Programación 2

La previa:
Recursión

Invocaciones

P

Cód-1

---

call Q1

Cód-2

---

p0

Q1

Cód-3

---

call Q2

Cód-4

Q2

Cód-5

---

call Q3

Cód-6

Q3

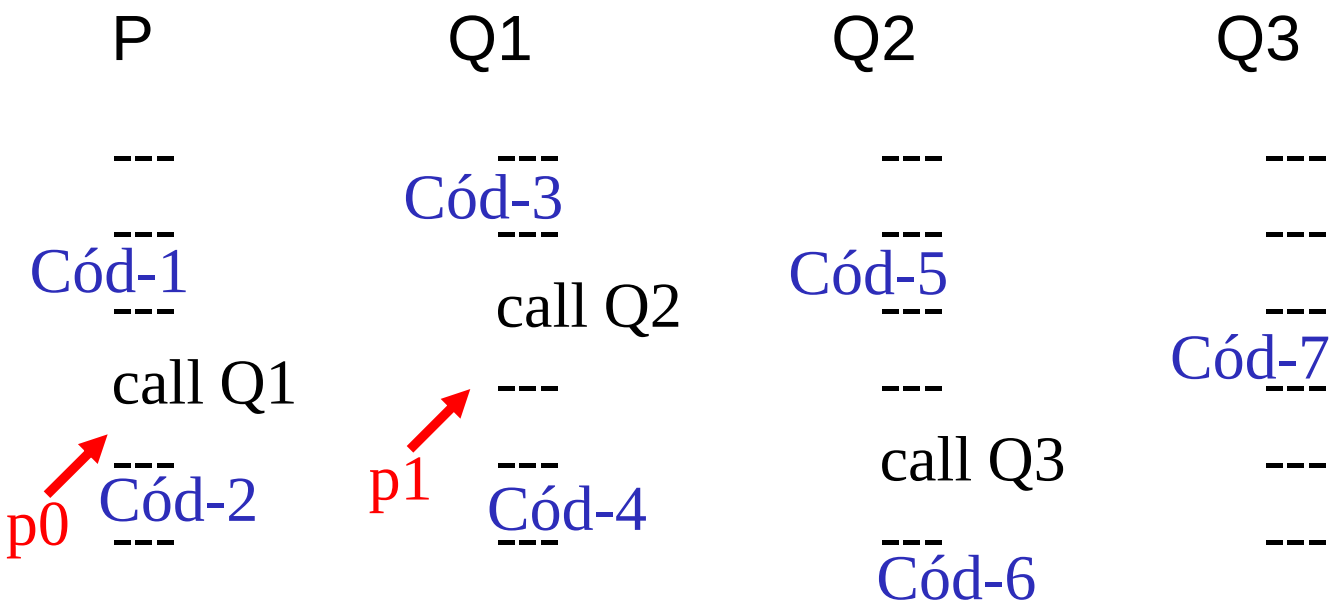
Cód-7

STACK

p_0

