Piensa en Haskell y en Python

(Ejercicios de programación con Haskell y con Python)

José A. Alonso Jiménez

Grupo de Lógica Computacional
Dpto. de Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, 8 de enero de 2023

Índice general

Introducción

Este libro es una introducción a la programación con Haskell y Python, a través de la resolución de ejercicios publicados diariamente en el blog Exercitium [1]. Estos ejercicios están organizados siguiendo el orden de los Temas de programación funcional con Haskell [2]. Las soluciones a los ejercicios están disponibles en dos repositorios de GitHub, uno con las soluciones en Haskell [3] y otro con las soluciones en Python [4]). Ambos repositorios están estructurados como proyectos utilizando Stack [5] y Poetry [6], respectivamente. Se han escrito soluciones en Python con un estilo funcional similar al de Haskell, y se han comprobado con mypy [7] que los tipos de las definiciones en Python son correctos. Además, se han dado varias soluciones a cada ejercicio, verificando su equivalencia (mediante QuickCheck [8] en Haskell e Hypothesis Hypothesis [9] en Python) y comparando su eficiencia.

El libro actualmente consta de cinco capítulos. Los tres primeros capítulos tratan sobre cómo definir funciones usando composición, comprensión y recursión. El cuarto capítulo introduce las funciones de orden superior y el quinto muestra cómo definir y usar nuevos tipos. Tenga en cuenta que algunas de las soluciones a los ejercicios pueden no corresponder con el contenido del capítulo donde se encuentran. En futuras versiones del libro, se ampliará el contenido hasta completar el curso de Programación funcional con Haskell [10].

[1] https://www.glc.us.es/ jalonso/exercitium
[2] https://jaalonso.github.io/materias/PFconHaskell/temas.html
[3] https://github.com/jaalonso/Exercitium
[4] https://github.com/jaalonso/Exercitium-Python
[5] https://docs.haskellstack.org/en/stable
[6] https://python-poetry.org
[7] http://mypy-lang.org
[8] https://hackage.haskell.org/package/QuickCheck
[9] https://hypothesis.readthedocs.io/en/latest
[10] https://jaalonso.github.io/materias/PFconHaskell

Capítulo 1

Definiciones elementales de funciones

En este capítulo se plantean ejercicios con definiciones elementales (no recursivas) de funciones. Se corresponden con los 4 primeros temas del Curso de programación funcional con Haskell [1].

Contenido

[1] https://jaalonso.github.io/materias/PFconHaskell/temas.html

1.1. Media aritmética de tres números

En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Definir la función
--    media3 :: Float -> Float -> Float -> Float
-- tal que (media3 x y z) es la media aritmética de los números x, y y
-- z. Por ejemplo,
--    media3 1 3 8     ==  4.0
--    media3 (-1) 0 7  ==  2.0
--    media3 (-3) 0 3  ==  0.0
-- ---------------------------------------------------------------------

module Media_aritmetica_de_tres_numeros where

media3 :: Float -> Float -> Float -> Float
media3 x y z = (x+y+z)/3
```

En Python

```python
# ---------------------------------------------------------------------
# Definir la función
#    media3 : (float, float, float) -> float
# tal que (media3 x y z) es la media aritmética de los números x, y y
# z. Por ejemplo,
#    media3(1, 3, 8)   ==  4.0
#    media3(-1, 0, 7)  ==  2.0
#    media3(-3, 0, 3)  ==  0.0
# ---------------------------------------------------------------------

def media3(x: float, y: float, z: float) -> float:
    return (x + y + z)/3
```

1.2. Suma de monedas

En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Definir la función
--    sumaMonedas :: Int -> Int -> Int -> Int -> Int -> Int
-- tal que (sumaMonedas a b c d e) es la suma de los euros
-- correspondientes a a monedas de 1 euro, b de 2 euros, c de 5 euros, d
-- 10 euros y e de 20 euros. Por ejemplo,
--    sumaMonedas 0 0 0 0 1  ==  20
--    sumaMonedas 0 0 8 0 3  == 100
--    sumaMonedas 1 1 1 1 1  ==  38
-- ---------------------------------------------------------------------

module Suma_de_monedas where

sumaMonedas :: Int -> Int -> Int -> Int -> Int -> Int
sumaMonedas a b c d e = 1*a+2*b+5*c+10*d+20*e
```

En Python

```python
# ---------------------------------------------------------------------
# Definir la función
#    sumaMonedas : (int, int, int, int, int) -> int
```

```python
# tal que sumaMonedas(a, b, c, d, e) es la suma de los euros
# correspondientes a a monedas de 1 euro, b de 2 euros, c de 5 euros, d
# 10 euros y e de 20 euros. Por ejemplo,
#    sumaMonedas(0, 0, 0, 0, 1)  ==  20
#    sumaMonedas(0, 0, 8, 0, 3)  == 100
#    sumaMonedas(1, 1, 1, 1, 1)  ==  38
# ---------------------------------------------------------------------

def sumaMonedas(a: int, b: int, c: int, d: int, e: int) -> int:
    return 1 * a + 2 * b + 5 * c + 10 * d + 20 * e
```

1.3. Volumen de la esfera

En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Definir la función
--    volumenEsfera :: Double -> Double
-- tal que (volumenEsfera r) es el volumen de la esfera de radio r. Por
-- ejemplo,
--    volumenEsfera 10  ==  4188.790204786391
-- ---------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-type-defaults #-}

module Volumen_de_la_esfera where

volumenEsfera :: Double -> Double
volumenEsfera r = (4/3)*pi*r^3
```

En Python

```python
# ---------------------------------------------------------------------
# Definir la función
#    volumenEsfera : (float) -> float
# tal que volumenEsfera(r) es el volumen de la esfera de radio r. Por
# ejemplo,
#    volumenEsfera(10)  ==  4188.790204786391
# ---------------------------------------------------------------------
```

```python
from math import pi


def volumenEsfera(r: float) -> float:
    return (4 / 3) * pi * r ** 3
```

1.4. Área de la corona circular

En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Definir la función
--    areaDeCoronaCircular :: Double -> Double -> Double
-- tal que (areaDeCoronaCircular r1 r2) es el área de una corona
-- circular de radio interior r1 y radio exterior r2. Por ejemplo,
--    areaDeCoronaCircular 1 2 == 9.42477796076938
--    areaDeCoronaCircular 2 5 == 65.97344572538566
--    areaDeCoronaCircular 3 5 == 50.26548245743669
-- ---------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-type-defaults #-}

module Area_corona_circular where

areaDeCoronaCircular :: Double -> Double -> Double
areaDeCoronaCircular r1 r2 = pi*(r2^2 -r1^2)
```

En Python

```python
# ---------------------------------------------------------------------
# Definir la función
#    areaDeCoronaCircular : (float, float) -> float
# tal que areaDeCoronaCircular(r1, r2) es el área de una corona
# circular de radio interior r1 y radio exterior r2. Por ejemplo,
#    areaDeCoronaCircular(1, 2) == 9.42477796076938
#    areaDeCoronaCircular(2, 5) == 65.97344572538566
#    areaDeCoronaCircular(3, 5) == 50.26548245743669
# ---------------------------------------------------------------------
```

```python
from math import pi


def areaDeCoronaCircular(r1: float, r2: float) -> float:
    return pi * (r2 ** 2 - r1 ** 2)
```

1.5. Último dígito

En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Definir la función
--    ultimoDigito :: Int -> Int
-- tal que (ultimoDigito x) es el último dígito del número x. Por
-- ejemplo,
--    ultimoDigito 325  ==  5
-- ---------------------------------------------------------------------

module Ultimo_digito where

ultimoDigito :: Int -> Int
ultimoDigito x = rem x 10
```

En Python

```python
# ---------------------------------------------------------------------
# Definir la función
#    ultimoDigito : (int) -> int
# tal que ultimoDigito(x) es el último dígito del número x. Por
# ejemplo,
#    ultimoDigito(325) == 5
# ---------------------------------------------------------------------

def ultimoDigito(x: int) -> int:
    return x % 10
```

1.6. Máximo de tres números

En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Definir la función
--    maxTres :: Int -> Int -> Int -> Int
-- tal que (maxTres x y z) es el máximo de x, y y z. Por ejemplo,
--    maxTres 6 2 4  ==  6
--    maxTres 6 7 4  ==  7
--    maxTres 6 7 9  ==  9
-- ---------------------------------------------------------------------

module Maximo_de_tres_numeros where

maxTres :: Int -> Int -> Int -> Int
maxTres x y z = max x (max y z)
```

En Python

```python
# ---------------------------------------------------------------------
# Definir la función
#    maxTres : (int, int, int) -> int
# tal que maxTres(x, y, z) es el máximo de x, y y z. Por ejemplo,
#    maxTres(6, 2, 4)  ==  6
#    maxTres(6, 7, 4)  ==  7
#    maxTres(6, 7, 9)  ==  9
# ---------------------------------------------------------------------

def maxTres(x: int, y: int, z: int) -> int:
    return max(x, max(y, z))
```

1.7. El primero al final

En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Definir la función
--    rota1 :: [a] -> [a]
```

```haskell
-- tal que (rota1 xs) es la lista obtenida poniendo el primer elemento
-- de xs al final de la lista. Por ejemplo,
--    rota1 [3,2,5,7]  ==  [2,5,7,3]
-- ---------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module El_primero_al_final where

import Test.QuickCheck

-- 1ª solución
-- ===========

rota1a :: [a] -> [a]
rota1a [] = []
rota1a xs = tail xs ++ [head xs]

-- 2ª solución
-- ===========

rota1b :: [a] -> [a]
rota1b []     = []
rota1b (x:xs) = xs ++ [x]

-- Comprobación de equivalencia
-- ============================

-- La propiedad es
prop_rota1 :: [Int] -> Bool
prop_rota1 xs =
  rota1a xs == rota1b xs

-- La comprobación es
--    λ> quickCheck prop_rota1
--    +++ OK, passed 100 tests.
```

En Python

```python
# ---------------------------------------------------------------------
# Definir la función
#    rota1 : (List[A]) -> List[A]
# tal que rota1(xs) es la lista obtenida poniendo el primer elemento de
# xs al final de la lista. Por ejemplo,
#    rota1([3, 2, 5, 7]) == [2, 5, 7, 3]
#    rota1(['a', 'b', 'c']) == ['b', 'c', 'a']
# ---------------------------------------------------------------------

from typing import TypeVar

from hypothesis import given
from hypothesis import strategies as st

A = TypeVar('A')

# 1ª solución
def rota1a(xs: list[A]) -> list[A]:
    if xs == []:
        return []
    return xs[1:] + [xs[0]]

# 2ª solución
def rota1b(xs: list[A]) -> list[A]:
    if xs == []:
        return []
    ys = xs[1:]
    ys.append(xs[0])
    return ys

# 3ª solución
def rota1c(xs: list[A]) -> list[A]:
    if xs == []:
        return []
    y, *ys = xs
    return ys + [y]

# La equivalencia de las definiciones es
@given(st.lists(st.integers()))
```

```
def test_rota1(xs: list[int]) -> None:
    assert rota1a(xs) == rota1b(xs) == rota1c(xs)

# La comprobación es
#    src> poetry run pytest -q el_primero_al_final.py
#    1 passed in 0.20s
```

1.8. Los primeros al final

En Haskell

```
-- ---------------------------------------------------------------------
-- Definir la función
--    rota :: Int -> [a] -> [a]
-- tal que (rota n xs) es la lista obtenida poniendo los n primeros
-- elementos de xs al final de la lista. Por ejemplo,
--    rota 1 [3,2,5,7]  ==  [2,5,7,3]
--    rota 2 [3,2,5,7]  ==  [5,7,3,2]
--    rota 3 [3,2,5,7]  ==  [7,3,2,5]
-- ---------------------------------------------------------------------

module Los_primeros_al_final where

rota :: Int -> [a] -> [a]
rota n xs = drop n xs ++ take n xs
```

En Python

```
# ---------------------------------------------------------------------
# Definir la función
#    rota : (int, List[A]) -> List[A]
# tal que rota(n, xs) es la lista obtenida poniendo los n primeros
# elementos de xs al final de la lista. Por ejemplo,
#    rota(1, [3, 2, 5, 7]) == [2, 5, 7, 3]
#    rota(2, [3, 2, 5, 7]) == [5, 7, 3, 2]
#    rota(3, [3, 2, 5, 7]) == [7, 3, 2, 5]
# ---------------------------------------------------------------------

from typing import TypeVar
```

```python
A = TypeVar('A')

def rota(n: int, xs: list[A]) -> list[A]:
    return xs[n:] + xs[:n]
```

1.9. Rango de una lista

En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Definir la función
--    rango :: [Int] -> [Int]
-- tal que (rango xs) es la lista formada por el menor y mayor elemento
-- de xs.
--    rango [3,2,7,5]  ==  [2,7]
-- ---------------------------------------------------------------------

module Rango_de_una_lista where

rango :: [Int] -> [Int]
rango xs = [minimum xs, maximum xs]
```

En Python

```python
# ---------------------------------------------------------------------
# Definir la función
#    rango : (List[int]) -> List[int]
# tal que rango(xs) es la lista formada por el menor y mayor elemento
# de xs.
#    rango([3, 2, 7, 5]) == [2, 7]
# ---------------------------------------------------------------------

def rango(xs: list[int]) -> list[int]:
    return [min(xs), max(xs)]
```

1.10. Reconocimiento de palindromos

En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Definir la función
--    palindromo :: Eq a => [a] -> Bool
-- tal que (palindromo xs) se verifica si xs es un palíndromo; es decir,
-- es lo mismo leer xs de izquierda a derecha que de derecha a
-- izquierda. Por ejemplo,
--    palindromo [3,2,5,2,3]    ==  True
--    palindromo [3,2,5,6,2,3]  ==  False
-- ---------------------------------------------------------------------

module Reconocimiento_de_palindromos where

palindromo :: Eq a => [a] -> Bool
palindromo xs =
  xs == reverse xs
```

En Python

```python
# ---------------------------------------------------------------------
# Definir la función
#    palindromo : (List[A]) -> bool
# tal que palindromo(xs) se verifica si xs es un palíndromo; es decir,
# es lo mismo leer xs de izquierda a derecha que de derecha a
# izquierda. Por ejemplo,
#    palindromo([3, 2, 5, 2, 3])     ==  True
#    palindromo([3, 2, 5, 6, 2, 3])  ==  False
# ---------------------------------------------------------------------

from typing import TypeVar

A = TypeVar('A')

def palindromo(xs: list[A]) -> bool:
    return xs == list(reversed(xs))
```

1.11. Interior de una lista

En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Definir la función
--    interior :: [a] -> [a]
-- tal que (interior xs) es la lista obtenida eliminando los extremos de
-- la lista xs. Por ejemplo,
--    interior [2,5,3,7,3]  ==  [5,3,7]
--    interior [2..7]       ==  [3,4,5,6]
-- ---------------------------------------------------------------------

module Interior_de_una_lista where

interior :: [a] -> [a]
interior xs = tail (init xs)
```

En Python

```python
# ---------------------------------------------------------------------
# Definir la función
#    interior : (list[A]) -> list[A]
# tal que interior(xs) es la lista obtenida eliminando los extremos de
# la lista xs. Por ejemplo,
#    interior([2, 5, 3, 7, 3])  ==  [5, 3, 7]
# ---------------------------------------------------------------------

from typing import TypeVar

from hypothesis import given
from hypothesis import strategies as st

A = TypeVar('A')

# 1ª solución
def interior1(xs: list[A]) -> list[A]:
    return xs[1:][:-1]

# 2ª solución
```

```python
def interior2(xs: list[A]) -> list[A]:
    return xs[1:-1]

# La propiedad de equivalencia es
@given(st.lists(st.integers()))
def test_interior(xs: list[int]) -> None:
    assert interior1(xs) == interior2(xs)

# La comprobación es
#    src> poetry run pytest -q interior_de_una_lista.py
#    1 passed in 0.21s
```

1.12. Elementos finales

En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Definir la función
--    finales :: Int -> [a] -> [a]
-- tal que (finales n xs) es la lista formada por los n finales
-- elementos de xs. Por ejemplo,
--    finales 3 [2,5,4,7,9,6]  ==  [7,9,6]
-- ---------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module Elementos_finales where
import Test.QuickCheck

-- 1ª definición
finales1 :: Int -> [a] -> [a]
finales1 n xs = drop (length xs - n) xs

-- 2ª definición
finales2 :: Int -> [a] -> [a]
finales2 n xs = reverse (take n (reverse xs))

-- Comprobación de equivalencia
-- ============================
```

```haskell
-- La propiedad es
prop_finales :: Int -> [Int] -> Bool
prop_finales n xs =
  finales1 n xs == finales2 n xs

-- La comprobación es
--    λ> quickCheck prop_finales
--    +++ OK, passed 100 tests.
```

En Python

```python
# ---------------------------------------------------------------------
# Definir la función
#    finales : (int, list[A]) -> list[A]
# tal que finales(n, xs) es la lista formada por los n finales
# elementos de xs. Por ejemplo,
#    finales(3, [2, 5, 4, 7, 9, 6])  ==  [7, 9, 6]
# ---------------------------------------------------------------------

from typing import TypeVar

from hypothesis import given
from hypothesis import strategies as st

A = TypeVar('A')

# 1ª definición
def finales1(n: int, xs: list[A]) -> list[A]:
    if len(xs) <= n:
        return xs
    return xs[len(xs) - n:]

# 2ª definición
def finales2(n: int, xs: list[A]) -> list[A]:
    if n == 0:
        return []
    return xs[-n:]

# 3ª definición
def finales3(n: int, xs: list[A]) -> list[A]:
```

```
    ys = list(reversed(xs))
    return list(reversed(ys[:n]))

# La propiedad de equivalencia es
@given(st.integers(min_value=0), st.lists(st.integers()))
def test_equiv_finales(n: int, xs: list[int]) -> None:
    assert finales1(n, xs) == finales2(n, xs) == finales3(n, xs)

# La comprobación es
#    src> poetry run pytest -q elementos_finales.py
#    1 passed in 0.18s
```

1.13. Segmento de una lista

En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Definir la función
--    segmento :: Int -> Int -> [a] -> [a]
-- tal que (segmento m n xs) es la lista de los elementos de xs
-- comprendidos entre las posiciones m y n. Por ejemplo,
--    segmento 3 4 [3,4,1,2,7,9,0]  ==  [1,2]
--    segmento 3 5 [3,4,1,2,7,9,0]  ==  [1,2,7]
--    segmento 5 3 [3,4,1,2,7,9,0]  ==  []
-- ---------------------------------------------------------------------

module Segmento_de_una_lista where

segmento :: Int -> Int -> [a] -> [a]
segmento m n xs = drop (m-1) (take n xs)
```

En Python

```python
# ---------------------------------------------------------------------
# Definir la función
#    segmento : (int, int, list[A]) -> list[A]
# tal que segmento(m, n, xs) es la lista de los elementos de xs
# comprendidos entre las posiciones m y n. Por ejemplo,
#    segmento(3, 4, [3, 4, 1, 2, 7, 9, 0])  ==  [1, 2]
```

```
#    segmento(3, 5, [3, 4, 1, 2, 7, 9, 0])  ==  [1, 2, 7]
#    segmento(5, 3, [3, 4, 1, 2, 7, 9, 0])  ==  []
# ---------------------------------------------------------------------

from typing import TypeVar

from hypothesis import given
from hypothesis import strategies as st

A = TypeVar('A')

# 1ª definición
def segmento1(m: int, n: int, xs: list[A]) -> list[A]:
    ys = xs[:n]
    if m == 0:
        return ys
    return ys[m - 1:]

# 2ª definición
def segmento2(m: int, n: int, xs: list[A]) -> list[A]:
    if m == 0:
        return xs[:n]
    return xs[m-1:n]

# La propiedad de equivalencia es
@given(st.integers(min_value=0),
       st.integers(min_value=0),
       st.lists(st.integers()))
def test_equiv_segmento(m: int, n: int, xs: list[int]) -> None:
    assert segmento1(m, n, xs) == segmento2(m, n, xs)

# La comprobación es
#    src> poetry run pytest -q segmento_de_una_lista.py
#    1 passed in 0.19s
```

1.14. Primeros y últimos elementos

En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Definir la función
--    extremos :: Int -> [a] -> [a]
-- tal que (extremos n xs) es la lista formada por los n primeros
-- elementos de xs y los n finales elementos de xs. Por ejemplo,
--    extremos 3 [2,6,7,1,2,4,5,8,9,2,3]  ==  [2,6,7,9,2,3]
-- ---------------------------------------------------------------------

module Primeros_y_ultimos_elementos where

extremos :: Int -> [a] -> [a]
extremos n xs = take n xs ++ drop (length xs - n) xs
```

En Python

```python
# ---------------------------------------------------------------------
# Definir la función
#    extremos : (int, list[A]) -> list[A]
# tal que extremos(n, xs) es la lista formada por los n primeros
# elementos de xs y los n finales elementos de xs. Por ejemplo,
#    extremos(3, [2, 6, 7, 1, 2, 4, 5, 8, 9, 2, 3])  ==  [2, 6, 7, 9, 2, 3]
# ---------------------------------------------------------------------

from typing import TypeVar

A = TypeVar('A')

def extremos(n: int, xs: list[A]) -> list[A]:
    return xs[:n] + xs[-n:]
```

1.15. Elemento mediano

En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Definir la función
--    mediano :: Int -> Int -> Int -> Int
-- tal que (mediano x y z) es el número mediano de los tres números x, y
-- y z. Por ejemplo,
--    mediano 3 2 5  ==  3
--    mediano 2 4 5  ==  4
--    mediano 2 6 5  ==  5
--    mediano 2 6 6  ==  6
-- ---------------------------------------------------------------------

module Elemento_mediano where

mediano :: Int -> Int -> Int -> Int
mediano x y z = x + y + z - minimum [x,y,z] - maximum [x,y,z]
```

En Python

```python
# ---------------------------------------------------------------------
# Definir la función
#    mediano : (int, int, int) -> int
# tal que mediano(x, y, z) es el número mediano de los tres números x, y
# y z. Por ejemplo,
#    mediano(3, 2, 5) == 3
#    mediano(2, 4, 5) == 4
#    mediano(2, 6, 5) == 5
#    mediano(2, 6, 6) == 6
# ---------------------------------------------------------------------

def mediano(x: int, y: int, z: int) -> int:
    return x + y + z - min([x, y, z]) - max([x, y, z])
```

1.16. Tres iguales

En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Definir la función
--    tresIguales :: Int -> Int -> Int -> Bool
-- tal que (tresIguales x y z) se verifica si los elementos x, y y z son
-- iguales. Por ejemplo,
--    tresIguales 4 4 4  ==  True
--    tresIguales 4 3 4  ==  False
-- ---------------------------------------------------------------------

module Tres_iguales where

tresIguales :: Int -> Int -> Int -> Bool
tresIguales x y z = x == y && y == z
```

En Python

```python
# ---------------------------------------------------------------------
# Definir la función
#    tresIguales : (int, int, int) -> bool
# tal que tresIguales(x, y, z) se verifica si los elementos x, y y z son
# iguales. Por ejemplo,
#    tresIguales(4, 4, 4)  ==  True
#    tresIguales(4, 3, 4)  ==  False
# ---------------------------------------------------------------------

from hypothesis import given
from hypothesis import strategies as st


# 1ª definición
def tresIguales1(x: int, y: int, z: int) -> bool:
    return x == y and y == z

# 2ª definición
def tresIguales2(x: int, y: int, z: int) -> bool:
    return x == y == z
```

```python
# La propiedad de equivalencia es
@given(st.integers(), st.integers(), st.integers())
def test_equiv_tresIguales(x: int, y: int, z: int) -> None:
    assert tresIguales1(x, y, z) == tresIguales2(x, y, z)

# La comprobación es
#    src> poetry run pytest -q tres_iguales.py
#    1 passed in 0.16s
```

1.17. Tres diferentes

En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Definir la función
--    tresDiferentes :: Int -> Int -> Int -> Bool
-- tal que (tresDiferentes x y z) se verifica si los elementos x, y y z
-- son distintos. Por ejemplo,
--    tresDiferentes 3 5 2  ==  True
--    tresDiferentes 3 5 3  ==  False
-- ---------------------------------------------------------------------

module Tres_diferentes where

tresDiferentes :: Int -> Int -> Int -> Bool
tresDiferentes x y z = x /= y && x /= z && y /= z
```

En Python

```python
# ---------------------------------------------------------------------
# Definir la función
#    tresDiferentes : (int, int, int) -> bool
# tal que tresDiferentes(x, y, z) se verifica si los elementos x, y y z
# son distintos. Por ejemplo,
#    tresDiferentes(3, 5, 2)  ==  True
#    tresDiferentes(3, 5, 3)  ==  False
# ---------------------------------------------------------------------
```

```python
def tresDiferentes(x: int, y: int, z: int) -> bool:
    return x != y and x != z and y != z
```

1.18. División segura

En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Definir la función
--    divisionSegura :: Double -> Double -> Double
-- tal que (divisionSegura x y) es x/y si y no es cero y 9999 en caso
-- contrario. Por ejemplo,
--    divisionSegura 7 2  ==  3.5
--    divisionSegura 7 0  ==  9999.0
-- ---------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module Division_segura where

import Test.QuickCheck

-- 1ª definición
divisionSegura1 :: Double -> Double -> Double
divisionSegura1 x y =
  if y == 0 then 9999 else x/y

-- 2ª definición
divisionSegura2 :: Double -> Double -> Double
divisionSegura2 _ 0 = 9999
divisionSegura2 x y = x/y

-- Comprobación de equivalencia
-- ============================

-- La propiedad es
prop_divisionSegura :: Double -> Double -> Bool
prop_divisionSegura x y =
  divisionSegura1 x y == divisionSegura2 x y
```

```
-- La comprobación es
--    λ> quickCheck prop_divisionSegura
--    +++ OK, passed 100 tests.
```

En Python

```python
# ---------------------------------------------------------------------
# Definir la función
#    divisionSegura : (float, float) -> float
# tal que divisionSegura(x, y) es x/y si y no es cero y 9999 en caso
# contrario. Por ejemplo,
#    divisionSegura(7, 2)  ==  3.5
#    divisionSegura(7, 0)  ==  9999.0
# ---------------------------------------------------------------------

from hypothesis import given
from hypothesis import strategies as st


# 1ª definición
def divisionSegura1(x: float, y: float) -> float:
    if y == 0:
        return 9999.0
    return x/y

# 2ª definición
def divisionSegura2(x: float, y: float) -> float:
    match y:
        case 0:
            return 9999.0
        case _:
            return x/y

# La propiedad de equivalencia es
@given(st.floats(allow_nan=False, allow_infinity=False),
       st.floats(allow_nan=False, allow_infinity=False))
def test_equiv_divisionSegura(x: float, y: float) -> None:
    assert divisionSegura1(x, y) == divisionSegura2(x, y)

# La comprobación es
```

```
#    src> poetry run pytest -q division_segura.py
#    1 passed in 0.37s
```

1.19. Disyunción excluyente

En Haskell

```
-- ---------------------------------------------------------------------
-- La disyunción excluyente de dos fórmulas se verifica si una es
-- verdadera y la otra es falsa. Su tabla de verdad es
--    x     | y     | xor x y
--    ------+-------+---------
--    True  | True  | False
--    True  | False | True
--    False | True  | True
--    False | False | False
--
-- Definir la función
--    xor :: Bool -> Bool -> Bool
-- tal que (xor x y) es la disyunción excluyente de x e y. Por ejemplo,
--    xor True  True  == False
--    xor True  False == True
--    xor False True  == True
--    xor False False == False
-- ---------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module Disyuncion_excluyente where

import Test.QuickCheck

-- 1ª solución
xor1 :: Bool -> Bool -> Bool
xor1 True  True  = False
xor1 True  False = True
xor1 False True  = True
xor1 False False = False

-- 2ª solución
```

```haskell
xor2 :: Bool -> Bool -> Bool
xor2 True  y = not y
xor2 False y = y

-- 3ª solución:
xor3 :: Bool -> Bool -> Bool
xor3 x y = (x || y) && not (x && y)

-- 4ª solución:
xor4 :: Bool -> Bool -> Bool
xor4 x y = (x && not y) || (y && not x)

-- 5ª solución:
xor5 :: Bool -> Bool -> Bool
xor5 x y = x /= y

-- Comprobación de equivalencia
-- ============================

-- La propiedad es
prop_xor :: Bool -> Bool -> Bool
prop_xor x y =
  all (== xor1 x y)
      [xor2 x y,
       xor3 x y,
       xor4 x y,
       xor5 x y]

-- La comprobación es
--    λ> quickCheck prop_xor
--    +++ OK, passed 100 tests.
```

En Python

```python
# ---------------------------------------------------------------------
# La disyunción excluyente de dos fórmulas se verifica si una es
# verdadera y la otra es falsa. Su tabla de verdad es
#    x     | y     | xor x y
#    ------+-------+---------
#    True  | True  | False
```

```
#    True  | False | True
#    False | True  | True
#    False | False | False
#
# Definir la función
#    xor : (bool, bool) -> bool
# tal que xor(x, y) es la disyunción excluyente de x e y. Por ejemplo,
#    xor(True, True) == False
#    xor(True, False) == True
#    xor(False, True) == True
#    xor(False, False) == False
# ---------------------------------------------------------------------

from typing import Any

from hypothesis import given
from hypothesis import strategies as st


# 1ª solución
def xor1(x: bool, y: bool) -> Any:
    match x, y:
        case True,  True: return False
        case True,  False: return True
        case False, True: return True
        case False, False: return False

# 2ª solución
def xor2(x: bool, y: bool) -> bool:
    if x:
        return not y
    return y

# 3ª solución
def xor3(x: bool, y: bool) -> bool:
    return (x or y) and not(x and y)

# 4ª solución
def xor4(x: bool, y: bool) -> bool:
    return (x and not y) or (y and not x)
```

```python
# 5ª solución
def xor5(x: bool, y: bool) -> bool:
    return x != y

# La propiedad de equivalencia es
@given(st.booleans(), st.booleans())
def test_equiv_xor(x: bool, y: bool) -> None:
    assert xor1(x, y) == xor2(x, y) == xor3(x, y) == xor4(x, y) == xor5(x, y)

# La comprobación es
#    src> poetry run pytest -q disyuncion_excluyente.py
#    1 passed in 0.11s
```

1.20. Mayor rectángulo

En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Las dimensiones de los rectángulos puede representarse por pares; por
-- ejemplo, (5,3) representa a un rectángulo de base 5 y altura 3.
--
-- Definir la función
--    mayorRectangulo :: (Num a, Ord a) => (a,a) -> (a,a) -> (a,a)
-- tal que (mayorRectangulo r1 r2) es el rectángulo de mayor área entre
-- r1 y r2. Por ejemplo,
--    mayorRectangulo (4,6) (3,7)  ==  (4,6)
--    mayorRectangulo (4,6) (3,8)  ==  (4,6)
--    mayorRectangulo (4,6) (3,9)  ==  (3,9)
-- ---------------------------------------------------------------------

module Mayor_rectangulo where

mayorRectangulo :: (Num a, Ord a) => (a,a) -> (a,a) -> (a,a)
mayorRectangulo (a,b) (c,d)
  | a*b >= c*d = (a,b)
  | otherwise  = (c,d)
```

En Python

```python
# ---------------------------------------------------------------------
# Las dimensiones de los rectángulos puede representarse por pares; por
# ejemplo, (5,3) representa a un rectángulo de base 5 y altura 3.
#
# Definir la función
#    mayorRectangulo : (tuple[float, float], tuple[float, float])
#                      -> tuple[float, float]
# tal que mayorRectangulo(r1, r2) es el rectángulo de mayor área entre
# r1 y r2. Por ejemplo,
#    mayorRectangulo((4, 6), (3, 7))  ==  (4, 6)
#    mayorRectangulo((4, 6), (3, 8))  ==  (4, 6)
#    mayorRectangulo((4, 6), (3, 9))  ==  (3, 9)
# ---------------------------------------------------------------------

def mayorRectangulo(r1: tuple[float, float],
                    r2: tuple[float, float]) -> tuple[float, float]:
    (a, b) = r1
    (c, d) = r2
    if a*b >= c*d:
        return (a, b)
    return (c, d)
```

1.21. Intercambio de componentes de un par

En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Definir la función
--    intercambia :: (a,b) -> (b,a)
-- tal que (intercambia p)  es el punto obtenido intercambiando las
-- coordenadas del punto p. Por ejemplo,
--    intercambia (2,5)  ==  (5,2)
--    intercambia (5,2)  ==  (2,5)
--
-- Comprobar con QuickCheck que la función intercambia es
-- idempotente; es decir, si se aplica dos veces es lo mismo que no
-- aplicarla ninguna.
-- ---------------------------------------------------------------------
```

```haskell
{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module Intercambio_de_componentes_de_un_par where

import Test.QuickCheck

intercambia :: (a,b) -> (b,a)
intercambia (x,y) = (y,x)

-- La propiedad es
prop_intercambia :: (Int,Int) -> Bool
prop_intercambia p = intercambia (intercambia p) == p

-- La comprobación es
--    λ> quickCheck prop_intercambia
--    +++ OK, passed 100 tests.
```

En Python

```python
# ---------------------------------------------------------------------
# Definir la función
#    intercambia : (tuple[A, B]) -> tuple[B, A]
# tal que intercambia(p)  es el punto obtenido intercambiando las
# coordenadas del punto p. Por ejemplo,
#    intercambia((2,5))  ==  (5,2)
#    intercambia((5,2))  ==  (2,5)
#
# Comprobar con Hypothesis que la función intercambia es
# idempotente; es decir, si se aplica dos veces es lo mismo que no
# aplicarla ninguna.
# ---------------------------------------------------------------------

from typing import TypeVar

from hypothesis import given
from hypothesis import strategies as st

A = TypeVar('A')
B = TypeVar('B')
```

```
def intercambia(p: tuple[A, B]) -> tuple[B, A]:
    (x, y) = p
    return (y, x)

# La propiedad de es
@given(st.tuples(st.integers(), st.integers()))
def test_equiv_intercambia(p: tuple[int, int]) -> None:
    assert intercambia(intercambia(p)) == p

# La comprobación es
#    src> poetry run pytest -q intercambio_de_componentes_de_un_par.py
#    1 passed in 0.15s
```

1.22. Distancia entre dos puntos

En Haskell

```
-- ---------------------------------------------------------------------
-- Definir la función
--    distancia :: (Double,Double) -> (Double,Double) -> Double
-- tal que (distancia p1 p2) es la distancia entre los puntos p1 y
-- p2. Por ejemplo,
--    distancia (1,2) (4,6)  ==  5.0

-- Comprobar con QuickCheck que se verifica la propiedad triangular de
-- la distancia; es decir, dados tres puntos p1, p2 y p3, la distancia
-- de p1 a p3 es menor o igual que la suma de la distancia de p1 a p2 y
-- la de p2 a p3.
-- ---------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-unused-imports #-}
{-# OPTIONS_GHC -fno-warn-type-defaults #-}

module Distancia_entre_dos_puntos where

import Test.QuickCheck

distancia :: (Double,Double) -> (Double,Double) -> Double
distancia (x1,y1) (x2,y2) = sqrt((x1-x2)^2+(y1-y2)^2)
```

```haskell
-- La propiedad es
prop_triangular :: (Double,Double) -> (Double,Double) -> (Double,Double)
                -> Property
prop_triangular p1 p2 p3 =
    all acotado [p1, p2, p3] ==>
    distancia p1 p3 <= distancia p1 p2 + distancia p2 p3
    where acotado (x, y) = abs x < cota && abs y < cota
          cota = 2^30

-- La comprobación es
--    ghci> quickCheck prop_triangular
--    +++ OK, passed 100 tests.

-- Nota: Por problemas de redondeo, la propiedad no se cumple en
-- general. Por ejemplo,
--    λ> p1 = (0, 9147936743096483)
--    λ> p2 = (0, 3)
--    λ> p3 = (0, 2)
--    λ> distancia p1 p3 <= distancia p1 p2 + distancia p2 p3
--    False
--    λ> distancia p1 p3
--    9.147936743096482e15
--    λ> distancia p1 p2 + distancia p2 p3
--    9.14793674309648e15
```

En Python

```python
# ---------------------------------------------------------------------
# Definir la función
#    distancia : (tuple[float, float], tuple[float, float]) -> float
# tal que distancia(p1, p2) es la distancia entre los puntos p1 y
# p2. Por ejemplo,
#    distancia((1, 2), (4, 6)) == 5.0
#
# Comprobar con Hypothesis que se verifica la propiedad triangular de
# la distancia; es decir, dados tres puntos p1, p2 y p3, la distancia
# de p1 a p3 es menor o igual que la suma de la distancia de p1 a p2 y
# la de p2 a p3.
# ---------------------------------------------------------------------
```

```python
from math import sqrt

from hypothesis import given
from hypothesis import strategies as st


def distancia(p1: tuple[float, float],
              p2: tuple[float, float]) -> float:
    (x1, y1) = p1
    (x2, y2) = p2
    return sqrt((x1-x2)**2+(y1-y2)**2)

# La propiedad es
cota = 2 ** 30

@given(st.tuples(st.integers(min_value=0, max_value=cota),
                 st.integers(min_value=0, max_value=cota)),
       st.tuples(st.integers(min_value=0, max_value=cota),
                 st.integers(min_value=0, max_value=cota)),
       st.tuples(st.integers(min_value=0, max_value=cota),
                 st.integers(min_value=0, max_value=cota)))
def test_triangular(p1: tuple[int, int],
                    p2: tuple[int, int],
                    p3: tuple[int, int]) -> None:
    assert distancia(p1, p3) <= distancia(p1, p2) + distancia(p2, p3)

# La comprobación es
#    src> poetry run pytest -q distancia_entre_dos_puntos.py
#    1 passed in 0.38s

# Nota: Por problemas de redondeo, la propiedad no se cumple en
# general. Por ejemplo,
#    λ> p1 = (0, 9147936743096483)
#    λ> p2 = (0, 3)
#    λ> p3 = (0, 2)
#    λ> distancia(p1, p3) <= distancia(p1, p2) + distancia (p2. p3)
#    False
#    λ> distancia(p1, p3)
#    9147936743096482.0
```

```
#    λ> distancia(p1, p2) + distancia(p2, p3)
#    9147936743096480.05
```

1.23. Permutación cíclica

En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Definir una función
--    ciclo :: [a] -> [a]
-- tal que (ciclo xs) es la lista obtenida permutando cíclicamente los
-- elementos de la lista xs, pasando el último elemento al principio de
-- la lista. Por ejemplo,
--    ciclo [2,5,7,9]  == [9,2,5,7]
--    ciclo []         == []
--    ciclo [2]        == [2]
--
-- Comprobar que la longitud es un invariante de la función ciclo; es
-- decir, la longitud de (ciclo xs) es la misma que la de xs.
-- ---------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module Permutacion_ciclica where

import Test.QuickCheck

ciclo :: [a] -> [a]
ciclo [] = []
ciclo xs = last xs : init xs

-- La propiedad es
prop_ciclo :: [Int] -> Bool
prop_ciclo xs = length (ciclo xs) == length xs

-- La comprobación es
--    λ> quickCheck prop_ciclo
--    +++ OK, passed 100 tests.
```

En Python

```
# ---------------------------------------------------------------------
# Definir una función
#    ciclo : (list[A]) -> list[A]
# tal que ciclo(xs) es la lista obtenida permutando cíclicamente los
# elementos de la lista xs, pasando el último elemento al principio de
# la lista. Por ejemplo,
#    ciclo([2, 5, 7, 9])  == [9, 2, 5, 7]
#    ciclo([])            == []
#    ciclo([2])           == [2]
#
# Comprobar que la longitud es un invariante de la función ciclo; es
# decir, la longitud de (ciclo xs) es la misma que la de xs.
# ---------------------------------------------------------------------

from typing import TypeVar

from hypothesis import given
from hypothesis import strategies as st

A = TypeVar('A')

def ciclo(xs: list[A]) -> list[A]:
    if xs:
        return [xs[-1]] + xs[:-1]
    return []

# La propiedad de es
@given(st.lists(st.integers()))
def test_equiv_ciclo(xs: list[int]) -> None:
    assert len(ciclo(xs)) == len(xs)

# La comprobación es
#    src> poetry run pytest -q permutacion_ciclica.py
#    1 passed in 0.39s
```

1.24. Mayor número con dos dígitos dados

En Haskell

```
-- ---------------------------------------------------------------------
-- Definir la función
--    numeroMayor :: Int -> Int -> Int
-- tal que (numeroMayor x y) es el mayor número de dos cifras que puede
-- construirse con los dígitos x e y. Por ejemplo,
--    numeroMayor 2 5 ==  52
--    numeroMayor 5 2 ==  52
-- ---------------------------------------------------------------------

module Mayor_numero_con_dos_digitos_dados where

-- 1ª definición:
numeroMayor1 :: Int -> Int -> Int
numeroMayor1 x y = 10 * max x y + min x y

-- 2ª definición:
numeroMayor2 :: Int -> Int -> Int
numeroMayor2 x y | x > y     = 10*x+y
                 | otherwise = 10*y+x

-- Comprobación de equivalencia
-- ============================

-- La propiedad es
prop_numeroMayor :: Bool
prop_numeroMayor =
  and [numeroMayor1 x y == numeroMayor2 x y | x <- [0..9], y <- [0..9]]

-- La comprobación es
--    λ> prop_numeroMayor
--    True
```

En Python

```
# ---------------------------------------------------------------------
# Definir la función
```

```
#    numeroMayor : (int, int) -> int
# tal que numeroMayor(x, y) es el mayor número de dos cifras que puede
# construirse con los dígitos x e y. Por ejemplo,
#    numeroMayor(2, 5) == 52
#    numeroMayor(5, 2) == 52
# ---------------------------------------------------------------------

# 1ª definición
def numeroMayor1(x: int, y: int) -> int:
    return 10 * max(x, y) + min(x, y)

# 2ª definición
def numeroMayor2(x: int, y: int) -> int:
    if x > y:
        return 10 * x + y
    return 10 * y + x

# La propiedad de equivalencia de las definiciones es
def test_equiv_numeroMayor():
    # type: () -> bool
    return all(numeroMayor1(x, y) == numeroMayor2(x, y)
               for x in range(10) for y in range(10))

# La comprobación es
#    >>> test_equiv_numeroMayor()
#    True
```

1.25. Número de raíces de la ecuación de segundo grado

En Haskell

```
-- ---------------------------------------------------------------------
-- Definir la función
--    numeroDeRaices :: (Num t, Ord t) => t -> t -> t -> Int
-- tal que (numeroDeRaices a b c) es el número de raíces reales de la
-- ecuación a*x^2 + b*x + c = 0. Por ejemplo,
--    numeroDeRaices 2 0 3    ==  0
--    numeroDeRaices 4 4 1    ==  1
```

```haskell
--    numeroDeRaices 5 23 12  ==  2
-- ---------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-type-defaults #-}

module Numero_de_raices_de_la_ecuacion_de_segundo_grado where

numeroDeRaices :: (Num t, Ord t) => t -> t -> t -> Int
numeroDeRaices a b c | d < 0     = 0
                     | d == 0    = 1
                     | otherwise = 2
  where d = b^2-4*a*c
```

En Python

```python
# ---------------------------------------------------------------------
# Definir la función
#    numeroDeRaices : (float, float, float) -> float
# tal que numeroDeRaices(a, b, c) es el número de raíces reales de la
# ecuación a*x^2 + b*x + c = 0. Por ejemplo,
#    numeroDeRaices(2, 0, 3)    ==  0
#    numeroDeRaices(4, 4, 1)    ==  1
#    numeroDeRaices(5, 23, 12)  ==  2
# ---------------------------------------------------------------------

def numeroDeRaices(a: float, b: float, c: float) -> float:
    d = b**2-4*a*c
    if d < 0:
        return 0
    if d == 0:
        return 1
    return 2
```

1.26. Raíces de la ecuación de segundo grado

En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Definir la función
```

```
--    raices :: Double -> Double -> Double -> [Double]
-- tal que (raices a b c) es la lista de las raíces reales de la
-- ecuación ax^2 + bx + c = 0. Por ejemplo,
--    raices 1 3 2    ==  [-1.0,-2.0]
--    raices 1 (-2) 1 ==  [1.0,1.0]
--    raices 1 0 1    ==  []
--
-- Comprobar con QuickCheck que la suma de las raíces de la ecuación
-- ax^2 + bx + c = 0 (con a no nulo) es -b/a y su  producto es c/a.
-- ---------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-type-defaults #-}
{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module Raices_de_la_ecuacion_de_segundo_grado where

import Test.QuickCheck

raices :: Double -> Double -> Double -> [Double]
raices a b c
    | d >= 0    = [(-b+e)/t,(-b-e)/t]
    | otherwise = []
    where d = b^2 - 4*a*c
          e = sqrt d
          t = 2*a

-- Para comprobar la propiedad se usará el operador
--    (~=) :: (Fractional a, Ord a) => a -> a -> Bool
-- tal que (x ~= y) se verifica si x e y son casi iguales; es decir si
-- el valor absoluto de su diferencia es menor que una milésima. Por
-- ejemplo,
--    12.3457 ~= 12.3459  ==  True
--    12.3457 ~= 12.3479  ==  False
(~=) :: (Fractional a, Ord a) => a -> a -> Bool
x ~= y  = abs (x-y) < 0.001

-- La propiedad es
prop_raices :: Double -> Double -> Double -> Property
prop_raices a b c =
    a /= 0 && not (null xs) ==> sum xs ~= (-b/a) && product xs ~= (c/a)
```

```haskell
    where xs = raices a b c

-- La comprobación es
--    λ> quickCheck prop_raices
--    +++ OK, passed 100 tests.
```

En Python

```python
# ---------------------------------------------------------------------
# Definir la función
#    raices : (float, float, float) -> list[float]
# tal que raices(a, b, c) es la lista de las raíces reales de la
# ecuación ax^2 + bx + c = 0. Por ejemplo,
#    raices(1, 3, 2)    ==  [-1.0,-2.0]
#    raices(1, (-2), 1) ==  [1.0,1.0]
#    raices(1, 0, 1)    ==  []
#
# Comprobar con Hypothesis que la suma de las raíces de la ecuación
# ax^2 + bx + c = 0 (con a no nulo) es -b/a y su  producto es c/a.
# ---------------------------------------------------------------------

from math import sqrt

from hypothesis import assume, given
from hypothesis import strategies as st


def raices(a: float, b: float, c: float) -> list[float]:
    d = b**2 - 4*a*c
    if d >= 0:
        e = sqrt(d)
        t = 2*a
        return [(-b+e)/t, (-b-e)/t]
    return []

# Para comprobar la propiedad se usará la función
#    casiIguales : (float, float) -> bool
# tal que casiIguales(x, y) se verifica si x e y son casi iguales; es
# decir si el valor absoluto de su diferencia es menor que una
# milésima. Por  ejemplo,
```

```python
#    casiIguales(12.3457, 12.3459)  ==  True
#    casiIguales(12.3457, 12.3479)  ==  False
def casiIguales(x: float, y: float) -> bool:
    return abs(x - y) < 0.001

# La propiedad es
@given(st.floats(min_value=-100, max_value=100),
       st.floats(min_value=-100, max_value=100),
       st.floats(min_value=-100, max_value=100))
def test_prop_raices(a: float, b: float, c: float) -> None:
    assume(abs(a) > 0.1)
    xs = raices(a, b, c)
    assume(xs)
    [x1, x2] = xs
    assert casiIguales(x1 + x2, -b / a)
    assert casiIguales(x1 * x2, c / a)

# La comprobación es
#    src> poetry run pytest -q raices_de_la_ecuacion_de_segundo_grado.py
#    1 passed in 0.35s
```

1.27. Fórmula de Herón para el área de un triángulo

En Haskell

```haskell
-- ---------------------------------------------------------------------
-- La fórmula de Herón, descubierta por Herón de Alejandría, dice que el
-- área de un triángulo cuyo lados miden a, b y c es la raíz cuadrada de
-- s(s-a)(s-b)(s-c) donde s es el semiperímetro
--    s = (a+b+c)/2
--
-- Definir la función
--    area :: Double -> Double -> Double -> Double
-- tal que (area a b c) es el área del triángulo de lados a, b y c. Por
-- ejemplo,
--    area 3 4 5  ==  6.0
-- ---------------------------------------------------------------------
```

```haskell
module Formula_de_Heron_para_el_area_de_un_triangulo where

area :: Double -> Double -> Double -> Double
area a b c = sqrt (s*(s-a)*(s-b)*(s-c))
  where s = (a+b+c)/2
```

En Python

```python
# ---------------------------------------------------------------------
# La fórmula de Herón, descubierta por Herón de Alejandría, dice que el
# área de un triángulo cuyo lados miden a, b y c es la raíz cuadrada de
# s(s-a)(s-b)(s-c) donde s es el semiperímetro
#    s = (a+b+c)/2
#
# Definir la función
#    area : (float, float, float) -> float
# tal que area(a, b, c) es el área del triángulo de lados a, b y c. Por
# ejemplo,
#    area(3, 4, 5) == 6.0
# ---------------------------------------------------------------------

from math import sqrt


def area(a: float, b: float, c: float) -> float:
    s = (a+b+c)/2
    return sqrt(s*(s-a)*(s-b)*(s-c))
```

1.28. Intersección de intervalos cerrados

En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Los intervalos cerrados se pueden representar mediante una lista de
-- dos números (el primero es el extremo inferior del intervalo y el
-- segundo el superior).
--
-- Definir la función
--    interseccion :: Ord a => [a] -> [a] -> [a]
```

```
-- tal que (interseccion i1 i2) es la intersección de los intervalos i1 e
-- i2. Por ejemplo,
--    interseccion [] [3,5]     ==  []
--    interseccion [3,5] []     ==  []
--    interseccion [2,4] [6,9]  ==  []
--    interseccion [2,6] [6,9]  ==  [6,6]
--    interseccion [2,6] [0,9]  ==  [2,6]
--    interseccion [2,6] [0,4]  ==  [2,4]
--    interseccion [4,6] [0,4]  ==  [4,4]
--    interseccion [5,6] [0,4]  ==  []
--
-- Comprobar con QuickCheck que la intersección de intervalos es
-- conmutativa.
-- ---------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-incomplete-patterns #-}

module Interseccion_de_intervalos_cerrados where

import Test.QuickCheck

interseccion :: Ord a => [a] -> [a] -> [a]
interseccion [] _ = []
interseccion _ [] = []
interseccion [a1,b1] [a2,b2]
    | a <= b    = [a,b]
    | otherwise = []
    where a = max a1 a2
          b = min b1 b2

-- La propiedad es
prop_interseccion :: Int -> Int -> Int -> Int -> Property
prop_interseccion a1 b1 a2 b2 =
  a1 <= b1 && a2 <= b2 ==>
  interseccion [a1,b1] [a2,b2] == interseccion [a2,b2] [a1,b1]

-- La comprobación es
--    λ> quickCheck prop_interseccion
--    +++ OK, passed 100 tests; 263 discarded.
```

En Python

```
# ---------------------------------------------------------------------
# Los intervalos cerrados se pueden representar mediante una lista de
# dos números (el primero es el extremo inferior del intervalo y el
# segundo el superior).
#
# Definir la función
#    interseccion : (list[float], list[float]) -> list[float]
# tal que interseccion(i1, i2) es la intersección de los intervalos i1 e
# i2. Por ejemplo,
#    interseccion([],     [3, 5]   ==  []
#    interseccion([3, 5], [])      ==  []
#    interseccion([2, 4], [6, 9])  ==  []
#    interseccion([2, 6], [6, 9])  ==  [6, 6]
#    interseccion([2, 6], [0, 9])  ==  [2, 6]
#    interseccion([2, 6], [0, 4])  ==  [2, 4]
#    interseccion([4, 6], [0, 4])  ==  [4, 4]
#    interseccion([5, 6], [0, 4])  ==  []
#
# Comprobar con Hypothesis que la intersección de intervalos es
# conmutativa.
# ---------------------------------------------------------------------

from hypothesis import assume, given
from hypothesis import strategies as st

Rectangulo = list[float]

def interseccion(i1: Rectangulo,
                 i2: Rectangulo) -> Rectangulo:
    if i1 and i2:
        [a1, b1] = i1
        [a2, b2] = i2
        a = max(a1, a2)
        b = min(b1, b2)
        if a <= b:
            return [a, b]
        return []
    return []
```

```
# La propiedad es
@given(st.floats(), st.floats(), st.floats(), st.floats())
def test_prop_raices(a1: float, b1: float, a2: float, b2: float) -> None:
    assume(a1 <= b1 and a2 <= b2)
    assert interseccion([a1, b1], [a2, b2]) == interseccion([a2, b2], [a1, b1])

# La comprobación es
#    src> poetry run pytest -q interseccion_de_intervalos_cerrados.py
#    1 passed in 0.64s
```

1.29. Números racionales

En Haskell

```
-- ---------------------------------------------------------------------
-- Los números racionales pueden representarse mediante pares de números
-- enteros. Por ejemplo, el número 2/5 puede representarse mediante el
-- par (2,5).
--
-- Definir las funciones
--    formaReducida    :: (Int,Int) -> (Int,Int)
--    sumaRacional     :: (Int,Int) -> (Int,Int) -> (Int,Int)
--    productoRacional :: (Int,Int) -> (Int,Int) -> (Int,Int)
--    igualdadRacional :: (Int,Int) -> (Int,Int) -> Bool
-- tales que
-- + (formaReducida x) es la forma reducida del número racional x. Por
--   ejemplo,
--      formaReducida (4,10)  ==  (2,5)
--      formaReducida (0,5)   ==  (0,1)
-- + (sumaRacional x y) es la suma de los números racionales x e y,
--   expresada en forma reducida. Por ejemplo,
--      sumaRacional (2,3) (5,6)  ==  (3,2)
--      sumaRacional (3,5) (-3,5) ==  (0,1)
-- + (productoRacional x y) es el producto de los números racionales x e
--   y, expresada en forma reducida. Por ejemplo,
--      productoRacional (2,3) (5,6)  ==  (5,9)
-- + (igualdadRacional x y) se verifica si los números racionales x e y
--   son iguales. Por ejemplo,
--      igualdadRacional (6,9) (10,15)  ==  True
--      igualdadRacional (6,9) (11,15)  ==  False
```

```
--      igualdadRacional (0,2) (0,-5)   ==  True
--
-- Comprobar con QuickCheck la propiedad distributiva del producto
-- racional respecto de la suma.
-- ---------------------------------------------------------------------

module Numeros_racionales where

import Test.QuickCheck

formaReducida :: (Int,Int) -> (Int,Int)
formaReducida (0,_) = (0,1)
formaReducida (a,b) = (a `div` c, b  `div` c)
    where c = gcd a b

sumaRacional :: (Int,Int) -> (Int,Int) -> (Int,Int)
sumaRacional (a,b) (c,d) = formaReducida (a*d+b*c, b*d)

productoRacional :: (Int,Int) -> (Int,Int) -> (Int,Int)
productoRacional (a,b) (c,d) = formaReducida (a*c, b*d)

igualdadRacional :: (Int,Int) -> (Int,Int) -> Bool
igualdadRacional (a,b) (c,d) =
    a*d == b*c

-- La propiedad es
prop_distributiva :: (Int,Int) -> (Int,Int) -> (Int,Int) -> Property
prop_distributiva x y z =
  snd x /= 0 && snd y /= 0 && snd z /= 0 ==>
  igualdadRacional (productoRacional x (sumaRacional y z))
                   (sumaRacional (productoRacional x y)
                                 (productoRacional x z))

-- La comprobación es
--    λ> quickCheck prop_distributiva
--    +++ OK, passed 100 tests; 21 discarded.
```

En Python

```python
# ---------------------------------------------------------------------
# Los números racionales pueden representarse mediante pares de números
# enteros. Por ejemplo, el número 2/5 puede representarse mediante el
# par (2,5).
#
# El tipo de los racionales se define por
#    Racional = tuple[int, int]
# Definir las funciones
#    formaReducida    : (Racional) -> Racional
#    sumaRacional     : (Racional, Racional) -> Racional
#    productoRacional : (Racional, Racional) -> Racional
#    igualdadRacional : (Racional, Racional) -> bool
# tales que
# + formaReducida(x) es la forma reducida del número racional x. Por
#   ejemplo,
#      formaReducida((4, 10))  ==  (2, 5)
#      formaReducida((0, 5))   ==  (0, 1)
# + sumaRacional(x, y) es la suma de los números racionales x e y,
#   expresada en forma reducida. Por ejemplo,
#      sumaRacional((2, 3), (5, 6))  ==  (3, 2)
#      sumaRacional((3, 5), (-3, 5)) ==  (0, 1)
# + productoRacional(x, y) es el producto de los números racionales x e
#   y, expresada en forma reducida. Por ejemplo,
#      productoRacional((2, 3), (5, 6))  ==  (5, 9)
# + igualdadRacional(x, y) se verifica si los números racionales x e y
#   son iguales. Por ejemplo,
#      igualdadRacional((6, 9), (10, 15))  ==  True
#      igualdadRacional((6, 9), (11, 15))  ==  False
#      igualdadRacional((0, 2), (0, -5))   ==  True
#
# Comprobar con Hypothesis la propiedad distributiva del producto
# racional respecto de la suma.
# ---------------------------------------------------------------------

from math import gcd

from hypothesis import assume, given
from hypothesis import strategies as st
```

```
Racional = tuple[int, int]

def formaReducida(x: Racional) -> Racional:
    (a, b) = x
    if a == 0:
        return (0, 1)
    c = gcd(a, b)
    return (a // c, b // c)

def sumaRacional(x: Racional,
                 y: Racional) -> Racional:
    (a, b) = x
    (c, d) = y
    return formaReducida((a*d+b*c, b*d))

def productoRacional(x: Racional,
                     y: Racional) -> Racional:
    (a, b) = x
    (c, d) = y
    return formaReducida((a*c, b*d))

def igualdadRacional(x: Racional,
                     y: Racional) -> bool:
    (a, b) = x
    (c, d) = y
    return a*d == b*c

# La propiedad es
@given(st.tuples(st.integers(), st.integers()),
       st.tuples(st.integers(), st.integers()),
       st.tuples(st.integers(), st.integers()))
def test_prop_distributiva(x: tuple[int, int],
                           y: tuple[int, int],
                           z: tuple[int, int]) -> None:
    (_, x2) = x
    (_, y2) = y
    (_, z2) = z
    assume(x2 != 0 and y2 != 0 and z2 != 0)
    assert igualdadRacional(productoRacional(x, sumaRacional(y, z)),
                            sumaRacional(productoRacional(x, y),
```

```
                                         productoRacional(x, z)))

# La comprobación es
#    src> poetry run pytest -q numeros_racionales.py
#    1 passed in 0.37s
```

Capítulo 2

Definiciones por comprensión

En este capítulo se presentan ejercicios con definiciones por comprensión. Se corresponden con el tema 5 del curso de programación funcional con Haskell [1].

Contenido

[1] https://jaalonso.github.io/materias/PFconHaskell/temas/tema-5.html

2.1. Reconocimiento de subconjunto

En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Definir la función
--    subconjunto :: Ord a => [a] -> [a] -> Bool
-- tal que (subconjunto xs ys) se verifica si xs es un subconjunto de
-- ys. por ejemplo,
--    subconjunto [3,2,3] [2,5,3,5]  ==  True
--    subconjunto [3,2,3] [2,5,6,5]  ==  False
-- ---------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module Reconocimiento_de_subconjunto where

import Data.List (nub, sort)
import Data.Set (fromList, isSubsetOf)
import Test.QuickCheck

-- 1ª solución
```

```
subconjunto1 :: Ord a => [a] -> [a] -> Bool
subconjunto1 xs ys =
  [x | x <- xs, x `elem` ys] == xs

-- 2ª solución
subconjunto2 :: Ord a => [a] -> [a] -> Bool
subconjunto2 []     _  = True
subconjunto2 (x:xs) ys = x `elem` ys && subconjunto2 xs ys

-- 3ª solución
subconjunto3 :: Ord a => [a] -> [a] -> Bool
subconjunto3 xs ys =
  all (`elem` ys) xs

-- 4ª solución
subconjunto4 :: Ord a => [a] -> [a] -> Bool
subconjunto4 xs ys =
  fromList xs `isSubsetOf` fromList ys

-- Comprobación de equivalencia
-- ============================

-- La propiedad es
prop_subconjunto :: [Int] -> [Int] -> Bool
prop_subconjunto xs ys =
  all (== subconjunto1 xs ys)
      [subconjunto2 xs ys,
       subconjunto3 xs ys,
       subconjunto4 xs ys]

-- La comprobación es
--    λ> quickCheck prop_subconjunto
--    +++ OK, passed 100 tests.

-- Comparación de eficiencia
-- =========================

-- La comparación es
--    λ> subconjunto1 [1..2*10^4] [1..2*10^4]
--    True
```

```
--    (1.81 secs, 5,992,448 bytes)
--    λ> subconjunto2 [1..2*10^4] [1..2*10^4]
--    True
--    (1.83 secs, 6,952,200 bytes)
--    λ> subconjunto3 [1..2*10^4] [1..2*10^4]
--    True
--    (1.75 secs, 4,712,304 bytes)
--    λ> subconjunto4 [1..2*10^4] [1..2*10^4]
--    True
--    (0.04 secs, 6,312,056 bytes)
```

En Python

```
# ---------------------------------------------------------------------
# Definir la función
#    subconjunto : (list[A], list[A]) -> bool
# tal que subconjunto(xs, ys) se verifica si xs es un subconjunto de
# ys. por ejemplo,
#    subconjunto([3, 2, 3], [2, 5, 3, 5])  ==  True
#    subconjunto([3, 2, 3], [2, 5, 6, 5])  ==  False
# ---------------------------------------------------------------------

from sys import setrecursionlimit
from timeit import Timer, default_timer
from typing import TypeVar

from hypothesis import given
from hypothesis import strategies as st

setrecursionlimit(10**6)

A = TypeVar('A')

# 1ª solución
def subconjunto1(xs: list[A],
                 ys: list[A]) -> bool:
    return [x for x in xs if x in ys] == xs

# 2ª solución
def subconjunto2(xs: list[A],
```

```
                 ys: list[A]) -> bool:
    if xs:
        return xs[0] in ys and subconjunto2(xs[1:], ys)
    return True

# 3ª solución
def subconjunto3(xs: list[A],
                 ys: list[A]) -> bool:
    return all(x in ys for x in xs)

# 4ª solución
def subconjunto4(xs: list[A],
                 ys: list[A]) -> bool:
    return set(xs) <= set(ys)

# Comprobación de equivalencia
# ============================

# La propiedad es
@given(st.lists(st.integers()),
       st.lists(st.integers()))
def test_subconjunto(xs: list[int], ys: list[int]) -> None:
    assert subconjunto1(xs, ys)\
           == subconjunto2(xs, ys)\
           == subconjunto3(xs, ys)\
           == subconjunto4(xs, ys)

# La comprobación es
#    src> poetry run pytest -q reconocimiento_de_subconjunto.py
#    1 passed in 0.34s

# Comparación de eficiencia
# =========================

def tiempo(e: str) -> None:
    """Tiempo (en segundos) de evaluar la expresión e."""
    t = Timer(e, "", default_timer, globals()).timeit(1)
    print(f"{t:0.2f} segundos")

# La comparación es
```

```
#    >>> xs = list(range(20000))
#    >>> tiempo('subconjunto1(xs, xs)')
#    1.27 segundos
#    >>> tiempo('subconjunto2(xs, xs)')
#    1.84 segundos
#    >>> tiempo('subconjunto3(xs, xs)')
#    1.19 segundos
#    >>> tiempo('subconjunto4(xs, xs)')
#    0.01 segundos
```

2.2. Igualdad de conjuntos

En Haskell

```
-- ---------------------------------------------------------------------
-- Definir la función
--    iguales :: Ord a => [a] -> [a] -> Bool
-- tal que (iguales xs ys) se verifica si xs e ys son iguales. Por
-- ejemplo,
--    iguales [3,2,3] [2,3]    ==  True
--    iguales [3,2,3] [2,3,2]  ==  True
--    iguales [3,2,3] [2,3,4]  ==  False
--    iguales [2,3] [4,5]      ==  False
-- ---------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module Igualdad_de_conjuntos where

import Data.List (nub, sort)
import Data.Set (fromList)
import Test.QuickCheck

-- 1ª solución
-- ===========

iguales1 :: Ord a => [a] -> [a] -> Bool
iguales1 xs ys =
  subconjunto xs ys && subconjunto ys xs
```

```
-- (subconjunto xs ys) se verifica si xs es un subconjunto de ys. Por
-- ejemplo,
--    subconjunto [3,2,3] [2,5,3,5]  ==  True
--    subconjunto [3,2,3] [2,5,6,5]  ==  False
subconjunto :: Ord a => [a] -> [a] -> Bool
subconjunto xs ys =
  [x | x <- xs, x `elem` ys] == xs

-- 2ª solución
-- ===========

iguales2 :: Ord a => [a] -> [a] -> Bool
iguales2 xs ys =
  nub (sort xs) == nub (sort ys)

-- 3ª solución
-- ===========

iguales3 :: Ord a => [a] -> [a] -> Bool
iguales3 xs ys =
  fromList xs == fromList ys

-- Comprobación de equivalencia
-- ============================

-- La propiedad es
prop_iguales :: [Int] -> [Int] -> Bool
prop_iguales xs ys =
  all (== iguales1 xs ys)
      [iguales2 xs ys,
       iguales3 xs ys]

-- La comprobación es
--    λ> quickCheck prop_iguales
--    +++ OK, passed 100 tests.

-- Comparación de eficiencia
-- =========================

-- La comparación es
```

```
--    λ> iguales1 [1..2*10^4] [1..2*10^4]
--    True
--    (4.05 secs, 8,553,104 bytes)
--    λ> iguales2 [1..2*10^4] [1..2*10^4]
--    True
--    (4.14 secs, 9,192,768 bytes)
--    λ> iguales3 [1..2*10^4] [1..2*10^4]
--    True
--    (0.01 secs, 8,552,232 bytes)
```

En Python

```python
# ---------------------------------------------------------------------
# Definir la función
#    iguales : (list[Any], list[Any]) -> bool
# tal que iguales(xs, ys) se verifica si xs e ys son iguales. Por
# ejemplo,
#    iguales([3, 2, 3], [2, 3])    == True
#    iguales([3, 2, 3], [2, 3, 2]) == True
#    iguales([3, 2, 3], [2, 3, 4]) == False
#    iguales([2, 3], [4, 5])       == False
# ---------------------------------------------------------------------

from timeit import Timer, default_timer
from typing import Any

from hypothesis import given
from hypothesis import strategies as st

# 1ª solución
# ===========

def subconjunto(xs: list[Any],
                ys: list[Any]) -> bool:
    return [x for x in xs if x in ys] == xs

def iguales1(xs: list[Any],
             ys: list[Any]) -> bool:
    return subconjunto(xs, ys) and subconjunto(ys, xs)
```

```
# 2ª solución
# ===========

def iguales2(xs: list[Any],
             ys: list[Any]) -> bool:
    return set(xs) == set(ys)

# Equivalencia de las definiciones
# ================================

# La propiedad es
@given(st.lists(st.integers()),
       st.lists(st.integers()))
def test_iguales(xs: list[int], ys: list[int]) -> None:
    assert iguales1(xs, ys) == iguales2(xs, ys)

# La comprobación es
#    src> poetry run pytest -q igualdad_de_conjuntos.py
#    1 passed in 0.28s

# Comparación de eficiencia
# =========================

def tiempo(e: str) -> None:
    """Tiempo (en segundos) de evaluar la expresión e."""
    t = Timer(e, "", default_timer, globals()).timeit(1)
    print(f"{t:0.2f} segundos")

# La comparación es
#    >>> xs = list(range(20000))
#    >>> tiempo('iguales1(xs, xs)')
#    2.71 segundos
#    >>> tiempo('iguales2(xs, xs)')
#    0.01 segundos
```

2.3. Unión conjuntista de listas

En Haskell

```
-- ---------------------------------------------------------------------
-- Definir la función
--    union :: Ord a => [a] -> [a] -> [a]
-- tal que (union xs ys) es la unión de las listas sin elementos
-- repetidos xs e ys. Por ejemplo,
--    union [3,2,5] [5,7,3,4]  ==  [3,2,5,7,4]
--
-- Comprobar con QuickCheck que la unión es conmutativa.
-- ---------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module Union_conjuntista_de_listas where

import Data.List (nub, sort, union)
import qualified Data.Set as S (fromList, toList, union)
import Test.QuickCheck

-- 1ª solución
-- ===========

union1 :: Ord a => [a] -> [a] -> [a]
union1 xs ys = xs ++ [y | y <- ys, y `notElem` xs]

-- 2ª solución
-- ===========

union2 :: Ord a => [a] -> [a] -> [a]
union2 [] ys = ys
union2 (x:xs) ys
  | x `elem` ys = xs `union2` ys
  | otherwise   = x : xs `union2` ys

-- 3ª solución
-- ===========

union3 :: Ord a => [a] -> [a] -> [a]
```

```haskell
union3 = union

-- 4ª solución
-- ===========

union4 :: Ord a => [a] -> [a] -> [a]
union4 xs ys =
  S.toList (S.fromList xs `S.union` S.fromList ys)

-- Comprobación de equivalencia
-- ============================

-- La propiedad es
prop_union :: [Int] -> [Int] -> Bool
prop_union xs ys =
  all (== sort (xs' `union1` ys'))
      [sort (xs' `union2` ys'),
       sort (xs' `union3` ys'),
       xs' `union4` ys']
  where xs' = nub xs
        ys' = nub ys

-- La comprobación es
--    λ> quickCheck prop_union
--    +++ OK, passed 100 tests.

