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 2024

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

Parte I

Introducción a la programación con Python

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
```

```
# 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

```
-- ---------------------------------------------------------------------
-- 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

```
# ---------------------------------------------------------------------
# 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, y, z)
```

1.7. El primero al final

En Haskell

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

{-# 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()))
```

```python
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

```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

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

```
-- 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

```
# ---------------------------------------------------------------------
# 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

```
-- ---------------------------------------------------------------------
-- 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

```
# ---------------------------------------------------------------------
# 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

```
-- ---------------------------------------------------------------------
-- 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

```
# ---------------------------------------------------------------------
# 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

```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')
```

```python
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

```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.
# ---------------------------------------------------------------------
```

```
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

```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

```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)
```

```
    where xs = raices a b c

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

En 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,
```

```
#    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

```
-- ---------------------------------------------------------------------
-- 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]
```

```haskell
-- 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

```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 []
```

```python
# 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

```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
```

```haskell
--      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

```
# ---------------------------------------------------------------------
# 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 lo sucesivo, usaremos la 4ª definición
subconjunto :: Ord a => [a] -> [a] -> Bool
subconjunto = subconjunto4
```

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
```

```haskell
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

```
# ---------------------------------------------------------------------
# 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
# ---------------------------------------------------------------------

# pylint: disable=arguments-out-of-order

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

```haskell
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]
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]  ==  []
```

```haskell
-- ---------------------------------------------------------------------

{-# 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]
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
```

```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('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

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

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 =
```

```haskell
  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
# ---------------------------------------------------------------------

# pylint: disable=unused-import

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)))

# 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)))
```

```python
# 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

# 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]
```

```haskell
--    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]
```

```python
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]
#    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
```

```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")

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

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

```haskell
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

```
# ---------------------------------------------------------------------
# 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
# ---------------------------------------------------------------------

# pylint: disable=unused-import

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]
```

```python
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

# 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

```
-- ---------------------------------------------------------------------
-- 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]
```

```haskell
-- 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

```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
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

```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 #-}

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
# ===========
```

```python
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
-- 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 Suma_de_los_cuadrados_de_los_primeros_numeros_naturales
  (sumaDeCuadrados1, sumaDeCuadrados2, sumaDeCuadrados3, sumaDeCuadrados4, sumaDe
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]

-- 2ª solución
-- ===========

euler6b :: Integer -> Integer
euler6b n = suma2 n ^ 2 - sumaDeCuadrados2 n

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

-- 3ª solución
-- ===========

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

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

-- 4ª solución
-- ===========

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

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

-- 5ª solución
-- ===========

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

suma5 :: Integer -> Integer
```

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

from src.suma_de_los_cuadrados_de_los_primeros_numeros_naturales import (
    sumaDeCuadrados1, sumaDeCuadrados2, sumaDeCuadrados3, sumaDeCuadrados4,
    sumaDeCuadrados5, sumaDeCuadrados6)
```

```
setrecursionlimit(10**6)

# 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))

# 2ª solución
# ===========

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

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

# 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))
```

```
# 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

# 6ª solución
# ===========

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

# 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

```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]
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))
```

```haskell
--    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
```

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

```python
@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

```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,
--    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
```

```haskell
-- 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

```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
```

```haskell
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 =
```

```haskell
  [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

```
# ---------------------------------------------------------------------
# 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
```

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

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

```
-- ---------------------------------------------------------------------
-- 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
```

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

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
-- ---------------------------------------------------------------------

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
```

```haskell
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 =
```

```haskell
  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

```
-- ---------------------------------------------------------------------
-- 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
```

```haskell
-- 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]
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]
```

```python
#    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]
-- ---------------------------------------------------------------------
```

```haskell
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
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..]
```

```haskell
-- 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

```
-- ---------------------------------------------------------------------
-- 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
```

```haskell
--    euler1 (10^30) == 2333333333333333333333333333331666666666666666666666666
--
-- 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
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
-- ============================
```

```haskell
-- 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

-- 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 =
```

```haskell
  [(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)
```

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

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

```
-- ---------------------------------------------------------------------
-- 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
-- ---------------------------------------------------------------------
```

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

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)
--    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

```
# ---------------------------------------------------------------------
# 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
# =============================
```

```python
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,
```

```haskell
--      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)
--    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]:
```

```python
    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

```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

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)]
-- ---------------------------------------------------------------------

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

```
# ---------------------------------------------------------------------
# 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

```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 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
```

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

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

-- 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]

-- 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
```

```haskell
-- 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

-- 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
```

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

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),
--                  ("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
```

```haskell
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),
            ("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']
# ---------------------------------------------------------------------

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
```

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

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

```
# ---------------------------------------------------------------------
# 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
```

```python
#
# 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

```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]
# ---------------------------------------------------------------------

# pylint: disable=unused-import

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
# ===========

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
```

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

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
# ---------------------------------------------------------------------

# pylint: disable=unused-import

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
```

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

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
# =========================

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

```
-- ---------------------------------------------------------------------
-- 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
```

```haskell
--    listaNumero [0,0,1]    == 1
-- ---------------------------------------------------------------------

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

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)

# 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

```
-- ---------------------------------------------------------------------
-- 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
```

```
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)
```

```haskell
--    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

```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))')
```

```
#    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

```
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
```

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

-- 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

```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,
```

```
--   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
```

```
# 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)
```

```python
# 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
```

```
# 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

```
-- ---------------------------------------------------------------------
-- 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

```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
```

```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('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

```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:
```

```python
        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

```
-- ---------------------------------------------------------------------
-- 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

```
-- ---------------------------------------------------------------------
-- 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

```haskell
--    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

```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
```

```python
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

```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

```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

```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]
```

```
# 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
```

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

```
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

```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:
    def f(x: int) -> int:
        return x + 4
    def p(x: int) -> bool:
        return 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
-- ===========
```

```haskell
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:
```

```
    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
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
```

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

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:])

# 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

```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
```

```python
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

```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)
--      Suc (Suc (Suc Cero))
--      λ> nat2int (suma (Suc (Suc Cero)) (Suc Cero))
--      3
--      λ> nat2int (suma (int2nat 2) (int2nat 1))
--      3
-- ---------------------------------------------------------------------
```

```haskell
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

```
-- ---------------------------------------------------------------------
-- 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 con valores en los nodos y en las hojas

5.5.1. En Haskell

```
-- El árbol binario
--         9
--        / \
--       /   \
--      3     7
--     / \
--    2   4
-- se puede representar por
--    N 9 (N 3 (H 2) (H 4)) (H 7)
-- usando el tipo de los árboles binarios definido como se muestra a
-- continuación.

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

module Arboles_binarios where

import Test.QuickCheck

data Arbol a = H a
             | N a (Arbol a) (Arbol a)
  deriving (Show, Eq)

-- Generador de árboles binarios
-- =============================

-- (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
```

5.5.2. En Python

```
# El árbol binario
#        9
#       / \
#      /   \
#     3     7
#    / \
#   2   4
# se puede representar por
#   N(9, N(3, H(2), H(4)), H(7))
# usando la definición de los árboles binarios que se muestra a
# continuación.

from dataclasses import dataclass
from random import choice, randint
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]

# Generador de árboles
# ====================

# 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))])
```

5.6. Pertenencia de un elemento a un árbol

5.6.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Usando el [tipo de los árboles binarios](https://bit.ly/3H53exA),
-- definir la función
--    pertenece :: Eq t => t -> Arbol t -> Bool
-- tal que (pertenece m a) se verifica si m pertenece en el árbol a. Por
-- ejemplo,
--    λ> pertenece 4 (N 5 (N 3 (H 1) (H 4)) (N 7 (H 6) (H 9)))
--    True
--    λ> pertenece 0 (N 5 (N 3 (H 1) (H 4)) (N 7 (H 6) (H 9)))
--    False
-- ---------------------------------------------------------------------

module Pertenencia_de_un_elemento_a_un_arbol where

import Arboles_binarios (Arbol (..))

pertenece :: Eq t => t -> Arbol t -> Bool
pertenece m (H n)     = m == n
pertenece m (N n i d) = m == n || pertenece m i || pertenece m d
```

5.6.2. En Python

```python
# ---------------------------------------------------------------------
# Usando el [tipo de los árboles binarios](https://bit.ly/3H53exA),
# definir la función
#    pertenece : (A, Arbol[A]) -> bool
# tal que pertenece(m, a) se verifica si m pertenece en el árbol a. Por
# ejemplo,
#    >>> pertenece(4, N(5, N(3, H(1), H(4)), N(7, H(6), (H(9)))))
#    True
#    >>> pertenece(0, N(5, N(3, H(1), H(4)), N(7, H(6), (H(9)))))
#    False
# ---------------------------------------------------------------------

from typing import TypeVar
```

```python
from src.arboles_binarios import Arbol, H, N

A = TypeVar("A")

def pertenece(m: A, a: Arbol[A]) -> bool:
    match a:
        case H(n):
            return m == n
        case N(n, i, d):
            return m == n or pertenece(m, i) or pertenece(m, d)
    assert False
```

5.7. Aplanamiento de un árbol

5.7.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Usando el [tipo de los árboles binarios](https://bit.ly/3H53exA),
-- definir la función
--    aplana :: Arbol t -> [t]
-- tal que (aplana a) es la lista obtenida aplanando el árbol a. Por
-- ejemplo,
--    λ> aplana (N 5 (N 3 (H 1) (H 4)) (N 7 (H 6) (H 9)))
--    [1,3,4,5,6,7,9]
-- ---------------------------------------------------------------------

module Aplanamiento_de_un_arbol where

import Arboles_binarios (Arbol (..))