Ejecución: Cód-1

Invocaciones



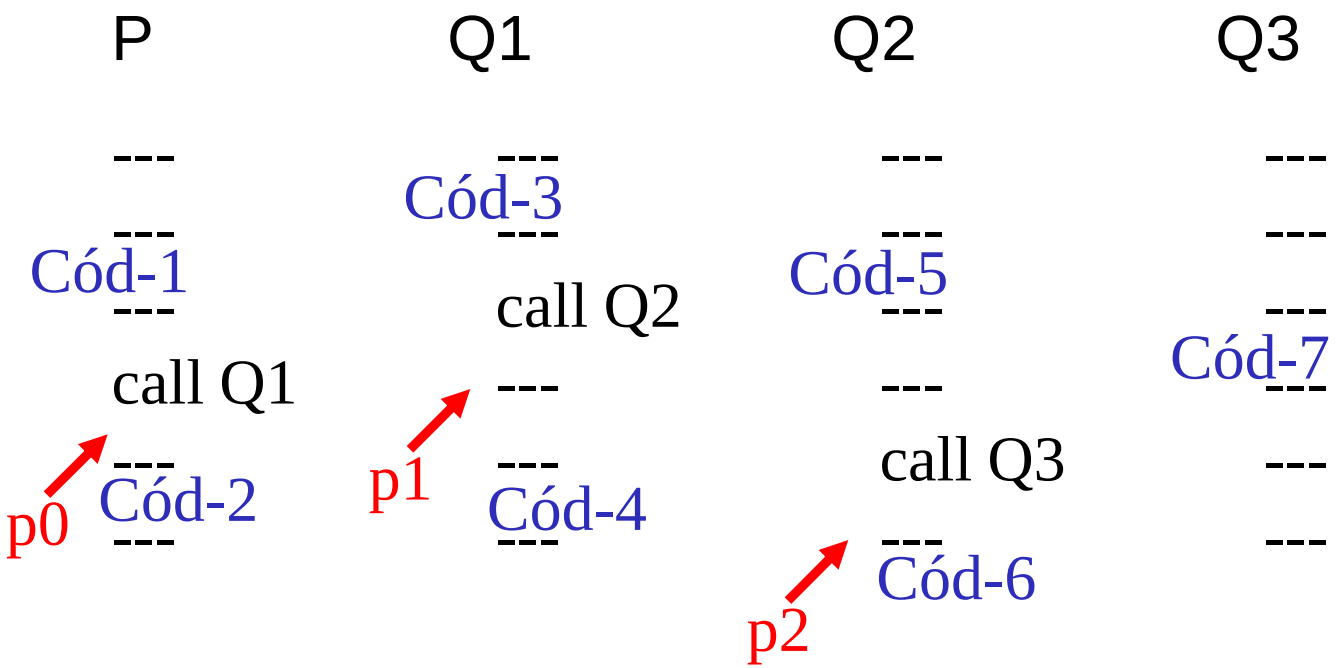
Ejecución: Cód-1, Cód-3

STACK

p_1

p_0

Invocaciones



Ejecución: Cód-1, Cód-3, Cód-5

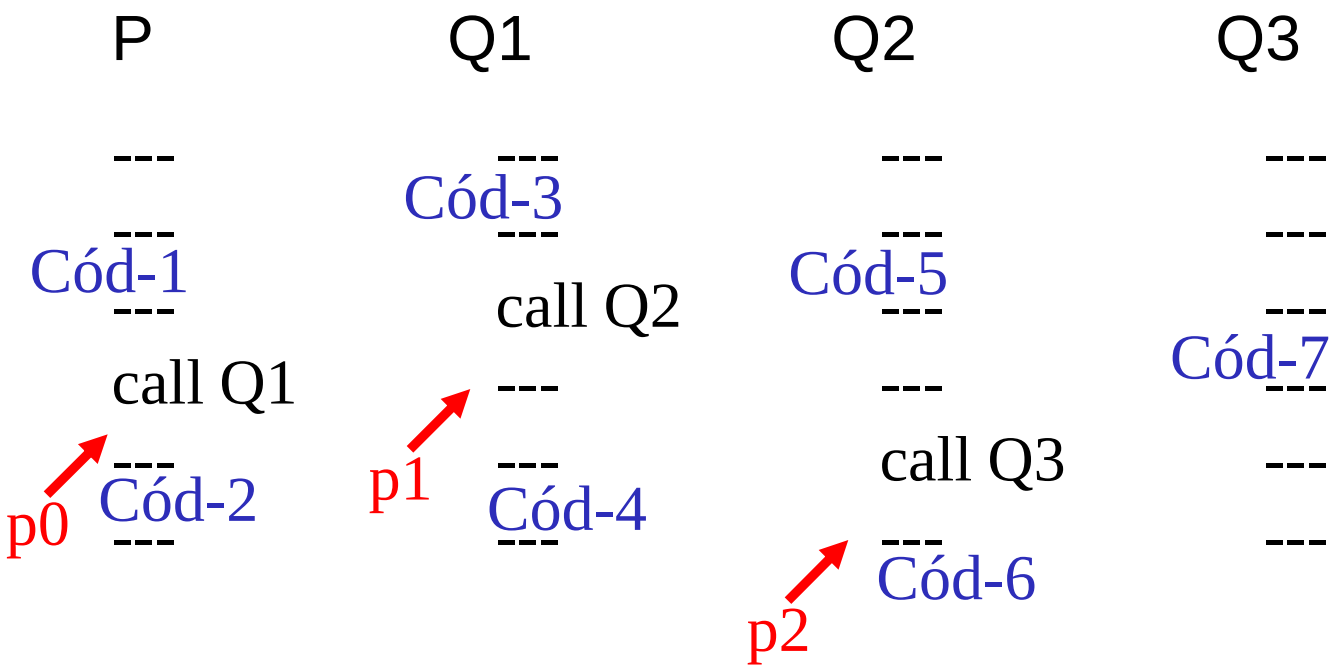
STACK

p2

p_1

p_0