-- Comparación de eficiencia
-- =========================

-- La comparación es
--    λ> length (union1 [0,2..3*10^4] [1,3..3*10^4])
--    30001
--    (2.37 secs, 7,153,536 bytes)
--    λ> length (union2 [0,2..3*10^4] [1,3..3*10^4])
--    30001
--    (2.38 secs, 6,553,752 bytes)
--    λ> length (union3 [0,2..3*10^4] [1,3..3*10^4])
--    30001
--    (11.56 secs, 23,253,553,472 bytes)
--    λ> length (union4 [0,2..3*10^4] [1,3..3*10^4])
```

```
--    30001
--    (0.04 secs, 10,992,056 bytes)

-- Comprobación de la propiedad
-- ============================

-- La propiedad es
prop_union_conmutativa :: [Int] -> [Int] -> Bool
prop_union_conmutativa xs ys =
  iguales (xs `union1` ys) (ys `union1` xs)

-- (iguales xs ys) se verifica si xs e ys son iguales. Por ejemplo,
--    iguales [3,2,3] [2,3]    ==  True
--    iguales [3,2,3] [2,3,2]  ==  True
--    iguales [3,2,3] [2,3,4]  ==  False
--    iguales [2,3] [4,5]      ==  False
iguales :: Ord a => [a] -> [a] -> Bool
iguales xs ys =
  S.fromList xs == S.fromList ys

-- La comprobación es
--    λ> quickCheck prop_union_conmutativa
--    +++ OK, passed 100 tests.
```

En Python

```
# ---------------------------------------------------------------------
# Definir la función
#    union : (list[A], list[A]) -> list[A]
# tal que union(xs, ys) es la unión de las listas sin elementos
# repetidos xs e ys. Por ejemplo,
#    union([3, 2, 5], [5, 7, 3, 4])  ==  [3, 2, 5, 7, 4]
#
# Comprobar con Hypothesis que la unión es conmutativa.
# ---------------------------------------------------------------------

from sys import setrecursionlimit
from timeit import Timer, default_timer
from typing import TypeVar
```

```
from hypothesis import given
from hypothesis import strategies as st

setrecursionlimit(10**6)

A = TypeVar('A')

# 1ª solución
# ===========

def union1(xs: list[A], ys: list[A]) -> list[A]:
    return xs + [y for y in ys if y not in xs]

# 2ª solución
# ===========

def union2(xs: list[A], ys: list[A]) -> list[A]:
    if not xs:
        return ys
    if xs[0] in ys:
        return union2(xs[1:], ys)
    return [xs[0]] + union2(xs[1:], ys)

# 3ª solución
# ===========

def union3(xs: list[A], ys: list[A]) -> list[A]:
    zs = ys[:]
    for x in xs:
        if x not in ys:
            zs.append(x)
    return zs

# 4ª solución
# ===========

def union4(xs: list[A], ys: list[A]) -> list[A]:
    return list(set(xs) | set(ys))

# Comprobación de equivalencia
```

```
# ============================
#
# La propiedad es
@given(st.lists(st.integers()),
       st.lists(st.integers()))
def test_union(xs: list[int], ys: list[int]) -> None:
    xs1 = list(set(xs))
    ys1 = list(set(ys))
    assert sorted(union1(xs1, ys1)) ==\
           sorted(union2(xs1, ys1)) ==\
           sorted(union3(xs1, ys1)) ==\
           sorted(union4(xs1, ys1))

# La comprobación es
#    src> poetry run pytest -q union_conjuntista_de_listas.py
#    1 passed in 0.36s

# Comparación de eficiencia
# =========================

def tiempo(e: str) -> None:
    """Tiempo (en segundos) de evaluar la expresión e."""
    t = Timer(e, "", default_timer, globals()).timeit(1)
    print(f"{t:0.2f} segundos")

# La comparación es
#    >>> tiempo('union1(list(range(0,30000,2)), list(range(1,30000,2)))')
#    1.30 segundos
#    >>> tiempo('union2(list(range(0,30000,2)), list(range(1,30000,2)))')
#    2.84 segundos
#    >>> tiempo('union3(list(range(0,30000,2)), list(range(1,30000,2)))')
#    1.45 segundos
#    >>> tiempo('union4(list(range(0,30000,2)), list(range(1,30000,2)))')
#    0.00 segundos

# Comprobación de la propiedad
# ============================

# iguales(xs, ys) se verifica si xs e ys son iguales como conjuntos. Por
# ejemplo,
```

```
#    iguales([3,2,3], [2,3])    ==  True
#    iguales([3,2,3], [2,3,2])  ==  True
#    iguales([3,2,3], [2,3,4])  ==  False
#    iguales([2,3], [4,5])      ==  False
def iguales(xs: list[A], ys: list[A]) -> bool:
    return set(xs) == set(ys)

# La propiedad es
@given(st.lists(st.integers()),
       st.lists(st.integers()))
def test_union_conmutativa(xs: list[int], ys: list[int]) -> None:
    xs1 = list(set(xs))
    ys1 = list(set(ys))
    assert iguales(union1(xs1, ys1), union1(ys1, xs1))

# La comprobación es
#    src> poetry run pytest -q union_conjuntista_de_listas.py
#    2 passed in 0.49s
```

2.4. Intersección conjuntista de listas

En Haskell

```
-- ---------------------------------------------------------------------
-- Definir la función
--    interseccion :: Eq a => [a] -> [a] -> [a]
-- tal que (interseccion xs ys) es la intersección de las listas sin
-- elementos repetidos xs e ys. Por ejemplo,
--    interseccion [3,2,5] [5,7,3,4]  ==  [3,5]
--    interseccion [3,2,5] [9,7,6,4]  ==  []
-- ---------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module Interseccion_conjuntista_de_listas where

import Data.List (nub, sort, intersect)
import qualified Data.Set as S (fromList, toList, intersection )
import Test.QuickCheck
```

```
-- 1ª solución
-- ===========

interseccion1 :: Eq a => [a] -> [a] -> [a]
interseccion1 xs ys =
  [x | x <- xs, x `elem` ys]

-- 2ª solución
-- ===========

interseccion2 :: Ord a => [a] -> [a] -> [a]
interseccion2 [] _ = []
interseccion2 (x:xs) ys
  | x `elem` ys = x : xs `interseccion2` ys
  | otherwise   = xs `interseccion2` ys

-- 3ª solución
-- ===========

interseccion3 :: Ord a => [a] -> [a] -> [a]
interseccion3 = intersect

-- 4ª solución
-- ===========

interseccion4 :: Ord a => [a] -> [a] -> [a]
interseccion4 xs ys =
  S.toList (S.fromList xs `S.intersection` S.fromList ys)

-- Comprobación de equivalencia
-- ============================

-- La propiedad es
prop_interseccion :: [Int] -> [Int] -> Bool
prop_interseccion xs ys =
  all (== sort (xs' `interseccion1` ys'))
      [sort (xs' `interseccion2` ys'),
       sort (xs' `interseccion3` ys'),
       xs' `interseccion4` ys']
  where xs' = nub xs
```

```
      ys' = nub ys

-- La comprobación es
--    λ> quickCheck prop_interseccion
--    +++ OK, passed 100 tests.

-- Comparación de eficiencia
-- =========================

-- La comparación es
--    λ> length (interseccion1 [0..3*10^4] [1,3..3*10^4])
--    15000
--    (2.94 secs, 6,673,360 bytes)
--    λ> length (interseccion2 [0..3*10^4] [1,3..3*10^4])
--    15000
--    (3.04 secs, 9,793,440 bytes)
--    λ> length (interseccion3 [0..3*10^4] [1,3..3*10^4])
--    15000
--    (5.39 secs, 6,673,472 bytes)
--    λ> length (interseccion4 [0..3*10^4] [1,3..3*10^4])
--    15000
--    (0.04 secs, 8,593,176 bytes)
```

En Python

```
# ---------------------------------------------------------------------
# Definir la función
#    interseccion : (list[A], list[A]) -> list[A]
# tal que interseccion(xs, ys) es la intersección de las listas sin
# elementos repetidos xs e ys. Por ejemplo,
#    interseccion([3, 2, 5], [5, 7, 3, 4]) == [3, 5]
#    interseccion([3, 2, 5], [9, 7, 6, 4]) == []
# ---------------------------------------------------------------------

from sys import setrecursionlimit
from timeit import Timer, default_timer
from typing import TypeVar

from hypothesis import given
from hypothesis import strategies as st
```

```
setrecursionlimit(10**6)

A = TypeVar('A')

# 1ª solución
# ===========

def interseccion1(xs: list[A], ys: list[A]) -> list[A]:
    return [x for x in xs if x in ys]

# 2ª solución
# ===========

def interseccion2(xs: list[A], ys: list[A]) -> list[A]:
    if not xs:
        return []
    if xs[0] in ys:
        return [xs[0]] + interseccion2(xs[1:], ys)
    return interseccion2(xs[1:], ys)

# 3ª solución
# ===========

def interseccion3(xs: list[A], ys: list[A]) -> list[A]:
    zs = []
    for x in xs:
        if x in ys:
            zs.append(x)
    return zs

# 4ª solución
# ===========

def interseccion4(xs: list[A], ys: list[A]) -> list[A]:
    return list(set(xs) & set(ys))

# Comprobación de equivalencia
# ============================
#
```

```
# La propiedad es
@given(st.lists(st.integers()),
       st.lists(st.integers()))
def test_interseccion(xs: list[int], ys: list[int]) -> None:
    xs1 = list(set(xs))
    ys1 = list(set(ys))
    assert sorted(interseccion1(xs1, ys1)) ==\
           sorted(interseccion2(xs1, ys1)) ==\
           sorted(interseccion3(xs1, ys1)) ==\
           sorted(interseccion4(xs1, ys1))

# La comprobación es
#    src> poetry run pytest -q interseccion_conjuntista_de_listas.py
#    1 passed in 0.33s

# Comparación de eficiencia
# =========================

def tiempo(e: str) -> None:
    """Tiempo (en segundos) de evaluar la expresión e."""
    t = Timer(e, "", default_timer, globals()).timeit(1)
    print(f"{t:0.2f} segundos")

# La comparación es
#    >>> tiempo('interseccion1(list(range(0,20000)), list(range(1,20000,2)))')
#    0.98 segundos
#    >>> tiempo('interseccion2(list(range(0,20000)), list(range(1,20000,2)))')
#    2.13 segundos
#    >>> tiempo('interseccion3(list(range(0,20000)), list(range(1,20000,2)))')
#    0.87 segundos
#    >>> tiempo('interseccion4(list(range(0,20000)), list(range(1,20000,2)))')
#    0.00 segundos
```

2.5. Diferencia conjuntista de listas

En Haskell

```
-- ---------------------------------------------------------------------
-- Definir la función
--    diferencia :: Eq a => [a] -> [a] -> [a]
```

```
-- tal que (diferencia xs ys) es la diferencia de las listas sin
-- elementos repetidos xs e ys. Por ejemplo,
--    diferencia [3,2,5,6] [5,7,3,4]  ==  [2,6]
--    diferencia [3,2,5] [5,7,3,2]    ==  []
-- ---------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module Diferencia_conjuntista_de_listas where

import Data.List (nub, sort, (\\))
import qualified Data.Set as S (fromList, toList, (\\) )
import Test.QuickCheck

-- 1ª solución
-- ===========

diferencia1 :: Eq a => [a] -> [a] -> [a]
diferencia1 xs ys =
  [x | x <- xs, x `notElem` ys]

-- 2ª solución
-- ===========

diferencia2 :: Ord a => [a] -> [a] -> [a]
diferencia2 [] _ = []
diferencia2 (x:xs) ys
  | x `elem` ys = xs `diferencia2` ys
  | otherwise   = x : xs `diferencia2` ys

-- 3ª solución
-- ===========

diferencia3 :: Ord a => [a] -> [a] -> [a]
diferencia3 = (\\)

-- 4ª solución
-- ===========

diferencia4 :: Ord a => [a] -> [a] -> [a]
```

```haskell
diferencia4 xs ys =
  S.toList (S.fromList xs S.\\ S.fromList ys)

-- Comprobación de equivalencia
-- ============================

-- La propiedad es
prop_diferencia :: [Int] -> [Int] -> Bool
prop_diferencia xs ys =
  all (== sort (xs' `diferencia1` ys'))
      [sort (xs' `diferencia2` ys'),
       sort (xs' `diferencia3` ys'),
       xs' `diferencia4` ys']
  where xs' = nub xs
        ys' = nub ys

-- La comprobación es
--    λ> quickCheck prop_diferencia
--    +++ OK, passed 100 tests.

-- Comparación de eficiencia
-- =========================

-- La comparación es
--    λ> length (diferencia1 [0..3*10^4] [1,3..3*10^4])
--    15001
--    (3.39 secs, 9,553,528 bytes)
--    λ> length (diferencia2 [0..3*10^4] [1,3..3*10^4])
--    15001
--    (2.98 secs, 9,793,528 bytes)
--    λ> length (diferencia3 [0..3*10^4] [1,3..3*10^4])
--    15001
--    (3.61 secs, 11,622,502,792 bytes)
--    λ> length (diferencia4 [0..3*10^4] [1,3..3*10^4])
--    15001
--    (0.02 secs, 10,092,832 bytes)
```

En Python

```
# ---------------------------------------------------------------------
# Definir la función
#    diferencia : (list[A], list[A]) -> list[A]
# tal que diferencia(xs, ys) es la diferencia de las listas sin
# elementos repetidos xs e ys. Por ejemplo,
#    diferencia([3, 2, 5, 6], [5, 7, 3, 4]) == [2, 6]
#    diferencia([3, 2, 5], [5, 7, 3, 2])    == []
# ---------------------------------------------------------------------

from sys import setrecursionlimit
from timeit import Timer, default_timer
from typing import TypeVar

from hypothesis import given
from hypothesis import strategies as st

setrecursionlimit(10**6)

A = TypeVar('A')

# 1ª solución
# ===========

def diferencia1(xs: list[A], ys: list[A]) -> list[A]:
    return [x for x in xs if x not in ys]

# 2ª solución
# ===========

def diferencia2(xs: list[A], ys: list[A]) -> list[A]:
    if not xs:
        return []
    if xs[0] in ys:
        return diferencia2(xs[1:], ys)
    return [xs[0]] + diferencia2(xs[1:], ys)

# 3ª solución
# ===========
```

```
def diferencia3(xs: list[A], ys: list[A]) -> list[A]:
    zs = []
    for x in xs:
        if x not in ys:
            zs.append(x)
    return zs

# 4ª solución
# ===========

def diferencia4(xs: list[A], ys: list[A]) -> list[A]:
    return list(set(xs) - set(ys))

# Comprobación de equivalencia
# ============================

# La propiedad es
@given(st.lists(st.integers()),
       st.lists(st.integers()))
def test_diferencia(xs: list[int], ys: list[int]) -> None:
    xs1 = list(set(xs))
    ys1 = list(set(ys))
    assert sorted(diferencia1(xs1, ys1)) ==\
           sorted(diferencia2(xs1, ys1)) ==\
           sorted(diferencia3(xs1, ys1)) ==\
           sorted(diferencia4(xs1, ys1))

# La comprobación es
#    src> poetry run pytest -q diferencia_conjuntista_de_listas.py
#    1 passed in 0.39s

# Comparación de eficiencia
# =========================

def tiempo(e: str) -> None:
    """Tiempo (en segundos) de evaluar la expresión e."""
    t = Timer(e, "", default_timer, globals()).timeit(1)
    print(f"{t:0.2f} segundos")

# La comparación es
```

```
#    >>> tiempo('diferencia1(list(range(0,20000)), list(range(1,20000,2)))')
#    0.89 segundos
#    >>> tiempo('diferencia2(list(range(0,20000)), list(range(1,20000,2)))')
#    2.11 segundos
#    >>> tiempo('diferencia3(list(range(0,20000)), list(range(1,20000,2)))')
#    1.06 segundos
#    >>> tiempo('diferencia4(list(range(0,20000)), list(range(1,20000,2)))')
#    0.01 segundos
```

2.6. Divisores de un número

En Haskell

```
-- ---------------------------------------------------------------------
-- Definir la función
--    divisores :: Integer -> [Integer]
-- tal que (divisores n) es el conjunto de divisores de n. Por
-- ejemplo,
--   divisores 30  ==  [1,2,3,5,6,10,15,30]
--   length (divisores (product [1..10]))  ==  270
--   length (divisores (product [1..25]))  ==  340032
-- ---------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-type-defaults #-}

module Divisores_de_un_numero where

import Data.List (group, inits, nub, sort, subsequences)
import Data.Numbers.Primes (primeFactors)
import Data.Set (toList)
import Math.NumberTheory.ArithmeticFunctions (divisors)
import Test.QuickCheck

-- 1ª solución
-- ===========

divisores1 :: Integer -> [Integer]
divisores1 n = [x | x <- [1..n], n `rem` x == 0]

-- 2ª solución
```

```haskell
-- ===========

divisores2 :: Integer -> [Integer]
divisores2 n = [x | x <- [1..n], x `esDivisorDe` n]

-- (esDivisorDe x n) se verifica si x es un divisor de n. Por ejemplo,
--    esDivisorDe 2 6  ==  True
--    esDivisorDe 4 6  ==  False
esDivisorDe :: Integer -> Integer -> Bool
esDivisorDe x n = n `rem` x == 0

-- 3ª solución
-- ===========

divisores3 :: Integer -> [Integer]
divisores3 n = filter (`esDivisorDe` n) [1..n]

-- 4ª solución
-- ===========

divisores4 :: Integer -> [Integer]
divisores4 = filter <$> flip esDivisorDe <*> enumFromTo 1

-- 5ª solución
-- ===========

divisores5 :: Integer -> [Integer]
divisores5 n = xs ++ [n `div` y | y <- ys]
  where xs = primerosDivisores1 n
        (z:zs) = reverse xs
        ys | z^2 == n  = zs
           | otherwise = z:zs

-- (primerosDivisores n) es la lista de los divisores del número n cuyo
-- cuadrado es menor o gual que n. Por ejemplo,
--    primerosDivisores 25  ==  [1,5]
--    primerosDivisores 30  ==  [1,2,3,5]
primerosDivisores1 :: Integer -> [Integer]
primerosDivisores1 n =
   [x | x <- [1..round (sqrt (fromIntegral n))],
```

```
        x `esDivisorDe` n]

-- 6ª solución
-- ===========

divisores6 :: Integer -> [Integer]
divisores6 n = aux [1..n]
  where aux [] = []
        aux (x:xs) | x `esDivisorDe` n = x : aux xs
                   | otherwise         = aux xs

-- 7ª solución
-- ===========

divisores7 :: Integer -> [Integer]
divisores7 n = xs ++ [n `div` y | y <- ys]
  where xs = primerosDivisores2 n
        (z:zs) = reverse xs
        ys | z^2 == n  = zs
           | otherwise = z:zs

primerosDivisores2 :: Integer -> [Integer]
primerosDivisores2 n = aux [1..round (sqrt (fromIntegral n))]
  where aux [] = []
        aux (x:xs) | x `esDivisorDe` n = x : aux xs
                   | otherwise         = aux xs

-- 8ª solución
-- ===========

divisores8 :: Integer -> [Integer]
divisores8 =
  nub . sort . map product . subsequences . primeFactors

-- 9ª solución
-- ===========

divisores9 :: Integer -> [Integer]
divisores9 = sort
           . map (product . concat)
```

```
          . productoCartesiano
          . map inits
          . group
          . primeFactors

-- (productoCartesiano xss) es el producto cartesiano de los conjuntos
-- xss. Por ejemplo,
--    λ> productoCartesiano [[1,3],[2,5],[6,4]]
--    [[1,2,6],[1,2,4],[1,5,6],[1,5,4],[3,2,6],[3,2,4],[3,5,6],[3,5,4]]
productoCartesiano :: [[a]] -> [[a]]
productoCartesiano []       = [[]]
productoCartesiano (xs:xss) =
  [x:ys | x <- xs, ys <- productoCartesiano xss]

-- 10ª solución
-- ============

divisores10 :: Integer -> [Integer]
divisores10 = sort
            . map (product . concat)
            . mapM inits
            . group
            . primeFactors

-- 11ª solución
-- ============

divisores11 :: Integer -> [Integer]
divisores11 = toList . divisors

-- Comprobación de equivalencia
-- ============================

-- La propiedad es
prop_divisores :: Positive Integer -> Bool
prop_divisores (Positive n) =
  all (== divisores1 n)
      [ divisores2 n
      , divisores3 n
      , divisores4 n
```

```
      , divisores5 n
      , divisores6 n
      , divisores7 n
      , divisores8 n
      , divisores9 n
      , divisores10 n
      , divisores11 n
      ]

-- La comprobación es
--    λ> quickCheck prop_divisores
--    +++ OK, passed 100 tests.

-- Comparación de la eficiencia
-- ============================

-- La comparación es
--    λ> length (divisores1 (product [1..11]))
--    540
--    (18.55 secs, 7,983,950,592 bytes)
--    λ> length (divisores2 (product [1..11]))
--    540
--    (18.81 secs, 7,983,950,592 bytes)
--    λ> length (divisores3 (product [1..11]))
--    540
--    (12.79 secs, 6,067,935,544 bytes)
--    λ> length (divisores4 (product [1..11]))
--    540
--    (12.51 secs, 6,067,935,592 bytes)
--    λ> length (divisores5 (product [1..11]))
--    540
--    (0.03 secs, 1,890,296 bytes)
--    λ> length (divisores6 (product [1..11]))
--    540
--    (21.46 secs, 9,899,961,392 bytes)
--    λ> length (divisores7 (product [1..11]))
--    540
--    (0.02 secs, 2,195,800 bytes)
--    λ> length (divisores8 (product [1..11]))
--    540
```

```
--    (0.09 secs, 107,787,272 bytes)
--    λ> length (divisores9 (product [1..11]))
--    540
--    (0.02 secs, 2,150,472 bytes)
--    λ> length (divisores10 (product [1..11]))
--    540
--    (0.01 secs, 1,652,120 bytes)
--    λ> length (divisores11 (product [1..11]))
--    540
--    (0.01 secs, 796,056 bytes)
--
--    λ> length (divisores5 (product [1..17]))
--    10752
--    (10.16 secs, 3,773,953,128 bytes)
--    λ> length (divisores7 (product [1..17]))
--    10752
--    (9.83 secs, 4,679,260,712 bytes)
--    λ> length (divisores9 (product [1..17]))
--    10752
--    (0.06 secs, 46,953,344 bytes)
--    λ> length (divisores10 (product [1..17]))
--    10752
--    (0.02 secs, 33,633,712 bytes)
--    λ> length (divisores11 (product [1..17]))
--    10752
--    (0.03 secs, 6,129,584 bytes)
--
--    λ> length (divisores10 (product [1..27]))
--    677376
--    (2.14 secs, 3,291,277,736 bytes)
--    λ> length (divisores11 (product [1..27]))
--    677376
--    (0.56 secs, 396,042,280 bytes)
```

En Python

```
# ---------------------------------------------------------------------
# Definir la función
#    divisores : (int) -> list[int]
# tal que divisores(n) es el conjunto de divisores de n. Por
```

```
#  ejemplo,
#    divisores(30)  ==  [1, 2, 3, 5, 6, 10, 15, 30]
#    len(divisores1(factorial(10)))  ==  270
#    len(divisores1(factorial(25)))  ==  340032
#  ---------------------------------------------------------------------

from math import factorial, sqrt
from sys import setrecursionlimit
from timeit import Timer, default_timer

from hypothesis import given
from hypothesis import strategies as st
from sympy import divisors

setrecursionlimit(10**6)

# 1ª solución
# ===========

def divisores1(n: int) -> list[int]:
    return [x for x in range(1, n + 1) if n % x == 0]

# 2ª solución
# ===========

# esDivisorDe(x, n) se verifica si x es un divisor de n. Por ejemplo,
#    esDivisorDe(2, 6)  ==  True
#    esDivisorDe(4, 6)  ==  False
def esDivisorDe(x: int, n: int) -> bool:
    return n % x == 0

def divisores2(n: int) -> list[int]:
    return [x for x in range(1, n + 1) if esDivisorDe(x, n)]

# 3ª solución
# ===========

def divisores3(n: int) -> list[int]:
    return list(filter(lambda x: esDivisorDe(x, n), range(1, n + 1)))
```

```python
# 4ª solución
# ===========

# primerosDivisores(n) es la lista de los divisores del número n cuyo
# cuadrado es menor o gual que n. Por ejemplo,
#    primerosDivisores(25)  ==  [1,5]
#    primerosDivisores(30)  ==  [1,2,3,5]
def primerosDivisores(n: int) -> list[int]:
    return [x for x in range(1, 1 + round(sqrt(n))) if n % x == 0]

def divisores4(n: int) -> list[int]:
    xs = primerosDivisores(n)
    zs = list(reversed(xs))
    if zs[0]**2 == n:
        return xs + [n // a for a in zs[1:]]
    return xs + [n // a for a in zs]

# 5ª solución
# ===========

def divisores5(n: int) -> list[int]:
    def aux(xs: list[int]) -> list[int]:
        if xs:
            if esDivisorDe(xs[0], n):
                return [xs[0]] + aux(xs[1:])
            return aux(xs[1:])
        return xs

    return aux(list(range(1, n + 1)))

# 6ª solución
# ============

def divisores6(n: int) -> list[int]:
    xs = []
    for x in range(1, n+1):
        if n % x == 0:
            xs.append(x)
    return xs
```

```python
# 7ª solución
# ===========

def divisores7(n: int) -> list[int]:
    x = 1
    xs = []
    ys = []
    while x * x < n:
        if n % x == 0:
            xs.append(x)
            ys.append(n // x)
        x = x + 1
    if x * x == n:
        xs.append(x)
    return xs + list(reversed(ys))

# 8ª solución
# ============

def divisores8(n: int) -> list[int]:
    return divisors(n)

# Comprobación de equivalencia
# ============================

# La propiedad es
@given(st.integers(min_value=2, max_value=1000))
def test_divisores(n: int) -> None:
    assert divisores1(n) ==\
           divisores2(n) ==\
           divisores3(n) ==\
           divisores4(n) ==\
           divisores5(n) ==\
           divisores6(n) ==\
           divisores7(n) ==\
           divisores8(n)

# La comprobación es
#    src> poetry run pytest -q divisores_de_un_numero.py
#    1 passed in 0.84s
```

```
# Comparación de eficiencia
# =========================

def tiempo(e: str) -> None:
    """Tiempo (en segundos) de evaluar la expresión e."""
    t = Timer(e, "", default_timer, globals()).timeit(1)
    print(f"{t:0.2f} segundos")

# La comparación es
#    >>> tiempo('divisores5(4*factorial(7))')
#    1.40 segundos
#
#    >>> tiempo('divisores1(factorial(11))')
#    1.79 segundos
#    >>> tiempo('divisores2(factorial(11))')
#    3.80 segundos
#    >>> tiempo('divisores3(factorial(11))')
#    5.22 segundos
#    >>> tiempo('divisores4(factorial(11))')
#    0.00 segundos
#    >>> tiempo('divisores6(factorial(11))')
#    3.51 segundos
#    >>> tiempo('divisores7(factorial(11))')
#    0.00 segundos
#    >>> tiempo('divisores8(factorial(11))')
#    0.00 segundos
#
#    >>> tiempo('divisores4(factorial(17))')
#    2.23 segundos
#    >>> tiempo('divisores7(factorial(17))')
#    3.22 segundos
#    >>> tiempo('divisores8(factorial(17))')
#    0.00 segundos
#
#    >>> tiempo('divisores8(factorial(27))')
#    0.28 segundos
```

2.7. Divisores primos

En Haskell

```
-- ---------------------------------------------------------------------
-- Definir la función
--    divisoresPrimos :: Integer -> [Integer]
-- tal que (divisoresPrimos x) es la lista de los divisores primos de x.
-- Por ejemplo,
--    divisoresPrimos 40 == [2,5]
--    divisoresPrimos 70 == [2,5,7]w
--    length (divisoresPrimos (product [1..20000])) == 2262
-- ------------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-type-defaults #-}

module Divisores_primos where

import Data.List (nub)
import Data.Set (toList)
import Data.Numbers.Primes (isPrime, primeFactors)
import Math.NumberTheory.ArithmeticFunctions (divisors)
import Test.QuickCheck

-- 1ª solución
-- ===========

divisoresPrimos1 :: Integer -> [Integer]
divisoresPrimos1 x = [n | n <- divisores1 x, primo1 n]

-- (divisores n) es la lista de los divisores del número n. Por ejemplo,
--    divisores 25  ==  [1,5,25]
--    divisores 30  ==  [1,2,3,5,6,10,15,30]
divisores1 :: Integer -> [Integer]
divisores1 n = [x | x <- [1..n], n `mod` x == 0]

-- (primo n) se verifica si n es primo. Por ejemplo,
--    primo 30  == False
--    primo 31  == True
primo1 :: Integer -> Bool
primo1 n = divisores1 n == [1, n]
```

```
-- 2ª solución
-- ===========

divisoresPrimos2 :: Integer -> [Integer]
divisoresPrimos2 x = [n | n <- divisores2 x, primo2 n]

divisores2 :: Integer -> [Integer]
divisores2 n = xs ++ [n `div` y | y <- ys]
  where xs = primerosDivisores2 n
        (z:zs) = reverse xs
        ys | z^2 == n  = zs
           | otherwise = z:zs

-- (primerosDivisores n) es la lista de los divisores del número n cuyo
-- cuadrado es menor o gual que n. Por ejemplo,
--    primerosDivisores 25  ==  [1,5]
--    primerosDivisores 30  ==  [1,2,3,5]
primerosDivisores2 :: Integer -> [Integer]
primerosDivisores2 n =
   [x | x <- [1..round (sqrt (fromIntegral n))],
        n `mod` x == 0]

primo2 :: Integer -> Bool
primo2 1 = False
primo2 n = primerosDivisores2 n == [1]

-- 3ª solución
-- ===========

divisoresPrimos3 :: Integer -> [Integer]
divisoresPrimos3 x = [n | n <- divisores3 x, primo3 n]

divisores3 :: Integer -> [Integer]
divisores3 n = xs ++ [n `div` y | y <- ys]
  where xs = primerosDivisores3 n
        (z:zs) = reverse xs
        ys | z^2 == n  = zs
           | otherwise = z:zs
```

```
primerosDivisores3 :: Integer -> [Integer]
primerosDivisores3 n =
  filter ((== 0) . mod n) [1..round (sqrt (fromIntegral n))]

primo3 :: Integer -> Bool
primo3 1 = False
primo3 n = primerosDivisores3 n == [1]

-- 4ª solución
-- ===========

divisoresPrimos4 :: Integer -> [Integer]
divisoresPrimos4 n
  | even n = 2 : divisoresPrimos4 (reducido n 2)
  | otherwise = aux n [3,5..n]
  where aux 1 _  = []
        aux _ [] = []
        aux m (x:xs) | m `mod` x == 0 = x : aux (reducido m x) xs
                     | otherwise      = aux m xs

-- (reducido m x) es el resultado de dividir repetidamente m por x,
-- mientras sea divisible. Por ejemplo,
--    reducido 36 2  ==  9
reducido :: Integer -> Integer -> Integer
reducido m x | m `mod` x == 0 = reducido (m `div` x) x
             | otherwise      = m

-- 5ª solución
-- ===========

divisoresPrimos5 :: Integer -> [Integer]
divisoresPrimos5 = nub . primeFactors

-- 6ª solución
-- ===========

divisoresPrimos6 :: Integer -> [Integer]
divisoresPrimos6 = filter isPrime . toList . divisors

-- Comprobación de equivalencia
```

```
-- ============================

-- La propiedad es
prop_divisoresPrimos :: Integer -> Property
prop_divisoresPrimos n =
  n > 1 ==>
  all (== divisoresPrimos1 n)
      [divisoresPrimos2 n,
       divisoresPrimos3 n,
       divisoresPrimos4 n,
       divisoresPrimos5 n,
       divisoresPrimos6 n]

-- La comprobación es
--    λ> quickCheck prop_divisoresPrimos
--    +++ OK, passed 100 tests; 108 discarded.

-- Comparación de eficiencia
-- =========================

-- La comparación es
--    λ> divisoresPrimos1 (product [1..11])
--    [2,3,5,7,11]
--    (18.34 secs, 7,984,382,104 bytes)
--    λ> divisoresPrimos2 (product [1..11])
--    [2,3,5,7,11]
--    (0.02 secs, 2,610,976 bytes)
--    λ> divisoresPrimos3 (product [1..11])
--    [2,3,5,7,11]
--    (0.02 secs, 2,078,288 bytes)
--    λ> divisoresPrimos4 (product [1..11])
--    [2,3,5,7,11]
--    (0.02 secs, 565,992 bytes)
--    λ> divisoresPrimos5 (product [1..11])
--    [2,3,5,7,11]
--    (0.01 secs, 568,000 bytes)
--    λ> divisoresPrimos6 (product [1..11])
--    [2,3,5,7,11]
--    (0.00 secs, 2,343,392 bytes)
--
```

```
--    λ> divisoresPrimos2 (product [1..16])
--    [2,3,5,7,11,13]
--    (2.32 secs, 923,142,480 bytes)
--    λ> divisoresPrimos3 (product [1..16])
--    [2,3,5,7,11,13]
--    (0.80 secs, 556,961,088 bytes)
--    λ> divisoresPrimos4 (product [1..16])
--    [2,3,5,7,11,13]
--    (0.01 secs, 572,368 bytes)
--    λ> divisoresPrimos5 (product [1..16])
--    [2,3,5,7,11,13]
--    (0.01 secs, 31,665,896 bytes)
--    λ> divisoresPrimos6 (product [1..16])
--    [2,3,5,7,11,13]
--    (0.01 secs, 18,580,584 bytes)
--
--    λ> length (divisoresPrimos4 (product [1..30]))
--    10
--    (0.01 secs, 579,168 bytes)
--    λ> length (divisoresPrimos5 (product [1..30]))
--    10
--    (0.01 secs, 594,976 bytes)
--    λ> length (divisoresPrimos6 (product [1..30]))
--    10
--    (3.38 secs, 8,068,783,408 bytes)
--
--    λ> length (divisoresPrimos4 (product [1..20000]))
--    2262
--    (1.20 secs, 1,940,069,976 bytes)
--    λ> length (divisoresPrimos5 (product [1..20000]))
--    2262
--    (1.12 secs, 1,955,921,736 bytes)
```

En Python

```
# ---------------------------------------------------------------------
# Definir la función
#    divisoresPrimos : (int) -> list[int]
# tal que divisoresPrimos(x) es la lista de los divisores primos de x.
# Por ejemplo,
```

```
#    divisoresPrimos(40) == [2, 5]
#    divisoresPrimos(70) == [2, 5, 7]
#    len(divisoresPrimos4(producto(list(range(1, 20001))))) == 2262
# ---------------------------------------------------------------------

from functools import reduce
from math import sqrt
from operator import mul
from sys import setrecursionlimit
from timeit import Timer, default_timer

from hypothesis import given
from hypothesis import strategies as st
from sympy import divisors, isprime, primefactors

setrecursionlimit(10**6)

# 1ª solución
# ===========

# divisores(n) es la lista de los divisores del número n. Por ejemplo,
#    divisores(30)  ==  [1,2,3,5,6,10,15,30]
def divisores1(n: int) -> list[int]:
    return [x for x in range(1, n + 1) if n % x == 0]

# primo(n) se verifica si n es primo. Por ejemplo,
#    primo(30)  == False
#    primo(31)  == True
def primo1(n: int) -> bool:
    return divisores1(n) == [1, n]

def divisoresPrimos1(x: int) -> list[int]:
    return [n for n in divisores1(x) if primo1(n)]

# 2ª solución
# ===========

# primerosDivisores(n) es la lista de los divisores del número n cuyo
# cuadrado es menor o gual que n. Por ejemplo,
#    primerosDivisores(25)  ==  [1,5]
```

```python
#    primerosDivisores(30)  ==  [1,2,3,5]
def primerosDivisores2(n: int) -> list[int]:
    return [x for x in range(1, 1 + round(sqrt(n))) if n % x == 0]

def divisores2(n: int) -> list[int]:
    xs = primerosDivisores2(n)
    zs = list(reversed(xs))
    if zs[0]**2 == n:
        return xs + [n // a for a in zs[1:]]
    return xs + [n // a for a in zs]

def primo2(n: int) -> bool:
    return divisores2(n) == [1, n]

def divisoresPrimos2(x: int) -> list[int]:
    return [n for n in divisores2(x) if primo2(n)]

# 3ª solución
# ===========

# reducido(m, x) es el resultado de dividir repetidamente m por x,
# mientras sea divisible. Por ejemplo,
#    reducido(36, 2)  ==  9
def reducido(m: int, x: int) -> int:
    if m % x == 0:
        return reducido(m // x, x)
    return m

def divisoresPrimos3(n: int) -> list[int]:
    if n % 2 == 0:
        return [2] + divisoresPrimos3(reducido(n, 2))

    def aux(m: int, xs: list[int]) -> list[int]:
        if m == 1:
            return []
        if xs == []:
            return []
        if m % xs[0] == 0:
            return [xs[0]] + aux(reducido(m, xs[0]), xs[1:])
        return aux(m, xs[1:])
```

```
    return aux(n, list(range(3, n + 1, 2)))

# 4ª solución
# ===========

def divisoresPrimos4(x: int) -> list[int]:
    return [n for n in divisors(x) if isprime(n)]

# 5ª solución
# ===========

def divisoresPrimos5(n: int) -> list[int]:
    return primefactors(n)

# Comprobación de equivalencia
# ============================

# La propiedad es
@given(st.integers(min_value=2, max_value=1000))
def test_divisoresPrimos(n: int) -> None:
    assert divisoresPrimos1(n) ==\
           divisoresPrimos2(n) ==\
           divisoresPrimos3(n) ==\
           divisoresPrimos4(n) ==\
           divisoresPrimos5(n)

# La comprobación es
#    src> poetry run pytest -q divisores_primos.py
#    1 passed in 0.70s

# Comparación de eficiencia
# =========================

def tiempo(e: str) -> None:
    """Tiempo (en segundos) de evaluar la expresión e."""
    t = Timer(e, "", default_timer, globals()).timeit(1)
    print(f"{t:0.2f} segundos")

def producto(xs: list[int]) -> int:
    return reduce(mul, xs)
```

```
# La comparación es
#    >>> tiempo('divisoresPrimos1(producto(list(range(1, 12))))')
#    11.14 segundos
#    >>> tiempo('divisoresPrimos2(producto(list(range(1, 12))))')
#    0.03 segundos
#    >>> tiempo('divisoresPrimos3(producto(list(range(1, 12))))')
#    0.00 segundos
#    >>> tiempo('divisoresPrimos4(producto(list(range(1, 12))))')
#    0.00 segundos
#    >>> tiempo('divisoresPrimos5(producto(list(range(1, 12))))')
#    0.00 segundos
#
#    >>> tiempo('divisoresPrimos2(producto(list(range(1, 17))))')
#    14.21 segundos
#    >>> tiempo('divisoresPrimos3(producto(list(range(1, 17))))')
#    0.00 segundos
#    >>> tiempo('divisoresPrimos4(producto(list(range(1, 17))))')
#    0.01 segundos
#    >>> tiempo('divisoresPrimos5(producto(list(range(1, 17))))')
#    0.00 segundos
#
#    >>> tiempo('divisoresPrimos3(producto(list(range(1, 32))))')
#    0.00 segundos
#    >>> tiempo('divisoresPrimos4(producto(list(range(1, 32))))')
#    4.59 segundos
#    >>> tiempo('divisoresPrimos5(producto(list(range(1, 32))))')
#    0.00 segundos
#
#    >>> tiempo('divisoresPrimos3(producto(list(range(1, 10001))))')
#    3.00 segundos
#    >>> tiempo('divisoresPrimos5(producto(list(range(1, 10001))))')
#    0.24 segundos
```

2.8. Números libres de cuadrados

En Haskell

```
-- ---------------------------------------------------------------------
-- Un número es libre de cuadrados si no es divisible por el cuadrado de
```

```haskell
-- ningún entero mayor que 1. Por ejemplo, 70 es libre de cuadrado
-- porque sólo es divisible por 1, 2, 5, 7 y 70; en cambio, 40 no es
-- libre de cuadrados porque es divisible por 2^2.
--
-- Definir la función
--    libreDeCuadrados :: Integer -> Bool
-- tal que (libreDeCuadrados x) se verifica si x es libre de cuadrados.
-- Por ejemplo,
--    libreDeCuadrados 70  ==  True
--    libreDeCuadrados 40  ==  False
--    libreDeCuadrados (product (take 30000 primes))  ==  True
-- ---------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-type-defaults #-}
{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module Numeros_libres_de_cuadrados where

import Data.List (nub)
import Data.Numbers.Primes (primeFactors, primes)
import Test.QuickCheck

-- 1ª solución
-- ===========

libreDeCuadrados1 :: Integer -> Bool
libreDeCuadrados1 n =
  null [x | x <- [2..n], rem n (x^2) == 0]

-- 2ª solución
-- ===========

libreDeCuadrados2 :: Integer -> Bool
libreDeCuadrados2 x =
  x == product (divisoresPrimos2 x)

-- (divisoresPrimos x) es la lista de los divisores primos de x. Por
-- ejemplo,
--    divisoresPrimos 40 == [2,5]
--    divisoresPrimos 70 == [2,5,7]
```

```
divisoresPrimos2 :: Integer -> [Integer]
divisoresPrimos2 x = [n | n <- divisores2 x, primo2 n]

-- (divisores n) es la lista de los divisores del número n. Por ejemplo,
--    divisores 25  ==  [1,5,25]
--    divisores 30  ==  [1,2,3,5,6,10,15,30]
divisores2 :: Integer -> [Integer]
divisores2 n = [x | x <- [1..n], n `mod` x == 0]

-- (primo n) se verifica si n es primo. Por ejemplo,
--    primo 30  == False
--    primo 31  == True
primo2 :: Integer -> Bool
primo2 n = divisores2 n == [1, n]

-- 3ª solución
-- ===========

libreDeCuadrados3 :: Integer -> Bool
libreDeCuadrados3 n
  | even n = n `mod` 4 /= 0 && libreDeCuadrados3 (n `div` 2)
  | otherwise = aux n [3,5..n]
  where aux 1 _  = True
        aux _ [] = True
        aux m (x:xs)
          | m `mod` x == 0 = m `mod` (x^2) /= 0 && aux (m `div` x) xs
          | otherwise      = aux m xs

-- 4ª solución
-- ===========

libreDeCuadrados4 :: Integer -> Bool
libreDeCuadrados4 x =
  x == product (divisoresPrimos4 x)

divisoresPrimos4 :: Integer -> [Integer]
divisoresPrimos4 = nub . primeFactors

-- 5ª solución
-- ===========
```

```
libreDeCuadrados5 :: Integer -> Bool
libreDeCuadrados5 =
  sinRepetidos . primeFactors

-- (sinRepetidos xs) se verifica si xs no tiene elementos repetidos. Por
-- ejemplo,
--    sinRepetidos [3,2,5]   ==  True
--    sinRepetidos [3,2,5,2]  ==  False
sinRepetidos :: [Integer] -> Bool
sinRepetidos xs =
  nub xs == xs

-- Comprobación de equivalencia
-- ============================

-- La propiedad es
prop_libreDeCuadrados :: Integer -> Property
prop_libreDeCuadrados x =
  x > 1 ==>
  all (== libreDeCuadrados1 x)
      [libreDeCuadrados2 x,
       libreDeCuadrados3 x,
       libreDeCuadrados4 x,
       libreDeCuadrados5 x]

-- La comprobación es
--    λ> quickCheck prop_libreDeCuadrados
--    +++ OK, passed 100 tests; 165 discarded.

-- Comparación de eficiencia
-- =========================

-- La comparación es
--    λ> libreDeCuadrados1 9699690
--    True
--    (8.54 secs, 6,441,144,248 bytes)
--    λ> libreDeCuadrados2 9699690
--    True
--    (4.78 secs, 1,940,781,632 bytes)
```

```
--    λ> libreDeCuadrados3 9699690
--    True
--    (0.01 secs, 561,400 bytes)
--    λ> libreDeCuadrados4 9699690
--    True
--    (0.01 secs, 568,160 bytes)
--    λ> libreDeCuadrados5 9699690
--    True
--    (0.01 secs, 567,536 bytes)
--
--    λ> libreDeCuadrados3 (product (take 30000 primes))
--    True
--    (2.30 secs, 2,369,316,208 bytes)
--    λ> libreDeCuadrados4 (product (take 30000 primes))
--    True
--    (6.68 secs, 4,565,617,408 bytes)
--    λ> libreDeCuadrados5 (product (take 30000 primes))
--    True
--    (5.54 secs, 3,411,701,752 bytes)
```

En Python

```python
# ---------------------------------------------------------------------
# Un número es libre de cuadrados si no es divisible por el cuadrado de
# ningún entero mayor que 1. Por ejemplo, 70 es libre de cuadrado
# porque sólo es divisible por 1, 2, 5, 7 y 70; en cambio, 40 no es
# libre de cuadrados porque es divisible por 2^2.
#
# Definir la función
#    libreDeCuadrados : (int) -> bool
# tal que (ibreDeCuadrados(x) se verifica si x es libre de cuadrados.
# Por ejemplo,
#    libreDeCuadrados(70)  ==  True
#    libreDeCuadrados(40)  ==  False
# ---------------------------------------------------------------------

from sys import setrecursionlimit
from timeit import Timer, default_timer

from hypothesis import given
```

```
from hypothesis import strategies as st
from sympy import primefactors, primerange

setrecursionlimit(10**6)

# 1ª solución
# ===========

def libreDeCuadrados1(n: int) -> bool:
    return [x for x in range(2, n + 2) if n % (x**2) == 0] == []

# 2ª solución
# ===========

# divisores(n) es la lista de los divisores del número n. Por ejemplo,
#    divisores(30)  ==  [1,2,3,5,6,10,15,30]
def divisores1(n: int) -> list[int]:
    return [x for x in range(1, n + 1) if n % x == 0]

# primo(n) se verifica si n es primo. Por ejemplo,
#    primo(30)  == False
#    primo(31)  == True
def primo1(n: int) -> bool:
    return divisores1(n) == [1, n]

# divisoresPrimos(x) es la lista de los divisores primos de x. Por
# ejemplo,
#    divisoresPrimos(40) == [2, 5]
#    divisoresPrimos(70) == [2, 5, 7]
def divisoresPrimos1(x: int) -> list[int]:
    return [n for n in divisores1(x) if primo1(n)]

# producto(xs) es el producto de los elementos de xs. Por ejemplo,
#    producto([3, 2, 5])  ==  30
def producto(xs: list[int]) -> int:
    if xs:
        return xs[0] * producto(xs[1:])
    return 1

def libreDeCuadrados2(x: int) -> bool:
```

```
    return x == producto(divisoresPrimos1(x))

# 3ª solución
# ===========

def libreDeCuadrados3(n: int) -> bool:
    if n % 2 == 0:
        return n % 4 != 0 and libreDeCuadrados3(n // 2)

    def aux(m: int, xs: list[int]) -> bool:
        if m == 1:
            return True
        if xs == []:
            return True
        if m % xs[0] == 0:
            return m % (xs[0]**2) != 0 and aux(m // xs[0], xs[1:])
        return aux(m, xs[1:])
    return aux(n, list(range(3, n + 1, 2)))

# 4ª solución
# ===========

def libreDeCuadrados4(x: int) -> bool:
    return x == producto(primefactors(x))

# Comprobación de equivalencia
# ============================

# La propiedad es
@given(st.integers(min_value=2, max_value=1000))
def test_libreDeCuadrados(n: int) -> None:
    assert libreDeCuadrados1(n) ==\
           libreDeCuadrados2(n) ==\
           libreDeCuadrados3(n) ==\
           libreDeCuadrados4(n)

# La comprobación es
#    src> poetry run pytest -q numeros_libres_de_cuadrados.py
#    1 passed in 0.59s
```

```python
# Comparación de eficiencia
# =========================

def tiempo(e: str) -> None:
    """Tiempo (en segundos) de evaluar la expresión e."""
    t = Timer(e, "", default_timer, globals()).timeit(1)
    print(f"{t:0.2f} segundos")

# La comparación es
#    >>> tiempo('libreDeCuadrados1(9699690)')
#    2.66 segundos
#    >>> tiempo('libreDeCuadrados2(9699690)')
#    2.58 segundos
#    >>> tiempo('libreDeCuadrados3(9699690)')
#    0.00 segundos
#    >>> tiempo('libreDeCuadrados4(9699690)')
#    0.00 segundos
#
#    >>> n = producto(list(primerange(1, 25000)))
#    >>> tiempo('libreDeCuadrados3(n)')
#    0.42 segundos
#    >>> tiempo('libreDeCuadrados4(n)')
#    0.14 segundos
```

2.9. Suma de los primeros números naturales

En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Definir la función
--    suma :: Integer -> Integer
-- tal (suma n) es la suma de los n primeros números. Por ejemplo,
--    suma 3  ==  6
--    length (show (suma (10^100)))  ==  200
-- ---------------------------------------------------------------------

module Suma_de_los_primeros_numeros_naturales where

import Data.List (foldl')
import Test.QuickCheck
```

```
-- 1ª solución
-- ===========

suma1 :: Integer -> Integer
suma1 n = sum [1..n]

-- 2ª solución
-- ===========

suma2 :: Integer -> Integer
suma2 n = (1+n)*n `div` 2

-- 3ª solución
-- ===========

suma3 :: Integer -> Integer
suma3 1 = 1
suma3 n = n + suma3 (n-1)

-- 4ª solución
-- ===========

suma4 :: Integer -> Integer
suma4 n = foldl (+) 0 [0..n]

-- 5ª solución
-- ===========

suma5 :: Integer -> Integer
suma5 n = foldl' (+) 0 [0..n]

-- Comprobación de equivalencia
-- ============================

-- La propiedad es
prop_suma :: Positive Integer -> Bool
prop_suma (Positive n) =
  all (== suma1 n)
      [suma2 n,
```

```
       suma3 n,
       suma4 n,
       suma5 n]

-- La comprobación es
--    λ> quickCheck prop_suma
--    +++ OK, passed 100 tests.

-- Comparación de eficiencia
-- =========================

-- La comparación es
--    λ> suma1 (5*10^6)
--    12500002500000
--    (1.23 secs, 806,692,792 bytes)
--    λ> suma2 (5*10^6)
--    12500002500000
--    (0.02 secs, 559,064 bytes)
--    λ> suma3 (5*10^6)
--    12500002500000
--    (3.06 secs, 1,214,684,352 bytes)
--    λ> suma4 (5*10^6)
--    12500002500000
--    (1.25 secs, 806,692,848 bytes)
--    λ> suma5 (5*10^6)
--    12500002500000
--    (0.26 secs, 440,559,048 bytes)
```

En Python

```
# ---------------------------------------------------------------------
# Definir la función
#    suma : (int) -> int
# tal suma(n) es la suma de los n primeros números. Por ejemplo,
#    suma(3)  ==  6
#    len(str(suma2(10**100)))  ==  200
# ---------------------------------------------------------------------

from functools import reduce
from operator import add
```

```python
from sys import setrecursionlimit
from timeit import Timer, default_timer

from hypothesis import given
from hypothesis import strategies as st

setrecursionlimit(10**8)

# 1ª solución
# ===========

def suma1(n: int) -> int:
    return sum(range(1, n + 1))

# 2ª solución
# ===========

def suma2(n: int) -> int:
    return (1 + n) * n // 2

# 3ª solución
# ===========

def suma3(n: int) -> int:
    if n == 1:
        return 1
    return n + suma3(n - 1)

# 4ª solución
# ===========

def suma4(n: int) -> int:
    return reduce(add, range(1, n + 1))

# 5ª solución
# ===========

def suma5(n: int) -> int:
    x, r = 1, 0
    while x <= n:
```

```
        r = r + x
        x = x + 1
    return r

# 6ª solución
# ===========

def suma6(n: int) -> int:
    r = 0
    for x in range(1, n + 1):
        r = r + x
    return r

# Comprobación de equivalencia
# ============================

# La propiedad es
@given(st.integers(min_value=1, max_value=1000))
def test_suma(n: int) -> None:
    r = suma1(n)
    assert suma2(n) == r
    assert suma3(n) == r
    assert suma4(n) == r
    assert suma5(n) == r
    assert suma6(n) == r

# La comprobación es
#    src> poetry run pytest -q suma_de_los_primeros_numeros_naturales.py
#    1 passed in 0.16s

# Comparación de eficiencia
# =========================

def tiempo(e: str) -> None:
    """Tiempo (en segundos) de evaluar la expresión e."""
    t = Timer(e, "", default_timer, globals()).timeit(1)
    print(f"{t:0.2f} segundos")

# La comparación es
#    >>> tiempo('suma1(20000)')
```

```
#    0.00 segundos
#    >>> tiempo('suma2(20000)')
#    0.00 segundos
#    >>> tiempo('suma3(20000)')
#    0.02 segundos
#    >>> tiempo('suma4(20000)')
#    0.00 segundos
#    >>> tiempo('suma5(20000)')
#    0.01 segundos
#    >>> tiempo('suma6(20000)')
#    0.00 segundos
#
#    >>> tiempo('suma1(10**8)')
#    1.55 segundos
#    >>> tiempo('suma2(10**8)')
#    0.00 segundos
#    >>> tiempo('suma4(10**8)')
#    3.69 segundos
#    >>> tiempo('suma5(10**8)')
#    7.04 segundos
#    >>> tiempo('suma6(10**8)')
#    4.23 segundos
```

2.10. Suma de los cuadrados de los primeros números naturales

En Haskell

```
-- ---------------------------------------------------------------------
-- Definir la función
--    sumaDeCuadrados :: Integer -> Integer
-- tal que (sumaDeCuadrados n) es la suma de los cuadrados de los
-- primeros n números; es decir, 1^2 + 2^2 + ... + n^2. Por ejemplo,
--    sumaDeCuadrados 3    ==  14
--    sumaDeCuadrados 100  ==  338350
--    length (show (sumaDeCuadrados (10^100)))  ==  300
-- ---------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-type-defaults #-}
```

```haskell
module Suma_de_los_cuadrados_de_los_primeros_numeros_naturales where

import Data.List (foldl')
import Test.QuickCheck

-- 1ª solución
-- ===========

sumaDeCuadrados1 :: Integer -> Integer
sumaDeCuadrados1 n = sum [x^2 | x <- [1..n]]

-- 2ª solución
-- ===========

sumaDeCuadrados2 :: Integer -> Integer
sumaDeCuadrados2 n = n*(n+1)*(2*n+1) `div` 6

-- 3ª solución
-- ===========

sumaDeCuadrados3 :: Integer -> Integer
sumaDeCuadrados3 1 = 1
sumaDeCuadrados3 n = n^2 + sumaDeCuadrados3 (n-1)

-- 4ª solución
-- ===========

sumaDeCuadrados4 :: Integer -> Integer
sumaDeCuadrados4 n = foldl (+) 0 (map (^2) [0..n])

-- 5ª solución
-- ===========

sumaDeCuadrados5 :: Integer -> Integer
sumaDeCuadrados5 n = foldl' (+) 0 (map (^2) [0..n])

-- Comprobación de equivalencia
-- ============================
```

```
-- La propiedad es
prop_sumaDeCuadrados :: Positive Integer -> Bool
prop_sumaDeCuadrados (Positive n) =
  all (== sumaDeCuadrados1 n)
      [sumaDeCuadrados2 n,
       sumaDeCuadrados3 n,
       sumaDeCuadrados4 n,
       sumaDeCuadrados5 n]

-- La comprobación es
--    λ> quickCheck prop_sumaDeCuadrados
--    +++ OK, passed 100 tests.

-- Comparación de eficiencia
-- =========================

-- La comparación es
--    λ> sumaDeCuadrados1 (2*10^6)
--    2666668666667000000
--    (1.90 secs, 1,395,835,576 bytes)
--    λ> sumaDeCuadrados2 (2*10^6)
--    2666668666667000000
--    (0.01 secs, 563,168 bytes)
--    λ> sumaDeCuadrados3 (2*10^6)
--    2666668666667000000
--    (2.37 secs, 1,414,199,400 bytes)
--    λ> sumaDeCuadrados4 (2*10^6)
--    2666668666667000000
--    (1.33 secs, 1,315,836,128 bytes)
--    λ> sumaDeCuadrados5 (2*10^6)
--    2666668666667000000
--    (0.71 secs, 1,168,563,384 bytes)
```

En Python

```
# ---------------------------------------------------------------------
# Definir la función
#    sumaDeCuadrados : (int) -> int
# tal sumaDeCuadrados(n) es la suma de los xuadrados de los n primeros
# números naturales. Por ejemplo,
```

```
#    sumaDeCuadrados(3)   ==  14
#    sumaDeCuadrados(100) ==  338350
#    len(str(sumaDeCuadrados(10**100)))  ==  300
# ---------------------------------------------------------------------

from functools import reduce
from operator import add
from sys import setrecursionlimit
from timeit import Timer, default_timer

from hypothesis import given
from hypothesis import strategies as st

setrecursionlimit(10**6)

# 1ª solución
# ===========

def sumaDeCuadrados1(n: int) -> int:
    return sum(x**2 for x in range(1, n + 1))

# 2ª solución
# ===========

def sumaDeCuadrados2(n: int) -> int:
    return n * (n + 1) * (2 * n + 1) // 6

# 3ª solución
# ===========

def sumaDeCuadrados3(n: int) -> int:
    if n == 1:
        return 1
    return n**2 + sumaDeCuadrados3(n - 1)

# 4ª solución
# ===========

def sumaDeCuadrados4(n: int) -> int:
    return reduce(add, (x**2 for x in range(1, n + 1)))
```

```
# 5ª solución
# ===========

def sumaDeCuadrados5(n: int) -> int:
    x, r = 1, 0
    while x <= n:
        r = r + x**2
        x = x + 1
    return r

# 6ª solución
# ===========

def sumaDeCuadrados6(n: int) -> int:
    r = 0
    for x in range(1, n + 1):
        r = r + x**2
    return r

# Comprobación de equivalencia
# ============================

# La propiedad es
@given(st.integers(min_value=1, max_value=1000))
def test_sumaDeCuadrados(n: int) -> None:
    r = sumaDeCuadrados1(n)
    assert sumaDeCuadrados2(n) == r
    assert sumaDeCuadrados3(n) == r
    assert sumaDeCuadrados4(n) == r
    assert sumaDeCuadrados5(n) == r
    assert sumaDeCuadrados6(n) == r

# La comprobación es
#    src> poetry run pytest -q suma_de_los_cuadrados_de_los_primeros_numeros_natu
#    1 passed in 0.19s

# Comparación de eficiencia
# =========================
```

```
def tiempo(e: str) -> None:
    """Tiempo (en segundos) de evaluar la expresión e."""
    t = Timer(e, "", default_timer, globals()).timeit(1)
    print(f"{t:0.2f} segundos")

# La comparación es
#    >>> tiempo('sumaDeCuadrados1(20000)')
#    0.01 segundos
#    >>> tiempo('sumaDeCuadrados2(20000)')
#    0.00 segundos
#    >>> tiempo('sumaDeCuadrados3(20000)')
#    0.02 segundos
#    >>> tiempo('sumaDeCuadrados4(20000)')
#    0.02 segundos
#    >>> tiempo('sumaDeCuadrados5(20000)')
#    0.02 segundos
#    >>> tiempo('sumaDeCuadrados6(20000)')
#    0.02 segundos
#
#    >>> tiempo('sumaDeCuadrados1(10**7)')
#    2.19 segundos
#    >>> tiempo('sumaDeCuadrados2(10**7)')
#    0.00 segundos
#    >>> tiempo('sumaDeCuadrados4(10**7)')
#    2.48 segundos
#    >>> tiempo('sumaDeCuadrados5(10**7)')
#    2.53 segundos
#    >>> tiempo('sumaDeCuadrados6(10**7)')
#    2.22 segundos
```

2.11. Suma de cuadrados menos cuadrado de la suma

En Haskell

```
-- ---------------------------------------------------------------------
-- Definir la función
--    euler6 :: Integer -> Integer
-- tal que (euler6 n) es la diferencia entre el cuadrado de la suma
```

```haskell
-- de los n primeros números y la suma de los cuadrados de los n
-- primeros números. Por ejemplo,
--    euler6 10       ==  2640
--    euler6 (10^10)  ==  2500000000166666666641666666665000000000
--
-- Nota: Este ejercicio está basado en el problema 6 del proyecto Euler
-- https://www.projecteuler.net/problem=6
-- ---------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-type-defaults #-}

module Suma_de_cuadrados_menos_cuadrado_de_la_suma where

import Data.List (foldl')
import Test.QuickCheck

-- 1ª solución
-- ===========

euler6a :: Integer -> Integer
euler6a n = (suma1 n)^2 - sumaDeCuadrados1 n

-- (suma n) es la suma de los n primeros números. Por ejemplo,
--    suma 3  ==  6
suma1 :: Integer -> Integer
suma1 n = sum [1..n]

-- (sumaDeCuadrados n) es la suma de los cuadrados de los
-- primeros n números; es decir, 1^2 + 2^2 + ... + n^2. Por ejemplo,
--    sumaDeCuadrados 3    ==  14
--    sumaDeCuadrados 100  ==  338350

sumaDeCuadrados1 :: Integer -> Integer
sumaDeCuadrados1 n = sum [x^2 | x <- [1..n]]

-- 2ª solución
-- ===========

euler6b :: Integer -> Integer
euler6b n = (suma2 n)^ 2 - sumaDeCuadrados2 n
```

```haskell
suma2 :: Integer -> Integer
suma2 n = (1+n)*n `div` 2

sumaDeCuadrados2 :: Integer -> Integer
sumaDeCuadrados2 n = n*(n+1)*(2*n+1) `div` 6

-- 3ª solución
-- ===========

euler6c :: Integer -> Integer
euler6c n = (suma3 n)^ 2 - sumaDeCuadrados3 n

suma3 :: Integer -> Integer
suma3 1 = 1
suma3 n = n + suma3 (n-1)

sumaDeCuadrados3 :: Integer -> Integer
sumaDeCuadrados3 1 = 1
sumaDeCuadrados3 n = n^2 + sumaDeCuadrados3 (n-1)

-- 4ª solución
-- ===========

euler6d :: Integer -> Integer
euler6d n = (suma4 n)^ 2 - sumaDeCuadrados4 n

suma4 :: Integer -> Integer
suma4 n = foldl (+) 0 [0..n]

sumaDeCuadrados4 :: Integer -> Integer
sumaDeCuadrados4 n = foldl (+) 0 (map (^2) [0..n])

-- 5ª solución
-- ===========

euler6e :: Integer -> Integer
euler6e n = (suma5 n)^ 2 - sumaDeCuadrados5 n

suma5 :: Integer -> Integer
```

```
suma5 n = foldl' (+) 0 [0..n]

sumaDeCuadrados5 :: Integer -> Integer
sumaDeCuadrados5 n = foldl' (+) 0 (map (^2) [0..n])

-- Comprobación de equivalencia
-- ============================

-- La propiedad es
prop_euler6 :: Positive Integer -> Bool
prop_euler6 (Positive n) =
  all (== euler6a n)
      [euler6b n,
       euler6c n,
       euler6d n,
       euler6e n]

-- La comprobación es
--    λ> quickCheck prop_euler6
--    +++ OK, passed 100 tests.

-- Comparación de eficiencia
-- =========================

-- La comparación es
--    λ> euler6a (3*10^6)
--    20250004499997749999500000
--    (3.32 secs, 2,577,174,640 bytes)
--    λ> euler6b (3*10^6)
--    20250004499997749999500000
--    (0.01 secs, 569,288 bytes)
--    λ> euler6c (3*10^6)
--    20250004499997749999500000
--    (5.60 secs, 2,849,479,288 bytes)
--    λ> euler6d (3*10^6)
--    20250004499997749999500000
--    (2.52 secs, 2,457,175,248 bytes)
--    λ> euler6e (3*10^6)
--    20250004499997749999500000
--    (1.08 secs, 2,016,569,472 bytes)
```

```
--
--    λ> euler6a (10^7)
--    2500000166666641666665000000
--    (11.14 secs, 8,917,796,648 bytes)
--    λ> euler6b (10^7)
--    2500000166666641666665000000
--    (0.01 secs, 570,752 bytes)
--    λ> euler6c (10^7)
--    *** Exception: stack overflow
--    λ> euler6d (10^7)
--    2500000166666641666665000000
--    (9.47 secs, 8,517,796,760 bytes)
--    λ> euler6e (10^7)
--    2500000166666641666665000000
--    (3.78 secs, 7,049,100,104 bytes)
```

En Python

```
# ---------------------------------------------------------------------
# Definir la función
#    euler6 : (int) -> int
# tal que euler6(n) es la diferencia entre el cuadrado de la suma
# de los n primeros números y la suma de los cuadrados de los n
# primeros números. Por ejemplo,
#    euler6(10)       ==  2640
#    euler6(10^10)  ==  2500000000166666666641666666665000000000
#
# Nota: Este ejercicio está basado en el problema 6 del proyecto Euler
# https://www.projecteuler.net/problem=6
# ---------------------------------------------------------------------

from functools import reduce
from operator import add
from sys import setrecursionlimit
from timeit import Timer, default_timer

from hypothesis import given
from hypothesis import strategies as st

setrecursionlimit(10**6)
```

```python
# 1ª solución
# ===========

def euler6a(n: int) -> int:
    return suma1(n)**2 - sumaDeCuadrados1(n)

# suma(n) es la suma de los n primeros números. Por ejemplo,
#    suma(3)  ==  6
def suma1(n: int) -> int:
    return sum(range(1, n + 1))

# sumaDeCuadrados(n) es la suma de los cuadrados de los
# primeros n números; es decir, 1^2 + 2^2 + ... + n^2. Por ejemplo,
#    sumaDeCuadrados(3)    ==  14
#    sumaDeCuadrados(100)  ==  338350
def sumaDeCuadrados1(n: int) -> int:
    return sum(x**2 for x in range(1, n + 1))

# 2ª solución
# ===========

def euler6b(n: int) -> int:
    return suma2(n)**2 - sumaDeCuadrados2(n)

def suma2(n: int) -> int:
    return (1 + n) * n // 2

def sumaDeCuadrados2(n: int) -> int:
    return n * (n + 1) * (2 * n + 1) // 6

# 3ª solución
# ===========

def euler6c(n: int) -> int:
    return suma3(n)**2 - sumaDeCuadrados3(n)

def suma3(n: int) -> int:
    if n == 1:
        return 1
```

```
    return n + suma3(n - 1)

def sumaDeCuadrados3(n: int) -> int:
    if n == 1:
        return 1
    return n**2 + sumaDeCuadrados3(n - 1)

# 4ª solución
# ===========

def euler6d(n: int) -> int:
    return suma4(n)**2 - sumaDeCuadrados4(n)

def suma4(n: int) -> int:
    return reduce(add, range(1, n + 1))

def sumaDeCuadrados4(n: int) -> int:
    return reduce(add, (x**2 for x in range(1, n + 1)))

# 5ª solución
# ===========

def euler6e(n: int) -> int:
    return suma5(n)**2 - sumaDeCuadrados5(n)

def suma5(n: int) -> int:
    x, r = 1, 0
    while x <= n:
        r = r + x
        x = x + 1
    return r

def sumaDeCuadrados5(n: int) -> int:
    x, r = 1, 0
    while x <= n:
        r = r + x**2
        x = x + 1
    return r

# 6ª solución
```

```python
# ===========

def euler6f(n: int) -> int:
    return suma6(n)**2 - sumaDeCuadrados6(n)

def suma6(n: int) -> int:
    r = 0
    for x in range(1, n + 1):
        r = r + x
    return r

def sumaDeCuadrados6(n: int) -> int:
    r = 0
    for x in range(1, n + 1):
        r = r + x**2
    return r

# Comprobación de equivalencia
# ============================

# La propiedad es
@given(st.integers(min_value=1, max_value=1000))
def test_euler6(n: int) -> None:
    r = euler6a(n)
    assert euler6b(n) == r
    assert euler6c(n) == r
    assert euler6d(n) == r
    assert euler6e(n) == r
    assert euler6f(n) == r

# La comprobación es
#    src> poetry run pytest -q suma_de_cuadrados_menos_cuadrado_de_la_suma.py
#    1 passed in 0.21s

# Comparación de eficiencia
# =========================

def tiempo(e: str) -> None:
    """Tiempo (en segundos) de evaluar la expresión e."""
    t = Timer(e, "", default_timer, globals()).timeit(1)
```

```
    print(f"{t:0.2f} segundos")

# La comparación es
#    >>> tiempo('euler6a(20000)')
#    0.02 segundos
#    >>> tiempo('euler6b(20000)')
#    0.00 segundos
#    >>> tiempo('euler6c(20000)')
#    0.02 segundos
#    >>> tiempo('euler6d(20000)')
#    0.01 segundos
#    >>> tiempo('euler6e(20000)')
#    0.01 segundos
#    >>> tiempo('euler6f(20000)')
#    0.01 segundos
#
#    >>> tiempo('euler6a(10**7)')
#    2.26 segundos
#    >>> tiempo('euler6b(10**7)')
#    0.00 segundos
#    >>> tiempo('euler6d(10**7)')
#    2.58 segundos
#    >>> tiempo('euler6e(10**7)')
#    2.89 segundos
#    >>> tiempo('euler6f(10**7)')
#    2.45 segundos
```

2.12. Triángulo aritmético

En Haskell

```
-- ---------------------------------------------------------------------
-- Los triángulos aritméticos se forman como sigue
--      1
--      2  3
--      4  5  6
--      7  8  9 10
--     11 12 13 14 15
--     16 17 18 19 20 21
--
```

```
-- Definir las funciones
--    linea     :: Integer -> [Integer]
--    triangulo :: Integer -> [[Integer]]
-- tales que
-- + (linea n) es la línea n-ésima de los triángulos aritméticos. Por
--   ejemplo,
--      linea 4  ==  [7,8,9,10]
--      linea 5  ==  [11,12,13,14,15]
--      head (linea (10^20)) == 4999999999999999999950000000000000000001
-- + (triangulo n) es el triángulo aritmético de altura n. Por ejemplo,
--      triangulo 3  ==  [[1],[2,3],[4,5,6]]
--      triangulo 4  ==  [[1],[2,3],[4,5,6],[7,8,9,10]]
-- ---------------------------------------------------------------------

module Triangulo_aritmetico where

import Test.QuickCheck

-- 1ª definición de línea
-- ======================

linea1 :: Integer -> [Integer]
linea1 n = [suma1 (n-1)+1..suma1 n]

-- (suma n) es la suma de los n primeros números. Por ejemplo,
--    suma 3  ==  6
suma1 :: Integer -> Integer
suma1 n = sum [1..n]

-- 2ª definición de línea
-- ======================

linea2 :: Integer -> [Integer]
linea2 n = [s+1..s+n]
  where s = suma1 (n-1)

-- 3ª definición de línea
-- ======================

linea3 :: Integer -> [Integer]
```

```haskell
linea3 n = [s+1..s+n]
  where s = suma2 (n-1)

suma2 :: Integer -> Integer
suma2 n = (1+n)*n `div` 2

-- Comprobación de equivalencia de linea
-- =====================================

-- La propiedad es
prop_linea :: Positive Integer -> Bool
prop_linea (Positive n) =
  all (== linea1 n)
      [linea2 n,
       linea3 n]

-- La comprobación es
--    λ> quickCheck prop_linea
--    +++ OK, passed 100 tests.

-- Comparación de eficiencia de linea
-- ==================================

-- La comparación es
--    λ> last (linea1 (10^7))
--    50000005000000
--    (5.10 secs, 3,945,159,856 bytes)
--    λ> last (linea2 (10^7))
--    50000005000000
--    (3.11 secs, 2,332,859,512 bytes)
--    λ> last (linea3 (10^7))
--    50000005000000
--    (0.16 secs, 720,559,384 bytes)

-- 1ª definición de triangulo
-- ==========================

triangulo1 :: Integer -> [[Integer]]
triangulo1 n = [linea1 m | m <- [1..n]]
```

```
-- 2ª definición de triangulo
-- ==========================

triangulo2 :: Integer -> [[Integer]]
triangulo2 n = [linea2 m | m <- [1..n]]

-- 3ª definición de triangulo
-- ==========================

triangulo3 :: Integer -> [[Integer]]
triangulo3 n = [linea3 m | m <- [1..n]]

-- Comprobación de equivalencia de triangulo
-- =========================================

-- La propiedad es
prop_triangulo :: Positive Integer -> Bool
prop_triangulo (Positive n) =
  all (== triangulo1 n)
      [triangulo2 n,
       triangulo3 n]

-- La comprobación es
--    λ> quickCheck prop_triangulo
--    +++ OK, passed 100 tests.

-- Comparación de eficiencia de triangulo
-- ======================================

-- La comparación es
--    λ> last (last (triangulo1 (3*10^6)))
--    4500001500000
--    (2.25 secs, 1,735,919,184 bytes)
--    λ> last (last (triangulo2 (3*10^6)))
--    4500001500000
--    (1.62 secs, 1,252,238,872 bytes)
--    λ> last (last (triangulo3 (3*10^6)))
--    4500001500000
--    (0.79 secs, 768,558,776 bytes)
```

