TEMA 4: DISEÑO DE ALGORITMOS RECURSIVOS

1. Principio de inducción
2. Diseño recursivo
3. Coste temporal de un algoritmo recursivo
4. Coste espacial de un algoritmo recursivo
5. Inmersión de programas
6. Inmersión de especificaciones

2. Diseño recursivo

Def.: Un algoritmo se dice que es recursivo cuando contiene en su definición una o más llamadas a si mismo.

Todo algoritmo recursivo tendrá al menos una instrucción alternativa que contemplará dos casos bien diferenciados:

- **Caso directo (o caso base):** Es el caso en el cual el problema tiene una solución directa. Es el último caso que se ejecuta en la recursión.
- **Caso recursivo:** Es donde se relaciona el resultado del algoritmo con resultados de casos más simples. Estos casos más simples corresponderán con las llamadas recursivas.

La condición de la instrucción alternativa que selecciona el caso directo se denomina **condición de parada**.

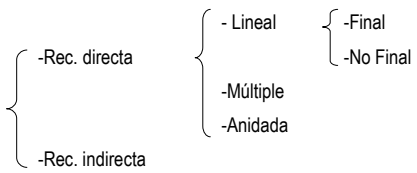
2. Diseño recursivo

Ejemplo: Función factorial

```
factorial(n: natural) dev f: natural
si n = 0 entonces                    // Condición de parada
   f <- 1                            // Caso base
sino
   f <- n * factorial(n - 1)         // Caso recursivo
fsi
```


2. Diseño recursivo

- **Tipos de algoritmos recursivos:**



2. Diseño recursivo

- **Recursión directa:** Cuando un algoritmo se llama a si mismo en su definición.
- **Recursión indirecta:** Cuando el algoritmo no tiene una llamada a si mismo, sino que llama a otro algoritmo que le llama a él.

Ejemplo: Funciones Par/Impar

```
Par(n:Nat) dev: p:bool
si n = 0
        p <- cierto
sino si n = 1
        p <- falso
sino
        p <- Impar(n-1)
Fsi
```

```
Impar(n:Nat) dev: p:bool
si n = 0
        p <- falso
sino si n = 1
        p <- cierto
sino
        p <- Par(n-1)
Fsi
```


2. Diseño recursivo

- **Recursión lineal:** Cuando sólo hay una llamada recursiva en el algoritmo. **Ejemplo:** Factorial
- **Recursión final:** Es una recursión lineal donde lo último que se ejecuta es la llamada recursiva.

Ejemplo:

```
Resto(n,m: N) dev r: N
si n < m entonces
   r <- n
sino
   r <- Resto(n - m, m)
fsi
```

2. Diseño recursivo

- **Recursión multiple:** Cuando en la definición aparece más de una llamada recursiva.

```
Fib(n: N) dev f: N
si n <= 1 entonces
  f <- n
sino
  f <- Fib(n - 1) + Fib(n - 2)
fsi
```


2. Diseño recursivo

- **Recursión anidada:** Cuando alguno de los argumentos de la llamada es a su vez una llamada recursiva.

```
Ack(n,m: N) dev ack: N
si n = 0 entonces
  ack <- m + 1
sino
  si m = 0 entonces
   ack <- Ack(n - 1, 1)
  sino
   ack <- Ack(n - 1, Ack(n, m - 1))
  fsi
fsi
```


2. Diseño recursivo

- **Esquema general de una función recursiva lineal:**

```
F(x: T1) dev r: T2
si d(x) entonces
  r <- h(x)
sino
  v <- F(s(x))
  r <- c(x, v)
fsi
```

Cálculo de los parámetros de entrada

Llamada recursiva

Cálculo del resultado

Donde x:T1 y r:T2 son listas de parámetros y no un único parámetro.

  - El esquema general de una función **recursiva final** será igual que el lineal pero sin la función 'c'.

2.1. Verificación de funciones recursivas

- Toda función recursiva contiene al menos una instrucción alternativa, por lo que la verificación consistirá en **verificar una instrucción alternativa.**
- En una de las ramas de la instrucción alternativa vamos a tener una llamada a la propia función, pero para verificar una llamada a una función, la propia función ha de estar ya verificada.
- La solución es aplicar el principio de inducción: Tomamos como **Hipótesis de Inducción** que la función ya está verificada para los casos anteriores (para las llamadas recursivas).
- Cuando tengamos un único parámetro de entrada que sea natural, podremos aplicar inducción débil o fuerte.
  - **Ejemplo:** Función factorial.

