
    return (elMejorPorAhora)
}
```

Notar que determinar un máximo o un mínimo implica que haya al menos un elemento en la lista (más aún, podría implicar que haya uno que sea más grande o más chico que todo el resto). Por lo tanto, este tipo de recorrido requiere asumir que la lista no está vacía.

La lista sobre la cual encontrar el máximo o mínimo puede ser de cualquier tipo, es decir "Lista de A", siendo "A" un tipo cualquiera. Lo que se devuelve en este caso es uno de los elementos, es decir, uno de esos "A".

Este esquema requiere de una subtarea **elMejorEntre_Y_** que dados dos elementos del tipo "A" (Lo que sea que haya en la lista), describe uno de ellos, aquel que cumple mejor el criterio de máximo o mínimo buscado.

Problemas que caen en la categoría de máximo o mínimo suelen tener enunciados de la forma:

- **Dada una lista de A, describir el elemento más X de la lista.**

Ejemplos concretos:

- Dada una lista de números, describir el número más grande

```
function elMásGrandeEn_(números) {
    // PARÁMETROS:
    // * números : Lista de Número
    // TIPO: Número
    elMásGrandeAlMomento := primero(números)
    foreach número in sinElPrimero(números) {
        elMásGrandeAlMomento := mayorEntre_Y_(número, elMásGrandeAlMomento)
                        // Un caso donde se usa la función de biblioteca
    }
    return (elMásGrandeAlMomento)
}
```

- Dada una lista de cualquier Personas, determinar la persona más joven

```
function másJovenEntre_(personas) {
    // PARÁMETROS:
    // * personas : Lista de Persona
    // TIPO: Persona
    másJovenAlMomento := primero(personas)
    foreach persona in sinElPrimero(personas) {
        másJovenAlMomento := elMásJovenEntre_Y_(persona, másJovenAlMomento)
    }
    return (másJovenAlMomento)
}
```

- Dada una lista de cualquier Guerreros, determinar el guerrero con más poder

```
function elMásPoderososEntre_(guerreros) {
    // PARÁMETROS:
    // * personas : Lista de Guerrero
    // TIPO: Guerrero
    másPoderosoAlMomento := primero(guerreros)
    foreach guerrero in sinElPrimero(guerreros) {
        másPoderosoAlMomento :=
                elMásPoderosoEntre_Y_(guerrero, másPoderosoAlMomento)
    }
    return (másPoderosoAlMomento)
}
```

## Recorrido de búsqueda (sabiendo que está el elemento)

Un recorrido de búsqueda (sabiendo que el elemento está) se da cuando queremos en general obtener un elemento de la lista que cumple algún criterio particular. Este esquema no utiliza foreach, sino while, por lo que usará una variable auxiliar para recorrer la lista.

```
function recuperarElementoEn_(lista) {
    pendientesPorVer := lista
    while (not esElBuscado_(primero(pendientesPorVer))) {
        pendientesPorVer := sinElPrimero(pendientesPorVer)
    }
    return (primero(pendientesPorVer))
}
```

En los recorridos de búsqueda, a diferencia de los de totalización, no se requiere de una variable para acumular resultado, ya que lo que se quiere determinar es si algo está o no en la lista, o recuperarlo sí es que está allí. Tampoco se van a recorrer necesariamente todos los elementos de la lista, ya que se termina ni bien se encontró lo que se buscaba.

En el caso de las búsquedas sabiendo que el elemento está, se suele querer encontrar u obtener cuál es el elemento que cumple una condición determinada, sabiendo a priori que hay uno que la cumple. Es decir, se requiere que haya algún elemento que cumpla la condición.

La lista sobre la cual buscar puede ser de cualquier tipo, es decir “Lista de A”, siendo “A” un tipo cualquiera. Lo que se devuelve, en general, para las búsquedas en las que se sabe que el elemento está, es uno de los elementos, es decir, uno de esos “A” (el que cumple la condición, o el primero de los que la cumplen).

Este esquema requiere de una subtarea **esElBuscado_** que dado un elemento del tipo “A” (Lo que sea que haya en la lista), describe un Booleano que indica si el elemento es o no el que se está buscando.

Problemas que caen en la categoría de búsqueda sabiendo que está suelen tener la forma de:

- **Dada una lista de A, describir el elemento que cumple X.**

Ejemplos concretos:

- Dada una lista de números, y sabiendo que hay en ella solo un número que es primo, describir el número primo de la lista.