En Python

```
# ---------------------------------------------------------------------
# Los triángulos aritméticos se forman como sigue
#      1
#      2  3
#      4  5  6
#      7  8  9 10
#     11 12 13 14 15
#     16 17 18 19 20 21
#
# Definir las funciones
#    linea : (int) -> list[int]
#    triangulo : (int) -> list[list[int]]
# tales que
# + linea(n) es la línea n-ésima de los triángulos aritméticos. Por
#   ejemplo,
#      linea(4)  ==  [7, 8, 9, 10]
#      linea(5)  ==  [11, 12, 13, 14, 15]
#      linea(10**8)[0] == 4999999950000001
# + triangulo(n) es el triángulo aritmético de altura n. Por ejemplo,
#      triangulo(3)  ==  [[1], [2, 3], [4, 5, 6]]
#      triangulo(4)  ==  [[1], [2, 3], [4, 5, 6], [7, 8, 9, 10]]
# ---------------------------------------------------------------------

from timeit import Timer, default_timer

from hypothesis import given
from hypothesis import strategies as st

# 1ª definición de línea
# ======================

# suma(n) es la suma de los n primeros números. Por ejemplo,
#    suma(3)  ==  6
def suma1(n: int) -> int:
    return sum(range(1, n + 1))

def linea1(n: int) -> list[int]:
    return list(range(suma1(n - 1) + 1, suma1(n) + 1))
```

```
# 2ª definición de línea
# ======================

def linea2(n: int) -> list[int]:
    s = suma1(n-1)
    return list(range(s + 1, s + n + 1))

# 3ª definición de línea
# ======================

def suma2(n: int) -> int:
    return (1 + n) * n // 2

def linea3(n: int) -> list[int]:
    s = suma2(n-1)
    return list(range(s + 1, s + n + 1))

# Comprobación de equivalencia de linea
# =====================================

@given(st.integers(min_value=1, max_value=1000))
def test_suma(n: int) -> None:
    r = linea1(n)
    assert linea2(n) == r
    assert linea3(n) == r

# La comprobación es
#    src> poetry run pytest -q triangulo_aritmetico.py
#    1 passed in 0.15s

# Comparación de eficiencia
# =========================

def tiempo(e: str) -> None:
    """Tiempo (en segundos) de evaluar la expresión e."""
    t = Timer(e, "", default_timer, globals()).timeit(1)
    print(f"{t:0.2f} segundos")

# La comparación es
#    >>> tiempo('linea1(10**7)')
```

```
#    0.53 segundos
#    >>> tiempo('linea2(10**7)')
#    0.40 segundos
#    >>> tiempo('linea3(10**7)')
#    0.29 segundos

# 1ª definición de triangulo
# ==========================

def triangulo1(n: int) -> list[list[int]]:
    return [linea1(m) for m in range(1, n + 1)]

# 2ª definición de triangulo
# ==========================

def triangulo2(n: int) -> list[list[int]]:
    return [linea2(m) for m in range(1, n + 1)]

# 3ª definición de triangulo
# ==========================

def triangulo3(n: int) -> list[list[int]]:
    return [linea3(m) for m in range(1, n + 1)]

# Comprobación de equivalencia de triangulo
# =========================================

@given(st.integers(min_value=1, max_value=1000))
def test_triangulo(n: int) -> None:
    r = triangulo1(n)
    assert triangulo2(n) == r
    assert triangulo3(n) == r

# La comprobación es
#    src> poetry run pytest -q triangulo_aritmetico.py
#    1 passed in 3.44s

# Comparación de eficiencia de triangulo
# ======================================
#
```

```
# La comparación es
#    >>> tiempo('triangulo1(10**4)')
#    2.58 segundos
#    >>> tiempo('triangulo2(10**4)')
#    1.91 segundos
#    >>> tiempo('triangulo3(10**4)')
#    1.26 segundos
```

2.13. Suma de divisores

En Haskell

```
-- ---------------------------------------------------------------------
-- Definir la función
--    sumaDivisores :: Integer -> Integer
-- tal que (sumaDivisores x) es la suma de los divisores de x. Por ejemplo,
--    sumaDivisores 12                  ==  28
--    sumaDivisores 25                  ==  31
--    sumaDivisores (product [1..25])  ==  93383273455325195473152000
--    length (show (sumaDivisores (product [1..30000])))  ==  121289
--    maximum (map sumaDivisores [1..2*10^6])             ==  8851392
-- ---------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-incomplete-patterns #-}

module Suma_de_divisores where

import Data.List (foldl', genericLength, group, inits)
import Data.Set (toList)
import Data.Numbers.Primes (primeFactors)
import Math.NumberTheory.ArithmeticFunctions (divisors, sigma)
import Test.QuickCheck

-- 1ª solución
-- ===========

sumaDivisores1 :: Integer -> Integer
sumaDivisores1 n = sum (divisores1 n)

-- (divisores x) es la lista de los divisores de x. Por ejemplo,
```

```haskell
--    divisores 60  ==  [1,5,3,15,2,10,6,30,4,20,12,60]
divisores1 :: Integer -> [Integer]
divisores1 n = [x | x <- [1..n], n `rem` x == 0]

-- 2ª solución
-- ===========

-- Sustituyendo la definición de divisores de la solución anterior por
-- cada una de las del ejercicio [Divisores de un número](https://bit.ly/3S1HYwi)
-- Se obtiene una nueva definición de sumaDivisores. La usada en la
-- definición anterior es la menos eficiente y la que se usa en la
-- siguiente definición es la más eficiente.

sumaDivisores2 :: Integer -> Integer
sumaDivisores2 = sum . divisores2

divisores2 :: Integer -> [Integer]
divisores2 = toList . divisors

-- 3ª solución
-- ===========

-- La solución anterior se puede simplificar

sumaDivisores3 :: Integer -> Integer
sumaDivisores3 = sum . divisors

-- 4ª solución
-- ===========

sumaDivisores4 :: Integer -> Integer
sumaDivisores4 = foldl' (+) 0 . divisores2

-- 5ª solución
-- ===========

sumaDivisores5 :: Integer -> Integer
sumaDivisores5 n = aux [1..n]
  where aux [] = 0
        aux (x:xs) | n `rem` x == 0 = x + aux xs
```

```
                     | otherwise      = aux xs

-- 6ª solución
-- ===========

sumaDivisores6 :: Integer -> Integer
sumaDivisores6 = sum
               . map (product . concat)
               . mapM inits
               . group
               . primeFactors

-- 7ª solución
-- ===========

-- Si la descomposición de x en factores primos es
--    x = p(1)^e(1) . p(2)^e(2) . .... . p(n)^e(n)
-- entonces la suma de los divisores de x es
--    p(1)^(e(1)+1) - 1     p(2)^(e(2)+1) - 1       p(n)^(e(2)+1) - 1
--   ------------------- . ------------------- ... -------------------
--        p(1)-1                p(2)-1                  p(n)-1
-- Ver la demostración en http://bit.ly/2zUXZPc

sumaDivisores7 :: Integer -> Integer
sumaDivisores7 x =
  product [(p^(e+1)-1) `div` (p-1) | (p,e) <- factorizacion x]

-- (factorizacion x) es la lista de las bases y exponentes de la
-- descomposición prima de x. Por ejemplo,
--    factorizacion 600  ==  [(2,3),(3,1),(5,2)]
factorizacion :: Integer -> [(Integer,Integer)]
factorizacion = map primeroYlongitud . group . primeFactors

-- (primeroYlongitud xs) es el par formado por el primer elemento de xs
-- y la longitud de xs. Por ejemplo,
--    primeroYlongitud [3,2,5,7] == (3,4)
primeroYlongitud :: [a] -> (a,Integer)
primeroYlongitud (x:xs) =
  (x, 1 + genericLength xs)
```

```
-- 8ª solución
-- ===========

sumaDivisores8 :: Integer -> Integer
sumaDivisores8 = sigma 1

-- Comprobación de equivalencia
-- ============================

-- La propiedad es
prop_sumaDivisores :: Positive Integer -> Bool
prop_sumaDivisores (Positive x) =
  all (== sumaDivisores1 x)
      [ sumaDivisores2 x
      , sumaDivisores3 x
      , sumaDivisores4 x
      , sumaDivisores5 x
      , sumaDivisores6 x
      , sumaDivisores7 x
      , sumaDivisores8 x
      ]

-- La comprobación es
--    λ> quickCheck prop_sumaDivisores
--    +++ OK, passed 100 tests.

-- Comparación de eficiencia
-- =========================

-- La comparación es
--    λ> sumaDivisores1 5336100
--    21386001
--    (2.25 secs, 1,067,805,248 bytes)
--    λ> sumaDivisores2 5336100
--    21386001
--    (0.01 secs, 659,112 bytes)
--    λ> sumaDivisores3 5336100
--    21386001
--    (0.01 secs, 635,688 bytes)
--    λ> sumaDivisores4 5336100
```

```
--    21386001
--    (0.01 secs, 648,992 bytes)
--    λ> sumaDivisores5 5336100
--    21386001
--    (2.44 secs, 1,323,924,176 bytes)
--    λ> sumaDivisores6 5336100
--    21386001
--    (0.01 secs, 832,104 bytes)
--    λ> sumaDivisores7 5336100
--    21386001
--    (0.01 secs, 571,040 bytes)
--    λ> sumaDivisores8 5336100
--    21386001
--    (0.00 secs, 558,296 bytes)
--
--    λ> sumaDivisores2 251888923423315469521109880000000
--    1471072204661054993275791673480320
--    (2.30 secs, 1,130,862,080 bytes)
--    λ> sumaDivisores3 251888923423315469521109880000000
--    1471072204661054993275791673480320
--    (1.83 secs, 896,386,232 bytes)
--    λ> sumaDivisores4 251888923423315469521109880000000
--    1471072204661054993275791673480320
--    (1.52 secs, 997,992,328 bytes)
--    λ> sumaDivisores6 251888923423315469521109880000000
--    1471072204661054993275791673480320
--    (2.35 secs, 5,719,848,600 bytes)
--    λ> sumaDivisores7 251888923423315469521109880000000
--    1471072204661054993275791673480320
--    (0.00 secs, 628,136 bytes)
--    λ> sumaDivisores8 251888923423315469521109880000000
--    1471072204661054993275791673480320
--    (0.00 secs, 591,352 bytes)
--
--    λ> length (show (sumaDivisores7 (product [1..30000])))
--    121289
--    (2.76 secs, 4,864,576,304 bytes)
--    λ> length (show (sumaDivisores8 (product [1..30000])))
--    121289
--    (1.65 secs, 3,173,319,312 bytes)
```

En Python

```
# ---------------------------------------------------------------------
# Definir la función
#    sumaDivisores : (int) -> int
# tal que sumaDivisores(x) es la suma de los divisores de x. Por ejemplo,
#    sumaDivisores(12)                ==  28
#    sumaDivisores(25)                ==  31
#    sumaDivisores (reduce(mul, range(1, 26)))  ==  93383273455325195473152000
#    len(str(sumaDivisores6(reduce(mul, range(1, 30001)))))  ==  121289
# ---------------------------------------------------------------------

from functools import reduce
from operator import mul
from sys import setrecursionlimit
from timeit import Timer, default_timer

from hypothesis import given
from hypothesis import strategies as st
from sympy import divisor_sigma, divisors, factorint

setrecursionlimit(10**6)

# 1ª solución
# ===========

# divisores(x) es la lista de los divisores de x. Por ejemplo,
#    divisores(60)  ==  [1,5,3,15,2,10,6,30,4,20,12,60]
def divisores(n: int) -> list[int]:
    return [x for x in range(1, n + 1) if n % x == 0]

def sumaDivisores1(n: int) -> int:
    return sum(divisores(n))

# 2ª solución
# ===========

# Sustituyendo la definición de divisores de la solución anterior por
# cada una de las del ejercicio Divisores de un número https://bit.ly/3S1HYwi)
# Se obtiene una nueva definición de sumaDivisores. La usada en la
# definición anterior es la menos eficiente y la que se usa en la
```

```
# siguiente definición es la más eficiente.
def sumaDivisores2(n: int) -> int:
    return sum(divisors(n))

# 3ª solución
# ===========

def sumaDivisores3(n: int) -> int:
    def aux(xs: list[int]) -> int:
        if xs:
            if n % xs[0] == 0:
                return xs[0] + aux(xs[1:])
            return aux(xs[1:])
        return 0

    return aux(list(range(1, n + 1)))

# 4ª solución
# ===========

# Si la descomposición de x en factores primos es
#    x = p(1)^e(1) . p(2)^e(2) . .... . p(n)^e(n)
# entonces la suma de los divisores de x es
#    p(1)^(e(1)+1) - 1     p(2)^(e(2)+1) - 1       p(n)^(e(2)+1) - 1
#   ------------------- . ------------------- ... -------------------
#        p(1)-1                p(2)-1                  p(n)-1
# Ver la demostración en http://bit.ly/2zUXZPc

def sumaDivisores4(n: int) -> int:
    return reduce(mul, [(p ** (e + 1) - 1) // (p - 1)
                        for (p, e) in factorint(n).items()])

# 5ª solución
# ===========

def sumaDivisores5(n: int) -> int:
    x = 1
    r1 = 0
    r2 = 0
    while x * x < n:
```

```
        if n % x == 0:
            r1 += x
            r2 += n // x
        x += 1
    if x * x == n:
        r1 += x
    return r1 + r2

# 6ª solución
# ===========

def sumaDivisores6(n: int) -> int:
    return divisor_sigma(n, 1)

# Comprobación de equivalencia
# ============================

# La propiedad es
@given(st.integers(min_value=2, max_value=1000))
def test_sumaDivisores(n: int) -> None:
    r = sumaDivisores1(n)
    assert sumaDivisores2(n) == r
    assert sumaDivisores3(n) == r
    assert sumaDivisores4(n) == r
    assert sumaDivisores5(n) == r
    assert sumaDivisores6(n) == r

# La comprobación es
#    src> poetry run pytest -q suma_de_divisores.py
#    1 passed in 0.90s

# Comparación de eficiencia
# =========================

def tiempo(e: str) -> None:
    """Tiempo (en segundos) de evaluar la expresión e."""
    t = Timer(e, "", default_timer, globals()).timeit(1)
    print(f"{t:0.2f} segundos")

# La comparación es
```

```
#    >>> tiempo('sumaDivisores1(5336100)')
#    0.29 segundos
#    >>> tiempo('sumaDivisores2(5336100)')
#    0.00 segundos
#    >>> tiempo('sumaDivisores3(5336100)')
#    Process Python terminado (killed)
#    >>> tiempo('sumaDivisores4(5336100)')
#    0.00 segundos
#    >>> tiempo('sumaDivisores5(5336100)')
#    0.00 segundos
#    >>> tiempo('sumaDivisores6(5336100)')
#    0.00 segundos
#
#    >>> tiempo('sumaDivisores1(2**9 * 3**8 * 5**2)')
#    4.52 segundos
#    >>> tiempo('sumaDivisores2(2**9 * 3**8 * 5**2)')
#    0.00 segundos
#    >>> tiempo('sumaDivisores4(2**9 * 3**8 * 5**2)')
#    0.00 segundos
#    >>> tiempo('sumaDivisores5(2**9 * 3**8 * 5**2)')
#    0.00 segundos
#    >>> tiempo('sumaDivisores6(2**9 * 3**8 * 5**2)')
#    0.00 segundos
#
#    >>> tiempo('sumaDivisores2(2**9 * 3**8 * 5**7 * 7**4)')
#    0.00 segundos
#    >>> tiempo('sumaDivisores4(2**9 * 3**8 * 5**7 * 7**4)')
#    0.00 segundos
#    >>> tiempo('sumaDivisores5(2**9 * 3**8 * 5**7 * 7**4)')
#    3.24 segundos
#    >>> tiempo('sumaDivisores6(2**9 * 3**8 * 5**7 * 7**4)')
#    0.00 segundos
#
#    >>> tiempo('sumaDivisores2(251888923423315469521109880000000)')
#    1.13 segundos
#    >>> tiempo('sumaDivisores4(251888923423315469521109880000000)')
#    0.00 segundos
#    >>> tiempo('sumaDivisores6(251888923423315469521109880000000)')
#    0.00 segundos
#
```

```
#    >>> tiempo('sumaDivisores4(reduce(mul, list(range(1, 30000))))')
#    1.89 segundos
#    >>> tiempo('sumaDivisores6(reduce(mul, list(range(1, 30000))))')
#    1.88 segundos
```

2.14. Números perfectos

En Haskell

```
-- ---------------------------------------------------------------------
-- Un números entero positivo es [perfecto](https://bit.ly/3BIN0be) si
-- es igual a la suma de sus divisores, excluyendo el propio número. Por
-- ejemplo, 6 es un número perfecto porque sus divisores propios son 1,
-- 2 y 3; y 6 = 1 + 2 + 3.
--
-- Definir la función
--    perfectos :: Integer -> [Integer]
-- tal que (perfectos n) es la lista de todos los números perfectos
-- menores que n. Por ejemplo,
--    perfectos 500     ==  [6,28,496]
--    perfectos (10^5)  ==  [6,28,496,8128]
-- ---------------------------------------------------------------------

module Numeros_perfectos where

import Math.NumberTheory.ArithmeticFunctions (sigma)
import Test.QuickCheck

-- 1ª solución
-- ===========

perfectos1 :: Integer -> [Integer]
perfectos1 n =
  [x | x <- [1..n],
       esPerfecto1 x]

-- (esPerfecto x) se verifica si x es un número perfecto. Por ejemplo,
--    esPerfecto 6  ==  True
--    esPerfecto 8  ==  False
esPerfecto1 :: Integer -> Bool
```

```
esPerfecto1 x =
  sumaDivisores1 x - x == x

-- (sumaDivisores x) es la suma de los divisores de x. Por ejemplo,
--    sumaDivisores 12                 ==  28
--    sumaDivisores 25                 ==  31
sumaDivisores1 :: Integer -> Integer
sumaDivisores1 n = sum (divisores1 n)

-- (divisores x) es la lista de los divisores de x. Por ejemplo,
--    divisores 60  ==  [1,5,3,15,2,10,6,30,4,20,12,60]
divisores1 :: Integer -> [Integer]
divisores1 n = [x | x <- [1..n], n `rem` x == 0]

-- 2ª solución
-- ===========

-- Sustituyendo la definición de sumaDivisores de la solución anterior por
-- cada una de las del ejercicio [Suma de divisores](https://bit.ly/3S9aonQ)
-- se obtiene una nueva definición deperfectos. La usada en la
-- definición anterior es la menos eficiente y la que se usa en la
-- siguiente definición es la más eficiente.

perfectos2 :: Integer -> [Integer]
perfectos2 n =
  [x | x <- [1..n],
       esPerfecto2 x]

esPerfecto2 :: Integer -> Bool
esPerfecto2 x =
  sumaDivisores2 x - x == x

sumaDivisores2 :: Integer -> Integer
sumaDivisores2 = sigma 1

-- 3ª solución
-- ===========

perfectos3 :: Integer -> [Integer]
perfectos3 n = filter esPerfecto2 [1..n]
```

```
-- 4ª solución
-- ===========

perfectos4 :: Integer -> [Integer]
perfectos4 = filter esPerfecto2 . enumFromTo 1

-- Comprobación de equivalencia
-- ============================

-- La propiedad es
prop_perfectos :: Positive Integer -> Bool
prop_perfectos (Positive n) =
  all (== perfectos1 n)
      [perfectos2 n,
       perfectos3 n,
       perfectos4 n]

-- La comprobación es
--    λ> quickCheck prop_perfectos
--    +++ OK, passed 100 tests.

-- Comparación de eficiencia
-- =========================

-- La comparación es
--    λ> perfectos1 (4*10^3)
--    [6,28,496]
--    (4.64 secs, 1,606,883,384 bytes)
--    λ> perfectos2 (4*10^3)
--    [6,28,496]
--    (0.02 secs, 9,167,208 bytes)
--
--    λ> perfectos2 (2*10^6)
--    [6,28,496,8128]
--    (3.32 secs, 5,120,880,728 bytes)
--    λ> perfectos3 (2*10^6)
--    [6,28,496,8128]
--    (2.97 secs, 5,040,880,632 bytes)
--    λ> perfectos4 (2*10^6)
```

```
--    [6,28,496,8128]
--    (2.80 secs, 5,040,880,608 bytes)
```

En Python

```python
# ---------------------------------------------------------------------
# Un números entero positivo es [perfecto](https://bit.ly/3BIN0be) si
# es igual a la suma de sus divisores, excluyendo el propio número. Por
# ejemplo, 6 es un número perfecto porque sus divisores propios son 1,
# 2 y 3; y 6 = 1 + 2 + 3.
#
# Definir la función
#    perfectos (int) -> list[int]
# tal que perfectos(n) es la lista de todos los números perfectos
# menores que n. Por ejemplo,
#    perfectos(500)    ==  [6, 28, 496]
#    perfectos(10^5)   ==  [6, 28, 496, 8128]
# ---------------------------------------------------------------------

from timeit import Timer, default_timer

from hypothesis import given
from hypothesis import strategies as st
from sympy import divisor_sigma

# 1ª solución
# ===========

# divisores(n) es la lista de los divisores del número n. Por ejemplo,
#    divisores(30)  ==  [1,2,3,5,6,10,15,30]
def divisores1(n: int) -> list[int]:
    return [x for x in range(1, n + 1) if n % x == 0]

# sumaDivisores(x) es la suma de los divisores de x. Por ejemplo,
#    sumaDivisores(12)                ==  28
#    sumaDivisores(25)                ==  31
def sumaDivisores1(n: int) -> int:
    return sum(divisores1(n))

# esPerfecto(x) se verifica si x es un número perfecto. Por ejemplo,
```

```
#    esPerfecto(6)  ==  True
#    esPerfecto(8)  ==  False
def esPerfecto1(x: int) -> bool:
    return sumaDivisores1(x) - x == x

def perfectos1(n: int) -> list[int]:
    return [x for x in range(1, n + 1) if esPerfecto1(x)]

# 2ª solución
# ===========

# Sustituyendo la definición de sumaDivisores de la solución anterior por
# cada una de las del ejercicio [Suma de divisores](https://bit.ly/3S9aonQ)
# se obtiene una nueva definición deperfectos. La usada en la
# definición anterior es la menos eficiente y la que se usa en la
# siguiente definición es la más eficiente.

def sumaDivisores2(n: int) -> int:
    return divisor_sigma(n, 1)

def esPerfecto2(x: int) -> bool:
    return sumaDivisores2(x) - x == x

def perfectos2(n: int) -> list[int]:
    return [x for x in range(1, n + 1) if esPerfecto2(x)]

# Comprobación de equivalencia
# ============================

# La propiedad es
@given(st.integers(min_value=2, max_value=1000))
def test_perfectos(n: int) -> None:
    assert perfectos1(n) == perfectos2(n)

# La comprobación es
#    src> poetry run pytest -q numeros_perfectos.py
#    1 passed in 1.43s

# Comparación de eficiencia
# =========================
```

```python
def tiempo(e: str) -> None:
    """Tiempo (en segundos) de evaluar la expresión e."""
    t = Timer(e, "", default_timer, globals()).timeit(1)
    print(f"{t:0.2f} segundos")

# La comparación es
#    >>> tiempo('perfectos1(10**4)')
#    2.97 segundos
#    >>> tiempo('perfectos2(10**4)')
#    0.57 segundos
```

2.15. Números abundantes

En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Un número natural n se denomina [abundante](https://bit.ly/3Uk4XUE)
-- si es menor que la suma de sus divisores propios. Por ejemplo, 12 es
-- abundante ya que la suma de sus divisores propios es 16
-- (= 1 + 2 + 3 + 4 + 6), pero 5 y 28 no lo son.
--
-- Definir la función
--    numeroAbundante :: Int -> Bool
-- tal que (numeroAbundante n) se verifica si n es un número
-- abundante. Por ejemplo,
--    numeroAbundante 5  == False
--    numeroAbundante 12 == True
--    numeroAbundante 28 == False
--    numeroAbundante 30 == True
--    numeroAbundante 100000000  ==  True
--    numeroAbundante 100000001  ==  False
-- ---------------------------------------------------------------------

module Numeros_abundantes where

import Math.NumberTheory.ArithmeticFunctions (sigma)
import Test.QuickCheck

-- 1ª solución
```

```haskell
-- ===========

numeroAbundante1 :: Integer -> Bool
numeroAbundante1 x =
  x < sumaDivisores1 x - x

-- (sumaDivisores x) es la suma de los divisores de x. Por ejemplo,
--    sumaDivisores 12                 ==  28
--    sumaDivisores 25                 ==  31
sumaDivisores1 :: Integer -> Integer
sumaDivisores1 n = sum (divisores1 n)

-- (divisores x) es la lista de los divisores de x. Por ejemplo,
--    divisores 60  ==  [1,5,3,15,2,10,6,30,4,20,12,60]
divisores1 :: Integer -> [Integer]
divisores1 n = [x | x <- [1..n], n `rem` x == 0]

-- 2ª solución
-- ===========

-- Sustituyendo la definición de sumaDivisores de la solución anterior por
-- cada una de las del ejercicio [Suma de divisores](https://bit.ly/3S9aonQ)
-- se obtiene una nueva definición de numeroAbundante. La usada en la
-- definición anterior es la menos eficiente y la que se usa en la
-- siguiente definición es la más eficiente.

numeroAbundante2 :: Integer -> Bool
numeroAbundante2 x =
  x < sumaDivisores2 x - x

sumaDivisores2 :: Integer -> Integer
sumaDivisores2 = sigma 1

-- Comprobación de equivalencia
-- ============================

-- La propiedad es
prop_numeroAbundante :: Positive Integer -> Bool
prop_numeroAbundante (Positive n) =
  numeroAbundante1 n == numeroAbundante2 n
```

```
-- La comprobación es
--    λ> quickCheck prop_numeroAbundante
--    +++ OK, passed 100 tests.

-- Comparación de eficiencia
-- =========================

-- La comparación es
--    λ> numeroAbundante1 (5*10^6)
--    True
--    (2.55 secs, 1,000,558,840 bytes)
--    λ> numeroAbundante2 (5*10^6)
--    True
--    (0.00 secs, 555,408 bytes)
```

En Python

```
# ---------------------------------------------------------------------
# Un número natural n se denomina [abundante](https://bit.ly/3Uk4XUE)
# si es menor que la suma de sus divisores propios. Por ejemplo, 12 es
# abundante ya que la suma de sus divisores propios es 16
# (= 1 + 2 + 3 + 4 + 6), pero 5 y 28 no lo son.
#
# Definir la función
#    numeroAbundante : (int) -> bool
# tal que numeroAbundante(n) se verifica si n es un número
# abundante. Por ejemplo,
#    numeroAbundante(5)  == False
#    numeroAbundante(12) == True
#    numeroAbundante(28) == False
#    numeroAbundante(30) == True
#    numeroAbundante(100000000)  ==  True
#    numeroAbundante(100000001)  ==  False
# ---------------------------------------------------------------------

from timeit import Timer, default_timer

from hypothesis import given
from hypothesis import strategies as st
```

```
from sympy import divisor_sigma

# 1ª solución
# ===========

# divisores(n) es la lista de los divisores del número n. Por ejemplo,
#    divisores(30)  ==  [1,2,3,5,6,10,15,30]
def divisores1(n: int) -> list[int]:
    return [x for x in range(1, n + 1) if n % x == 0]

# sumaDivisores(x) es la suma de los divisores de x. Por ejemplo,
#    sumaDivisores(12)                ==  28
#    sumaDivisores(25)                ==  31
def sumaDivisores1(n: int) -> int:
    return sum(divisores1(n))

def numeroAbundante1(x: int) -> bool:
    return x < sumaDivisores1(x) - x

# 2ª solución
# ===========

# Sustituyendo la definición de sumaDivisores de la solución anterior por
# cada una de las del ejercicio [Suma de divisores](https://bit.ly/3S9aonQ)
# se obtiene una nueva definición de numeroAbundante. La usada en la
# definición anterior es la menos eficiente y la que se usa en la
# siguiente definición es la más eficiente.

def sumaDivisores2(n: int) -> int:
    return divisor_sigma(n, 1)

def numeroAbundante2(x: int) -> bool:
    return x < sumaDivisores2(x) - x

# Comprobación de equivalencia
# ============================

# La propiedad es
@given(st.integers(min_value=2, max_value=1000))
def test_numeroAbundante(n: int) -> None:
```

```
    assert numeroAbundante1(n) == numeroAbundante2(n)

# La comprobación es
#    src> poetry run pytest -q numeros_abundantes.py
#    1 passed in 0.38s

# Comparación de eficiencia
# =========================

def tiempo(e: str) -> None:
    """Tiempo (en segundos) de evaluar la expresión e."""
    t = Timer(e, "", default_timer, globals()).timeit(1)
    print(f"{t:0.2f} segundos")

# La comparación es
#    >>> tiempo('numeroAbundante1(4 * 10**7)')
#    2.02 segundos
#    >>> tiempo('numeroAbundante2(4 * 10**7)')
#    0.00 segundos
```

2.16. Números abundantes menores o iguales que n

En Haskell

```
-- ---------------------------------------------------------------------
-- Un número natural n se denomina [abundante](https://bit.ly/3Uk4XUE)
-- si es menor que la suma de sus divisores propios. Por ejemplo, 12 es
-- abundante ya que la suma de sus divisores propios es 16
-- (= 1 + 2 + 3 + 4 + 6), pero 5 y 28 no lo son.
--
-- Definir la función
--    numerosAbundantesMenores :: Integer -> [Integer]
-- tal que (numerosAbundantesMenores n) es la lista de números
-- abundantes menores o iguales que n. Por ejemplo,
--    numerosAbundantesMenores 50  ==  [12,18,20,24,30,36,40,42,48]
--    numerosAbundantesMenores 48  ==  [12,18,20,24,30,36,40,42,48]
--    length (numerosAbundantesMenores (10^6)) ==  247545
-- ---------------------------------------------------------------------
```

```haskell
module Numeros_abundantes_menores_o_iguales_que_n where

import Math.NumberTheory.ArithmeticFunctions (sigma)
import Test.QuickCheck

-- 1ª solución
-- ===========

numerosAbundantesMenores1 :: Integer -> [Integer]
numerosAbundantesMenores1 n =
  [x | x <- [1..n],
       numeroAbundante1 x]

-- (numeroAbundante n) se verifica si n es un número abundante. Por
-- ejemplo,
--    numeroAbundante 5  == False
--    numeroAbundante 12 == True
--    numeroAbundante 28 == False
--    numeroAbundante 30 == True
numeroAbundante1 :: Integer -> Bool
numeroAbundante1 x =
  x < sumaDivisores1 x - x

-- (sumaDivisores x) es la suma de los divisores de x. Por ejemplo,
--    sumaDivisores 12                 ==  28
--    sumaDivisores 25                 ==  31
sumaDivisores1 :: Integer -> Integer
sumaDivisores1 n = sum (divisores1 n)

-- (divisores x) es la lista de los divisores de x. Por ejemplo,
--    divisores 60  ==  [1,5,3,15,2,10,6,30,4,20,12,60]
divisores1 :: Integer -> [Integer]
divisores1 n = [x | x <- [1..n], n `rem` x == 0]

-- 2ª solución
-- ===========

-- Sustituyendo la definición de numeroAbundante de la solución anterior por
-- cada una de las del ejercicio [Números abundantes](https://bit.ly/3xSlWDU)
```

```
-- se obtiene una nueva definición de numerosAbundantesMenores. La usada en la
-- definición anterior es la menos eficiente y la que se usa en la
-- siguiente definición es la más eficiente.

numerosAbundantesMenores2 :: Integer -> [Integer]
numerosAbundantesMenores2 n =
  [x | x <- [1..n],
      numeroAbundante2 x]

numeroAbundante2 :: Integer -> Bool
numeroAbundante2 x =
  x < sumaDivisores2 x - x

sumaDivisores2 :: Integer -> Integer
sumaDivisores2 = sigma 1

-- 3ª solución
-- ===========

numerosAbundantesMenores3 :: Integer -> [Integer]
numerosAbundantesMenores3 n =
  filter numeroAbundante2 [1..n]

-- 4ª solución
-- ===========

numerosAbundantesMenores4 :: Integer -> [Integer]
numerosAbundantesMenores4 =
  filter numeroAbundante2 . enumFromTo 1

-- Comprobación de equivalencia
-- ============================

-- La propiedad es
prop_numerosAbundantesMenores :: Positive Integer -> Bool
prop_numerosAbundantesMenores (Positive n) =
  all (== numerosAbundantesMenores1 n)
      [numerosAbundantesMenores2 n,
       numerosAbundantesMenores3 n,
       numerosAbundantesMenores4 n]
```

```
-- La comprobación es
--    λ> quickCheck prop_numerosAbundantesMenores
--    +++ OK, passed 100 tests.

-- Comparación de eficiencia
-- =========================

-- La comparación es
--    λ> length (numerosAbundantesMenores1 (5*10^3))
--    1239
--    (5.49 secs, 2,508,692,808 bytes)
--    λ> length (numerosAbundantesMenores2 (5*10^3))
--    1239
--    (0.01 secs, 11,501,944 bytes)

--    λ> length (numerosAbundantesMenores2 (10^6))
--    247545
--    (1.48 secs, 2,543,048,024 bytes)
--    λ> length (numerosAbundantesMenores3 (10^6))
--    247545
--    (1.30 secs, 2,499,087,272 bytes)
--    λ> length (numerosAbundantesMenores4 (10^6))
--    247545
--    (1.30 secs, 2,499,087,248 bytes)
```

En Python

```
# ---------------------------------------------------------------------
# Un número natural n se denomina [abundante](https://bit.ly/3Uk4XUE)
# si es menor que la suma de sus divisores propios. Por ejemplo, 12 es
# abundante ya que la suma de sus divisores propios es 16
# (= 1 + 2 + 3 + 4 + 6), pero 5 y 28 no lo son.
#
# Definir la función
#    numerosAbundantesMenores : (int) -> list[Int]
# tal que numerosAbundantesMenores(n) es la lista de números
# abundantes menores o iguales que n. Por ejemplo,
#    numerosAbundantesMenores(50)  ==  [12,18,20,24,30,36,40,42,48]
#    numerosAbundantesMenores(48)  ==  [12,18,20,24,30,36,40,42,48]
```

```
#    leng(numerosAbundantesMenores(10**6)) ==  247545
# ---------------------------------------------------------------------

from timeit import Timer, default_timer

from hypothesis import given
from hypothesis import strategies as st
from sympy import divisor_sigma

# 1ª solución
# ===========

# divisores(n) es la lista de los divisores del número n. Por ejemplo,
#    divisores(30)  ==  [1,2,3,5,6,10,15,30]
def divisores1(n: int) -> list[int]:
    return [x for x in range(1, n + 1) if n % x == 0]

# sumaDivisores(x) es la suma de los divisores de x. Por ejemplo,
#    sumaDivisores(12)                ==  28
#    sumaDivisores(25)                ==  31
def sumaDivisores1(n: int) -> int:
    return sum(divisores1(n))

# numeroAbundante(n) se verifica si n es un número abundante. Por
# ejemplo,
#    numeroAbundante(5)  == False
#    numeroAbundante(12) == True
#    numeroAbundante(28) == False
#    numeroAbundante(30) == True
def numeroAbundante1(x: int) -> bool:
    return x < sumaDivisores1(x) - x

def numerosAbundantesMenores1(n: int) -> list[int]:
    return [x for x in range(1, n + 1) if numeroAbundante1(x)]

# 2ª solución
# ===========

# Sustituyendo la definición de numeroAbundante de la solución anterior por
# cada una de las del ejercicio [Números abundantes](https://bit.ly/3xSlWDU)
```

```
# se obtiene una nueva definición de numerosAbundantesMenores. La usada en la
# definición anterior es la menos eficiente y la que se usa en la
# siguiente definición es la más eficiente.

def sumaDivisores2(n: int) -> int:
    return divisor_sigma(n, 1)

def numeroAbundante2(x: int) -> bool:
    return x < sumaDivisores2(x) - x

def numerosAbundantesMenores2(n: int) -> list[int]:
    return [x for x in range(1, n + 1) if numeroAbundante2(x)]

# Comprobación de equivalencia
# ============================

# La propiedad es
@given(st.integers(min_value=2, max_value=1000))
def test_numerosAbundantesMenores(n: int) -> None:
    assert numerosAbundantesMenores1(n) == numerosAbundantesMenores2(n)

# La comprobación es
#    src> poetry run pytest -q numeros_abundantes_menores_o_iguales_que_n.py
#    1 passed in 1.54s

# Comparación de eficiencia
# =========================

def tiempo(e: str) -> None:
    """Tiempo (en segundos) de evaluar la expresión e."""
    t = Timer(e, "", default_timer, globals()).timeit(1)
    print(f"{t:0.2f} segundos")

# La comparación es
#    >>> tiempo('len(numerosAbundantesMenores1(10**4))')
#    2.21 segundos
#    >>> tiempo('len(numerosAbundantesMenores2(10**4))')
#    0.55 segundos
#
#    >>> tiempo('len(numerosAbundantesMenores2(10**5))')
```

```
#    5.96 segundos
```

2.17. Todos los abundantes hasta n son pares

En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Definir la función
--    todosPares :: Integer -> Bool
-- tal que (todosPares n) se verifica si todos los números abundantes
-- menores o iguales que n son pares. Por ejemplo,
--    todosPares 10    ==  True
--    todosPares 100   ==  True
--    todosPares 1000  ==  False
-- ---------------------------------------------------------------------

module Todos_los_abundantes_hasta_n_son_pares where

import Math.NumberTheory.ArithmeticFunctions (sigma)
import Test.QuickCheck

-- 1ª solución
-- ===========

todosPares1 :: Integer -> Bool
todosPares1 n = and [even x | x <- numerosAbundantesMenores1 n]

-- (numerosAbundantesMenores n) es la lista de números abundantes
-- menores o iguales que n. Por ejemplo,
--    numerosAbundantesMenores 50  ==  [12,18,20,24,30,36,40,42,48]
--    numerosAbundantesMenores 48  ==  [12,18,20,24,30,36,40,42,48]
numerosAbundantesMenores1 :: Integer -> [Integer]
numerosAbundantesMenores1 n =
  [x | x <- [1..n],
      numeroAbundante1 x]

-- (numeroAbundante n) se verifica si n es un número abundante. Por
-- ejemplo,
--    numeroAbundante 5  == False
--    numeroAbundante 12 == True
```

```
--    numeroAbundante 28 == False
--    numeroAbundante 30 == True
numeroAbundante1 :: Integer -> Bool
numeroAbundante1 x =
  x < sumaDivisores1 x - x

-- (sumaDivisores x) es la suma de los divisores de x. Por ejemplo,
--    sumaDivisores 12                 ==  28
--    sumaDivisores 25                 ==  31
sumaDivisores1 :: Integer -> Integer
sumaDivisores1 n = sum (divisores1 n)

-- (divisores x) es la lista de los divisores de x. Por ejemplo,
--    divisores 60  ==  [1,5,3,15,2,10,6,30,4,20,12,60]
divisores1 :: Integer -> [Integer]
divisores1 n = [x | x <- [1..n], n `rem` x == 0]

-- 2ª solución
-- ===========

-- Sustituyendo la definición de numerosAbundantesMenores de la solución
-- anterior por cada una de las del ejercicio anterior se obtiene una
-- nueva definición de todosPares. La usada en la definición anterior es
-- la menos eficiente y la que se usa en la siguiente definición es la
-- más eficiente.

todosPares2 :: Integer -> Bool
todosPares2 n = and [even x | x <- numerosAbundantesMenores2 n]

numerosAbundantesMenores2 :: Integer -> [Integer]
numerosAbundantesMenores2 n =
  [x | x <- [1..n],
       numeroAbundante2 x]

numeroAbundante2 :: Integer -> Bool
numeroAbundante2 x =
  x < sumaDivisores2 x - x

sumaDivisores2 :: Integer -> Integer
sumaDivisores2 = sigma 1
```

```
-- 3ª solución
-- ===========

todosPares3 :: Integer -> Bool
todosPares3 1 = True
todosPares3 n | numeroAbundante1 n = even n && todosPares3 (n-1)
              | otherwise          = todosPares3 (n-1)

-- 4ª solución
-- ===========

todosPares4 :: Integer -> Bool
todosPares4 n = all even (numerosAbundantesMenores1 n)

-- 5ª solución
-- ===========

todosPares5 :: Integer -> Bool
todosPares5 = all even . numerosAbundantesMenores1

-- Comprobación de equivalencia
-- ============================

-- La propiedad es
prop_todosPares :: Positive Integer -> Bool
prop_todosPares (Positive n) =
  all (== todosPares1 n)
      [todosPares2 n,
       todosPares3 n,
       todosPares4 n,
       todosPares5 n]

-- La comprobación es
--    λ> quickCheck prop_todosPares
--    +++ OK, passed 100 tests.

-- Comparación de eficiencia
-- =========================
```

```
-- La comparación es
--    λ> todosPares1 (10^3)
--    False
--    (0.22 secs, 91,257,744 bytes)
--    λ> todosPares2 (10^3)
--    False
--    (0.01 secs, 2,535,656 bytes)
--    λ> todosPares3 (10^3)
--    False
--    (0.03 secs, 11,530,528 bytes)
--    λ> todosPares4 (10^3)
--    False
--    (0.24 secs, 91,231,144 bytes)
--    λ> todosPares5 (10^3)
--    False
--    (0.22 secs, 91,231,208 bytes)
```

En Python

```
# ---------------------------------------------------------------------
# Definir la función
#    todosPares : (int) -> bool
# tal que todosPares(n) se verifica si todos los números abundantes
# menores o iguales que n son pares. Por ejemplo,
#    todosPares(10)    ==  True
#    todosPares(100)   ==  True
#    todosPares(1000)  ==  False
# ---------------------------------------------------------------------

from timeit import Timer, default_timer

from hypothesis import given
from hypothesis import strategies as st
from sympy import divisor_sigma

# 1ª solución
# ===========

# divisores(n) es la lista de los divisores del número n. Por ejemplo,
#    divisores(30)  ==  [1,2,3,5,6,10,15,30]
```

```python
def divisores1(n: int) -> list[int]:
    return [x for x in range(1, n + 1) if n % x == 0]

# sumaDivisores(x) es la suma de los divisores de x. Por ejemplo,
#    sumaDivisores(12)                == 28
#    sumaDivisores(25)                == 31
def sumaDivisores1(n: int) -> int:
    return sum(divisores1(n))

# numeroAbundante(n) se verifica si n es un número abundante. Por
# ejemplo,
#    numeroAbundante(5)  == False
#    numeroAbundante(12) == True
#    numeroAbundante(28) == False
#    numeroAbundante(30) == True
def numeroAbundante1(x: int) -> bool:
    return x < sumaDivisores1(x) - x

# numerosAbundantesMenores(n) es la lista de números abundantes menores
# o iguales que n. Por ejemplo,
#    numerosAbundantesMenores(50)  ==  [12,18,20,24,30,36,40,42,48]
#    numerosAbundantesMenores(48)  ==  [12,18,20,24,30,36,40,42,48]
def numerosAbundantesMenores1(n: int) -> list[int]:
    return [x for x in range(1, n + 1) if numeroAbundante1(x)]

def todosPares1(n: int) -> bool:
    return False not in [x % 2 == 0 for x in numerosAbundantesMenores1(n)]

# 2ª solución
# ===========

# Sustituyendo la definición de numerosAbundantesMenores de la solución
# anterior por cada una de las del ejercicio anterior se obtiene una
# nueva definición de todosPares. La usada en la definición anterior es
# la menos eficiente y la que se usa en la siguiente definición es la
# más eficiente.

def sumaDivisores2(n: int) -> int:
    return divisor_sigma(n, 1)
```

```
def numeroAbundante2(x: int) -> bool:
    return x < sumaDivisores2(x) - x

def numerosAbundantesMenores2(n: int) -> list[int]:
    return [x for x in range(1, n + 1) if numeroAbundante2(x)]

def todosPares2(n: int) -> bool:
    return False not in [x % 2 == 0 for x in numerosAbundantesMenores2(n)]

# 3ª solución
# ===========

def todosPares3(n: int) -> bool:
    return all(x % 2 == 0 for x in numerosAbundantesMenores1(n))

# Comprobación de equivalencia
# ============================

# La propiedad es
@given(st.integers(min_value=2, max_value=1000))
def test_todosPares(n: int) -> None:
    assert todosPares1(n) == todosPares2(n) == todosPares3(n)

# La comprobación es
#    src> poetry run pytest -q todos_los_abundantes_hasta_n_son_pares.py
#    1 passed in 2.63s

# Comparación de eficiencia
# =========================

def tiempo(e: str) -> None:
    """Tiempo (en segundos) de evaluar la expresión e."""
    t = Timer(e, "", default_timer, globals()).timeit(1)
    print(f"{t:0.2f} segundos")

# La comparación es
#    >>> tiempo('todosPares1(1000)')
#    0.03 segundos
#    >>> tiempo('todosPares2(1000)')
#    0.05 segundos
```

```
#    >>> tiempo('todosPares3(1000)')
#    0.02 segundos
#
#    >>> tiempo('todosPares1(10000)')
#    2.07 segundos
#    >>> tiempo('todosPares2(10000)')
#    0.47 segundos
#    >>> tiempo('todosPares3(10000)')
#    2.42 segundos
```

2.18. Números abundantes impares

En Haskell

```
-- ---------------------------------------------------------------------
-- Definir la lista
--   abundantesImpares :: [Integer]
-- cuyos elementos son los números abundantes impares. Por ejemplo,
--    λ> take 12 abundantesImpares
--    [945,1575,2205,2835,3465,4095,4725,5355,5775,5985,6435,6615]
-- ---------------------------------------------------------------------

module Numeros_abundantes_impares where

import Math.NumberTheory.ArithmeticFunctions (sigma)
import Test.QuickCheck


-- 1ª solución
-- ===========

abundantesImpares1 :: [Integer]
abundantesImpares1 = [x | x <- [1,3..], numeroAbundante1 x]

-- (numeroAbundante n) se verifica si n es un número abundante. Por
-- ejemplo,
--    numeroAbundante 5  == False
--    numeroAbundante 12 == True
--    numeroAbundante 28 == False
--    numeroAbundante 30 == True
```

```haskell
numeroAbundante1 :: Integer -> Bool
numeroAbundante1 x =
  x < sumaDivisores1 x - x

-- (sumaDivisores x) es la suma de los divisores de x. Por ejemplo,
--    sumaDivisores 12                 ==  28
--    sumaDivisores 25                 ==  31
sumaDivisores1 :: Integer -> Integer
sumaDivisores1 n = sum (divisores1 n)

-- (divisores x) es la lista de los divisores de x. Por ejemplo,
--    divisores 60  ==  [1,5,3,15,2,10,6,30,4,20,12,60]
divisores1 :: Integer -> [Integer]
divisores1 n = [x | x <- [1..n], n `rem` x == 0]

-- 2ª solución
-- ===========

abundantesImpares2 :: [Integer]
abundantesImpares2 = filter numeroAbundante1 [1,3..]

-- 3ª solución
-- ===========

-- Sustituyendo la definición de numeroAbundante1 de las soluciones
-- anteriores por cada una de las del ejercicio "Números abundantes"
-- https://bit.ly/3xSlWDU se obtiene una nueva definición de abundantes
-- impares. La usada en las definiciones anteriores es la menos
-- eficiente y la que se usa en la siguiente definición es la más eficiente.

abundantesImpares3 :: [Integer]
abundantesImpares3 = filter numeroAbundante3 [1,3..]

numeroAbundante3 :: Integer -> Bool
numeroAbundante3 x =
  x < sumaDivisores3 x - x

sumaDivisores3 :: Integer -> Integer
sumaDivisores3 = sigma 1
```

```
-- Comprobación de equivalencia
-- ============================

-- La propiedad es
prop_abundantesImpares :: Positive Int -> Bool
prop_abundantesImpares (Positive n) =
  all (== take n abundantesImpares1)
      [take n abundantesImpares2,
       take n abundantesImpares3]

-- La comprobación es
--    λ> quickCheckWith (stdArgs {maxSize=10}) prop_abundantesImpares
--    +++ OK, passed 100 tests.

-- Comparación de eficiencia
-- =========================

-- La comparación es
--    λ> abundantesImpares1 !! 5
--    4095
--    (2.07 secs, 841,525,368 bytes)
--    λ> abundantesImpares2 !! 5
--    4095
--    (2.06 secs, 841,443,112 bytes)
--    λ> abundantesImpares3 !! 5
--    4095
--    (0.01 secs, 550,776 bytes)
```

En Python

```
# ---------------------------------------------------------------------
# Definir la función
#   abundantesImpares : (int) -> list[int]
# tal que abundantesImpares(n) son los números abundantes impares
# menores que n. Por ejemplo,
#    >>> abundantesImpares1(10000)[:12]
#    [945, 1575, 2205, 2835, 3465, 4095, 4725, 5355, 5775, 5985, 6435, 6615]
# ---------------------------------------------------------------------

from timeit import Timer, default_timer
```

```
from hypothesis import given
from hypothesis import strategies as st
from sympy import divisor_sigma

# 1ª solución
# ===========

def abundantesImpares1(n: int) -> list[int]:
    return [x for x in range(1, n, 2) if numeroAbundante1(x)]

# divisores(n) es la lista de los divisores del número n. Por ejemplo,
#    divisores(30)  ==  [1,2,3,5,6,10,15,30]
def divisores1(n: int) -> list[int]:
    return [x for x in range(1, n + 1) if n % x == 0]

# sumaDivisores(x) es la suma de los divisores de x. Por ejemplo,
#    sumaDivisores(12)                ==  28
#    sumaDivisores(25)                ==  31
def sumaDivisores1(n: int) -> int:
    return sum(divisores1(n))

# numeroAbundante(n) se verifica si n es un número abundante. Por
# ejemplo,
#    numeroAbundante(5)  == False
#    numeroAbundante(12) == True
#    numeroAbundante(28) == False
#    numeroAbundante(30) == True
def numeroAbundante1(x: int) -> bool:
    return x < sumaDivisores1(x) - x

# 2ª solución
# ===========

def abundantesImpares2(n: int) -> list[int]:
    return list(filter(numeroAbundante1, range(1, n, 2)))

# 3ª solución
# ===========
#
```

```
# Sustituyendo la definición de numeroAbundante1 de las soluciones
# anteriores por cada una de las del ejercicio "Números abundantes"
# https://bit.ly/3xSlWDU se obtiene una nueva definición de abundantes
# impares. La usada en las definiciones anteriores es la menos
# eficiente y la que se usa en la siguiente definición es la más eficiente.

def abundantesImpares3(n: int) -> list[int]:
    return list(filter(numeroAbundante3, range(1, n, 2)))

def sumaDivisores3(n: int) -> int:
    return divisor_sigma(n, 1)

def numeroAbundante3(x: int) -> bool:
    return x < sumaDivisores3(x) - x

# Comprobación de equivalencia
# ============================

# La propiedad es
@given(st.integers(min_value=1, max_value=1000))
def test_abundantesImpares(n: int) -> None:
    r = abundantesImpares1(n)
    assert abundantesImpares2(n) == r
    assert abundantesImpares3(n) == r

# La comprobación es
#    src> poetry run pytest -q numeros_abundantes_impares.py
#    1 passed in 1.42s

# Comparación de eficiencia
# =========================

def tiempo(e: str) -> None:
    """Tiempo (en segundos) de evaluar la expresión e."""
    t = Timer(e, "", default_timer, globals()).timeit(1)
    print(f"{t:0.2f} segundos")

# La comparación es
#    >>> tiempo('abundantesImpares1(10000)[5]')
#    1.25 segundos
```

```
#    >>> tiempo('abundantesImpares2(10000)[5]')
#    1.22 segundos
#    >>> tiempo('abundantesImpares3(10000)[5]')
#    0.33 segundos
```

2.19. Suma de múltiplos de 3 ó 5

En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Definir la función
--    euler1 :: Integer -> Integer
-- tal que (euler1 n) es la suma de todos los múltiplos de 3 ó 5 menores
-- que n. Por ejemplo,
--    euler1 10      == 23
--    euler1 (10^2)  == 2318
--    euler1 (10^3)  == 233168
--    euler1 (10^4)  == 23331668
--    euler1 (10^5)  == 2333316668
--    euler1 (10^10) == 23333333331666666668
--    euler1 (10^20) == 2333333333333333333316666666666666666668
--    euler1 (10^30) == 233333333333333333333333333333166666666666666666666666666
--
-- Nota: Este ejercicio está basado en el problema 1 del Proyecto Euler
-- https://projecteuler.net/problem=1
-- ---------------------------------------------------------------------

module Suma_de_multiplos_de_3_o_5 where

import Data.List (nub, union)
import qualified Data.Set as S (fromAscList, union)
import Test.QuickCheck

-- 1ª solución
-- ===========

euler1a :: Integer -> Integer
euler1a n =
  sum [x | x <- [1..n-1],
           multiplo x 3 || multiplo x 5]
```

```
-- (multiplo x y) se verifica si x es un múltiplo de y. Por ejemplo.
--    multiplo 12 3  ==  True
--    multiplo 14 3  ==  False
multiplo :: Integer -> Integer -> Bool
multiplo x y = mod x y == 0

-- 2ª solución                                                          --
-- ===========

euler1b :: Integer -> Integer
euler1b n =
  sum [x | x <- [1..n-1],
           gcd x 15 > 1]

-- 3ª solución                                                          --
-- ===========

euler1c :: Integer -> Integer
euler1c n =
  sum [3,6..n-1] + sum [5,10..n-1] - sum [15,30..n-1]

-- 4ª solución                                                          --
-- ===========

euler1d :: Integer -> Integer
euler1d n =
  sum (nub ([3,6..n-1] ++ [5,10..n-1]))

-- 5ª solución                                                          --
-- ===========

euler1e :: Integer -> Integer
euler1e n =
  sum ([3,6..n-1] `union` [5,10..n-1])

-- 6ª solución                                                          --
-- ===========

euler1f :: Integer -> Integer
```

```haskell
euler1f n =
  sum (S.fromAscList [3,6..n-1] `S.union` S.fromAscList [5,10..n-1])

-- 7ª solución                                                          --
-- ===========

euler1g :: Integer -> Integer
euler1g n =
  suma 3 n + suma 5 n - suma 15 n

-- (suma d x) es la suma de los múltiplos de d menores que x. Por
-- ejemplo,
--    suma 3 20  ==  63
suma :: Integer -> Integer -> Integer
suma d x = (a+b)*n `div` 2
    where a = d
          b = d * ((x-1) `div` d)
          n = 1 + (b-a) `div` d

-- Comprobación de equivalencia
-- ============================

-- La propiedad es
prop_euler1 :: Positive Integer -> Bool
prop_euler1 (Positive n) =
  all (== euler1a n)
      [euler1b n,
       euler1c n,
       euler1d n,
       euler1e n,
       euler1f n,
       euler1g n]

-- La comprobación es
--    λ> quickCheck prop_euler1
--    +++ OK, passed 100 tests.

-- Comparación de eficiencia
-- =========================
```

```
-- La comparación es
--    λ> euler1a (5*10^4)
--    583291668
--    (0.05 secs, 21,895,296 bytes)
--    λ> euler1b (5*10^4)
--    583291668
--    (0.05 secs, 26,055,096 bytes)
--    λ> euler1c (5*10^4)
--    583291668
--    (0.01 secs, 5,586,072 bytes)
--    λ> euler1d (5*10^4)
--    583291668
--    (2.83 secs, 7,922,304 bytes)
--    λ> euler1e (5*10^4)
--    583291668
--    (4.56 secs, 12,787,705,248 bytes)
--    λ> euler1f (5*10^4)
--    583291668
--    (0.01 secs, 8,168,584 bytes)
--    λ> euler1g (5*10^4)
--    583291668
--    (0.02 secs, 557,488 bytes)
--
--    λ> euler1a (3*10^6)
--    2099998500000
--    (2.72 secs, 1,282,255,816 bytes)
--    λ> euler1b (3*10^6)
--    2099998500000
--    (2.06 secs, 1,531,855,776 bytes)
--    λ> euler1c (3*10^6)
--    2099998500000
--    (0.38 secs, 305,127,480 bytes)
--    λ> euler1f (3*10^6)
--    2099998500000
--    (0.54 secs, 457,358,232 bytes)
--    λ> euler1g (3*10^6)
--    2099998500000
--    (0.01 secs, 560,472 bytes)
--
--    λ> euler1c (10^7)
```

```
--    23333331666668
--    (1.20 secs, 1,015,920,024 bytes)
--    λ> euler1f (10^7)
--    23333331666668
--    (2.00 secs, 1,523,225,648 bytes)
--    λ> euler1g (10^7)
--    23333331666668
--    (0.01 secs, 561,200 bytes)
```

En Python

```
# ---------------------------------------------------------------------
# Definir la función
#    euler1 : (int) -> int
# tal que euler1(n) es la suma de todos los múltiplos de 3 ó 5 menores
# que n. Por ejemplo,
#    euler1(10)     == 23
#    euler1(10**2)  == 2318
#    euler1(10**3)  == 233168
#    euler1(10**4)  == 23331668
#    euler1(10**5)  == 2333316668
#    euler1(10**10) == 23333333331666666668
#    euler1(10**20) == 2333333333333333333316666666666666666668
#
# Nota: Este ejercicio está basado en el problema 1 del Proyecto Euler
# https://projecteuler.net/problem=1
# ---------------------------------------------------------------------

from math import gcd
from timeit import Timer, default_timer

from hypothesis import given
from hypothesis import strategies as st

# 1ª solución
# ===========

# multiplo(x, y) se verifica si x es un múltiplo de y. Por ejemplo.
#    multiplo(12, 3)  ==  True
#    multiplo(14, 3)  ==  False
```

```
def multiplo(x: int, y: int) -> int:
    return x % y == 0

def euler1a(n: int) -> int:
    return sum(x for x in range(1, n)
               if (multiplo(x, 3) or multiplo(x, 5)))

# 2ª solución                                                        --
# ===========

def euler1b(n: int) -> int:
    return sum(x for x in range(1, n)
               if gcd(x, 15) > 1)

# 3ª solución                                                        --
# ===========

def euler1c(n: int) -> int:
    return sum(range(3, n, 3)) + \
           sum(range(5, n, 5)) - \
           sum(range(15, n, 15))

# 4ª solución                                                        --
# ===========

def euler1d(n: int) -> int:
    return sum(set(list(range(3, n, 3)) + list(range(5, n, 5))))

# 5ª solución                                                        --
# ===========

def euler1e(n: int) -> int:
    return sum(set(list(range(3, n, 3))) | set(list(range(5, n, 5))))

# 6ª solución                                                        --
# ===========

# suma(d, x) es la suma de los múltiplos de d menores que x. Por
# ejemplo,
#    suma(3, 20)  ==  63
```

```
def suma(d: int, x: int) -> int:
    a = d
    b = d * ((x - 1) // d)
    n = 1 + (b - a) // d
    return (a + b) * n // 2

def euler1f(n: int) -> int:
    return suma(3, n) + suma(5, n) - suma(15, n)

# Comprobación de equivalencia
# ============================

# La propiedad es
@given(st.integers(min_value=1, max_value=1000))
def test_euler1(n: int) -> None:
    r = euler1a(n)
    assert euler1b(n) == r
    assert euler1c(n) == r
    assert euler1d(n) == r
    assert euler1e(n) == r

# La comprobación es
#    src> poetry run pytest -q suma_de_multiplos_de_3_o_5.py
#    1 passed in 0.16s

# Comparación de eficiencia
# =========================

def tiempo(e: str) -> None:
    """Tiempo (en segundos) de evaluar la expresión e."""
    t = Timer(e, "", default_timer, globals()).timeit(1)
    print(f"{t:0.2f} segundos")

# La comparación es
#    >>> tiempo('euler1a(10**7)')
#    1.49 segundos
#    >>> tiempo('euler1b(10**7)')
#    0.93 segundos
#    >>> tiempo('euler1c(10**7)')
#    0.07 segundos
```

```
#    >>> tiempo('euler1d(10**7)')
#    0.42 segundos
#    >>> tiempo('euler1e(10**7)')
#    0.69 segundos
#    >>> tiempo('euler1f(10**7)')
#    0.00 segundos
#
#    >>> tiempo('euler1c(10**8)')
#    0.72 segundos
#    >>> tiempo('euler1f(10**8)')
#    0.00 segundos
```

2.20. Puntos dentro del círculo

En Haskell

```
-- ---------------------------------------------------------------------
-- En el círculo de radio 2 hay 6 puntos cuyas coordenadas son puntos
-- naturales:
--    (0,0),(0,1),(0,2),(1,0),(1,1),(2,0)
-- y en de radio 3 hay 11:
--    (0,0),(0,1),(0,2),(0,3),(1,0),(1,1),(1,2),(2,0),(2,1),(2,2),(3,0)
--
-- Definir la función
--    circulo :: Int -> Int
-- tal que (circulo n) es el la cantidad de pares de números naturales
-- (x,y) que se encuentran en el círculo de radio n. Por ejemplo,
--    circulo 1    ==  3
--    circulo 2    ==  6
--    circulo 3    ==  11
--    circulo 4    ==  17
--    circulo 100  ==  7955
-- ---------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-type-defaults #-}

module Puntos_dentro_del_circulo where

import Test.QuickCheck
```

```haskell
-- 1ª solución
-- ===========

circulo1 :: Int -> Int
circulo1 n = length (enCirculo1 n)

enCirculo1 :: Int -> [(Int, Int)]
enCirculo1 n = [(x,y) | x <- [0..n],
                        y <- [0..n],
                        x*x+y*y <= n*n]

-- 2ª solución
-- ===========

circulo2 :: Int -> Int
circulo2 0 = 1
circulo2 n =
  2 * length (enSemiCirculo n) + ceiling(fromIntegral n / sqrt 2)

enSemiCirculo :: Int -> [(Int, Int)]
enSemiCirculo n =
  [(x,y) | x <- [0..floor (sqrt (fromIntegral (n * n)))],
           y <- [x+1..truncate (sqrt (fromIntegral (n*n - x*x)))]]

-- Comprobación de equivalencia
-- ============================

-- La propiedad es
prop_circulo :: Positive Int -> Bool
prop_circulo (Positive n) =
  circulo1 n == circulo2 n

-- La comprobación es
--    λ> quickCheck prop_circulo
--    +++ OK, passed 100 tests.

-- Comparación de eficiencia
-- =========================

-- La comparación es
```

```
--    λ> circulo1 (2*10^3)
--    3143587
--    (3.58 secs, 1,744,162,600 bytes)
--    λ> circulo2 (2*10^3)
--    3143587
--    (0.41 secs, 266,374,208 bytes)
```

En Python

```python
# ---------------------------------------------------------------------
# En el círculo de radio 2 hay 6 puntos cuyas coordenadas son puntos
# naturales:
#    (0,0),(0,1),(0,2),(1,0),(1,1),(2,0)
# y en de radio 3 hay 11:
#    (0,0),(0,1),(0,2),(0,3),(1,0),(1,1),(1,2),(2,0),(2,1),(2,2),(3,0)
#
# Definir la función
#    circulo : (int) -> int
# tal que circulo(n) es el la cantidad de pares de números naturales
# (x,y) que se encuentran en el círculo de radio n. Por ejemplo,
#    circulo(1)    ==  3
#    circulo(2)    ==  6
#    circulo(3)    ==  11
#    circulo(4)    ==  17
#    circulo(100)  ==  7955
# ---------------------------------------------------------------------

from math import ceil, sqrt, trunc
from timeit import Timer, default_timer

from hypothesis import given
from hypothesis import strategies as st

# 1ª solución
# ===========

def circulo1(n: int) -> int:
    return len([(x, y)
                for x in range(0, n + 1)
                for y in range(0, n + 1)
```

```
                if x * x + y * y <= n * n])

# 2ª solución
# ===========

def enSemiCirculo(n: int) -> list[tuple[int, int]]:
    return [(x, y)
            for x in range(0, ceil(sqrt(n**2)) + 1)
            for y in range(x+1, trunc(sqrt(n**2 - x**2)) + 1)]

def circulo2(n: int) -> int:
    if n == 0:
        return 1
    return (2 * len(enSemiCirculo(n)) + ceil(n / sqrt(2)))

# 3ª solución
# ===========

def circulo3(n: int) -> int:
    r = 0
    for x in range(0, n + 1):
        for y in range(0, n + 1):
            if x**2 + y**2 <= n**2:
                r = r + 1
    return r

# 4ª solución
# ===========

def circulo4(n: int) -> int:
    r = 0
    for x in range(0, ceil(sqrt(n**2)) + 1):
        for y in range(x + 1, trunc(sqrt(n**2 - x**2)) + 1):
            if x**2 + y**2 <= n**2:
                r = r + 1
    return 2 * r + ceil(n / sqrt(2))

# Comprobación de equivalencia
# ============================
```

```
# La propiedad es
@given(st.integers(min_value=1, max_value=100))
def test_circulo(n: int) -> None:
    r = circulo1(n)
    assert circulo2(n) == r
    assert circulo3(n) == r
    assert circulo4(n) == r

# La comprobación es
#    src> poetry run pytest -q puntos_dentro_del_circulo.py
#    1 passed in 0.60s

# Comparación de eficiencia
# =========================

def tiempo(e: str) -> None:
    """Tiempo (en segundos) de evaluar la expresión e."""
    t = Timer(e, "", default_timer, globals()).timeit(1)
    print(f"{t:0.2f} segundos")

# La comparación es
#    >>> tiempo('circulo1(2000)')
#    0.71 segundos
#    >>> tiempo('circulo2(2000)')
#    0.76 segundos
#    >>> tiempo('circulo3(2000)')
#    2.63 segundos
#    >>> tiempo('circulo4(2000)')
#    1.06 segundos
```

2.21. Aproximación del número e

En Haskell

```
-- ---------------------------------------------------------------------
-- El [número e](https://bit.ly/3y17R7l) se define como el límite de la
-- sucesión (1+1/n)**n; es decir,
--    e = lim (1+1/n)**n
--
-- Definir las funciones
```

```
--    aproxE      :: Int -> [Double]
--    errorAproxE :: Double -> Int
-- tales que
-- + (aproxE k) es la lista de los k primeros términos de la sucesión
--   (1+1/n)**m. Por ejemplo,
--      aproxE 4 == [2.0,2.25,2.37037037037037,2.44140625]
--      last (aproxE (7*10^7))  ==  2.7182818287372563
-- + (errorE x) es el menor número de términos de la sucesión
--   (1+1/m)**m necesarios para obtener su límite con un error menor que
--   x. Por ejemplo,
--      errorAproxE 0.1    ==  13
--      errorAproxE 0.01   ==  135
--      errorAproxE 0.001  ==  1359
--
-- Indicación: En Haskell, e se calcula como (exp 1).
-- ---------------------------------------------------------------------

module Aproximacion_del_numero_e where

import Test.QuickCheck

-- 1ª definición de aproxE
-- =======================

aproxE1 :: Int -> [Double]
aproxE1 k = [(1+1/n)**n | n <- [1..k']]
  where k' = fromIntegral k

-- 2ª definición de aproxE
-- =======================

aproxE2 :: Int -> [Double]
aproxE2 0 = []
aproxE2 n = aproxE2 (n-1) ++ [(1+1/n')**n']
  where n' = fromIntegral n

-- 3ª definición de aproxE
-- =======================

aproxE3 :: Int -> [Double]
```

```
aproxE3 = reverse . aux . fromIntegral
  where aux 0 = []
        aux n = (1+1/n)**n : aux (n-1)

-- 4ª definición de aproxE
-- =======================

aproxE4 :: Int -> [Double]
aproxE4 k = aux [] (fromIntegral k)
  where aux xs 0 = xs
        aux xs n = aux ((1+1/n)**n : xs) (n-1)

-- 5ª definición de aproxE
-- =======================

aproxE5 :: Int -> [Double]
aproxE5 k = map (\ n -> (1+1/n)**n) [1..k']
  where k' = fromIntegral k

-- Comprobación de equivalencia de aproxE
-- ======================================

-- La propiedad es
prop_aproxE :: Positive Int -> Bool
prop_aproxE (Positive k) =
  all (== aproxE1 k)
      [aproxE2 k,
       aproxE3 k,
       aproxE4 k,
       aproxE5 k]

-- La comprobación es
--    λ> quickCheck prop_aproxE
--    +++ OK, passed 100 tests.

-- Comparación de eficiencia de aproxE
-- ===================================

-- La comparación es
--    λ> last (aproxE1 (2*10^4))
```

```
--     2.718213874533619
--     (0.04 secs, 5,368,968 bytes)
--     λ> last (aproxE2 (2*10^4))
--     2.718213874533619
--     (5.93 secs, 17,514,767,104 bytes)
--     λ> last (aproxE3 (2*10^4))
--     2.718213874533619
--     (0.05 secs, 9,529,336 bytes)
--     λ> last (aproxE4 (2*10^4))
--     2.718213874533619
--     (0.05 secs, 9,529,184 bytes)
--     λ> last (aproxE5 (2*10^4))
--     2.718213874533619
--     (0.01 secs, 4,888,960 bytes)
--
--     λ> last (aproxE1 (2*10^6))
--     2.7182811492688552
--     (0.54 secs, 480,570,120 bytes)
--     λ> last (aproxE3 (2*10^6))
--     2.7182811492688552
--     (2.07 secs, 896,570,280 bytes)
--     λ> last (aproxE4 (2*10^6))
--     2.7182811492688552
--     (2.18 secs, 896,570,336 bytes)
--     λ> last (aproxE5 (2*10^6))
--     2.7182811492688552
--     (0.09 secs, 432,570,112 bytes)

-- 1ª definición de errorAproxE
-- ============================

errorAproxE1 :: Double -> Int
errorAproxE1 x =
  round (head [n | n <- [1..], abs (exp 1 - (1+1/n)**n) < x])

-- 2ª definición de errorAproxE
-- ============================

errorAproxE2 :: Double -> Int
errorAproxE2 x = aux 1
```

```
  where aux n | abs (exp 1 - (1+1/n)**n) < x = round n
              | otherwise                    = aux (n+1)

-- 3ª definición de errorAproxE
-- ============================

errorAproxE3 :: Double -> Int
errorAproxE3 x =
  round (head (dropWhile (\ n -> abs (exp 1 - (1+1/n)**n) >= x) [1..]))

-- Comprobación de equivalencia de errorAproxE
-- ===========================================

-- La propiedad es
prop_errorAproxE :: Positive Double -> Bool
prop_errorAproxE (Positive x) =
  all (== errorAproxE1 x)
      [errorAproxE2 x,
       errorAproxE3 x]

-- La comprobación es
--    λ> quickCheck prop_errorAproxE
--    +++ OK, passed 100 tests.

