

Algoritmos y Estructuras de Datos I

Segundo cuatrimestre de 2025

Departamento de Computación - FCEyN - UBA

Recursión sobre enteros

1

IP - AED I: Temario de la clase

- Recursión sobre enteros
 - ¿Qué es la recursión?
 - Reducción en la recursión
 - ¿Cómo pensar recursivamente?
 - Inducción y recursión
 - Generalización de funciones
 - Recursión en más de un parámetro
 - Algunos ejercicios de la guía 4

2

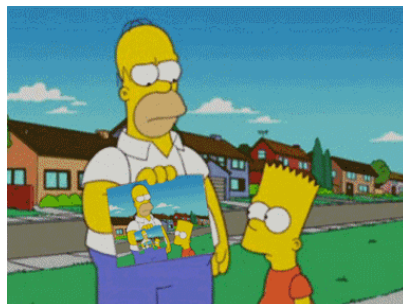
Recursión

- Hasta ahora, especificamos funciones que consistían en “expresiones sencillas”.
- ¿Cómo es una función en Haskell para calcular el factorial de un número $n \in \mathbb{N}_0$?

$$n! = \prod_{k=1}^n k \qquad n! = \begin{cases} 1 & \text{si } n = 0 \\ n \times (n-1)! & \text{si } n > 0 \end{cases}$$

¡La segunda definición de `factorial` involucra a esta misma función del lado derecho!

```
factorial :: Int → Int
factorial n
| n == 0 = 1
| n > 0 = n * factorial (n-1)
```



3

Recursión y reducción

¿Podemos definirla usando *otherwise*?

```
factorial :: Int → Int
factorial n | n == 0 = 1
            | otherwise = n * factorial (n-1)
```

¿Podemos definirla usando *pattern matching*?

```
factorial :: Int → Int
factorial 0 = 1
factorial n = n * factorial (n-1)
```

¿Cómo reduce la expresión `factorial 3`?

```
factorial 3 ~> 3 * factorial 2 ~> 3 * (2 * factorial 1) ~>
~> 3 * (2 * (1 * factorial 0)) ~> 3 * (2 * (1 * 1)) ~> 3 * (2 * 1)
~> 3 * 2 ~> 6
```

4

Asegurarse de llegar a un caso base

Veamos este programa recursivo para determinar si un entero positivo es par:

```
esPar :: Int → Bool
esPar n | n==0 = True
        | otherwise = esPar (n-2)
```

¿Qué problema tiene esta función?

¿Cómo se arregla?

```
esPar :: Int → Bool
esPar n | n==0 = True
        | n==1 = False
        | otherwise = esPar (n-2)
```

```
esPar :: Int → Bool
esPar n | n==0 = True
        | otherwise = not (esPar (n-1))
```

5

¿Cómo pensar recursivamente?

- ▶ Si queremos definir una función recursiva, por ejemplo factorial,
 - ▶ en el paso recursivo, **suponiendo** que tenemos el resultado para el caso anterior, ¿qué falta para poder obtener el resultado que quiero? En este caso, suponemos ya calculado factorial (n-1) y lo combinamos multiplicándolo por n para lograr obtener factorial n.
 - ▶ además, identificamos el (o los) **casos base**. En el ejemplo de factorial, definimos como casos base la función sobre 0: $\text{factorial } n \mid n == 0 = 1$
- ▶ Propiedades de una definición recursiva:
 - ▶ las **llamadas recursivas** tienen que “acercarse” a un caso base.
 - ▶ tiene que tener uno o más **casos base** que dependerán del tipo de llamado recursivo. Un caso base, es aquella expresión que no tiene un llamado recursivo.

6

¿Cómo pensar recursivamente?

- ▶ Casos bases: identificar el o los casos bases.
- ▶ Casos recursivos: **suponiendo que la llamada recursiva es correcta**, ¿qué tengo que hacer para completar la solución?

Otro Ejemplo:

```
sumaLosPrimerosNImpares :: Integer → Integer
sumaLosPrimerosNImpares n
  | n == 1 = 1
  | n > 1 = sumaLosPrimerosNImpares (n-1) + n
```

- ▶ Verificar que (n==1) es el caso base, está bien definido y no hay otros.
- ▶ Si podemos dar una solución correcta en base a una llamada recursiva correcta entonces, por inducción, ¡todos van a ser correctos!

Con el paso anterior resuelto: ¿Qué falta para que el nuevo paso esté resuelto?

```
| n > 1 = n_esimoImpar + sumaLosPrimerosNImpares (n-1)
```

Cambiamos el problema: ahora sólo falta definir n_esimoImpar.

```
| n > 1 = n_esimoImpar + sumaLosPrimerosNImpares (n-1)
  where n_esimoImpar = 2*n - 1
```