```
function primoEn_(números) {
    // PARÁMETROS:
    // * números : Lista de Número
    // TIPO: Número
    pendientesPorVer := números
    while (not esPrimo_(primero(pendientesPorVer))) {
        pendientesPorVer := sinElPrimero(pendientesPorVer)
    }
    return (primero(pendientesPorVer))
}
```

- Dada una lista de cualquier Personas en donde no hay dos con el mismo DNI, describir la Persona con DNI 1234

```
function personaConDNI1234En_(personas) {
    // PARÁMETROS:
    // * personas : Lista de Persona
    // TIPO: Persona
    pendientesPorVer := personas
    while (dni(primero(pendientesPorVer)) /= “1234”) {
        pendientesPorVer := sinElPrimero(pendientesPorVer)
    }
    return (primero(pendientesPorVer))
}
```

- Dada una lista de cualquier Guerreros, describir el primer guerrero que use un hacha, sabiendo que hay al menos uno en la lista.

```
function elPrimeroQueUsaHachaEn_(guerreros) {
    // PARÁMETROS:
    // * personas : Lista de Persona
    // TIPO: Persona
    pendientesPorVer := guerreros
    while (not usaHacha_(primero(pendientesPorVer))) {
        pendientesPorVer := sinElPrimero(pendientesPorVer)
    }
    return (primero(pendientesPorVer))
}
```

## Recorrido de búsqueda (sin saber si está el elemento)

Un recorrido de búsqueda (sin saber si está el elemento) se da cuando queremos verificar si en la lista hay algún elemento que cumpla determinada condición (o que no la cumpla). A diferencia del anterior, funciona sin restricciones sobre la lista.

```
function existeElementoEn_(lista) {
    pendientesPorVer := lista
    while (not esVacía(pendientesPorVer)
              && not esElBuscado_(primero(pendientesPorVer))) {
        pendientesPorVer := sinElPrimero(pendientesPorVer)
    }
    return (not esVacía(pendientesPorVer))
}
```

Notar que la subtarea **esElBuscado_** funciona igual que para el recorrido anterior, por lo que se pueden recorrer el mismo estilo de listas, y de la misma forma, y la subtarea hasta puede ser la misma.

Lo que cambia suele ser el return, que implica ver si efectivamente encontramos el elemento buscado. Por lo que este tipo de recorridos describen un Booleano.

Problemas que caen en la categoría de búsqueda sin saber que el elemento está suelen tener la forma de:

- **Dada una lista de A, indicar sí hay un elemento que cumple X.**

Ejemplos concretos:

- Dada una lista de números, indica si hay un número primo en la lista.

```
function hayPrimoEn_(números) {
    // PARÁMETROS:
    // * números : Lista de Número
    // TIPO: Booleano
    pendientesPorVer := números
    while (not esVacía(pendientesPorVer)
              && not esPrimo_(primero(pendientesPorVer))) {
        pendientesPorVer := sinElPrimero(pendientesPorVer)
    }
    return (not esVacía(pendientesPorVer))
}
```

- Dada una lista de Personas, indicar si hay una con DNI 1234

```
function hayPersonaConDNI1234En_(personas) {
    // PARÁMETROS:
    // * personas : Lista de Persona
    // TIPO: Booleano
    pendientesPorVer := personas
    while (not esVacía(pendientesPorVer)
              && dni(primero(pendientesPorVer)) /= “1234”) {
        pendientesPorVer := sinElPrimero(pendientesPorVer)
    }
    return (not esVacía(pendientesPorVer))
}
```

- Dada una lista de cualquier Guerreros, indicar si hay uno que use un hacha.

```
function hayGuerreroQueUsaHachaEn_(guerreros) {
    // PARÁMETROS:
    // * personas : Lista de Guerrero
    // TIPO: Booleano
    pendientesPorVer := guerreros
    while (not esVacía(pendientesPorVer)
              && not usaHacha_(primero(pendientesPorVer))) {
        pendientesPorVer := sinElPrimero(pendientesPorVer)
    }
    return (not esVacía(pendientesPorVer))
}
```

## Conclusiones:

Al igual que con los recorridos sobre tablero, los esquemas son una guía útil, pero no son regla fija. Hay problemas que no se pueden resolver aplicando un esquema de los mencionados, y requieren pensar un poco más, usando partes de diferentes esquemas (por ej. una búsqueda con una acumulación, o una transformación con filtro). Adicionalmente, puede que haya problemas que calcen casi perfecto en el esquema, pero que requieran leves modificaciones, por ej. una búsqueda cuando lo que queremos es saber que el elemento NO está en la lista.

Por otro lado, vemos un criterio claro sobre cuándo usar un tipo de repetición u otra. Usaremos repetición indexada cuando tengamos que recorrer la totalidad de la lista, un elemento a la vez, y estemos recorriendo una única lista. Usaremos repetición condicional en otros escenarios.