-- Comparación de eficiencia de errorAproxE
-- ========================================

-- La comparación es
--    λ> errorAproxE1 0.000001
--    1358611
--    (1.70 secs, 674,424,552 bytes)
--    λ> errorAproxE2 0.000001
--    1358611
--    (1.79 secs, 739,637,704 bytes)
--    λ> errorAproxE3 0.000001
--    1358611
--    (1.20 secs, 609,211,144 bytes)
```

En Python

```
# ---------------------------------------------------------------------
# El [número e](https://bit.ly/3y17R7l) se define como el límite de la
# sucesión (1+1/n)**n; es decir,
#    e = lim (1+1/n)**n
#
# Definir las funciones
#    aproxE      : (int) -> list[float]
#    errorAproxE : (float) -> int
# tales que
# + aproxE(k) es la lista de los k primeros términos de la sucesión
#   (1+1/n)**m. Por ejemplo,
#      aproxE(4) == [2.0, 2.25, 2.37037037037037, 2.44140625]
#      aproxE6(7*10**7)[-1] ==  2.7182818287372563
# + errorE(x) es el menor número de términos de la sucesión
#   (1+1/m)**m necesarios para obtener su límite con un error menor que
#   x. Por ejemplo,
#      errorAproxE(0.1)    ==  13
#      errorAproxE(0.01)   ==  135
#      errorAproxE(0.001)  ==  1359
# ---------------------------------------------------------------------

from itertools import dropwhile, islice
from math import e
from sys import setrecursionlimit
from timeit import Timer, default_timer
from typing import Iterator

from hypothesis import given
from hypothesis import strategies as st

setrecursionlimit(10**6)

# 1ª definición de aproxE
# =======================

def aproxE1(k: int) -> list[float]:
    return [(1 + 1/n)**n for n in range(1, k + 1)]

# 2ª definición de aproxE
```

```python
# ========================

def aproxE2(n: int) -> list[float]:
    if n == 0:
        return []
    return aproxE2(n - 1) + [(1 + 1/n)**n]

# 3ª definición de aproxE
# ========================

def aproxE3(n: int) -> list[float]:
    def aux(n: int) -> list[float]:
        if n == 0:
            return []
        return [(1 + 1/n)**n] + aux(n - 1)

    return list(reversed(aux(n)))

# 4ª definición de aproxE
# ========================

def aproxE4(n: int) -> list[float]:
    def aux(xs: list[float], n: int) -> list[float]:
        if n == 0:
            return xs
        return aux([(1 + 1/n)**n] + xs, n - 1)

    return aux([], n)

# 5ª definición de aproxE
# ========================

def aproxE5(n: int) -> list[float]:
    return list(map((lambda k: (1+1/k)**k), range(1, n+1)))

# 6ª definición de aproxE
# ========================

def aproxE6(n: int) -> list[float]:
    r = []
```

```
    for k in range(1, n+1):
        r.append((1+1/k)**k)
    return r

# Comprobación de equivalencia de aproxE
# ======================================

# La propiedad es
@given(st.integers(min_value=1, max_value=100))
def test_aproxE(n: int) -> None:
    r = aproxE1(n)
    assert aproxE2(n) == r
    assert aproxE3(n) == r
    assert aproxE4(n) == r
    assert aproxE5(n) == r
    assert aproxE6(n) == r

# La comprobación es
#    src> poetry run pytest -q aproximacion_del_numero_e.py
#    1 passed in 0.60s

# Comparación de eficiencia de aproxE
# ===================================

def tiempo(ex: str) -> None:
    """Tiempo (en segundos) de evaluar la expresión e."""
    t = Timer(ex, "", default_timer, globals()).timeit(1)
    print(f"{t:0.2f} segundos")

# La comparación es
#    >>> tiempo('aproxE1(20000)')
#    0.00 segundos
#    >>> tiempo('aproxE2(20000)')
#    0.43 segundos
#    >>> tiempo('aproxE3(20000)')
#    0.60 segundos
#    >>> tiempo('aproxE4(20000)')
#    1.23 segundos
#    >>> tiempo('aproxE5(20000)')
#    0.00 segundos
```

```
#    >>> tiempo('aproxE6(20000)')
#    0.00 segundos

#    >>> tiempo('aproxE1(10**7)')
#    1.18 segundos
#    >>> tiempo('aproxE5(10**7)')
#    1.48 segundos
#    >>> tiempo('aproxE6(10**7)')
#    1.43 segundos

# 1ª definición de errorAproxE
# ============================

# naturales es el generador de los números naturales positivos, Por
# ejemplo,
#    >>> list(islice(naturales(), 5))
#    [1, 2, 3, 4, 5]
def naturales() -> Iterator[int]:
    i = 1
    while True:
        yield i
        i += 1

def errorAproxE1(x: float) -> int:
    return list(islice((n for n in naturales()
                        if abs(e - (1 + 1/n)**n) < x), 1))[0]

# # 2ª definición de errorAproxE
# # ============================

def errorAproxE2(x: float) -> int:
    def aux(n: int) -> int:
        if abs(e - (1 + 1/n)**n) < x:
            return n
        return aux(n + 1)

    return aux(1)

# 3ª definición de errorAproxE
# ============================
```

```
def errorAproxE3(x: float) -> int:
    return list(islice(dropwhile(lambda n: abs(e - (1 + 1/n)**n) >= x,
                                 naturales()),
                       1))[0]


# Comprobación de equivalencia de errorAproxE
# ===========================================

@given(st.integers(min_value=1, max_value=100))
def test_errorAproxE(n: int) -> None:
    r = errorAproxE1(n)
    assert errorAproxE2(n) == r
    assert errorAproxE3(n) == r

# La comprobación es
#    src> poetry run pytest -q aproximacion_del_numero_e.py
#    2 passed in 0.60s

# Comparación de eficiencia de aproxE
# ===================================

# La comparación es
#    >>> tiempo('errorAproxE1(0.0001)')
#    0.00 segundos
#    >>> tiempo('errorAproxE2(0.0001)')
#    0.00 segundos
#    >>> tiempo('errorAproxE3(0.0001)')
#    0.00 segundos
#
#    >>> tiempo('errorAproxE1(0.0000001)')
#    2.48 segundos
#    >>> tiempo('errorAproxE3(0.0000001)')
#    2.61 segundos
```

2.22. Aproximación al límite de sen(x)/x cuando x tiende a cero

En Haskell

```haskell
-- ---------------------------------------------------------------------
-- El limite de sen(x)/x, cuando x tiende a cero, se puede calcular como
-- el límite de la sucesión sen(1/n)/(1/n), cuando n tiende a infinito.
--
-- Definir las funciones
--    aproxLimSeno :: Int -> [Double]
--    errorLimSeno :: Double -> Int
-- tales que
-- + (aproxLimSeno n) es la lista cuyos elementos son los n primeros
--   términos de la sucesión sen(1/m)/(1/m). Por ejemplo,
--      aproxLimSeno 1 == [0.8414709848078965]
--      aproxLimSeno 2 == [0.8414709848078965,0.958851077208406]
-- + (errorLimSeno x) es el menor número de términos de la sucesión
--   sen(1/m)/(1/m) necesarios para obtener su límite con un error menor
--   que x. Por ejemplo,
--      errorLimSeno 0.1      ==   2
--      errorLimSeno 0.01     ==   5
--      errorLimSeno 0.001    ==  13
--      errorLimSeno 0.0001   ==  41
-- ---------------------------------------------------------------------

module Limite_del_seno where

import Test.QuickCheck

-- 1ª definición de aproxLimSeno
-- =============================

aproxLimSeno1 :: Int -> [Double]
aproxLimSeno1 k = [sin(1/n)/(1/n) | n <- [1..k']]
  where k' = fromIntegral k

-- 2ª definición de aproxLimSeno
-- =============================
```

```haskell
aproxLimSeno2 :: Int -> [Double]
aproxLimSeno2 0 = []
aproxLimSeno2 n = aproxLimSeno2 (n-1) ++ [sin(1/n')/(1/n')]
  where n' = fromIntegral n

-- 3ª definición de aproxLimSeno
-- =============================

aproxLimSeno3 :: Int -> [Double]
aproxLimSeno3 = reverse . aux . fromIntegral
  where aux 0 = []
        aux n = sin(1/n)/(1/n) : aux (n-1)

-- 4ª definición de aproxLimSeno
-- =============================

aproxLimSeno4 :: Int -> [Double]
aproxLimSeno4 k = aux [] (fromIntegral k)
  where aux xs 0 = xs
        aux xs n = aux (sin(1/n)/(1/n) : xs) (n-1)

-- 5ª definición de aproxLimSeno
-- =============================

aproxLimSeno5 :: Int -> [Double]
aproxLimSeno5 k =  map (\ n -> sin(1/n)/(1/n)) [1..k']
  where k' = fromIntegral k

-- Comprobación de equivalencia de aproxLimSeno
-- ============================================

-- La propiedad es
prop_aproxLimSeno :: Positive Int -> Bool
prop_aproxLimSeno (Positive k) =
  all (== aproxLimSeno1 k)
      [aproxLimSeno2 k,
       aproxLimSeno3 k,
       aproxLimSeno4 k,
       aproxLimSeno5 k]
```

```
-- La comprobación es
--    λ> quickCheck prop_aproxLimSeno
--    +++ OK, passed 100 tests.

-- Comparación de eficiencia de aproxLimSeno
-- =========================================

-- La comparación es
--    λ> last (aproxLimSeno1 (2*10^4))
--    0.9999999995833334
--    (0.01 secs, 5,415,816 bytes)
--    λ> last (aproxLimSeno2 (2*10^4))
--    0.9999999995833334
--    (4.48 secs, 17,514,768,064 bytes)
--    λ> last (aproxLimSeno3 (2*10^4))
--    0.9999999995833334
--    (0.02 secs, 9,530,120 bytes)
--    λ> last (aproxLimSeno4 (2*10^4))
--    0.9999999995833334
--    (0.02 secs, 9,529,968 bytes)
--    λ> last (aproxLimSeno5 (2*10^4))
--    0.9999999995833334
--    (0.01 secs, 4,889,720 bytes)
--
--    λ> last (aproxLimSeno1 (2*10^6))
--    0.9999999999999583
--    (0.46 secs, 480,569,808 bytes)
--    λ> last (aproxLimSeno3 (2*10^6))
--    0.9999999999999583
--    (1.96 secs, 896,569,992 bytes)
--    λ> last (aproxLimSeno4 (2*10^6))
--    0.9999999999999583
--    (1.93 secs, 896,570,048 bytes)
--    λ> last (aproxLimSeno5 (2*10^6))
--    0.9999999999999583
--    (0.05 secs, 432,569,800 bytes)
--
--    λ> last (aproxLimSeno1 (10^7))
--    0.9999999999999983
--    (2.26 secs, 2,400,569,760 bytes)
```

```haskell
--    λ> last (aproxLimSeno5 (10^7))
--    0.9999999999999983
--    (0.24 secs, 2,160,569,752 bytes)

-- 1ª definición de errorLimSeno
-- ==============================

errorLimSeno1 :: Double -> Int
errorLimSeno1 x =
  round (head [m | m <- [1..], abs (1 - sin(1/m)/(1/m)) < x])

-- 2ª definición de errorLimSeno
-- ==============================

errorLimSeno2 :: Double -> Int
errorLimSeno2 x = aux 1
  where aux n | abs (1 - sin(1/n)/(1/n)) < x = round n
              | otherwise                    = aux (n+1)

-- 3ª definición de errorLimSeno
-- ==============================

errorLimSeno3 :: Double -> Int
errorLimSeno3 x =
  round (head (dropWhile (\ n -> abs (1 - sin(1/n)/(1/n)) >= x) [1..]))

-- Comprobación de equivalencia de errorLimSeno
-- ============================================

-- La propiedad es
prop_errorLimSeno :: Positive Double -> Bool
prop_errorLimSeno (Positive x) =
  all (== errorLimSeno1 x)
      [errorLimSeno2 x,
       errorLimSeno3 x]

-- La comprobación es
--    λ> quickCheck prop_errorLimSeno
--    +++ OK, passed 100 tests.
```

```
-- Comparación de eficiencia de errorLimSeno
-- ==========================================

-- La comparación es
--    λ> errorLimSeno1 (10**(-12))
--    408230
--    (0.41 secs, 206,300,808 bytes)
--    λ> errorLimSeno2 (10**(-12))
--    408230
--    (0.46 secs, 225,895,672 bytes)
--    λ> errorLimSeno3 (10**(-12))
--    408230
--    (0.37 secs, 186,705,688 bytes)
```

En Python

```python
# ---------------------------------------------------------------------
# El limite de sen(x)/x, cuando x tiende a cero, se puede calcular como
# el límite de la sucesión sen(1/n)/(1/n), cuando n tiende a infinito.
#
# Definir las funciones
#    aproxLimSeno : (int) -> list[float]
#    errorLimSeno : (float) -> int
# tales que
# + aproxLimSeno(n) es la lista cuyos elementos son los n primeros
#   términos de la sucesión sen(1/m)/(1/m). Por ejemplo,
#      aproxLimSeno(1) == [0.8414709848078965]
#      aproxLimSeno(2) == [0.8414709848078965,0.958851077208406]
# + errorLimSeno(x) es el menor número de términos de la sucesión
#   sen(1/m)/(1/m) necesarios para obtener su límite con un error menor
#   que x. Por ejemplo,
#      errorLimSeno(0.1)     ==   2
#      errorLimSeno(0.01)    ==   5
#      errorLimSeno(0.001)   ==  13
#      errorLimSeno(0.0001)  ==  41
# ---------------------------------------------------------------------

from itertools import dropwhile, islice
from math import sin
from sys import setrecursionlimit
```

```
from timeit import Timer, default_timer
from typing import Iterator

from hypothesis import given
from hypothesis import strategies as st

setrecursionlimit(10**6)

# 1ª definición de aproxLimSeno
# =============================

def aproxLimSeno1(k: int) -> list[float]:
    return [sin(1/n)/(1/n) for n in range(1, k + 1)]

# 2ª definición de aproxLimSeno
# =============================

def aproxLimSeno2(n: int) -> list[float]:
    if n == 0:
        return []
    return aproxLimSeno2(n - 1) + [sin(1/n)/(1/n)]

# 3ª definición de aproxLimSeno
# =============================

def aproxLimSeno3(n: int) -> list[float]:
    def aux(n: int) -> list[float]:
        if n == 0:
            return []
        return [sin(1/n)/(1/n)] + aux(n - 1)

    return list(reversed(aux(n)))

# 4ª definición de aproxLimSeno
# =============================

def aproxLimSeno4(n: int) -> list[float]:
    def aux(xs: list[float], n: int) -> list[float]:
        if n == 0:
            return xs
```

```
        return aux([sin(1/n)/(1/n)] + xs, n - 1)

    return aux([], n)

# 5ª definición de aproxLimSeno
# =============================

def aproxLimSeno5(n: int) -> list[float]:
    return list(map((lambda k: sin(1/k)/(1/k)), range(1, n+1)))

# 6ª definición de aproxLimSeno
# =============================

def aproxLimSeno6(n: int) -> list[float]:
    r = []
    for k in range(1, n+1):
        r.append(sin(1/k)/(1/k))
    return r

# Comprobación de equivalencia de aproxLimSeno
# ============================================

# La propiedad es
@given(st.integers(min_value=1, max_value=100))
def test_aproxLimSeno(n: int) -> None:
    r = aproxLimSeno1(n)
    assert aproxLimSeno2(n) == r
    assert aproxLimSeno3(n) == r
    assert aproxLimSeno4(n) == r
    assert aproxLimSeno5(n) == r
    assert aproxLimSeno6(n) == r

# La comprobación es
#    src> poetry run pytest -q limite_del_seno.py
#    1 passed in 0.60s

# Comparación de eficiencia de aproxLimSeno
# =========================================

def tiempo(e: str) -> None:
```

```
    """Tiempo (en segundos) de evaluar la expresión e."""
    t = Timer(e, "", default_timer, globals()).timeit(1)
    print(f"{t:0.2f} segundos")

# La comparación es
#    >>> tiempo('aproxLimSeno1(3*10**5)')
#    0.03 segundos
#    >>> tiempo('aproxLimSeno2(3*10**5)')
#    Process Python violación de segmento (core dumped)
#    >>> tiempo('aproxLimSeno3(3*10**5)')
#    Process Python violación de segmento (core dumped)
#    >>> tiempo('aproxLimSeno4(3*10**5)')
#    Process Python violación de segmento (core dumped)
#    >>> tiempo('aproxLimSeno5(3*10**5)')
#    0.04 segundos
#    >>> tiempo('aproxLimSeno6(3*10**5)')
#    0.07 segundos
#
#    >>> tiempo('aproxLimSeno1(10**7)')
#    1.29 segundos
#    >>> tiempo('aproxLimSeno5(10**7)')
#    1.40 segundos
#    >>> tiempo('aproxLimSeno6(10**7)')
#    1.45 segundos

# 1ª definición de errorLimSeno
# ============================

# naturales es el generador de los números naturales positivos, Por
# ejemplo,
#    >>> list(islice(naturales(), 5))
#    [1, 2, 3, 4, 5]
def naturales() -> Iterator[int]:
    i = 1
    while True:
        yield i
        i += 1

def errorLimSeno1(x: float) -> int:
    return list(islice((n for n in naturales()
```

```
                        if abs(1 - sin(1/n)/(1/n)) < x), 1))[0]

# 2ª definición de errorLimSeno
# ============================

def errorLimSeno2(x: float) -> int:
    def aux(n: int) -> int:
        if abs(1 - sin(1/n)/(1/n)) < x:
            return n
        return aux(n + 1)

    return aux(1)

# 3ª definición de errorLimSeno
# =============================

def errorLimSeno3(x: float) -> int:
    return list(islice(dropwhile(lambda n: abs(1 - sin(1/n)/(1/n)) >= x,
                                 naturales()),
                       1))[0]


# Comprobación de equivalencia de errorLimSeno
# ============================================

@given(st.integers(min_value=1, max_value=100))
def test_errorLimSeno(n: int) -> None:
    r = errorLimSeno1(n)
    assert errorLimSeno2(n) == r
    assert errorLimSeno3(n) == r

# La comprobación es
#    src> poetry run pytest -q limite_del_seno.py
#    2 passed in 0.60s

# Comparación de eficiencia de errorLimSeno
# =========================================

# La comparación es
#    >>> tiempo('errorLimSeno1(10**(-12))')
```

```
#    0.07 segundos
#    >>> tiempo('errorLimSeno2(10**(-12))')
#    Process Python violación de segmento (core dumped)
#    >>> tiempo('errorLimSeno3(10**(-12))')
#    0.10 segundos
```

2.23. Cálculo del número π mediante la fórmula de Leibniz

En Haskell

```
-- ---------------------------------------------------------------------
-- El número π puede calcularse con la [fórmula de
-- Leibniz](https://bit.ly/3ERCwZd)
--    π/4 = 1 - 1/3 + 1/5 - 1/7 + ...+ (-1)**n/(2*n+1) + ...
--
-- Definir las funciones
--    calculaPi :: Int -> Double
--    errorPi   :: Double -> Int
-- tales que
-- + (calculaPi n) es la aproximación del número π calculada
--   mediante la expresión
--      4*(1 - 1/3 + 1/5 - 1/7 + ...+ (-1)**n/(2*n+1))
--   Por ejemplo,
--      calculaPi 3    ==  2.8952380952380956
--      calculaPi 300  ==  3.1449149035588526
-- + (errorPi x) es el menor número de términos de la serie
--    necesarios para obtener pi con un error menor que x. Por ejemplo,
--      errorPi 0.1    ==    9
--      errorPi 0.01   ==   99
--      errorPi 0.001  ==  999
-- ---------------------------------------------------------------------

module Calculo_de_pi_mediante_la_formula_de_Leibniz where

import Test.QuickCheck

-- 1ª definición de calculaPi
-- ==========================
```

```
calculaPi1 :: Int -> Double
calculaPi1 k = 4 * sum [(-1)**n/(2*n+1) | n <- [0..k']]
  where k' = fromIntegral k

-- 2ª definición de calculaPi
-- ==========================

calculaPi2 :: Int -> Double
calculaPi2 0 = 4
calculaPi2 n = calculaPi2 (n-1) + 4*(-1)**n'/(2*n'+1)
  where n' = fromIntegral n

-- 3ª definición de calculaPi
-- ==========================

calculaPi3 :: Int -> Double
calculaPi3 = aux . fromIntegral
  where aux 0 = 4
        aux n = 4*(-1)**n/(2*n+1) + aux (n-1)

-- Comprobación de equivalencia de calculaPi
-- =========================================

-- La propiedad es
prop_calculaPi :: Positive Int -> Bool
prop_calculaPi (Positive k) =
  all (== calculaPi1 k)
      [calculaPi2 k,
       calculaPi3 k]

-- La comprobación es
--    λ> quickCheck prop_calculaPi
--    +++ OK, passed 100 tests.

-- Comparación de eficiencia de calculaPi
-- ======================================

-- La comparación es
--    λ> calculaPi1 (10^6)
```

```haskell
--    3.1415936535887745
--    (1.31 secs, 609,797,408 bytes)
--    λ> calculaPi2 (10^6)
--    3.1415936535887745
--    (1.68 secs, 723,032,272 bytes)
--    λ> calculaPi3 (10^6)
--    3.1415936535887745
--    (2.22 secs, 1,099,032,608 bytes)


-- 1ª definición de errorPi
-- ========================

errorPi1 :: Double -> Int
errorPi1 x =
  head [n | n <- [1..]
          , abs (pi - calculaPi1 n) < x]

-- 2ª definición de errorPi
-- ========================

errorPi2 :: Double -> Int
errorPi2 x = aux 1
  where aux n | abs (pi - calculaPi1 n) < x = n
              | otherwise                   = aux (n+1)

-- Comprobación de equivalencia de errorPi
-- =============================================

-- La propiedad es
prop_errorPi :: Positive Double -> Bool
prop_errorPi (Positive x) =
  errorPi1 x == errorPi2 x

-- La comprobación es
--    λ> quickCheck prop_errorPi
--    +++ OK, passed 100 tests.

-- Comparación de eficiencia de errorPi
-- ====================================
```

```
-- La comparación es
--    λ> errorPi1 0.0005
--    1999
--    (1.88 secs, 1,189,226,384 bytes)
--    λ> errorPi2 0.0005
--    1999
--    (1.87 secs, 1,213,341,096 bytes)
```

En Python

```
# ---------------------------------------------------------------------
# El número π puede calcularse con la [fórmula de
# Leibniz](https://bit.ly/3ERCwZd)
#    π/4 = 1 - 1/3 + 1/5 - 1/7 + ...+ (-1)**n/(2*n+1) + ...
#
# Definir las funciones
#    calculaPi : (int) -> float
#    errorPi   : (float) -> int
# tales que
# + calculaPi(n) es la aproximación del número π calculada
#   mediante la expresión
#      4*(1 - 1/3 + 1/5 - 1/7 + ...+ (-1)**n/(2*n+1))
#   Por ejemplo,
#      calculaPi(3)    ==  2.8952380952380956
#      calculaPi(300)  ==  3.1449149035588526
# + errorPi(x) es el menor número de términos de la serie
#   necesarios para obtener pi con un error menor que x. Por ejemplo,
#      errorPi(0.1)    ==    9
#      errorPi(0.01)   ==   99
#      errorPi(0.001)  ==  999
# ---------------------------------------------------------------------

from itertools import islice
from math import pi
from sys import setrecursionlimit
from timeit import Timer, default_timer
from typing import Iterator

from hypothesis import given
```

```
from hypothesis import strategies as st

setrecursionlimit(10**6)

# 1ª definición de calculaPi
# ==========================

def calculaPi1(k: int) -> float:
    return 4 * sum(((-1)**n/(2*n+1) for n in range(0, k+1)))

# 2ª definición de calculaPi
# ==========================

def calculaPi2(n: int) -> float:
    if n == 0:
        return 4
    return calculaPi2(n-1) + 4*(-1)**n/(2*n+1)

# 3ª definición de calculaPi
# ==========================

def calculaPi3(n: int) -> float:
    r = 1
    for k in range(1, n+1):
        r = r + (-1)**k/(2*k+1)
    return 4 * r

# Comprobación de equivalencia de calculaPi
# =========================================

# La propiedad es
@given(st.integers(min_value=1, max_value=100))
def test_calculaPi(n: int) -> None:
    r = calculaPi1(n)
    assert calculaPi2(n) == r
    assert calculaPi3(n) == r

# La comprobación es
#    src> poetry run pytest -q calculo_de_pi_mediante_la_formula_de_Leibniz.py
#    1 passed in 0.14s
```

```
# Comparación de eficiencia de calculaPi
# ======================================

def tiempo(e: str) -> None:
    """Tiempo (en segundos) de evaluar la expresión e."""
    t = Timer(e, "", default_timer, globals()).timeit(1)
    print(f"{t:0.2f} segundos")

# La comparación es
#    >>> tiempo('calculaPi1(10**6)')
#    0.37 segundos
#    >>> tiempo('calculaPi2(10**6)')
#    Process Python violación de segmento (core dumped)
#    >>> tiempo('calculaPi3(10**6)')
#    0.39 segundos

# 1ª definición de errorPi
# ========================

# naturales es el generador de los números naturales positivos, Por
# ejemplo,
#    >>> list(islice(naturales(), 5))
#    [1, 2, 3, 4, 5]
def naturales() -> Iterator[int]:
    i = 1
    while True:
        yield i
        i += 1

def errorPi1(x: float) -> int:
    return list(islice((n for n in naturales()
                        if abs(pi - calculaPi1(n)) < x), 1))[0]

# 2ª definición de errorPi
# ========================

def errorPi2(x: float) -> int:
    def aux(n: int) -> int:
        if abs(pi - calculaPi1(n)) < x:
```

```
            return n
        return aux(n + 1)

    return aux(1)

# Comprobación de equivalencia de errorPi
# =======================================

@given(st.integers(min_value=1, max_value=100))
def test_errorPi(n: int) -> None:
    assert errorPi1(n) == errorPi2(n)

# La comprobación es
#    src> poetry run pytest -q calculo_de_pi_mediante_la_formula_de_Leibniz.py
#    2 passed in 0.60s

# Comparación de eficiencia de errorPi
# ====================================

# La comparación es
#    >>> tiempo('errorPi1(0.0005)')
#    0.63 segundos
#    >>> tiempo('errorPi2(0.0005)')
#    0.58 segundos
```

2.24. Ternas pitagóricas

En Haskell

```
-- ---------------------------------------------------------------------
-- Una terna (x,y,z) de enteros positivos es pitagórica si x^2 + y^2 =
-- z^2 y x < y < z.
--
-- Definir, por comprensión, la función
--    pitagoricas :: Int -> [(Int,Int,Int)]
-- tal que (pitagoricas n) es la lista de todas las ternas pitagóricas
-- cuyas componentes están entre 1 y n. Por ejemplo,
--    pitagoricas 10  ==  [(3,4,5),(6,8,10)]
--    pitagoricas 15  ==  [(3,4,5),(5,12,13),(6,8,10),(9,12,15)]
-- ---------------------------------------------------------------------
```

```
module Ternas_pitagoricas where

import Test.QuickCheck

-- 1ª solución
-- ===========

pitagoricas1 :: Int -> [(Int,Int,Int)]
pitagoricas1 n = [(x,y,z) | x <- [1..n]
                          , y <- [1..n]
                          , z <- [1..n]
                          , x^2 + y^2 == z^2
                          , x < y && y < z]

-- 2ª solución
-- ===========

pitagoricas2 :: Int -> [(Int,Int,Int)]
pitagoricas2 n = [(x,y,z) | x <- [1..n]
                          , y <- [x+1..n]
                          , z <- [ceiling (sqrt (fromIntegral (x^2+y^2)))..n]
                          , x^2 + y^2 == z^2]

-- 3ª solución
-- ===========

pitagoricas3 :: Int -> [(Int,Int,Int)]
pitagoricas3 n = [(x,y,z) | x <- [1..n]
                          , y <- [x+1..n]
                          , let z = round (sqrt (fromIntegral (x^2+y^2)))
                          , y < z
                          , z <= n
                          , x^2 + y^2 == z^2]

-- Comprobación de equivalencia
-- ============================

-- La propiedad es
prop_pitagoricas :: Positive Int -> Bool
```

```
prop_pitagoricas (Positive n) =
  all (== pitagoricas1 n)
      [pitagoricas2 n,
       pitagoricas3 n]

-- La comprobación es
--    λ> quickCheck prop_pitagoricas
--    +++ OK, passed 100 tests.

-- Comparación de eficiencia
-- =========================

-- La comparación es
--    λ> length (pitagoricas1 200)
--    127
--    (12.25 secs, 12,680,320,400 bytes)
--    λ> length (pitagoricas2 200)
--    127
--    (1.61 secs, 1,679,376,824 bytes)
--    λ> length (pitagoricas3 200)
--    127
--    (0.06 secs, 55,837,072 bytes)
```

En Python

```
# ---------------------------------------------------------------------
# Una terna (x,y,z) de enteros positivos es pitagórica si x^2 + y^2 =
# z^2 y x < y < z.
#
# Definir, por comprensión, la función
#    pitagoricas : (int) -> list[tuple[int,int,int]]
# tal que pitagoricas(n) es la lista de todas las ternas pitagóricas
# cuyas componentes están entre 1 y n. Por ejemplo,
#    pitagoricas(10) == [(3, 4, 5), (6, 8, 10)]
#    pitagoricas(15) == [(3, 4, 5), (5, 12, 13), (6, 8, 10), (9, 12, 15)]
# ---------------------------------------------------------------------

from math import ceil, sqrt
from timeit import Timer, default_timer
```

```python
from hypothesis import given
from hypothesis import strategies as st

# 1ª solución
# ===========

def pitagoricas1(n: int) -> list[tuple[int, int, int]]:
    return [(x, y, z)
            for x in range(1, n+1)
            for y in range(1, n+1)
            for z in range(1, n+1)
            if x**2 + y**2 == z**2 and x < y < z]

# 2ª solución
# ===========

def pitagoricas2(n: int) -> list[tuple[int, int, int]]:
    return [(x, y, z)
            for x in range(1, n+1)
            for y in range(x+1, n+1)
            for z in range(ceil(sqrt(x**2+y**2)), n+1)
            if x**2 + y**2 == z**2]

# 3ª solución
# ===========

def pitagoricas3(n: int) -> list[tuple[int, int, int]]:
    return [(x, y, z)
            for x in range(1, n+1)
            for y in range(x+1, n+1)
            for z in [ceil(sqrt(x**2+y**2))]
            if y < z <= n and x**2 + y**2 == z**2]

# Comprobación de equivalencia
# ============================

# La propiedad es
@given(st.integers(min_value=1, max_value=50))
def test_pitagoricas(n: int) -> None:
    r = pitagoricas1(n)
```

```
    assert pitagoricas2(n) == r
    assert pitagoricas3(n) == r

# La comprobación es
#    src> poetry run pytest -q ternas_pitagoricas.py
#    1 passed in 1.83s

# Comparación de eficiencia
# =========================

def tiempo(e: str) -> None:
    """Tiempo (en segundos) de evaluar la expresión e."""
    t = Timer(e, "", default_timer, globals()).timeit(1)
    print(f"{t:0.2f} segundos")

# La comparación es
#    >>> tiempo('pitagoricas1(200)')
#    4.76 segundos
#    >>> tiempo('pitagoricas2(200)')
#    0.69 segundos
#    >>> tiempo('pitagoricas3(200)')
#    0.02 segundos
```

2.25. Ternas pitagóricas con suma dada

En Haskell

```
-- ---------------------------------------------------------------------
-- Una terna pitagórica es una terna de números naturales (a,b,c) tal
-- que a<b<c y a^2+b^2=c^2. Por ejemplo (3,4,5) es una terna pitagórica.
--
-- Definir la función
--    ternasPitagoricas :: Integer -> [(Integer,Integer,Integer)]
-- tal que (ternasPitagoricas x) es la lista de las ternas pitagóricas
-- cuya suma es x. Por ejemplo,
--    ternasPitagoricas 12     == [(3,4,5)]
--    ternasPitagoricas 60     == [(10,24,26),(15,20,25)]
--    ternasPitagoricas (10^6) == [(218750,360000,421250),(200000,375000,425000)]
-- ---------------------------------------------------------------------
```

```haskell
module Ternas_pitagoricas_con_suma_dada where

import Data.List (nub,sort)
import Test.QuickCheck

-- 1ª solución                                                    --
-- ===========

ternasPitagoricas1 :: Integer -> [(Integer,Integer,Integer)]
ternasPitagoricas1 x =
  [(a,b,c) | a <- [0..x],
             b <- [a+1..x],
             c <- [b+1..x],
             a^2 + b^2 == c^2,
             a+b+c == x]

-- 2ª solución                                                    --
-- ===========

ternasPitagoricas2 :: Integer -> [(Integer,Integer,Integer)]
ternasPitagoricas2 x =
  [(a,b,c) | a <- [1..x],
             b <- [a+1..x-a],
             let c = x-a-b,
             a^2+b^2 == c^2]

-- 3ª solución                                                    --
-- ===========

-- Todas las ternas pitagóricas primitivas (a,b,c) pueden representarse
-- por
--    a = m^2 - n^2, b = 2*m*n, c = m^2 + n^2,
-- con 1 <= n < m. (Ver en https://bit.ly/35UNY6L ).

ternasPitagoricas3 :: Integer -> [(Integer,Integer,Integer)]
ternasPitagoricas3 x =
  nub [(d*a,d*b,d*c) | d <- [1..x],
                       x `mod` d == 0,
                       (a,b,c) <- aux (x `div` d)]
  where
```

```
    aux y = [(a,b,c) | m <- [2..limite],
                       n <- [1..m-1],
                       let [a,b] = sort [m^2 - n^2, 2*m*n],
                       let c = m^2 + n^2,
                       a+b+c == y]
      where limite = ceiling (sqrt (fromIntegral y))

-- Equivalencia de las definiciones
-- ================================

-- La propiedad es
prop_ternasPitagoricas :: Positive Integer -> Bool
prop_ternasPitagoricas (Positive x) =
  all (== (ternasPitagoricas1 x))
      [ternasPitagoricas2 x,
       ternasPitagoricas3 x]

-- La comprobación es
--    λ> quickCheck prop_ternasPitagoricas
--    +++ OK, passed 100 tests.

-- Comparación de eficiencia
-- =========================

-- La comparación es
--    λ> ternasPitagoricas1 200
--    [(40,75,85)]
--    (1.90 secs, 2,404,800,856 bytes)
--    λ> ternasPitagoricas2 200
--    [(40,75,85)]
--    (0.06 secs, 19,334,232 bytes)
--    λ> ternasPitagoricas3 200
--    [(40,75,85)]
--    (0.01 secs, 994,224 bytes)
--
--    λ> ternasPitagoricas2 3000
--    [(500,1200,1300),(600,1125,1275),(750,1000,1250)]
--    (4.41 secs, 4,354,148,136 bytes)
--    λ> ternasPitagoricas3 3000
--    [(500,1200,1300),(600,1125,1275),(750,1000,1250)]
```

```
--    (0.05 secs, 17,110,360 bytes)
```

En Python

```python
# ---------------------------------------------------------------------
# Una terna pitagórica es una terna de números naturales (a,b,c) tal
# que a<b<c y a^2+b^2=c^2. Por ejemplo (3,4,5) es una terna pitagórica.
#
# Definir la función
#    ternasPitagoricas : (int) -> list[tuple[int, int, int]]
# tal que ternasPitagoricas(x) es la lista de las ternas pitagóricas
# cuya suma es x. Por ejemplo,
#    ternasPitagoricas(12)    == [(3, 4, 5)]
#    ternasPitagoricas(60)    == [(10, 24, 26), (15, 20, 25)]
#    ternasPitagoricas(10**6) == [(218750, 360000, 421250),
#                                 (200000, 375000, 425000)]
# ---------------------------------------------------------------------

from math import ceil, sqrt
from timeit import Timer, default_timer

from hypothesis import given
from hypothesis import strategies as st

# 1ª solución                                                        --
# ===========

def ternasPitagoricas1(x: int) -> list[tuple[int, int, int]]:
    return [(a, b, c)
            for a in range(0, x+1)
            for b in range(a+1, x+1)
            for c in range(b+1, x+1)
            if a**2 + b**2 == c**2 and a + b + c == x]

# 2ª solución                                                        --
# ===========

def ternasPitagoricas2(x: int) -> list[tuple[int, int, int]]:
    return [(a, b, c)
            for a in range(1, x+1)
```

```
            for b in range(a+1, x-a+1)
            for c in [x - a - b]
            if a**2 + b**2 == c**2]

# 3ª solución                                                       --
# ===========

# Todas las ternas pitagóricas primitivas (a,b,c) pueden representarse
# por
#    a = m^2 - n^2, b = 2*m*n, c = m^2 + n^2,
# con 1 <= n < m. (Ver en https://bit.ly/35UNY6L ).

def ternasPitagoricas3(x: int) -> list[tuple[int, int, int]]:
    def aux(y: int) -> list[tuple[int, int, int]]:
        return [(a, b, c)
                for m in range(2, 1 + ceil(sqrt(y)))
                for n in range(1, m)
                for a in [min(m**2 - n**2, 2*m*n)]
                for b in [max(m**2 - n**2, 2*m*n)]
                for c in [m**2 + n**2]
                if a+b+c == y]

    return list(set(((d*a, d*b, d*c)
                     for d in range(1, x+1)
                     for (a, b, c) in aux(x // d)
                     if x % d == 0)))

# Comprobación de equivalencia
# ============================

# La propiedad es
@given(st.integers(min_value=1, max_value=50))
def test_ternasPitagoricas(n: int) -> None:
    r = set(ternasPitagoricas1(n))
    assert set(ternasPitagoricas2(n)) == r
    assert set(ternasPitagoricas3(n)) == r

# La comprobación es
#    src> poetry run pytest -q ternas_pitagoricas_con_suma_dada.py
#    1 passed in 0.35s
```

```
# Comparación de eficiencia
# =========================

def tiempo(e: str) -> None:
    """Tiempo (en segundos) de evaluar la expresión e."""
    t = Timer(e, "", default_timer, globals()).timeit(1)
    print(f"{t:0.2f} segundos")

# La comparación es
#    >>> tiempo('ternasPitagoricas1(300)')
#    2.83 segundos
#    >>> tiempo('ternasPitagoricas2(300)')
#    0.01 segundos
#    >>> tiempo('ternasPitagoricas3(300)')
#    0.00 segundos
#
#    >>> tiempo('ternasPitagoricas2(3000)')
#    1.48 segundos
#    >>> tiempo('ternasPitagoricas3(3000)')
#    0.02 segundos
```

2.26. Producto escalar

En Haskell

```
-- ---------------------------------------------------------------------
-- El producto escalar de dos listas de enteros xs y ys de longitud n
-- viene dado por la suma de los productos de los elementos
-- correspondientes.
--
-- Definir la función
--    productoEscalar :: [Integer] -> [Integer] -> Integer
-- tal que (productoEscalar xs ys) es el producto escalar de las listas
-- xs e ys. Por ejemplo,
--    productoEscalar [1,2,3] [4,5,6]  ==  32
-- ---------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-unused-imports #-}
```

```
module Producto_escalar where

import Numeric.LinearAlgebra ((<.>), vector)
import Test.QuickCheck (quickCheck)

-- 1ª solución
-- ===========

productoEscalar1 :: [Integer] -> [Integer] -> Integer
productoEscalar1 xs ys = sum [x*y | (x,y) <- zip xs ys]

-- 2ª solución
-- ===========

productoEscalar2 :: [Integer] -> [Integer] -> Integer
productoEscalar2 xs ys = sum (zipWith (*) xs ys)

-- 3ª solución
-- ===========

productoEscalar3 :: [Integer] -> [Integer] -> Integer
productoEscalar3 = (sum .) . zipWith (*)

-- 4ª solución
-- ===========

productoEscalar4 :: [Integer] -> [Integer] -> Integer
productoEscalar4 [] _           = 0
productoEscalar4 _ []           = 0
productoEscalar4 (x:xs) (y:ys) = x*y + productoEscalar4 xs ys

-- 5ª solución
-- ===========

productoEscalar5 :: [Integer] -> [Integer] -> Integer
productoEscalar5 (x:xs) (y:ys) = x*y + productoEscalar5 xs ys
productoEscalar5 _ _           = 0

-- 6ª solución
-- ===========
```

```
productoEscalar6 :: [Integer] -> [Integer] -> Integer
productoEscalar6 xs ys =
  round (vector xs' <.> vector ys')
  where xs' = map fromIntegral xs
        ys' = map fromIntegral ys

-- Comprobación de equivalencia
-- ============================

-- La propiedad es
prop_productoEscalar :: [Integer] -> [Integer] -> Bool
prop_productoEscalar xs ys =
  all (== productoEscalar1 xs ys)
      [productoEscalar2 xs ys,
       productoEscalar3 xs ys,
       productoEscalar4 xs ys,
       productoEscalar5 xs ys,
       productoEscalar6 xs' ys']
  where n = min (length xs) (length ys)
        xs' = take n xs
        ys' = take n ys

-- La comprobación es
--    λ> quickCheck prop_productoEscalar
--    +++ OK, passed 100 tests.

-- Comparación de eficiencia
-- =========================

-- La comparación es
--    λ> productoEscalar1 (replicate (2*10^6) 1) (replicate (2*10^6) 1)
--    2000000
--    (1.37 secs, 803,827,520 bytes)
--    λ> productoEscalar2 (replicate (2*10^6) 1) (replicate (2*10^6) 1)
--    2000000
--    (0.69 secs, 611,008,272 bytes)
--    λ> productoEscalar3 (replicate (2*10^6) 1) (replicate (2*10^6) 1)
--    2000000
--    (0.69 secs, 611,008,536 bytes)
```

```
--     λ> productoEscalar4 (replicate (2*10^6) 1) (replicate (2*10^6) 1)
--     2000000
--     (1.64 secs, 742,290,272 bytes)
--     λ> productoEscalar5 (replicate (2*10^6) 1) (replicate (2*10^6) 1)
--     2000000
--     (1.63 secs, 742,290,064 bytes)
--     λ> productoEscalar6 (replicate (2*10^6) 1) (replicate (2*10^6) 1)
--     2000000
--     (0.32 secs, 835,679,200 bytes)
--
--     λ> productoEscalar2 (replicate (6*10^6) 1) (replicate (6*10^6) 1)
--     6000000
--     (1.90 secs, 1,831,960,336 bytes)
--     λ> productoEscalar3 (replicate (6*10^6) 1) (replicate (6*10^6) 1)
--     6000000
--     (1.87 secs, 1,831,960,600 bytes)
--     λ> productoEscalar6 (replicate (6*10^6) 1) (replicate (6*10^6) 1)
--     6000000
--     (0.78 secs, 2,573,005,952 bytes)
```

En Python

```
# ---------------------------------------------------------------------
# El producto escalar de dos listas de enteros xs y ys de longitud n
# viene dado por la suma de los productos de los elementos
# correspondientes.
#
# Definir la función
#    productoEscalar : (list[int], list[int]) -> int
# tal que productoEscalar(xs, ys) es el producto escalar de las listas
# xs e ys. Por ejemplo,
#    productoEscalar([1, 2, 3], [4, 5, 6])  ==  32
# ---------------------------------------------------------------------

from operator import mul
from sys import setrecursionlimit
from timeit import Timer, default_timer

from hypothesis import given
from hypothesis import strategies as st
```

```
from numpy import dot

setrecursionlimit(10**6)

# 1ª solución
# ===========

def productoEscalar1(xs: list[int], ys: list[int]) -> int:
    return sum(x * y for (x, y) in zip(xs, ys))

# 2ª solución
# ===========

def productoEscalar2(xs: list[int], ys: list[int]) -> int:
    return sum(map(mul, xs, ys))

# 3ª solución
# ===========

def productoEscalar3(xs: list[int], ys: list[int]) -> int:
    if xs and ys:
        return xs[0] * ys[0] + productoEscalar3(xs[1:], ys[1:])
    return 0

# 4ª solución
# ===========

def productoEscalar4(xs: list[int], ys: list[int]) -> int:
    return dot(xs, ys)

# Comprobación de equivalencia
# ============================

# La propiedad es
@given(st.lists(st.integers(min_value=1, max_value=100)),
       st.lists(st.integers(min_value=1, max_value=100)))
def test_productoEscalar(xs: list[int], ys: list[int]) -> None:
    r = productoEscalar1(xs, ys)
    assert productoEscalar2(xs, ys) == r
    assert productoEscalar3(xs, ys) == r
```

```
    n = min(len(xs), len(ys))
    xs1 = xs[:n]
    ys1 = ys[:n]
    assert productoEscalar4(xs1, ys1) == r

# La comprobación es
#    src> poetry run pytest -q producto_escalar.py
#    1 passed in 0.37s

# Comparación de eficiencia
# =========================

def tiempo(e: str) -> None:
    """Tiempo (en segundos) de evaluar la expresión e."""
    t = Timer(e, "", default_timer, globals()).timeit(1)
    print(f"{t:0.2f} segundos")

# La comparación es
#    >>> tiempo('productoEscalar1([1]*(10**4), [1]*(10**4))')
#    0.00 segundos
#    >>> tiempo('productoEscalar3([1]*(10**4), [1]*(10**4))')
#    0.55 segundos
#
#    >>> tiempo('productoEscalar1([1]*(10**7), [1]*(10**7))')
#    0.60 segundos
#    >>> tiempo('productoEscalar2([1]*(10**7), [1]*(10**7))')
#    0.26 segundos
#    >>> tiempo('productoEscalar4([1]*(10**7), [1]*(10**7))')
#    1.73 segundos
```

2.27. Representación densa de polinomios

En Haskell

```
-- ---------------------------------------------------------------------
-- Los polinomios pueden representarse de forma dispersa o densa. Por
-- ejemplo, el polinomio 6x^4-5x^2+4x-7 se puede representar de forma
-- dispersa por [6,0,-5,4,-7] y de forma densa por [(4,6),(2,-5),(1,4),(0,-7)].
--
-- Definir la función
```

```
--    densa :: [Int] -> [(Int,Int)]
-- tal que (densa xs) es la representación densa del polinomio cuya
-- representación dispersa es xs. Por ejemplo,
--   densa [6,0,-5,4,-7]  ==  [(4,6),(2,-5),(1,4),(0,-7)]
--   densa [6,0,0,3,0,4]  ==  [(5,6),(2,3),(0,4)]
--   densa [0]            ==  [(0,0)]
-- ---------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-incomplete-patterns #-}

module Representacion_densa_de_polinomios where

import Test.QuickCheck

-- 1ª solución
-- ===========

densa1 :: [Int] -> [(Int,Int)]
densa1 xs =
  [(x,y) | (x,y) <- zip [n-1,n-2..1] xs, y /= 0]
  ++ [(0, last xs)]
  where n = length xs

-- 2ª solución
-- ===========

densa2 :: [Int] -> [(Int,Int)]
densa2 xs =
  filter (\ (_,y) -> y /= 0) (zip [n-1,n-2..1] xs)
  ++ [(0, last xs)]
  where n = length xs

-- 3ª solución
-- ===========

densa3 :: [Int] -> [(Int,Int)]
densa3 xs = filter ((/= 0) . snd) (zip [n-1,n-2..1] xs)
  ++ [(0, last xs)]
  where n = length xs
```

```haskell
-- 4ª solución
-- ===========

densa4 :: [Int] -> [(Int,Int)]
densa4 xs = aux xs (length xs - 1)
  where aux [y] 0 = [(0, y)]
        aux (y:ys) n | y == 0    = aux ys (n-1)
                     | otherwise = (n,y) : aux ys (n-1)

-- Comprobación de equivalencia
-- ============================

-- La propiedad es
prop_densa :: NonEmptyList Int -> Bool
prop_densa (NonEmpty xs) =
  all (== densa1 xs)
      [densa2 xs,
       densa3 xs,
       densa4 xs]

-- La comprobación es
--    λ> quickCheck prop_densa
--    +++ OK, passed 100 tests.

-- Comparación de eficiencia
-- =========================

-- La comparación es
--    λ> last (densa1 [1..2*10^6])
--    (0,2000000)
--    (0.95 secs, 880,569,400 bytes)
--    λ> last (densa2 [1..2*10^6])
--    (0,2000000)
--    (0.52 secs, 800,569,432 bytes)
--    λ> last (densa3 [1..2*10^6])
--    (0,2000000)
--    (0.53 secs, 752,569,552 bytes)
--    λ> last (densa4 [1..2*10^6])
--    (0,2000000)
--    (3.05 secs, 1,267,842,032 bytes)
```

```
--
--    λ> last (densa1 [1..10^7])
--    (0,10000000)
--    (5.43 secs, 4,400,570,128 bytes)
--    λ> last (densa2 [1..10^7])
--    (0,10000000)
--    (3.03 secs, 4,000,570,160 bytes)
--    λ> last (densa3 [1..10^7])
--    (0,10000000)
--    (2.34 secs, 3,760,570,280 bytes)
```

En Python

```
# ---------------------------------------------------------------------
# Los polinomios pueden representarse de forma dispersa o densa. Por
# ejemplo, el polinomio 6x^4-5x^2+4x-7 se puede representar de forma
# dispersa por [6,0,-5,4,-7] y de forma densa por
# [(4,6),(2,-5),(1,4),(0,-7)].
#
# Definir la función
#    densa : (list[int]) -> list[tuple[int, int]]
# tal que densa(xs) es la representación densa del polinomio cuya
# representación dispersa es xs. Por ejemplo,
#   densa([6, 0, -5, 4, -7])  == [(4, 6), (2, -5), (1, 4), (0, -7)]
#   densa([6, 0, 0, 3, 0, 4]) == [(5, 6), (2, 3), (0, 4)]
#   densa([0])                == [(0, 0)]
# ---------------------------------------------------------------------

from sys import setrecursionlimit
from timeit import Timer, default_timer

from hypothesis import given
from hypothesis import strategies as st

setrecursionlimit(10**6)

# 1ª solución
# ===========

def densa1(xs: list[int]) -> list[tuple[int, int]]:
```

```
    n = len(xs)
    return [(x, y)
            for (x, y) in zip(range(n-1, 0, -1), xs)
            if y != 0] + [(0, xs[-1])]

# 2ª solución
# ===========

def densa2(xs: list[int]) -> list[tuple[int, int]]:
    n = len(xs)
    return list(filter(lambda p: p[1] != 0,
                       zip(range(n-1, 0, -1), xs))) + [(0, xs[-1])]

# 3ª solución
# ===========

def densa3(xs: list[int]) -> list[tuple[int, int]]:
    def aux(ys: list[int], n: int) -> list[tuple[int, int]]:
        if n == 0:
            return [(0, ys[0])]
        if ys[0] == 0:
            return aux(ys[1:], n-1)
        return [(n, ys[0])] + aux(ys[1:], n-1)

    return aux(xs, len(xs) - 1)

# Comprobación de equivalencia
# ============================

# La propiedad es
@given(st.lists(st.integers(), min_size=1))
def test_densa(xs: list[int]) -> None:
    r = densa1(xs)
    assert densa2(xs) == r
    assert densa3(xs) == r

# La comprobación es
#    src> poetry run pytest -q representacion_densa_de_polinomios.py
#    1 passed in 0.27s
```

```
# Comparación de eficiencia
# ========================

def tiempo(e: str) -> None:
    """Tiempo (en segundos) de evaluar la expresión e."""
    t = Timer(e, "", default_timer, globals()).timeit(1)
    print(f"{t:0.2f} segundos")

# La comparación es
#    >>> tiempo('densa1(range(1, 10**4))')
#    0.00 segundos
#    >>> tiempo('densa2(range(1, 10**4))')
#    0.00 segundos
#    >>> tiempo('densa3(range(1, 10**4))')
#    0.25 segundos
#
#    >>> tiempo('densa1(range(1, 10**7))')
#    1.87 segundos
#    >>> tiempo('densa2(range(1, 10**7))')
#    2.15 segundos
```

2.28. Base de datos de actividades.

En Haskell

```
-- ---------------------------------------------------------------------
-- Las bases de datos sobre actividades de personas pueden representarse
-- mediante listas de elementos de la forma (a,b,c,d), donde a es el
-- nombre de la persona, b su actividad, c su fecha de nacimiento y d la
-- de su fallecimiento. Un ejemplo es la siguiente que usaremos a lo
-- largo de este ejercicio,
--    personas :: [(String,String,Int,Int)]
--    personas = [("Cervantes","Literatura",1547,1616),
--                ("Velazquez","Pintura",1599,1660),
--                ("Picasso","Pintura",1881,1973),
--                ("Beethoven","Musica",1770,1823),
--                ("Poincare","Ciencia",1854,1912),
--                ("Quevedo","Literatura",1580,1654),
--                ("Goya","Pintura",1746,1828),
--                ("Einstein","Ciencia",1879,1955),
```

```haskell
--                 ("Mozart","Musica",1756,1791),
--                 ("Botticelli","Pintura",1445,1510),
--                 ("Borromini","Arquitectura",1599,1667),
--                 ("Bach","Musica",1685,1750)]
--
-- Definir las funciones
--    nombres   :: [(String,String,Int,Int)] -> [String]
--    musicos   :: [(String,String,Int,Int)] -> [String]
--    seleccion :: [(String,String,Int,Int)] -> String -> [String]
--    musicos'  :: [(String,String,Int,Int)] -> [String]
--    vivas     :: [(String,String,Int,Int)] -> Int -> [String]
-- tales que
-- + (nombres bd) es la lista de los nombres de las personas de la- base
--   de datos bd. Por ejemplo,
--      λ> nombres personas
--      ["Cervantes","Velazquez","Picasso","Beethoven","Poincare",
--       "Quevedo","Goya","Einstein","Mozart","Botticelli","Borromini",
--       "Bach"]
-- + (musicos bd) es la lista de los nombres de los músicos de la base
--   de datos bd. Por ejemplo,
--      musicos personas  ==  ["Beethoven","Mozart","Bach"]
-- + (seleccion bd m) es la lista de los nombres de las personas de la
--   base de datos bd cuya actividad es m. Por ejemplo,
--      λ> seleccion personas "Pintura"
--      ["Velazquez","Picasso","Goya","Botticelli"]
--      λ> seleccion personas "Musica"
--      ["Beethoven","Mozart","Bach"]
-- + (musicos' bd) es la lista de los nombres de los músicos de la base
--   de datos bd. Por ejemplo,
--      musicos' personas  ==  ["Beethoven","Mozart","Bach"]
-- + (vivas bd a) es la lista de los nombres de las personas de la base
--   de datos bd  que estaban vivas en el año a. Por ejemplo,
--      λ> vivas personas 1600
--      ["Cervantes","Velazquez","Quevedo","Borromini"]
-- ---------------------------------------------------------------------

module Base_de_dato_de_actividades where

personas :: [(String,String,Int,Int)]
personas = [("Cervantes","Literatura",1547,1616),
```

```haskell
            ("Velazquez","Pintura",1599,1660),
            ("Picasso","Pintura",1881,1973),
            ("Beethoven","Musica",1770,1823),
            ("Poincare","Ciencia",1854,1912),
            ("Quevedo","Literatura",1580,1654),
            ("Goya","Pintura",1746,1828),
            ("Einstein","Ciencia",1879,1955),
            ("Mozart","Musica",1756,1791),
            ("Botticelli","Pintura",1445,1510),
            ("Borromini","Arquitectura",1599,1667),
            ("Bach","Musica",1685,1750)]

nombres :: [(String,String,Int,Int)] -> [String]
nombres bd = [x | (x,_,_,_) <- bd]

musicos :: [(String,String,Int,Int)] -> [String]
musicos bd = [x | (x,"Musica",_,_) <- bd]

seleccion :: [(String,String,Int,Int)] -> String -> [String]
seleccion bd m = [ x | (x,m',_,_) <- bd, m == m' ]

musicos' :: [(String,String,Int,Int)] -> [String]
musicos' bd = seleccion bd "Musica"

vivas :: [(String,String,Int,Int)] -> Int -> [String]
vivas bd a = [x | (x,_,a1,a2) <- bd, a1 <= a, a <= a2]
```

En Python

```python
# ---------------------------------------------------------------------
# Las bases de datos sobre actividades de personas pueden representarse
# mediante listas de elementos de la forma (a,b,c,d), donde a es el
# nombre de la persona, b su actividad, c su fecha de nacimiento y d la
# de su fallecimiento. Un ejemplo es la siguiente que usaremos a lo
# largo de este ejercicio,
#    BD = list[tuple[str, str, int, int]]
#
#    personas: BD = [
#        ("Cervantes", "Literatura", 1547, 1616),
#        ("Velazquez", "Pintura", 1599, 1660),
```

```
#        ("Picasso", "Pintura", 1881, 1973),
#        ("Beethoven", "Musica", 1770, 1823),
#        ("Poincare", "Ciencia", 1854, 1912),
#        ("Quevedo", "Literatura", 1580, 1654),
#        ("Goya", "Pintura", 1746, 1828),
#        ("Einstein", "Ciencia", 1879, 1955),
#        ("Mozart", "Musica", 1756, 1791),
#        ("Botticelli", "Pintura", 1445, 1510),
#        ("Borromini", "Arquitectura", 1599, 1667),
#        ("Bach", "Musica", 1685, 1750)]
#
# Definir las funciones
#    nombres   : (BD) -> list[str]
#    musicos   : (BD) -> list[str]
#    seleccion : (BD, str) -> list[str]
#    musicos2  : (BD) -> list[str]
#    vivas     : (BD, int) -> list[str]
# tales que
# + nombres(bd) es la lista de los nombres de las personas de la- base
#   de datos bd. Por ejemplo,
#      >>> nombres(personas)
#      ['Cervantes','Velazquez','Picasso','Beethoven','Poincare',
#       'Quevedo','Goya','Einstein','Mozart','Botticelli','Borromini',
#       'Bach']
# + musicos(bd) es la lista de los nombres de los músicos de la base
#   de datos bd. Por ejemplo,
#      musicos(personas)  ==  ['Beethoven','Mozart','Bach']
# + seleccion(bd, m) es la lista de los nombres de las personas de la
#   base de datos bd cuya actividad es m. Por ejemplo,
#      >>> seleccion(personas, 'Pintura')
#      ['Velazquez', 'Picasso', 'Goya', 'Botticelli']
#      >>> seleccion(personas, 'Musica')
#      ['Beethoven', 'Mozart', 'Bach']
# + musicos2(bd) es la lista de los nombres de los músicos de la base
#   de datos bd. Por ejemplo,
#      musicos2(personas)  ==  ['Beethoven','Mozart','Bach']
# + vivas(bd, a) es la lista de los nombres de las personas de la base
#   de datos bd  que estaban vivas en el año a. Por ejemplo,
#      >>> vivas(personas, 1600)
#      ['Cervantes', 'Velazquez', 'Quevedo', 'Borromini']
```

```python
# ----------------------------------------------------------------------

BD = list[tuple[str, str, int, int]]

personas: BD = [
    ("Cervantes", "Literatura", 1547, 1616),
    ("Velazquez", "Pintura", 1599, 1660),
    ("Picasso", "Pintura", 1881, 1973),
    ("Beethoven", "Musica", 1770, 1823),
    ("Poincare", "Ciencia", 1854, 1912),
    ("Quevedo", "Literatura", 1580, 1654),
    ("Goya", "Pintura", 1746, 1828),
    ("Einstein", "Ciencia", 1879, 1955),
    ("Mozart", "Musica", 1756, 1791),
    ("Botticelli", "Pintura", 1445, 1510),
    ("Borromini", "Arquitectura", 1599, 1667),
    ("Bach", "Musica", 1685, 1750)]

def nombres(bd: BD) -> list[str]:
    return [p[0] for p in bd]

def musicos(bd: BD) -> list[str]:
    return [p[0] for p in bd if p[1] == "Musica"]

def seleccion(bd: BD, m: str) -> list[str]:
    return [p[0] for p in bd if p[1] == m]

def musicos2(bd: BD) -> list[str]:
    return seleccion(bd, "Musica")

def vivas(bd: BD, a: int) -> list[str]:
    return [p[0] for p in bd if p[2] <= a <= p[3]]
```

Capítulo 3

Definiciones por recursión

En este capítulo se presentan ejercicios con definiciones por comprensión. Se corresponden con el tema 6 del curso de programación funcional con Haskell [1].

Contenido

[1] https://jaalonso.github.io/materias/PFconHaskell/temas/tema-6.html

3.1. Potencia entera

En Haskell

```
-- ---------------------------------------------------------------------
-- Definir la función
--    potencia :: Integer -> Integer -> Integer
-- tal que (potencia x n) es x elevado al número natural n. Por ejemplo,
--    potencia 2 3  ==  8
-- ---------------------------------------------------------------------

module Potencia_entera where

import Data.List (foldl')
import Control.Arrow ((***))
import Test.QuickCheck

-- 1ª solución
-- ===========

potencia1 :: Integer -> Integer -> Integer
potencia1 _ 0 = 1
potencia1 m n = m * potencia1 m (n-1)

-- 2ª solución
-- ===========

potencia2 :: Integer -> Integer -> Integer
potencia2 m = aux
  where aux 0 = 1
        aux n = m * aux (n-1)

-- 3ª solución
-- ===========

potencia3 :: Integer -> Integer -> Integer
potencia3 m = aux 1
  where aux r 0 = r
        aux r n = aux (r*m) (n-1)

-- 4ª solución
```

```haskell
-- ===========

potencia4 :: Integer -> Integer -> Integer
potencia4 m = aux 1
  where aux r 0 = r
        aux r n = (aux $! (r*m)) $! (n-1)

-- 5ª solución
-- ===========

potencia5 :: Integer -> Integer -> Integer
potencia5 m n = product [m | _ <- [1..n]]

-- 6ª solución
-- ===========

potencia6 :: Integer -> Integer -> Integer
potencia6 m n = foldl' (*) 1 [m | _ <- [1..n]]

-- 7ª solución
-- ===========

potencia7 :: Integer -> Integer -> Integer
potencia7 m n =
  fst (until (\ (_,k) -> k == n)
             (\ (r,k) -> (r*m, k+1))
             (1,0))

-- 8ª solución
-- ===========

potencia8 :: Integer -> Integer -> Integer
potencia8 m n =
  fst (until ((== n) . snd)
             ((m *) *** (1 +))
             (1,0))

-- 9ª solución
-- ===========
```

```
potencia9 :: Integer -> Integer -> Integer
potencia9 m n = m^n

-- Comprobación de equivalencia
-- ============================

-- La propiedad es
prop_potencia :: Integer -> NonNegative Integer -> Bool
prop_potencia m (NonNegative n) =
  all (== potencia1 m n)
      [potencia2 m n,
       potencia3 m n,
       potencia4 m n,
       potencia5 m n,
       potencia6 m n,
       potencia7 m n,
       potencia8 m n,
       potencia9 m n]

-- La comprobación es
--    λ> quickCheck prop_potencia
--    +++ OK, passed 100 tests.

-- Comparación de eficiencia
-- =========================

-- La comparación es
--    λ> length (show (potencia1 2 (2*10^5)))
--    60206
--    (2.97 secs, 2,602,252,408 bytes)
--    λ> length (show (potencia2 2 (2*10^5)))
--    60206
--    (2.63 secs, 2,624,652,624 bytes)
--    λ> length (show (potencia3 2 (2*10^5)))
--    60206
--    (3.41 secs, 2,619,606,368 bytes)
--    λ> length (show (potencia4 2 (2*10^5)))
--    60206
--    (0.64 secs, 2,636,888,928 bytes)
--    λ> length (show (potencia5 2 (2*10^5)))
```

```
--    60206
--    (2.47 secs, 2,597,108,000 bytes)
--    λ> length (show (potencia6 2 (2*10^5)))
--    60206
--    (0.35 secs, 2,582,488,824 bytes)
--    λ> length (show (potencia7 2 (2*10^5)))
--    60206
--    (2.48 secs, 2,616,406,272 bytes)
--    λ> length (show (potencia8 2 (2*10^5)))
--    60206
--    (2.40 secs, 2,608,652,736 bytes)
--    λ> length (show (potencia9 2 (2*10^5)))
--    60206
--    (0.01 secs, 4,212,968 bytes)
--
--    λ> length (show (potencia4 2 (10^6)))
--    301030
--    (10.39 secs, 63,963,999,656 bytes)
--    λ> length (show (potencia6 2 (10^6)))
--    301030
--    (8.90 secs, 63,691,999,552 bytes)
--    λ> length (show (potencia9 2 (10^6)))
--    301030
--    (0.04 secs, 19,362,032 bytes)
```

En Python

```python
# ---------------------------------------------------------------------
# Definir la función
#    potencia : (int, int) -> int
# tal que potencia(x, n) es x elevado al número natural n. Por ejemplo,
#    potencia(2, 3)  ==  8
# ---------------------------------------------------------------------

from functools import reduce
from operator import mul
from sys import setrecursionlimit
from timeit import Timer, default_timer

from hypothesis import given
```

```python
from hypothesis import strategies as st

setrecursionlimit(10**6)

# 1ª solución
# ===========

def potencia1(m: int, n: int) -> int:
    if n == 0:
        return 1
    return m * potencia1(m, n-1)

# 2ª solución
# ===========

def potencia2(m: int, n: int) -> int:
    def aux(k: int) -> int:
        if k == 0:
            return 1
        return m * aux(k-1)
    return aux(n)

# 3ª solución
# ===========

def potencia3(m: int, n: int) -> int:
    def aux(r: int, k: int) -> int:
        if k == 0:
            return r
        return aux(r*m, k-1)
    return aux(1, n)

# 4ª solución
# ===========

# producto(xs) es el producto de los elementos de xs. Por ejemplo,
#    producto([2, 3, 5]) == 30
def producto(xs: list[int]) -> int:
    return reduce(mul, xs, 1)
```

```
def potencia4(m: int, n: int) -> int:
    return producto([m]*n)

# 5ª solución
# ===========

def potencia5(m: int, n: int) -> int:
    r = 1
    for _ in range(0, n):
        r = r * m
    return r

# 6ª solución
# ===========

def potencia6(m: int, n: int) -> int:
    return m**n

# Comprobación de equivalencia
# ============================

# La propiedad es
@given(st.integers(),
       st.integers(min_value=0, max_value=100))
def test_potencia(m: int, n: int) -> None:
    r = potencia1(m, n)
    assert potencia2(m, n) == r
    assert potencia3(m, n) == r
    assert potencia4(m, n) == r
    assert potencia5(m, n) == r
    assert potencia6(m, n) == r

# La comprobación es
#    src> poetry run pytest -q potencia_entera.py
#    1 passed in 0.17s

# Comparación de eficiencia
# =========================

def tiempo(e: str) -> None:
```

```
    """Tiempo (en segundos) de evaluar la expresión e."""
    t = Timer(e, "", default_timer, globals()).timeit(1)
    print(f"{t:0.2f} segundos")

# La comparación es
#    >>> tiempo('potencia1(2, 2*10**4)')
#    0.01 segundos
#    >>> tiempo('potencia2(2, 2*10**4)')
#    0.01 segundos
#    >>> tiempo('potencia3(2, 2*10**4)')
#    0.02 segundos
#    >>> tiempo('potencia4(2, 2*10**4)')
#    0.01 segundos
#    >>> tiempo('potencia5(2, 2*10**4)')
#    0.01 segundos
#    >>> tiempo('potencia6(2, 2*10**4)')
#    0.00 segundos
#
#    >>> tiempo('potencia4(2, 5*10**5)')
#    2.87 segundos
#    >>> tiempo('potencia5(2, 5*10**5)')
#    3.17 segundos
#    >>> tiempo('potencia6(2, 5*10**5)')
#    0.00 segundos
```

3.2. Algoritmo de Euclides del mcd

En Haskell

```
-- ---------------------------------------------------------------------
-- Dados dos números naturales, a y b, es posible calcular su máximo
-- común divisor mediante el Algoritmo de Euclides. Este algoritmo se
-- puede resumir en la siguiente fórmula:
--    mcd(a,b) = a,                   si b = 0
--             = mcd (b, a módulo b), si b > 0
--
-- Definir la función
--    mcd :: Integer -> Integer -> Integer
-- tal que (mcd a b) es el máximo común divisor de a y b calculado
-- mediante el algoritmo de Euclides. Por ejemplo,
```

```haskell
--     mcd 30 45  ==  15
--
-- Comprobar con QuickCheck que el máximo común divisor de dos números a
-- y b (ambos mayores que 0) es siempre mayor o igual que 1 y además es
-- menor o igual que el menor de los números a  y b.
-- ---------------------------------------------------------------------

module Algoritmo_de_Euclides_del_mcd where

import Test.QuickCheck

mcd :: Integer -> Integer -> Integer
mcd a 0 = a
mcd a b = mcd b (a `mod` b)

-- La propiedad es
prop_mcd :: Positive Integer -> Positive Integer -> Bool
prop_mcd (Positive a) (Positive b) =
  m >= 1 && m <= min a b
  where m = mcd a b

-- La comprobación es
--     λ> quickCheck prop_mcd
--     OK, passed 100 tests.
```

En Python

```python
# ---------------------------------------------------------------------
# Dados dos números naturales, a y b, es posible calcular su máximo
# común divisor mediante el Algoritmo de Euclides. Este algoritmo se
# puede resumir en la siguiente fórmula:
#    mcd(a,b) = a,                    si b = 0
#             = mcd (b, a módulo b), si b > 0
#
# Definir la función
#    mcd : (int, nt) -> int
# tal que mcd(a, b) es el máximo común divisor de a y b calculado
# mediante el algoritmo de Euclides. Por ejemplo,
#    mcd(30, 45)  ==  15
#    mcd(45, 30)  ==  15
```

```
#
# Comprobar con Hypothesis que el máximo común divisor de dos números a
# y b (ambos mayores que 0) es siempre mayor o igual que 1 y además es
# menor o igual que el menor de los números a  y b.
# ---------------------------------------------------------------------

from hypothesis import given
from hypothesis import strategies as st


def mcd(a: int, b: int) -> int:
    if b == 0:
        return a
    return mcd(b, a % b)

# -- La propiedad es
@given(st.integers(min_value=1, max_value=1000),
       st.integers(min_value=1, max_value=1000))
def test_mcd(a: int, b: int) -> None:
    assert 1 <= mcd(a, b) <= min(a, b)

# La comprobación es
#    src> poetry run pytest -q algoritmo_de_Euclides_del_mcd.py
#    1 passed in 0.22s
```

3.3. Dígitos de un número

En Haskell

```
-- ---------------------------------------------------------------------
-- Definir la función
--    digitos :: Integer -> [Int]
-- tal que (digitos n) es la lista de los dígitos del número n. Por
-- ejemplo,
--    digitos 320274  ==  [3,2,0,2,7,4]
-- ---------------------------------------------------------------------

module Digitos_de_un_numero where

import Data.Char (digitToInt)
```

```haskell
import qualified Data.Digits as D (digits)
import qualified Data.FastDigits as FD (digits)
import Test.QuickCheck

-- 1ª solución
-- ===========

digitos1 :: Integer -> [Int]
digitos1 n = map fromInteger (aux n)
  where aux :: Integer -> [Integer]
        aux m
          | m < 10    = [m]
          | otherwise = aux (m `div` 10) ++ [m `rem` 10]

-- 2ª solución
-- ===========

digitos2 :: Integer -> [Int]
digitos2 n = map fromInteger (reverse (aux n))
  where aux :: Integer -> [Integer]
        aux m
          | m < 10    = [m]
          | otherwise = (m `rem` 10) : aux (m `div` 10)

-- 3ª solución
-- ===========

digitos3 :: Integer -> [Int]
digitos3 n = map fromInteger (aux [] n)
  where aux :: [Integer] -> Integer -> [Integer]
        aux ds m
          | m < 10    = m : ds
          | otherwise = aux (m `rem` 10 : ds) (m `div` 10)

-- 4ª solución
-- ===========

digitos4 :: Integer -> [Int]
digitos4 n = [read [x] | x <- show n]
```

```
-- 5ª solución
-- ===========

digitos5 :: Integer -> [Int]
digitos5 n = map (\ x -> read [x]) (show n)

-- 6ª solución
-- ===========

digitos6 :: Integer -> [Int]
digitos6 = map (read . return) . show

-- 7ª solución
-- ===========

digitos7 :: Integer -> [Int]
digitos7 n = map digitToInt (show n)

-- 8ª solución
-- ===========

digitos8 :: Integer -> [Int]
digitos8 = map digitToInt . show

-- 9ª solución
-- ===========

digitos9 :: Integer -> [Int]
digitos9 0 = [0]
digitos9 n = map fromInteger (D.digits 10 n)

-- 10ª solución
-- ===========

digitos10 :: Integer -> [Int]
digitos10 0 = [0]
digitos10 n = reverse (FD.digits 10 n)

-- Comprobación de equivalencia
-- ============================
```

```
-- La propiedad es
prop_digitos :: NonNegative Integer -> Bool
prop_digitos (NonNegative n) =
  all (== digitos1 n)
      [digitos2 n,
       digitos3 n,
       digitos4 n,
       digitos5 n,
       digitos6 n,
       digitos7 n,
       digitos8 n,
       digitos9 n,
       digitos10 n]

-- La comprobación es
--    λ> quickCheck prop_digitos
--    +++ OK, passed 100 tests.

