Más sobre recursión

Taller de Álgebra I

2do Cuatrimestre 2017

Recordando recursión

La clase pasada vimos:

- **Recursión**. El modo de pensar funciones recursivas.

Recordando recursión

La clase pasada vimos:

- **Recursión**. El modo de pensar funciones recursivas.

Para pensar funciones recursivas:

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

Recordando recursión

La clase pasada vimos:

- **Recursión**. El modo de pensar funciones recursivas.

Para pensar funciones recursivas:

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

```
sumaLosPrimerosNImpares :: Integer -> Integer
sumaLosPrimerosNImpares n
      | n == 1 = 1
      | n > 0 = ... sumaLosPrimerosNImpares (n-1) ...
```

- Fíjense que el último llamado recursivo (n=1) es efectivamente correcto, es el caso base.
- Si podemos dar una solución correcta en base a una llamada recursiva correcta entonces, por inducción, ¡todos van a ser correctos! En particular el segundo. Y por lo tanto el tercero. Y por...

Recordando recursión

La clase pasada vimos:

- **Recursión**. El modo de pensar funciones recursivas.

Para pensar funciones recursivas:

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

```
sumaLosPrimerosNImpares :: Integer -> Integer
sumaLosPrimerosNImpares n
      | n == 1 = 1
      | n > 0 = ... sumaLosPrimerosNImpares (n-1) ...
```

- Fíjense que el último llamado recursivo (n=1) es efectivamente correcto, es el caso base.
- Si podemos dar una solución correcta en base a una llamada recursiva correcta entonces, por inducción, ¡todos van a ser correctos! En particular el segundo. Y por lo tanto el tercero. Y por...

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

Recordando recursión

La clase pasada vimos:

- **Recursión**. El modo de pensar funciones recursivas.

Para pensar funciones recursivas:

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

```
sumaLosPrimerosNImpares :: Integer -> Integer
sumaLosPrimerosNImpares n
      | n == 1 = 1
      | n > 0 = ... sumaLosPrimerosNImpares (n-1) ...
```

- Fíjense que el último llamado recursivo (n=1) es efectivamente correcto, es el caso base.
- Si podemos dar una solución correcta en base a una llamada recursiva correcta entonces, por inducción, ¡todos van a ser correctos! En particular el segundo. Y por lo tanto el tercero. Y por...

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

```
      | n > 0 = n_esimoImpar + sumaLosPrimerosNImpares (n-1)
```

Cambiamos el problema: ahora solo falta definir `n_esimoImpar`.

Recordando recursión

La clase pasada vimos:

- **Recursión**. El modo de pensar funciones recursivas.

Para pensar funciones recursivas:

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

```
sumaLosPrimerosNImpares :: Integer -> Integer
sumaLosPrimerosNImpares n
      | n == 1 = 1
      | n > 0 = ... sumaLosPrimerosNImpares (n-1) ...
```

- Fíjense que el último llamado recursivo (n=1) es efectivamente correcto, es el caso base.
- Si podemos dar una solución correcta en base a una llamada recursiva correcta entonces, por inducción, ¡todos van a ser correctos! En particular el segundo. Y por lo tanto el tercero. Y por...

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

```
      | n > 0 = n_esimoImpar + sumaLosPrimerosNImpares (n-1)
```

Cambiamos el problema: ahora solo falta definir `n_esimoImpar`.

Recordando recursión

La clase pasada vimos:

- **Recursión**. El modo de pensar funciones recursivas.

Para pensar funciones recursivas:

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

```
sumaLosPrimerosNImpares :: Integer -> Integer
sumaLosPrimerosNImpares n
      | n == 1 = 1
      | n > 0 = ... sumaLosPrimerosNImpares (n-1) ...
```

- Fíjense que el último llamado recursivo (n=1) es efectivamente correcto, es el caso base.
- Si podemos dar una solución correcta en base a una llamada recursiva correcta entonces, por inducción, ¡todos van a ser correctos! En particular el segundo. Y por lo tanto el tercero. Y por...