2.1. Verificación de funciones recursivas

- Cuando no tengamos un parámetro de entrada natural, tendremos que aplicar **inducción noetheriana**.
- Se deberá definir un preorden bien fundado sobre los parámetros de entrada. Para ello utilizaremos una función definida sobre el dominio de los parámetros de entrada que devuelva un natural. Esta función la denominaremos **función de cota** (t: $\mathcal{D} \rightarrow \mathcal{N}$).
- La función de cota deberá ser **estrictamente decreciente** en cada llamada recursiva para poder aplicar el principio de inducción.

  t(s(x)) < t(x)
- Con la función de cota también estamos demostrando que la función recursiva tiene final.

2.1. Verificación de funciones recursivas

- Ejemplos de funciones de cota:
  - Dos números (m, n)
    - Uno siempre decrece: Define la función de cota
    - m + n
    - min(m, n) o max(m, n)
    - Orden lexicográfico
  - Tipos abstractos de datos
    - Pila: altura
    - Cola: longitud
    - Árbol: tamaño o altura
    - Lista_PI: pend (elementos pendientes)

2.1. Verificación de funciones recursivas

- Ejemplos de verificación de funciones recursivas.
  - Verificar función suma de una cola.

```
1) suma(c_nula) ≡ 0
2) suma(pide_turno(x, c)) ≡ x + suma(c)

Func suma_c(c:Cola) dev s:Nat
{Pre: cierto, Dec: ??}
   [nula(c) → s := 0
   -nula(c) → s := suma_c(avance(c))
               s := s + primero(c)
   ]
{Post: s = suma(c)}
```

  - Verificar función recursiva lineal genérica.

2.1. Verificación de funciones recursivas

- Ejemplos de verificación de funciones recursivas.
  - Verificar función de Fibonaci.

```
1) fibo(0) ≡ 0
2) fibo(1) ≡ 1
3) fibo(n+2) ≡ fibo(n+1) + fibo(n)

Func fib(n: Nat) dev f:Nat
{Pre: cierto; Dec: ??}
   [n = 0 → f := 0
    n = 1 → f := 1
    n > 1 → f1 := fib(n-1);
            f2 := fib(n-2);
            f := f1 + f2
   ]
{Post: f = fibo(n)}
```


2.1. Verificación de funciones recursivas

- **Verificación de programas en C++.** Dos posibilidades:
  - Reescribirlo en el lenguaje while.
  - Verificar directamente en C++ siempre que tengamos clara la semántica de las instrucciones implicadas.

Ejemplo: Función para multiplicar dos números. Verifícala.

```
int mult(int a, int b)
{  int p;
   if (a == 0)
      p = 0;
   else
      p = mult(a-1, b) + b;
   return p;
}
```

¿Precondición? ¿Postcondición?¿Función de cota?

2.2. Finalización de funciones recursivas

- En ocasiones sólo interesa saber si una función recursiva acabará. Para ello basta con realizar los siguientes pasos:
  - **Verificar que las llamadas recursivas se pueden realizar:** que se cumple la precondición y se pueden evaluar los parámetros.
  - **Buscar una función de cota y comprobar que es correcta:** que pertenece a los naturales y que es estrictamente decreciente.

2.2. Finalización de funciones recursivas

- **Ejercicio:** Comprobar para que valores de los parámetros la siguiente función recursiva acaba.

```
int f(int a, int b)
{  int r;

   if(a+b) == 0
      r = 0;
   else
   {
      r = f(2*a, b-a-1);
      r = r + 1;
   }
}
```

¿Precondición? ¿Función de cota?

2.3. Derivación de funciones recursivas

- **Lo primero es tener claro el esquema de una función recursiva:**

si (condición de parada (d(x))) entonces
    caso directo (h(x))
sino
    cálculo de los parámetros de entrada para el caso "n-1" (s(x))
    llamada a la función
    cálculo del resultado a partir del resultado del caso "n-1" (c(x,v))

2.3. Derivación de funciones recursivas

- **Para derivar una función recursiva habrá que seguir los siguientes pasos:**
  1. Especificación: Cabecera, precondición y postcondición.
  2. Buscar función de cota: Es decir, decidir sobre que parámetros se va a realizar la recursión.
  3. Identificar casos directos y recursivos: Al menos uno de cada. Comprobar que se cubren todos los casos posibles. Escribir el esqueleto de la función.
  4. Verificar que la función de cota es correcta.
  5. Deducir el código de los casos directos y recursivos.

2.3. Derivación de funciones recursivas

- **Ejemplo 1:** Función factorial.
- **Ejemplo 2:** Ejemplo en C++: Convertir un natural a binario.
- **Ejemplo 3:** Ejemplo en C++: Invertir una cola.
- **Ejemplo 4:** Suma de un vector.
- **Ejemplo 5:** Tamaño de un árbol.