-- Comparación de eficiencia
-- =========================

-- La comparación es
--    λ> n = product [1..5000]
--    λ> length (digitos1 n)
--    16326
--    (3.00 secs, 11,701,450,912 bytes)
--    λ> length (digitos2 n)
--    16326
--    (0.13 secs, 83,393,816 bytes)
--    λ> length (digitos3 n)
--    16326
--    (0.11 secs, 83,132,552 bytes)
--    λ> length (digitos4 n)
--    16326
--    (0.01 secs, 23,054,920 bytes)
--    λ> length (digitos5 n)
--    16326
--    (0.01 secs, 22,663,088 bytes)
--    λ> length (digitos6 n)
```

```
--    16326
--    (0.06 secs, 22,663,224 bytes)
--    λ> length (digitos7 n)
--    16326
--    (0.01 secs, 22,663,064 bytes)
--    λ> length (digitos8 n)
--    16326
--    (0.03 secs, 22,663,192 bytes)
--    λ> length (digitos9 n)
--    16326
--    (0.05 secs, 82,609,944 bytes)
--    λ> length (digitos10 n)
--    16326
--    (0.01 secs, 26,295,416 bytes)
--
--    λ> n = product [1..5*10^4]
--    λ> length (digitos2 n)
--    213237
--    (10.17 secs, 12,143,633,056 bytes)
--    λ> length (digitos3 n)
--    213237
--    (10.54 secs, 12,140,221,216 bytes)
--    λ> length (digitos4 n)
--    213237
--    (1.29 secs, 2,638,199,328 bytes)
--    λ> length (digitos5 n)
--    213237
--    (2.48 secs, 2,633,081,632 bytes)
--    λ> length (digitos6 n)
--    213237
--    (2.59 secs, 2,633,081,600 bytes)
--    λ> length (digitos7 n)
--    213237
--    (2.55 secs, 2,633,081,608 bytes)
--    λ> length (digitos8 n)
--    213237
--    (2.49 secs, 2,633,081,600 bytes)
--    λ> length (digitos9 n)
--    213237
--    (7.07 secs, 12,133,397,456 bytes)
```

```
--    λ> length (digitos10 n)
--    213237
--    (2.47 secs, 2,725,182,064 bytes)
```

En Python

```
# ---------------------------------------------------------------------
# Definir la función
#    digitos : (int) -> list[int]
# tal que digitos(n) es la lista de los dígitos del número n. Por
# ejemplo,
#    digitos(320274)  ==  [3, 2, 0, 2, 7, 4]
# ---------------------------------------------------------------------

from math import factorial
from sys import setrecursionlimit
from timeit import Timer, default_timer

from hypothesis import given
from hypothesis import strategies as st
from sympy.ntheory.digits import digits

setrecursionlimit(10**6)

# 1ª solución
# ===========

def digitos1(n: int) -> list[int]:
    if n < 10:
        return [n]
    return digitos1(n // 10) + [n % 10]

# 2ª solución
# ===========

def digitos2(n: int) -> list[int]:
    return [int(x) for x in str(n)]

# 3ª solución
# ===========
```

```python
def digitos3(n: int) -> list[int]:
    r: list[int] = []
    for x in str(n):
        r.append(int(x))
    return r

# 4ª solución
# ===========

def digitos4(n: int) -> list[int]:
    return list(map(int, list(str(n))))

# 5ª solución
# ===========

def digitos5(n: int) -> list[int]:
    r: list[int] = []
    while n > 0:
        r = [n % 10] + r
        n = n // 10
    return r

# 6ª solución
# ===========

def digitos6(n: int) -> list[int]:
    r: list[int] = []
    while n > 0:
        r.append(n % 10)
        n = n // 10
    return list(reversed(r))

# 7ª solución
# ===========

def digitos7(n: int) -> list[int]:
    return digits(n)[1:]

# Comprobación de equivalencia
```

```
# ============================

# La propiedad es
@given(st.integers(min_value=1))
def test_digitos(n: int) -> None:
    r = digitos1(n)
    assert digitos2(n) == r
    assert digitos3(n) == r
    assert digitos4(n) == r
    assert digitos5(n) == r
    assert digitos6(n) == r
    assert digitos7(n) == r

# La comprobación es
#    src> poetry run pytest -q digitos_de_un_numero.py
#    1 passed in 0.49s

# Comparación de eficiencia
# =========================

def tiempo(ex: str) -> None:
    """Tiempo (en segundos) de evaluar la expresión e."""
    t = Timer(ex, "", default_timer, globals()).timeit(1)
    print(f"{t:0.2f} segundos")

# La comparación es
#    >>> tiempo('digitos1(factorial(6000))')
#    0.58 segundos
#    >>> tiempo('digitos2(factorial(6000))')
#    0.01 segundos
#    >>> tiempo('digitos3(factorial(6000))')
#    0.01 segundos
#    >>> tiempo('digitos4(factorial(6000))')
#    0.01 segundos
#    >>> tiempo('digitos5(factorial(6000))')
#    0.60 segundos
#    >>> tiempo('digitos6(factorial(6000))')
#    0.17 segundos
#    >>> tiempo('digitos7(factorial(6000))')
#    0.10 segundos
```

```
#
#    >>> tiempo('digitos2(factorial(2*10**4))')
#    0.10 segundos
#    >>> tiempo('digitos3(factorial(2*10**4))')
#    0.10 segundos
#    >>> tiempo('digitos4(factorial(2*10**4))')
#    0.09 segundos
#    >>> tiempo('digitos6(factorial(2*10**4))')
#    2.33 segundos
#    >>> tiempo('digitos7(factorial(2*10**4))')
#    1.18 segundos
#
#    >>> tiempo('digitos2(factorial(10**5))')
#    3.53 segundos
#    >>> tiempo('digitos3(factorial(10**5))')
#    3.22 segundos
#    >>> tiempo('digitos4(factorial(10**5))')
#    3.02 segundos
```

3.4. Suma de los digitos de un número

En Haskell

```
-- ---------------------------------------------------------------------
-- Definir la función
--    sumaDigitos :: Integer -> Integer
-- tal que (sumaDigitos n) es la suma de los dígitos de n. Por ejemplo,
--    sumaDigitos 3     ==  3
--    sumaDigitos 2454  == 15
--    sumaDigitos 20045 == 11
-- ---------------------------------------------------------------------

module Suma_de_los_digitos_de_un_numero where

import Data.List (foldl')
import Test.QuickCheck

-- 1ª solución
-- ===========
```

```haskell
sumaDigitos1 :: Integer -> Integer
sumaDigitos1 n = sum (digitos n)

-- (digitos n) es la lista de los dígitos del número n. Por ejemplo,
--    digitos 320274  ==  [3,2,0,2,7,4]
digitos :: Integer -> [Integer]
digitos n = [read [x] | x <- show n]

-- Nota. En lugar de la definición anterior de digitos se puede usar
-- cualquiera del ejercicio ”Dígitos de un número” https://bit.ly/3Tkhc2T

-- 2ª solución
-- ===========

sumaDigitos2 :: Integer -> Integer
sumaDigitos2 n = foldl' (+) 0 (digitos n)

-- 3ª solución
-- ===========

sumaDigitos3 :: Integer -> Integer
sumaDigitos3 n
  | n < 10    = n
  | otherwise = n `rem` 10 + sumaDigitos3 (n `div` 10)

-- 4ª solución
-- ===========

sumaDigitos4 :: Integer -> Integer
sumaDigitos4 = aux 0
  where aux r n
          | n < 10    = r + n
          | otherwise = aux (r + n `rem` 10) (n `div` 10)

-- Comprobación de equivalencia
-- ============================

-- La propiedad es
prop_sumaDigitos :: NonNegative Integer -> Bool
prop_sumaDigitos (NonNegative n) =
```

```
  all (== sumaDigitos1 n)
      [sumaDigitos2 n,
       sumaDigitos3 n,
       sumaDigitos4 n]

-- La comprobación es
--    λ> quickCheck prop_sumaDigitos
--    +++ OK, passed 100 tests.

-- Comparación de eficiencia
-- =========================

-- La comparación es
--    λ> sumaDigitos1 (product [1..2*10^4])
--    325494
--    (0.64 secs, 665,965,832 bytes)
--    λ> sumaDigitos2 (product [1..2*10^4])
--    325494
--    (0.41 secs, 660,579,064 bytes)
--    λ> sumaDigitos3 (product [1..2*10^4])
--    325494
--    (1.58 secs, 1,647,082,224 bytes)
--    λ> sumaDigitos4 (product [1..2*10^4])
--    325494
--    (1.72 secs, 1,662,177,792 bytes)
--
--    λ> sumaDigitos1 (product [1..5*10^4])
--    903555
--    (2.51 secs, 3,411,722,136 bytes)
--    λ> sumaDigitos2 (product [1..5*10^4])
--    903555
--    (2.30 secs, 3,396,802,856 bytes)
```

En Python

```
# ---------------------------------------------------------------------
# Definir la función
#    sumaDigitos : (int) -> int
# tal que sumaDigitos(n) es la suma de los dígitos de n. Por ejemplo,
#    sumaDigitos(3)     ==  3
```

```
#    sumaDigitos(2454)  == 15
#    sumaDigitos(20045) == 11
# ---------------------------------------------------------------------

from functools import reduce
from math import factorial
from operator import add
from sys import setrecursionlimit
from timeit import Timer, default_timer

from hypothesis import given
from hypothesis import strategies as st

setrecursionlimit(10**6)

# 1ª solución
# ===========

# digitos(n) es la lista de los dígitos del número n. Por ejemplo,
#    digitos(320274)  ==  [3, 2, 0, 2, 7, 4]
def digitos(n: int) -> list[int]:
    return list(map(int, list(str(n))))

def sumaDigitos1(n: int) -> int:
    return sum(digitos(n))

# Nota. En lugar de la definición anterior de digitos se puede usar
# cualquiera del ejercicio "Dígitos de un número" https://bit.ly/3Tkhc2T

# 2ª solución
# ===========

def sumaDigitos2(n: int) -> int:
    return reduce(add, digitos(n))

# 3ª solución
# ===========

def sumaDigitos3(n: int) -> int:
    if n < 10:
```

```
        return n
    return n % 10 + sumaDigitos3(n // 10)

# 4ª solución
# ===========

def sumaDigitos4(n: int) -> int:
    def aux(r: int, m: int) -> int:
        if m < 10:
            return r + m
        return aux(r + m % 10, m // 10)
    return aux(0, n)

# 5ª solución
# ===========

def sumaDigitos5(n: int) -> int:
    r = 0
    for x in digitos(n):
        r = r + x
    return r

# Comprobación de equivalencia
# ============================

# La propiedad es
@given(st.integers(min_value=0, max_value=1000))
def test_sumaDigitos(n: int) -> None:
    r = sumaDigitos1(n)
    assert sumaDigitos2(n) == r
    assert sumaDigitos3(n) == r
    assert sumaDigitos4(n) == r
    assert sumaDigitos5(n) == r

# La comprobación es
#    src> poetry run pytest -q suma_de_los_digitos_de_un_numero.py
#    1 passed in 0.35s

# Comparación de eficiencia
# =========================
```

```python
def tiempo(e: str) -> None:
    """Tiempo (en segundos) de evaluar la expresión e."""
    t = Timer(e, "", default_timer, globals()).timeit(1)
    print(f"{t:0.2f} segundos")

# La comparación es
#    >>> tiempo('sumaDigitos1(factorial(6*10**3))')
#    0.01 segundos
#    >>> tiempo('sumaDigitos2(factorial(6*10**3))')
#    0.01 segundos
#    >>> tiempo('sumaDigitos3(factorial(6*10**3))')
#    0.13 segundos
#    >>> tiempo('sumaDigitos4(factorial(6*10**3))')
#    0.13 segundos
#    >>> tiempo('sumaDigitos5(factorial(6*10**3))')
#    0.01 segundos
#
#    >>> tiempo('sumaDigitos1(factorial(10**5))')
#    2.20 segundos
#    >>> tiempo('sumaDigitos2(factorial(10**5))')
#    2.22 segundos
#    >>> tiempo('sumaDigitos5(factorial(10**5))')
#    2.19 segundos
```

3.5. Número a partir de sus dígitos

En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Definir la función
--    listaNumero :: [Integer] -> Integer
-- tal que (listaNumero xs) es el número formado por los dígitos xs. Por
-- ejemplo,
--    listaNumero [5]       == 5
--    listaNumero [1,3,4,7] == 1347
--    listaNumero [0,0,1]   == 1
-- ---------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-unused-imports #-}
```

```haskell
{-# OPTIONS_GHC -fno-warn-type-defaults #-}

module Numero_a_partir_de_sus_digitos where

import Data.List (foldl')
import Data.Digits (unDigits)
import Test.QuickCheck

-- 1ª solución
-- ===========

listaNumero1 :: [Integer] -> Integer
listaNumero1 = aux . reverse
  where
    aux :: [Integer] -> Integer
    aux []     = 0
    aux (x:xs) = x + 10 * aux xs

-- 2ª solución
-- ===========

listaNumero2 :: [Integer] -> Integer
listaNumero2 = aux 0
  where
    aux :: Integer -> [Integer] -> Integer
    aux r []     = r
    aux r (x:xs) = aux (x+10*r) xs

-- 3ª solución
-- ===========

listaNumero3 :: [Integer] -> Integer
listaNumero3 = aux 0
  where
    aux :: Integer -> [Integer] -> Integer
    aux = foldl (\ r x -> x + 10 * r)

-- 4ª solución
-- ===========
```

```haskell
listaNumero4 :: [Integer] -> Integer
listaNumero4 = foldl' (\ r x -> x + 10 * r) 0

-- 5ª solución
-- ===========

listaNumero5 :: [Integer] -> Integer
listaNumero5 xs = sum [y*10^n | (y,n) <- zip (reverse xs) [0..]]

-- 6ª solución
-- ===========

listaNumero6 :: [Integer] -> Integer
listaNumero6 xs = sum (zipWith (\ y n -> y*10^n) (reverse xs) [0..])

-- 7ª solución
-- ===========

listaNumero7 :: [Integer] -> Integer
listaNumero7 = unDigits 10

-- 7ª solución
-- ===========

listaNumero8 :: [Integer] -> Integer
listaNumero8 = read . concatMap show

-- Comprobación de equivalencia
-- ============================

-- La propiedad es
prop_listaNumero :: NonEmptyList Integer -> Bool
prop_listaNumero (NonEmpty xs) =
  all (== listaNumero1 ys)
      [listaNumero2 ys,
       listaNumero3 ys,
       listaNumero4 ys,
       listaNumero5 ys,
       listaNumero6 ys,
       listaNumero7 ys,
```

```
      listaNumero8 ys]
  where ys = map (`mod` 10) xs

-- La comprobación es
--    λ> quickCheck prop_listaNumero
--    +++ OK, passed 100 tests.

-- Comparación de eficiencia
-- =========================

-- La comparación es
--    λ> length (show (listaNumero1 (replicate (10^5) 9)))
--    100000
--    (4.01 secs, 4,309,740,064 bytes)
--    λ> length (show (listaNumero2 (replicate (10^5) 9)))
--    100000
--    (4.04 secs, 4,307,268,856 bytes)
--    λ> length (show (listaNumero3 (replicate (10^5) 9)))
--    100000
--    (4.08 secs, 4,300,868,816 bytes)
--    λ> length (show (listaNumero4 (replicate (10^5) 9)))
--    100000
--    (0.42 secs, 4,288,480,208 bytes)
--    λ> length (show (listaNumero4 (replicate (10^5) 9)))
--    100000
--    (0.41 secs, 4,288,480,208 bytes)
--    λ> length (show (listaNumero5 (replicate (10^5) 9)))
--    100000
--    (43.35 secs, 10,702,827,328 bytes)
--    λ> length (show (listaNumero6 (replicate (10^5) 9)))
--    100000
--    (46.89 secs, 10,693,227,280 bytes)
--    λ> length (show (listaNumero7 (replicate (10^5) 9)))
--    100000
--    (4.33 secs, 4,297,499,344 bytes)
--    λ> length (show (listaNumero8 (replicate (10^5) 9)))
--    100000
--    (0.03 secs, 60,760,360 bytes)
```

En Python

```
# ---------------------------------------------------------------------
# Definir la función
#    listaNumero : (list[int]) -> int
# tal que listaNumero(xs) es el número formado por los dígitos xs. Por
# ejemplo,
#    listaNumero([5])           == 5
#    listaNumero([1, 3, 4, 7])  == 1347
#    listaNumero([0, 0, 1])     == 1
# ---------------------------------------------------------------------

from functools import reduce
from sys import setrecursionlimit
from timeit import Timer, default_timer

from hypothesis import given
from hypothesis import strategies as st

setrecursionlimit(10**6)

# 1ª solución
# ===========

def listaNumero1(xs: list[int]) -> int:
    def aux(ys: list[int]) -> int:
        if ys:
            return ys[0] + 10 * aux(ys[1:])
        return 0
    return aux(list(reversed(xs)))

# 2ª solución
# ===========

def listaNumero2(xs: list[int]) -> int:
    def aux(r: int, ys: list[int]) -> int:
        if ys:
            return aux(ys[0] + 10 * r, ys[1:])
        return r
    return aux(0, xs)
```

```python
# 3ª solución
# ===========

def listaNumero3(xs: list[int]) -> int:
    return reduce((lambda r, x: x + 10 * r), xs)

# 4ª solución
# ===========

def listaNumero4(xs: list[int]) -> int:
    r = 0
    for x in xs:
        r = x + 10 * r
    return r

# 5ª solución
# ===========

def listaNumero5(xs: list[int]) -> int:
    return sum((y * 10**n
                for (y, n) in zip(list(reversed(xs)), range(0, len(xs)))))

# 6ª solución
# ===========

def listaNumero6(xs: list[int]) -> int:
    return int("".join(list(map(str, xs))))

# Comprobación de equivalencia
# ============================

# La propiedad es
@given(st.lists(st.integers(min_value=0, max_value=9), min_size=1))
def test_listaNumero(xs: list[int]) -> None:
    r = listaNumero1(xs)
    assert listaNumero2(xs) == r
    assert listaNumero3(xs) == r
    assert listaNumero4(xs) == r
    assert listaNumero5(xs) == r
    assert listaNumero6(xs) == r
```

```
# La comprobación es
#    src> poetry run pytest -q numero_a_partir_de_sus_digitos.py
#    1 passed in 0.27s

# Comparación de eficiencia
# =========================

def tiempo(e: str) -> None:
    """Tiempo (en segundos) de evaluar la expresión e."""
    t = Timer(e, "", default_timer, globals()).timeit(1)
    print(f"{t:0.2f} segundos")

# La comparación es
#    >>> tiempo('listaNumero1([9]*(10**4))')
#    0.28 segundos
#    >>> tiempo('listaNumero2([9]*(10**4))')
#    0.16 segundos
#    >>> tiempo('listaNumero3([9]*(10**4))')
#    0.01 segundos
#    >>> tiempo('listaNumero4([9]*(10**4))')
#    0.01 segundos
#    >>> tiempo('listaNumero5([9]*(10**4))')
#    0.41 segundos
#    >>> tiempo('listaNumero6([9]*(10**4))')
#    0.00 segundos
#
#    >>> tiempo('listaNumero3([9]*(2*10**5))')
#    3.45 segundos
#    >>> tiempo('listaNumero4([9]*(2*10**5))')
#    3.29 segundos
#    >>> tiempo('listaNumero6([9]*(2*10**5))')
#    0.19 segundos
```

3.6. Exponente de la mayor potencia de x que divide a y

En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Definir la función
--    mayorExponente :: Integer -> Integer -> Integer
-- tal que (mayorExponente a b) es el exponente de la mayor potencia de
-- a que divide b. Por ejemplo,
--    mayorExponente 2 8    ==  3
--    mayorExponente 2 9    ==  0
--    mayorExponente 5 100  ==  2
--    mayorExponente 2 60   ==  2
--
-- Nota: Se supone que a > 1 y b > 0.
-- ---------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-type-defaults #-}

module Exponente_mayor where

import Test.QuickCheck

-- 1ª solución
-- ===========

mayorExponente1 :: Integer -> Integer -> Integer
mayorExponente1 a b
  | rem b a /= 0 = 0
  | otherwise    = 1 + mayorExponente1 a (b `div` a)

-- 2ª solución
-- ===========

mayorExponente2 :: Integer -> Integer -> Integer
mayorExponente2 a b = aux b 0
  where
    aux c r | rem c a /= 0 = r
            | otherwise    = aux (c `div` a) (r + 1)
```

```haskell
-- 3ª solución
-- ===========

mayorExponente3 :: Integer -> Integer -> Integer
mayorExponente3 a b = head [x-1 | x <- [0..], mod b (a^x) /= 0]

-- 4ª solución
-- ===========

mayorExponente4 :: Integer -> Integer -> Integer
mayorExponente4 a b =
  fst (until (\ (_,c) -> rem c a /= 0)
             (\ (r,c) -> (r+1, c `div` a))
             (0,b))

-- Comprobación de equivalencia
-- ============================

-- La propiedad es
prop_mayorExponente :: Integer -> Integer -> Property
prop_mayorExponente a b =
  a > 1 && b > 0 ==>
  all (== mayorExponente1 a b)
      [mayorExponente2 a b,
       mayorExponente3 a b,
       mayorExponente4 a b]

-- La comprobación es
--    λ> quickCheck prop_mayorExponente
--    +++ OK, passed 100 tests; 457 discarded.

-- Comparación de eficiencia
-- =========================

-- La comparación es
--    λ> mayorExponente1 2 (2^(5*10^4))
--    50000
--    (0.12 secs, 179,578,424 bytes)
--    λ> mayorExponente2 2 (2^(5*10^4))
```

```
--     50000
--     (0.13 secs, 181,533,376 bytes)
--     λ> mayorExponente3 2 (2^(5*10^4))
--     50000
--     (3.88 secs, 818,319,096 bytes)
--     λ> mayorExponente4 2 (2^(5*10^4))
--     50000
--     (0.13 secs, 181,133,344 bytes)
--
--     λ> mayorExponente1 2 (2^(3*10^5))
--     300000
--     (2.94 secs, 5,762,199,064 bytes)
--     λ> mayorExponente2 2 (2^(3*10^5))
--     300000
--     (2.91 secs, 5,773,829,624 bytes)
--     λ> mayorExponente4 2 (2^(3*10^5))
--     300000
--     (3.70 secs, 5,771,396,824 bytes)
```

En Python

```
# ---------------------------------------------------------------------
# Definir la función
#    mayorExponente : (int, int) -> int
# tal que mayorExponente(a, b) es el exponente de la mayor potencia de
# a que divide b. Por ejemplo,
#    mayorExponente(2, 8)    ==  3
#    mayorExponente(2, 9)    ==  0
#    mayorExponente(5, 100)  ==  2
#    mayorExponente(2, 60)   ==  2
#
# Nota: Se supone que a > 1 y b > 0.
# ---------------------------------------------------------------------

from itertools import islice
from sys import setrecursionlimit
from timeit import Timer, default_timer
from typing import Iterator

from hypothesis import given
```

```python
from hypothesis import strategies as st

setrecursionlimit(10**6)

# 1ª solución
# ===========

def mayorExponente1(a: int, b: int) -> int:
    if b % a != 0:
        return 0
    return 1 + mayorExponente1(a, b // a)

# 2ª solución
# ===========

def mayorExponente2(a: int, b: int) -> int:
    def aux(c: int, r: int) -> int:
        if c % a != 0:
            return r
        return aux(c // a, r + 1)
    return aux(b, 0)

# 3ª solución
# ===========

# naturales es el generador de los números naturales, Por ejemplo,
#    >>> list(islice(naturales(), 5))
#    [0, 1, 2, 3, 4]
def naturales() -> Iterator[int]:
    i = 0
    while True:
        yield i
        i += 1

def mayorExponente3(a: int, b: int) -> int:
    return list(islice((x - 1 for x in naturales() if b % (a**x) != 0), 1))[0]

# 4ª solución
# ===========
```

```python
def mayorExponente4(a: int, b: int) -> int:
    r = 0
    while b % a == 0:
        b = b // a
        r = r + 1
    return r

# Comprobación de equivalencia
# ============================

def prueba1() -> None:
    for x in range(2, 11):
        for y in range(1, 11):
            print(x, y, mayorExponente4(x, y))


# La propiedad es
@given(st.integers(min_value=2, max_value=10),
       st.integers(min_value=1, max_value=10))
def test_mayorExponente(a: int, b: int) -> None:
    r = mayorExponente1(a, b)
    assert mayorExponente2(a, b) == r
    assert mayorExponente3(a, b) == r
    assert mayorExponente4(a, b) == r

# La comprobación es
#    src> poetry run pytest -q exponente_mayor.py
#    1 passed in 0.16s

# Comparación de eficiencia
# =========================

def tiempo(e: str) -> None:
    """Tiempo (en segundos) de evaluar la expresión e."""
    t = Timer(e, "", default_timer, globals()).timeit(1)
    print(f"{t:0.2f} segundos")

# La comparación es
#    >>> tiempo('mayorExponente1(2, 2**(2*10**4))')
#    0.13 segundos
```

```
#    >>> tiempo('mayorExponente2(2, 2**(2*10**4))')
#    0.13 segundos
#    >>> tiempo('mayorExponente3(2, 2**(2*10**4))')
#    1.81 segundos
#    >>> tiempo('mayorExponente4(2, 2**(2*10**4))')
#    0.12 segundos
#
#    >>> tiempo('mayorExponente4(2, 2**(2*10**5))')
#    12.19 segundos
```

3.7. Producto cartesiano de dos conjuntos

En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Definir la función
--    producto :: [a] -> [b] -> [(a,b)]
-- tal que (producto xs ys) es el producto cartesiano de xs e ys. Por
-- ejemplo,
--    producto [1,3] [2,4] == [(1,2),(1,4),(3,2),(3,4)]
--
-- Comprobar con QuickCheck que el número de elementos de (producto xs
-- ys) es el producto del número de elementos de xs y de ys.
-- ---------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module Producto_cartesiano_de_dos_conjuntos where

import Test.QuickCheck

-- 1ª solución
-- ===========

producto1 :: [a] -> [a] -> [(a,a)]
producto1 xs ys = [(x,y) | x <- xs, y <- ys]

-- 2ª solución
-- ===========
```

```
producto2 :: [a] -> [a] -> [(a,a)]
producto2 []     _  = []
producto2 (x:xs) ys = [(x,y) | y <- ys] ++ producto2 xs ys

-- Comprobación de equivalencia
-- ============================

-- La propiedad es
prop_producto :: [Int] -> [Int] -> Bool
prop_producto xs ys =
  producto1 xs ys `iguales` producto2 xs ys

-- (iguales xs ys) se verifica si xs e ys son iguales. Por ejemplo,
--    iguales [3,2,3] [2,3]    ==  True
--    iguales [3,2,3] [2,3,2]  ==  True
--    iguales [3,2,3] [2,3,4]  ==  False
--    iguales [2,3] [4,5]      ==  False
iguales :: Ord a => [a] -> [a] -> Bool
iguales xs ys =
  subconjunto xs ys && subconjunto ys xs

-- (subconjunto xs ys) se verifica si xs es un subconjunto de ys. por
-- ejemplo,
--    subconjunto [3,2,3] [2,5,3,5]  ==  True
--    subconjunto [3,2,3] [2,5,6,5]  ==  False
subconjunto :: Ord a => [a] -> [a] -> Bool
subconjunto xs ys =
  [x | x <- xs, x `elem` ys] == xs

-- La comprobación es
--    λ> quickCheck prop_producto
--    +++ OK, passed 100 tests.

-- Comparación de eficiencia
-- =========================

-- La comparación es
--    λ> length (producto1 [1..4000] [1..4000])
--    16000000
--    (2.33 secs, 1,537,551,208 bytes)
```

```
--    λ> length (producto2 [1..4000] [1..4000])
--    16000000
--    (2.87 secs, 2,434,095,160 bytes)

-- Comprobación de la propiedad
-- ============================

-- La propiedad es
prop_elementos_producto :: [Int] -> [Int] -> Bool
prop_elementos_producto xs ys =
  length (producto1 xs ys) == length xs * length ys

-- La comprobación es
--    λ> quickCheck prop_elementos_producto
--    +++ OK, passed 100 tests.
```

En Python

```
# ---------------------------------------------------------------------
# Definir la función
#    producto : (list[A], list[B]) -> list[tuple[(A, B)]]
# tal que producto(xs, ys) es el producto cartesiano de xs e ys. Por
# ejemplo,
#    producto([1, 3], [2, 4]) == [(1, 2), (1, 4), (3, 2), (3, 4)]
#
# Comprobar con Hypothesis que el número de elementos de (producto xs
# ys) es el producto del número de elementos de xs y de ys.
# ---------------------------------------------------------------------

from sys import setrecursionlimit
from timeit import Timer, default_timer
from typing import TypeVar

from hypothesis import given
from hypothesis import strategies as st

setrecursionlimit(10**6)

A = TypeVar('A')
B = TypeVar('B')
```

```
# 1ª solución
# ===========

def producto1(xs: list[A], ys: list[B]) -> list[tuple[A, B]]:
    return [(x, y) for x in xs for y in ys]

# 2ª solución
# ===========

def producto2(xs: list[A], ys: list[B]) -> list[tuple[A, B]]:
    if xs:
        return [(xs[0], y) for y in ys] + producto2(xs[1:], ys)
    return []

# Comprobación de equivalencia
# ============================

# La propiedad es
@given(st.lists(st.integers()),
       st.lists(st.integers()))
def test_producto(xs: list[int], ys: list[int]) -> None:
    assert sorted(producto1(xs, ys)) == sorted(producto2(xs, ys))

# La comprobación es
#    src> poetry run pytest -q producto_cartesiano_de_dos_conjuntos.py
#    1 passed in 0.31s

# Comparación de eficiencia
# =========================

def tiempo(e: str) -> None:
    """Tiempo (en segundos) de evaluar la expresión e."""
    t = Timer(e, "", default_timer, globals()).timeit(1)
    print(f"{t:0.2f} segundos")

# La comparación es
#    >>> tiempo('len(producto1(range(0, 1000), range(0, 500)))')
#    0.03 segundos
#    >>> tiempo('len(producto2(range(0, 1000), range(0, 500)))')
```

```
#    2.58 segundos

# Comprobación de la propiedad
# ============================

# La propiedad es
@given(st.lists(st.integers()),
       st.lists(st.integers()))
def test_elementos_producto(xs: list[int], ys: list[int]) -> None:
    assert len(producto1(xs, ys)) == len(xs) * len(ys)

# La comprobación es
#    src> poetry run pytest -q producto_cartesiano_de_dos_conjuntos.py
#    2 passed in 0.48s
```

3.8. Subconjuntos de un conjunto

En Haskell

```
-- ---------------------------------------------------------------------
-- Definir la función
--    subconjuntos :: [a] -> [[a]]
-- tal que (subconjuntos xs) es la lista de las subconjuntos de la lista
-- xs. Por ejemplo,
--    λ> subconjuntos [2,3,4]
--    [[2,3,4],[2,3],[2,4],[2],[3,4],[3],[4],[]]
--    λ> subconjuntos [1,2,3,4]
--    [[1,2,3,4],[1,2,3],[1,2,4],[1,2],[1,3,4],[1,3],[1,4],[1],
--       [2,3,4],  [2,3],  [2,4],  [2],  [3,4],  [3],  [4], []]
--
-- Comprobar con QuickChek que el número de elementos de
-- (subconjuntos xs) es 2 elevado al número de elementos de xs.
--
-- Nota. Al hacer la comprobación limitar el tamaño de las pruebas como
-- se indica a continuación
--    quickCheckWith (stdArgs {maxSize=7}) prop_length_subconjuntos
-- ---------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-unused-imports #-}
```

```haskell
module Subconjuntos_de_un_conjunto where

import Data.List (sort, subsequences)
import Test.QuickCheck

-- 1ª solución
-- ===========

subconjuntos1 :: [a] -> [[a]]
subconjuntos1 []     = [[]]
subconjuntos1 (x:xs) = [x:ys | ys <- sub] ++ sub
  where sub = subconjuntos1 xs

-- 2ª solución
-- ===========

subconjuntos2 :: [a] -> [[a]]
subconjuntos2 []     = [[]]
subconjuntos2 (x:xs) = map (x:) sub ++ sub
  where sub = subconjuntos2 xs

-- 3ª solución
-- ===========

subconjuntos3 :: [a] -> [[a]]
subconjuntos3 = subsequences

-- Comprobación de equivalencia
-- ============================

-- La propiedad es
prop_subconjuntos :: [Int] -> Bool
prop_subconjuntos xs =
  all (== sort (subconjuntos1 xs))
      [sort (subconjuntos2 xs),
       sort (subconjuntos3 xs)]

-- La comprobación es
--    λ> quickCheckWith (stdArgs {maxSize=7}) prop_subconjuntos
--    +++ OK, passed 100 tests.
```

```haskell
-- Comparación de eficiencia
-- =========================

-- La comparación es
--    λ> length (subconjuntos1 [1..23])
--    8388608
--    (2.05 secs, 1,476,991,840 bytes)
--    λ> length (subconjuntos2 [1..23])
--    8388608
--    (0.87 secs, 1,208,555,312 bytes)
--    λ> length (subconjuntos3 [1..23])
--    8388608
--    (0.09 secs, 873,006,608 bytes)

-- Comprobación de la propiedad
-- ============================

-- La propiedad es
prop_length_subconjuntos :: [Int] -> Bool
prop_length_subconjuntos xs =
  length (subconjuntos1 xs) == 2 ^ length xs

-- La comprobación es
--    λ> quickCheckWith (stdArgs {maxSize=7}) prop_length_subconjuntos
--    +++ OK, passed 100 tests.
```

En Python

```python
# ---------------------------------------------------------------------
# Definir la función
#    subconjuntos : (list[A]) -> list[list[A]]
# tal que subconjuntos(xs) es la lista de las subconjuntos de la lista
# xs. Por ejemplo,
#    >>> subconjuntos([2, 3, 4])
#    [[2,3,4], [2,3], [2,4], [2], [3,4], [3], [4], []]
#    >>> subconjuntos([1, 2, 3, 4])
#    [[1,2,3,4], [1,2,3], [1,2,4], [1,2], [1,3,4], [1,3], [1,4], [1],
#       [2,3,4],   [2,3],   [2,4],   [2],   [3,4],   [3],   [4], []]
#
```

```
# Comprobar con Hypothesis que el número de elementos de
# (subconjuntos xs) es 2 elevado al número de elementos de xs.
# ---------------------------------------------------------------------

from itertools import combinations
from sys import setrecursionlimit
from timeit import Timer, default_timer
from typing import TypeVar

from hypothesis import given
from hypothesis import strategies as st
from sympy import FiniteSet

setrecursionlimit(10**6)

A = TypeVar('A')

# 1ª solución
# ===========

def subconjuntos1(xs: list[A]) -> list[list[A]]:
    if xs:
        sub = subconjuntos1(xs[1:])
        return [[xs[0]] + ys for ys in sub] + sub
    return [[]]

# 2ª solución
# ===========

def subconjuntos2(xs: list[A]) -> list[list[A]]:
    if xs:
        sub = subconjuntos1(xs[1:])
        return list(map((lambda ys: [xs[0]] + ys), sub)) + sub
    return [[]]

# 3ª solución
# ===========

def subconjuntos3(xs: list[A]) -> list[list[A]]:
    c = FiniteSet(*xs)
```

```
    return list(map(list, c.powerset()))

# 4ª solución
# ===========

def subconjuntos4(xs: list[A]) -> list[list[A]]:
    return [list(ys)
            for r in range(len(xs)+1)
            for ys in combinations(xs, r)]

# Comprobación de equivalencia
# ============================

# La propiedad es
@given(st.lists(st.integers(), max_size=5))
def test_subconjuntos(xs: list[int]) -> None:
    ys = list(set(xs))
    r = sorted([sorted(zs) for zs in subconjuntos1(ys)])
    assert sorted([sorted(zs) for zs in subconjuntos2(ys)]) == r
    assert sorted([sorted(zs) for zs in subconjuntos3(ys)]) == r
    assert sorted([sorted(zs) for zs in subconjuntos4(ys)]) == r

# La comprobación es
#    src> poetry run pytest -q subconjuntos_de_un_conjunto.py
#    1 passed in 0.89s

# Comparación de eficiencia
# =========================

def tiempo(e: str) -> None:
    """Tiempo (en segundos) de evaluar la expresión e."""
    t = Timer(e, "", default_timer, globals()).timeit(1)
    print(f"{t:0.2f} segundos")

# La comparación es
#    >>> tiempo('subconjuntos1(range(14))')
#    0.00 segundos
#    >>> tiempo('subconjuntos2(range(14))')
#    0.00 segundos
#    >>> tiempo('subconjuntos3(range(14))')
```

```python
#    6.01 segundos
#    >>> tiempo('subconjuntos4(range(14))')
#    0.00 segundos
#
#    >>> tiempo('subconjuntos1(range(23))')
#    1.95 segundos
#    >>> tiempo('subconjuntos2(range(23))')
#    2.27 segundos
#    >>> tiempo('subconjuntos4(range(23))')
#    1.62 segundos

# Comprobación de la propiedad
# ============================

# La propiedad es
@given(st.lists(st.integers(), max_size=7))
def test_length_subconjuntos(xs: list[int]) -> None:
    assert len(subconjuntos1(xs)) == 2 ** len(xs)

# La comprobación es
#    src> poetry run pytest -q subconjuntos_de_un_conjunto.py
#    2 passed in 0.95s
```

3.9. El algoritmo de Luhn

En Haskell

```haskell
module El_algoritmo_de_Luhn where

-- ---------------------------------------------------------------------
-- El objetivo de este ejercicio es estudiar un algoritmo para validar
-- algunos identificadores numéricos como los números de algunas tarjetas
-- de crédito; por ejemplo, las de tipo Visa o Master Card.
--
-- El algoritmo que vamos a estudiar es el [algoritmo de
-- Luhn](https://bit.ly/3DX1llv) consistente en aplicar los siguientes
-- pasos a los dígitos del número de la tarjeta.
--    1. Se invierten los dígitos del número; por ejemplo, [9,4,5,5] se
--       transforma en [5,5,4,9].
--    2. Se duplican los dígitos que se encuentra en posiciones impares
```

```
--       (empezando a contar en 0); por ejemplo, [5,5,4,9] se transforma
--       en [5,10,4,18].
--    3. Se suman los dígitos de cada número; por ejemplo, [5,10,4,18]
--       se transforma en 5 + (1 + 0) + 4 + (1 + 8) = 19.
--    4. Si el último dígito de la suma es 0, el número es válido; y no
--       lo es, en caso contrario.
--
-- A los números válidos, se les llama números de Luhn.
--
-- Definir las siguientes funciones:
--    digitosInv    :: Integer -> [Integer]
--    doblePosImpar :: [Integer] -> [Integer]
--    sumaDigitos   :: [Integer] -> Integer
--    ultimoDigito  :: Integer -> Integer
--    luhn          :: Integer -> Bool
-- tales que
-- + (digitosInv n) es la lista de los dígitos del número n. en orden
--   inverso. Por ejemplo,
--      digitosInv 320274  ==  [4,7,2,0,2,3]
-- + (doblePosImpar ns) es la lista obtenida doblando los elementos en
--   las posiciones impares (empezando a contar en cero y dejando igual
--   a los que están en posiciones pares. Por ejemplo,
--      doblePosImpar [4,9,5,5]    ==  [4,18,5,10]
--      doblePosImpar [4,9,5,5,7]  ==  [4,18,5,10,7]
-- + (sumaDigitos ns) es la suma de los dígitos de ns. Por ejemplo,
--      sumaDigitos [10,5,18,4] = 1 + 0 + 5 + 1 + 8 + 4 =
--                              = 19
-- + (ultimoDigito n) es el último dígito de n. Por ejemplo,
--      ultimoDigito 123 == 3
--      ultimoDigito   0 == 0
-- + (luhn n) se verifica si n es un número de Luhn. Por ejemplo,
--      luhn 5594589764218858  ==  True
--      luhn 1234567898765432  ==  False
-- ---------------------------------------------------------------------

-- Definición de digitosInv
-- ========================

digitosInv :: Integer -> [Integer]
digitosInv n = [read [x] | x <- reverse (show n)]
```

```
-- Nota: En el ejercicio "Dígitos de un número" https://bit.ly/3Tkhc2T
-- se presentan otras definiciones.

-- Definiciones de doblePosImpar
-- =============================

-- 1ª definición
doblePosImpar :: [Integer] -> [Integer]
doblePosImpar []       = []
doblePosImpar [x]      = [x]
doblePosImpar (x:y:zs) = x : 2*y : doblePosImpar zs

-- 2ª definición
doblePosImpar2 :: [Integer] -> [Integer]
doblePosImpar2 (x:y:zs) = x : 2*y : doblePosImpar2 zs
doblePosImpar2 xs       = xs

-- 3ª definición
doblePosImpar3 :: [Integer] -> [Integer]
doblePosImpar3 xs = [f n x | (n,x) <- zip [0..] xs]
  where f n x | odd n     = 2*x
              | otherwise = x

-- Definiciones de sumaDigitos
-- ===========================

sumaDigitos :: [Integer] -> Integer
sumaDigitos ns = sum [sum (digitosInv n) | n <- ns]

-- Nota: En el ejercicio "Suma de los dígitos de un número"
-- https://bit.ly/3U4u7WR se presentan otras definiciones.

-- Definición de ultimoDigito
-- ==========================

ultimoDigito :: Integer -> Integer
ultimoDigito n = n `rem` 10

-- Definiciones de luhn
```

```
-- ====================

-- 1ª definición
luhn1 :: Integer -> Bool
luhn1 n =
  ultimoDigito (sumaDigitos (doblePosImpar (digitosInv n))) == 0

-- 2ª definición
luhn2 :: Integer -> Bool
luhn2 =
  (==0) . ultimoDigito . sumaDigitos . doblePosImpar . digitosInv
```

En Python

```
# ---------------------------------------------------------------------
# El objetivo de este ejercicio es estudiar un algoritmo para validar
# algunos identificadores numéricos como los números de algunas tarjetas
# de crédito; por ejemplo, las de tipo Visa o Master Card.
#
# El algoritmo que vamos a estudiar es el [algoritmo de
# Luhn](https://bit.ly/3DX1llv) consistente en aplicar los siguientes
# pasos a los dígitos del número de la tarjeta.
#    1. Se invierten los dígitos del número; por ejemplo, [9,4,5,5] se
#       transforma en [5,5,4,9].
#    2. Se duplican los dígitos que se encuentra en posiciones impares
#       (empezando a contar en 0); por ejemplo, [5,5,4,9] se transforma
#       en [5,10,4,18].
#    3. Se suman los dígitos de cada número; por ejemplo, [5,10,4,18]
#       se transforma en 5 + (1 + 0) + 4 + (1 + 8) = 19.
#    4. Si el último dígito de la suma es 0, el número es válido; y no
#       lo es, en caso contrario.
#
# A los números válidos, se les llama números de Luhn.
#
# Definir las siguientes funciones:
#    digitosInv    : (int) -> list[int]
#    doblePosImpar : (list[int]) -> list[int]
#    sumaDigitos   : (list[int]) -> int
#    ultimoDigito  : (int) -> int
#    luhn          : (int) -> bool
```

```
# tales que
# + digitosInv(n) es la lista de los dígitos del número n. en orden
#   inverso. Por ejemplo,
#      digitosInv(320274)  ==  [4,7,2,0,2,3]
# + doblePosImpar(ns) es la lista obtenida doblando los elementos en
#   las posiciones impares (empezando a contar en cero y dejando igual
#   a los que están en posiciones pares. Por ejemplo,
#      doblePosImpar([4,9,5,5])    ==  [4,18,5,10]
#      doblePosImpar([4,9,5,5,7])  ==  [4,18,5,10,7]
# + sumaDigitos(ns) es la suma de los dígitos de ns. Por ejemplo,
#      sumaDigitos([10,5,18,4]) = 1 + 0 + 5 + 1 + 8 + 4 =
#                               = 19
# + ultimoDigito(n) es el último dígito de n. Por ejemplo,
#      ultimoDigito(123) == 3
#      ultimoDigito(0)   == 0
# + luhn(n) se verifica si n es un número de Luhn. Por ejemplo,
#      luhn(5594589764218858)  ==  True
#      luhn(1234567898765432)  ==  False
# ---------------------------------------------------------------------

# Definición de digitosInv
# ========================

def digitosInv(n: int) -> list[int]:
    return [int(x) for x in reversed(str(n))]

# Nota: En el ejercicio ”Dígitos de un número” https://bit.ly/3Tkhc2T
# se presentan otras definiciones.

# Definiciones de doblePosImpar
# =============================

# 1ª definición
def doblePosImpar(xs: list[int]) -> list[int]:
    if len(xs) <= 1:
        return xs
    return [xs[0]] + [2*xs[1]] + doblePosImpar(xs[2:])

# 2ª definición
def doblePosImpar2(xs: list[int]) -> list[int]:
```

```
    def f(n: int, x: int) -> int:
        if n % 2 == 1:
            return 2 * x
        return x
    return [f(n, x) for (n, x) in enumerate(xs)]

# Definiciones de sumaDigitos
# ===========================

def sumaDigitos(ns: list[int]) -> int:
    return sum((sum(digitosInv(n)) for n in ns))

# Nota: En el ejercicio "Suma de los dígitos de un número"
# https://bit.ly/3U4u7WR se presentan otras definiciones.

# Definición de ultimoDigito
# ==========================

def ultimoDigito(n: int) -> int:
    return n % 10

# Definiciones de luhn
# ====================

def luhn(n: int) -> bool:
    return ultimoDigito(sumaDigitos(doblePosImpar(digitosInv(n)))) == 0
```

3.10. Números de Lychrel

En Haskell

```
-- ---------------------------------------------------------------------
-- Un [número de Lychrel](http://bit.ly/2X4DzMf) es un número natural
-- para el que nunca se obtiene un capicúa mediante el proceso de
-- invertir las cifras y sumar los dos números. Por ejemplo, los
-- siguientes números no son números de Lychrel:
-- + 56, ya que en un paso se obtiene un capicúa: 56+65=121.
-- + 57, ya que en dos pasos se obtiene un capicúa: 57+75=132,
--   132+231=363
-- + 59, ya que en dos pasos se obtiene un capicúa: 59+95=154,
```

```haskell
--   154+451=605, 605+506=1111
-- + 89, ya que en 24 pasos se obtiene un capicúa.
-- En esta serie de ejercicios vamos a buscar el primer número de
-- Lychrel.
-- ---------------------------------------------------------------------

module Numeros_de_Lychrel where
import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1. Definir la función
--    esCapicua :: Integer -> Bool
-- tal que (esCapicua x) se verifica si x es capicúa. Por ejemplo,
--    esCapicua 252  ==  True
--    esCapicua 253  ==  False
-- ---------------------------------------------------------------------

esCapicua :: Integer -> Bool
esCapicua x = x' == reverse x'
  where x' = show x

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    inverso :: Integer -> Integer
-- tal que (inverso x) es el número obtenido escribiendo las cifras de x
-- en orden inverso. Por ejemplo,
--    inverso 253  ==  352
-- ---------------------------------------------------------------------

inverso :: Integer -> Integer
inverso = read . reverse . show

-- ---------------------------------------------------------------------
-- Ejercicio 3. Definir la función
--    siguiente :: Integer -> Integer
-- tal que (siguiente x) es el número obtenido sumándole a x su
-- inverso. Por ejemplo,
--    siguiente 253  ==  605
-- ---------------------------------------------------------------------
```

```haskell
siguiente :: Integer -> Integer
siguiente x = x + inverso x

-- ---------------------------------------------------------------------
-- Ejercicio 4. Definir la función
--    busquedaDeCapicua :: Integer -> [Integer]
-- tal que (busquedaDeCapicua x) es la lista de los números tal que el
-- primero es x, el segundo es (siguiente de x) y así sucesivamente
-- hasta que se alcanza un capicúa. Por ejemplo,
--    busquedaDeCapicua 253  ==  [253,605,1111]
-- ---------------------------------------------------------------------

busquedaDeCapicua :: Integer -> [Integer]
busquedaDeCapicua x | esCapicua x = [x]
                    | otherwise   = x : busquedaDeCapicua (siguiente x)

-- ---------------------------------------------------------------------
-- Ejercicio 5. Definir la función
--    capicuaFinal :: Integer -> Integer
-- tal que (capicuaFinal x) es la capicúa con la que termina la búsqueda
-- de capicúa a partir de x. Por ejemplo,
--    capicuaFinal 253  ==  1111
-- ---------------------------------------------------------------------

capicuaFinal :: Integer -> Integer
capicuaFinal x = last (busquedaDeCapicua x)

-- ---------------------------------------------------------------------
-- Ejercicio 6. Definir la función
--    orden :: Integer -> Integer
-- tal que (orden x) es el número de veces que se repite el proceso de
-- calcular el inverso a partir de x hasta alcanzar un número
-- capicúa. Por ejemplo,
--    orden 253  ==  2
-- ---------------------------------------------------------------------

orden :: Integer -> Integer
orden x | esCapicua x = 0
        | otherwise   = 1 + orden (siguiente x)
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 7. Definir la función
--    ordenMayor :: Integer -> Integer -> Bool
-- tal que (ordenMayor x n) se verifica si el orden de x es mayor o
-- igual que n. Dar la definición sin necesidad de evaluar el orden de
-- x. Por ejemplo,
--    λ> ordenMayor 1186060307891929990 2
--    True
--    λ> orden 1186060307891929990
--    261
-- ---------------------------------------------------------------------

ordenMayor :: Integer -> Integer -> Bool
ordenMayor x n | esCapicua x = n == 0
               | n <= 0      = True
               | otherwise   = ordenMayor (siguiente x) (n-1)

-- ---------------------------------------------------------------------
-- Ejercicio 8. Definir la función
--    ordenEntre :: Integer -> Integer -> [Integer]
-- tal que (ordenEntre m n) es la lista de los elementos cuyo orden es
-- mayor o igual que m y menor que n. Por ejemplo,
--    take 5 (ordenEntre 10 11)  ==  [829,928,9059,9149,9239]
-- ---------------------------------------------------------------------

ordenEntre :: Integer -> Integer -> [Integer]
ordenEntre m n = [x | x <- [1..], ordenMayor x m, not (ordenMayor x n)]

-- ---------------------------------------------------------------------
-- Ejercicio 9. Definir la función
--    menorDeOrdenMayor :: Integer -> Integer
-- tal que (menorDeOrdenMayor n) es el menor elemento cuyo orden es
-- mayor que n. Por ejemplo,
--    menorDeOrdenMayor 2   ==  19
--    menorDeOrdenMayor 20  ==  89
-- ---------------------------------------------------------------------

menorDeOrdenMayor :: Integer -> Integer
menorDeOrdenMayor n = head [x | x <- [1..], ordenMayor x n]
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 10. Definir la función
--    menoresdDeOrdenMayor :: Integer -> [(Integer,Integer)]
-- tal que (menoresdDeOrdenMayor m) es la lista de los pares (n,x) tales
-- que n es un número entre 1 y m y x es el menor elemento de orden
-- mayor que n. Por ejemplo,
--    menoresdDeOrdenMayor 5  ==  [(1,10),(2,19),(3,59),(4,69),(5,79)]
-- ---------------------------------------------------------------------

menoresdDeOrdenMayor :: Integer -> [(Integer,Integer)]
menoresdDeOrdenMayor m = [(n,menorDeOrdenMayor n) | n <- [1..m]]

-- ---------------------------------------------------------------------
-- Ejercicio 11. A la vista de los resultados de (menoresdDeOrdenMayor 5)
-- conjeturar sobre la última cifra de menorDeOrdenMayor.
-- ---------------------------------------------------------------------

-- Solución: La conjetura es que para n mayor que 1, la última cifra de
-- (menorDeOrdenMayor n) es 9.

-- ---------------------------------------------------------------------
-- Ejercicio 12. Decidir con QuickCheck la conjetura.
-- ---------------------------------------------------------------------

-- La conjetura es
prop_menorDeOrdenMayor :: Integer -> Property
prop_menorDeOrdenMayor n =
  n > 1 ==> menorDeOrdenMayor n `mod` 10 == 9

-- La comprobación es
--    λ> quickCheck prop_menorDeOrdenMayor
--    *** Failed! Falsifiable (after 22 tests and 2 shrinks):
--    25

-- Se puede comprobar que 25 es un contraejemplo,
--    λ> menorDeOrdenMayor 25
--    196

-- ---------------------------------------------------------------------
-- Ejercicio 13. Calcular (menoresdDeOrdenMayor 50)
```

```
-- ---------------------------------------------------------------------

-- Solución: El cálculo es
--    λ> menoresdDeOrdenMayor 50
--    [(1,10),(2,19),(3,59),(4,69),(5,79),(6,79),(7,89),(8,89),(9,89),
--     (10,89),(11,89),(12,89),(13,89),(14,89),(15,89),(16,89),(17,89),
--     (18,89),(19,89),(20,89),(21,89),(22,89),(23,89),(24,89),(25,196),
--     (26,196),(27,196),(28,196),(29,196),(30,196),(31,196),(32,196),
--     (33,196),(34,196),(35,196),(36,196),(37,196),(38,196),(39,196),
--     (40,196),(41,196),(42,196),(43,196),(44,196),(45,196),(46,196),
--     (47,196),(48,196),(49,196),(50,196)]

-- ---------------------------------------------------------------------
-- Ejercicio 14. A la vista de (menoresdDeOrdenMayor 50), conjeturar el
-- orden de 196.
-- ---------------------------------------------------------------------

-- Solución: El orden de 196 es infinito y, por tanto, 196 es un número
-- del Lychrel.

-- ---------------------------------------------------------------------
-- Ejercicio 15. Comprobar con QuickCheck la conjetura sobre el orden de
-- 196.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_ordenDe196 :: Integer -> Bool
prop_ordenDe196 n =
  ordenMayor 196 n

-- La comprobación es
--    λ> quickCheck prop_ordenDe196
--    +++ OK, passed 100 tests.
```

En Python

```
# ---------------------------------------------------------------------
# Un [número de Lychrel](http://bit.ly/2X4DzMf) es un número natural
# para el que nunca se obtiene un capicúa mediante el proceso de
# invertir las cifras y sumar los dos números. Por ejemplo, los
```

```python
# siguientes números no son números de Lychrel:
# + 56, ya que en un paso se obtiene un capicúa: 56+65=121.
# + 57, ya que en dos pasos se obtiene un capicúa: 57+75=132,
#   132+231=363
# + 59, ya que en dos pasos se obtiene un capicúa: 59+95=154,
#   154+451=605, 605+506=1111
# + 89, ya que en 24 pasos se obtiene un capicúa.
# En esta serie de ejercicios vamos a buscar el primer número de
# Lychrel.
# ---------------------------------------------------------------------

from itertools import islice
from sys import setrecursionlimit
from typing import Generator, Iterator

from hypothesis import given, settings
from hypothesis import strategies as st

setrecursionlimit(10**6)

# ---------------------------------------------------------------------
# Ejercicio 1. Definir la función
#    esCapicua : (int) -> bool
# tal que esCapicua(x) se verifica si x es capicúa. Por ejemplo,
#    esCapicua(252)  ==  True
#    esCapicua(253)  ==  False
# ---------------------------------------------------------------------

def esCapicua(x: int) -> bool:
    return x == int(str(x)[::-1])

# ---------------------------------------------------------------------
# Ejercicio 2. Definir la función
#    inverso : (int) -> int
# tal que inverso(x) es el número obtenido escribiendo las cifras de x
# en orden inverso. Por ejemplo,
#    inverso(253)  ==  352
# ---------------------------------------------------------------------

def inverso(x: int) -> int:
```

```
    return int(str(x)[::-1])

# ---------------------------------------------------------------------
# Ejercicio 3. Definir la función
#    siguiente : (int) -> int
# tal que siguiente(x) es el número obtenido sumándole a x su
# inverso. Por ejemplo,
#    siguiente(253)  ==  605
# ---------------------------------------------------------------------

def siguiente(x: int) -> int:
    return x + inverso(x)

# ---------------------------------------------------------------------
# Ejercicio 4. Definir la función
#    busquedaDeCapicua : (int) -> list[int]
# tal que busquedaDeCapicua(x) es la lista de los números tal que el
# primero es x, el segundo es (siguiente de x) y así sucesivamente
# hasta que se alcanza un capicúa. Por ejemplo,
#    busquedaDeCapicua(253)  ==  [253,605,1111]
# ---------------------------------------------------------------------

def busquedaDeCapicua(x: int) -> list[int]:
    if esCapicua(x):
        return [x]
    return [x] + busquedaDeCapicua(siguiente(x))

# ---------------------------------------------------------------------
# Ejercicio 5. Definir la función
#    capicuaFinal : (int) -> int
# tal que (capicuaFinal x) es la capicúa con la que termina la búsqueda
# de capicúa a partir de x. Por ejemplo,
#    capicuaFinal(253)  ==  1111
# ---------------------------------------------------------------------

def capicuaFinal(x: int) -> int:
    return busquedaDeCapicua(x)[-1]

# ---------------------------------------------------------------------
# Ejercicio 6. Definir la función
```

```
#    orden : (int) -> int
# tal que orden(x) es el número de veces que se repite el proceso de
# calcular el inverso a partir de x hasta alcanzar un número capicúa.
# Por ejemplo,
#    orden(253)  ==  2
# ---------------------------------------------------------------------

def orden(x: int) -> int:
    if esCapicua(x):
        return 0
    return 1 + orden(siguiente(x))

# ---------------------------------------------------------------------
# Ejercicio 7. Definir la función
#    ordenMayor : (int, int) -> bool:
# tal que ordenMayor(x, n) se verifica si el orden de x es mayor o
# igual que n. Dar la definición sin necesidad de evaluar el orden de
# x. Por ejemplo,
#    >>> ordenMayor(1186060307891929990, 2)
#    True
#    >>> orden(1186060307891929990)
#    261
# ---------------------------------------------------------------------

def ordenMayor(x: int, n: int) -> bool:
    if esCapicua(x):
        return n == 0
    if n <= 0:
        return True
    return ordenMayor(siguiente(x), n - 1)

# ---------------------------------------------------------------------
# Ejercicio 8. Definir la función
#    ordenEntre : (int, int) -> Generator[int, None, None]
# tal que ordenEntre(m, n) es la lista de los elementos cuyo orden es
# mayor o igual que m y menor que n. Por ejemplo,
#    >>> list(islice(ordenEntre(10, 11), 5))
#    [829, 928, 9059, 9149, 9239]
# ---------------------------------------------------------------------
```

```python
# naturales es el generador de los números naturales positivos, Por
# ejemplo,
#    >>> list(islice(naturales(), 5))
#    [1, 2, 3, 4, 5]
def naturales() -> Iterator[int]:
    i = 1
    while True:
        yield i
        i += 1

def ordenEntre(m: int, n: int) -> Generator[int, None, None]:
    return (x for x in naturales()
            if ordenMayor(x, m) and not ordenMayor(x, n))

# ---------------------------------------------------------------------
# Ejercicio 9. Definir la función
#    menorDeOrdenMayor : (int) -> int
# tal que menorDeOrdenMayor(n) es el menor elemento cuyo orden es
# mayor que n. Por ejemplo,
#    menorDeOrdenMayor(2)   ==  19
#    menorDeOrdenMayor(20)  ==  89
# ---------------------------------------------------------------------

def menorDeOrdenMayor(n: int) -> int:
    return list(islice((x for x in naturales() if ordenMayor(x, n)), 1))[0]

# ---------------------------------------------------------------------
# Ejercicio 10. Definir la función
#    menoresdDeOrdenMayor : (int) -> list[tuple[int, int]]
# tal que (menoresdDeOrdenMayor m) es la lista de los pares (n,x) tales
# que n es un número entre 1 y m y x es el menor elemento de orden
# mayor que n. Por ejemplo,
#    menoresdDeOrdenMayor(5)  ==  [(1,10),(2,19),(3,59),(4,69),(5,79)]
# ---------------------------------------------------------------------

def menoresdDeOrdenMayor(m: int) -> list[tuple[int, int]]:
    return [(n, menorDeOrdenMayor(n)) for n in range(1, m + 1)]

# ---------------------------------------------------------------------
# Ejercicio 11. A la vista de los resultados de (menoresdDeOrdenMayor 5)
```

```
# conjeturar sobre la última cifra de menorDeOrdenMayor.
# ---------------------------------------------------------------------

# Solución: La conjetura es que para n mayor que 1, la última cifra de
# (menorDeOrdenMayor n) es 9.

# ---------------------------------------------------------------------
# Ejercicio 12. Decidir con Hypothesis la conjetura.
# ---------------------------------------------------------------------

# La conjetura es
# @given(st.integers(min_value=2, max_value=200))
# def test_menorDeOrdenMayor(n: int) -> None:
#     assert menorDeOrdenMayor(n) % 10 == 9

# La comprobación es
#    src> poetry run pytest -q numeros_de_Lychrel.py
#    E       assert (196 % 10) == 9
#    E        +  where 196 = menorDeOrdenMayor(25)
#    E       Falsifying example: test_menorDeOrdenMayor(
#    E           n=25,
#    E       )

# Se puede comprobar que 25 es un contraejemplo,
#    >>> menorDeOrdenMayor(25)
#    196

# ---------------------------------------------------------------------
# Ejercicio 13. Calcular menoresdDeOrdenMayor(50)
# ---------------------------------------------------------------------

# Solución: El cálculo es
#    λ> menoresdDeOrdenMayor 50
#    [(1,10),(2,19),(3,59),(4,69),(5,79),(6,79),(7,89),(8,89),(9,89),
#     (10,89),(11,89),(12,89),(13,89),(14,89),(15,89),(16,89),(17,89),
#     (18,89),(19,89),(20,89),(21,89),(22,89),(23,89),(24,89),(25,196),
#     (26,196),(27,196),(28,196),(29,196),(30,196),(31,196),(32,196),
#     (33,196),(34,196),(35,196),(36,196),(37,196),(38,196),(39,196),
#     (40,196),(41,196),(42,196),(43,196),(44,196),(45,196),(46,196),
#     (47,196),(48,196),(49,196),(50,196)]
```

```
# ---------------------------------------------------------------------
# Ejercicio 14. A la vista de menoresdDeOrdenMayor(50), conjeturar el
# orden de 196.
# ---------------------------------------------------------------------

# Solución: El orden de 196 es infinito y, por tanto, 196 es un número
# del Lychrel.

# ---------------------------------------------------------------------
# Ejercicio 15. Comprobar con Hypothesis la conjetura sobre el orden de
# 196.
# ---------------------------------------------------------------------

# La propiedad es
@settings(deadline=None)
@given(st.integers(min_value=2, max_value=5000))
def test_ordenDe196(n: int) -> None:
    assert ordenMayor(196, n)

# La comprobación es
#    src> poetry run pytest -q numeros_de_Lychrel.py
#    1 passed in 7.74s
```

3.11. Suma de los dígitos de una cadena

En Haskell

```
-- ---------------------------------------------------------------------
-- Definir la función
--    sumaDigitos :: String -> Int
-- tal que (sumaDigitos xs) es la suma de los dígitos de la cadena
-- xs. Por ejemplo,
--    sumaDigitos "SE 2431 X"  ==  10
-- ---------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module Suma_de_digitos_de_cadena where
```

```haskell
import Data.Char (digitToInt, isDigit)
import Test.QuickCheck

-- 1ª solución
-- ===========

sumaDigitos1 :: String -> Int
sumaDigitos1 xs = sum [digitToInt x | x <- xs, isDigit x]

-- 2ª solución
-- ===========

sumaDigitos2 :: String -> Int
sumaDigitos2 [] = 0
sumaDigitos2 (x:xs)
  | isDigit x  = digitToInt x + sumaDigitos2 xs
  | otherwise  = sumaDigitos2 xs

-- 3ª solución
-- ===========

sumaDigitos3 :: String -> Int
sumaDigitos3 xs = sum (map digitToInt (filter isDigit xs))

-- 4ª solución
-- ===========

sumaDigitos4 :: String -> Int
sumaDigitos4 = sum . map digitToInt . filter isDigit

-- Comprobación de equivalencia
-- ============================

-- La propiedad es
prop_sumaDigitos :: String -> Bool
prop_sumaDigitos xs =
  all (== sumaDigitos1 xs)
      [sumaDigitos2 xs,
       sumaDigitos3 xs,
       sumaDigitos4 xs]
```

```
-- La comprobación es
--    λ> quickCheck prop_sumaDigitos
--    +++ OK, passed 100 tests.

-- Comparación de eficiencia
-- =========================

-- La comparación es
--    λ> sumaDigitos1 (take (4*10^6) (cycle "ab12"))
--    3000000
--    (1.92 secs, 819,045,328 bytes)
--    λ> sumaDigitos2 (take (4*10^6) (cycle "ab12"))
--    3000000
--    (1.79 secs, 856,419,112 bytes)
--    λ> sumaDigitos3 (take (4*10^6) (cycle "ab12"))
--    3000000
--    (0.62 secs, 723,045,296 bytes)
--    λ> sumaDigitos4 (take (4*10^6) (cycle "ab12"))
--    3000000
--    (0.63 secs, 723,045,552 bytes)
```

En Python

```
# ---------------------------------------------------------------------
# Definir la función
#    sumaDigitos : (str) -> int
# tal que sumaDigitos(xs) es la suma de los dígitos de la cadena
# xs. Por ejemplo,
#    sumaDigitos("SE 2431 X")  ==  10
# ---------------------------------------------------------------------

from sys import setrecursionlimit
from timeit import Timer, default_timer

from hypothesis import given
from hypothesis import strategies as st

setrecursionlimit(10**6)
```

```
# 1ª solución
# ===========

def sumaDigitos1(xs: str) -> int:
    return sum((int(x) for x in xs if x.isdigit()))

# 2ª solución
# ===========

def sumaDigitos2(xs: str) -> int:
    if xs:
        if xs[0].isdigit():
            return int(xs[0]) + sumaDigitos2(xs[1:])
        return sumaDigitos2(xs[1:])
    return 0

# 3ª solución
# ===========

def sumaDigitos3(xs: str) -> int:
    r = 0
    for x in xs:
        if x.isdigit():
            r = r + int(x)
    return r

# Comprobación de equivalencia
# ============================

# La propiedad es
@given(st.text())
def test_sumaDigitos(xs: str) -> None:
    r = sumaDigitos1(xs)
    assert sumaDigitos2(xs) == r
    assert sumaDigitos3(xs) == r

# La comprobación es
#    src> poetry run pytest -q suma_de_digitos_de_cadena.py
#    1 passed in 0.41s
```

```python
# Comparación de eficiencia
# =========================

def tiempo(e: str) -> None:
    """Tiempo (en segundos) de evaluar la expresión e."""
    t = Timer(e, "", default_timer, globals()).timeit(1)
    print(f"{t:0.2f} segundos")

# La comparación es
#    >>> tiempo('sumaDigitos1("ab12"*5000)')
#    0.00 segundos
#    >>> tiempo('sumaDigitos2("ab12"*5000)')
#    0.02 segundos
#    >>> tiempo('sumaDigitos3("ab12"*5000)')
#    0.00 segundos
#
#    >>> tiempo('sumaDigitos1("ab12"*(5*10**6))')
#    1.60 segundos
#    >>> tiempo('sumaDigitos3("ab12"*(5*10**6))')
#    1.83 segundos
```

3.12. Primera en mayúscula y restantes en minúscula

En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Definir la función
--    mayusculaInicial :: String -> String
-- tal que (mayusculaInicial xs) es la palabra xs con la letra inicial
-- en mayúscula y las restantes en minúsculas. Por ejemplo,
--    mayusculaInicial "sEviLLa"  ==  "Sevilla"
--    mayusculaInicial ""         ==  ""
-- ---------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module Mayuscula_inicial where
```

```haskell
import Data.Char (toUpper, toLower)
import Test.QuickCheck

-- 1ª solución
-- ===========

mayusculaInicial1 :: String -> String
mayusculaInicial1 []     = []
mayusculaInicial1 (x:xs) = toUpper x : [toLower y | y <- xs]

-- 2ª solución
-- ===========

mayusculaInicial2 :: String -> String
mayusculaInicial2 [] = []
mayusculaInicial2 (x:xs) = toUpper x : aux xs
  where aux (y:ys) = toLower y : aux ys
        aux []     = []

-- 3ª solución
-- ===========

mayusculaInicial3 :: String -> String
mayusculaInicial3 [] = []
mayusculaInicial3 (x:xs) = toUpper x : map toLower xs

-- Comprobación de equivalencia
-- ============================

-- La propiedad es
prop_mayusculaInicial :: String -> Bool
prop_mayusculaInicial xs =
  all (== mayusculaInicial1 xs)
      [mayusculaInicial2 xs,
       mayusculaInicial3 xs]

-- La comprobación es
--    λ> quickCheck prop_mayusculaInicial
--    +++ OK, passed 100 tests.
```

```
-- Comparación de eficiencia
-- ========================

-- La comparación es
--    λ> length (mayusculaInicial1 (take (10^7) (cycle "aA")))
--    10000000
--    (2.22 secs, 1,680,592,240 bytes)
--    λ> length (mayusculaInicial2 (take (10^7) (cycle "aA")))
--    10000000
--    (2.57 secs, 2,240,592,192 bytes)
--    λ> length (mayusculaInicial3 (take (10^7) (cycle "aA")))
--    10000000
--    (0.16 secs, 1,440,592,192 bytes)
```

En Python

```
# ---------------------------------------------------------------------
# Definir la función
#    mayusculaInicial : (str) -> str
# tal que mayusculaInicial(xs) es la palabra xs con la letra inicial
# en mayúscula y las restantes en minúsculas. Por ejemplo,
#    mayusculaInicial("sEviLLa")  ==  "Sevilla"
#    mayusculaInicial("")         ==  ""
# ---------------------------------------------------------------------

from sys import setrecursionlimit
from timeit import Timer, default_timer

from hypothesis import given
from hypothesis import strategies as st

setrecursionlimit(10**6)

# 1ª solución
# ===========

def mayusculaInicial1(xs: str) -> str:
    if xs:
        return "".join([xs[0].upper()] + [y.lower() for y in xs[1:]])
    return ""
```

```python
# 2ª solución
# ===========

def mayusculaInicial2(xs: str) -> str:
    def aux(ys: str) -> str:
        if ys:
            return ys[0].lower() + aux(ys[1:])
        return ""
    if xs:
        return "".join(xs[0].upper() + aux(xs[1:]))
    return ""

# 3ª solución
# ===========

def mayusculaInicial3(xs: str) -> str:
    if xs:
        return "".join([xs[0].upper()] + list(map(str.lower, xs[1:])))
    return ""

# 4ª solución
# ===========

def mayusculaInicial4(xs: str) -> str:
    return xs.capitalize()

# Comprobación de equivalencia
# ============================

# La propiedad es
@given(st.text())
def test_mayusculaInicial(xs: str) -> None:
    r = mayusculaInicial1(xs)
    assert mayusculaInicial2(xs) == r
    assert mayusculaInicial3(xs) == r

# La comprobación es
#    src> poetry run pytest -q mayuscula_inicial.py
#    1 passed in 0.26s
```

```
# Comparación de eficiencia
# =========================

def tiempo(e: str) -> None:
    """Tiempo (en segundos) de evaluar la expresión e."""
    t = Timer(e, "", default_timer, globals()).timeit(1)
    print(f"{t:0.2f} segundos")

# La comparación es
#    >>> tiempo('len(mayusculaInicial1("aB"*(10**7)))')
#    1.92 segundos
#    >>> tiempo('len(mayusculaInicial2("aB"*(10**7)))')
#    Process Python terminado (killed)
#    >>> tiempo('len(mayusculaInicial3("aB"*(10**7)))')
#    1.59 segundos
#    >>> tiempo('len(mayusculaInicial4("aB"*(10**7)))')
#    0.13 segundos
```

3.13. Mayúsculas iniciales

En Haskell

```
-- ---------------------------------------------------------------------
-- Se consideran las siguientes reglas de mayúsculas iniciales para los
-- títulos:
-- + la primera palabra comienza en mayúscula y
-- + todas las palabras que tienen 4 letras como mínimo empiezan con
--   mayúsculas
--
-- Definir la función
--    titulo :: [String] -> [String]
-- tal que (titulo ps) es la lista de las palabras de ps con
-- las reglas de mayúsculas iniciales de los títulos. Por ejemplo,
--    λ> titulo ["eL","arTE","DE","La","proGraMacion"]
--    ["El","Arte","de","la","Programacion"]
-- ---------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-unused-imports #-}
```

```haskell
module Mayusculas_iniciales where

import Data.Char (toUpper, toLower)
import Test.QuickCheck

-- 1ª solución
-- ===========

titulo1 :: [String] -> [String]
titulo1 []     = []
titulo1 (p:ps) = mayusculaInicial p : [transforma q | q <- ps]

-- (mayusculaInicial xs) es la palabra xs con la letra inicial
-- en mayúscula y las restantes en minúsculas. Por ejemplo,
--    mayusculaInicial "sEviLLa"  ==  "Sevilla"
mayusculaInicial :: String -> String
mayusculaInicial []     = []
mayusculaInicial (x:xs) = toUpper x : [toLower y | y <- xs]

-- (transforma p) es la palabra p con mayúscula inicial si su longitud
-- es mayor o igual que 4 y es p en minúscula en caso contrario
transforma :: String -> String
transforma p | length p >= 4 = mayusculaInicial p
             | otherwise     = minuscula p

-- (minuscula xs) es la palabra xs en minúscula.
minuscula :: String -> String
minuscula xs = [toLower x | x <- xs]

-- 2ª solución
-- ===========

titulo2 :: [String] -> [String]
titulo2 []     = []
titulo2 (p:ps) = mayusculaInicial p : aux ps
  where aux []     = []
        aux (q:qs) = transforma q : aux qs

-- 3ª solución
-- ===========
```

```
titulo3 :: [String] -> [String]
titulo3 []     = []
titulo3 (p:ps) = mayusculaInicial p : map transforma ps

-- Comprobación de equivalencia
-- ============================

-- La propiedad es
prop_titulo :: [String] -> Bool
prop_titulo xs =
  all (== titulo1 xs)
      [titulo2 xs,
       titulo3 xs]

-- La comprobación es
--    λ> quickCheck prop_titulo
--    +++ OK, passed 100 tests.