Invocaciones



Ejecución: Cód-1, Cód-3, Cód-5, Cód-7

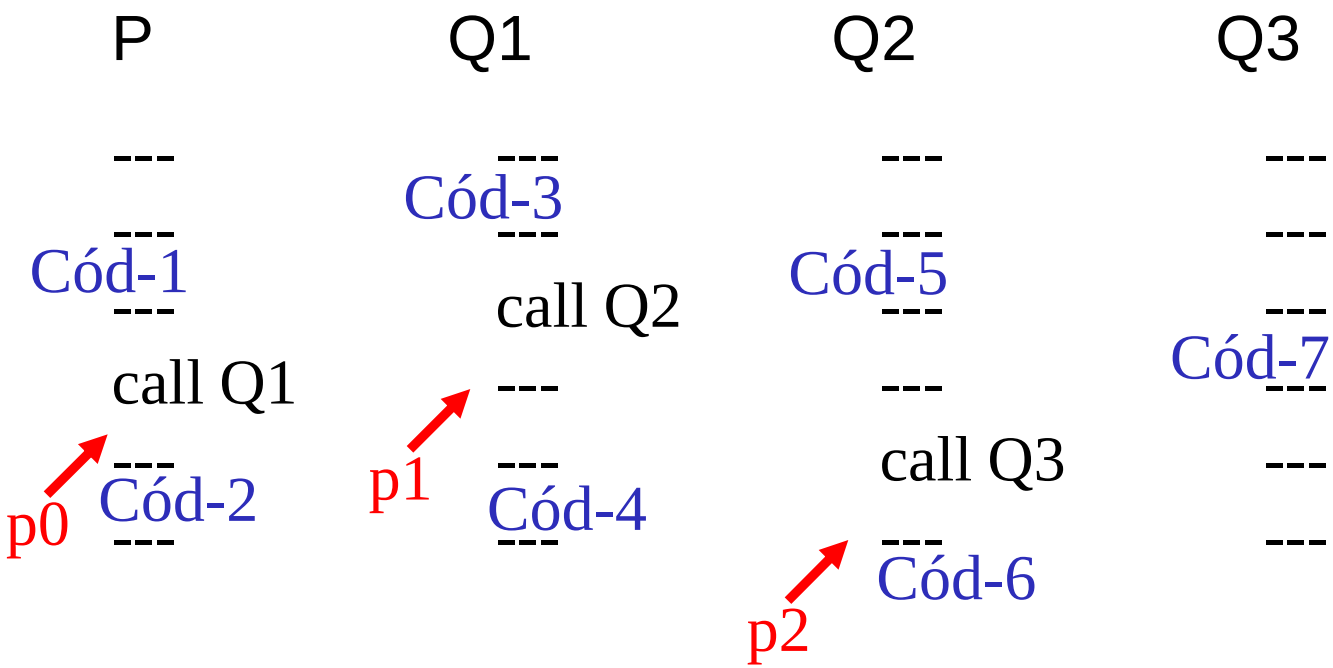
STACK

p2

p_1

p_0

Invocaciones



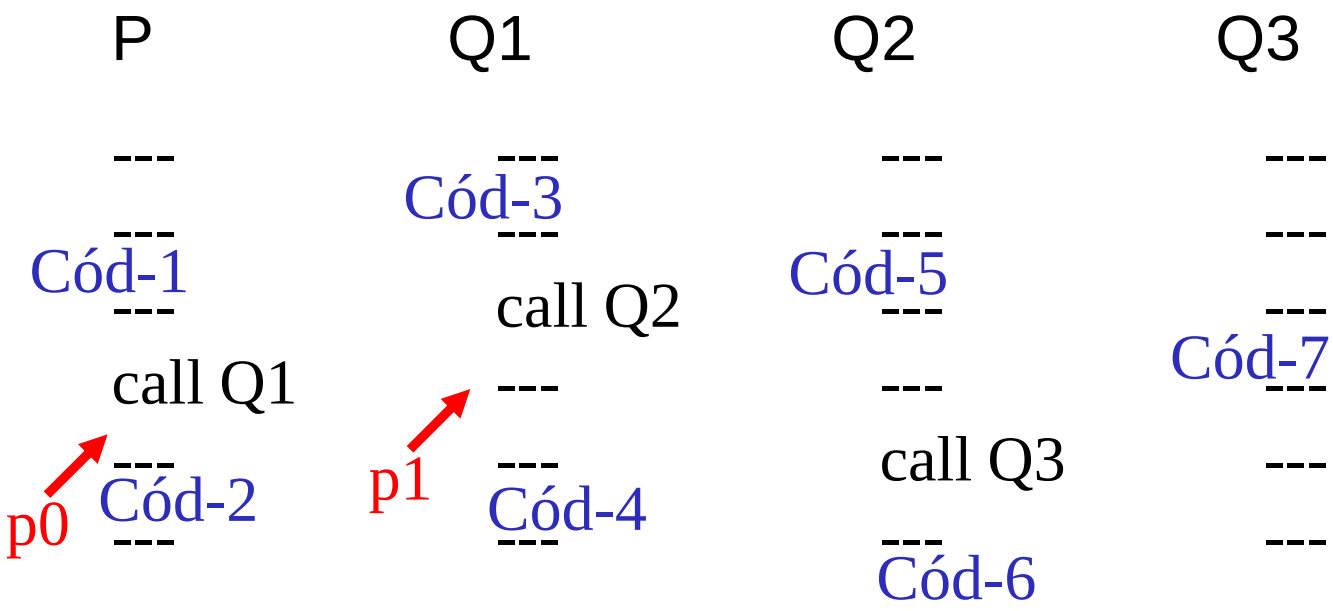
Ejecución: Cód-1, Cód-3, Cód-5, Cód-7, Cód-6