aplana :: Arbol t -> [t]
aplana (H n)     = [n]
aplana (N n i d) = aplana i ++ [n] ++ aplana d
```

5.7.2. En Python

```python
# ---------------------------------------------------------------------
# Usando el [tipo de los árboles binarios](https://bit.ly/3H53exA),
# definir la función
```

```
#    aplana :: Arbol t -> [t]
# tal que (aplana a) es la lista obtenida aplanando el árbol a. Por
# ejemplo,
#    >>> aplana (N(5, N(3, H(1), H(4)), N(7, H(6), (H(9)))))
#    [1, 3, 4, 5, 6, 7, 9]
# ---------------------------------------------------------------------

from typing import TypeVar

from src.arboles_binarios import Arbol, H, N

A = TypeVar("A")

def aplana(a: Arbol[A]) -> list[A]:
    match a:
        case H(n):
            return [n]
        case N(n, i, d):
            return aplana(i) + [n] + aplana(d)
    assert False
```

5.8. Número de hojas de un árbol binario

En Haskell

```
-- ---------------------------------------------------------------------
-- Usando el [tipo de los árboles binarios](https://bit.ly/3H53exA),
-- 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.
-- ---------------------------------------------------------------------
```

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

module Numero_de_hojas_de_un_arbol_binario where

import Arboles_binarios (Arbol (..))
import Test.QuickCheck

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
-- ============================

-- 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
# ---------------------------------------------------------------------
# Usando el [tipo de los árboles binarios](https://bit.ly/3H53exA),
# 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
#
```

```python
# Comprobar con Hypothesis que en todo árbol binario el número de sus
# hojas es igual al número de sus nodos más uno.
# ---------------------------------------------------------------------

from typing import TypeVar

from hypothesis import given
from hypothesis import strategies as st

from src.arboles_binarios import Arbol, H, N, arbolArbitrario

A = TypeVar("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
# ============================

# 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
```

5.9. Profundidad de un árbol binario

5.9.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Usando el [tipo de los árboles binarios](https://bit.ly/3H53exA),
-- definir las funciones
--    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 Arboles_binarios (Arbol (..))
import Numero_de_hojas_de_un_arbol_binario (nNodos)

import Test.QuickCheck

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_nNodosProfundidad :: Arbol Int -> Bool
prop_nNodosProfundidad x =
  nNodos x <= 2 ^ profundidad x - 1

-- La comprobación es
```

```
--    λ> quickCheck prop_nNodosProfundidad
--    OK, passed 100 tests.
```

En Python

```python
# ---------------------------------------------------------------------
# Usando el [tipo de los árboles binarios](https://bit.ly/3H53exA),
# 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 typing import TypeVar

from hypothesis import given
from hypothesis import strategies as st

from src.arboles_binarios import Arbol, H, N, arbolArbitrario
from src.numero_de_hojas_de_un_arbol_binario import nNodos

A = TypeVar("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
# ============================

# 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
```

5.10. Recorrido de árboles binarios

5.10.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Usando el [tipo de los árboles binarios](https://bit.ly/3H53exA),
-- 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,
--      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 Arboles_binarios (Arbol (..))
```

```haskell
import Numero_de_hojas_de_un_arbol_binario (nNodos, nHojas)
import Test.QuickCheck

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
-- ============================

-- 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

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

5.10.2. En Python

```python
# ---------------------------------------------------------------------
# Usando el [tipo de los árboles binarios](https://bit.ly/3H53exA),
# 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
#   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 typing import TypeVar

from hypothesis import given
from hypothesis import strategies as st

from src.arboles_binarios import Arbol, H, N, arbolArbitrario
from src.numero_de_hojas_de_un_arbol_binario import nHojas, nNodos

A = TypeVar("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
# ============================
```

```python
# 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
```

5.11. Imagen especular de un árbol binario

5.11.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Usando el [tipo de los árboles binarios](https://bit.ly/3H53exA),
-- 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 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 Arboles_binarios (Arbol (..))
import Recorrido_de_arboles_binarios (preorden, postorden)
import Test.QuickCheck

espejo :: Arbol a -> Arbol a
espejo (H x)     = H x
```

```haskell
espejo (N x i d) = N x (espejo d) (espejo i)

-- 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.
```

5.11.2. En Python

```python
# ---------------------------------------------------------------------
# Usando el [tipo de los árboles binarios](https://bit.ly/3H53exA),
# 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 typing import TypeVar

from hypothesis import given
from hypothesis import strategies as st

from src.arboles_binarios import Arbol, H, N, arbolArbitrario
from src.recorrido_de_arboles_binarios import postorden, preorden
```

```python
A = TypeVar("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

# 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
```

5.12. Subárbol de profundidad dada

5.12.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- La función take está definida por
--    take :: Int -> [a] -> [a]
--    take 0            = []
--    take (n+1) []     = []
--    take (n+1) (x:xs) = x : take n xs
--
-- Usando el [tipo de los árboles binarios](https://bit.ly/3H53exA),
-- definir la función
--    takeArbol ::  Int -> Arbol a -> Arbol a
-- tal que (takeArbol n t) es el subárbol de t de profundidad n. Por
```

```haskell
-- 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 Arboles_binarios (Arbol (..))
import Profundidad_de_un_arbol_binario (profundidad)
import Test.QuickCheck

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)

-- 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.
```

5.12.2. En Python

```python
# ---------------------------------------------------------------------
# Usando el [tipo de los árboles binarios](https://bit.ly/3H53exA),
# 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 typing import TypeVar

from hypothesis import given
from hypothesis import strategies as st

from src.arboles_binarios import Arbol, H, N, arbolArbitrario
from src.profundidad_de_un_arbol_binario import profundidad

A = TypeVar("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

# 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
```

5.13. Árbol de profundidad n con nodos iguales

5.13.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Usando el [tipo de los árboles binarios](https://bit.ly/3H53exA),
-- 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 Arboles_binarios (Arbol (..))
import Numero_de_hojas_de_un_arbol_binario (nHojas)
```

```haskell
import Test.QuickCheck

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)

-- 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.
```

5.13.2. En Python

```python
# ---------------------------------------------------------------------
# Usando el [tipo de los árboles binarios](https://bit.ly/3H53exA),
# 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)
#    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 typing import TypeVar

from hypothesis import given
from hypothesis import strategies as st

from src.arboles_binarios import Arbol, H, N
from src.numero_de_hojas_de_un_arbol_binario import nHojas

A = TypeVar("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

# 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
```

5.14. Árboles con igual estructura

5.14.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Usaremos el [tipo de los árboles binarios con valores en los nodos y
-- en las hojas](https://bit.ly/3H53exA).
--
-- 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

import Arboles_binarios (Arbol (H, N))

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
```

5.14.2. En Python

```
# ---------------------------------------------------------------------
# Usaremos el [tipo de los árboles binarios](https://bit.ly/3H53exA).
#
# 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 typing import TypeVar

from src.arboles_binarios import Arbol, H, N
```

```
A = TypeVar("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
```

5.15. Existencia de elementos del árbol que verifican una propiedad

5.15.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Usando el [tipo de los árboles binarios con valores en los nodos y
-- en las hojas](https://bit.ly/3H53exA), 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

import Arboles_binarios (Arbol (H, N))

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
```

5.15.2. En Python

```
# ---------------------------------------------------------------------
# Usando el [tipo de los árboles binarios](https://bit.ly/3H53exA),
# 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 typing import Callable, TypeVar

from src.arboles_binarios import Arbol, H, N

A = TypeVar("A")

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
```

5.16. Elementos del nivel k de un árbol

5.16.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Un elemento de un árbol se dirá de nivel k si aparece en el árbol a
-- distancia k  de la raíz.
--
-- Usando el [tipo de los árboles binarios con valores en los nodos y
```

```haskell
-- en las hojas](https://bit.ly/3H53exA), 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

import Arboles_binarios (Arbol (H, N))

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
```

5.16.2. En Python

```python
# ---------------------------------------------------------------------
# Usando el [tipo de los árboles binarios](https://bit.ly/3H53exA),
# 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)))
#     []
# ---------------------------------------------------------------------
```

```python
from typing import TypeVar

from src.arboles_binarios import Arbol, H, N

A = TypeVar("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
```

5.17. El tipo de los árboles binarios con valores en las hojas

5.17.1. 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))
-- usando el tipo de los árboles binarios con valores en las hojas
-- definido como se muestra a continuación.

module Arbol_binario_valores_en_hojas where

import Test.QuickCheck
```

```haskell
data Arbol a = Hoja a
             | Nodo (Arbol a) (Arbol a)
  deriving (Eq, Show)

-- (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]
```

5.17.2. En Python

```python
# El árbol binario
#          ·
#         / \
#        /   \
#       ·     ·
#      / \   / \
#     1   4 6   9
```

```
# se puede representar por
#    ejArbol = Nodo(Nodo(Hoja(1), Hoja(4)),
#                   Nodo(Hoja(6), Hoja(9)))
# usando el tipo de los árboles binarios con valores en las hojas
# definido como se muestra a continuación.

from dataclasses import dataclass
from random import randint
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]

# Generador
# =========

# arbolArbitrario(n) es un árbol aleatorio de orden n. Por ejemplo,
#    >>> arbolArbitrario(2)
#    Nodo(i=Nodo(i=Hoja(x=6), d=Hoja(x=3)), d=Nodo(i=Hoja(x=4), d=Hoja(x=4)))
#    >>> arbolArbitrario(2)
#    Nodo(i=Nodo(i=Hoja(x=9), d=Hoja(x=6)), d=Nodo(i=Hoja(x=9), d=Hoja(x=8)))
def arbolArbitrario(n: int) -> Arbol[int]:
    if n == 0:
        return Hoja(randint(1, 10))
    if n == 1:
        return Nodo(arbolArbitrario(0), arbolArbitrario(0))
    k = min(randint(1, n + 1), n - 1)
    return Nodo(arbolArbitrario(k), arbolArbitrario(n - k))
```