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

```
      | n > 0 = n_esimoImpar + sumaLosPrimerosNImpares (n-1)
```

Cambiamos el problema: ahora solo falta definir `n_esimoImpar`.

```
      | n > 0 = n_esimoImpar + sumaLosPrimerosNImpares (n-1)
      where n_esimoImpar = 2 * n - 1
```

Inducción vs. Recursión

- Probar por inducción $P(n) : \sum_{i=1}^{n}(2i-1) = n^2$

- Implementar una función recursiva para $f(n) = \sum_{i=1}^{n}(2i-1)$

Inducción vs. Recursión

- Probar por inducción $P(n) : \sum_{i=1}^{n}(2i-1) = n^2$
- Vale para $n = 1 : \sum_{i=1}^{1}(2i-1) = 1^2$

- Implementar una función recursiva para $f(n) = \sum_{i=1}^{n}(2i-1)$
- Caso base: `f n | (n == 1) = 1`

Inducción vs. Recursión

- 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 $P(n+1)$

- Implementar una función recursiva para $f(n) = \sum_{i=1}^{n}(2i-1)$
- Caso base: `f n | (n == 1) = 1`
- Supongo que ya se calcular $f(n-1)$, quiero calcular $f(n)$

Inducción vs. Recursión

- 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 $\sum_{i=1}^{n+1}(2i-1)$?

$$\sum_{i=1}^{n+1}(2i-1) = \left(\sum_{i=1}^{n}(2i-1)\right) + 2n + 1$$

- Implementar una función recursiva para $f(n) = \sum_{i=1}^{n}(2i-1)$
- Caso base: `f n | (n == 1) = 1`
- Supongo que ya se calcular $f(n-1)$, quiero calcular $f(n)$
- ¿Qué relación hay entre $\sum_{i=1}^{n-1}(2i-1)$ y $\sum_{i=1}^{n}(2i-1)$?

$$\sum_{i=1}^{n}(2i-1) = \left(\sum_{i=1}^{n-1}(2i-1)\right) + 2n - 1$$

Inducción vs. Recursión

- 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 $P(n)$:

$$\sum_{i=1}^{n+1}(2i-1) = n^2 + 2n + 1 = (n+1)^2$$

- Implementar una función recursiva para $f(n) = \sum_{i=1}^{n}(2i-1)$
- Caso base: `f n | (n == 1) = 1`
- Supongo que ya se 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 se calcular:

$$f(n) = f(n-1) + 2n - 1$$

Inducción vs. Recursión

- 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?!

- Implementar una función recursiva para $f(n) = \sum_{i=1}^{n}(2i-1)$
- Caso base: `f n | (n == 1) = 1`
- Supongo que ya se 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 se calcular:

$$f(n) = f(n-1) + 2n - 1$$

- ¡¿Pero cómo?! ¡¿Estoy usando la función que quiero definir?!

Inducción vs. Recursión

- 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 $P(n) => P(n+1)$, entonces vale para todo n!

- Implementar una función recursiva para $f(n) = \sum_{i=1}^{n}(2i-1)$
- Caso base: `f n | (n == 1) = 1`
- Supongo que ya se 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 se calcular:

$$f(n) = f(n-1) + 2n - 1$$

- ¡¿Pero cómo?! ¡¿Estoy usando la función que quiero definir?!
- A ver... funciona!

¿Recursión con `Floats`? ¿Inducción sobre los reales?

Atención! A veces ciertas funciones esperan un `Float` y nosotros tenemos un `Integer`. Para estos casos podemos utilizar la función `fromInteger :: Integer -> Float`.

Ejercicios

- Implementar una función `eAprox :: Integer -> Float` que aproxime el valor del número e a partir de la siguiente sumatoria:

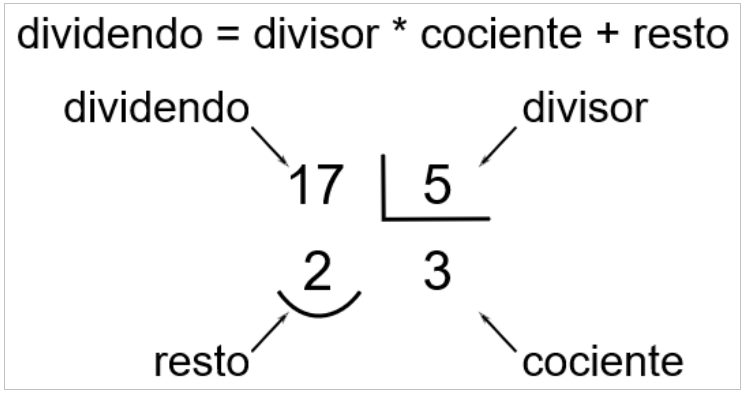
$$\hat{e}(n) = \sum_{i=0}^{n} \frac{1}{i!}$$

- Definir la constante `e :: Float` como la aproximación de e a partir de los primeros 100 términos de la serie anterior.
- Implementar una función `parteEntera :: Float -> Integer` que calcule la parte entera de un número real positivo.
- Cambiar la implementación de `parteEntera :: Float -> Integer` para que también funcione con números negativos.