7

Inducción vs. Recursión

- | | |
|--|--|
| <ul style="list-style-type: none">▶ Probar por inducción $P(n) : \sum_{i=1}^n (2i-1) = n^2$▶ Vale para $n=1 : \sum_{i=1}^1 (2i-1) = 1^2$▶ Supongo que vale $P(n)$, quiero probar $P(n+1)$▶ ¿Qué relación hay entre $\sum_{i=1}^n (2i-1)$ y $\sum_{i=1}^{n+1} (2i-1)$? $\sum_{i=1}^{n+1} (2i-1) = \left(\sum_{i=1}^n (2i-1) \right) + 2n+1$▶ Uso la Hipótesis Inductiva $P(n)$: $\sum_{i=1}^{n+1} (2i-1) = n^2 + 2n+1 = (n+1)^2$▶ ¡¿Pero cómo?! ¡¿Estoy usando lo que quiero probar?! Ah, claro... vale $P(1)$ y $P(n) \Rightarrow P(n+1)$, entonces ¡vale para todo n! (por teo Inducción) | <ul style="list-style-type: none">▶ Implementar una función recursiva para $f(n) = \sum_{i=1}^n (2i-1)$▶ Caso base en Haskell: $f \ 1 = 1$▶ Supongo que ya sé calcular $f(n-1)$, quiero calcular $f(n)$▶ ¿Qué relación hay entre $\sum_{i=1}^{n-1} (2i-1)$ y $\sum_{i=1}^n (2i-1)$? $\sum_{i=1}^n (2i-1) = \left(\sum_{i=1}^{n-1} (2i-1) \right) + 2n-1$▶ Uso la función que sé calcular: $f(n) = f(n-1) + 2n-1$ En Haskell: $f \ n = f \ (n-1) + 2*n - 1$▶ ¡¿Pero cómo?! ¡¿Estoy usando la función que quiero definir?! Ah, claro... está definido $f(1)$ y con $f(n-1)$ sé obtener $f(n)$, entonces ¡puedo calcular f para todo n! |
|--|--|

8

Generalización de funciones

¿Una fácil?... o no tanto

- Implementar una función `sumaDivisores :: Integer → Integer` que calcule la suma de los divisores de un número entero positivo.

problema *sumaDivisores*($n : \mathbb{Z}$) : \mathbb{Z} {
 requiere: $\{n > 0\}$
 asegura: $\{res = \sum_{i=1}^n (si\ (n \bmod i = 0)\ \text{entonces}\ i\ \text{sino}\ 0)\}$ }

Pregunta clave: ¿alcanza con hacer recursión sobre n ?

No hay ninguna relación sencilla entre `sumaDivisores n` y `sumaDivisores (n-k)` (para ningún k particular).

¿Qué sucede si definimos primero una función **más general** que devuelve la suma de los divisores de un número hasta cierto punto?

`sumaDivisoresHasta :: Integer → Integer → Integer`

Ahora **sí** existe una relación sencilla entre `sumaDivisoresHasta n k` y `sumaDivisoresHasta n (k-1)`. ¿Por qué?

9

Generalización de funciones

Veamos cómo sería la especificación:

problema *sumaDivisoresHasta*($n : \mathbb{Z}, k : \mathbb{Z}$) : \mathbb{Z} {
 requiere: $\{(n > 0) \wedge (k > 0)\}$
 asegura: $\{res = \sum_{i=1}^k (si\ (n \bmod i = 0)\ \text{entonces}\ i\ \text{sino}\ 0)\}$ }

Ahora podemos definir esta función en Haskell recursivamente

```
sumaDivisoresHasta :: Integer → Integer → Integer
sumaDivisoresHasta n 1 = 1
sumaDivisoresHasta n i | (mod n i == 0) = i +
    sumaDivisoresHasta n (i-1)
    | otherwise = sumaDivisoresHasta n (i-1)
```

¿Y por último, cómo definimos `SumaDivisores` utilizando lo anterior?

```
sumaDivisores :: Integer → Integer
sumaDivisores n = sumaDivisoresHasta n n
```

Entonces, `SumaDivisores`, ¿es una función recursiva?

10

Recursión en más de un parámetro

Implementar la siguiente función:

$$f(n, m) = \sum_{i=1}^n \sum_{j=1}^m i^j$$

Veamos primero la especificación:

problema *sumatoriaDoble*($n : \mathbb{Z}, m : \mathbb{Z}$) : \mathbb{Z} {
 requiere: $\{(n > 0) \wedge (m > 0)\}$
 asegura: $\{res = \sum_{i=1}^n \sum_{j=1}^m i^j\}$
}

Pregunta clave: ¿alcanza con hacer recursión sobre n ?

¿Qué sucede si definimos primero una función **más específica** que devuelve la sumatoria interna?

`sumatoriaInterna :: Integer → Integer → Integer`

Ahora parece más sencillo definir `sumatoriaDoble n m` utilizando `sumatoriaInterna n m`. ¿Cómo lo hacemos?

11

Recursión en más de un parámetro

Veamos cómo sería la especificación:

problema *sumatoriaInterna*($n : \mathbb{Z}, m : \mathbb{Z}$) : \mathbb{Z} {
 requiere: $\{(n > 0) \wedge (m > 0)\}$
 asegura: $\{res = \sum_{j=1}^m n^j\}$
}

Ahora podemos definir esta función en Haskell recursivamente

```
sumatoriaInterna :: Integer → Integer → Integer
sumatoriaInterna _ 0 = 0
sumatoriaInterna n j = n^j + sumatoriaInterna n (j-1)
```

¿Y por último, cómo definimos `sumatoriaDoble` utilizando lo anterior?

```
sumatoriaDoble :: Integer → Integer → Integer
sumatoriaDoble 0 _ = 0
sumatoriaDoble n m = sumatoriaDoble (n-1) m + sumatoriaInterna n m
```

Entonces, `sumatoriaDoble`, ¿cuántas recursiones involucra?

12