-- Comparación de eficiencia
-- =========================

-- La comparación es
--    λ> length (titulo1 (take (10^7) (cycle ["hOy","Es","juEves","dE","Noviembre
--    10000000
--    (2.17 secs, 1,680,592,512 bytes)
--    λ> length (titulo2 (take (10^7) (cycle ["hOy","Es","juEves","dE","Noviembre
--    10000000
--    (2.45 secs, 2,240,592,464 bytes)
--    λ> length (titulo3 (take (10^7) (cycle ["hOy","Es","juEves","dE","Noviembre
--    10000000
--    (0.16 secs, 1,440,592,464 bytes)
```

En Python

```
# ---------------------------------------------------------------------
# Se consideran las siguientes reglas de mayúsculas iniciales para los
# títulos:
# + la primera palabra comienza en mayúscula y
# + todas las palabras que tienen 4 letras como mínimo empiezan con
```

```
#   mayúsculas
#
# Definir la función
#    titulo : (list[str]) -> list[str]
# tal que titulo(ps) es la lista de las palabras de ps con
# las reglas de mayúsculas iniciales de los títulos. Por ejemplo,
#    >>> titulo(["eL", "arTE", "DE", "La", "proGraMacion"])
#    ["El", "Arte", "de", "la", "Programacion"]
# ---------------------------------------------------------------------

from sys import setrecursionlimit
from timeit import Timer, default_timer

from hypothesis import given
from hypothesis import strategies as st

setrecursionlimit(10**6)

# 1ª solución
# ===========

# (mayusculaInicial xs) es la palabra xs con la letra inicial
# en mayúscula y las restantes en minúsculas. Por ejemplo,
#    mayusculaInicial("sEviLLa")  ==  "Sevilla"
def mayusculaInicial(xs: str) -> str:
    return xs.capitalize()

# (minuscula xs) es la palabra xs en minúscula.
def minuscula(xs: str) -> str:
    return xs.lower()

# (transforma p) es la palabra p con mayúscula inicial si su longitud
# es mayor o igual que 4 y es p en minúscula en caso contrario
def transforma(p: str) -> str:
    if len(p) >= 4:
        return mayusculaInicial(p)
    return minuscula(p)

def titulo1(ps: list[str]) -> list[str]:
    if ps:
```

```
        return [mayusculaInicial(ps[0])] + [transforma(q) for q in ps[1:]]
    return []

# 2ª solución
# ===========

def titulo2(ps: list[str]) -> list[str]:
    def aux(qs: list[str]) -> list[str]:
        if qs:
            return [transforma(qs[0])] + aux(qs[1:])
        return []
    if ps:
        return [mayusculaInicial(ps[0])] + aux(ps[1:])
    return []

# 3ª solución
# ===========

def titulo3(ps: list[str]) -> list[str]:
    if ps:
        return [mayusculaInicial(ps[0])] + list(map(transforma, ps[1:]))
    return []

# Comprobación de equivalencia
# ============================

# La propiedad es
@given(st.lists(st.text()))
def test_titulo(ps: list[str]) -> None:
    r = titulo1(ps)
    assert titulo2(ps) == r
    assert titulo3(ps) == r

# La comprobación es
#    src> poetry run pytest -q mayusculas_iniciales.py
#    1 passed in 0.55s

# Comparación de eficiencia
# =========================
```

```
def tiempo(e: str) -> None:
    """Tiempo (en segundos) de evaluar la expresión e."""
    t = Timer(e, "", default_timer, globals()).timeit(1)
    print(f"{t:0.2f} segundos")

# La comparación es
#    >>> tiempo('titulo1(["eL","arTE","DE","La","proGraMacion "]*1900)')
#    0.00 segundos
#    >>> tiempo('titulo2(["eL","arTE","DE","La","proGraMacion "]*1900)')
#    0.30 segundos
#    >>> tiempo('titulo3(["eL","arTE","DE","La","proGraMacion "]*1900)')
#    0.00 segundos
#
#    >>> tiempo('titulo1(["eL","arTE","DE","La","proGraMacion "]*(2*10**6))')
#    2.93 segundos
#    >>> tiempo('titulo3(["eL","arTE","DE","La","proGraMacion "]*(2*10**6))')
#    2.35 segundos
```

3.14. Posiciones de un carácter en una cadena

En Haskell

```
-- ---------------------------------------------------------------------
-- Definir la función
--    posiciones :: Char -> String -> [Int]
-- tal que (posiciones x ys) es la lista de la posiciones del carácter x
-- en la cadena ys. Por ejemplo,
--    posiciones 'a' "Salamamca"   ==  [1,3,5,8]
-- ---------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module Posiciones_de_un_caracter_en_una_cadena where

import Data.List (elemIndices)
import Test.QuickCheck

-- 1ª solución
-- ===========
```

```
posiciones1 :: Char -> String -> [Int]
posiciones1 x ys = [n | (y,n) <- zip ys [0..], y == x]

-- 2ª solución
-- ===========

posiciones2 :: Char -> String -> [Int]
posiciones2 x ys = aux x ys 0
  where
    aux _ [] _ = []
    aux b (a:as) n | a == b    = n : aux b as (n+1)
                   | otherwise = aux b as (n+1)

-- 3ª solución
-- ===========

posiciones3 :: Char -> String -> [Int]
posiciones3 = elemIndices

-- Comprobación de equivalencia
-- ============================

-- La propiedad es
prop_posiciones :: Char -> String -> Bool
prop_posiciones x ys =
  all (== posiciones1 x ys)
      [posiciones2 x ys,
       posiciones3 x ys]

-- La comprobación es
--    λ> quickCheck prop_posiciones
--    +++ OK, passed 100 tests.

-- Comparación de eficiencia
-- =========================

-- La comparación es
--    λ> length (posiciones1 'a' (take (6*10^6) (cycle "abc")))
--    2000000
--    (2.48 secs, 1,680,591,672 bytes)
```

```
--    λ> length (posiciones2 'a' (take (6*10^6) (cycle "abc")))
--    2000000
--    (2.98 secs, 1,584,591,720 bytes)
--    λ> length (posiciones3 'a' (take (6*10^6) (cycle "abc")))
--    2000000
--    (0.11 secs, 496,591,600 bytes)
```

En Python

```
# ---------------------------------------------------------------------
# Definir la función
#    posiciones : (str, str) -> list[int]
# tal que (posiciones x ys) es la lista de la posiciones del carácter x
# en la cadena ys. Por ejemplo,
#    posiciones('a', "Salamamca")   ==  [1,3,5,8]
# ---------------------------------------------------------------------

from sys import setrecursionlimit
from timeit import Timer, default_timer

from hypothesis import given
from hypothesis import strategies as st

setrecursionlimit(10**6)

# -- 1ª solución
# -- ===========

def posiciones1(x: str, ys: str) -> list[int]:
    return [n for (n, y) in enumerate(ys) if y == x]

# -- 2ª solución
# -- ===========

def posiciones2(x: str, ys: str) -> list[int]:
    def aux(a: str, bs: str, n: int) -> list[int]:
        if bs:
            if a == bs[0]:
                return [n] + aux(a, bs[1:], n + 1)
            return aux(a, bs[1:], n + 1)
```

```
        return []
    return aux(x, ys, 0)

# -- 3ª solución
# -- ===========

def posiciones3(x: str, ys: str) -> list[int]:
    r = []
    for n, y in enumerate(ys):
        if x == y:
            r.append(n)
    return r

# Comprobación de equivalencia
# ============================

# La propiedad es
@given(st.text(), st.text())
def test_posiciones(x: str, ys: str) -> None:
    r = posiciones1(x, ys)
    assert posiciones2(x, ys) == r
    assert posiciones3(x, ys) == r

# La comprobación es
#    src> poetry run pytest -q posiciones_de_un_caracter_en_una_cadena.py
#    1 passed in 0.29s

# Comparación de eficiencia
# =========================

def tiempo(e: str) -> None:
    """Tiempo (en segundos) de evaluar la expresión e."""
    t = Timer(e, "", default_timer, globals()).timeit(1)
    print(f"{t:0.2f} segundos")

# La comparación es
#    >>> tiempo('posiciones1("a", "abc"*6000)')
#    0.00 segundos
#    >>> tiempo('posiciones2("a", "abc"*6000)')
#    0.06 segundos
```

```
#    >>> tiempo('posiciones3("a", "abc"*6000)')
#    0.00 segundos
#
#    >>> tiempo('posiciones1("a", "abc"*(2*10**7))')
#    3.02 segundos
#    >>> tiempo('posiciones3("a", "abc"*(2*10**7))')
#    3.47 segundos
```

3.15. Reconocimiento de subcadenas

En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Definir, por recursión, la función
--    esSubcadena :: String -> String -> Bool
-- tal que (esSubcadena xs ys) se verifica si xs es una subcadena de
-- ys. Por ejemplo,
--    esSubcadena "casa" "escasamente"   ==  True
--    esSubcadena "cante" "escasamente"  ==  False
--    esSubcadena "" ""                  ==  True
-- ---------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module Reconocimiento_de_subcadenas where

import Data.List (isPrefixOf, isInfixOf, tails)
import Test.QuickCheck

-- 1ª solución
-- ===========

esSubcadena1 :: String -> String -> Bool
esSubcadena1 [] _      = True
esSubcadena1  _ []     = False
esSubcadena1 xs (y:ys) = xs `isPrefixOf` (y:ys) || xs `esSubcadena1` ys

-- 2ª solución
-- ===========
```

```
esSubcadena2 :: String -> String -> Bool
esSubcadena2 xs ys =
  or [xs `isPrefixOf` zs | zs <- sufijos ys]

-- (sufijos xs) es la lista de sufijos de xs. Por ejemplo,
--    sufijos "abc"  ==  ["abc","bc","c",""]
sufijos :: String -> [String]
sufijos xs = [drop i xs | i <- [0..length xs]]

-- 3ª solución
-- ===========

esSubcadena3 :: String -> String -> Bool
esSubcadena3 xs ys =
  or [xs `isPrefixOf` zs | zs <- tails ys]

-- 4ª solución
-- ===========

esSubcadena4 :: String -> String -> Bool
esSubcadena4 xs ys =
  any (xs `isPrefixOf`) (tails ys)

-- 5ª solución
-- ===========

esSubcadena5 :: String -> String -> Bool
esSubcadena5 = (. tails) . any . isPrefixOf

-- 6ª solución
-- ===========

esSubcadena6 :: String -> String -> Bool
esSubcadena6 = isInfixOf

-- Comprobación de equivalencia
-- ============================

-- La propiedad es
prop_esSubcadena :: String -> String -> Bool
```

```haskell
prop_esSubcadena xs ys =
  all (== esSubcadena1 xs ys)
      [esSubcadena2 xs ys,
       esSubcadena3 xs ys,
       esSubcadena4 xs ys,
       esSubcadena5 xs ys,
       esSubcadena6 xs ys]

-- La comprobación es
--    λ> quickCheck prop_esSubcadena
--    +++ OK, passed 100 tests.

-- Comparación de eficiencia
-- =========================

-- La comparación es
--    λ> esSubcadena1 "abc" (replicate (5*10^4) 'd' ++ "abc")
--    True
--    (0.03 secs, 17,789,392 bytes)
--    λ> esSubcadena2 "abc" (replicate (5*10^4) 'd' ++ "abc")
--    True
--    (6.32 secs, 24,989,912 bytes)
--
--    λ> esSubcadena1 "abc" (replicate (5*10^6) 'd' ++ "abc")
--    True
--    (3.24 secs, 1,720,589,432 bytes)
--    λ> esSubcadena3 "abc" (replicate (5*10^6) 'd' ++ "abc")
--    True
--    (1.81 secs, 1,720,589,656 bytes)
--    λ> esSubcadena4 "abc" (replicate (5*10^6) 'd' ++ "abc")
--    True
--    (0.71 secs, 1,120,589,480 bytes)
--    λ> esSubcadena5 "abc" (replicate (5*10^6) 'd' ++ "abc")
--    True
--    (0.41 secs, 1,120,589,584 bytes)
--    λ> esSubcadena6 "abc" (replicate (5*10^6) 'd' ++ "abc")
--    True
--    (0.11 secs, 560,589,200 bytes)
```

En Python

```
# ---------------------------------------------------------------------
# Definir la función
#    esSubcadena : (str, str) -> bool
# tal que esSubcadena(xs ys) se verifica si xs es una subcadena de ys.
# Por ejemplo,
#    esSubcadena("casa", "escasamente")   ==  True
#    esSubcadena("cante", "escasamente")  ==  False
#    esSubcadena("", "")                  ==  True
# ---------------------------------------------------------------------

from sys import setrecursionlimit
from timeit import Timer, default_timer

from hypothesis import given
from hypothesis import strategies as st

setrecursionlimit(10**6)

# 1ª solución
# ===========

def esSubcadena1(xs: str, ys: str) -> bool:
    if not xs:
        return True
    if not ys:
        return False
    return ys.startswith(xs) or esSubcadena1(xs, ys[1:])

# 2ª solución
# ===========

# sufijos(xs) es la lista de sufijos de xs. Por ejemplo,
#    sufijos("abc")  ==  ['abc', 'bc', 'c', '']
def sufijos(xs: str) -> list[str]:
    return [xs[i:] for i in range(len(xs) + 1)]

def esSubcadena2(xs: str, ys: str) -> bool:
    return any(zs.startswith(xs) for zs in sufijos(ys))
```

```
# 3ª solución
# ===========

def esSubcadena3(xs: str, ys: str) -> bool:
    return xs in ys

# Comprobación de equivalencia
# ============================

# La propiedad es
@given(st.text(), st.text())
def test_esSubcadena(xs: str, ys: str) -> None:
    r = esSubcadena1(xs, ys)
    assert esSubcadena2(xs, ys) == r
    assert esSubcadena3(xs, ys) == r

# La comprobación es
#    src> poetry run pytest -q reconocimiento_de_subcadenas.py
#    1 passed in 0.35s

# Comparación de eficiencia
# =========================

def tiempo(e: str) -> None:
    """Tiempo (en segundos) de evaluar la expresión e."""
    t = Timer(e, "", default_timer, globals()).timeit(1)
    print(f"{t:0.2f} segundos")

# La comparación es
#    >>> tiempo('esSubcadena1("abc", "d"*(10**4) + "abc")')
#    0.02 segundos
#    >>> tiempo('esSubcadena2("abc", "d"*(10**4) + "abc")')
#    0.01 segundos
#    >>> tiempo('esSubcadena3("abc", "d"*(10**4) + "abc")')
#    0.00 segundos
#
#    >>> tiempo('esSubcadena2("abc", "d"*(10**5) + "abc")')
#    1.74 segundos
#    >>> tiempo('esSubcadena3("abc", "d"*(10**5) + "abc")')
#    0.00 segundos
```

Capítulo 4

Funciones de orden superior

En este capítulo se presentan ejercicios con definiciones por comprensión. Se corresponden con el tema 7 del curso de programación funcional con Haskell [1].

Contenido

4.1. Segmentos cuyos elementos cumplen una propiedad

En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Definir la función
--    segmentos :: (a -> Bool) -> [a] -> [[a]]
-- tal que (segmentos p xs) es la lista de los segmentos de xs cuyos
-- elementos verifican la propiedad p. Por ejemplo,
```

[1] https://jaalonso.github.io/materias/PFconHaskell/temas/tema-7.html

```
--    segmentos even [1,2,0,4,9,6,4,5,7,2]  ==  [[2,0,4],[6,4],[2]]
--    segmentos odd  [1,2,0,4,9,6,4,5,7,2]  ==  [[1],[9],[5,7]]
-- ---------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module Segmentos_cuyos_elementos_cumple_una_propiedad where

import Data.List.Split (splitWhen)
import Test.QuickCheck.HigherOrder (quickCheck')

-- 1ª solución
-- ===========

segmentos1 :: (a -> Bool) -> [a] -> [[a]]
segmentos1 _ [] = []
segmentos1 p (x:xs)
  | p x       = takeWhile p (x:xs) : segmentos1 p (dropWhile p xs)
  | otherwise = segmentos1 p xs

-- 2ª solución
-- ===========

segmentos2 :: (a -> Bool) -> [a] -> [[a]]
segmentos2 p xs = filter (not .null) (splitWhen (not . p) xs)

-- 3ª solución
-- ===========

segmentos3 :: (a -> Bool) -> [a] -> [[a]]
segmentos3 = (filter (not . null) .) . splitWhen . (not .)

-- Comprobación de equivalencia
-- ============================

-- La propiedad es
prop_segmentos :: (Int -> Bool) -> [Int] -> Bool
prop_segmentos p xs =
  all (== segmentos1 p xs)
      [segmentos2 p xs,
```

```
       segmentos3 p xs]

-- La comprobación es
--    λ> quickCheck' prop_segmentos
--    +++ OK, passed 100 tests.

-- Comparación de eficiencia
-- =========================

-- La comparación es
--    λ> length (segmentos1 even [1..5*10^6])
--    2500000
--    (2.52 secs, 2,080,591,088 bytes)
--    λ> length (segmentos2 even [1..5*10^6])
--    2500000
--    (0.78 secs, 2,860,591,688 bytes)
--    λ> length (segmentos3 even [1..5*10^6])
--    2500000
--    (0.82 secs, 2,860,592,000 bytes)
```

En Python

```
# ---------------------------------------------------------------------
# Definir la función
#    segmentos : (Callable[[A], bool], list[A]) -> list[list[A]]
# tal que segmentos(p, xs) es la lista de los segmentos de xs cuyos
# elementos verifican la propiedad p. Por ejemplo,
#    >>> segmentos1((lambda x: x % 2 == 0), [1,2,0,4,9,6,4,5,7,2])
#    [[2, 0, 4], [6, 4], [2]]
#    >>> segmentos1((lambda x: x % 2 == 1), [1,2,0,4,9,6,4,5,7,2])
#    [[1], [9], [5, 7]]
# ---------------------------------------------------------------------

from itertools import dropwhile, takewhile
from sys import setrecursionlimit
from timeit import Timer, default_timer
from typing import Callable, TypeVar

from more_itertools import split_at
```

```
setrecursionlimit(10**6)

A = TypeVar('A')

# 1ª solución
# ===========

def segmentos1(p: Callable[[A], bool], xs: list[A]) -> list[list[A]]:
    if not xs:
        return []
    if p(xs[0]):
        return [list(takewhile(p, xs))] + \
            segmentos1(p, list(dropwhile(p, xs[1:])))
    return segmentos1(p, xs[1:])

# 2ª solución
# ===========

def segmentos2(p: Callable[[A], bool], xs: list[A]) -> list[list[A]]:
    return list(filter((lambda x: x), split_at(xs, lambda x: not p(x))))

# Comparación de eficiencia
# =========================

def tiempo(e: str) -> None:
    """Tiempo (en segundos) de evaluar la expresión e."""
    t = Timer(e, "", default_timer, globals()).timeit(1)
    print(f"{t:0.2f} segundos")

# La comparación es
#    >>> tiempo('segmentos1(lambda x: x % 2 == 0, range(10**4))')
#    0.55 segundos
#    >>> tiempo('segmentos2(lambda x: x % 2 == 0, range(10**4))')
#    0.00 segundos
```

4.2. Elementos consecutivos relacionados

En Haskell

```
-- ---------------------------------------------------------------------
-- Definir la función
--    relacionados :: (a -> a -> Bool) -> [a] -> Bool
-- tal que (relacionados r xs) se verifica si para todo par (x,y) de
-- elementos consecutivos de xs se cumple la relación r. Por ejemplo,
--    relacionados (<) [2,3,7,9]                 ==  True
--    relacionados (<) [2,3,1,9]                 ==  False
-- ---------------------------------------------------------------------

module Elementos_consecutivos_relacionados where

-- 1ª solución
-- ===========

relacionados1 :: (a -> a -> Bool) -> [a] -> Bool
relacionados1 r xs = and [r x y | (x,y) <- zip xs (tail xs)]

-- 2ª solución
-- ===========

relacionados2 :: (a -> a -> Bool) -> [a] -> Bool
relacionados2 r (x:y:zs) = r x y && relacionados2 r (y:zs)
relacionados2 _ _        = True

-- 3ª solución
-- ===========

relacionados3 :: (a -> a -> Bool) -> [a] -> Bool
relacionados3 r xs = and (zipWith r xs (tail xs))

-- 4ª solución
-- ===========

relacionados4 :: (a -> a -> Bool) -> [a] -> Bool
relacionados4 r xs = all (uncurry r) (zip xs (tail xs))
```

En Python

```
# ---------------------------------------------------------------------
# Definir la función
#    relacionados : (Callable[[A, A], bool], list[A]) -> bool
# tal que relacionados(r, xs) se verifica si para todo par (x,y) de
# elementos consecutivos de xs se cumple la relación r. Por ejemplo,
#    >>> relacionados(lambda x, y: x < y, [2, 3, 7, 9])
#    True
#    >>> relacionados(lambda x, y: x < y, [2, 3, 1, 9])
#    False
# ---------------------------------------------------------------------

from typing import Callable, TypeVar

A = TypeVar('A')

# 1ª solución
# ===========

def relacionados1(r: Callable[[A, A], bool], xs: list[A]) -> bool:
    return all((r(x, y) for (x, y) in zip(xs, xs[1:])))

# 2ª solución
# ===========

def relacionados2(r: Callable[[A, A], bool], xs: list[A]) -> bool:
    if len(xs) >= 2:
        return r(xs[0], xs[1]) and relacionados2(r, xs[1:])
    return True
```

4.3. Agrupación de elementos por posición

En Haskell

```
-- ---------------------------------------------------------------------
-- Definir la función
--    agrupa :: Eq a => [[a]] -> [[a]]
-- tal que (agrupa xss) es la lista de las listas obtenidas agrupando
-- los primeros elementos, los segundos, ... Por ejemplo,
```

```haskell
--    agrupa [[1..6],[7..9],[10..20]]  ==  [[1,7,10],[2,8,11],[3,9,12]]
--
-- Comprobar con QuickChek que la longitud de todos los elementos de
-- (agrupa xs) es igual a la longitud de xs.
-- ---------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module Agrupacion_de_elementos_por_posicion where

import Data.List (transpose)
import qualified Data.Matrix as M (fromLists, toLists, transpose)
import Test.QuickCheck

-- 1ª solución
-- ===========

-- (primeros xss) es la lista de los primeros elementos de xss. Por
-- ejemplo,
--    primeros [[1..6],[7..9],[10..20]]  ==  [1,7,10]
primeros :: [[a]] -> [a]
primeros = map head

-- (restos xss) es la lista de los restos de elementos de xss. Por
-- ejemplo,
--    restos [[1..3],[7,8],[4..7]]  ==  [[2,3],[8],[5,6,7]]
restos :: [[a]] -> [[a]]
restos = map tail

agrupa1 :: Eq a => [[a]] -> [[a]]
agrupa1 []  = []
agrupa1 xss
  | [] `elem` xss = []
  | otherwise     = primeros xss : agrupa1 (restos xss)

-- 2ª solución
-- ===========

-- (conIgualLongitud xss) es la lista obtenida recortando los elementos
-- de xss para que todos tengan la misma longitud. Por ejemplo,
```

```
--    > conIgualLongitud [[1..6],[7..9],[10..20]]
--    [[1,2,3],[7,8,9],[10,11,12]]
conIgualLongitud :: [[a]] -> [[a]]
conIgualLongitud xss = map (take n) xss
  where n = minimum (map length xss)

agrupa2 :: Eq a => [[a]] -> [[a]]
agrupa2 = transpose . conIgualLongitud

-- 3ª solución
-- ===========

agrupa3 :: Eq a => [[a]] -> [[a]]
agrupa3 = M.toLists . M.transpose . M.fromLists . conIgualLongitud

-- Comprobación de equivalencia
-- ============================

-- La propiedad es
prop_agrupa :: NonEmptyList [Int] -> Bool
prop_agrupa (NonEmpty xss) =
  all (== agrupa1 xss)
      [agrupa2 xss,
       agrupa3 xss]

-- Comparación de eficiencia
-- =========================

-- La comparación es
--    λ> length (agrupa1 [[1..10^4] | _ <- [1..10^4]])
--    10000
--    (3.96 secs, 16,012,109,904 bytes)
--    λ> length (agrupa2 [[1..10^4] | _ <- [1..10^4]])
--    10000
--    (25.80 secs, 19,906,197,528 bytes)
--    λ> length (agrupa3 [[1..10^4] | _ <- [1..10^4]])
--    10000
--    (9.56 secs, 7,213,797,984 bytes)

-- La comprobación es
```

```
--    λ> quickCheck prop_agrupa
--    +++ OK, passed 100 tests.

-- La propiedad es
prop_agrupa_length :: [[Int]] -> Bool
prop_agrupa_length xss =
  and [length xs == n | xs <- agrupa1 xss]
  where n = length xss

-- La comprobación es
--    λ> quickCheck prop_agrupa_length
--    +++ OK, passed 100 tests.
```

En Python

```
# ---------------------------------------------------------------------
# Definir la función
#    agrupa : (list[list[A]]) -> list[list[A]]
# tal que agrupa(xss) es la lista de las listas obtenidas agrupando
# los primeros elementos, los segundos, ... Por ejemplo,
#    >>> agrupa([[1,6],[7,8,9],[3,4,5]])
#    [[1, 7, 3], [6, 8, 4]]
#
# Comprobar con QuickChek que la longitud de todos los elementos de
# (agrupa xs) es igual a la longitud de xs.
# ---------------------------------------------------------------------

from sys import setrecursionlimit
from timeit import Timer, default_timer
from typing import TypeVar

from hypothesis import given
from hypothesis import strategies as st
from numpy import array, transpose

setrecursionlimit(10**6)

A = TypeVar('A')

# 1ª solución
```

```
# ===========

# primeros(xss) es la lista de los primeros elementos de xss. Por
# ejemplo,
#    primeros([[1,6],[7,8,9],[3,4,5]])  ==  [1, 7, 3]
def primeros(xss: list[list[A]]) -> list[A]:
    return [xs[0] for xs in xss]

# restos(xss) es la lista de los restos de elementos de xss. Por
# ejemplo,
#    >>> restos([[1,6],[7,8,9],[3,4,5]])
#    [[6], [8, 9], [4, 5]]
def restos(xss: list[list[A]]) -> list[list[A]]:
    return [xs[1:] for xs in xss]

def agrupa1(xss: list[list[A]]) -> list[list[A]]:
    if not xss:
        return []
    if [] in xss:
        return []
    return [primeros(xss)] + agrupa1(restos(xss))

# 2ª solución
# ===========

# conIgualLongitud(xss) es la lista obtenida recortando los elementos
# de xss para que todos tengan la misma longitud. Por ejemplo,
#    >>> conIgualLongitud([[1,6],[7,8,9],[3,4,5]])
#    [[1, 6], [7, 8], [3, 4]]
def conIgualLongitud(xss: list[list[A]]) -> list[list[A]]:
    n = min(map(len, xss))
    return [xs[:n] for xs in xss]

def agrupa2(xss: list[list[A]]) -> list[list[A]]:
    yss = conIgualLongitud(xss)
    return [[ys[i] for ys in yss] for i in range(len(yss[0]))]

# 3ª solución
# ===========
```

```
def agrupa3(xss: list[list[A]]) -> list[list[A]]:
    yss = conIgualLongitud(xss)
    return list(map(list, zip(*yss)))

# 4ª solución
# ===========

def agrupa4(xss: list[list[A]]) -> list[list[A]]:
    yss = conIgualLongitud(xss)
    return (transpose(array(yss))).tolist()

# 5ª solución
# ===========

def agrupa5(xss: list[list[A]]) -> list[list[A]]:
    yss = conIgualLongitud(xss)
    r = []
    for i in range(len(yss[0])):
        f = []
        for xs in xss:
            f.append(xs[i])
        r.append(f)
    return r

# Comprobación de equivalencia
# ============================

# La propiedad es
@given(st.lists(st.lists(st.integers()), min_size=1))
def test_agrupa(xss: list[list[int]]) -> None:
    r = agrupa1(xss)
    assert agrupa2(xss) == r
    assert agrupa3(xss) == r
    assert agrupa4(xss) == r
    assert agrupa5(xss) == r

# La comprobación es
#    src> poetry run pytest -q agrupacion_de_elementos_por_posicion.py
#    1 passed in 0.74s
```

```
# Comparación de eficiencia
# =========================

def tiempo(e: str) -> None:
    """Tiempo (en segundos) de evaluar la expresión e."""
    t = Timer(e, "", default_timer, globals()).timeit(1)
    print(f"{t:0.2f} segundos")

# La comparación es
#    >>> tiempo('agrupa1([list(range(10**3)) for _ in range(10**3)])')
#    4.44 segundos
#    >>> tiempo('agrupa2([list(range(10**3)) for _ in range(10**3)])')
#    0.10 segundos
#    >>> tiempo('agrupa3([list(range(10**3)) for _ in range(10**3)])')
#    0.10 segundos
#    >>> tiempo('agrupa4([list(range(10**3)) for _ in range(10**3)])')
#    0.12 segundos
#    >>> tiempo('agrupa5([list(range(10**3)) for _ in range(10**3)])')
#    0.15 segundos
#
#    >>> tiempo('agrupa2([list(range(10**4)) for _ in range(10**4)])')
#    21.25 segundos
#    >>> tiempo('agrupa3([list(range(10**4)) for _ in range(10**4)])')
#    20.82 segundos
#    >>> tiempo('agrupa4([list(range(10**4)) for _ in range(10**4)])')
#    13.46 segundos
#    >>> tiempo('agrupa5([list(range(10**4)) for _ in range(10**4)])')
#    21.70 segundos

# La propiedad es
@given(st.lists(st.lists(st.integers()), min_size=1))
def test_agrupa_length(xss: list[list[int]]) -> None:
    n = len(xss)
    assert all((len(xs) == n for xs in agrupa2(xss)))

# La comprobación es
#    src> poetry run pytest -q agrupacion_de_elementos_por_posicion.py
#    2 passed in 1.25s
```

4.4. Concatenación de una lista de listas

En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Definir, por recursión, la función
--    conc :: [[a]] -> [a]
-- tal que (conc xss) es la concenación de las listas de xss. Por
-- ejemplo,
--    conc [[1,3],[2,4,6],[1,9]]  ==  [1,3,2,4,6,1,9]
--
-- Comprobar con QuickCheck que la longitud de (conc xss) es la suma de
-- las longitudes de los elementos de xss.
-- ---------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module Contenacion_de_una_lista_de_listas where

import Test.QuickCheck

-- 1ª solución
-- ===========

conc1 :: [[a]] -> [a]
conc1 xss = [x | xs <- xss, x <- xs]

-- 2ª solución
-- ===========

conc2 :: [[a]] -> [a]
conc2 []       = []
conc2 (xs:xss) = xs ++ conc2 xss

-- 3ª solución
-- ===========

conc3 :: [[a]] -> [a]
conc3 = foldr (++) []

-- 4ª solución
```

```haskell
-- ===========

conc4 :: [[a]] -> [a]
conc4 = concat

-- Comprobación de equivalencia
-- ============================

-- La propiedad es
prop_conc :: [[Int]] -> Bool
prop_conc xss =
  all (== conc1 xss)
      [conc2 xss,
       conc3 xss,
       conc4 xss]

-- La comprobación es
--    λ> quickCheck prop_conc
--    +++ OK, passed 100 tests.

-- Comparación de eficiencia
-- =========================

-- La comparación es
--    λ> length (conc1 [[1..n] | n <- [1..5000]])
--    12502500
--    (2.72 secs, 1,802,391,200 bytes)
--    λ> length (conc2 [[1..n] | n <- [1..5000]])
--    12502500
--    (0.27 secs, 1,602,351,160 bytes)
--    λ> length (conc3 [[1..n] | n <- [1..5000]])
--    12502500
--    (0.28 secs, 1,602,071,192 bytes)
--    λ> length (conc4 [[1..n] | n <- [1..5000]])
--    12502500
--    (0.26 secs, 1,602,071,184 bytes)

-- Comprobación de la propiedad
-- ============================
```

```
-- La propiedad es
prop_long_conc :: [[Int]] -> Bool
prop_long_conc xss =
  length (conc1 xss) == sum (map length xss)

-- La comprobación es
--    λ> quickCheck prop_long_conc
--    +++ OK, passed 100 tests.
```

En Python

```
# ---------------------------------------------------------------------
# Definir, por recursión, la función
#    conc : (list[list[A]]) -> list[A]
# tal que conc(xss) es la concenación de las listas de xss. Por
# ejemplo,
#    conc([[1,3],[2,4,6],[1,9]])  ==  [1,3,2,4,6,1,9]
#
# Comprobar con hypothesis que la longitud de conc(xss) es la suma de
# las longitudes de los elementos de xss.
# ---------------------------------------------------------------------

from functools import reduce
from operator import concat
from sys import setrecursionlimit
from timeit import Timer, default_timer
from typing import Any, TypeVar

from hypothesis import given
from hypothesis import strategies as st

setrecursionlimit(10**6)

A = TypeVar('A')

# 1ª solución
# ===========

def conc1(xss: list[list[A]]) -> list[A]:
    return [x for xs in xss for x in xs]
```

```python
# 2ª solución
# ===========

def conc2(xss: list[list[A]]) -> list[A]:
    if not xss:
        return []
    return xss[0] + conc2(xss[1:])

# 3ª solución
# ===========

def conc3(xss: Any) -> Any:
    return reduce(concat, xss)

# 4ª solución
# ===========

def conc4(xss: list[list[A]]) -> list[A]:
    r = []
    for xs in xss:
        for x in xs:
            r.append(x)
    return r

# La propiedad es
@given(st.lists(st.lists(st.integers()), min_size=1))
def test_conc(xss: list[list[int]]) -> None:
    r = conc1(xss)
    assert conc2(xss) == r
    assert conc3(xss) == r
    assert conc4(xss) == r

# La comprobación es
#    src> poetry run pytest -q concatenacion_de_una_lista_de_listas.py
#    1 passed in 0.63s

# Comparación de eficiencia
# =========================
```

```
def tiempo(e: str) -> None:
    """Tiempo (en segundos) de evaluar la expresión e."""
    t = Timer(e, "", default_timer, globals()).timeit(1)
    print(f"{t:0.2f} segundos")

# La comparación es
#    >>> tiempo('conc1([list(range(n)) for n in range(1500)])')
#    0.04 segundos
#    >>> tiempo('conc2([list(range(n)) for n in range(1500)])')
#    6.28 segundos
#    >>> tiempo('conc3([list(range(n)) for n in range(1500)])')
#    2.55 segundos
#    >>> tiempo('conc4([list(range(n)) for n in range(1500)])')
#    0.09 segundos
#
#    >>> tiempo('conc1([list(range(n)) for n in range(10000)])')
#    2.01 segundos
#    >>> tiempo('conc4([list(range(n)) for n in range(10000)])')
#    2.90 segundos
#
# Comprobación de la propiedad
# ============================

# La propiedad es
@given(st.lists(st.lists(st.integers()), min_size=1))
def test_long_conc(xss: list[list[int]]) -> None:
    assert len(conc1(xss)) == sum(map(len, xss))

# La comprobación es
#    src> poetry run pytest -q concatenacion_de_una_lista_de_listas.py
#    2 passed in 0.81s
```

4.5. Aplica según propiedad

En Haskell

```
-- ---------------------------------------------------------------------
-- Definir la función
--    filtraAplica :: (a -> b) -> (a -> Bool) -> [a] -> [b]
-- tal que (filtraAplica f p xs) es la lista obtenida aplicándole a los
```

```
-- elementos de xs que cumplen el predicado p la función f. Por ejemplo,
--    filtraAplica (4+) (<3) [1..7]  ==  [5,6]
-- ---------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module Aplica_segun_propiedad where

import Test.QuickCheck.HigherOrder (quickCheck')

-- 1ª solución
-- ===========

filtraAplica1 :: (a -> b) -> (a -> Bool) -> [a] -> [b]
filtraAplica1 f p xs = [f x | x <- xs, p x]

-- 2ª solución
-- ===========

filtraAplica2 :: (a -> b) -> (a -> Bool) -> [a] -> [b]
filtraAplica2 f p xs = map f (filter p xs)

-- 3ª solución
-- ===========

filtraAplica3 :: (a -> b) -> (a -> Bool) -> [a] -> [b]
filtraAplica3 _ _ [] = []
filtraAplica3 f p (x:xs) | p x       = f x : filtraAplica3 f p xs
                         | otherwise = filtraAplica3 f p xs

-- 4ª solución
-- ===========

filtraAplica4 :: (a -> b) -> (a -> Bool) -> [a] -> [b]
filtraAplica4 f p = foldr g []
  where g x y | p x       = f x : y
              | otherwise = y

-- 5ª solución
-- ===========
```

```haskell
filtraAplica5 :: (a -> b) -> (a -> Bool) -> [a] -> [b]
filtraAplica5 f p =
  foldr (\x y -> if p x then f x : y else y) []

-- Comprobación de equivalencia
-- ============================

-- La propiedad es
prop_filtraAplica :: (Int -> Int) -> (Int -> Bool) -> [Int] -> Bool
prop_filtraAplica f p xs =
  all (== filtraAplica1 f p xs)
      [filtraAplica2 f p xs,
       filtraAplica3 f p xs,
       filtraAplica4 f p xs,
       filtraAplica5 f p xs]

-- La comprobación es
--    λ> quickCheck' prop_filtraAplica
--    +++ OK, passed 100 tests.

-- Comparación de eficiencia
-- =========================

-- La comparación es
--    λ> sum (filtraAplica1 id even [1..5*10^6])
--    6250002500000
--    (2.92 secs, 1,644,678,696 bytes)
--    λ> sum (filtraAplica2 id even [1..5*10^6])
--    6250002500000
--    (1.17 secs, 1,463,662,848 bytes)
--    λ> sum (filtraAplica3 id even [1..5*10^6])
--    6250002500000
--    (3.18 secs, 1,964,678,640 bytes)
--    λ> sum (filtraAplica4 id even [1..5*10^6])
--    6250002500000
--    (2.64 secs, 1,924,678,752 bytes)
--    λ> sum (filtraAplica5 id even [1..5*10^6])
--    6250002500000
--    (2.61 secs, 1,824,678,712 bytes)
```

En Python

```
# ---------------------------------------------------------------------
# Definir la función
#    filtraAplica : (Callable[[A], B], Callable[[A], bool], list[A])
#                   -> list[B]
# tal que filtraAplica(f, p, xs) es la lista obtenida aplicándole a los
# elementos de xs que cumplen el predicado p la función f. Por ejemplo,
#    >>> filtraAplica(lambda x: x + 4, lambda x: x < 3, range(1, 7))
#    [5, 6]
# ---------------------------------------------------------------------

from functools import reduce
from sys import setrecursionlimit
from timeit import Timer, default_timer
from typing import Callable, TypeVar

from hypothesis import given
from hypothesis import strategies as st

setrecursionlimit(10**6)

A = TypeVar('A')
B = TypeVar('B')

# 1ª solución
# ===========

def filtraAplica1(f: Callable[[A], B],
                  p: Callable[[A], bool],
                  xs: list[A]) -> list[B]:
    return [f(x) for x in xs if p(x)]

# 2ª solución
# ===========

def filtraAplica2(f: Callable[[A], B],
                  p: Callable[[A], bool],
                  xs: list[A]) -> list[B]:
    return list(map(f, filter(p, xs)))
```

```
# 3ª solución
# ===========

def filtraAplica3(f: Callable[[A], B],
                  p: Callable[[A], bool],
                  xs: list[A]) -> list[B]:
    if not xs:
        return []
    if p(xs[0]):
        return [f(xs[0])] + filtraAplica3(f, p, xs[1:])
    return filtraAplica3(f, p, xs[1:])

# 4ª solución
# ===========

def filtraAplica4(f: Callable[[A], B],
                  p: Callable[[A], bool],
                  xs: list[A]) -> list[B]:
    def g(ys: list[B], x: A) -> list[B]:
        if p(x):
            return ys + [f(x)]
        return ys

    return reduce(g, xs, [])

# 5ª solución
# ===========

def filtraAplica5(f: Callable[[A], B],
                  p: Callable[[A], bool],
                  xs: list[A]) -> list[B]:
    r = []
    for x in xs:
        if p(x):
            r.append(f(x))
    return r

# Comprobación de equivalencia
# ============================
```

```
# La propiedad es
@given(st.lists(st.integers()))
def test_filtraAplica(xs: list[int]) -> None:
    f = lambda x: x + 4
    p = lambda x: x < 3
    r = filtraAplica1(f, p, xs)
    assert filtraAplica2(f, p, xs) == r
    assert filtraAplica3(f, p, xs) == r
    assert filtraAplica4(f, p, xs) == r
    assert filtraAplica5(f, p, xs) == r

# La comprobación es
#    src> poetry run pytest -q aplica_segun_propiedad.py
#    1 passed in 0.25s

# Comparación de eficiencia
# =========================

def tiempo(e: str) -> None:
    ”””Tiempo (en segundos) de evaluar la expresión e.”””
    t = Timer(e, ””, default_timer, globals()).timeit(1)
    print(f”{t:0.2f} segundos”)

# La comparación es
#    >>> tiempo('filtraAplica1(lambda x: x, lambda x: x % 2 == 0, range(10**5))')
#    0.02 segundos
#    >>> tiempo('filtraAplica2(lambda x: x, lambda x: x % 2 == 0, range(10**5))')
#    0.01 segundos
#    >>> tiempo('filtraAplica3(lambda x: x, lambda x: x % 2 == 0, range(10**5))')
#    Process Python violación de segmento (core dumped)
#    >>> tiempo('filtraAplica4(lambda x: x, lambda x: x % 2 == 0, range(10**5))')
#    4.07 segundos
#    >>> tiempo('filtraAplica5(lambda x: x, lambda x: x % 2 == 0, range(10**5))')
#    0.01 segundos
#
#    >>> tiempo('filtraAplica1(lambda x: x, lambda x: x % 2 == 0, range(10**7))')
#    1.66 segundos
#    >>> tiempo('filtraAplica2(lambda x: x, lambda x: x % 2 == 0, range(10**7))')
#    1.00 segundos
#    >>> tiempo('filtraAplica5(lambda x: x, lambda x: x % 2 == 0, range(10**7))')
```

```
#    1.21 segundos
```

4.6. Máximo de una lista

En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Definir la función
--    maximo :: Ord a => [a] -> a
-- tal que (maximo xs) es el máximo de la lista xs. Por ejemplo,
--    maximo [3,7,2,5]                  ==  7
--    maximo ["todo","es","falso"]      ==  "todo"
--    maximo ["menos","alguna","cosa"]  ==  "menos"
-- ---------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-incomplete-patterns #-}

module Maximo_de_una_lista where

import Data.List (foldl1')
import Test.QuickCheck

-- 1ª solución
-- ===========

maximo1 :: Ord a => [a] -> a
maximo1 [x]      = x
maximo1 (x:y:ys) = max x (maximo1 (y:ys))

-- 2ª solución
-- ===========

maximo2 :: Ord a => [a] -> a
maximo2 = foldr1 max

-- 3ª solución
-- ===========

maximo3 :: Ord a => [a] -> a
maximo3 = foldl1' max
```

```
-- 4ª solución
-- ===========

maximo4 :: Ord a => [a] -> a
maximo4 = maximum

-- Comprobación de equivalencia
-- ============================

-- La propiedad es
prop_maximo :: NonEmptyList Int -> Bool
prop_maximo (NonEmpty xs) =
  all (== maximo1 xs)
      [maximo2 xs,
       maximo3 xs]

-- La comprobación es
--    λ> quickCheck prop_maximo
--    +++ OK, passed 100 tests.

-- Comparación de eficiencia
-- =========================

-- La comparación es
--    λ> maximo1 [0..5*10^6]
--    5000000
--    (3.42 secs, 1,783,406,728 bytes)
--    λ> maximo2 [0..5*10^6]
--    5000000
--    (0.80 secs, 934,638,080 bytes)
--    λ> maximo3 [0..5*10^6]
--    5000000
--    (0.12 secs, 360,591,360 bytes)
--    λ> maximo4 [0..5*10^6]
--    5000000
--    (1.40 secs, 892,891,608 bytes)
```

En Python

```
# ---------------------------------------------------------------------
# Definir la función
#    maximo : (list[A]) -> A:
# tal que maximo(xs) es el máximo de la lista xs. Por ejemplo,
#    maximo([3,7,2,5])                    ==  7
#    maximo(["todo","es","falso"])        ==  "todo"
#    maximo(["menos","alguna","cosa"])    ==  "menos"
# ---------------------------------------------------------------------

from functools import reduce
from sys import setrecursionlimit
from timeit import Timer, default_timer
from typing import TypeVar, Union

from hypothesis import given
from hypothesis import strategies as st

setrecursionlimit(10**6)

A = TypeVar('A', bound=Union[int, float, str])

# 1ª solución
# ===========

def maximo1(xs: list[A]) -> A:
    if len(xs) == 1:
        return xs[0]
    return max(xs[0], maximo1(xs[1:]))

# 2ª solución
# ===========

def maximo2(xs: list[A]) -> A:
    return reduce(max, xs)

# 3ª solución
# ===========

def maximo3(xs: list[A]) -> A:
```

```python
    return max(xs)

# Comprobación de equivalencia
# ============================

# La propiedad es
@given(st.lists(st.integers(), min_size=2))
def test_maximo(xs: list[int]) -> None:
    r = maximo1(xs)
    assert maximo2(xs) == r
    assert maximo3(xs) == r

# La comprobación es
#    src> poetry run pytest -q maximo_de_una_lista.py
#    1 passed in 0.33s

# Comparación de eficiencia
# =========================

def tiempo(e: str) -> None:
    """Tiempo (en segundos) de evaluar la expresión e."""
    t = Timer(e, "", default_timer, globals()).timeit(1)
    print(f"{t:0.2f} segundos")

# La comparación es
#    >>> tiempo('maximo1(range(2*10**4))')
#    0.03 segundos
#    >>> tiempo('maximo2(range(2*10**4))')
#    0.00 segundos
#    >>> tiempo('maximo3(range(2*10**4))')
#    0.00 segundos
#
#    >>> tiempo('maximo2(range(5*10**6))')
#    0.38 segundos
#    >>> tiempo('maximo3(range(5*10**6))')
#    0.21 segundos
```

Capítulo 5

Tipos definidos y tipos de datos algebraicos

En este capítulo se presentan ejercicios con definiciones por comprensión. Se corresponden con el tema 9 del curso de programación funcional con Haskell [1].

Contenido

[1] https://jaalonso.github.io/materias/PFconHaskell/temas/tema-9.html

5.1. Movimientos en el plano

En Haskell

```
-- ---------------------------------------------------------------------
-- Se consideran el tipo de las posiciones del plano definido por
--    type Posicion = (Int,Int)
-- y el tipo de las direcciones definido por
--    data Direccion = Izquierda | Derecha | Arriba | Abajo
--      deriving Show
--
-- Definir las siguientes funciones
--    opuesta     :: Direccion -> Direccion
--    movimiento  :: Posicion -> Direccion -> Posicion
--    movimientos :: Posicion -> [Direccion] -> Posicion
-- tales que
```

```
-- + (opuesta d) es la dirección opuesta de d. Por ejemplo,
--      opuesta Izquierda == Derecha
-- + (movimiento p d) es la posición reultante de moverse, desde la
--   posición p, un paso en la dirección d . Por ejemplo,
--      movimiento (2,5) Arriba          == (2,6)
--      movimiento (2,5) (opuesta Abajo) == (2,6)
-- + (movimientos p ds) es la posición obtenida aplicando la lista de
--   movimientos según las direcciones de ds a la posición p. Por ejemplo,
--      movimientos (2,5)  [Arriba, Izquierda] == (1,6)
-- ---------------------------------------------------------------------

module Movimientos_en_el_plano where

type Posicion = (Int,Int)

data Direccion = Izquierda | Derecha | Arriba | Abajo
  deriving Show

-- Definición de opuesta
-- =====================

opuesta :: Direccion -> Direccion
opuesta Izquierda = Derecha
opuesta Derecha   = Izquierda
opuesta Arriba    = Abajo
opuesta Abajo     = Arriba

-- 1ª definición de movimiento
-- ===========================

movimiento1 :: Posicion -> Direccion -> Posicion
movimiento1 (x,y) Izquierda = (x-1,y)
movimiento1 (x,y) Derecha   = (x+1,y)
movimiento1 (x,y) Arriba    = (x,y+1)
movimiento1 (x,y) Abajo     = (x,y-1)

-- 2ª definición de movimiento
-- ===========================

movimiento2 :: Posicion -> Direccion -> Posicion
```

```haskell
movimiento2 (x,y) d =
  case d of
    Izquierda -> (x-1,y)
    Derecha   -> (x+1,y)
    Arriba    -> (x,y+1)
    Abajo     -> (x,y-1)

-- 1ª definición de movimientos
-- ============================

movimientos1 :: Posicion -> [Direccion] -> Posicion
movimientos1 p []     = p
movimientos1 p (d:ds) = movimientos1 (movimiento1 p d) ds

-- 2ª definición de movimientos
-- ============================

movimientos2 :: Posicion -> [Direccion] -> Posicion
movimientos2 = foldl movimiento1
```

En Python

```python
# ---------------------------------------------------------------------
# Se consideran el tipo de las posiciones del plano definido por
#    Posicion = tuple[int, int]
# y el tipo de las direcciones definido por
#    Direccion = Enum('Direccion', ['Izquierda', 'Derecha', 'Arriba', 'Abajo'])
# Definir las siguientes funciones
#    opuesta     : (Direccion) -> Direccion
#    movimiento  : (Posicion, Direccion) -> Posicion
#    movimientos : (Posicion, list[Direccion]) -> Posicion
# tales que
# + opuesta(d) es la dirección opuesta de d. Por ejemplo,
#      opuesta1(Direccion.Izquierda) == Direccion.Derecha
# + movimiento(p d) es la posición reultante de moverse, desde la
#   posición p, un paso en la dirección d . Por ejemplo,
#      movimiento1((2, 5), Direccion.Arriba)          == (2, 6)
#      movimiento1((2, 5), opuesta1(Direccion.Abajo)) == (2, 6)
# + movimientos(p, ds) es la posición obtenida aplicando la lista de
#   movimientos según las direcciones de ds a la posición p. Por
```

```
#   ejemplo,
#     >>> movimientos1((2, 5), [Direccion.Arriba, Direccion.Izquierda])
#     (1, 6)
# ---------------------------------------------------------------------

from enum import Enum
from functools import reduce

Posicion = tuple[int, int]

Direccion = Enum('Direccion', ['Izquierda', 'Derecha', 'Arriba', 'Abajo'])

# 1ª definición de opuesta
# ========================

def opuesta1(d: Direccion) -> Direccion:
    if d == Direccion.Izquierda:
        return Direccion.Derecha
    if d == Direccion.Derecha:
        return Direccion.Izquierda
    if d == Direccion.Arriba:
        return Direccion.Abajo
    if d == Direccion.Abajo:
        return Direccion.Arriba
    assert False

# 2ª definición de opuesta
# ========================

def opuesta2(d: Direccion) -> Direccion:
    match d:
        case Direccion.Izquierda:
            return Direccion.Derecha
        case Direccion.Derecha:
            return Direccion.Izquierda
        case Direccion.Arriba:
            return Direccion.Abajo
        case Direccion.Abajo:
            return Direccion.Arriba
    assert False
```

```
# 1ª definición de movimiento
# ===========================

def movimiento1(p: Posicion, d: Direccion) -> Posicion:
    (x, y) = p
    if d == Direccion.Izquierda:
        return (x - 1, y)
    if d == Direccion.Derecha:
        return (x + 1, y)
    if d == Direccion.Arriba:
        return (x, y + 1)
    if d == Direccion.Abajo:
        return (x, y - 1)
    assert False

# 2ª definición de movimiento
# ===========================

def movimiento2(p: Posicion, d: Direccion) -> Posicion:
    (x, y) = p
    match d:
        case Direccion.Izquierda:
            return (x - 1, y)
        case Direccion.Derecha:
            return (x + 1, y)
        case Direccion.Arriba:
            return (x, y + 1)
        case Direccion.Abajo:
            return (x, y - 1)
    assert False

# 1ª definición de movimientos
# ============================

def movimientos1(p: Posicion, ds: list[Direccion]) -> Posicion:
    if not ds:
        return p
    return movimientos1(movimiento1(p, ds[0]), ds[1:])
```

```python
# 2ª definición de movimientos
# ============================

def movimientos2(p: Posicion, ds: list[Direccion]) -> Posicion:
    return reduce(movimiento1, ds, p)
```

5.2. El tipo de figuras geométricas

En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Se consideran las figuras geométricas formadas por circulos
-- (definidos por su radio) y rectángulos (definidos por su base y su
-- altura). El tipo de las figura geométricas se define por
--    data Figura = Circulo Float | Rect Float Float
--
-- Definir las funciones
--    area     :: Figura -> Float
--    cuadrado :: Float -> Figura
-- tales que
-- + (area f) es el área de la figura f. Por ejemplo,
--      area (Circulo 1)   ==  3.1415927
--      area (Circulo 2)   ==  12.566371
--      area (Rect 2 5)    ==  10.0
-- + (cuadrado n) es el cuadrado de lado n. Por ejemplo,
--      area (cuadrado 3)  ==  9.0
-- ---------------------------------------------------------------------

module El_tipo_de_figuras_geometricas where

data Figura = Circulo Float | Rect Float Float

area :: Figura -> Float
area (Circulo r) = pi*r^2
area (Rect x y)  = x*y

cuadrado :: Float -> Figura
cuadrado n = Rect n n
```

En Python

```
# ---------------------------------------------------------------------
# Se consideran las figuras geométricas formadas por circulos
# (definidos por su radio) y rectángulos (definidos por su base y su
# altura). El tipo de las figura geométricas se define por
#    @dataclass
#    class Figura:
#        """Figuras geométricas"""
#
#    @dataclass
#    class Circulo(Figura):
#        r: float
#
#    @dataclass
#    class Rect(Figura):
#        x: float
#        y: float
#
# Definir las funciones
#    area     : (Figura) -> float
#    cuadrado : (float) -> Figura
# tales que
# + area(f) es el área de la figura f. Por ejemplo,
#      area(Circulo(1))  ==  3.141592653589793
#      area(Circulo(2))  ==  12.566370614359172
#      area(Rect(2, 5))  ==  10
# + cuadrado(n) es el cuadrado de lado n. Por ejemplo,
#      area(cuadrado(3))  ==  9.0
# ---------------------------------------------------------------------

from dataclasses import dataclass
from math import pi


@dataclass
class Figura:
    """Figuras geométricas"""

@dataclass
class Circulo(Figura):
```

```
    r: float

@dataclass
class Rect(Figura):
    x: float
    y: float

def area(f: Figura) -> float:
    match f:
        case Circulo(r):
            return pi * r**2
        case Rect(x, y):
            return x * y
    assert False

def cuadrado(n: float) -> Figura:
    return Rect(n, n)
```

5.3. El tipo de los números naturales

En Haskell

```
-- ---------------------------------------------------------------------
-- El tipo de los números raturales se puede definir por
--    data Nat = Cero | Suc Nat
--      deriving (Show, Eq)
-- de forma que (Suc (Suc (Suc Cero))) representa el número 3.
--
-- Definir las siguientes funciones
--    nat2int :: Nat -> Int
--    int2nat :: Int -> Nat
--    suma    :: Nat -> Nat -> Nat
-- tales que
-- + (nat2int n) es el número entero correspondiente al número natural
--   n. Por ejemplo,
--      nat2int (Suc (Suc (Suc Cero)))  ==  3
-- + (int2nat n) es el número natural correspondiente al número entero n. Por eje
--      int2nat 3  ==  Suc (Suc (Suc Cero))
-- + (suma m n) es la suma de los número naturales m y n. Por ejemplo,
--      λ> suma (Suc (Suc Cero)) (Suc Cero)
```

```haskell
--      Suc (Suc (Suc Cero))
--      λ> nat2int (suma (Suc (Suc Cero)) (Suc Cero))
--      3
--      λ> nat2int (suma (int2nat 2) (int2nat 1))
--      3
-- ---------------------------------------------------------------------

module El_tipo_de_los_numeros_naturales where

data Nat = Cero | Suc Nat
  deriving (Show, Eq)

nat2int :: Nat -> Int
nat2int Cero    = 0
nat2int (Suc n) = 1 + nat2int n

int2nat :: Int -> Nat
int2nat 0 = Cero
int2nat n = Suc (int2nat (n-1))

suma :: Nat -> Nat -> Nat
suma Cero    n = n
suma (Suc m) n = Suc (suma m n)
```

En Python

```python
# ---------------------------------------------------------------------
# El tipo de los números raturales se puede definir por
#    @dataclass
#    class Nat:
#        pass
#
#    @dataclass
#    class Cero(Nat):
#        pass
#
#    @dataclass
#    class Suc(Nat):
#        n: Nat
# de forma que Suc(Suc(Suc(Cero()))) representa el número 3.
```

```
#
# Definir las siguientes funciones
#    nat2int : (Nat) -> int
#    int2nat : (int) -> Nat
#    suma    : (Nat, Nat) -> Nat
# tales que
# + nat2int(n) es el número entero correspondiente al número natural
#   n. Por ejemplo,
#      nat2int(Suc(Suc(Suc(Cero())))) == 3
# + int2nat(n) es el número natural correspondiente al número entero
#   n. Por ejemplo,
#      int2nat(3) == Suc(Suc(Suc(Cero())))
# + suma(m, n) es la suma de los número naturales m y n. Por ejemplo,
#      >>> suma(Suc(Suc(Cero())), Suc(Cero()))
#      Suc(Suc(Suc(Cero())))
#      >>> nat2int(suma(Suc(Suc(Cero())), Suc(Cero())))
#      3
#      >>> nat2int(suma(int2nat(2), int2nat(1)))
#      3
# ---------------------------------------------------------------------

from dataclasses import dataclass


@dataclass
class Nat:
    pass

@dataclass
class Cero(Nat):
    pass

@dataclass
class Suc(Nat):
    n: Nat

def nat2int(n: Nat) -> int:
    match n:
        case Cero():
            return 0
```

```
        case Suc(n):
            return 1 + nat2int(n)
    assert False

def int2nat(n: int) -> Nat:
    if n == 0:
        return Cero()
    return Suc(int2nat(n - 1))

def suma(m: Nat, n: Nat) -> Nat:
    match m:
        case Cero():
            return n
        case Suc(m):
            return Suc(suma(m, n))
    assert False
```

5.4. El tipo de las listas

En Haskell

```haskell
-- ---------------------------------------------------------------------
-- El tipo de las listas, con elementos de tipo a, se puede definir por
--    data Lista a = Nil | Cons a (Lista a)
-- Por ejemplo, la lista [4,2,5] se representa por
-- Cons 4 (Cons 2 (Cons 5 Nil)).
--
-- Definir la función
--    longitud :: Lista a -> Int
-- tal que  (longitud xs) es la longitud de la lista xs. Por ejemplo,
--    longitud (Cons 4 (Cons 2 (Cons 5 Nil)))  ==  3
-- ---------------------------------------------------------------------

module El_tipo_de_las_listas where

data Lista a = Nil | Cons a (Lista a)

longitud :: Lista a -> Int
longitud Nil         = 0
longitud (Cons _ xs) = 1 + longitud xs
```

En Python

```
# ---------------------------------------------------------------------
# El tipo de las listas, con elementos de tipo a, se puede definir por
#    @dataclass
#    class Lista(Generic[A]):
#        pass
#
#    @dataclass
#    class Nil(Lista[A]):
#        pass
#
#    @dataclass
#    class Cons(Lista[A]):
#        x: A
#        xs: Lista[A]
# Por ejemplo, la lista [4,2,5] se representa por
# Cons(4, Cons(2, Cons(5, Nil()))).
#
# Definir la función
#    longitud :: Lista a -> Int
# tal que  (longitud xs) es la longitud de la lista xs. Por ejemplo,
#    >>> longitud(Cons(4, Cons(2, Cons(5, Nil()))))
#    3
# ---------------------------------------------------------------------

from dataclasses import dataclass
from typing import Generic, TypeVar

A = TypeVar("A")

@dataclass
class Lista(Generic[A]):
    pass

@dataclass
class Nil(Lista[A]):
    pass

@dataclass
class Cons(Lista[A]):
```

```
    x: A
    xs: Lista[A]

def longitud(xs: Lista[A]) -> int:
    match xs:
        case Nil():
            return 0
        case Cons(_, xs):
            return 1 + longitud(xs)
    assert False
```

5.5. El tipo de los árboles binarios

En Haskell

```
-- ---------------------------------------------------------------------
-- El árbol binario
--        5
--       / \
--      /   \
--     3     7
--    / \   / \
--   1   4 6   9
-- se puede representar por
--   ejArbol = Nodo (Nodo (Hoja 1) 3 (Hoja 4))
--                  5
--                  (Nodo (Hoja 6) 7 (Hoja 9))
--
-- El tipo de los árboles binarios se puede definir por
--   data Arbol = Hoja Int
--              | Nodo Arbol Int Arbol
--
-- Definir las funciones
--   ocurre :: Int -> Arbol -> Bool
--   aplana :: Arbol -> [Int]
-- tales que
-- + (ocurre m a) se verifica si m ocurre en el árbol a. Por ejemplo,
--     ocurre 4  ejArbol  ==  True
--     ocurre 10 ejArbol  ==  False
-- + (aplana a) es la lista obtenida aplanando el árbol a. Por ejemplo,
```

```
--      aplana ejArbol  ==  [1,3,4,5,6,7,9]
-- ---------------------------------------------------------------------

module El_tipo_de_los_arboles_binarios where

data Arbol = Hoja Int
           | Nodo Arbol Int Arbol

ejArbol :: Arbol
ejArbol = Nodo (Nodo (Hoja 1) 3 (Hoja 4))
               5
               (Nodo (Hoja 6) 7 (Hoja 9))

ocurre :: Int -> Arbol -> Bool
ocurre m (Hoja n)     = m == n
ocurre m (Nodo i n d) = m == n || ocurre m i || ocurre m d

aplana :: Arbol -> [Int]
aplana (Hoja n)     = [n]
aplana (Nodo i n d) = aplana i ++ [n] ++ aplana d
```

En Python

```
# ---------------------------------------------------------------------
# El árbol binario
#         5
#        / \
#       /   \
#      3     7
#     / \   / \
#    1   4 6   9
# se puede representar por
#    ejArbol = Nodo (Nodo (Hoja 1) 3 (Hoja 4))
#                   5
#                   (Nodo (Hoja 6) 7 (Hoja 9))
#
# El tipo de los árboles binarios se puede definir por
#    @dataclass
#    class Arbol:
#        pass
```

```
#
#    @dataclass
#    class Hoja(Arbol):
#        x: int
#
#    @dataclass
#    class Nodo(Arbol):
#        i: Arbol
#        x: int
#        d: Arbol
#
# Definir las funciones
#    ocurre : (int, Arbol) -> bool
#    aplana : (Arbol) -> list[int]
# tales que
# + ocurre(m, a) se verifica si m ocurre en el árbol a. Por ejemplo,
#      ocurre(4, ejArbol)  ==  True
#      ocurre(0, ejArbol)  ==  False
# + aplana(a) es la lista obtenida aplanando el árbol a. Por ejemplo,
#      aplana(ejArbol)  ==  [1,3,4,5,6,7,9]
# ---------------------------------------------------------------------

from dataclasses import dataclass


@dataclass
class Arbol:
    pass

@dataclass
class Hoja(Arbol):
    x: int

@dataclass
class Nodo(Arbol):
    i: Arbol
    x: int
    d: Arbol

ejArbol = Nodo(Nodo(Hoja(1), 3, Hoja(4)),
```

```
               5,
               Nodo(Hoja(6), 7, Hoja(9)))

def ocurre(m: int, a: Arbol) -> bool:
    match a:
        case Hoja(n):
            return m == n
        case Nodo(i, n, d):
            return m == n or ocurre(m, i) or ocurre(m, d)
    assert False

def aplana(a: Arbol) -> list[int]:
    match a:
        case Hoja(n):
            return [n]
        case Nodo(i, n, d):
            return aplana(i) + [n] + aplana(d)
    assert False
```

5.6. El tipo de las fórmulas: Variables de una fórmula

En Haskell

```
-- ---------------------------------------------------------------------
-- El tipo de las fórmulas proposicionales se puede definir por
--    data FProp = Const Bool
--               | Var Char
--               | Neg FProp
--               | Conj FProp FProp
--               | Impl FProp FProp
--      deriving Show
-- de modo que la fórmula A → ⊥ ∧ ¬B se representa por
--    Impl (Var 'A') (Conj (Const False) (Neg (Var 'B')))
--
-- Definir la función
--    variables :: FProp -> [Char]
-- tal que (variables p) es la lista de las variables de la fórmula
-- p. Por ejemplo,
```

```haskell
--    λ> variables (Impl (Var 'A') (Conj (Const False) (Neg (Var 'B'))))
--    "AB"
--    λ> variables (Impl (Var 'A') (Conj (Var 'A') (Neg (Var 'B'))))
--    "AAB"
-- ---------------------------------------------------------------------

module Variables_de_una_formula where

data FProp = Const Bool
           | Var Char
           | Neg FProp
           | Conj FProp FProp
           | Impl FProp FProp
  deriving Show

variables :: FProp -> [Char]
variables (Const _)  = []
variables (Var x)    = [x]
variables (Neg p)    = variables p
variables (Conj p q) = variables p ++ variables q
variables (Impl p q) = variables p ++ variables q
```

En Python

```python
# ---------------------------------------------------------------------
# El tipo de las fórmulas proposicionales se puede definir por
#    @dataclass
#    class FProp:
#        pass
#
#    @dataclass
#    class Const(FProp):
#        x: bool
#
#    @dataclass
#    class Var(FProp):
#        x: str
#
#    @dataclass
#    class Neg(FProp):
```

```
#        x: FProp
#
#    @dataclass
#    class Conj(FProp):
#        x: FProp
#        y: FProp
#
#    @dataclass
#    class Impl(FProp):
#        x: FProp
#        y: FProp
# de modo que la fórmula A → ⊥ ∧ ¬B se representa por
#    Impl(Var('A'), Conj(Const(False), Neg (Var('B'))))
#
# Definir la función
#    variables : (FProp) -> list[str]:
# tal que variables(p) es la lista de las variables de la fórmula
# p. Por ejemplo,
#    >>> variables (Impl(Var('A'), Conj(Const(False), Neg (Var('B')))))
#    ['A', 'B']
#    >>> variables (Impl(Var('A'), Conj(Var('A'), Neg (Var('B')))))
#    ['A', 'A', 'B']
# ---------------------------------------------------------------------

from dataclasses import dataclass


@dataclass
class FProp:
    pass

@dataclass
class Const(FProp):
    x: bool

@dataclass
class Var(FProp):
    x: str

@dataclass
```

```python
class Neg(FProp):
    x: FProp

@dataclass
class Conj(FProp):
    x: FProp
    y: FProp

@dataclass
class Impl(FProp):
    x: FProp
    y: FProp

def variables(f: FProp) -> list[str]:
    match f:
        case Const(_):
            return []
        case Var(x):
            return [x]
        case Neg(p):
            return variables(p)
        case Conj(p, q):
            return variables(p) + variables(q)
        case Impl(p, q):
            return variables(p) + variables(q)
    assert False
```

5.7. El tipo de las fórmulas: Valor de una fórmula

En Haskell

```haskell
-- ---------------------------------------------------------------------
-- El tipo de las fórmulas proposicionales se puede definir por
--    data FProp = Const Bool
--               | Var Char
--               | Neg FProp
--               | Conj FProp FProp
--               | Impl FProp FProp
```

```
--      deriving Show
-- de modo que la fórmula A → ⊥ ∧ ¬B se representa por
--    Impl (Var 'A') (Conj (Const False) (Neg (Var 'B')))
--
-- Una interpretación de una fórmula es una función de sus variables en
-- los booleanos. Por ejemplo, la interpretación que a la variable A le
-- asigna verdadero y a la B falso se puede representar por
--    [('A', True), ('B', False)]
--
-- El tipo de las intepretaciones de puede definir por
--    type Interpretacion = [(Char, Bool)]
--
-- El valor de una fórmula en una interpretación se calcula usando las
-- funciones de verdad de las conectivas que se muestran a continuación
--    |---+----|   |---+---+-------+-------|
--    | p | ¬p |   | p | q | p ∧ q | p → q |
--    |---+----|   |---+---+-------+-------|
--    | T | F  |   | T | T | T     | T     |
--    | F | T  |   | T | F | F     | F     |
--    |---+----|   | F | T | F     | T     |
--                 | F | F | F     | T     |
--                 |---+---+-------+-------|
--
-- Definir la función
--    valor :: Interpretacion -> FProp -> Bool
-- tal que (valor i p) es el valor de la fórmula p en la interpretación
-- i. Por ejemplo,
--    λ> p = Impl (Var 'A') (Conj (Var 'A') (Var 'B'))
--    λ> valor [('A',False),('B',False)] p
--    True
--    λ> valor [('A',True),('B',False)] p
--    False
-- ---------------------------------------------------------------------

module Valor_de_una_formula where

data FProp = Const Bool
           | Var Char
           | Neg FProp
           | Conj FProp FProp
```

```
           | Impl FProp FProp
  deriving Show

type Interpretacion = [(Char, Bool)]

valor :: Interpretacion -> FProp -> Bool
valor _ (Const b)  = b
valor i (Var x)    = busca x i
valor i (Neg p)    = not (valor i p)
valor i (Conj p q) = valor i p && valor i q
valor i (Impl p q) = valor i p <= valor i q

-- (busca c t) es el valor del primer elemento de la lista de asociación
-- t cuya clave es c. Por ejemplo,
--    busca 2 [(1,'a'),(3,'d'),(2,'c')]  ==  'c'
busca :: Eq c => c -> [(c,v)] -> v
busca c t = head [v | (c',v) <- t, c == c']
```