Cociente y resto – Algoritmo de división

Dados a (dividendo), d (divisor) $\in \mathbb{Z}$, $d \neq 0$, existen únicos q (cociente), r (resto) $\in \mathbb{Z}$ tales que

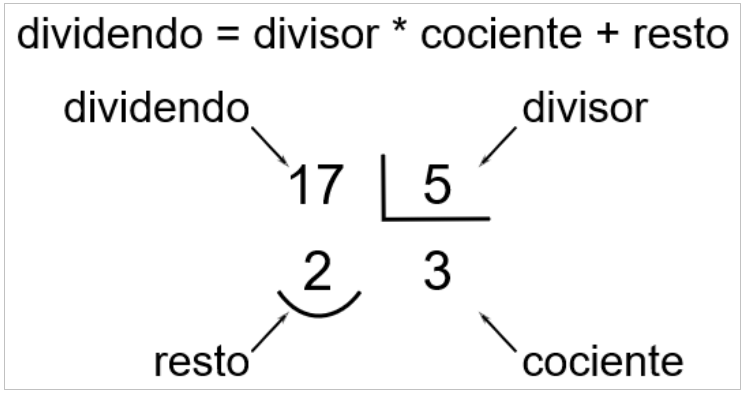
- $a = dq + r$,
- $0 \leq r < |d|$.



Cociente y resto – Algoritmo de división

Dados a (dividendo), d (divisor) $\in \mathbb{Z}$, $d \neq 0$, existen únicos q (cociente), r (resto) $\in \mathbb{Z}$ tales que

- $a = dq + r$,
- $0 \leq r < |d|$.



Implementar la siguiente función

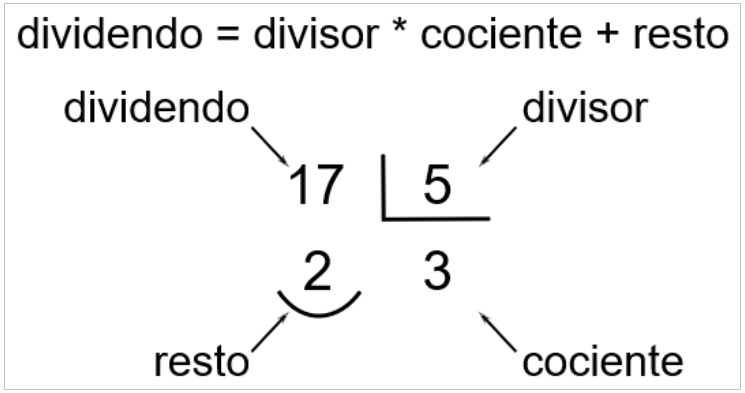
```
division :: Integer -> Integer -> (Integer, Integer)
```

Debe funcionar para $a \geq 0$, $d > 0$ y no se pueden usar `div`, `mod` ni `/`.

Cociente y resto – Algoritmo de división

Dados a (dividendo), d (divisor) $\in \mathbb{Z}$, $d \neq 0$, existen únicos q (cociente), r (resto) $\in \mathbb{Z}$ tales que

- $a = dq + r$,
- $0 \leq r < |d|$.



Implementar la siguiente función

```
division :: Integer -> Integer -> (Integer, Integer)
```

Debe funcionar para $a \geq 0$, $d > 0$ y no se pueden usar `div`, `mod` ni `/`.

Ideas

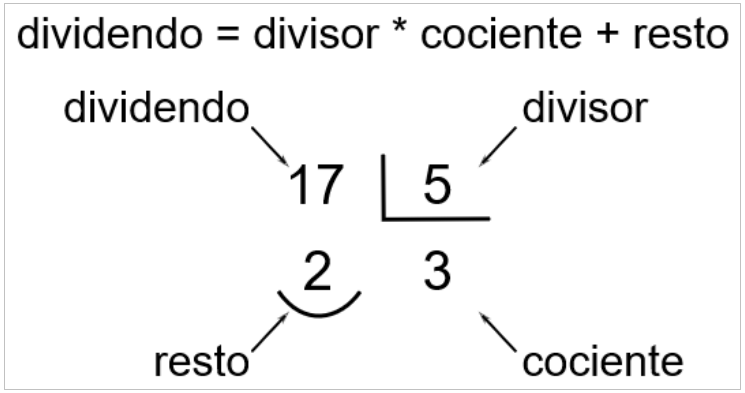
Usaremos recursión (idea sacada de la demostración del teorema de la división)

- `division a d = ??`

Cociente y resto – Algoritmo de división

Dados a (dividendo), d (divisor) $\in \mathbb{Z}$, $d \neq 0$, existen únicos q (cociente), r (resto) $\in \mathbb{Z}$ tales que

- $a = dq + r$,
- $0 \leq r < |d|$.