5.18. Altura de un árbol binario

En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Usando el [tipo de los árboles binarios con los valores en las hojas]
-- (https://bit.ly/3N5RuyE), 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

import Arbol_binario_valores_en_hojas (Arbol (..))

altura :: Arbol a -> Int
altura (Hoja _)   = 0
altura (Nodo i d) = 1 + max (altura i) (altura d)
```

En Python

```python
# ---------------------------------------------------------------------
# Usando el [tipo de los árboles binarios con los valores en las hojas]
# (https://bit.ly/3N5RuyE), 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 typing import TypeVar

from src.arbol_binario_valores_en_hojas import Arbol, Hoja, Nodo

A = TypeVar("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
```

5.19. Aplicación de una función a un árbol

En Haskell

```
-- ---------------------------------------------------------------------
-- Usando el [tipo de los árboles binarios con los valores en las hojas]
-- (https://bit.ly/3N5RuyE), 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

import Arbol_binario_valores_en_hojas (Arbol (..))

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

```
# ---------------------------------------------------------------------
# Usando el [tipo de los árboles binarios con los valores en las hojas]
# (https://bit.ly/3N5RuyE), 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))
# ---------------------------------------------------------------------

from typing import Callable, TypeVar

from src.arbol_binario_valores_en_hojas import Arbol, Hoja, Nodo

A = TypeVar("A")
B = TypeVar("B")

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
```

5.20. Árboles con la misma forma

En Haskell

```
-- ---------------------------------------------------------------------
-- Usando el [tipo de los árboles binarios con los valores en las hojas]
-- (https://bit.ly/3N5RuyE), 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
```

```haskell
--     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 Arbol_binario_valores_en_hojas (Arbol (..))
import Aplicacion_de_una_funcion_a_un_arbol (mapArbol)
import Test.QuickCheck

-- 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 ())

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

-- 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

```
# ---------------------------------------------------------------------
# Usando el [tipo de los árboles binarios con los valores en las hojas]
# (https://bit.ly/3N5RuyE), 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 typing import TypeVar

from hypothesis import given
from hypothesis import strategies as st

from src.aplicacion_de_una_funcion_a_un_arbol import mapArbol
from src.arbol_binario_valores_en_hojas import (Arbol, Hoja, Nodo,
                                                arbolArbitrario)

A = TypeVar("A")
B = TypeVar("B")

# -- 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
# -- ===========

def mismaForma2(a: Arbol[A], b: Arbol[B]) -> bool:
    return mapArbol(lambda x: 0, a) == mapArbol(lambda x: 0, b)

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

# 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
```

5.21. Árboles con bordes iguales

5.21.1. En Haskell

```
-- ---------------------------------------------------------------------
-- En este ejercicio se usará el [tipo de los árboles binarios con los
-- valores en las hojas](https://bit.ly/3N5RuyE).
--
-- 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 = Nodo (Hoja 1) (Nodo (Hoja 2) (Hoja 3))
--    arbol2 = Nodo (Nodo (Hoja 1) (Hoja 2)) (Hoja 3)
--    arbol3 = Nodo (Nodo (Hoja 1) (Hoja 4)) (Hoja 3)
--    arbol4 = Nodo (Nodo (Hoja 2) (Hoja 3)) (Hoja 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
-- ---------------------------------------------------------------------

module Arboles_con_bordes_iguales where

import Arbol_binario_valores_en_hojas (Arbol (Hoja, Nodo))

arbol1, arbol2, arbol3, arbol4 :: Arbol Int
arbol1 = Nodo (Hoja 1) (Nodo (Hoja 2) (Hoja 3))
arbol2 = Nodo (Nodo (Hoja 1) (Hoja 2)) (Hoja 3)
arbol3 = Nodo (Nodo (Hoja 1) (Hoja 4)) (Hoja 3)
arbol4 = Nodo (Nodo (Hoja 2) (Hoja 3)) (Hoja 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 (Nodo i d) = borde i ++ borde d
borde (Hoja x)   = [x]
```

5.21.2. En Python

```
# ---------------------------------------------------------------------
# Usaremos el [tipo de los árboles binarios con los valores en las
# hojas](https://bit.ly/3N5RuyE).
#
# 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 typing import TypeVar

from src.arbol_binario_valores_en_hojas import Arbol, Hoja, Nodo

A = TypeVar("A")

arbol1: Arbol[int] = Nodo(Hoja(1), Nodo(Hoja(2), Hoja(3)))
arbol2: Arbol[int] = Nodo(Nodo(Hoja(1), Hoja(2)), Hoja(3))
arbol3: Arbol[int] = Nodo(Nodo(Hoja(1), Hoja(4)), Hoja(3))
arbol4: Arbol[int] = Nodo(Nodo(Hoja(2), Hoja(3)), Hoja(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 Hoja(x):
            return [x]
        case Nodo(i, d):
            return borde(i) + borde(d)
    assert False

def igualBorde(t1: Arbol[A], t2: Arbol[A]) -> bool:
    return borde(t1) == borde(t2)
```

5.22. Árbol con las hojas en la profundidad dada

En Haskell

```
-- ---------------------------------------------------------------------
-- Usando el [tipo de los árboles binarios con los valores en las hojas]
-- (https://bit.ly/3N5RuyE), 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

import Arbol_binario_valores_en_hojas (Arbol (..))

creaArbol :: Int -> Arbol ()
creaArbol h
  | h <= 0    = Hoja ()
  | otherwise = Nodo x x
  where x = creaArbol (h - 1)
```

En Python

```python
# ---------------------------------------------------------------------
# Usando el [tipo de los árboles binarios con los valores en las hojas]
# (https://bit.ly/3N5RuyE), 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 typing import Any, TypeVar

from src.arbol_binario_valores_en_hojas import Arbol, Hoja, Nodo

A = TypeVar("A")

def creaArbol(h: int) -> Arbol[Any]:
    if h <= 0:
        return Hoja(None)
    x = creaArbol(h - 1)
    return Nodo(x, x)
```

5.23. El tipo de los árboles binarios con valores en los nodos

5.23.1. En Haskell

```haskell
-- El árbol, con valores en los nodos,
--            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))
-- usando el tipo de los árboles con valores en los nodos definido como
-- se muestra a continuación.

module Arbol_binario_valores_en_nodos where

data Arbol a = H
             | N a (Arbol a) (Arbol a)
  deriving (Show, Eq)
```

5.23.2. En Python

```
# El árbol binario, con valores en  los nodos,
#            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(
# usando el tipo de los árboles binarios con valores en los nodos
# definido como se muestra a continuación.

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]
```

5.24. Suma de un árbol

5.24.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Usando el [tipo de los árboles binarios con valores en los nodos]
-- (https://bit.ly/40Pplzj), definir la función
--    sumaArbol :: Num a => Arbol 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

import Arbol_binario_valores_en_nodos (Arbol (H, N))

sumaArbol :: Num a => Arbol a -> a
sumaArbol H         = 0
sumaArbol (N x i d) = x + sumaArbol i + sumaArbol d
```

5.24.2. En Python

```
# ---------------------------------------------------------------------
# Usando el [tipo de los árboles binarios con valores en los nodos]
# (https://bit.ly/40Pplzj), 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
# ---------------------------------------------------------------------
```

```python
from src.arbol_binario_valores_en_nodos import Arbol, H, N


def sumaArbol(a: Arbol[int]) -> int:
    match a:
        case H():
            return 0
        case N(x, i, d):
            return x + sumaArbol(i) + sumaArbol(d)
    assert False
```

5.25. Rama izquierda de un árbol binario

5.25.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Usando el [tipo de los árboles binarios con valores en los nodos]
-- (https://bit.ly/40Pplzj), definir la función
--    ramaIzquierda :: Arbol a -> [a]
-- 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

import Arbol_binario_valores_en_nodos (Arbol (H, N))

ramaIzquierda :: Arbol a -> [a]
ramaIzquierda H         = []
ramaIzquierda (N x i _) = x : ramaIzquierda i
```

5.25.2. En Python

```python
# ---------------------------------------------------------------------
# Usando el [tipo de los árboles binarios con valores en los nodos]
# (https://bit.ly/40Pplzj), 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 typing import TypeVar

from src.arbol_binario_valores_en_nodos import Arbol, H, N

A = TypeVar("A")

def ramaIzquierda(a: Arbol[A]) -> list[A]:
    match a:
        case H():
            return []
        case N(x, i, _):
            return [x] + ramaIzquierda(i)
    assert False
```

5.26. Árboles balanceados

5.26.1. En Haskell

```
-- ---------------------------------------------------------------------
-- 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.
--
-- Usando el [tipo de los árboles binarios con valores en los nodos]
-- (https://bit.ly/40Pplzj), 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
-- ---------------------------------------------------------------------
```

```haskell
module Arboles_balanceados where

import Arbol_binario_valores_en_nodos (Arbol (H, N))