STACK

p_1

p_0

Invocaciones

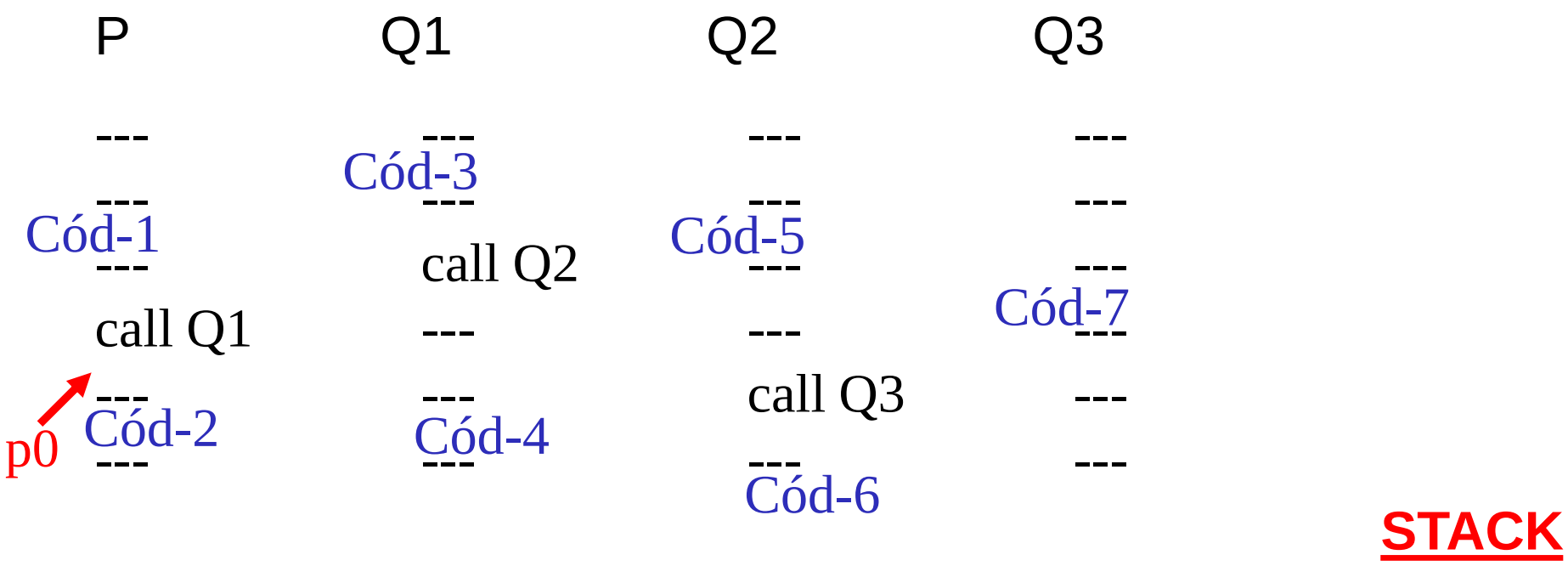


STACK

p_0

Ejecución: Cód-1, Cód-3, Cód-5, Cód-7, Cód-6, Cód-4

Invocaciones



Ejecución: Cód-1, Cód-3, Cód-5, Cód-7, Cód-6, Cód-4, Cód-2

A ver si entendimos…

```
Procedimiento P (x)
    Si x = 0 entonces
        Imprimir x
    Sino
        Imprimir x
        P (x-1)
        Imprimir (-1) * x
```

El llamado P(3), ¿qué salida produce?

A ver si entendimos…

```
Procedimiento P (x)
    Si x = 0 entonces
        Imprimir x
    Sino
        Imprimir x
        P (x-1)
        Imprimir (-1) * x
```

p

STACK (p)

1) Imprimir (-1)*x, x=3

Llamados: P(3) -> P(2)

Se imprime: 3,

A ver si entendimos…

```
Procedimiento P (x)
    Si x = 0 entonces
        Imprimir x
    Sino
        Imprimir x
        P (x-1)
        Imprimir (-1) * x
```

p

STACK (p)

2) Imprimir (-1)*x, x=2

1) Imprimir (-1)*x, x=3

Llamados: P(3) -> P(2) -> P(1)

Se imprime: 3, 2

A ver si entendimos…

```
Procedimiento P (x)
    Si x = 0 entonces
        Imprimir x
    Sino
        Imprimir x
        P (x-1)
        Imprimir (-1) * x
```

p

STACK (p)

3) Imprimir (-1)*x, x=1
2) Imprimir (-1)*x, x=2
1) Imprimir (-1)*x, x=3

Llamados: P(3) -> P(2) -> P(1) -> P(0)

Se imprime: 3, 2, 1,

A ver si entendimos…

```
Procedimiento P (x)
    Si x = 0 entonces
        Imprimir x
    Sino
        Imprimir x
        P (x-1)
        Imprimir (-1) * x
```

p

STACK (p)

3) Imprimir (-1)*x, x=1
2) Imprimir (-1)*x, x=2
1) Imprimir (-1)*x, x=3

Llamados: P(0)

Se imprime: 3, 2, 1, 0

A ver si entendimos…

```
Procedimiento P (x)
    Si x = 0 entonces
        Imprimir x