Implementar la siguiente función

```
division :: Integer -> Integer -> (Integer, Integer)
```

Debe funcionar para $a \geq 0$, $d > 0$ y no se pueden usar `div`, `mod` ni `/`.

Ideas

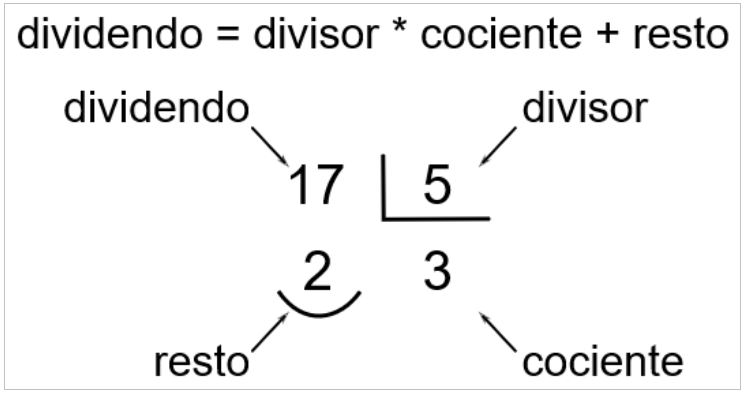
Usaremos recursión (idea sacada de la demostración del teorema de la división)

- `division a d = ??`
- Hacemos recursión... sobre d? Para `division 15 4`, me sirve `division 15 3`?

Cociente y resto – Algoritmo de división

Dados a (dividendo), d (divisor) $\in \mathbb{Z}$, $d \neq 0$, existen únicos q (cociente), r (resto) $\in \mathbb{Z}$ tales que

- $a = dq + r$,
- $0 \leq r < |d|$.



Implementar la siguiente función

```
division :: Integer -> Integer -> (Integer, Integer)
```

Debe funcionar para $a \geq 0$, $d > 0$ y no se pueden usar `div`, `mod` ni `/`.

Ideas

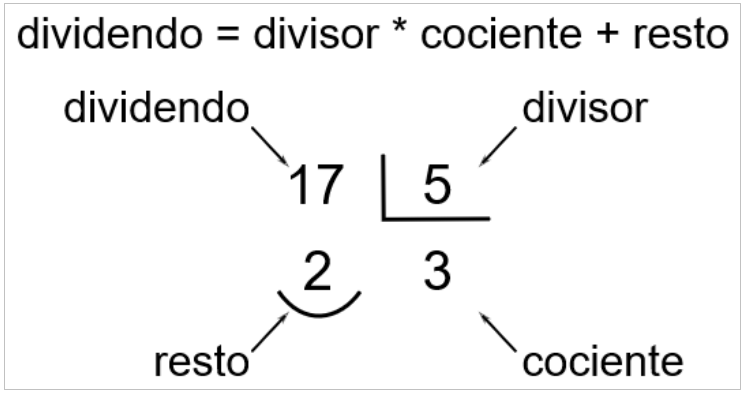
Usaremos recursión (idea sacada de la demostración del teorema de la división)

- `division a d = ??`
- Hacemos recursión... sobre d? Para `division 15 4`, me sirve `division 15 3`?
- Hacemos recursión... sobre a? Para `division 15 4`, me sirve `division 14 4`?

Cociente y resto – Algoritmo de división

Dados a (dividendo), d (divisor) $\in \mathbb{Z}$, $d \neq 0$, existen únicos q (cociente), r (resto) $\in \mathbb{Z}$ tales que

- $a = dq + r$,
- $0 \leq r < |d|$.



Implementar la siguiente función

```
division :: Integer -> Integer -> (Integer, Integer)
```

Debe funcionar para $a \geq 0$, $d > 0$ y no se pueden usar `div`, `mod` ni `/`.

Ideas

Usaremos recursión (idea sacada de la demostración del teorema de la división)

- `division a d = ??`
- Hacemos recursión... sobre d? Para `division 15 4`, me sirve `division 15 3`?
- Hacemos recursión... sobre a? Para `division 15 4`, me sirve `division 14 4`?
- Para `division 15 4`, me sirve `division k 4` para algún k?
- Para determinar lo anterior, pensar qué quiere decir dividir un número por otro.