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
```

5.26.2. En Python

```python
# ---------------------------------------------------------------------
# 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.
#
# Usando el [tipo de los árboles binarios con valores en los nodos]
# (https://bit.ly/40Pplzj), 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 typing import TypeVar

from src.arbol_binario_valores_en_nodos import Arbol, H, N

A = TypeVar(”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
```

5.27. Árbol de factorización

5.27.1. 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
```

```haskell
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.
```

5.27.2. 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

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)
```

```python
    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
```

5.28. Valor de un árbol booleano

5.28.1. 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
```

```haskell
--    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
-- ---------------------------------------------------------------------

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
```

5.28.2. 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
```

5.29. El tipo de las fórmulas proposicionales

5.29.1. En Haskell

```
-- La fórmula A → ⊥ ∧ ¬B se representa por
--    Impl(Var('A'), Conj(Const(False), Neg (Var('B'))))
-- usando el tipo de las fórmulas proposicionales definido como se
-- muestra a continuación.

module Tipo_de_formulas where

data FProp = Const Bool
           | Var Char
           | Neg FProp
           | Conj FProp FProp
           | Impl FProp FProp
  deriving Show
```

5.29.2. En Python

```
# La fórmula A → ⊥ ∧ ¬B se representa por
#    Impl(Var('A'), Conj(Const(False), Neg (Var('B'))))
# usando el tipo de las fórmulas proposicionales definido como se
# muestra a continuación.

from dataclasses import dataclass
```

```python
@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
```

5.30. El tipo de las fórmulas: Variables de una fórmula

En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Usando el tipo de las fórmulas proposicionales definido en el
-- [ejercicio anterior](https://bit.ly/3L3G2SX), 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

import Tipo_de_formulas (FProp(..))

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
# ---------------------------------------------------------------------
# Usando el [tipo de las fórmulas proposicionales](https://bit.ly/3L3G2SX),
# 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 src.tipo_de_formulas import Conj, Const, FProp, Impl, Neg, Var


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
```

5.31. El tipo de las fórmulas: Valor de una fórmula

En Haskell

```
-- ---------------------------------------------------------------------
-- 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     |
--                 |---+---+-------+-------|
--
-- Usando el tipo de las fórmulas proposicionales definido en el
-- [ejercicio anterior](https://bit.ly/3L3G2SX), 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

import Tipo_de_formulas (FProp(..))

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

```
# ---------------------------------------------------------------------
# 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     |
#                  |---+---+-------+-------|
#
# Usando el [tipo de las fórmulas proposicionales](https://bit.ly/3L3G2SX),
# 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,
#    >>> p = Impl(Var('A'), Conj(Var('A'), Var('B')))
#    >>> valor([('A',False),('B',False)], p)
#    True
#    >>> valor([('A',True),('B',False)], p)
#    False
# ---------------------------------------------------------------------

from src.tipo_de_formulas import Conj, Const, FProp, Impl, Neg, Var

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)
        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
```

5.32. El tipo de las fórmulas: Interpretaciones de una fórmula

En Haskell

```
-- ---------------------------------------------------------------------
-- Usando el [tipo de las fórmulas proposicionales](https://bit.ly/3L3G2SX),
-- definir la función
--    interpretaciones :: FProp -> [Interpretacion]
-- tal que (interpretaciones p) es la lista de las interpretaciones de
-- 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 Tipo_de_formulas (FProp(..))
import Variables_de_una_formula (variables)
import Valor_de_una_formula (Interpretacion)
import Data.List (nub)

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)
```

En Python

```
# ---------------------------------------------------------------------
# Usando el [tipo de las fórmulas proposicionales](https://bit.ly/3L3G2SX),
# 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)]]
# ---------------------------------------------------------------------

# pylint: disable=unused-import

from src.tipo_de_formulas import Conj, Const, FProp, Impl, Neg, Var
from src.valor_de_una_formula import Interpretacion
from src.variables_de_una_formula import variables


# 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))]
```

5.33. El tipo de las fórmulas: Reconocedor de tautologías

En Haskell

```
-- ---------------------------------------------------------------------
-- 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
--
-- Usando el tipo de las fórmulas proposicionales definido en el
-- [ejercicio anterior](https://bit.ly/3L3G2SX), 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
-- ---------------------------------------------------------------------

module Validez_de_una_formula where

import Tipo_de_formulas (FProp(..))
import Valor_de_una_formula (valor)
import Interpretaciones_de_una_formula (interpretaciones)

esTautologia :: FProp -> Bool
esTautologia p =
  and [valor i p | i <- interpretaciones p]
```

En Python

```
# ---------------------------------------------------------------------
# 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
#
# Usando el [tipo de las fórmulas proposicionales](https://bit.ly/3L3G2SX),
# 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
# ---------------------------------------------------------------------

# pylint: disable=unused-import

from src.interpretaciones_de_una_formula import interpretaciones
from src.tipo_de_formulas import Conj, Const, FProp, Impl, Neg, Var
from src.valor_de_una_formula import valor


def esTautologia(p: FProp) -> bool:
    return all((valor(i, p) for i in interpretaciones(p)))
```

5.34. El tipo de las expresiones aritméticas

5.34.1. En Haskell

```
-- ---------------------------------------------------------------------
-- El tipo de las expresiones aritméticas formadas 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))
-- se define como se muestra a continuación.
```

```haskell
module Tipo_expresion_aritmetica where

data Expr = Lit Int
          | Suma Expr Expr
          | Op Expr
          | SiCero Expr Expr Expr
  deriving (Eq, Show)
```

5.34.2. En Python

```python
# El tipo de las expresiones aritméticas formadas 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))
# se define como se muestra a continuación.

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
```

5.35. El tipo de las expresiones aritméticas: Valor de una expresión

En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Usando el [tipo de las expresiones aritméticas](https://bit.ly/40vCQUh),
-- 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

import Tipo_expresion_aritmetica (Expr (..))

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
# ---------------------------------------------------------------------
# Usando el [tipo de las expresiones aritméticas](https://bit.ly/40vCQUh),
# 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 src.tipo_expresion_aritmetica import Expr, Lit, Op, SiCero, Suma

# 1ª solución
# ===========

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
```

5.36. El tipo de las expresiones aritméticas: Valor de la resta

En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Usando el [tipo de las expresiones aritméticas](https://bit.ly/40vCQUh),
-- 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
-- ---------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-orphans #-}

module Valor_de_la_resta where

import Tipo_expresion_aritmetica (Expr (..))
import Valor_de_una_expresion_aritmetica (valor)
import Test.QuickCheck

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

```
# ---------------------------------------------------------------------
# Usando el [tipo de las expresiones aritméticas](https://bit.ly/40vCQUh),
# 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,
```

```python
#    resta(Lit(42), Lit(2))  ==  Suma(Lit(42), Op(Lit(2)))
#
# Comprobar con Hypothesis que
#    valor(resta(x, y)) == valor(x) - valor(y)
# ---------------------------------------------------------------------

from random import choice, randint

from hypothesis import given
from hypothesis import strategies as st

from src.tipo_expresion_aritmetica import Expr, Lit, Op, SiCero, Suma
from src.valor_de_una_expresion_aritmetica import valor


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
```

5.37. El tipos de las expresiones aritméticas básicas

5.37.1. En Haskell

```
-- La expresión aritmética 2*(3+7) se representa por
--    P (C 2) (S (C 3) (C 7))
-- usando el tipo de dato definido a continuación.

module Expresion_aritmetica_basica where

data Expr = C Int
          | S Expr Expr
          | P Expr Expr
  deriving (Eq, Show)
```

5.37.2. En Python

```
# La expresión aritmética 2*(3+7) se representa por
#    P(C(2), S(C(3), C(7)))
# usando el tipo de dato definido a continuación.

from dataclasses import dataclass


@dataclass
class Expr:
```

```python
    pass

@dataclass
class C(Expr):
    x: int

@dataclass
class S(Expr):
    x: Expr
    y: Expr

@dataclass
class P(Expr):
    x: Expr
    y: Expr
```

5.38. Valor de una expresión aritmética básica

5.38.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Usando el [tipo de las expresiones aritméticas básicas]
-- (https://bit.ly/43EuWL4), 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

import Expresion_aritmetica_basica (Expr(..))

valor :: Expr -> Int
valor (C x)   = x
valor (S x y) = valor x + valor y
valor (P x y) = valor x * valor y
```

5.38.2. En Python

```python
# ---------------------------------------------------------------------
# Usando el [tipo de las expresiones aritméticas básicas]
# (https://bit.ly/43EuWL4), 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 src.expresion_aritmetica_basica import C, Expr, P, S


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
```

5.39. Aplicación de una función a una expresión aritmética

5.39.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Usando el [tipo de las expresiones aritméticas básicas]
-- (https://bit.ly/43EuWL4), 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))
-- ---------------------------------------------------------------------
```

```haskell
module Aplicacion_de_una_funcion_a_una_expresion_aritmetica where

import Expresion_aritmetica_basica (Expr(..))

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)
```

5.39.2. En Python

```python
# ---------------------------------------------------------------------
# Usando el [tipo de las expresiones aritméticas básicas]
# (https://bit.ly/43EuWL4), 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 typing import Callable

from src.expresion_aritmetica_basica import C, Expr, P, S


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
```

5.40. El tipo de las expresiones aritméticas con una variable

5.40.1. En Haskell

```haskell
-- La expresión X*(13+X) se representa por
--    P(X(), S(C(13), X()))
-- usando el tipo de las expresiones aritméticas con una variable
-- (denotada por X) que se define como se muestra a continuación,

module Expresion_aritmetica_con_una_variable where

data Expr = X
          | C Int
          | S Expr Expr
          | P Expr Expr
```