    Sino
        Imprimir x
        P (x-1)
        Imprimir (-1) * x
```

p

STACK (p)

2) Imprimir (-1)*x, x=2

1) Imprimir (-1)*x, x=3

Se imprime: 3, 2, 1, 0, -1

A ver si entendimos…

```
Procedimiento P (x)
    Si x = 0 entonces
        Imprimir x
    Sino
        Imprimir x
        P (x-1)
        Imprimir (-1) * x
```

p

STACK (p)

1) Imprimir (-1)*x, x=3

Se imprime: 3, 2, 1, 0, -1, -2

A ver si entendimos…

```
Procedimiento P (x)
    Si x = 0 entonces
        Imprimir x
    Sino
        Imprimir x
        P (x-1)
        Imprimir (-1) * x
```

p

STACK (p)

Se imprime: 3, 2, 1, 0, -1, -2, -3

De recursión (de cola) a iteración

¿Cómo transformar este código a otro equivalente, sin recursión?

Procedimiento P (x)

Si CasoBase (x) entonces

AcciónBase (x)

Sino

AcciónAntes (x)

P (Transformación (x))

~~AcciónDespués (x)~~ **“recursión de cola”**

De recursión (de cola) a iteración

Procedimiento P' (x)

x' = x

Mientras NO CasoBase (x')

AcciónAntes (x')

x' = Transformación (x')

FinMientras

AcciónBase (x')

¿Es conveniente la recursión cuando es de cola?

De recursión a iteración

¿Cómo transformar este código a otro equivalente, sin recursión?

Procedimiento P(x)

Si CasoBase (x) entonces

AcciónBase (x)

Sino

AcciónAntes (x)

P (Transformación (x))

AcciónDespués (x)

De recursión a iteración

Procedimiento P' (x)

x' = x

Pila s Vacía

Mientras NO CasoBase (x')

AcciónAntes (x')

Aplilar (x', s)

x' = Transformación (x')

AcciónBase (x')

Mientras NO PilaVacía (s)

AcciónDespués (Tope (s))

DesapilarTope (s)

¿Es conveniente la iteración para una recursión que no es de cola?