Algoritmo de división

```
division :: Integer -> Integer -> (Integer, Integer)
division a d | a < d = (0, a)
             | otherwise = (fst (division (a-d) d) + 1,
                            snd (division (a-d) d))
```

Algoritmo de división

```
division :: Integer -> Integer -> (Integer, Integer)
division a d | a < d = (0, a)
             | otherwise = (fst (division (a-d) d) + 1,
                            snd (division (a-d) d))
```

¿Se puede no poner dos veces `division (a-d) d`? Sí:

Algoritmo de división

```
division :: Integer -> Integer -> (Integer, Integer)
division a d | a < d = (0, a)
             | otherwise = (fst (division (a-d) d) + 1,
                            snd (division (a-d) d))
```

¿Se puede no poner dos veces `division (a-d) d`? Sí:

```
division :: Integer -> Integer -> (Integer, Integer)
division a d | a < d = (0, a)
             | otherwise = (fst qr' + 1, snd qr')
              where qr' = division (a-d) d
```

Ejercicio

Extender la función `division :: Integer -> Integer -> (Integer, Integer)` para que funcione para $a \in \mathbb{Z}$, $d > 0$.

Generalización de funciones

¿Una fácil?.. o no tanto

- `sumaDivisores :: Integer -> Integer` que calcule la suma de los divisores un número.

Generalización de funciones

¿Una fácil?.. o no tanto

- `sumaDivisores :: Integer -> Integer` que calcule la suma de los divisores un número.

Pregunta clave: ¿alcanza hacer recursión sobre n?

¿Una fácil?.. o no tanto

- `sumaDivisores :: Integer -> Integer` que calcule la suma de los divisores un número.

Pregunta clave: ¿alcanza hacer recursión sobre n?

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

Generalización de funciones

¿Una fácil?.. o no tanto

- `sumaDivisores :: Integer -> Integer` que calcule la suma de los divisores un número.

Pregunta clave: ¿alcanza hacer recursión sobre *n*?

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

¿Qué sucede si construimos una funcion **más general** que nos facilita el trabajo?

```
sumaDivisoresHasta :: Integer -> Integer -> Integer
```

que devuelve la suma de los divisores de un número hasta cierto punto.

Generalización de funciones

¿Una fácil?.. o no tanto

- `sumaDivisores :: Integer -> Integer` que calcule la suma de los divisores un número.

Pregunta clave: ¿alcanza hacer recursión sobre *n*?

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

¿Qué sucede si construimos una funcion **más general** que nos facilita el trabajo?

```
sumaDivisoresHasta :: Integer -> Integer -> Integer
```

que devuelve la suma de los divisores de un número hasta cierto punto.

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

Generalización de funciones

¿Una fácil?.. o no tanto

- `sumaDivisores :: Integer -> Integer` que calcule la suma de los divisores un número.

Pregunta clave: ¿alcanza hacer recursión sobre *n*?

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

¿Qué sucede si construimos una funcion **más general** que nos facilita el trabajo?

```
sumaDivisoresHasta :: Integer -> Integer -> Integer
```

que devuelve la suma de los divisores de un número hasta cierto punto.

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

Generalización de funciones

¿Una fácil?.. o no tanto

- `sumaDivisores :: Integer -> Integer` que calcule la suma de los divisores un número.

Pregunta clave: ¿alcanza hacer recursión sobre n?

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

¿Qué sucede si construimos una funcion **más general** que nos facilita el trabajo?

```
sumaDivisoresHasta :: Integer -> Integer -> Integer
```

que devuelve la suma de los divisores de un número hasta cierto punto.

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

Ejercicios

- Implementar una función `sumaDivisoresHasta :: Integer -> Integer -> Integer`.
- Implementar la función `sumaDivisores` en función de la anterior.

Ejercicios

Un entero $p > 1$ es **primo** si ningún natural k tal que $1 < k < p$ divide a p.

1. `menorDivisor :: Integer -> Integer` que calcule el menor divisor (mayor que 1) de un natural n.
2. Implementar la función `esPrimo :: Integer -> Bool`.

Ejercicios (sumatorias dobles)

3. Implementar la siguiente función:

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

4. Implementar una función `sumaPotencias n m` que sume todas las potencias de la forma q^a con $1 \leq q \leq n$ y $1 \leq a \leq m$.
5. Implementar una función `sumaRacionales n m` que sume todos los números racionales de la forma p/q con $1 \leq p \leq n$ y $1 \leq q \leq m$.