5.40.2. En Python

```python
# La expresión X*(13+X) se representa por
#    P(X(), S(C(13), X()))
# usando el tipo de las expresiones aritméticas con una variable
# (denotada por X) que se define como se muestra a continuación,

from dataclasses import dataclass


@dataclass
class Exp:
    pass

@dataclass
class X(Exp):
    pass

@dataclass
class C(Exp):
    x: int

@dataclass
```

```python
class S(Exp):
    x: Exp
    y: Exp

@dataclass
class P(Exp):
    x: Exp
    y: Exp
```

5.41. Valor de una expresión aritmética con una variable

5.41.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Usando el [tipo de las expresiones aritméticas con una variable]
-- (https://bit.ly/40mwjeF), definir la función
--    valor :: Expr -> Int -> Int
-- tal que (valor e n) es el valor de la expresión e cuando se
-- sustituye su variable por n. Por ejemplo,
--    valor (P X (S (C 13) X)) 2  ==  30
-- ---------------------------------------------------------------------

module Valor_de_una_expresion_aritmetica_con_una_variable where

import Expresion_aritmetica_con_una_variable (Expr (..))

valor :: Expr -> Int -> Int
valor X         n = n
valor (C a)     _ = a
valor (S e1 e2) n = valor e1 n + valor e2 n
valor (P e1 e2) n = valor e1 n * valor e2 n
```

5.41.2. En Python

```python
# ---------------------------------------------------------------------
# Usando el [tipo de las expresiones aritméticas con una variable](https://bit.ly
# definir la función
#    valor : (Exp, int) -> int
```

```
# tal que valor(e, n) es el valor de la expresión e cuando se
# sustituye su variable por n. Por ejemplo,
#    valor(P(X(), S(C(13), X())), 2)  ==  30
# ---------------------------------------------------------------------

from src.expresion_aritmetica_con_una_variable import C, Exp, P, S, X


def valor(e: Exp, n: int) -> int:
    match e:
        case X():
            return n
        case C(a):
            return a
        case S(e1, e2):
            return valor(e1, n) + valor(e2, n)
        case P(e1, e2):
            return valor(e1, n) * valor(e2, n)
    assert False
```

5.42. Número de variables de una expresión aritmética

5.42.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Usando el [tipo de las expresiones aritméticas con una variable]
-- (https://bit.ly/40mwjeF), definir la función
--    numVars :: Expr -> Int
-- tal que (numVars e) es el número de variables en la expresión e. Por
-- ejemplo,
--    numVars (C 3)                 ==  0
--    numVars X                     ==  1
--    numVars (P X (S (C 13) X))    ==  2
-- ---------------------------------------------------------------------

module Numero_de_variables_de_una_expresion_aritmetica where

import Expresion_aritmetica_con_una_variable (Expr (..))
```

```haskell
numVars :: Expr -> Int
numVars X       = 1
numVars (C _)   = 0
numVars (S a b) = numVars a + numVars b
numVars (P a b) = numVars a + numVars b
```

5.42.2. En Python

```python
# ---------------------------------------------------------------------
# Usando el [tipo de las expresiones aritméticas con una variable](https://bit.ly
# definir la función
#    numVars : (Exp) -> int
# tal que numVars(e) es el número de variables en la expresión e. Por
# ejemplo,
#    numVars(C(3))                      ==  0
#    numVars(X())                       ==  1
#    numVars(P(X(), S(C(13), X())))  ==  2
# ---------------------------------------------------------------------

from src.expresion_aritmetica_con_una_variable import C, Exp, P, S, X


def numVars(e: Exp) -> int:
    match e:
        case X():
            return 1
        case C(_):
            return 0
        case S(e1, e2):
            return numVars(e1) + numVars(e2)
        case P(e1, e2):
            return numVars(e1) + numVars(e2)
    assert False
```

5.43. El tipo de las expresiones aritméticas con variables

5.43.1. En Haskell

```haskell
-- La expresión 2*(a+5) puede representarse por
--    P (C 2) (S (V 'a') (C 5))
-- usando el tipo de las expresiones aritméticas con variables definido
-- como se muestra a continuación.

module Expresion_aritmetica_con_variables where

data Expr = C Int
          | V Char
          | S Expr Expr
          | P Expr Expr
  deriving (Eq, Show)
```

5.43.2. En Python

```python
# La expresión 2*(a+5) puede representarse por
#    P(C(2), S(V('a'), C(5)))
# usando el tipo de las expresiones aritméticas con variables definido
# como se muestra a continuación.

from dataclasses import dataclass


@dataclass
class Expr:
    pass

@dataclass
class C(Expr):
    x: int

@dataclass
class V(Expr):
    x: str
```

```
@dataclass
class S(Expr):
    x: Expr
    y: Expr

@dataclass
class P(Expr):
    x: Expr
    y: Expr
```

5.44. Valor de una expresión aritmética con variables

5.44.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Usando el [tipo de las expresiones aritméticas con variables]
-- (https://bit.ly/3HfB0QO), definir la función
--    valor :: Expr -> [(Char,Int)] -> Int
-- tal que (valor x e) es el valor de la expresión x en el entorno e (es
-- decir, el valor de la expresión donde las variables de x se sustituyen
-- por los valores según se indican en el entorno e). Por ejemplo,
--    λ> valor (P (C 2) (S (V 'a') (V 'b'))) [('a',2),('b',5)]
--    14
-- ---------------------------------------------------------------------

module Valor_de_una_expresion_aritmetica_con_variables where

import Expresion_aritmetica_con_variables (Expr (C, V, S, P))

valor :: Expr -> [(Char,Int)] -> Int
valor (C x)   _ = x
valor (V x)   e = head [y | (z,y) <- e, z == x]
valor (S x y) e = valor x e + valor y e
valor (P x y) e = valor x e * valor y e
```

5.44.2. En Python

```python
# ---------------------------------------------------------------------
# Usando el [tipo de las expresiones aritméticas con variables]
# (https://bit.ly/3HfB0QO), definir la función
#    valor : (Expr, list[tuple[str, int]]) -> int
# tal que valor(x, e) es el valor de la expresión x en el entorno e (es
# decir, el valor de la expresión donde las variables de x se sustituyen
# por los valores según se indican en el entorno e). Por ejemplo,
#    λ> valor(P(C(2), S(V('a'), V('b'))), [('a', 2), ('b', 5)])
#    14
# ---------------------------------------------------------------------

from src.expresion_aritmetica_con_variables import C, Expr, P, S, V


def valor(e: Expr, xs: list[tuple[str, int]]) -> int:
    match e:
        case C(a):
            return a
        case V(x):
            return [y for (z, y) in xs if z == x][0]
        case S(e1, e2):
            return valor(e1, xs) + valor(e2, xs)
        case P(e1, e2):
            return valor(e1, xs) * valor(e2, xs)
    assert False
```

5.45. Número de sumas en una expresión aritmética

5.45.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Usando el [tipo de las expresiones aritméticas con variables]
-- (https://bit.ly/3HfB0QO), definir la función
--    sumas :: Expr -> Int
-- tal que (sumas e) es el número de sumas en la expresión e. Por
-- ejemplo,
--    sumas (P (V 'z') (S (C 3) (V 'x')))  ==  1
```

```haskell
--    sumas (S (V 'z') (S (C 3) (V 'x')))  ==  2
--    sumas (P (V 'z') (P (C 3) (V 'x')))  ==  0
-- ---------------------------------------------------------------------

module Numero_de_sumas_en_una_expresion_aritmetica where

import Expresion_aritmetica_con_variables (Expr (C, V, S, P))

sumas :: Expr -> Int
sumas (V _)   = 0
sumas (C _)   = 0
sumas (S x y) = 1 + sumas x + sumas y
sumas (P x y) = sumas x + sumas y
```

5.45.2. En Python

```python
# ---------------------------------------------------------------------
# Usando el [tipo de las expresiones aritméticas con variables]
# (https://bit.ly/3HfB0QO), definir la función
#    sumas : (Expr) -> int
# tal que sumas(e) es el número de sumas en la expresión e. Por
# ejemplo,
#    sumas(P(V('z'), S(C(3), V('x')))) == 1
#    sumas(S(V('z'), S(C(3), V('x')))) == 2
#    sumas(P(V('z'), P(C(3), V('x')))) == 0
# ---------------------------------------------------------------------

from src.expresion_aritmetica_con_variables import C, Expr, P, S, V


def sumas(e: Expr) -> int:
    match e:
        case C(_):
            return 0
        case V(_):
            return 0
        case S(e1, e2):
            return 1 + sumas(e1) + sumas(e2)
        case P(e1, e2):
            return sumas(e1) + sumas(e2)
```

```
    assert False
```

5.46. Sustitución en una expresión aritmética

5.46.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Usando el [tipo de las expresiones aritméticas con variables]
-- (https://bit.ly/3HfB0QO), definir la función
--    sustitucion :: Expr -> [(Char, Int)] -> Expr
-- tal que (sustitucion e s) es la expresión obtenida sustituyendo las
-- variables de la expresión e según se indica en la sustitución s. Por
-- ejemplo,
--    λ> sustitucion (P (V 'z') (S (C 3) (V 'x'))) [('x',7),('z',9)]
--    P (C 9) (S (C 3) (C 7))
--    λ> sustitucion (P (V 'z') (S (C 3) (V 'y'))) [('x',7),('z',9)]
--    P (C 9) (S (C 3) (V 'y'))
-- ---------------------------------------------------------------------

module Sustitucion_en_una_expresion_aritmetica where

import Expresion_aritmetica_con_variables (Expr (C, V, S, P))

sustitucion :: Expr -> [(Char, Int)] -> Expr
sustitucion e []          = e
sustitucion (V c) ((d,n):ps)
  | c == d                = C n
  | otherwise             = sustitucion (V c) ps
sustitucion (C n) _      = C n
sustitucion (S e1 e2) ps = S (sustitucion e1 ps) (sustitucion e2 ps)
sustitucion (P e1 e2) ps = P (sustitucion e1 ps) (sustitucion e2 ps)
```

5.46.2. En Python