Formalmente: conjuntos inductivos, pruebas por inducción y recursión

Regla 1 : **0** es un natural (N)

Regla 2 : Si **n** es un natural entonces (**S n**) es otro natural

Regla 3 : Esos son todos los naturales

- **0** y **S** son llamados (operadores) CONSTRUCTORES del conjunto N
- La Regla 3 permite justificar el PRINCIPIO de DEMOSTRACIÓN por INDUCCIÓN ESTRUCTURAL Y EL ESQUEMA DE RECURSIÓN ESTRUCTURAL

f : N → …

f(0) = …

f(S n) = … f(n)

Formalmente: conjuntos inductivos, pruebas por inducción y recursión

Regla 1 : **0** es un natural (N)

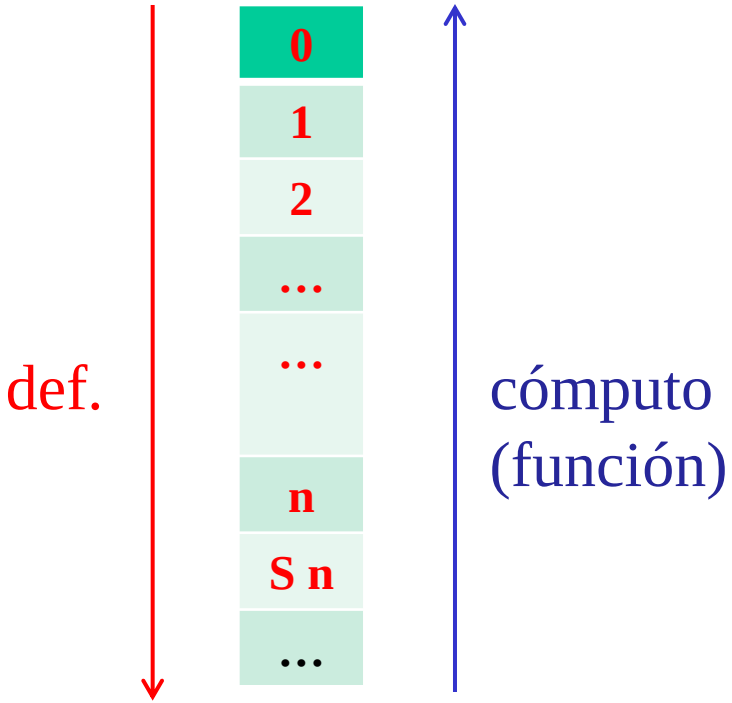
Regla 2 : Si **n** es un natural entonces (**S n**) es otro natural

Regla 3 : Esos son todos los naturales

f : N → …

f(0) = …

f(S n) = … f(n)



Factorial

fact: N $\rightarrow$ N

fact(0) = 1

fact(S n) = (S n) * fact(n)

Factorial

fact: N $\rightarrow$ N

fact(0) = 1

fact(S n) = (S n) * fact(n)

Ej: fac 3 = 3 * fac 2 = 3 * 2 * fac 1 = 3 * 2 * 1 * fac 0 = 3 * 2 * 1 * 1 = 6

Factorial

fact: N → N

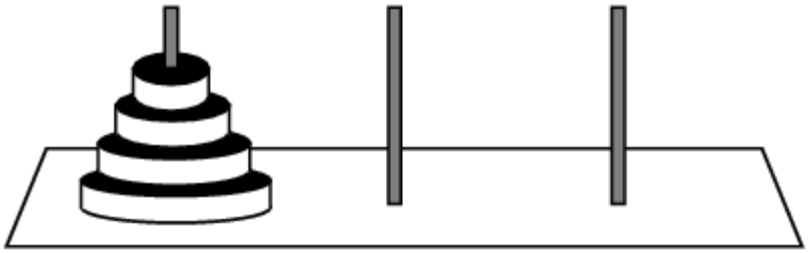
fact(0) = 1

fact(S n) = (S n) * fact(n)

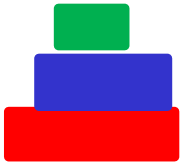
Ej: fac 3 = 3 * fac 2 = 3 * 2 * fac 1 = 3 * 2 * 1 * fac 0 = 3 * 2 * 1 * 1 = 6