En Python

```
# ---------------------------------------------------------------------
# El tipo de las fórmulas proposicionales se puede definir por
#    @dataclass
#    class FProp:
#        pass
#
#    @dataclass
#    class Const(FProp):
#        x: bool
#
#    @dataclass
#    class Var(FProp):
#        x: str
#
#    @dataclass
#    class Neg(FProp):
#        x: FProp
#
#    @dataclass
#    class Conj(FProp):
```

```
#        x: FProp
#        y: FProp
#
#    @dataclass
#    class Impl(FProp):
#        x: FProp
#        y: FProp
# de modo que la fórmula A → ⊥ ∧ ¬B se representa por
#    Impl(Var('A'), Conj(Const(False), Neg (Var('B'))))
#
# Una interpretación de una fórmula es una función de sus variables en
# los booleanos. Por ejemplo, la interpretación que a la variable A le
# asigna verdadero y a la B falso se puede representar por
#    [('A', True), ('B', False)]
#
# El tipo de las intepretaciones de puede definir por
#    Interpretacion = list[tuple[str, bool]]
#
# El valor de una fórmula en una interpretación se calcula usando las
# funciones de verdad de las conectivas que se muestran a continuación
#    |---+----|    |---+---+-------+-------|
#    | p | ¬p |    | p | q | p ∧ q | p → q |
#    |---+----|    |---+---+-------+-------|
#    | T | F  |    | T | T | T     | T     |
#    | F | T  |    | T | F | F     | F     |
#    |---+----|    | F | T | F     | T     |
#                  | F | F | F     | T     |
#                  |---+---+-------+-------|
#
# Definir la función
#    (i: Interpretacion, f: FProp) -> bool:
# tal que valor(i, p) es el valor de la fórmula p en la interpretación
# i. Por ejemplo,
#    >>> p = Impl(Var('A'), Conj(Var('A'), Var('B')))
#    >>> valor([('A',False),('B',False)], p)
#    True
#    >>> valor([('A',True),('B',False)], p)
#    False
# ---------------------------------------------------------------------
```

```python
from dataclasses import dataclass


@dataclass
class FProp:
    pass

@dataclass
class Const(FProp):
    x: bool

@dataclass
class Var(FProp):
    x: str

@dataclass
class Neg(FProp):
    x: FProp

@dataclass
class Conj(FProp):
    x: FProp
    y: FProp

@dataclass
class Impl(FProp):
    x: FProp
    y: FProp

Interpretacion = list[tuple[str, bool]]

# busca(c, t) es el valor del primer elemento de la lista de asociación
# t cuya clave es c. Por ejemplo,
#    >>> busca('B', [('A', True), ('B', False), ('C', True)])
#    False
def busca(c: str, i: Interpretacion) -> bool:
    return [v for (d, v) in i if d == c][0]

def valor(i: Interpretacion, f: FProp) -> bool:
    match f:
```

```python
        case Const(b):
            return b
        case Var(x):
            return busca(x, i)
        case Neg(p):
            return not valor(i, p)
        case Conj(p, q):
            return valor(i, p) and valor(i, q)
        case Impl(p, q):
            return valor(i, p) <= valor(i, q)
    assert False
```

5.8. El tipo de las fórmulas: Interpretaciones de una fórmula

En Haskell

```haskell
-- ---------------------------------------------------------------------
-- El tipo de las fórmulas proposicionales se puede definir por
--    data FProp = Const Bool
--               | Var Char
--               | Neg FProp
--               | Conj FProp FProp
--               | Impl FProp FProp
--      deriving Show
-- de modo que la fórmula A → ⊥ ∧ ¬B se representa por
--    Impl (Var 'A') (Conj (Const False) (Neg (Var 'B')))
--
-- Una interpretación de una fórmula es una función de sus variables en
-- los booleanos. Por ejemplo, la interpretación que a la variable A le
-- asigna verdadero y a la B falso se puede representar por
--    [('A', True), ('B', False)]
--
-- El tipo de las intepretaciones de puede definir por
--    type Interpretacion = [(Char, Bool)]
--
-- Definir la función
--    interpretaciones :: FProp -> [Interpretacion]
-- tal que (interpretaciones p) es la lista de las interpretaciones de
```

```haskell
-- la fórmula p. Por ejemplo,
--    λ> interpretaciones (Impl (Var 'A') (Conj (Var 'A') (Var 'B')))
--    [[('A',False),('B',False)],
--     [('A',False),('B',True)],
--     [('A',True),('B',False)],
--     [('A',True),('B',True)]]
-- ---------------------------------------------------------------------

module Interpretaciones_de_una_formula where

import Data.List (nub)

data FProp = Const Bool
           | Var Char
           | Neg FProp
           | Conj FProp FProp
           | Impl FProp FProp
  deriving Show

type Interpretacion = [(Char, Bool)]

interpretaciones :: FProp -> [Interpretacion]
interpretaciones p =
  [zip vs i | i <- interpretacionesVar (length vs)]
  where vs = nub (variables p)

-- (interpretacionesVar n) es la lista de las interpretaciones de n
-- variables. Por ejemplo,
--    λ> interpretacionesVar 2
--    [[False,False],
--     [False,True],
--     [True,False],
--     [True,True]]
interpretacionesVar :: Int -> [[Bool]]
interpretacionesVar 0 = [[]]
interpretacionesVar n = map (False:) bss ++ map (True:) bss
  where bss = interpretacionesVar (n-1)

-- (variables p) es la lista de las variables de la fórmula p. Por
-- ejemplo,
```

```
--    λ> variables (Impl (Var 'A') (Conj (Const False) (Neg (Var 'B'))))
--    "AB"
--    λ> variables (Impl (Var 'A') (Conj (Var 'A') (Neg (Var 'B'))))
--    "AAB"
variables :: FProp -> [Char]
variables (Const _)  = []
variables (Var x)    = [x]
variables (Neg p)    = variables p
variables (Conj p q) = variables p ++ variables q
variables (Impl p q) = variables p ++ variables q
```

En Python

```
# ---------------------------------------------------------------------
# El tipo de las fórmulas proposicionales se puede definir por
#    @dataclass
#    class FProp:
#        pass
#
#    @dataclass
#    class Const(FProp):
#        x: bool
#
#    @dataclass
#    class Var(FProp):
#        x: str
#
#    @dataclass
#    class Neg(FProp):
#        x: FProp
#
#    @dataclass
#    class Conj(FProp):
#        x: FProp
#        y: FProp
#
#    @dataclass
#    class Impl(FProp):
#        x: FProp
#        y: FProp
```

```
# de modo que la fórmula A → ⊥ ∧ ¬B se representa por
#    Impl(Var('A'), Conj(Const(False), Neg (Var('B'))))
#
# Una interpretación de una fórmula es una función de sus variables en
# los booleanos. Por ejemplo, la interpretación que a la variable A le
# asigna verdadero y a la B falso se puede representar por
#    [('A', True), ('B', False)]
#
# El tipo de las intepretaciones de puede definir por
#    Interpretacion = list[tuple[str, bool]]
#
# Definir la función
#    interpretaciones : (FProp) -> list[Interpretacion]
# tal que interpretaciones(p) es la lista de las interpretaciones de
# la fórmula p. Por ejemplo,
#    >>> interpretaciones(Impl(Var('A'), Conj(Var('A'), Var('B'))))
#    [[('B', False), ('A', False)],
#     [('B', False), ('A', True)],
#     [('B', True), ('A', False)],
#     [('B', True), ('A', True)]]
# ---------------------------------------------------------------------

from dataclasses import dataclass


@dataclass
class FProp:
    pass

@dataclass
class Const(FProp):
    x: bool

@dataclass
class Var(FProp):
    x: str

@dataclass
class Neg(FProp):
    x: FProp
```

```python
@dataclass
class Conj(FProp):
    x: FProp
    y: FProp

@dataclass
class Impl(FProp):
    x: FProp
    y: FProp

Interpretacion = list[tuple[str, bool]]

# variables(p) es la lista de las variables de la fórmula p. Por
# ejemplo,
#    >>> variables (Impl(Var('A'), Conj(Const(False), Neg (Var('B')))))
#    ['A', 'B']
#    >>> variables (Impl(Var('A'), Conj(Var('A'), Neg (Var('B')))))
#    ['A', 'A', 'B']
def variables(f: FProp) -> list[str]:
    match f:
        case Const(_):
            return []
        case Var(x):
            return [x]
        case Neg(p):
            return variables(p)
        case Conj(p, q):
            return variables(p) + variables(q)
        case Impl(p, q):
            return variables(p) + variables(q)
    assert False

# interpretacionesVar(n) es la lista de las interpretaciones de n
# variables. Por ejemplo,
#    >>> interpretacionesVar 2
#    [[False, False],
#     [False, True],
#     [True, False],
#     [True, True]]
```

```
def interpretacionesVar(n: int) -> list[list[bool]]:
    if n == 0:
        return [[]]
    bss = interpretacionesVar(n-1)
    return [[False] + x for x in bss] + [[True] + x for x in bss]

def interpretaciones(f: FProp) -> list[Interpretacion]:
    vs = list(set(variables(f)))
    return [list(zip(vs, i)) for i in interpretacionesVar(len(vs))]
```

5.9. El tipo de las fórmulas: Reconocedor de tautologías

En Haskell

```
-- ---------------------------------------------------------------------
-- El tipo de las fórmulas proposicionales se puede definir por
--    data FProp = Const Bool
--               | Var Char
--               | Neg FProp
--               | Conj FProp FProp
--               | Impl FProp FProp
--      deriving Show
-- de modo que la fórmula A → ⊥ ∧ ¬B se representa por
--    Impl (Var 'A') (Conj (Const False) (Neg (Var 'B')))
--
-- Una fórmula es una tautología si es verdadera en todas sus
-- interpretaciones. Por ejemplo,
-- + (A ∧ B) → A es una tautología
-- + A → (A ∧ B) no es una tautología
--
-- Definir la función
--    esTautologia :: FProp -> Bool
-- tal que (esTautologia p) se verifica si la fórmula p es una
-- tautología. Por ejemplo,
--    λ> esTautologia (Impl (Conj (Var 'A') (Var 'B')) (Var 'A'))
--    True
--    λ> esTautologia (Impl (Var 'A') (Conj (Var 'A') (Var 'B')))
--    False
```

```haskell
-- ---------------------------------------------------------------------

module Validez_de_una_formula where

import Data.List (nub)

data FProp = Const Bool
           | Var Char
           | Neg FProp
           | Conj FProp FProp
           | Impl FProp FProp
  deriving Show

type Interpretacion = [(Char, Bool)]

esTautologia :: FProp -> Bool
esTautologia p =
  and [valor i p | i <- interpretaciones p]

-- (valor i p) es el valor de la fórmula p en la interpretación i. Por
-- ejemplo,
--    λ> p = Impl (Var 'A') (Conj (Var 'A') (Var 'B'))
--    λ> valor [('A',False),('B',False)] p
--    True
--    λ> valor [('A',True),('B',False)] p
--    False
valor :: Interpretacion -> FProp -> Bool
valor _ (Const b)  = b
valor i (Var x)    = busca x i
valor i (Neg p)    = not (valor i p)
valor i (Conj p q) = valor i p && valor i q
valor i (Impl p q) = valor i p <= valor i q

-- (busca c t) es el valor del primer elemento de la lista de asociación
-- t cuya clave es c. Por ejemplo,
--    busca 2 [(1,'a'),(3,'d'),(2,'c')]  ==  'c'
busca :: Eq c => c -> [(c,v)] -> v
busca c t = head [v | (c',v) <- t, c == c']

interpretaciones :: FProp -> [Interpretacion]
```

```haskell
interpretaciones p =
  [zip vs i | i <- interpretacionesVar (length vs)]
  where vs = nub (variables p)

-- (interpretacionesVar n) es la lista de las interpretaciones de n
-- variables. Por ejemplo,
--    λ> interpretacionesVar 2
--    [[False,False],
--     [False,True],
--     [True,False],
--     [True,True]]
interpretacionesVar :: Int -> [[Bool]]
interpretacionesVar 0 = [[]]
interpretacionesVar n = map (False:) bss ++ map (True:) bss
  where bss = interpretacionesVar (n-1)

-- (variables p) es la lista de las variables de la fórmula p. Por
-- ejemplo,
--    λ> variables (Impl (Var 'A') (Conj (Const False) (Neg (Var 'B'))))
--    "AB"
--    λ> variables (Impl (Var 'A') (Conj (Var 'A') (Neg (Var 'B'))))
--    "AAB"
variables :: FProp -> [Char]
variables (Const _)  = []
variables (Var x)    = [x]
variables (Neg p)    = variables p
variables (Conj p q) = variables p ++ variables q
variables (Impl p q) = variables p ++ variables q
```

En Python

```python
# ---------------------------------------------------------------------
# El tipo de las fórmulas proposicionales se puede definir por
#    @dataclass
#    class FProp:
#        pass
#
#    @dataclass
#    class Const(FProp):
#        x: bool
```

```
#
#    @dataclass
#    class Var(FProp):
#        x: str
#
#    @dataclass
#    class Neg(FProp):
#        x: FProp
#
#    @dataclass
#    class Conj(FProp):
#        x: FProp
#        y: FProp
#
#    @dataclass
#    class Impl(FProp):
#        x: FProp
#        y: FProp
# de modo que la fórmula A → ⊥ ∧ ¬B se representa por
#    Impl(Var('A'), Conj(Const(False), Neg (Var('B'))))
#
# Una fórmula es una tautología si es verdadera en todas sus
# interpretaciones. Por ejemplo,
# + (A ∧ B) → A es una tautología
# + A → (A ∧ B) no es una tautología
#
# Definir la función
#    esTautologia :: FProp -> Bool
# tal que (esTautologia p) se verifica si la fórmula p es una
# tautología. Por ejemplo,
#    >>> esTautologia(Impl(Conj(Var('A'), Var ('B')), Var('A')))
#    True
#    >>> esTautologia(Impl(Var('A'), Conj(Var('A'), Var('B'))))
#    False
# ---------------------------------------------------------------------

from dataclasses import dataclass


@dataclass
```

```
class FProp:
    pass

@dataclass
class Const(FProp):
    x: bool

@dataclass
class Var(FProp):
    x: str

@dataclass
class Neg(FProp):
    x: FProp

@dataclass
class Conj(FProp):
    x: FProp
    y: FProp

@dataclass
class Impl(FProp):
    x: FProp
    y: FProp

Interpretacion = list[tuple[str, bool]]

# busca(c, t) es el valor del primer elemento de la lista de asociación
# t cuya clave es c. Por ejemplo,
#    >>> busca('B', [('A', True), ('B', False), ('C', True)])
#    False
def busca(c: str, i: Interpretacion) -> bool:
    return [v for (d, v) in i if d == c][0]

def valor(i: Interpretacion, f: FProp) -> bool:
    match f:
        case Const(b):
            return b
        case Var(x):
            return busca(x, i)
```

```python
        case Neg(p):
            return not valor(i, p)
        case Conj(p, q):
            return valor(i, p) and valor(i, q)
        case Impl(p, q):
            return valor(i, p) <= valor(i, q)
    assert False

# variables(p) es la lista de las variables de la fórmula p. Por
# ejemplo,
#    >>> variables (Impl(Var('A'), Conj(Const(False), Neg (Var('B')))))
#    ['A', 'B']
#    >>> variables (Impl(Var('A'), Conj(Var('A'), Neg (Var('B')))))
#    ['A', 'A', 'B']
def variables(f: FProp) -> list[str]:
    match f:
        case Const(_):
            return []
        case Var(x):
            return [x]
        case Neg(p):
            return variables(p)
        case Conj(p, q):
            return variables(p) + variables(q)
        case Impl(p, q):
            return variables(p) + variables(q)
    assert False

# interpretacionesVar(n) es la lista de las interpretaciones de n
# variables. Por ejemplo,
#    >>> interpretacionesVar 2
#    [[False, False],
#     [False, True],
#     [True, False],
#     [True, True]]
def interpretacionesVar(n: int) -> list[list[bool]]:
    if n == 0:
        return [[]]
    bss = interpretacionesVar(n-1)
    return [[False] + x for x in bss] + [[True] + x for x in bss]
```

```python
# interpretaciones(p) es la lista de las interpretaciones de la fórmula
# p. Por ejemplo,
#    >>> interpretaciones(Impl(Var('A'), Conj(Var('A'), Var('B'))))
#    [[('B', False), ('A', False)],
#     [('B', False), ('A', True)],
#     [('B', True), ('A', False)],
#     [('B', True), ('A', True)]]
def interpretaciones(f: FProp) -> list[Interpretacion]:
    vs = list(set(variables(f)))
    return [list(zip(vs, i)) for i in interpretacionesVar(len(vs))]

def esTautologia(p: FProp) -> bool:
    return all((valor(i, p) for i in interpretaciones(p)))
```

5.10. Altura de un árbol binario

En Haskell

```haskell
-- ---------------------------------------------------------------------
-- El árbol binario
--         ·
--        / \
--       /   \
--      ·     ·
--     / \   / \
--    1   4 6   9
-- se puede representar por
--    ejArbol = Nodo (Nodo (Hoja 1) (Hoja 4))
--                   (Nodo (Hoja 6) (Hoja 9))
--
-- El tipo de los árboles binarios se puede definir por
--    data Arbol a = Hoja a
--                 | Nodo (Arbol a) (Arbol a)
--
-- Definir la función
--    altura :: Arbol a -> Int
-- tal que (altura t) es la altura del árbol t. Por ejemplo,
--    λ> altura (Hoja 1)
--    0
```

```
--     λ> altura (Nodo (Hoja 1) (Hoja 6))
--     1
--     λ> altura (Nodo (Nodo (Hoja 1) (Hoja 6)) (Hoja 2))
--     2
--     λ> altura (Nodo (Nodo (Hoja 1) (Hoja 6)) (Nodo (Hoja 2) (Hoja 7)))
--     2
-- ---------------------------------------------------------------------

module Altura_de_un_arbol_binario where

data Arbol a = Hoja a
             | Nodo (Arbol a) (Arbol a)

altura :: Arbol a -> Int
altura (Hoja _)   = 0
altura (Nodo i d) = 1 + max (altura i) (altura d)
```

En Python

```
# ---------------------------------------------------------------------
# El árbol binario
#         .
#        / \
#       /   \
#      .     .
#     / \   / \
#    1   4 6   9
# se puede representar por
#    ejArbol = Nodo(Nodo(Hoja(1), Hoja(4)),
#                   Nodo(Hoja(6), Hoja(9)))
#
# El tipo de los árboles binarios se puede definir por
#    @dataclass
#    class Arbol(Generic[A]):
#        pass
#
#    @dataclass
#    class Hoja(Arbol[A]):
#        x: A
#
```

```
#    @dataclass
#    class Nodo(Arbol[A]):
#        i: Arbol
#        d: Arbol
#
# Definir la función
#    altura : (Arbol) -> int
# tal que altura(t) es la altura del árbol t. Por ejemplo,
#    >>> altura(Hoja(1))
#    0
#    >>> altura(Nodo(Hoja(1), Hoja(6)))
#    1
#    >>> altura(Nodo(Nodo(Hoja(1), Hoja(6)), Hoja(2)))
#    2
#    >>> altura(Nodo(Nodo(Hoja(1), Hoja(6)), Nodo(Hoja(2), Hoja(7))))
#    2
# ---------------------------------------------------------------------

from dataclasses import dataclass
from typing import Generic, TypeVar

A = TypeVar("A")

@dataclass
class Arbol(Generic[A]):
    pass

@dataclass
class Hoja(Arbol[A]):
    x: A

@dataclass
class Nodo(Arbol[A]):
    i: Arbol[A]
    d: Arbol[A]

def altura(a: Arbol[A]) -> int:
    match a:
        case Hoja(_):
            return 0
```

```
        case Nodo(i, d):
            return 1 + max(altura(i), altura(d))
    assert False
```

5.11. Aplicación de una función a un árbol

En Haskell

```
-- ---------------------------------------------------------------------
-- El árbol binario
--         ·
--        / \
--       /   \
--      ·     ·
--     / \   / \
--    1   4 6   9
-- se puede representar por
--    ejArbol = Nodo (Nodo (Hoja 1) (Hoja 4))
--                   (Nodo (Hoja 6) (Hoja 9))
--
-- El tipo de los árboles binarios se puede definir por
--    data Arbol a = Hoja a
--                 | Nodo (Arbol a) (Arbol a)
--      deriving (Show, Eq)
--
-- Definir la función
--    mapArbol :: (a -> b) -> Arbol a -> Arbol b
-- tal que (mapArbol f t) es el árbol obtenido aplicando la función f a
-- los elementos del árbol t. Por ejemplo,
--    λ> mapArbol (+ 1) (Nodo (Hoja 2) (Hoja 4))
--    Nodo (Hoja 3) (Hoja 5)
-- ---------------------------------------------------------------------

module Aplicacion_de_una_funcion_a_un_arbol where

data Arbol a = Hoja a
             | Nodo (Arbol a) (Arbol a)
  deriving (Show, Eq)

mapArbol :: (a -> b) -> Arbol a -> Arbol b
```

```
mapArbol f (Hoja a)   = Hoja (f a)
mapArbol f (Nodo l r) = Nodo (mapArbol f l) (mapArbol f r)
```

En Python

```
# ---------------------------------------------------------------------
# El árbol binario
#         ·
#        / \
#       /   \
#      ·     ·
#     / \   / \
#    1   4 6   9
# se puede representar por
#    ejArbol = Nodo(Nodo(Hoja(1), Hoja(4)),
#                   Nodo(Hoja(6), Hoja(9)))
#
#
# El tipo de los árboles binarios se puede definir por
#    @dataclass
#    class Arbol(Generic[A]):
#        pass
#
#    @dataclass
#    class Hoja(Arbol[A]):
#        x: A
#
#    @dataclass
#    class Nodo(Arbol[A]):
#        i: Arbol
#        d: Arbol
#
# Definir la función
#    mapArbol : (Callable[[A], B], Arbol[A]) -> Arbol[B]
# tal que mapArbol(f, t) es el árbolo obtenido aplicando la función f a
# los elementos del árbol t. Por ejemplo,
#    >>> mapArbol(lambda x: 1 + x, Nodo(Hoja(2), Hoja(4)))
#    Nodo(i=Hoja(x=3), d=Hoja(x=5))
# ---------------------------------------------------------------------
```

```python
from dataclasses import dataclass
from typing import Callable, Generic, TypeVar

A = TypeVar("A")
B = TypeVar("B")

@dataclass
class Arbol(Generic[A]):
    pass

@dataclass
class Hoja(Arbol[A]):
    x: A

@dataclass
class Nodo(Arbol[A]):
    i: Arbol[A]
    d: Arbol[A]

def mapArbol(f: Callable[[A], B], a: Arbol[A]) -> Arbol[B]:
    match a:
        case Hoja(x):
            return Hoja(f(x))
        case Nodo(i, d):
            return Nodo(mapArbol(f, i), mapArbol(f, d))
    assert False
```

5.12. Árboles con la misma forma

En Haskell

```haskell
-- ----------------------------------------------------------------------
-- El árbol binario
--         .
--        / \
--       /   \
--      .     .
--     / \   / \
--    1   4 6   9
-- se puede representar por
```

```haskell
--    ejArbol = Nodo (Nodo (Hoja 1) (Hoja 4))
--                   (Nodo (Hoja 6) (Hoja 9))
--
-- El tipo de los árboles binarios se puede definir por
--    data Arbol a = Hoja a
--                 | Nodo (Arbol a) (Arbol a)
--      deriving (Show, Eq)
--
-- Definir la función
--    mismaForma :: Arbol a -> Arbol b -> Bool
-- tal que (mismaForma t1 t2) se verifica si t1 y t2 tienen la misma
-- estructura. Por ejemplo,
--    λ> arbol1 = Hoja 5
--    λ> arbol2 = Hoja 3
--    λ> mismaForma arbol1 arbol2
--    True
--    λ> arbol3 = Nodo (Hoja 6) (Hoja 7)
--    λ> mismaForma arbol1 arbol3
--    False
--    λ> arbol4 = Nodo (Hoja 9) (Hoja 5)
--    λ> mismaForma arbol3 arbol4
--    True
-- ---------------------------------------------------------------------

module Arboles_con_la_misma_forma where

import Test.QuickCheck

data Arbol a = Hoja a
             | Nodo (Arbol a) (Arbol a)
  deriving (Show, Eq)

-- 1ª solución
-- ===========

mismaForma1 :: Arbol a -> Arbol b -> Bool
mismaForma1 (Hoja _)   (Hoja _)     = True
mismaForma1 (Nodo l r) (Nodo l' r') = mismaForma1 l l' && mismaForma1 r r'
mismaForma1 _          _            = False
```

```
-- 2ª solución
-- ===========

mismaForma2 :: Arbol a -> Arbol b -> Bool
mismaForma2 x y = f x == f y
  where
    f = mapArbol (const ())

-- (mapArbol f t) es el árbol obtenido aplicando la función f a los
-- elementos del árbol t. Por ejemplo,
--    λ> mapArbol (+ 1) (Nodo (Hoja 2) (Hoja 4))
--    Nodo (Hoja 3) (Hoja 5)
mapArbol :: (a -> b) -> Arbol a -> Arbol b
mapArbol f (Hoja a)   = Hoja (f a)
mapArbol f (Nodo i d) = Nodo (mapArbol f i) (mapArbol f d)

-- Comprobación de equivalencia
-- ============================

-- (arbolArbitrario n) es un árbol aleatorio de altura n. Por ejemplo,
--    λ> sample (arbolArbitrario 3 :: Gen (Arbol Int))
--    Nodo (Nodo (Nodo (Hoja 0) (Hoja 0)) (Hoja 0)) (Hoja 0)
--    Nodo (Nodo (Hoja 4) (Hoja 8)) (Hoja (-4))
--    Nodo (Nodo (Nodo (Hoja 4) (Hoja 10)) (Hoja (-6))) (Hoja (-1))
--    Nodo (Nodo (Hoja 3) (Hoja 6)) (Hoja (-5))
--    Nodo (Nodo (Hoja (-11)) (Hoja (-13))) (Hoja 14)
--    Nodo (Nodo (Hoja (-7)) (Hoja 15)) (Hoja (-2))
--    Nodo (Nodo (Hoja (-9)) (Hoja (-2))) (Hoja (-6))
--    Nodo (Nodo (Hoja (-15)) (Hoja (-16))) (Hoja (-20))
arbolArbitrario :: Arbitrary a => Int -> Gen (Arbol a)
arbolArbitrario n
  | n <= 1    = Hoja <$> arbitrary
  | otherwise = do
      k <- choose (2, n - 1)
      Nodo <$> arbolArbitrario k <*> arbolArbitrario (n - k)

-- Arbol es subclase de Arbitraria
instance Arbitrary a => Arbitrary (Arbol a) where
  arbitrary = sized arbolArbitrario
  shrink (Hoja x)   = Hoja <$> shrink x
```

```
  shrink (Nodo l r) = l :
                      r :
                      [Nodo l' r | l' <- shrink l] ++
                      [Nodo l r' | r' <- shrink r]

-- La propiedad es
prop_mismaForma :: Arbol Int -> Arbol Int -> Property
prop_mismaForma a1 a2 =
  mismaForma1 a1 a2 === mismaForma2 a1 a2

-- La comprobación es
--    λ> quickCheck prop_mismaForma
--    +++ OK, passed 100 tests.
```

En Python

```
# ---------------------------------------------------------------------
# El árbol binario
#         ·
#        / \
#       /   \
#      ·     ·
#     / \   / \
#    1   4 6   9
# se puede representar por
#    ejArbol = Nodo(Nodo(Hoja(1), Hoja(4)),
#                   Nodo(Hoja(6), Hoja(9)))
#
#
# El tipo de los árboles binarios se puede definir por
#    @dataclass
#    class Arbol(Generic[A]):
#        pass
#
#    @dataclass
#    class Hoja(Arbol[A]):
#        x: A
#
#    @dataclass
#    class Nodo(Arbol[A]):
```

```
#       i: Arbol
#       d: Arbol
#
# Definir la función
#    mismaForma : (Arbol[A], Arbol[B]) -> bool
# tal que mismaForma(t1, t2) se verifica si t1 y t2 tienen la misma
# estructura. Por ejemplo,
#    >>> arbol1 = Hoja(5)
#    >>> arbol2 = Hoja(3)
#    >>> mismaForma(arbol1, arbol2)
#    True
#    >>> arbol3 = Nodo(Hoja(6), Hoja(7))
#    >>> mismaForma(arbol1, arbol3)
#    False
#    >>> arbol4 = Nodo(Hoja(9), Hoja(5))
#    >>> mismaForma(arbol3, arbol4)
#    True
# ---------------------------------------------------------------------

from dataclasses import dataclass
from random import randint
from typing import Callable, Generic, TypeVar

from hypothesis import given
from hypothesis import strategies as st

A = TypeVar("A")
B = TypeVar("B")

@dataclass
class Arbol(Generic[A]):
    pass

@dataclass
class Hoja(Arbol[A]):
    x: A

@dataclass
class Nodo(Arbol[A]):
    i: Arbol[A]
```

```
    d: Arbol[A]

# -- 1ª solución
# -- ===========

def mismaForma1(a: Arbol[A], b: Arbol[B]) -> bool:
    match (a, b):
        case (Hoja(_), Hoja(_)):
            return True
        case (Nodo(i1, d1), Nodo(i2, d2)):
            return mismaForma1(i1, i2) and mismaForma1(d1, d2)
        case (_, _):
            return False
    assert False

# -- 2ª solución
# -- ===========

# mapArbol(f, t) es el árbolo obtenido aplicando la función f a
# los elementos del árbol t. Por ejemplo,
#    >>> mapArbol(lambda x: 1 + x, Nodo(Hoja(2), Hoja(4)))
#    Nodo(i=Hoja(x=3), d=Hoja(x=5))
def mapArbol(f: Callable[[A], B], a: Arbol[A]) -> Arbol[B]:
    match a:
        case Hoja(x):
            return Hoja(f(x))
        case Nodo(i, d):
            return Nodo(mapArbol(f, i), mapArbol(f, d))
    assert False

def mismaForma2(a: Arbol[A], b: Arbol[B]) -> bool:
    return mapArbol(lambda x: 0, a) == mapArbol(lambda x: 0, b)

# Comprobación de equivalencia
# ============================

# arbolArbitrario(n) es un árbol aleatorio de orden n. Por ejemplo,
#    >>> arbolArbitrario(3)
#    Nodo(i=Hoja(x=2), d=Nodo(i=Hoja(x=5), d=Hoja(x=2)))
#    >>> arbolArbitrario(3)
```

```
#    Nodo(i=Nodo(i=Hoja(x=6), d=Hoja(x=9)), d=Hoja(x=1))
def arbolArbitrario(n: int) -> Arbol[int]:
    if n == 0:
        return Hoja(randint(1, 10 * n))
    k = min(randint(1, n), n - 1)
    return Nodo(arbolArbitrario(k), arbolArbitrario(n - k))

# La propiedad es
@given(st.integers(min_value=1, max_value=10),
       st.integers(min_value=1, max_value=10))
def test_mismaForma(n1: int, n2: int) -> None:
    a1 = arbolArbitrario(n1)
    a2 = arbolArbitrario(n2)
    assert mismaForma1(a1, a2) == mismaForma2(a1, a2)

# La comprobación es
#    src> poetry run pytest -q arboles_con_la_misma_forma.py
#    1 passed in 0.22s
```

5.13. Árbol con las hojas en la profundidad dada

En Haskell

```
-- ----------------------------------------------------------------------
-- El árbol binario
--         .
--        / \
--       /   \
--      .     .
--     / \   / \
--    1   4 6   9
-- se puede representar por
--    ejArbol = Nodo (Nodo (Hoja 1) (Hoja 4))
--                   (Nodo (Hoja 6) (Hoja 9))
--
-- El tipo de los árboles binarios se puede definir por
--    data Arbol a = Hoja a
--                 | Nodo (Arbol a) (Arbol a)
```

```
--      deriving (Show, Eq)
--
-- Definir la función
--    creaArbol :: Int -> Arbol ()
-- tal que (creaArbol n) es el árbol cuyas hoyas están en la profundidad
-- n. Por ejemplo,
--    λ> creaArbol 2
--    Nodo (Nodo (Hoja ()) (Hoja ())) (Nodo (Hoja ()) (Hoja ()))
-- ---------------------------------------------------------------------

module Arbol_con_las_hojas_en_la_profundidad_dada where

data Arbol a = Hoja a
             | Nodo (Arbol a) (Arbol a)
  deriving (Show, Eq)

creaArbol :: Int -> Arbol ()
creaArbol h
  | h <= 0    = Hoja ()
  | otherwise = Nodo x x
  where x = creaArbol (h - 1)
```

En Python

```
# ---------------------------------------------------------------------
# El árbol binario
#         ·
#        / \
#       /   \
#      ·     ·
#     / \   / \
#    1   4 6   9
# se puede representar por
#    ejArbol = Nodo(Nodo(Hoja(1), Hoja(4)),
#                   Nodo(Hoja(6), Hoja(9)))
#
#
# El tipo de los árboles binarios se puede definir por
#    @dataclass
#    class Arbol(Generic[A]):
```

```
#        pass
#
#    @dataclass
#    class Hoja(Arbol[A]):
#        x: A
#
#    @dataclass
#    class Nodo(Arbol[A]):
#        i: Arbol
#        d: Arbol
#
# Definir la función
#    creaArbol : (int) -> Arbol[Any]:
# tal que creaArbol(n) es el árbol cuyas hoyas están en la profundidad
# n. Por ejemplo,
#    >>> creaArbol(2)
#    Nodo(Nodo(Hoja(None), Hoja(None)), Nodo(Hoja(None), Hoja(None)))
# ---------------------------------------------------------------------

from dataclasses import dataclass
from typing import Any, Generic, TypeVar

A = TypeVar("A")

@dataclass
class Arbol(Generic[A]):
    pass

@dataclass
class Hoja(Arbol[A]):
    x: A

@dataclass
class Nodo(Arbol[A]):
    i: Arbol[A]
    d: Arbol[A]

def creaArbol(h: int) -> Arbol[Any]:
    if h <= 0:
        return Hoja(None)
```

```
    x = creaArbol(h - 1)
    return Nodo(x, x)
```

5.14. El tipo de las expresiones aritméticas: Valor de una expresión

En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Se considera el tipo de las expresiones aritméticas definido por
--    data Expr =
--        Lit Int
--      | Suma Expr Expr
--      | Op Expr
--      | SiCero Expr Expr Expr
--      deriving (Eq, Show)
-- formado por
-- + literales (p.e. Lit 7),
-- + sumas (p.e. Suma (Lit 7) (Suma (Lit 3) (Lit 5)))
-- + opuestos (p.e. Op (Suma (Op (Lit 7)) (Suma (Lit 3) (Lit 5))))
-- + expresiones condicionales (p.e. (SiCero (Lit 3) (Lit 4) (Lit 5))
--
-- Definir la función
--    valor :: Expr -> Int
-- tal que (valor e) es el valor de la expresión e (donde el valor de
-- (SiCero e e1 e2) es el valor de e1 si el valor de e es cero y el es
-- el valor de e2, en caso contrario). Por ejemplo,
--    valor (Op (Suma (Lit 3) (Lit 5)))      ==  -8
--    valor (SiCero (Lit 0) (Lit 4) (Lit 5)) == 4
--    valor (SiCero (Lit 1) (Lit 4) (Lit 5)) == 5
-- ---------------------------------------------------------------------

module Valor_de_una_expresion_aritmetica where

data Expr =
    Lit Int
  | Suma Expr Expr
  | Op Expr
  | SiCero Expr Expr Expr
```

```haskell
  deriving (Eq, Show)

valor :: Expr -> Int
valor (Lit n)          = n
valor (Suma x y)       = valor x + valor y
valor (Op x)           = - valor x
valor (SiCero x y z) | valor x == 0 = valor y
                     | otherwise    = valor z
```

En Python

```python
# ---------------------------------------------------------------------
# Se considera el tipo de las expresiones aritméticas definido por
#    @dataclass
#    class Expr:
#        pass
#
#    @dataclass
#    class Lit(Expr):
#        x: int
#
#    @dataclass
#    class Suma(Expr):
#        x: Expr
#        y: Expr
#
#    @dataclass
#    class Op(Expr):
#        x: Expr
#
#    @dataclass
#    class SiCero(Expr):
#        x: Expr
#        y: Expr
#        z: Expr
#
# formado por
# + literales (p.e. Lit 7),
# + sumas (p.e. Suma (Lit 7) (Suma (Lit 3) (Lit 5)))
# + opuestos (p.e. Op (Suma (Op (Lit 7)) (Suma (Lit 3) (Lit 5))))
```

```python
# + expresiones condicionales (p.e. (SiCero (Lit 3) (Lit 4) (Lit 5))
#
# Definir la función
#    valor : (Expr) -> int
# tal que valor(e) es el valor de la expresión e (donde el valor de
# (SiCero e e1 e2) es el valor de e1 si el valor de e es cero y el es
# el valor de e2, en caso contrario). Por ejemplo,
#    valor(Op(Suma(Lit(3), Lit(5))))       == -8
#    valor(SiCero(Lit(0), Lit(4), Lit(5))) == 4
#    valor(SiCero(Lit(1), Lit(4), Lit(5))) == 5
# ---------------------------------------------------------------------

from dataclasses import dataclass


@dataclass
class Expr:
    pass

@dataclass
class Lit(Expr):
    x: int

@dataclass
class Suma(Expr):
    x: Expr
    y: Expr

@dataclass
class Op(Expr):
    x: Expr

@dataclass
class SiCero(Expr):
    x: Expr
    y: Expr
    z: Expr

def valor(e: Expr) -> int:
    match e:
```

```
        case Lit(n):
            return n
        case Suma(x, y):
            return valor(x) + valor(y)
        case Op(x):
            return -valor(x)
        case SiCero(x, y, z):
            return valor(y) if valor(x) == 0 else valor(z)
    assert False

# 2ª solución
# ===========

def valor2(e: Expr) -> int:
    if isinstance(e, Lit):
        return e.x
    if isinstance(e, Suma):
        return valor2(e.x) + valor2(e.y)
    if isinstance(e, Op):
        return -valor2(e.x)
    if isinstance(e, SiCero):
        if valor2(e.x) == 0:
            return valor2(e.y)
        return valor2(e.z)
    assert False
```

5.15. El tipo de las expresiones aritméticas: Valor de la resta

En Haskell

```
-- ---------------------------------------------------------------------
-- Se considera el tipo de las expresiones aritméticas definido por
--    data Expr = Lit Int
--              | Suma Expr Expr
--              | Op Expr
--              | SiCero Expr Expr Expr
--      deriving (Eq, Show)
-- formado por
```

```haskell
-- + literales (p.e. Lit 7),
-- + sumas (p.e. Suma (Lit 7) (Suma (Lit 3) (Lit 5)))
-- + opuestos (p.e. Op (Suma (Op (Lit 7)) (Suma (Lit 3) (Lit 5))))
-- + expresiones condicionales (p.e. (SiCero (Lit 3) (Lit 4) (Lit 5))
--
-- La función para calcular el valor de una expresión es
--    valor :: Expr -> Int
--    valor (Lit n)        = n
--    valor (Suma x y)     = valor x + valor y
--    valor (Op x)         = - valor x
--    valor (SiCero x y z) | valor x == 0 = valor y
--                         | otherwise    = valor z
--
-- Definir la función
--    resta :: Expr -> Expr -> Expr
-- tal que (resta e1 e2) es la expresión correspondiente a la diferencia
-- de e1 y e2. Por ejemplo,
--    resta (Lit 42) (Lit 2)  ==  Suma (Lit 42) (Op (Lit 2))
--
-- Comprobar con QuickCheck que
--    valor (resta x y) == valor x - valor y
-- ---------------------------------------------------------------------

module Valor_de_la_resta where

import Test.QuickCheck

data Expr = Lit Int
          | Suma Expr Expr
          | Op Expr
          | SiCero Expr Expr Expr
  deriving (Eq, Show)

valor :: Expr -> Int
valor (Lit n)        = n
valor (Suma x y)     = valor x + valor y
valor (Op x)         = - valor x
valor (SiCero x y z) | valor x == 0 = valor y
                     | otherwise    = valor z
```

```
resta :: Expr -> Expr -> Expr
resta x y = Suma x (Op y)

-- Comprobación de la propiedad
-- ============================

-- (exprArbitraria n) es una expresión aleatoria de tamaño n. Por
-- ejemplo,
--    λ> sample (exprArbitraria 3)
--    Op (Op (Lit 0))
--    SiCero (Lit 0) (Lit (-2)) (Lit (-1))
--    Op (Suma (Lit 3) (Lit 0))
--    Op (Lit 5)
--    Op (Lit (-1))
--    Op (Op (Lit 9))
--    Suma (Lit (-12)) (Lit (-12))
--    Suma (Lit (-9)) (Lit 10)
--    Op (Suma (Lit 8) (Lit 15))
--    SiCero (Lit 16) (Lit 9) (Lit (-5))
--    Suma (Lit (-3)) (Lit 1)
exprArbitraria :: Int -> Gen Expr
exprArbitraria n
  | n <= 1 = Lit <$> arbitrary
  | otherwise = oneof
                [ Lit <$> arbitrary
                , let m = div n 2
                  in Suma <$> exprArbitraria m <*> exprArbitraria m
                , Op <$> exprArbitraria (n - 1)
                , let m = div n 3
                  in SiCero <$> exprArbitraria m
                            <*> exprArbitraria m
                            <*> exprArbitraria m ]

-- Expr es subclase de Arbitrary
instance Arbitrary Expr where
  arbitrary = sized exprArbitraria


-- La propiedad es
prop_resta :: Expr -> Expr -> Property
```

```
prop_resta x y =
  valor (resta x y) === valor x - valor y

-- La comprobación es
--    λ> quickCheck prop_resta
--    +++ OK, passed 100 tests.
```

En Python

```
# ---------------------------------------------------------------------
# Se considera el tipo de las expresiones aritméticas definido por
#    @dataclass
#    class Expr:
#        pass
#
#    @dataclass
#    class Lit(Expr):
#        x: int
#
#    @dataclass
#    class Suma(Expr):
#        x: Expr
#        y: Expr
#
#    @dataclass
#    class Op(Expr):
#        x: Expr
#
#    @dataclass
#    class SiCero(Expr):
#        x: Expr
#        y: Expr
#        z: Expr
#
# formado por
# + literales (p.e. Lit 7),
# + sumas (p.e. Suma (Lit 7) (Suma (Lit 3) (Lit 5)))
# + opuestos (p.e. Op (Suma (Op (Lit 7)) (Suma (Lit 3) (Lit 5))))
# + expresiones condicionales (p.e. (SiCero (Lit 3) (Lit 4) (Lit 5))
#
```

```python
# La función para calcular el valor de una expresión es
#    def valor(e: Expr) -> int:
#        match e:
#            case Lit(n):
#                return n
#            case Suma(x, y):
#                return valor(x) + valor(y)
#            case Op(x):
#                return -valor(x)
#            case SiCero(x, y, z):
#                return valor(y) if valor(x) == 0 else valor(z)
#        assert False
#
# Definir la función
#    resta : (Expr, Expr) -> Expr
# tal que resta(e1, e2) es la expresión correspondiente a la diferencia
# de e1 y e2. Por ejemplo,
#    resta(Lit(42), Lit(2))  ==  Suma(Lit(42), Op(Lit(2)))
#
# Comprobar con Hypothesis que
#    valor(resta(x, y)) == valor(x) - valor(y)
# ---------------------------------------------------------------------

from dataclasses import dataclass
from random import choice, randint

from hypothesis import given
from hypothesis import strategies as st


@dataclass
class Expr:
    pass

@dataclass
class Lit(Expr):
    x: int

@dataclass
class Suma(Expr):
```

```
    x: Expr
    y: Expr

@dataclass
class Op(Expr):
    x: Expr

@dataclass
class SiCero(Expr):
    x: Expr
    y: Expr
    z: Expr

def valor(e: Expr) -> int:
    match e:
        case Lit(n):
            return n
        case Suma(x, y):
            return valor(x) + valor(y)
        case Op(x):
            return -valor(x)
        case SiCero(x, y, z):
            return valor(y) if valor(x) == 0 else valor(z)
    assert False

def resta(x: Expr, y: Expr) -> Expr:
    return Suma(x, Op(y))

# -- Comprobación de la propiedad
# -- ============================

# exprArbitraria(n) es una expresión aleatoria de tamaño n. Por
# ejemplo,
#    >>> exprArbitraria(3)
#    Op(x=Op(x=Lit(x=9)))
#    >>> exprArbitraria(3)
#    Op(x=SiCero(x=Lit(x=6), y=Lit(x=2), z=Lit(x=6)))
#    >>> exprArbitraria(3)
#    Suma(x=Lit(x=8), y=Lit(x=2))
def exprArbitraria(n: int) -> Expr:
```

```
    if n <= 1:
        return Lit(randint(0, 10))
    m = n // 2
    return choice([Lit(randint(0, 10)),
                   Suma(exprArbitraria(m), exprArbitraria(m)),
                   Op(exprArbitraria(n - 1)),
                   SiCero(exprArbitraria(m),
                          exprArbitraria(m),
                          exprArbitraria(m))])

# La propiedad es
@given(st.integers(min_value=1, max_value=10),
       st.integers(min_value=1, max_value=10))
def test_mismaForma(n1: int, n2: int) -> None:
    x = exprArbitraria(n1)
    y = exprArbitraria(n2)
    assert valor(resta(x, y)) == valor(x) - valor(y)

# La comprobación es
#    src> poetry run pytest -q valor_de_la_resta.py
#    1 passed in 0.21s
```

5.16. Número de hojas de un árbol binario

En Haskell

```
-- ---------------------------------------------------------------------
-- El árbol binario
--        9
--       / \
--      /   \
--     3     7
--    / \
--   2   4
-- se puede representar por
--    N 9 (N 3 (H 2) (H 4)) (H 7)
--
-- El tipo de los árboles binarios se puede definir por
--    data Arbol a = H a
--                 | N a (Arbol a) (Arbol a)
```

```haskell
--      deriving (Show, Eq)
--
-- Definir las funciones
--    nHojas :: Arbol a -> Int
--    nNodos :: Arbol a -> Int
-- tales que
-- + (nHojas x) es el número de hojas del árbol x. Por ejemplo,
--      nHojas (N 9 (N 3 (H 2) (H 4)) (H 7))  ==  3
-- + (nNodos x) es el número de nodos del árbol x. Por ejemplo,
--      nNodos (N 9 (N 3 (H 2) (H 4)) (H 7))  ==  2
--
-- Comprobar con QuickCheck que en todo árbol binario el número de sus
-- hojas es igual al número de sus nodos más uno.
-- ---------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module Numero_de_hojas_de_un_arbol_binario where

import Test.QuickCheck

data Arbol a = H a
             | N a (Arbol a) (Arbol a)
  deriving (Show, Eq)

nHojas :: Arbol a -> Int
nHojas (H _)     = 1
nHojas (N _ i d) = nHojas i + nHojas d

nNodos :: Arbol a -> Int
nNodos (H _)     = 0
nNodos (N _ i d) = 1 + nNodos i + nNodos d

-- Comprobación de equivalencia
-- ============================

-- (arbolArbitrario n) es un árbol aleatorio de altura n. Por ejemplo,
--    λ> sample (arbolArbitrario 3 :: Gen (Arbol Int))
--    N 0 (H 0) (H 0)
--    N 1 (N (-2) (H (-1)) (H 1)) (H 2)
```

```haskell
--    N 3 (H 1) (H 2)
--    N 6 (N 0 (H 5) (H (-5))) (N (-5) (H (-5)) (H 4))
--    H 7
--    N (-8) (H (-8)) (H 9)
--    H 2
--    N (-1) (H 7) (N 9 (H (-2)) (H (-8)))
--    H (-3)
--    N 0 (N 16 (H (-14)) (H (-18))) (H 7)
--    N (-16) (H 18) (N (-19) (H (-15)) (H (-18)))
arbolArbitrario :: Arbitrary a => Int -> Gen (Arbol a)
arbolArbitrario 0 = H <$> arbitrary
arbolArbitrario n =
  oneof [H <$> arbitrary,
         N <$> arbitrary <*> arbolArbitrario (div n 2) <*> arbolArbitrario (div n

-- Arbol es subclase de Arbitrary
instance Arbitrary a => Arbitrary (Arbol a) where
  arbitrary = sized arbolArbitrario

-- La propiedad es
prop_nHojas :: Arbol Int -> Bool
prop_nHojas x =
  nHojas x == nNodos x + 1

-- La comprobación es
--    λ> quickCheck prop_nHojas
--    OK, passed 100 tests.
```

En Python

```python
# ---------------------------------------------------------------------
# El árbol binario
#         9
#        / \
#       /   \
#      3     7
#     / \
#    2   4
# se puede representar por
#    N(9, N(3, H(2), H(4)), H(7))
```

```
#
# El tipo de los árboles binarios se puede definir por
#    @dataclass
#    class Arbol(Generic[A]):
#        pass
#
#    @dataclass
#    class H(Arbol[A]):
#        x: A
#
#    @dataclass
#    class N(Arbol[A]):
#        x: A
#        i: Arbol[A]
#        d: Arbol[A]
#
# Definir las funciones
#    nHojas : (Arbol[A]) -> int
#    nNodos : (Arbol[A]) -> int
# tales que
# + nHojas(x) es el número de hojas del árbol x. Por ejemplo,
#      nHojas(N(9, N(3, H(2), H(4)), H(7)))  ==  3
# + nNodos(x) es el número de nodos del árbol x. Por ejemplo,
#      nNodos(N(9, N(3, H(2), H(4)), H(7)))  ==  2
#
# Comprobar con Hypothesis que en todo árbol binario el número de sus
# hojas es igual al número de sus nodos más uno.
# ---------------------------------------------------------------------

from dataclasses import dataclass
from random import choice, randint
from typing import Generic, TypeVar

from hypothesis import given
from hypothesis import strategies as st

A = TypeVar("A")

@dataclass
class Arbol(Generic[A]):
```

```
    pass

@dataclass
class H(Arbol[A]):
    x: A

@dataclass
class N(Arbol[A]):
    x: A
    i: Arbol[A]
    d: Arbol[A]

def nHojas(a: Arbol[A]) -> int:
    match a:
        case H(_):
            return 1
        case N(_, i, d):
            return nHojas(i) + nHojas(d)
    assert False

def nNodos(a: Arbol[A]) -> int:
    match a:
        case H(_):
            return 0
        case N(_, i, d):
            return 1 + nNodos(i) + nNodos(d)
    assert False

# Comprobación de equivalencia
# ============================

# (arbolArbitrario n) es un árbol aleatorio de orden n. Por ejemplo,
#    >>> arbolArbitrario(4)
#    N(x=2, i=H(x=1), d=H(x=9))
#    >>> arbolArbitrario(4)
#    H(x=10)
#    >>> arbolArbitrario(4)
#    N(x=4, i=N(x=7, i=H(x=4), d=H(x=0)), d=H(x=6))
def arbolArbitrario(n: int) -> Arbol[int]:
    if n <= 1:
```

```
        return H(randint(0, 10))
    m = n // 2
    return choice([H(randint(0, 10)),
                   N(randint(0, 10),
                     arbolArbitrario(m),
                     arbolArbitrario(m))])

# La propiedad es
@given(st.integers(min_value=1, max_value=10))
def test_nHojas(n: int) -> None:
    a = arbolArbitrario(n)
    assert nHojas(a) == nNodos(a) + 1

# La comprobación es
#    src> poetry run pytest -q numero_de_hojas_de_un_arbol_binario.py
#    1 passed in 0.10s
```

5.17. Profundidad de un árbol binario

5.17.1. En Haskell

```
-- ---------------------------------------------------------------------
-- El árbol binario
--        9
--       / \
--      /   \
--     3     7
--    / \
--   2   4
-- se puede representar por
--   N 9 (N 3 (H 2) (H 4)) (H 7)
--
-- El tipo de los árboles binarios se puede definir por
--   data Arbol a = H a
--                | N a (Arbol a) (Arbol a)
--     deriving (Show, Eq)
--
-- Definir la función
--   profundidad :: Arbol a -> Int
-- tal que (profundidad x) es la profundidad del árbol x. Por ejemplo,
```

```
--    profundidad (N 9 (N 3 (H 2) (H 4)) (H 7))              ==  2
--    profundidad (N 9 (N 3 (H 2) (N 1 (H 4) (H 5))) (H 7))  ==  3
--    profundidad (N 4 (N 5 (H 4) (H 2)) (N 3 (H 7) (H 4)))  ==  2
--
-- Comprobar con QuickCheck que para todo árbol biario x, se tiene que
--    nNodos x <= 2^(profundidad x) - 1
-- ---------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module Profundidad_de_un_arbol_binario where

import Test.QuickCheck

data Arbol a = H a
             | N a (Arbol a) (Arbol a)
  deriving (Show, Eq)

profundidad :: Arbol a -> Int
profundidad (H _)     = 0
profundidad (N _ i d) = 1 + max (profundidad i) (profundidad d)

-- Comprobación de equivalencia
-- ============================

-- (arbolArbitrario n) es un árbol aleatorio de altura n. Por ejemplo,
--    λ> sample (arbolArbitrario 3 :: Gen (Arbol Int))
--    N 0 (H 0) (H 0)
--    N 1 (N (-2) (H (-1)) (H 1)) (H 2)
--    N 3 (H 1) (H 2)
--    N 6 (N 0 (H 5) (H (-5))) (N (-5) (H (-5)) (H 4))
--    H 7
--    N (-8) (H (-8)) (H 9)
--    H 2
--    N (-1) (H 7) (N 9 (H (-2)) (H (-8)))
--    H (-3)
--    N 0 (N 16 (H (-14)) (H (-18))) (H 7)
--    N (-16) (H 18) (N (-19) (H (-15)) (H (-18)))
arbolArbitrario :: Arbitrary a => Int -> Gen (Arbol a)
arbolArbitrario 0 = H <$> arbitrary
```

```
arbolArbitrario n =
  oneof [H <$> arbitrary,
         N <$> arbitrary <*> arbolArbitrario (div n 2) <*> arbolArbitrario (div n

-- Arbol es subclase de Arbitrary
instance Arbitrary a => Arbitrary (Arbol a) where
  arbitrary = sized arbolArbitrario

-- La propiedad es
prop_nNodosProfundidad :: Arbol Int -> Bool
prop_nNodosProfundidad x =
   nNodos x <= 2 ^ profundidad x - 1

-- (nNodos x) es el número de nodos del árbol x. Por ejemplo,
--      nNodos (N 9 (N 3 (H 2) (H 4)) (H 7))  ==  2
nNodos :: Arbol a -> Int
nNodos (H _)     = 0
nNodos (N _ i d) = 1 + nNodos i + nNodos d

-- La comprobación es
--    λ> quickCheck prop_nNodosProfundidad
--    OK, passed 100 tests.
```

En Python

```
# ---------------------------------------------------------------------
# El árbol binario
#         9
#        / \
#       /   \
#      3     7
#     / \
#    2   4
# se puede representar por
#    N(9, N(3, H(2), H(4)), H(7))
#
# El tipo de los árboles binarios se puede definir por
#    @dataclass
#    class Arbol(Generic[A]):
#        pass
```

```
#
#    @dataclass
#    class H(Arbol[A]):
#        x: A
#
#    @dataclass
#    class N(Arbol[A]):
#        x: A
#        i: Arbol[A]
#        d: Arbol[A]
#
# Definir la función
#    profundidad : (Arbol[A]) -> int
# tal que profundidad(x) es la profundidad del árbol x. Por ejemplo,
#    profundidad(N(9, N(3, H(2), H(4)), H(7)))              ==  2
#    profundidad(N(9, N(3, H(2), N(1, H(4), H(5))), H(7)))  ==  3
#    profundidad(N(4, N(5, H(4), H(2)), N(3, H(7), H(4))))  ==  2
#
# Comprobar con Hypothesis que para todo árbol biario x, se tiene que
#    nNodos(x) <= 2^profundidad(x) - 1
# ---------------------------------------------------------------------

from dataclasses import dataclass
from random import choice, randint
from typing import Generic, TypeVar

from hypothesis import given
from hypothesis import strategies as st

A = TypeVar("A")

@dataclass
class Arbol(Generic[A]):
    pass

@dataclass
class H(Arbol[A]):
    x: A

@dataclass
```

```
class N(Arbol[A]):
    x: A
    i: Arbol[A]
    d: Arbol[A]

def profundidad(a: Arbol[A]) -> int:
    match a:
        case H(_):
            return 0
        case N(_, i, d):
            return 1 + max(profundidad(i), profundidad(d))
    assert False

# Comprobación de equivalencia
# ============================

# (arbolArbitrario n) es un árbol aleatorio de orden n. Por ejemplo,
#    >>> arbolArbitrario(4)
#    N(x=2, i=H(x=1), d=H(x=9))
#    >>> arbolArbitrario(4)
#    H(x=10)
#    >>> arbolArbitrario(4)
#    N(x=4, i=N(x=7, i=H(x=4), d=H(x=0)), d=H(x=6))
def arbolArbitrario(n: int) -> Arbol[int]:
    if n <= 1:
        return H(randint(0, 10))
    m = n // 2
    return choice([H(randint(0, 10)),
                   N(randint(0, 10),
                     arbolArbitrario(m),
                     arbolArbitrario(m))])

# nNodos(x) es el número de nodos del árbol x. Por ejemplo,
#    nNodos(N(9, N(3, H(2), H(4)), H(7)))  ==  2
def nNodos(a: Arbol[A]) -> int:
    match a:
        case H(_):
            return 0
        case N(_, i, d):
            return 1 + nNodos(i) + nNodos(d)
```

```python
    assert False

# La propiedad es
@given(st.integers(min_value=1, max_value=10))
def test_nHojas(n: int) -> None:
    a = arbolArbitrario(n)
    assert nNodos(a) <= 2 ** profundidad(a) - 1

# La comprobación es
#    src> poetry run pytest -q profundidad_de_un_arbol_binario.py
#    1 passed in 0.11s
```

5.18. Recorrido de árboles binarios

5.18.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- El árbol binario
--         9
--        / \
--       /   \
--      3     7
--     / \
--    2   4
-- se puede representar por
--    N 9 (N 3 (H 2) (H 4)) (H 7)
--
-- El tipo de los árboles binarios se puede definir por
--    data Arbol a = H a
--                 | N a (Arbol a) (Arbol a)
--      deriving (Show, Eq)
--
-- Definir las funciones
--    preorden  :: Arbol a -> [a]
--    postorden :: Arbol a -> [a]
-- tales que
-- + (preorden x) es la lista correspondiente al recorrido preorden del
--   árbol x; es decir, primero visita la raíz del árbol, a continuación
--   recorre el subárbol izquierdo y, finalmente, recorre el subárbol
--   derecho. Por ejemplo,
```

```haskell
--      preorden (N 9 (N 3 (H 2) (H 4)) (H 7))  ==  [9,3,2,4,7]
-- + (postorden x) es la lista correspondiente al recorrido postorden
--   del árbol x; es decir, primero recorre el subárbol izquierdo, a
--   continuación el subárbol derecho y, finalmente, la raíz del
--   árbol. Por ejemplo,
--      postorden (N 9 (N 3 (H 2) (H 4)) (H 7))  ==  [2,4,3,7,9]
--
-- Comprobar con QuickCheck que la longitud de la lista
-- obtenida recorriendo un árbol en cualquiera de los sentidos es igual
-- al número de nodos del árbol más el número de hojas.
-- ---------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module Recorrido_de_arboles_binarios where

import Test.QuickCheck

data Arbol a = H a
             | N a (Arbol a) (Arbol a)
  deriving (Show, Eq)

preorden :: Arbol a -> [a]
preorden (H x)     = [x]
preorden (N x i d) = x : preorden i ++ preorden d

postorden :: Arbol a -> [a]
postorden (H x)     = [x]
postorden (N x i d) = postorden i ++ postorden d ++ [x]

-- Comprobación de la propiedad
-- ============================

-- (arbolArbitrario n) es un árbol aleatorio de altura n. Por ejemplo,
--    λ> sample (arbolArbitrario 3 :: Gen (Arbol Int))
--    N 0 (H 0) (H 0)
--    N 1 (N (-2) (H (-1)) (H 1)) (H 2)
--    N 3 (H 1) (H 2)
--    N 6 (N 0 (H 5) (H (-5))) (N (-5) (H (-5)) (H 4))
--    H 7
```

```
--    N (-8) (H (-8)) (H 9)
--    H 2
--    N (-1) (H 7) (N 9 (H (-2)) (H (-8)))
--    H (-3)
--    N 0 (N 16 (H (-14)) (H (-18))) (H 7)
--    N (-16) (H 18) (N (-19) (H (-15)) (H (-18)))
arbolArbitrario :: Arbitrary a => Int -> Gen (Arbol a)
arbolArbitrario 0 = H <$> arbitrary
arbolArbitrario n =
  oneof [H <$> arbitrary,
         N <$> arbitrary <*> arbolArbitrario (div n 2) <*> arbolArbitrario (div n

-- Arbol es subclase de Arbitrary
instance Arbitrary a => Arbitrary (Arbol a) where
  arbitrary = sized arbolArbitrario

-- La propiedad es
prop_longitud_recorrido :: Arbol Int -> Bool
prop_longitud_recorrido x =
   length (preorden x)  == n &&
   length (postorden x) == n
   where n = nNodos x + nHojas x

-- (nNodos x) es el número de nodos del árbol x. Por ejemplo,
--    nNodos (N 9 (N 3 (H 2) (H 4)) (H 7))  ==  2
nNodos :: Arbol a -> Int
nNodos (H _)     = 0
nNodos (N _ i d) = 1 + nNodos i + nNodos d

-- (nHojas x) es el número de hojas del árbol x. Por ejemplo,
--    nHojas (N 9 (N 3 (H 2) (H 4)) (H 7))  ==  3
nHojas :: Arbol a -> Int
nHojas (H _)     = 1
nHojas (N _ i d) = nHojas i + nHojas d

-- La comprobación es
--    λ> quickCheck prop_longitud_recorrido
--    OK, passed 100 tests.
```

5.18.2. En Python

```
# ---------------------------------------------------------------------
# El árbol binario
#         9
#        / \
#       /   \
#      3     7
#     / \
#    2   4
# se puede representar por
#    N(9, N(3, H(2), H(4)), H(7))
#
# El tipo de los árboles binarios se puede definir por
#    @dataclass
#    class Arbol(Generic[A]):
#        pass
#
#    @dataclass
#    class H(Arbol[A]):
#        x: A
#
#    @dataclass
#    class N(Arbol[A]):
#        x: A
#        i: Arbol[A]
#        d: Arbol[A]
#
# Definir las funciones
#    preorden  : (Arbol[A]) -> list[A]
#    postorden : (Arbol[A]) -> list[A]
# tales que
# + preorden(x) es la lista correspondiente al recorrido preorden del
#   árbol x; es decir, primero visita la raíz del árbol, a continuación
#   recorre el subárbol izquierdo y, finalmente, recorre el subárbol
#   derecho. Por ejemplo,
#      >>> preorden(N(9, N(3, H(2), H(4)), H(7)))
#      [9, 3, 2, 4, 7]
# + (postorden x) es la lista correspondiente al recorrido postorden
#   del árbol x; es decir, primero recorre el subárbol izquierdo, a
#   continuación el subárbol derecho y, finalmente, la raíz del
```

```python
#   árbol. Por ejemplo,
#      >>> postorden(N(9, N(3, H(2), H(4)), H(7)))
#      [2, 4, 3, 7, 9]
#
# Comprobar con Hypothesis que la longitud de la lista obtenida
# recorriendo un árbol en cualquiera de los sentidos es igual al número
# de nodos del árbol más el número de hojas.
# ---------------------------------------------------------------------

from dataclasses import dataclass
from random import choice, randint
from typing import Generic, TypeVar

from hypothesis import given
from hypothesis import strategies as st

A = TypeVar("A")

@dataclass
class Arbol(Generic[A]):
    pass

@dataclass
class H(Arbol[A]):
    x: A

@dataclass
class N(Arbol[A]):
    x: A
    i: Arbol[A]
    d: Arbol[A]

def preorden(a: Arbol[A]) -> list[A]:
    match a:
        case H(x):
            return [x]
        case N(x, i, d):
            return [x] + preorden(i) + preorden(d)
    assert False
```

```
def postorden(a: Arbol[A]) -> list[A]:
    match a:
        case H(x):
            return [x]
        case N(x, i, d):
            return postorden(i) + postorden(d) + [x]
    assert False

# Comprobación de la propiedad
# ============================

# (arbolArbitrario n) es un árbol aleatorio de orden n. Por ejemplo,
#    >>> arbolArbitrario(4)
#    N(x=2, i=H(x=1), d=H(x=9))
#    >>> arbolArbitrario(4)
#    H(x=10)
#    >>> arbolArbitrario(4)
#    N(x=4, i=N(x=7, i=H(x=4), d=H(x=0)), d=H(x=6))
def arbolArbitrario(n: int) -> Arbol[int]:
    if n <= 1:
        return H(randint(0, 10))
    m = n // 2
    return choice([H(randint(0, 10)),
                   N(randint(0, 10),
                     arbolArbitrario(m),
                     arbolArbitrario(m))])

# nNodos(x) es el número de nodos del árbol x. Por ejemplo,
#    nNodos(N(9, N(3, H(2), H(4)), H(7)))  ==  2
def nNodos(a: Arbol[A]) -> int:
    match a:
        case H(_):
            return 0
        case N(_, i, d):
            return 1 + nNodos(i) + nNodos(d)
    assert False

# (nHojas x) es el número de hojas del árbol x. Por ejemplo,
#    nHojas (N 9 (N 3 (H 2) (H 4)) (H 7))  ==  3
def nHojas(a: Arbol[A]) -> int:
```

```
    match a:
        case H(_):
            return 1
        case N(_, i, d):
            return nHojas(i) + nHojas(d)
    assert False

# La propiedad es
@given(st.integers(min_value=1, max_value=10))
def test_recorrido(n: int) -> None:
    a = arbolArbitrario(n)
    m = nNodos(a) + nHojas(a)
    assert len(preorden(a)) == m
    assert len(postorden(a)) == m

# La comprobación es
#    src> poetry run pytest -q recorrido_de_arboles_binarios.py
#    1 passed in 0.16s
```

5.19. Imagen especular de un árbol binario

5.19.1. En Haskell

```
-- ---------------------------------------------------------------------
-- El árbol binario
--          9
--         / \
--        /   \
--       3     7
--      / \
--     2   4
-- se puede representar por
--    N 9 (N 3 (H 2) (H 4)) (H 7)
--
-- El tipo de los árboles binarios se puede definir por
--    data Arbol a = H a
--                 | N a (Arbol a) (Arbol a)
--      deriving (Show, Eq)
--
-- Definir la función
```

```haskell
--    espejo :: Arbol a -> Arbol a
-- tal que (espejo x) es la imagen especular del árbol x. Por ejemplo,
--    espejo (N 9 (N 3 (H 2) (H 4)) (H 7)) == N 9 (H 7) (N 3 (H 4) (H 2))
--
-- Comprobar con QuickCheck las siguientes propiedades, para todo árbol
-- x,
--    espejo (espejo x) = x
--    reverse (preorden (espejo x)) = postorden x
--    postorden (espejo x) = reverse (preorden x)
-- ---------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module Imagen_especular_de_un_arbol_binario where

import Test.QuickCheck

data Arbol a = H a
             | N a (Arbol a) (Arbol a)
  deriving (Show, Eq)

espejo :: Arbol a -> Arbol a
espejo (H x)     = H x
espejo (N x i d) = N x (espejo d) (espejo i)

-- Generador para las comprobaciones
-- =================================

-- (arbolArbitrario n) es un árbol aleatorio de altura n. Por ejemplo,
--    λ> sample (arbolArbitrario 3 :: Gen (Arbol Int))
--    N 0 (H 0) (H 0)
--    N 1 (N (-2) (H (-1)) (H 1)) (H 2)
--    N 3 (H 1) (H 2)
--    N 6 (N 0 (H 5) (H (-5))) (N (-5) (H (-5)) (H 4))
--    H 7
--    N (-8) (H (-8)) (H 9)
--    H 2
--    N (-1) (H 7) (N 9 (H (-2)) (H (-8)))
--    H (-3)
--    N 0 (N 16 (H (-14)) (H (-18))) (H 7)
```

```
--    N (-16) (H 18) (N (-19) (H (-15)) (H (-18)))
arbolArbitrario :: Arbitrary a => Int -> Gen (Arbol a)
arbolArbitrario 0 = H <$> arbitrary
arbolArbitrario n =
  oneof [H <$> arbitrary,
         N <$> arbitrary <*> arbolArbitrario (div n 2) <*> arbolArbitrario (div n

-- Arbol es subclase de Arbitrary
instance Arbitrary a => Arbitrary (Arbol a) where
  arbitrary = sized arbolArbitrario

-- Funciones auxiliares para la comprobación
-- =========================================

-- (preorden x) es la lista correspondiente al recorrido preorden del
-- árbol x; es decir, primero visita la raíz del árbol, a continuación
-- recorre el subárbol izquierdo y, finalmente, recorre el subárbol
-- derecho. Por ejemplo,
--    preorden (N 9 (N 3 (H 2) (H 4)) (H 7))  ==  [9,3,2,4,7]
preorden :: Arbol a -> [a]
preorden (H x)     = [x]
preorden (N x i d) = x : preorden i ++ preorden d

-- (postorden x) es la lista correspondiente al recorrido postorden
-- del árbol x; es decir, primero recorre el subárbol izquierdo, a
-- continuación el subárbol derecho y, finalmente, la raíz del
-- árbol. Por ejemplo,
--    postorden (N 9 (N 3 (H 2) (H 4)) (H 7))  ==  [2,4,3,7,9]
postorden :: Arbol a -> [a]
postorden (H x)     = [x]
postorden (N x i d) = postorden i ++ postorden d ++ [x]

-- Comprobación de las propiedades
-- ===============================

-- Las propiedades son
prop_espejo :: Arbol Int -> Bool
prop_espejo x =
  espejo (espejo x) == x &&
  reverse (preorden (espejo x)) == postorden x &&
```

```
  postorden (espejo x) == reverse (preorden x)

-- La comprobación es
--    λ> quickCheck prop_espejo
--    OK, passed 100 tests.
```

5.19.2. En Python

```
# ---------------------------------------------------------------------
# El árbol binario
#         9
#        / \
#       /   \
#      3     7
#     / \
#    2   4
# se puede representar por
#    N(9, N(3, H(2), H(4)), H(7))
#
# El tipo de los árboles binarios se puede definir por
#    @dataclass
#    class Arbol(Generic[A]):
#        pass
#
#    @dataclass
#    class H(Arbol[A]):
#        x: A
#
#    @dataclass
#    class N(Arbol[A]):
#        x: A
#        i: Arbol[A]
#        d: Arbol[A]
#
# Definir la función
#    espejo : (Arbol[A]) -> Arbol[A]
# tal que espejo(x) es la imagen especular del árbol x. Por ejemplo,
#    espejo(N(9, N(3, H(2), H(4)), H(7))) == N(9, H(7), N(3, H(4), H(2)))
#
# Comprobar con Hypothesis las siguientes propiedades, para todo árbol
```

```
# x,
#    espejo(espejo(x)) = x
#    list(reversed(preorden(espejo(x)))) == postorden(x)
#    postorden(espejo(x)) == list(reversed(preorden(x)))
# ---------------------------------------------------------------------

from dataclasses import dataclass
from random import choice, randint
from typing import Generic, TypeVar

from hypothesis import given
from hypothesis import strategies as st

A = TypeVar("A")

@dataclass
class Arbol(Generic[A]):
    pass

@dataclass
class H(Arbol[A]):
    x: A

@dataclass
class N(Arbol[A]):
    x: A
    i: Arbol[A]
    d: Arbol[A]

def espejo(a: Arbol[A]) -> Arbol[A]:
    match a:
        case H(x):
            return H(x)
        case N(x, i, d):
            return N(x, espejo(d), espejo(i))
    assert False

# Generador para las comprobaciones
# =================================
```

```
# (arbolArbitrario n) es un árbol aleatorio de orden n. Por ejemplo,
#    >>> arbolArbitrario(4)
#    N(x=2, i=H(x=1), d=H(x=9))
#    >>> arbolArbitrario(4)
#    H(x=10)
#    >>> arbolArbitrario(4)
#    N(x=4, i=N(x=7, i=H(x=4), d=H(x=0)), d=H(x=6))
def arbolArbitrario(n: int) -> Arbol[int]:
    if n <= 1:
        return H(randint(0, 10))
    m = n // 2
    return choice([H(randint(0, 10)),
                   N(randint(0, 10),
                     arbolArbitrario(m),
                     arbolArbitrario(m))])

# Funciones auxiliares para la comprobación
# =========================================

# preorden(x) es la lista correspondiente al recorrido preorden del
# árbol x; es decir, primero visita la raíz del árbol, a continuación
# recorre el subárbol izquierdo y, finalmente, recorre el subárbol
# derecho. Por ejemplo,
#    >>> preorden(N(9, N(3, H(2), H(4)), H(7)))
#    [9, 3, 2, 4, 7]
def preorden(a: Arbol[A]) -> list[A]:
    match a:
        case H(x):
            return [x]
        case N(x, i, d):
            return [x] + preorden(i) + preorden(d)
    assert False

# (postorden x) es la lista correspondiente al recorrido postorden
# del árbol x; es decir, primero recorre el subárbol izquierdo, a
# continuación el subárbol derecho y, finalmente, la raíz del
# árbol. Por ejemplo,
#    >>> postorden(N(9, N(3, H(2), H(4)), H(7)))
#    [2, 4, 3, 7, 9]
def postorden(a: Arbol[A]) -> list[A]:
```

```
    match a:
        case H(x):
            return [x]
        case N(x, i, d):
            return postorden(i) + postorden(d) + [x]
    assert False

# Comprobación de las propiedades
# ===============================

# Las propiedades son
@given(st.integers(min_value=1, max_value=10))
def test_espejo(n: int) -> None:
    x = arbolArbitrario(n)
    assert espejo(espejo(x)) == x
    assert list(reversed(preorden(espejo(x)))) == postorden(x)
    assert postorden(espejo(x)) == list(reversed(preorden(x)))

# La comprobación es
#    src> poetry run pytest -q imagen_especular_de_un_arbol_binario.py
#    1 passed in 0.16s
```