```python
# ---------------------------------------------------------------------
# Usando el [tipo de las expresiones aritméticas con variables]
# (https://bit.ly/3HfB0QO), definir la función
#    sustitucion : (Expr, list[tuple[str, int]]) -> Expr
# tal que sustitucion(e s) es la expresión obtenida sustituyendo las
```

```
# variables de la expresión e según se indica en la sustitución s. Por
# ejemplo,
#    >>> sustitucion(P(V('z'), S(C(3), V('x'))), [('x', 7), ('z', 9)])
#    P(C(9), S(C(3), C(7)))
#    >>> sustitucion(P(V('z'), S(C(3), V('y'))), [('x', 7), ('z', 9)])
#    P(C(9), S(C(3), V('y')))
# ---------------------------------------------------------------------

from src.expresion_aritmetica_con_variables import C, Expr, P, S, V


def sustitucion(e: Expr, ps: list[tuple[str, int]]) -> Expr:
    match (e, ps):
        case(e, []):
            return e
        case (V(c), ps):
            if c == ps[0][0]:
                return C(ps[0][1])
            return sustitucion(V(c), ps[1:])
        case (C(n), _):
            return C(n)
        case (S(e1, e2), ps):
            return S(sustitucion(e1, ps), sustitucion(e2, ps))
        case (P(e1, e2), ps):
            return P(sustitucion(e1, ps), sustitucion(e2, ps))
    assert False
```

5.47. Expresiones aritméticas reducibles

5.47.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Usando el [tipo de las expresiones aritméticas con variables]
-- (https://bit.ly/3HfB0QO), definir la función
--    reducible :: Expr -> Bool
-- tal que (reducible a) se verifica si a es una expresión reducible; es
-- decir, contiene una operación en la que los dos operandos son números.
-- Por ejemplo,
--    reducible (S (C 3) (C 4))             == True
--    reducible (S (C 3) (V 'x'))           == False
```

```haskell
--    reducible (S (C 3) (P (C 4) (C 5)))   == True
--    reducible (S (V 'x') (P (C 4) (C 5))) == True
--    reducible (S (C 3) (P (V 'x') (C 5))) == False
--    reducible (C 3)                       == False
--    reducible (V 'x')                     == False
-- ---------------------------------------------------------------------

module Expresiones_aritmeticas_reducibles where

import Expresion_aritmetica_con_variables (Expr (C, V, S, P))

reducible :: Expr -> Bool
reducible (C _)           = False
reducible (V _)           = False
reducible (S (C _) (C _)) = True
reducible (S a b)         = reducible a || reducible b
reducible (P (C _) (C _)) = True
reducible (P a b)         = reducible a || reducible b
```

5.47.2. En Python

```python
# ---------------------------------------------------------------------
# Usando el [tipo de las expresiones aritméticas con variables]
# (https://bit.ly/3HfB0QO), definir la función
#    reducible : (Expr) -> bool
# tal que reducible(a) se verifica si a es una expresión reducible; es
# decir, contiene una operación en la que los dos operandos son números.
# Por ejemplo,
#    reducible(S(C(3), C(4)))             == True
#    reducible(S(C(3), V('x')))           == False
#    reducible(S(C(3), P(C(4), C(5))))    == True
#    reducible(S(V('x'), P(C(4), C(5)))) == True
#    reducible(S(C(3), P(V('x'), C(5)))) == False
#    reducible(C(3))                      == False
#    reducible(V('x'))                    == False
# ---------------------------------------------------------------------

from src.expresion_aritmetica_con_variables import C, Expr, P, S, V
```

```python
def reducible(e: Expr) -> bool:
    match e:
        case C(_):
            return False
        case V(_):
            return False
        case S(C(_), C(_)):
            return True
        case S(a, b):
            return reducible(a) or reducible(b)
        case P(C(_), C(_)):
            return True
        case P(a, b):
            return reducible(a) or reducible(b)
    assert False
```

5.48. Máximos valores de una expresión aritmética

5.48.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Las expresiones aritméticas generales se pueden definir usando el
-- siguiente tipo de datos
--    data Expr = C Int
--              | X
--              | S Expr Expr
--              | R Expr Expr
--              | P Expr Expr
--              | E Expr Int
--      deriving (Eq, Show)
-- Por ejemplo, la expresión
--    3*x - (x+2)^7
-- se puede definir por
--    R (P (C 3) X) (E (S X (C 2)) 7)
--
-- Definir la función
--    maximo :: Expr -> [Int] -> (Int,[Int])
-- tal que (maximo e xs) es el par formado por el máximo valor de la
```

```haskell
-- expresión e para los puntos de xs y en qué puntos alcanza el
-- máximo. Por ejemplo,
--    λ> maximo (E (S (C 10) (P (R (C 1) X) X)) 2) [-3..3]
--    (100,[0,1])
-- ---------------------------------------------------------------------

module Maximos_valores_de_una_expresion_aritmetica where

data Expr = C Int
          | X
          | S Expr Expr
          | R Expr Expr
          | P Expr Expr
          | E Expr Int
  deriving (Eq, Show)

maximo :: Expr -> [Int] -> (Int,[Int])
maximo e ns = (m,[n | n <- ns, valor e n == m])
  where m = maximum [valor e n | n <- ns]

valor :: Expr -> Int -> Int
valor (C x) _     = x
valor X     n     = n
valor (S e1 e2) n = valor e1 n + valor e2 n
valor (R e1 e2) n = valor e1 n - valor e2 n
valor (P e1 e2) n = valor e1 n * valor e2 n
valor (E e1 m1) n = valor e1 n ^ m1
```

5.48.2. En Python

```python
# ---------------------------------------------------------------------
# Las expresiones aritméticas generales se pueden definir usando el
# siguiente tipo de datos
#    @dataclass
#    class Expr:
#        pass
#
#    @dataclass
#    class C(Expr):
#        x: int
```

```
#
#    @dataclass
#    class X(Expr):
#        pass
#
#    @dataclass
#    class S(Expr):
#        x: Expr
#        y: Expr
#
#    @dataclass
#    class R(Expr):
#        x: Expr
#        y: Expr
#
#    @dataclass
#    class P(Expr):
#        x: Expr
#        y: Expr
#
#    @dataclass
#    class E(Expr):
#        x: Expr
#        y: int
# Por ejemplo, la expresión
#    3*x - (x+2)^7
# se puede definir por
#    R(P(C(3), X()), E(S(X(), C(2)), 7))
#
# Definir la función
#    maximo : (Expr, list[int]) -> tuple[int, list[int]]
# tal que maximo(e, xs) es el par formado por el máximo valor de la
# expresión e para los puntos de xs y en qué puntos alcanza el
# máximo. Por ejemplo,
#    >>> maximo(E(S(C(10), P(R(C(1), X()), X())), 2), range(-3, 4))
#    (100, [0, 1])
# ---------------------------------------------------------------------

from dataclasses import dataclass
```

```python
@dataclass
class Expr:
    pass

@dataclass
class C(Expr):
    x: int

@dataclass
class X(Expr):
    pass

@dataclass
class S(Expr):
    x: Expr
    y: Expr

@dataclass
class R(Expr):
    x: Expr
    y: Expr

@dataclass
class P(Expr):
    x: Expr
    y: Expr

@dataclass
class E(Expr):
    x: Expr
    y: int

def valor(e: Expr, n: int) -> int:
    match e:
        case C(a):
            return a
        case X():
            return n
        case S(e1, e2):
```

```python
            return valor(e1, n) + valor(e2, n)
        case R(e1, e2):
            return valor(e1, n) - valor(e2, n)
        case P(e1, e2):
            return valor(e1, n) * valor(e2, n)
        case E(e1, m):
            return valor(e1, n) ** m
    assert False

def maximo(e: Expr, ns: list[int]) -> tuple[int, list[int]]:
    m = max((valor(e, n) for n in ns))
    return (m, [n for n in ns if valor(e, n) == m])
```

5.49. Valor de expresiones aritméticas generales

5.49.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Las operaciones de suma, resta y  multiplicación se pueden
-- representar mediante el siguiente tipo de datos
--    data Op = S | R | M
-- La expresiones aritméticas con dichas operaciones se pueden
-- representar mediante el siguiente tipo de dato algebraico
--    data Expr = C Int
--              | A Op Expr Expr
-- Por ejemplo, la expresión
--    (7-3)+(2*5)
-- se representa por
--    A S (A R (C 7) (C 3)) (A M (C 2) (C 5))
--
-- Definir la función
--    valor :: Expr -> Int
-- tal que (valor e) es el valor de la expresión e. Por ejemplo,
--    valor (A S (A R (C 7) (C 3)) (A M (C 2) (C 5)))  ==  14
--    valor (A M (A R (C 7) (C 3)) (A S (C 2) (C 5)))  ==  28
-- ---------------------------------------------------------------------

module Valor_de_expresiones_aritmeticas_generales where
```

```haskell
data Op = S | R | M

data Expr = C Int
          | A Op Expr Expr

-- 1ª solución
-- ===========

valor :: Expr -> Int
valor (C x)      = x
valor (A o e1 e2) = aplica o (valor e1) (valor e2)
  where aplica :: Op -> Int -> Int -> Int
        aplica S x y = x+y
        aplica R x y = x-y
        aplica M x y = x*y

-- 2ª solución
-- ===========

valor2 :: Expr -> Int
valor2 (C n)    = n
valor2 (A o x y) = sig o (valor2 x) (valor2 y)
  where sig :: Op -> Int -> Int -> Int
        sig S = (+)
        sig M = (*)
        sig R = (-)
```