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

Torres de Hanoi



Origen *Destino* *Auxiliar*



Origen *Destino* *Auxiliar*

Hanoi


```
void hanoi(int n, char origen, char destino, char auxiliar){
      if(n > 0){

          /* Mover los n-1 discos de "origen" a "auxiliar" usando "destino" como auxiliar */
          hanoi(n-1, origen, auxiliar, destino);

          /* Mover disco n de "origen" para "destino" */
          printf("\n Mover disco %d de base %c para a base %c", n, origen, destino);

          /* Mover los n-1 discos de "auxiliar" a "destino" usando "origen" como auxiliar */
          hanoi(n-1, auxiliar, destino, origen);
}

main(){
          int n;
          printf("Digite el número de discos: ");
          scanf("%d",&n);
          hanoi(n, 'A', 'C', 'B');
          return 0;
}
```

Listas

LISTAS:

- Vamos a definir el conjunto de las listas secuenciales finitas de naturales Inductivamente:
  - Regla 1**: lista vacía**

$$\frac{}{[] : \text{Lista}}$$

  - Regla 2**: listas no vacías (cons)**

$$\frac{n : \mathbb{N} \qquad S : \text{Lista}}{n.S : \text{Lista}}$$

  - Regla 3**: esas son todas las listas**
- Ejemplos**:**

[]
1.[] **([1])**
3.1.[] **([3,1]) notación sintética**

Recursión estructural en listas

- **[] : Lista**
- Si **x : N y S : Lista entonces x.S : Lista**

f : Lista → …

f([]) = …

f(x.S) = … f(S)

Largo

f : Lista → …

f([]) = …
f(x.S) = … f(S)

Ejemplo:

largo : Lista → N

largo([]) = …
largo(x.S) = … largo(S)

Largo

f : Lista → …

f([]) = …
f(x.S) = … f(S)

Ejemplo:

largo : Lista → N

largo([]) = 0
largo(x.S) = 1 + largo(S)

Pertenece

– Chequear si un elemento está en una lista.

pertenece: N x Lista → bool

pertenece (e,[]) = …

pertenece (e,x.S) = … pertenece(e,S)

Pertenece

– Chequear si un elemento está en una lista.

pertenece: N x Lista → bool

pertenece (e,[]) = false

pertenece (e,x.S) = (e==x) || pertenece(e,S)

??

– Qué hace?

f: N x Lista → bool

f (e,[]) = true

f (e,x.S) = (e==x) && f(e,S)

Inserción ordenada

– Insertar de manera ordenada un elemento en una lista ordenada.

Precondición: lista parámetro ordenada ($\leq$)

insOrd: N x Lista $\rightarrow$ Lista

insOrd (e,[]) = …

insOrd (e,x.S) = … insOrd(e,S)

Inserción ordenada

– Insertar de manera ordenada un elemento en una lista ordenada.

Precondición: lista parámetro ordenada ($\leq$)

insOrd: N x Lista $\rightarrow$ Lista

insOrd (e,[]) = e.[]

insOrd (e,x.S) = e.x.S, si e<=x

x.insOrd(e,S), sino

Ordenación por inserción

– Ordenar una lista de menor a mayor.

Ord: Lista → Lista

Ord ([]) = …

Ord (x.S) = … Ord(S)

Ordenación por inserción

– Ordenar una lista de menor a mayor.

Ord: Lista → Lista

Ord ([]) = []