5.20. Subárbol de profundidad dada

5.20.1. En Haskell

```
-- ---------------------------------------------------------------------
-- El árbol binario
--         9
--        / \
--       /   \
--      3     7
--     / \
--    2   4
-- se puede representar por
--    N 9 (N 3 (H 2) (H 4)) (H 7)
--
-- El tipo de los árboles binarios se puede definir por
--    data Arbol a = H a
--                 | N a (Arbol a) (Arbol a)
```

```haskell
--      deriving (Show, Eq)
--
-- La función take está definida por
--    take :: Int -> [a] -> [a]
--    take 0              = []
--    take (n+1) []       = []
--    take (n+1) (x:xs) = x : take n xs
--
-- Definir la función
--    takeArbol ::  Int -> Arbol a -> Arbol a
-- tal que (takeArbol n t) es el subárbol de t de profundidad n. Por
-- ejemplo,
--    takeArbol 0 (N 9 (N 3 (H 2) (H 4)) (H 7)) == H 9
--    takeArbol 1 (N 9 (N 3 (H 2) (H 4)) (H 7)) == N 9 (H 3) (H 7)
--    takeArbol 2 (N 9 (N 3 (H 2) (H 4)) (H 7)) == N 9 (N 3 (H 2) (H 4)) (H 7)
--    takeArbol 3 (N 9 (N 3 (H 2) (H 4)) (H 7)) == N 9 (N 3 (H 2) (H 4)) (H 7)
--
-- Comprobar con QuickCheck que la profundidad de (takeArbol n x) es
-- menor o igual que n, para todo número natural n y todo árbol x.
-- ---------------------------------------------------------------------

module Subarbol_de_profundidad_dada where

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

import Test.QuickCheck

data Arbol a = H a
             | N a (Arbol a) (Arbol a)
  deriving (Show, Eq)

takeArbol :: Int -> Arbol a -> Arbol a
takeArbol _ (H x)     = H x
takeArbol 0 (N x _ _) = H x
takeArbol n (N x i d) = N x (takeArbol (n-1) i) (takeArbol (n-1) d)

-- Generador para las comprobaciones
-- =================================

-- (arbolArbitrario n) es un árbol aleatorio de altura n. Por ejemplo,
```

```
--    λ> sample (arbolArbitrario 3 :: Gen (Arbol Int))
--    N 0 (H 0) (H 0)
--    N 1 (N (-2) (H (-1)) (H 1)) (H 2)
--    N 3 (H 1) (H 2)
--    N 6 (N 0 (H 5) (H (-5))) (N (-5) (H (-5)) (H 4))
--    H 7
--    N (-8) (H (-8)) (H 9)
--    H 2
--    N (-1) (H 7) (N 9 (H (-2)) (H (-8)))
--    H (-3)
--    N 0 (N 16 (H (-14)) (H (-18))) (H 7)
--    N (-16) (H 18) (N (-19) (H (-15)) (H (-18)))
arbolArbitrario :: Arbitrary a => Int -> Gen (Arbol a)
arbolArbitrario 0 = H <$> arbitrary
arbolArbitrario n =
  oneof [H <$> arbitrary,
         N <$> arbitrary <*> arbolArbitrario (div n 2) <*> arbolArbitrario (div n

-- Arbol es subclase de Arbitrary
instance Arbitrary a => Arbitrary (Arbol a) where
  arbitrary = sized arbolArbitrario

-- Función auxiliar para la comprobación
-- =====================================

-- (profundidad x) es la profundidad del árbol x. Por ejemplo,
--    profundidad (N 9 (N 3 (H 2) (H 4)) (H 7))              ==  2
--    profundidad (N 9 (N 3 (H 2) (N 1 (H 4) (H 5))) (H 7))  ==  3
--    profundidad (N 4 (N 5 (H 4) (H 2)) (N 3 (H 7) (H 4)))  ==  2
profundidad :: Arbol a -> Int
profundidad (H _)     = 0
profundidad (N _ i d) = 1 + max (profundidad i) (profundidad d)

-- Comprobación de la propiedad
-- ============================

-- La propiedad es
prop_takeArbol :: Int -> Arbol Int -> Property
prop_takeArbol n x =
  n >= 0 ==> profundidad (takeArbol n x) <= n
```

```
-- La comprobación es
--    λ> quickCheck prop_takeArbol
--    +++ OK, passed 100 tests.
```

5.20.2. En Python

```
# ---------------------------------------------------------------------
# El árbol binario
#         9
#        / \
#       /   \
#      3     7
#     / \
#    2   4
# se puede representar por
#    N(9, N(3, H(2), H(4)), H(7))
#
# El tipo de los árboles binarios se puede definir por
#    @dataclass
#    class Arbol(Generic[A]):
#        pass
#
#    @dataclass
#    class H(Arbol[A]):
#        x: A
#
#    @dataclass
#    class N(Arbol[A]):
#        x: A
#        i: Arbol[A]
#        d: Arbol[A]
#
# Definir la función
#    takeArbol : (int, Arbol[A]) -> Arbol[A]
# tal que takeArbol(n, t) es el subárbol de t de profundidad n. Por
# ejemplo,
#    >>> takeArbol(0, N(9, N(3, H(2), H(4)), H(7)))
#    H(9)
#    >>> takeArbol(1, N(9, N(3, H(2), H(4)), H(7)))
```

```
#    N(9, H(3), H(7))
#    >>> takeArbol(2, N(9, N(3, H(2), H(4)), H(7)))
#    N(9, N(3, H(2), H(4)), H(7))
#    >>> takeArbol(3, N(9, N(3, H(2), H(4)), H(7)))
#    N(9, N(3, H(2), H(4)), H(7))
#
# Comprobar con Hypothesis que la profundidad de takeArbol(n, x) es
# menor o igual que n, para todo número natural n y todo árbol x.
# ---------------------------------------------------------------------

from dataclasses import dataclass
from random import choice, randint
from typing import Generic, TypeVar

from hypothesis import given
from hypothesis import strategies as st

A = TypeVar("A")

@dataclass
class Arbol(Generic[A]):
    pass

@dataclass
class H(Arbol[A]):
    x: A

@dataclass
class N(Arbol[A]):
    x: A
    i: Arbol[A]
    d: Arbol[A]

def takeArbol(n: int, a: Arbol[A]) -> Arbol[A]:
    match (n, a):
        case (_, H(x)):
            return H(x)
        case (0, N(x, _, _)):
            return H(x)
        case (n, N(x, i, d)):
```

```
            return N(x, takeArbol(n - 1, i), takeArbol(n - 1, d))
    assert False

# Generador para las comprobaciones
# =================================

# (arbolArbitrario n) es un árbol aleatorio de orden n. Por ejemplo,
#    >>> arbolArbitrario(4)
#    N(x=2, i=H(x=1), d=H(x=9))
#    >>> arbolArbitrario(4)
#    H(x=10)
#    >>> arbolArbitrario(4)
#    N(x=4, i=N(x=7, i=H(x=4), d=H(x=0)), d=H(x=6))
def arbolArbitrario(n: int) -> Arbol[int]:
    if n <= 1:
        return H(randint(0, 10))
    m = n // 2
    return choice([H(randint(0, 10)),
                   N(randint(0, 10),
                     arbolArbitrario(m),
                     arbolArbitrario(m))])

# Función auxiliar para la comprobación
# =====================================

# profundidad(x) es la profundidad del árbol x. Por ejemplo,
#    profundidad(N(9, N(3, H(2), H(4)), H(7)))              ==  2
#    profundidad(N(9, N(3, H(2), N(1, H(4), H(5))), H(7)))  ==  3
#    profundidad(N(4, N(5, H(4), H(2)), N(3, H(7), H(4))))  ==  2
def profundidad(a: Arbol[A]) -> int:
    match a:
        case H(_):
            return 0
        case N(_, i, d):
            return 1 + max(profundidad(i), profundidad(d))
    assert False

# Comprobación de la propiedad
# ============================
```

```
# La propiedad es
@given(st.integers(min_value=0, max_value=12),
       st.integers(min_value=1, max_value=10))
def test_takeArbol(n: int, m: int) -> None:
    x = arbolArbitrario(m)
    assert profundidad(takeArbol(n, x)) <= n

# La comprobación es
#    src> poetry run pytest -q subarbol_de_profundidad_dada.py
#    1 passed in 0.23s
```

5.21. Árbol de profundidad n con nodos iguales

5.21.1. En Haskell

```
-- ---------------------------------------------------------------------
-- El árbol binario
--         9
--        / \
--       /   \
--      3     7
--     / \
--    2   4
-- se puede representar por
--    N 9 (N 3 (H 2) (H 4)) (H 7)
--
-- El tipo de los árboles binarios se puede definir por
--    data Arbol a = H a
--                 | N a (Arbol a) (Arbol a)
--      deriving (Show, Eq)
--
-- Definir las funciones
--    repeatArbol    :: a -> Arbol a
--    replicateArbol :: Int -> a -> Arbol a
-- tales que
-- + (repeatArbol x) es es árbol con infinitos nodos x. Por ejemplo,
--      takeArbol 0 (repeatArbol 3) == H 3
--      takeArbol 1 (repeatArbol 3) == N 3 (H 3) (H 3)
```

```
--      takeArbol 2 (repeatArbol 3) == N 3 (N 3 (H 3) (H 3)) (N 3 (H 3) (H 3))
-- + (replicate n x) es el árbol de profundidad n cuyos nodos son x. Por
--   ejemplo,
--      replicateArbol 0 5  ==  H 5
--      replicateArbol 1 5  ==  N 5 (H 5) (H 5)
--      replicateArbol 2 5  ==  N 5 (N 5 (H 5) (H 5)) (N 5 (H 5) (H 5))
--
-- Comprobar con QuickCheck que el número de hojas de
-- (replicateArbol n x) es 2^n, para todo número natural n.
-- ---------------------------------------------------------------------

module Arbol_de_profundidad_n_con_nodos_iguales where

import Test.QuickCheck

data Arbol a = H a
             | N a (Arbol a) (Arbol a)
  deriving (Show, Eq)

repeatArbol :: a -> Arbol a
repeatArbol x = N x t t
  where t = repeatArbol x

replicateArbol :: Int -> a -> Arbol a
replicateArbol n = takeArbol n . repeatArbol

-- (takeArbol n t) es el subárbol de t de profundidad n. Por ejemplo,
--    takeArbol 0 (N 9 (N 3 (H 2) (H 4)) (H 7)) == H 9
--    takeArbol 1 (N 9 (N 3 (H 2) (H 4)) (H 7)) == N 9 (H 3) (H 7)
--    takeArbol 2 (N 9 (N 3 (H 2) (H 4)) (H 7)) == N 9 (N 3 (H 2) (H 4)) (H 7)
--    takeArbol 3 (N 9 (N 3 (H 2) (H 4)) (H 7)) == N 9 (N 3 (H 2) (H 4)) (H 7)
takeArbol :: Int -> Arbol a -> Arbol a
takeArbol _ (H x)     = H x
takeArbol 0 (N x _ _) = H x
takeArbol n (N x i d) = N x (takeArbol (n-1) i) (takeArbol (n-1) d)

-- Generador para las comprobaciones
-- =================================

-- (arbolArbitrario n) es un árbol aleatorio de altura n. Por ejemplo,
```

```
--    λ> sample (arbolArbitrario 3 :: Gen (Arbol Int))
--    N 0 (H 0) (H 0)
--    N 1 (N (-2) (H (-1)) (H 1)) (H 2)
--    N 3 (H 1) (H 2)
--    N 6 (N 0 (H 5) (H (-5))) (N (-5) (H (-5)) (H 4))
--    H 7
--    N (-8) (H (-8)) (H 9)
--    H 2
--    N (-1) (H 7) (N 9 (H (-2)) (H (-8)))
--    H (-3)
--    N 0 (N 16 (H (-14)) (H (-18))) (H 7)
--    N (-16) (H 18) (N (-19) (H (-15)) (H (-18)))
arbolArbitrario :: Arbitrary a => Int -> Gen (Arbol a)
arbolArbitrario 0 = H <$> arbitrary
arbolArbitrario n =
  oneof [H <$> arbitrary,
         N <$> arbitrary <*> arbolArbitrario (div n 2) <*> arbolArbitrario (div n

-- Arbol es subclase de Arbitrary
instance Arbitrary a => Arbitrary (Arbol a) where
  arbitrary = sized arbolArbitrario

-- Función auxiliar para la comprobación
-- =====================================

-- (nHojas x) es el número de hojas del árbol x. Por ejemplo,
--    nHojas (N 9 (N 3 (H 2) (H 4)) (H 7))  ==  3
nHojas :: Arbol a -> Int
nHojas (H _)     = 1
nHojas (N _ i d) = nHojas i + nHojas d

-- Comprobación de la propiedad
-- ============================

-- La propiedad es
prop_replicateArbol :: Int -> Int -> Property
prop_replicateArbol n x =
  n >= 0 ==> nHojas (replicateArbol n x) == 2^n

-- La comprobación es
```

```
--    λ> quickCheckWith (stdArgs {maxSize=7}) prop_replicateArbol
--    +++ OK, passed 100 tests.
```

5.21.2. En Python

```
# ---------------------------------------------------------------------
# El árbol binario
#          9
#         / \
#        /   \
#       3     7
#      / \
#     2   4
# se puede representar por
#    N(9, N(3, H(2), H(4)), H(7))
#
# El tipo de los árboles binarios se puede definir por
#    @dataclass
#    class Arbol(Generic[A]):
#        pass
#
#    @dataclass
#    class H(Arbol[A]):
#        x: A
#
#    @dataclass
#    class N(Arbol[A]):
#        x: A
#        i: Arbol[A]
#        d: Arbol[A]
#
# Definir la función
#    replicateArbol : (int, A) -> Arbol[A]
# tal que (replicate n x) es el árbol de profundidad n cuyos nodos son
# x. Por ejemplo,
#    >>> replicateArbol(0, 5)
#    H(5)
#    >>> replicateArbol(1, 5)
#    N(5, H(5), H(5))
#    >>> replicateArbol(2, 5)
```

```python
#    N(5, N(5, H(5), H(5)), N(5, H(5), H(5)))
#
# Comprobar con Hypothesis que el número de hojas de
# replicateArbol(n, x) es 2^n, para todo número natural n.
# ---------------------------------------------------------------------

from dataclasses import dataclass
from random import choice, randint
from typing import Generic, TypeVar

from hypothesis import given
from hypothesis import strategies as st

A = TypeVar("A")

@dataclass
class Arbol(Generic[A]):
    pass

@dataclass
class H(Arbol[A]):
    x: A

@dataclass
class N(Arbol[A]):
    x: A
    i: Arbol[A]
    d: Arbol[A]

def replicateArbol(n: int, x: A) -> Arbol[A]:
    match n:
        case 0:
            return H(x)
        case n:
            t = replicateArbol(n - 1, x)
            return N(x, t, t)
    assert False

# Generador para las comprobaciones
# =================================
```

```
# (arbolArbitrario n) es un árbol aleatorio de orden n. Por ejemplo,
#    >>> arbolArbitrario(4)
#    N(x=2, i=H(x=1), d=H(x=9))
#    >>> arbolArbitrario(4)
#    H(x=10)
#    >>> arbolArbitrario(4)
#    N(x=4, i=N(x=7, i=H(x=4), d=H(x=0)), d=H(x=6))
def arbolArbitrario(n: int) -> Arbol[int]:
    if n <= 1:
        return H(randint(0, 10))
    m = n // 2
    return choice([H(randint(0, 10)),
                   N(randint(0, 10),
                     arbolArbitrario(m),
                     arbolArbitrario(m))])

# Función auxiliar para la comprobación
# =====================================

# nHojas(x) es el número de hojas del árbol x. Por ejemplo,
#    nHojas(N(9, N(3, H(2), H(4)), H(7)))  ==  3
def nHojas(a: Arbol[A]) -> int:
    match a:
        case H(_):
            return 1
        case N(_, i, d):
            return nHojas(i) + nHojas(d)
    assert False

# Comprobación de la propiedad
# ============================

# La propiedad es
@given(st.integers(min_value=1, max_value=10),
       st.integers(min_value=1, max_value=10))
def test_nHojas(n: int, x: int) -> None:
    assert nHojas(replicateArbol(n, x)) == 2**n

# La comprobación es
```

```
#    src> poetry run pytest -q arbol_de_profundidad_n_con_nodos_iguales.py
#    1 passed in 0.20s
```

5.22. Suma de un árbol

5.22.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Los árboles binarios con valores en los nodos se pueden definir por
--    data Arbol a = H
--                 | N a (Arbol a) (Arbol a)
--      deriving (Show, Eq)
-- Por ejemplo, el árbol
--         9
--        / \
--       /   \
--      8     6
--     / \   / \
--    3   2 4   5
-- se puede representar por
--    N 9 (N 8 (N 3 H H) (N 2 H H)) (N 6 (N 4 H H) (N 5 H H))
--
-- Definir por recursión la función
--    sumaArbol :: Num a => Arbol1 a -> a
-- tal (sumaArbol x) es la suma de los valores que hay en el árbol
-- x. Por ejemplo,
--    sumaArbol (N 2 (N 5 (N 3 H H) (N 7 H H)) (N 4 H H)) == 21
-- ---------------------------------------------------------------------

module Suma_de_un_arbol where

data Arbol1 a = H
              | N a (Arbol1 a) (Arbol1 a)
  deriving (Show, Eq)

sumaArbol :: Num a => Arbol1 a -> a
sumaArbol H         = 0
sumaArbol (N x i d) = x + sumaArbol i + sumaArbol d
```

5.22.2. En Python

```
# ---------------------------------------------------------------------
# Los árboles binarios con valores en los nodos se pueden definir por
#    @dataclass
#    class Arbol:
#        pass
#
#    @dataclass
#    class H(Arbol):
#        pass
#
#    @dataclass
#    class N(Arbol):
#        x: int
#        i: Arbol
#        d: Arbol
# Por ejemplo, el árbol
#         9
#        / \
#       /   \
#      8     6
#     / \   / \
#    3   2 4   5
# se puede representar por
#    N(9, N(8, N(3, H(), H()), N(2, H(), H())), N(6, N(4, H(), H()), N(5, H(), H(
#
# Definir la función
#    sumaArbol : (Arbol) -> int
# tal sumaArbol(x) es la suma de los valores que hay en el árbol x.
# Por ejemplo,
#    >>> sumaArbol(N(2, N(5, N(3, H(), H()), N(7, H(), H())), N(4, H(), H())))
#    21
# ---------------------------------------------------------------------

from dataclasses import dataclass


@dataclass
class Arbol:
    pass
```

```python
@dataclass
class H(Arbol):
    pass

@dataclass
class N(Arbol):
    x: int
    i: Arbol
    d: Arbol

def sumaArbol(a: Arbol) -> int:
    match a:
        case H():
            return 0
        case N(x, i, d):
            return x + sumaArbol(i) + sumaArbol(d)
    assert False
```

5.23. Rama izquierda de un árbol binario

5.23.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Los árboles binarios con valores en los nodos se pueden definir por
--    data Arbol a = H
--                 | N a (Arbol a) (Arbol a)
--      deriving (Show, Eq)
-- Por ejemplo, el árbol
--          9
--         / \
--        /   \
--       8     6
--      / \   / \
--     3   2 4   5
-- se puede representar por
--    N 9 (N 8 (N 3 H H) (N 2 H H)) (N 6 (N 4 H H) (N 5 H H))
--
-- Definir la función
--    ramaIzquierda :: Arbol a -> [a]
```

```haskell
-- tal que (ramaIzquierda a) es la lista de los valores de los nodos de
-- la rama izquierda del árbol a. Por ejemplo,
--    λ> ramaIzquierda (N 2 (N 5 (N 3 H H) (N 7 H H)) (N 4 H H))
--    [2,5,3]
-- ---------------------------------------------------------------------

module Rama_izquierda_de_un_arbol_binario where

data Arbol a = H
             | N a (Arbol a) (Arbol a)
  deriving (Show, Eq)

ramaIzquierda :: Arbol a -> [a]
ramaIzquierda H         = []
ramaIzquierda (N x i _) = x : ramaIzquierda i
```

5.23.2. En Python

```python
# ---------------------------------------------------------------------
# Los árboles binarios con valores en los nodos se pueden definir por
#    @dataclass
#    class Arbol(Generic[A]):
#        pass
#
#    @dataclass
#    class H(Arbol[A]):
#        pass
#
#    @dataclass
#    class N(Arbol[A]):
#        x: A
#        i: Arbol[A]
#        d: Arbol[A]
# Por ejemplo, el árbol
#         9
#        / \
#       /   \
#      8     6
#     / \   / \
#    3   2 4   5
```

```
# se puede representar por
#    N(9, N(8, N(3, H(), H()), N(2, H(), H())), N(6, N(4, H(), H()), N(5, H(), H(
#
# Definir la función
#    ramaIzquierda : (Arbol[A]) -> list[A]
# tal que ramaIzquierda(a) es la lista de los valores de los nodos de
# la rama izquierda del árbol a. Por ejemplo,
#    >>> ramaIzquierda(N(2, N(5, N(3, H(), H()), N(7, H(), H())), N(4, H(), H()))
#    [2, 5, 3]
# ---------------------------------------------------------------------

from dataclasses import dataclass
from typing import Generic, TypeVar

A = TypeVar("A")

@dataclass
class Arbol(Generic[A]):
    pass

@dataclass
class H(Arbol[A]):
    pass

@dataclass
class N(Arbol[A]):
    x: A
    i: Arbol[A]
    d: Arbol[A]

def ramaIzquierda(a: Arbol[A]) -> list[A]:
    match a:
        case H():
            return []
        case N(x, i, _):
            return [x] + ramaIzquierda(i)
    assert False
```

5.24. Árboles balanceados

5.24.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Los árboles binarios con valores en los nodos se pueden definir por
--    data Arbol a = H
--                 | N a (Arbol a) (Arbol a)
--      deriving (Show, Eq)
-- Por ejemplo, el árbol
--          9
--         / \
--        /   \
--       8     6
--      / \   / \
--     3   2 4   5
-- se puede representar por
--    N 9 (N 8 (N 3 H H) (N 2 H H)) (N 6 (N 4 H H) (N 5 H H))
--
-- Diremos que un árbol está balanceado si para cada nodo la diferencia
-- entre el número de nodos de sus subárboles izquierdo y derecho es
-- menor o igual que uno.
--
-- Definir la función
--    balanceado :: Arbol a -> Bool
-- tal que (balanceado a) se verifica si el árbol a está balanceado. Por
-- ejemplo,
--    λ> balanceado (N 5 H (N 3 H H))
--    True
--    λ> balanceado (N 4 (N 3 (N 2 H H) H) (N 5 H (N 6 H (N 7 H H))))
--    False
-- ---------------------------------------------------------------------

module Arboles_balanceados where

data Arbol a = H
             | N a (Arbol a) (Arbol a)
  deriving (Show, Eq)

balanceado :: Arbol a -> Bool
```

```
balanceado H         = True
balanceado (N _ i d) = abs (numeroNodos i - numeroNodos d) <= 1
                       && balanceado i
                       && balanceado d

-- (numeroNodos a) es el número de nodos del árbol a. Por ejemplo,
--    numeroNodos (N 5 H (N 3 H H)) ==  2
numeroNodos :: Arbol a -> Int
numeroNodos H         = 0
numeroNodos (N _ i d) = 1 + numeroNodos i + numeroNodos d
```

5.24.2. En Python

```
# ---------------------------------------------------------------------
# Los árboles binarios con valores en los nodos se pueden definir por
#    @dataclass
#    class Arbol(Generic[A]):
#        pass
#
#    @dataclass
#    class H(Arbol[A]):
#        pass
#
#    @dataclass
#    class N(Arbol[A]):
#        x: A
#        i: Arbol[A]
#        d: Arbol[A]
# Por ejemplo, el árbol
#         9
#        / \
#       /   \
#      8     6
#     / \   / \
#    3   2 4   5
# se puede representar por
#    N(9, N(8, N(3, H(), H()), N(2, H(), H())), N(6, N(4, H(), H()), N(5, H(), H(
#
# Diremos que un árbol está balanceado si para cada nodo la diferencia
# entre el número de nodos de sus subárboles izquierdo y derecho es
```

```
# menor o igual que uno.
#
# Definir la función
#    balanceado : (Arbol[A]) -> bool
# tal que balanceado(a) se verifica si el árbol a está balanceado. Por
# ejemplo,
#    >>> balanceado(N(5, H(), N(3, H(), H())))
#    True
#    >>> balanceado(N(4, N(3, N(2, H(), H()), H()), N(5, H(), N(6, H(), N(7, H(),
#    False
# ---------------------------------------------------------------------

from dataclasses import dataclass
from typing import Generic, TypeVar

A = TypeVar("A")

@dataclass
class Arbol(Generic[A]):
    pass

@dataclass
class H(Arbol[A]):
    pass

@dataclass
class N(Arbol[A]):
    x: A
    i: Arbol[A]
    d: Arbol[A]

def numeroNodos(a: Arbol[A]) -> int:
    match a:
        case H():
            return 0
        case N(_, i, d):
            return 1 + numeroNodos(i) + numeroNodos(d)
    assert False

def balanceado(a: Arbol[A]) -> bool:
```

```
    match a:
        case H():
            return True
        case N(_, i, d):
            return abs(numeroNodos(i) - numeroNodos(d)) <= 1 \
                and balanceado(i) and balanceado(d)
    assert False
```

5.25. Árboles con bordes iguales

5.25.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Los árboles binarios con valores en las hojas se pueden definir por
--    data Arbol a = H a
--                 | N (Arbol a) (Arbol a)
--                 deriving Show
-- Por ejemplo, los árboles
--    árbol1          árbol2       árbol3     árbol4
--       o              o            o           o
--      / \            / \          / \         / \
--     1   o          o   3        o   3       o   1
--        / \        / \          / \         / \
--       2   3      1   2        1   4       2   3
-- se representan por
--    arbol1, arbol2, arbol3, arbol4 :: Arbol Int
--    arbol1 = N (H 1) (N (H 2) (H 3))
--    arbol2 = N (N (H 1) (H 2)) (H 3)
--    arbol3 = N (N (H 1) (H 4)) (H 3)
--    arbol4 = N (N (H 2) (H 3)) (H 1)
--
-- Definir la función
--    igualBorde :: Eq a => Arbol a -> Arbol a -> Bool
-- tal que (igualBorde t1 t2) se verifica si los bordes de los árboles
-- t1 y t2 son iguales. Por ejemplo,
--    igualBorde arbol1 arbol2  ==  True
--    igualBorde arbol1 arbol3  ==  False
--    igualBorde arbol1 arbol4  ==  False
-- ---------------------------------------------------------------------
```

```haskell
module Arboles_con_bordes_iguales where

data Arbol a = N (Arbol a) (Arbol a)
             | H a
             deriving Show

arbol1, arbol2, arbol3, arbol4 :: Arbol Int
arbol1 = N (H 1) (N (H 2) (H 3))
arbol2 = N (N (H 1) (H 2)) (H 3)
arbol3 = N (N (H 1) (H 4)) (H 3)
arbol4 = N (N (H 2) (H 3)) (H 1)

igualBorde :: Eq a => Arbol a -> Arbol a -> Bool
igualBorde t1 t2 = borde t1 == borde t2

-- (borde t) es el borde del árbol t; es decir, la lista de las hojas
-- del árbol t leídas de izquierda a derecha. Por ejemplo,
--    borde arbol4  ==  [2,3,1]
borde :: Arbol a -> [a]
borde (N i d) = borde i ++ borde d
borde (H x)   = [x]
```

5.25.2. En Python

```python
# ---------------------------------------------------------------------
# Los árboles binarios con valores en las hojas se pueden definir por
#    @dataclass
#    class Arbol(Generic[A]):
#        pass
#
#    @dataclass
#    class H(Arbol[A]):
#        x: A
#
#    @dataclass
#    class N(Arbol[A]):
#        i: Arbol[A]
#        d: Arbol[A]
# Por ejemplo, los árboles
#    árbol1          árbol2       árbol3     árbol4
```

```
#       o             o           o           o
#      / \           / \         / \         / \
#     1   o         o   3       o   3       o   1
#        / \       / \         / \         / \
#       2   3     1   2       1   4       2   3
# se representan por
#    arbol1: Arbol[int] = N(H(1), N(H(2), H(3)))
#    arbol2: Arbol[int] = N(N(H(1), H(2)), H(3))
#    arbol3: Arbol[int] = N(N(H(1), H(4)), H(3))
#    arbol4: Arbol[int] = N(N(H(2), H(3)), H(1))
#
# Definir la función
#    igualBorde : (Arbol[A], Arbol[A]) -> bool
# tal que igualBorde(t1, t2) se verifica si los bordes de los árboles
# t1 y t2 son iguales. Por ejemplo,
#    igualBorde(arbol1, arbol2)  ==  True
#    igualBorde(arbol1, arbol3)  ==  False
#    igualBorde(arbol1, arbol4)  ==  False
# ---------------------------------------------------------------------

from dataclasses import dataclass
from typing import Generic, TypeVar

A = TypeVar("A")

@dataclass
class Arbol(Generic[A]):
    pass

@dataclass
class H(Arbol[A]):
    x: A

@dataclass
class N(Arbol[A]):
    i: Arbol[A]
    d: Arbol[A]

arbol1: Arbol[int] = N(H(1), N(H(2), H(3)))
arbol2: Arbol[int] = N(N(H(1), H(2)), H(3))
```

```
arbol3: Arbol[int] = N(N(H(1), H(4)), H(3))
arbol4: Arbol[int] = N(N(H(2), H(3)), H(1))

# borde(t) es el borde del árbol t; es decir, la lista de las hojas
# del árbol t leídas de izquierda a derecha. Por ejemplo,
#    borde(arbol4)  ==  [2, 3, 1]
def borde(a: Arbol[A]) -> list[A]:
    match a:
        case H(x):
            return [x]
        case N(i, d):
            return borde(i) + borde(d)
    assert False

def igualBorde(t1: Arbol[A], t2: Arbol[A]) -> bool:
    return borde(t1) == borde(t2)
```

5.26. Árboles con igual estructura

5.26.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Los árboles binarios con valores en las hojas y en los nodos se
-- definen por
--    data Arbol a = H a
--                 | N a (Arbol a) (Arbol a)
--      deriving Show
-- Por ejemplo, los árboles
--          5              8             5           5
--         / \            / \           / \         / \
--        /   \          /   \         /   \       /   \
--       9     7        9     3       9     2     4     7
--      / \   / \      / \   / \     / \               / \
--     1   4 6   8    1   4 6   2   1   4             6   2
-- se pueden representar por
--    ej3arbol1, ej3arbol2, ej3arbol3, ej3arbol4 :: Arbol Int
--    ej3arbol1 = N 5 (N 9 (H 1) (H 4)) (N 7 (H 6) (H 8))
--    ej3arbol2 = N 8 (N 9 (H 1) (H 4)) (N3 3 (H 6) (H 2))
--    ej3arbol3 = N 5 (N 9 (H 1) (H 4)) (H 2)
--    ej3arbol4 = N 5 (H 4) (N 7 (H 6) (H 2))
```

```
--
-- Definir la función
--    igualEstructura :: Arbol -> Arbol -> Bool
-- tal que (igualEstructura a1 a2) se verifica si los árboles a1 y a2
-- tienen la misma estructura. Por ejemplo,
--    igualEstructura ej3arbol1 ej3arbol2 == True
--    igualEstructura ej3arbol1 ej3arbol3 == False
--    igualEstructura ej3arbol1 ej3arbol4 == False
-- ---------------------------------------------------------------------

module Arboles_con_igual_estructura where

data Arbol a = H a
             | N a (Arbol a) (Arbol a)
  deriving (Show, Eq)

ej3arbol1, ej3arbol2, ej3arbol3, ej3arbol4 :: Arbol Int
ej3arbol1 = N 5 (N 9 (H 1) (H 4)) (N 7 (H 6) (H 8))
ej3arbol2 = N 8 (N 9 (H 1) (H 4)) (N 3 (H 6) (H 2))
ej3arbol3 = N 5 (N 9 (H 1) (H 4)) (H 2)
ej3arbol4 = N 5 (H 4) (N 7 (H 6) (H 2))

igualEstructura :: Arbol a -> Arbol a -> Bool
igualEstructura (H _) (H _)             = True
igualEstructura (N _ i1 d1) (N _ i2 d2) =
  igualEstructura i1 i2 &&
  igualEstructura d1 d2
igualEstructura _ _                     = False
```

5.26.2. En Python

```
# ---------------------------------------------------------------------
# Los árboles binarios con valores en las hojas y en los nodos se
# definen por
#    @dataclass
#    class Arbol(Generic[A]):
#        pass
#
#    @dataclass
#    class H(Arbol[A]):
```

```
#        x: A
#
#    @dataclass
#    class N(Arbol[A]):
#        x: A
#        i: Arbol[A]
#        d: Arbol[A]
# Por ejemplo, los árboles
#         5              8             5           5
#        / \            / \           / \         / \
#       /   \          /   \         /   \       /   \
#      9     7        9     3       9     2     4     7
#     / \   / \      / \   / \     / \                / \
#    1   4 6   8    1   4 6   2   1   4             6   2
# se pueden representar por
#     ej3arbol1: Arbol[int] = N(5, N(9, H(1), H(4)), N(7, H(6), H(8)))
#     ej3arbol2: Arbol[int] = N(8, N(9, H(1), H(4)), N(3, H(6), H(2)))
#     ej3arbol3: Arbol[int] = N(5, N(9, H(1), H(4)), H(2))
#     ej3arbol4: Arbol[int] = N(5, H(4), N(7, H(6), H(2)))
#
# Definir la función
#    igualEstructura : (Arbol[A], Arbol[A]) -> bool
# tal que igualEstructura(a1, a2) se verifica si los árboles a1 y a2
# tienen la misma estructura. Por ejemplo,
#    igualEstructura(ej3arbol1, ej3arbol2) == True
#    igualEstructura(ej3arbol1, ej3arbol3) == False
#    igualEstructura(ej3arbol1, ej3arbol4) == False
# ---------------------------------------------------------------------

from dataclasses import dataclass
from typing import Generic, TypeVar

A = TypeVar("A")

@dataclass
class Arbol(Generic[A]):
    pass

@dataclass
class H(Arbol[A]):
```

```
    x: A

@dataclass
class N(Arbol[A]):
    x: A
    i: Arbol[A]
    d: Arbol[A]

ej3arbol1: Arbol[int] = N(5, N(9, H(1), H(4)), N(7, H(6), H(8)))
ej3arbol2: Arbol[int] = N(8, N(9, H(1), H(4)), N(3, H(6), H(2)))
ej3arbol3: Arbol[int] = N(5, N(9, H(1), H(4)), H(2))
ej3arbol4: Arbol[int] = N(5, H(4), N(7, H(6), H(2)))

def igualEstructura(a: Arbol[A], b: Arbol[A]) -> bool:
    match (a, b):
        case (H(_), H(_)):
            return True
        case (N(_, i1, d1), N(_, i2, d2)):
            return igualEstructura(i1, i2) and igualEstructura(d1, d2)
        case (_, _):
            return False
    assert False
```

5.27. Existencia de elementos del árbol que verifican una propiedad

5.27.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Los árboles binarios con valores en las hojas y en los nodos se
-- definen por
--    data Arbol a = H a
--                 | N a (Arbol a) (Arbol a)
--      deriving Show
-- Por ejemplo, el árbol
--         5
--        / \
--       /   \
--      3     2
```

```
--      / \
--     1   4
-- se representa por
--     N 5 (N 3 (H 1) (H 4)) (H 2)
--
-- Definir la función
--     algunoArbol :: Arbol t -> (t -> Bool) -> Bool
-- tal que (algunoArbol a p) se verifica si algún elemento del árbol a
-- cumple la propiedad p. Por ejemplo,
--     algunoArbol (N 5 (N 3 (H 1) (H 4)) (H 2)) (>4)  ==  True
--     algunoArbol (N 5 (N 3 (H 1) (H 4)) (H 2)) (>7)  ==  False
-- ---------------------------------------------------------------------

module Existencia_de_elemento_del_arbol_con_propiedad where

data Arbol a = H a
             | N a (Arbol a) (Arbol a)
  deriving Show

algunoArbol :: Arbol a -> (a -> Bool) -> Bool
algunoArbol (H x) p     = p x
algunoArbol (N x i d) p = p x || algunoArbol i p || algunoArbol d p
```

5.27.2. En Python

```
# ---------------------------------------------------------------------
# Los árboles binarios con valores en las hojas y en los nodos se
# definen por
#    @dataclass
#    class Arbol(Generic[A]):
#        pass
#
#    @dataclass
#    class H(Arbol[A]):
#        x: A
#
#    @dataclass
#    class N(Arbol[A]):
#        x: A
#        i: Arbol[A]
```

```
#        d: Arbol[A]
# Por ejemplo, el árbol
#         5
#        / \
#       /   \
#      3     2
#     / \
#    1   4
# se representa por
#    N(5, N(3, H(1), H(4)), H(2))
#
# Definir la función
#    algunoArbol : (Arbol[A], Callable[[A], bool]) -> bool
# tal que algunoArbol(a, p) se verifica si algún elemento del árbol a
# cumple la propiedad p. Por ejemplo,
#    >>> algunoArbol(N(5, N(3, H(1), H(4)), H(2)), lambda x: x > 4)
#    True
#    >>> algunoArbol(N(5, N(3, H(1), H(4)), H(2)), lambda x: x > 7)
#    False
# ---------------------------------------------------------------------

from dataclasses import dataclass
from typing import Callable, Generic, TypeVar

A = TypeVar("A")

@dataclass
class Arbol(Generic[A]):
    pass

@dataclass
class H(Arbol[A]):
    x: A

@dataclass
class N(Arbol[A]):
    x: A
    i: Arbol[A]
    d: Arbol[A]
```

```python
def algunoArbol(a: Arbol[A], p: Callable[[A], bool]) -> bool:
    match a:
        case H(x):
            return p(x)
        case N(x, i, d):
            return p(x) or algunoArbol(i, p) or algunoArbol(d, p)
    assert False
```

5.28. Elementos del nivel k de un árbol

5.28.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Los árboles binarios con valores en las hojas y en los nodos se
-- definen por
--    data Arbol a = H a
--                 | N a (Arbol a) (Arbol a)
-- Por ejemplo, el árbol
--         7
--        / \
--       /   \
--      2     9
--     / \
--    5   4
-- se representa por
--    N 7 (N 2 (H 5) (H 4)) (H 9)
--
-- Un elemento de un árbol se dirá de nivel k si aparece en el árbol a
-- distancia k  de la raíz.
--
-- Definir la función
--    nivel :: Int -> Arbol a -> [a]
-- tal que (nivel k a) es la lista de los elementos de nivel k del árbol
-- a. Por ejemplo,
--    nivel 0 (H 5)                          ==  [5]
--    nivel 1 (H 5)                          ==  []
--    nivel 0 (N 7 (N 2 (H 5) (H 4)) (H 9))  ==  [7]
--    nivel 1 (N 7 (N 2 (H 5) (H 4)) (H 9))  ==  [2,9]
--    nivel 2 (N 7 (N 2 (H 5) (H 4)) (H 9))  ==  [5,4]
--    nivel 3 (N 7 (N 2 (H 5) (H 4)) (H 9))  ==  []
```

```
-- ---------------------------------------------------------------------

module Elementos_del_nivel_k_de_un_arbol where

data Arbol a = H a
             | N a (Arbol a) (Arbol a)

nivel :: Int -> Arbol a -> [a]
nivel 0 (H x)      = [x]
nivel 0 (N x _  _) = [x]
nivel _ (H _ )     = []
nivel k (N _ i d)  = nivel (k-1) i ++ nivel (k-1) d
```

5.28.2. En Python

```
# ---------------------------------------------------------------------
# Los árboles binarios con valores en las hojas y en los nodos se
# definen por
#    @dataclass
#    class Arbol(Generic[A]):
#        pass
#
#    @dataclass
#    class H(Arbol[A]):
#        x: A
#
#    @dataclass
#    class N(Arbol[A]):
#        x: A
#        i: Arbol[A]
#        d: Arbol[A]
# Por ejemplo, el árbol
#         7
#        / \
#       /   \
#      2     9
#     / \
#    5   4
# se representa por
#    N(7, N(2, H(5), H(4)), H(9))
```

```
#
# Un elemento de un árbol se dirá de nivel k si aparece en el árbol a
# distancia k  de la raíz.
#
# Definir la función
#    nivel : (int, Arbol[A]) -> list[A]
# tal que nivel(k, a) es la lista de los elementos de nivel k del árbol
# a. Por ejemplo,
#     >>> nivel(0, N(7, N(2, H(5), H(4)), H(9)))
#     [7]
#     >>> nivel(1, N(7, N(2, H(5), H(4)), H(9)))
#     [2, 9]
#     >>> nivel(2, N(7, N(2, H(5), H(4)), H(9)))
#     [5, 4]
#     >>> nivel(3, N(7, N(2, H(5), H(4)), H(9)))
#     []
# ---------------------------------------------------------------------

from dataclasses import dataclass
from typing import Generic, TypeVar

A = TypeVar("A")

@dataclass
class Arbol(Generic[A]):
    pass

@dataclass
class H(Arbol[A]):
    x: A

@dataclass
class N(Arbol[A]):
    x: A
    i: Arbol[A]
    d: Arbol[A]

def nivel(k: int, a: Arbol[A]) -> list[A]:
    match (k, a):
        case (0, H(x)):
```

```
            return [x]
        case (0, N(x, _, _)):
            return [x]
        case (_, H(_)):
            return []
        case (_, N(_, i, d)):
            return nivel(k - 1, i) + nivel(k - 1, d)
    assert False
```

5.29. Árbol de factorización

5.29.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Los divisores medios de un número son los que ocupan la posición
-- media entre los divisores de n, ordenados de menor a mayor. Por
-- ejemplo, los divisores de 60 son [1,2,3,4,5,6,10,12,15,20,30,60] y
-- sus divisores medios son 6 y 10. Para los números que son cuadrados
-- perfectos, sus divisores medios de son sus raíces cuadradas; por
-- ejemplos, los divisores medios de 9 son 3 y 3.
--
-- El árbol de factorización de un número compuesto n se construye de la
-- siguiente manera:
--    * la raíz es el número n,
--    * la rama izquierda es el árbol de factorización de su divisor
--      medio menor y
--    * la rama derecha es el árbol de factorización de su divisor
--      medio mayor
-- Si el número es primo, su árbol de factorización sólo tiene una hoja
-- con dicho número. Por ejemplo, el árbol de factorización de 60 es
--        60
--       /  \
--      6    10
--     / \   / \
--    2   3 2   5
--
-- Definir la función
--    arbolFactorizacion :: Int -> Arbol
-- tal que (arbolFactorizacion n) es el árbol de factorización de n. Por
-- ejemplo,
```

```
--    arbolFactorizacion 60 == N 60 (N 6 (H 2) (H 3)) (N 10 (H 2) (H 5))
--    arbolFactorizacion 45 == N 45 (H 5) (N 9 (H 3) (H 3))
--    arbolFactorizacion 7  == H 7
--    arbolFactorizacion 9  == N 9 (H 3) (H 3)
--    arbolFactorizacion 14 == N 14 (H 2) (H 7)
--    arbolFactorizacion 28 == N 28 (N 4 (H 2) (H 2)) (H 7)
--    arbolFactorizacion 84 == N 84 (H 7) (N 12 (H 3) (N 4 (H 2) (H 2)))
-- ---------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-type-defaults #-}

module Arbol_de_factorizacion where

import Test.QuickCheck

-- 1ª solución
-- ===========

data Arbol = H Int
           | N Int Arbol Arbol
  deriving (Eq, Show)

arbolFactorizacion1 :: Int -> Arbol
arbolFactorizacion1 n
  | esPrimo n = H n
  | otherwise = N n (arbolFactorizacion1 x) (arbolFactorizacion1 y)
  where (x,y) = divisoresMedio n

-- (esPrimo n) se verifica si n es primo. Por ejemplo,
--    esPrimo 7  ==  True
--    esPrimo 9  ==  False
esPrimo :: Int -> Bool
esPrimo n = divisores n == [1,n]

-- (divisoresMedio n) es el par formado por los divisores medios de
-- n. Por ejemplo,
--    divisoresMedio 30  ==  (5,6)
--    divisoresMedio  7  ==  (1,7)
--    divisoresMedio 16  ==  (4,4)
divisoresMedio :: Int -> (Int,Int)
```

```
divisoresMedio n = (n `div` x,x)
  where xs = divisores n
        x  = xs !! (length xs `div` 2)

-- (divisores n) es la lista de los divisores de n. Por ejemplo,
--    divisores 30  ==  [1,2,3,5,6,10,15,30]
divisores :: Int -> [Int]
divisores n = [x | x <- [1..n], n `rem` x == 0]

-- 2ª solución
-- ===========

arbolFactorizacion2 :: Int -> Arbol
arbolFactorizacion2 n
  | x == 1    = H n
  | otherwise = N n (arbolFactorizacion2 x) (arbolFactorizacion2 y)
  where (x,y) = divisoresMedio n

-- (divisoresMedio2 n) es el par formado por los divisores medios de
-- n. Por ejemplo,
--    divisoresMedio2 30  ==  (5,6)
--    divisoresMedio2  7  ==  (1,7)
divisoresMedio2 :: Int -> (Int,Int)
divisoresMedio2 n = (n `div` x,x)
  where m = ceiling (sqrt (fromIntegral n))
        x = head [y | y <- [m..n], n `rem` y == 0]

-- Comprobación de equivalencia
-- ============================

-- La propiedad es
prop_arbolFactorizacion :: Int -> Property
prop_arbolFactorizacion n =
  n > 1 ==> arbolFactorizacion1 n == arbolFactorizacion2 n

-- La comprobación es
--    λ> quickCheck prop_arbolFactorizacion
--    +++ OK, passed 100 tests; 162 discarded.
```

5.29.2. En Python

```
# ---------------------------------------------------------------------
# Los divisores medios de un número son los que ocupan la posición
# media entre los divisores de n, ordenados de menor a mayor. Por
# ejemplo, los divisores de 60 son [1,2,3,4,5,6,10,12,15,20,30,60] y
# sus divisores medios son 6 y 10. Para los números que son cuadrados
# perfectos, sus divisores medios de son sus raíces cuadradas; por
# ejemplos, los divisores medios de 9 son 3 y 3.
#
# El árbol de factorización de un número compuesto n se construye de la
# siguiente manera:
#    * la raíz es el número n,
#    * la rama izquierda es el árbol de factorización de su divisor
#      medio menor y
#    * la rama derecha es el árbol de factorización de su divisor
#      medio mayor
# Si el número es primo, su árbol de factorización sólo tiene una hoja
# con dicho número. Por ejemplo, el árbol de factorización de 60 es
#        60
#       /  \
#      6    10
#     / \   / \
#    2   3 2   5
#
# Definir la función
#    arbolFactorizacion :: Int -> Arbol
# tal que (arbolFactorizacion n) es el árbol de factorización de n. Por
# ejemplo,
#    arbolFactorizacion(60) == N(60, N(6, H(2), H(3)), N(10, H(2), H(5)))
#    arbolFactorizacion(45) == N(45, H(5), N(9, H(3), H(3)))
#    arbolFactorizacion(7)  == H(7)
#    arbolFactorizacion(9)  == N(9, H(3), H(3))
#    arbolFactorizacion(14) == N(14, H(2), H(7))
#    arbolFactorizacion(28) == N(28, N(4, H(2), H(2)), H(7))
#    arbolFactorizacion(84) == N(84, H(7), N(12, H(3), N(4, H(2), H(2))))
# ---------------------------------------------------------------------

from dataclasses import dataclass
from math import ceil, sqrt
```

```python
from hypothesis import given
from hypothesis import strategies as st

# 1ª solución
# ===========

@dataclass
class Arbol:
    pass

@dataclass
class H(Arbol):
    x: int

@dataclass
class N(Arbol):
    x: int
    i: Arbol
    d: Arbol

# divisores(n) es la lista de los divisores de n. Por ejemplo,
#    divisores(30)  ==  [1,2,3,5,6,10,15,30]
def divisores(n: int) -> list[int]:
    return [x for x in range(1, n + 1) if n % x == 0]

# divisoresMedio(n) es el par formado por los divisores medios de
# n. Por ejemplo,
#    divisoresMedio(30)  ==  (5,6)
#    divisoresMedio(7)   ==  (1,7)
#    divisoresMedio(16)  ==  (4,4)
def divisoresMedio(n: int) -> tuple[int, int]:
    xs = divisores(n)
    x = xs[len(xs) // 2]
    return (n // x, x)

# esPrimo(n) se verifica si n es primo. Por ejemplo,
#    esPrimo(7)  ==  True
#    esPrimo(9)  ==  False
def esPrimo(n: int) -> bool:
    return divisores(n) == [1, n]
```

```
def arbolFactorizacion1(n: int) -> Arbol:
    if esPrimo(n):
        return H(n)
    (x, y) = divisoresMedio(n)
    return N(n, arbolFactorizacion1(x), arbolFactorizacion1(y))

# 2ª solución
# ===========

# divisoresMedio2(n) es el par formado por los divisores medios de
# n. Por ejemplo,
#    divisoresMedio2(30) ==  (5,6)
#    divisoresMedio2(7)  ==  (1,7)
#    divisoresMedio2(16) ==  (4,4)
def divisoresMedio2(n: int) -> tuple[int, int]:
    m = ceil(sqrt(n))
    x = [y for y in range(m, n + 1) if n % y == 0][0]
    return (n // x, x)

def arbolFactorizacion2(n: int) -> Arbol:
    if esPrimo(n):
        return H(n)
    (x, y) = divisoresMedio2(n)
    return N(n, arbolFactorizacion2(x), arbolFactorizacion2(y))

# Comprobación de equivalencia
# ============================

# La propiedad es
@given(st.integers(min_value=2, max_value=200))
def test_arbolFactorizacion(n: int) -> None:
    assert arbolFactorizacion1(n) == arbolFactorizacion2(n)

# La comprobación es
#    src> poetry run pytest -q arbol_de_factorizacion.py
#    1 passed in 0.14s
```

5.30. Valor de un árbol booleano

5.30.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Se consideran los árboles con operaciones booleanas definidos por
--    data Arbol = H Bool
--               | Conj Arbol Arbol
--               | Disy Arbol Arbol
--               | Neg Arbol
--
-- Por ejemplo, los árboles
--                Conj                               Conj
--               /   \                              /   \
--              /     \                            /     \
--           Disy      Conj                     Disy      Conj
--          /   \       /  \                   /   \      /   \
--       Conj    Neg   Neg True             Conj    Neg   Neg  True
--       /  \    |      |                   /  \    |      |
--    True False False False             True False True  False
--
-- se definen por
--    ej1, ej2:: Arbol
--    ej1 = Conj (Disy (Conj (H True) (H False))
--                      (Neg (H False)))
--                (Conj (Neg (H False))
--                      (H True))
--
--    ej2 = Conj (Disy (Conj (H True) (H False))
--                      (Neg (H True)))
--                (Conj (Neg (H False))
--                      (H True))
--
-- Definir la función
--    valor :: Arbol -> Bool
-- tal que (valor ar) es el resultado de procesar el árbol realizando
-- las operaciones booleanas especificadas en los nodos. Por ejemplo,
--    valor ej1 == True
--    valor ej2 == False
-- ---------------------------------------------------------------------
```

```haskell
module Valor_de_un_arbol_booleano where

data Arbol = H Bool
           | Conj Arbol Arbol
           | Disy Arbol Arbol
           | Neg Arbol

ej1, ej2 :: Arbol
ej1 = Conj (Disy (Conj (H True) (H False))
                 (Neg (H False)))
           (Conj (Neg (H False))
                 (H True))

ej2 = Conj (Disy (Conj (H True) (H False))
                 (Neg (H True)))
           (Conj (Neg (H False))
                 (H True))

valor :: Arbol -> Bool
valor (H x)      = x
valor (Neg a)    = not (valor a)
valor (Conj i d) = valor i && valor d
valor (Disy i d) = valor i || valor d
```

5.30.2. En Python

```python
# ---------------------------------------------------------------------
# Se consideran los árboles con operaciones booleanas definidos por
#    @dataclass
#    class Arbol:
#        pass
#
#    @dataclass
#    class H(Arbol):
#        x: bool
#
#    @dataclass
#    class Conj(Arbol):
#        i: Arbol
#        d: Arbol
```

```
#
#    @dataclass
#    class Disy(Arbol):
#        i: Arbol
#        d: Arbol
#
#    @dataclass
#    class Neg(Arbol):
#        a: Arbol
#
# Por ejemplo, los árboles
#                Conj                           Conj
#               /   \                          /   \
#              /     \                        /     \
#          Disy      Conj                 Disy      Conj
#         /   \       /  \               /   \      /   \
#      Conj    Neg   Neg True         Conj    Neg   Neg  True
#      /  \     |      |              /  \     |      |
#    True False False False         True False True  False
#
# se definen por
#    ej1: Arbol = Conj(Disy(Conj(H(True), H(False)),
#                            (Neg(H(False)))),
#                       (Conj(Neg(H(False)),
#                             (H(True)))))
#
#    ej2: Arbol = Conj(Disy(Conj(H(True), H(False)),
#                            (Neg(H(True)))),
#                       (Conj(Neg(H(False)),
#                             (H(True)))))
#
# Definir la función
#    valor : (Arbol) -> bool
# tal que valor(a) es el resultado de procesar el árbol a realizando
# las operaciones booleanas especificadas en los nodos. Por ejemplo,
#    valor(ej1) == True
#    valor(ej2) == False
# ---------------------------------------------------------------------

from dataclasses import dataclass
```

```
@dataclass
class Arbol:
    pass

@dataclass
class H(Arbol):
    x: bool

@dataclass
class Conj(Arbol):
    i: Arbol
    d: Arbol

@dataclass
class Disy(Arbol):
    i: Arbol
    d: Arbol

@dataclass
class Neg(Arbol):
    a: Arbol

ej1: Arbol = Conj(Disy(Conj(H(True), H(False)),
                       (Neg(H(False)))),
                  (Conj(Neg(H(False)),
                        (H(True)))))

ej2: Arbol = Conj(Disy(Conj(H(True), H(False)),
                       (Neg(H(True)))),
                  (Conj(Neg(H(False)),
                        (H(True)))))

def valor(a: Arbol) -> bool:
    match a:
        case H(x):
            return x
        case Neg(b):
            return not valor(b)
```

```
        case Conj(i, d):
            return valor(i) and valor(d)
        case Disy(i, d):
            return valor(i) or valor(d)
    assert False
```

5.31. Valor de una expresión aritmética básica

5.31.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Las expresiones aritméticas básicas pueden representarse usando el
-- siguiente tipo de datos
--    data Expr = C Int
--              | S Expr Expr
--              | P Expr Expr
-- Por ejemplo, la expresión 2*(3+7) se representa por
--    P (C 2) (S (C 3) (C 7))
--
-- Definir la función
--    valor :: Expr -> Int
-- tal que (valor e) es el valor de la expresión aritmética e. Por
-- ejemplo,
--    valor (P (C 2) (S (C 3) (C 7)))  ==  20
-- ---------------------------------------------------------------------

module Valor_de_una_expresion_aritmetica_basica where

data Expr = C Int
          | S Expr Expr
          | P Expr Expr

valor :: Expr -> Int
valor (C x)   = x
valor (S x y) = valor x + valor y
valor (P x y) = valor x * valor y
```

5.31.2. En Python

```
# ---------------------------------------------------------------------
# Las expresiones aritméticas básicas pueden representarse usando el
# siguiente tipo de datos
#    @dataclass
#    class Expr:
#        pass
#
#    @dataclass
#    class C(Expr):
#        x: int
#
#    @dataclass
#    class S(Expr):
#        x: Expr
#        y: Expr
#
#    @dataclass
#    class P(Expr):
#        x: Expr
#        y: Expr
# Por ejemplo, la expresión 2*(3+7) se representa por
#    P(C(2), S(C(3), C(7)))
#
# Definir la función
#    valor : (Expr) -> int:
# tal que valor(e) es el valor de la expresión aritmética e. Por
# ejemplo,
#    valor(P(C(2), S(C(3), C(7))))  ==  20
# ---------------------------------------------------------------------

from dataclasses import dataclass


@dataclass
class Expr:
    pass

@dataclass
class C(Expr):
```

```python
    x: int

@dataclass
class S(Expr):
    x: Expr
    y: Expr

@dataclass
class P(Expr):
    x: Expr
    y: Expr

def valor(e: Expr) -> int:
    match e:
        case C(x):
            return x
        case S(x, y):
            return valor(x) + valor(y)
        case P(x, y):
            return valor(x) * valor(y)
    assert False
```

5.32. Aplicación de una función a una expresión aritmética

5.32.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Las expresiones aritméticas básicas pueden representarse usando el
-- siguiente tipo de datos
--    data Expr = C Int
--              | S Expr Expr
--              | P Expr Expr
--      deriving (Show, Eq)
-- Por ejemplo, la expresión 2*(3+7) se representa por
--    P (C 2) (S (C 3) (C 7))
--
-- Definir la función
--    aplica :: (Int -> Int) -> Expr -> Expr
```

```
-- tal que (aplica f e) es la expresión obtenida aplicando la función f
-- a cada uno de los números de la expresión e. Por ejemplo,
--    λ> aplica (+2) (S (P (C 3) (C 5)) (P (C 6) (C 7)))
--    S (P (C 5) (C 7)) (P (C 8) (C 9))
--    λ> aplica (*2) (S (P (C 3) (C 5)) (P (C 6) (C 7)))
--    S (P (C 6) (C 10)) (P (C 12) (C 14))
-- ---------------------------------------------------------------------

module Aplicacion_de_una_funcion_a_una_expresion_aritmetica where

data Expr = C Int
          | S Expr Expr
          | P Expr Expr
  deriving (Show, Eq)

aplica :: (Int -> Int) -> Expr -> Expr
aplica f (C x)     = C (f x)
aplica f (S e1 e2) = S (aplica f e1) (aplica f e2)
aplica f (P e1 e2) = P (aplica f e1) (aplica f e2)
```

5.32.2. En Python

```
# ---------------------------------------------------------------------
# Las expresiones aritméticas básicas pueden representarse usando el
# siguiente tipo de datos
#    @dataclass
#    class Expr:
#        pass
#
#    @dataclass
#    class C(Expr):
#        x: int
#
#    @dataclass
#    class S(Expr):
#        x: Expr
#        y: Expr
#
#    @dataclass
#    class P(Expr):
```

```
#        x: Expr
#        y: Expr
# Por ejemplo, la expresión 2*(3+7) se representa por
#    P(C(2), S(C(3), C(7)))
#
# Definir la función
#    aplica : (Callable[[int], int], Expr) -> Expr
# tal que aplica(f, e) es la expresión obtenida aplicando la función f
# a cada uno de los números de la expresión e. Por ejemplo,
#    >>> aplica(lambda x: 2 + x, S(P(C(3), C(5)), P(C(6), C(7))))
#    S(P(C(5), C(7)), P(C(8), C(9)))
#    >>> aplica(lambda x: 2 * x, S(P(C(3), C(5)), P(C(6), C(7))))
#    S(P(C(6), C(10)), P(C(12), C(14)))
# ---------------------------------------------------------------------

from dataclasses import dataclass
from typing import Callable


@dataclass
class Expr:
    pass

@dataclass
class C(Expr):
    x: int

@dataclass
class S(Expr):
    x: Expr
    y: Expr

@dataclass
class P(Expr):
    x: Expr
    y: Expr

def aplica(f: Callable[[int], int], e: Expr) -> Expr:
    match e:
        case C(x):
```

```
            return C(f(x))
        case S(x, y):
            return S(aplica(f, x), aplica(f, y))
        case P(x, y):
            return P(aplica(f, x), aplica(f, y))
    assert False
```

Apéndices

Apéndice A

Método de Pólya para la resolución de problemas

A.1. Método de Pólya para la resolución de problemas matemáticos

Para resolver un problema se necesita:

Paso 1: Entender el problema

- ¿Cuál es la incógnita?, ¿Cuáles son los datos?
- ¿Cuál es la condición? ¿Es la condición suficiente para determinar la incógnita? ¿Es insuficiente? ¿Redundante? ¿Contradictoria?

Paso 2: Configurar un plan

- ¿Te has encontrado con un problema semejante? ¿O has visto el mismo problema planteado en forma ligeramente diferente?
- ¿Conoces algún problema relacionado con éste? ¿Conoces algún teorema que te pueda ser útil? Mira atentamente la incógnita y trata de recordar un problema que sea familiar y que tenga la misma incógnita o una incógnita similar.
- He aquí un problema relacionado al tuyo y que ya has resuelto ya. ¿Puedes utilizarlo? ¿Puedes utilizar su resultado? ¿Puedes emplear su método? ¿Te hace falta introducir algún elemento auxiliar a fin de poder utilizarlo?

- ¿Puedes enunciar al problema de otra forma? ¿Puedes plantearlo en forma diferente nuevamente? Recurre a las definiciones.
- Si no puedes resolver el problema propuesto, trata de resolver primero algún problema similar. ¿Puedes imaginarte un problema análogo un tanto más accesible? ¿Un problema más general? ¿Un problema más particular? ¿Un problema análogo? ¿Puede resolver una parte del problema? Considera sólo una parte de la condición; descarta la otra parte; ¿en qué medida la incógnita queda ahora determinada? ¿En qué forma puede variar? ¿Puedes deducir algún elemento útil de los datos? ¿Puedes pensar en algunos otros datos apropiados para determinar la incógnita? ¿Puedes cambiar la incógnita? ¿Puedes cambiar la incógnita o los datos, o ambos si es necesario, de tal forma que estén más cercanos entre sí?
- ¿Has empleado todos los datos? ¿Has empleado toda la condición? ¿Has considerado todas las nociones esenciales concernientes al problema?

Paso 3: Ejecutar el plan

- Al ejercutar tu plan de la solución, comprueba cada uno de los pasos
- ¿Puedes ver claramente que el paso es correcto? ¿Puedes demostrarlo?

Paso 4: Examinar la solución obtenida

- ¿Puedes verificar el resultado? ¿Puedes el razonamiento?
- ¿Puedes obtener el resultado en forma diferente? ¿Puedes verlo de golpe? ¿Puedes emplear el resultado o el método en algún otro problema?

G. Polya "Cómo plantear y resolver problemas" (Ed. Trillas, 1978) p. 19

A.2. Método de Pólya para resolver problemas de programación

Para resolver un problema se necesita:

Paso 1: Entender el problema

- ¿Cuáles son las *argumentos*? ¿Cuál es el *resultado*? ¿Cuál es *nombre* de la función? ¿Cuál es su *tipo*?
- ¿Cuál es la *especificación* del problema? ¿Puede satisfacerse la especificación? ¿Es insuficiente? ¿Redundante? ¿Contradictoria? ¿Qué restricciones se suponen sobre los argumentos y el resultado?
- ¿Puedes descomponer el problema en partes? Puede ser útil dibujar diagramas con ejemplos de argumentos y resultados.

Paso 2: Diseñar el programa

- ¿Te has encontrado con un problema semejante? ¿O has visto el mismo problema planteado en forma ligeramente diferente?
- ¿Conoces algún problema *relacionado* con éste? ¿Conoces alguna función que te pueda ser útil? Mira atentamente el tipo y trata de recordar un problema que sea familiar y que tenga el mismo tipo o un tipo similar.
- ¿Conoces algún problema familiar con una *especificación* similar?
- He aquí un problema *relacionado* al tuyo y que ya has resuelto. ¿Puedes utilizarlo? ¿Puedes utilizar su resultado? ¿Puedes emplear su método? ¿Te hace falta introducir alguna función auxiliar a fin de poder utilizarlo?
- Si no puedes resolver el problema propuesto, trata de resolver primero algún problema similar. ¿Puedes imaginarte un problema análogo un tanto más *accesible*? ¿Un problema más *general*? ¿Un problema más *particular*? ¿Un problema *análogo*?
- ¿Puede resolver una *parte* del problema? ¿Puedes deducir algún elemento útil de los datos? ¿Puedes pensar en algunos otros datos apropiados para determinar la incógnita? ¿Puedes cambiar la incógnita? ¿Puedes cambiar la incógnita o los datos, o ambos si es necesario, de tal forma que estén más cercanos entre sí?
- ¿Has empleado todos los datos? ¿Has empleado todas las restricciones sobre los datos? ¿Has considerado todas los requisitos de la especificación?

Paso 3: Escribir el programa

- Al escribir el programa, comprueba cada uno de los pasos y funciones auxiliares.
- ¿Puedes ver claramente que cada paso o función auxiliar es correcta?
- Puedes escribir el programa en *etapas*. Piensas en los diferentes *casos* en los que se divide el problema; en particular, piensas en los diferentes casos para los datos. Puedes pensar en el cálculo de los casos independientemente y *unirlos* para obtener el resultado final
- Puedes pensar en la solución del problema descomponiéndolo en problemas con datos más simples y uniendo las soluciones parciales para obtener la solución del problema; esto es, por *recursión*.
- En su diseño se puede usar problemas más generales o más particulares. Escribe las soluciones de estos problemas; ellas puede servir como guía para la solución del problema original, o se pueden usar en su solución.
- ¿Puedes apoyarte en otros problemas que has resuelto? ¿Pueden usarse? ¿Pueden modificarse? ¿Pueden guiar la solución del problema original?

Paso 4: Examinar la solución obtenida

- ¿Puedes comprobar el funcionamiento del programa sobre una colección de argumentos?
- ¿Puedes comprobar propiedades del programa?
- ¿Puedes escribir el programa en una forma diferente?
- ¿Puedes emplear el programa o el método en algún otro programa?

Simon Thompson *How to program it*, basado en G. Polya *Cómo plantear y resolver problemas*.

Bibliografía

[1] C. Allen, J. Moronuki, and S. Syrek. *Haskell programming from first principles*. Lorepub LLC, 2016.

[2] J. A. Alonso. Temas de programación funcional con Haskell. Technical report, Univ. de Sevilla, 2019.

[3] J. A. Alonso and als. Exámenes de programación funcional con Haskell. Technical report, Univ. de Sevilla, 2021.

[4] J. A. Alonso and M. J. Hidalgo. Piensa en Haskell (Ejercicios de programación funcional con Haskell). Technical report, Univ. de Sevilla, 2012.

[5] J. A. Alonso and M. J. Hidalgo. Ejercicios de programación funcional con Haskell. Technical report, Univ. de Sevilla, 2022.

[6] R. Bird. *Introducción a la programación funcional con Haskell*. Prentice–Hall, 1999.

[7] R. Bird. *Pearls of functional algorithm design*. Cambridge University Press, 2010.

[8] R. Bird. *Thinking functionally with Haskell*. Cambridge University Press, 2014.

[9] R. Bird and J. Gibbons. *Algorithm design with Haskell*. Cambridge University Press, 2020.

[10] A. Casamayou-Boucau, P. Chauvin, and G. Connan. *Programmation en Python pour les mathématiques*. Dunod, 2012.

[11] A. Downey, J. Elkner, and C. Meyers. *Aprenda a pensar como un programador con Python*. Green Tea Press, 2002.

[12] M. Goodrich, R. Tamassia, and M. Goldwasser. *Data structures and algorithms in Python*. Wiley, 2013.

[13] J. Guttag. *Introduction to computation and programming using python, second edition*. MIT Press, 2016.

[14] T. Hall and J. Stacey. *Python 3 for absolute beginners*. Apress, 2010.

[15] M. Hetland. *Python algorithms: Mastering basic algorithms in the Python language*. Apress, 2011.

[16] P. Hudak. The Haskell school of music (From signals to symphonies). Technical report, Yale University, 2012.

[17] J. Hunt. *A beginners guide to Python 3 programming*. Springer International Publishing, 2019.

[18] J. Hunt. *Advanced guide to Python 3 programming*. Springer International Publishing, 2019.

[19] G. Hutton. *Programming in Haskell (2nd ed.)*. Cambridge University Press, 2016.

[20] W. Kurt. *Get programming with Haskell*. Manning Publications, 2018.

[21] M. Lipovača. ¡Aprende Haskell por el bien de todos! En `http://aprendehaskell.es`.

[22] S. Lott. *Functional Python programming, 2nd Edition*. Packt Publishing, 2018.

[23] C. Okasaki. *Purely functional data structures*. Cambridge University Press, 1999.

[24] B. O'Sullivan, D. Stewart, and J. Goerzen. *Real world Haskell*. O'Reilly, 2008.

[25] T. Padmanabhan. *Programming with Python*. Springer Singapore, 2017.

[26] G. Pólya. *Cómo plantear y resolver problemas*. Editorial Trillas, 1965.

[27] F. Rabhi and G. Lapalme. *Algorithms: A functional programming approach*. Addison–Wesley, 1999.

[28] M. Rubio-Sanchez. *Introduction to recursive programming*. CRC Press, 2017.

[29] B. C. Ruiz, F. Gutiérrez, P. Guerrero, and J. Gallardo. *Razonando con Haskell (Un curso sobre programación funcional)*. Thompson, 2004.

[30] A. Saha. *Doing Math with Python: Use Programming to explore algebra, statistics, calculus, and more!* No Starch Press, 2015.

[31] Y. Sajanikar. *Haskell cookbook*. Packt Publishing, 2017.

[32] D. Sannella, M. Fourman, H. Peng, and P. Wadler. *Introduction to computation: Haskell, logic and automata*. Springer International Publishing, 2022.

[33] A. Serrano. *Beginning Haskell: A project-based approach*. Apress, 2014.

[34] N. Shukla. *Haskell data analysis cookbook*. Packt Publishing, 2014.

[35] B. Stephenson. *The Python workbook: A brief introduction with exercises and solutions*. Springer International Publishing, 2015.

[36] S. Thompson. *Haskell: The craft of functional programming*. Addison–Wesley, third edition, 2011.

[37] R. van Hattem. *Mastering Python: Write powerful and efficient code using the full range of Python's capabilities*. Packt Publishing, 2022.