5.49.2. En Python

```python
# ---------------------------------------------------------------------
# Las operaciones de suma, resta y  multiplicación se pueden
# representar mediante el siguiente tipo de datos
#    Op = Enum('Op', ['S', 'R', 'M'])
# La expresiones aritméticas con dichas operaciones se pueden
# representar mediante el siguiente tipo de dato algebraico
#    @dataclass
#    class Expr:
#        pass
#
```

```
#    @dataclass
#    class C(Expr):
#        x: int
#
#    @dataclass
#    class A(Expr):
#        o: Op
#        x: Expr
#        y: Expr
# Por ejemplo, la expresión
#    (7-3)+(2*5)
# se representa por
#    A(Op.S, A(Op.R, C(7), C(3)), A(Op.M, C(2), C(5)))
#
# Definir la función
#    valor : (Expr) -> int
# tal que valor(e) es el valor de la expresión e. Por ejemplo,
#    >>> valor(A(Op.S, A(Op.R, C(7), C(3)), A(Op.M, C(2), C(5))))
#    14
#    >>> valor(A(Op.M, A(Op.R, C(7), C(3)), A(Op.S, C(2), C(5))))
#    28
# ---------------------------------------------------------------------

from dataclasses import dataclass
from enum import Enum

Op = Enum('Op', ['S', 'R', 'M'])

@dataclass
class Expr:
    pass

@dataclass
class C(Expr):
    x: int

@dataclass
class A(Expr):
    o: Op
    x: Expr
```

```
    y: Expr

def aplica(o: Op, x: int, y: int) -> int:
    match o:
        case Op.S:
            return x + y
        case Op.R:
            return x - y
        case Op.M:
            return x * y
    assert False

def valor(e: Expr) -> int:
    match e:
        case C(x):
            return x
        case A(o, e1, e2):
            return aplica(o, valor(e1), valor(e2))
    assert False
```

5.50. Valor de una expresión vectorial

5.50.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Se consideran las expresiones vectoriales formadas por un vector, la
-- suma de dos expresiones vectoriales o el producto de un entero por
-- una expresión vectorial. El siguiente tipo de dato define las
-- expresiones vectoriales
--    data ExpV = Vec Int Int
--              | Sum ExpV ExpV
--              | Mul Int ExpV
--      deriving Show
--
-- Definir la función
--    valorEV :: ExpV -> (Int,Int)
-- tal que (valorEV e) es el valorEV de la expresión vectorial c. Por
-- ejemplo,
--    valorEV (Vec 1 2)                                 ==  (1,2)
--    valorEV (Sum (Vec 1 2) (Vec 3 4))                 ==  (4,6)
```

```
--    valorEV (Mul 2 (Vec 3 4))                          ==  (6,8)
--    valorEV (Mul 2 (Sum (Vec 1 2 ) (Vec 3 4)))         ==  (8,12)
--    valorEV (Sum (Mul 2 (Vec 1 2)) (Mul 2 (Vec 3 4)))  ==  (8,12)
-- ---------------------------------------------------------------------

module Valor_de_una_expresion_vectorial where

data ExpV = Vec Int Int
          | Sum ExpV ExpV
          | Mul Int ExpV
  deriving Show

-- 1ª solución
-- ===========

valorEV1 :: ExpV -> (Int,Int)
valorEV1 (Vec x y)   = (x,y)
valorEV1 (Sum e1 e2) = (x1+x2,y1+y2)
  where (x1,y1) = valorEV1 e1
        (x2,y2) = valorEV1 e2
valorEV1 (Mul n e)   = (n*x,n*y)
  where (x,y) = valorEV1 e

-- 2ª solución
-- ===========

valorEV2 :: ExpV -> (Int,Int)
valorEV2 (Vec a b)   = (a, b)
valorEV2 (Sum e1 e2) = suma (valorEV2 e1) (valorEV2 e2)
valorEV2 (Mul n e1)  = multiplica n (valorEV2 e1)

suma :: (Int,Int) -> (Int,Int) -> (Int,Int)
suma (a,b) (c,d) = (a+c,b+d)

multiplica :: Int -> (Int, Int) -> (Int, Int)
multiplica n (a,b) = (n*a,n*b)
```

5.50.2. En Python

```
# ---------------------------------------------------------------------
# Se consideran las expresiones vectoriales formadas por un vector, la
# suma de dos expresiones vectoriales o el producto de un entero por
# una expresión vectorial. El siguiente tipo de dato define las
# expresiones vectoriales
#    @dataclass
#    class ExpV:
#        pass
#
#    @dataclass
#    class Vec(ExpV):
#        x: int
#        y: int
#
#    @dataclass
#    class Sum(ExpV):
#        x: ExpV
#        y: ExpV
#
#    @dataclass
#    class Mul(ExpV):
#        x: int
#        y: ExpV
#
# Definir la función
#    valorEV : (ExpV) -> tuple[int, int]
# tal que valorEV(e) es el valorEV de la expresión vectorial c. Por
# ejemplo,
#    valorEV(Vec(1, 2))                                  ==  (1,2)
#    valorEV(Sum(Vec(1, 2), Vec(3, 4)))                  ==  (4,6)
#    valorEV(Mul(2, Vec(3, 4)))                          ==  (6,8)
#    valorEV(Mul(2, Sum(Vec(1, 2), Vec(3, 4))))          ==  (8,12)
#    valorEV(Sum(Mul(2, Vec(1, 2)), Mul(2, Vec(3, 4))))  ==  (8,12)
# ---------------------------------------------------------------------

from dataclasses import dataclass


@dataclass
```

```python
class ExpV:
    pass

@dataclass
class Vec(ExpV):
    x: int
    y: int

@dataclass
class Sum(ExpV):
    x: ExpV
    y: ExpV

@dataclass
class Mul(ExpV):
    x: int
    y: ExpV

# 1ª solución
# ===========

def valorEV1(e: ExpV) -> tuple[int, int]:
    match e:
        case Vec(x, y):
            return (x, y)
        case Sum(e1, e2):
            x1, y1 = valorEV1(e1)
            x2, y2 = valorEV1(e2)
            return (x1 + x2, y1 + y2)
        case Mul(n, e):
            x, y = valorEV1(e)
            return (n * x, n * y)
    assert False

# 2ª solución
# ===========

def suma(p: tuple[int, int], q: tuple[int, int]) -> tuple[int, int]:
    a, b = p
    c, d = q
```

```
    return (a + c, b + d)

def multiplica(n: int, p: tuple[int, int]) -> tuple[int, int]:
    a, b = p
    return (n * a, n * b)

def valorEV2(e: ExpV) -> tuple[int, int]:
    match e:
        case Vec(x, y):
            return (x, y)
        case Sum(e1, e2):
            return suma(valorEV2(e1), valorEV2(e2))
        case Mul(n, e):
            return multiplica(n, valorEV2(e))
    assert False
```

Parte II

Algorítmica

Capítulo 6

El tipo abstracto de datos de las pilas

Contenido

6.1. El tipo abstracto de datos de las pilas

6.1.1. En Haskell

```haskell
-- Una pila es una estructura de datos, caracterizada por ser una
-- secuencia de elementos en la que las operaciones de inserción y
-- extracción se realizan por el mismo extremo.
--
-- Las operaciones que definen a tipo abstracto de datos (TAD) de las
```

```
-- pilas (cuyos elementos son del tipo a) son las siguientes:
--    vacia    :: Pila a
--    apila    :: a -> Pila a -> Pila a
--    cima     :: Pila a -> a
--    desapila :: Pila a -> Pila a
--    esVacia  :: Pila a -> Bool
-- tales que
--    + vacia es la pila vacía.
--    + (apila x p) es la pila obtenida añadiendo x al principio de p.
--    + (cima p) es la cima de la pila p.
--    + (desapila p) es la pila obtenida suprimiendo la cima de p.
--    + (esVacia p) se verifica si p es la pila vacía.
--
-- Las operaciones tienen que verificar las siguientes propiedades:
--    + cima(apila(x, p) == x
--    + desapila(apila(x, p)) == p
--    + esVacia(vacia)
--    + not esVacia(apila(x, p))
--
-- Para usar el TAD hay que usar una implementación concreta. En
-- principio, consideraremos dos una usando listas y otra usando
-- sucesiones. Hay que elegir la que se desee utilizar, descomentándola
-- y comentando las otras.