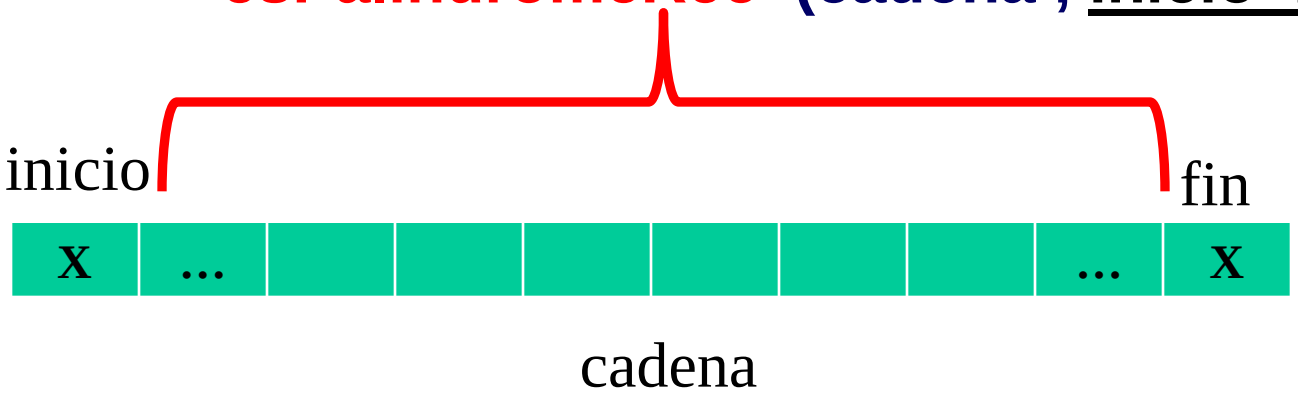
Ord (x.S) = insOrd(x, Ord(S))

Práctico de Recursión
Palíndrome – Ej.2

Implemente un algoritmo recursivo que determine si un string es palíndrome.

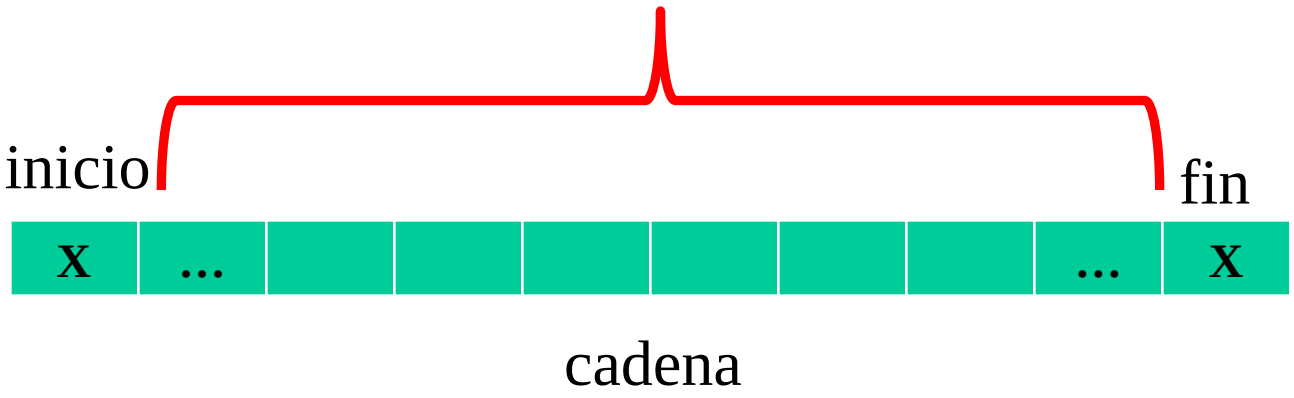
Palíndrome – Ej.2 Recursión

```
bool esPalindromeRec (char * cadena , int inicio , int fin) {

    if (inicio >= fin) return true;

    else return (cadena[inicio] == cadena[fin]) &&
                esPalindromeRec  (cadena , inicio+1 , fin-1);
}
```



Palíndrome – Ej.2 Recursión

```
bool esPalindromeRec (char * cadena , int inicio , int fin) {

    return (inicio >= fin) ||
            ((cadena[inicio] == cadena[fin]) &&
            esPalindromeRec  (cadena , inicio+1 , fin-1));
}
```



Palíndrome – Ej.2 Recursión

```
int main() { // ejemplo
    char cadena[cte];
    scanf("%s", &cadena);
    int longitud = strlen(cadena);
    bool resultado = esPalindromeRec (cadena, 0 , longitud - 1);
    if (resultado)
        printf(“Recursivamente, '%s' es palíndrome\n", cadena);
    else
        printf(“Recursivamente, '%s’ no es palíndrome\n", cadena);
  }
```

Palíndrome – Ej.2 Recursión

Un arreglo de caracteres con su largo

```
bool esPalindromeRec (char * cadena , int inicio , int fin) {

    return (inicio >= fin) ||
            ((cadena[inicio] == cadena[fin]) &&
            esPalindromeRec  (cadena , inicio+1 , fin-1));
}


bool esPalindrome (char * cadena , int largo) {

    return esPalindromeRec  (cadena , 0 , largo-1));
}
```