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

module TAD.Pila
  (Pila,
   vacia,      -- Pila a
   apila,      -- a -> Pila a -> Pila a
   cima,       -- Pila a -> a
   desapila,   -- Pila a -> Pila a
   esVacia,    -- Pila a -> Bool
  ) where

import TAD.PilaConListas
-- import TAD.PilaConSucesiones
```

6.1.2. En Python

```
# Una pila es una estructura de datos, caracterizada por ser una
# secuencia de elementos en la que las operaciones de inserción y
# extracción se realizan por el mismo extremo.
#
# Las operaciones que definen a tipo abstracto de datos (TAD) de las
# pilas (cuyos elementos son del tipo a) son las siguientes:
#    vacia    :: Pila a
#    apila    :: a -> Pila a -> Pila a
#    cima     :: Pila a -> a
#    desapila :: Pila a -> Pila a
#    esVacia  :: Pila a -> Bool
# tales que
# + vacia es la pila vacía.
# + (apila x p) es la pila obtenida añadiendo x al principio de p.
# + (cima p) es la cima de la pila p.
# + (desapila p) es la pila obtenida suprimiendo la cima de p.
# + (esVacia p) se verifica si p es la pila vacía.
#
# Las operaciones tienen que verificar las siguientes propiedades:
# + cima(apila(x, p) == x
# + desapila(apila(x, p)) == p
# + esVacia(vacia)
# + not esVacia(apila(x, p))
#
# Para usar el TAD hay que usar una implementación concreta. En
# principio, consideraremos dos una usando listas y otra usando
# sucesiones. Hay que elegir la que se desee utilizar, descomentándola
# y comentando las otras.

__all__ = [
    'Pila',
    'vacia',
    'apila',
    'esVacia',
    'cima',
    'desapila',
    'pilaAleatoria'
]
from src.TAD.pilaConListas import (Pila, apila, cima, desapila, esVacia,
```

```
                                   pilaAleatoria, vacia)

# from src.TAD.pilaConDeque import (Pila, apila, cima, desapila, esVacia,
#                                   pilaAleatoria, vacia)
```

6.2. El tipo de datos de las pilas mediante listas

6.2.1. En Haskell

```haskell
{-# OPTIONS_GHC -fno-warn-unused-top-binds #-}

module TAD.PilaConListas
  (Pila,
   vacia,      -- Pila a
   apila,      -- a -> Pila a -> Pila a
   cima,       -- Pila a -> a
   desapila,   -- Pila a -> Pila a
   esVacia,    -- Pila a -> Bool
   escribePila -- Show a => Pila a -> String
  ) where

import Test.QuickCheck

-- Representación de las pilas mediante listas.
newtype Pila a = P [a]
  deriving Eq

-- (escribePila p) es la cadena correspondiente a la pila p. Por
-- ejemplo,
--    escribePila (apila 5 (apila 2 (apila 3 vacia))) == "5 | 2 | 3"
escribePila :: Show a => Pila a -> String
escribePila (P [])     = "-"
escribePila (P [x])    = show x
escribePila (P (x:xs)) = show x ++ " | " ++ escribePila (P xs)

-- Procedimiento de escritura de pilas.
instance Show a => Show (Pila a) where
  show = escribePila
```

```haskell
-- Ejemplo de pila:
--    λ> apila 1 (apila 2 (apila 3 vacia))
--    1 | 2 | 3

-- vacia es la pila vacía. Por ejemplo,
--    λ> vacia
--    -
vacia   :: Pila a
vacia = P []

-- (apila x p) es la pila obtenida añadiendo x encima de la pila p. Por
-- ejemplo,
--    λ> apila 4 (apila 3 (apila 2 (apila 5 vacia)))
--    4 | 3 | 2 | 5
apila :: a -> Pila a -> Pila a
apila x (P xs) = P (x:xs)

-- (cima p) es la cima de la pila p. Por ejemplo,
--    λ> cima (apila 4 (apila 3 (apila 2 (apila 5 vacia))))
--    4
cima :: Pila a -> a
cima (P [])    = error "cima de la pila vacia"
cima (P (x:_)) = x

-- (desapila p) es la pila obtenida suprimiendo la cima de la pila
-- p. Por ejemplo,
--    λ> desapila (apila 4 (apila 3 (apila 2 (apila 5 vacia))))
--    3 | 2 | 5
desapila :: Pila a -> Pila a
desapila (P [])     = error "desapila la pila vacia"
desapila (P (_:xs)) = P  xs

-- (esVacia p) se verifica si p es la pila vacía. Por ejemplo,
--    esVacia (apila 1 (apila 2 (apila 3 vacia))) ==  False
--    esVacia vacia                               ==  True
esVacia :: Pila a -> Bool
esVacia (P xs) = null xs

-- Generador de pilas                                              --
```

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

-- genPila es un generador de pilas. Por ejemplo,
--    λ> sample genPila
--    -
--    0|0|-
--    -
--    -6|4|-3|3|0|-
--    -
--    9|5|-1|-3|0|-8|-5|-7|2|-
--    -3|-10|-3|-12|11|6|1|-2|0|-12|-6|-
--    2|-14|-5|2|-
--    5|9|-
--    -1|-14|5|-
--    6|13|0|17|-12|-7|-8|-19|-14|-5|10|14|3|-18|2|-14|-11|-6|-
genPila :: (Arbitrary a, Num a) => Gen (Pila a)
genPila = do
  xs <- listOf arbitrary
  return (foldr apila vacia xs)

-- El tipo pila es una instancia del arbitrario.
instance (Arbitrary a, Num a) => Arbitrary (Pila a) where
  arbitrary = genPila

-- Propiedades
-- ===========

-- Las propiedades son
prop_pilas :: Int -> Pila Int -> Bool
prop_pilas x p =
  cima (apila x p) == x &&
  desapila (apila x p) == p &&
  esVacia vacia &&
  not (esVacia (apila x p))

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

6.2.2. En Python

```
# Se define la clase Pila con los siguientes métodos:
#    + apila(x) añade x al principio de la pila.
#    + cima() devuelve la cima de la pila.
#    + desapila() elimina la cima de la pila.
#    + esVacia() se verifica si la pila es vacía.
# Por ejemplo,
#    >>> p = Pila()
#    >>> p
#    -
#    >>> p.apila(5)
#    >>> p.apila(2)
#    >>> p.apila(3)
#    >>> p.apila(4)
#    >>> p
#    4 | 3 | 2 | 5
#    >>> p.cima()
#    4
#    >>> p.desapila()
#    >>> p
#    3 | 2 | 5
#    >>> p.esVacia()
#    False
#    >>> p = Pila()
#    >>> p.esVacia()
#    True
#
# Además se definen las correspondientes funciones. Por ejemplo,
#    >>> vacia()
#    -
#    >>> apila(4, apila(3, apila(2, apila(5, vacia()))))
#    4 | 3 | 2 | 5
#    >>> cima(apila(4, apila(3, apila(2, apila(5, vacia())))))
#    4
#    >>> desapila(apila(4, apila(3, apila(2, apila(5, vacia())))))
#    3 | 2 | 5
#    >>> esVacia(apila(4, apila(3, apila(2, apila(5, vacia())))))
#    False
#    >>> esVacia(vacia())
#    True
```

```
#
# Finalmente, se define un generador aleatorio de pilas y se comprueba
# que las pilas cumplen las propiedades de su especificación.

__all__ = [
    'Pila',
    'vacia',
    'apila',
    'esVacia',
    'cima',
    'desapila',
    'pilaAleatoria'
]

from copy import deepcopy
from dataclasses import dataclass, field
from typing import Generic, TypeVar

from hypothesis import given
from hypothesis import strategies as st

A = TypeVar('A')

# Clase de las pilas mediante Listas
# ==================================

@dataclass
class Pila(Generic[A]):
    _elementos: list[A] = field(default_factory=list)

    def __repr__(self) -> str:
        """
        Devuelve una cadena con los elementos de la pila separados por " | ".
        Si la pila está vacía, devuelve "-".
        """
        if len(self._elementos) == 0:
            return '-'
        return " | ".join(str(x) for x in self._elementos)

    def apila(self, x: A) -> None:
```

```python
        """
        Agrega el elemento x al inicio de la pila.
        """
        self._elementos.insert(0, x)

    def esVacia(self) -> bool:
        """
        Verifica si la pila está vacía.

        Devuelve True si la pila está vacía, False en caso contrario.
        """
        return not self._elementos

    def cima(self) -> A:
        """
        Devuelve el elemento en la cima de la pila.
        """
        return self._elementos[0]

    def desapila(self) -> None:
        """
        Elimina el elemento en la cima de la pila.
        """
        self._elementos.pop(0)

# Funciones del tipo de las listas
# ================================

def vacia() -> Pila[A]:
    """
    Crea y devuelve una pila vacía de tipo A.
    """
    p: Pila[A] = Pila()
    return p

def apila(x: A, p: Pila[A]) -> Pila[A]:
    """
    Añade un elemento x al tope de la pila p y devuelve una copia de la
    pila modificada.
    """
```

```
    aux = deepcopy(p)
    aux.apila(x)
    return aux

def esVacia(p: Pila[A]) -> bool:
    """
    Devuelve True si la pila está vacía, False si no lo está.
    """
    return p.esVacia()

def cima(p: Pila[A]) -> A:
    """
    Devuelve el elemento en la cima de la pila p.
    """
    return p.cima()

def desapila(p: Pila[A]) -> Pila[A]:
    """
    Elimina el elemento en la cima de la pilla p y devuelve una copia de la
    pila resultante.
    """
    aux = deepcopy(p)
    aux.desapila()
    return aux

# Generador de pilas
# ==================

def pilaAleatoria() -> st.SearchStrategy[Pila[int]]:
    """
    Genera una estrategia de búsqueda para generar pilas de enteros de
    forma aleatoria.

    Utiliza la librería Hypothesis para generar una lista de enteros y
    luego se convierte en una instancia de la clase pila.
    """
    return st.lists(st.integers()).map(Pila)

# Comprobación de las propiedades de las pilas
# ============================================
```

```python
# Las propiedades son
@given(p=pilaAleatoria(), x=st.integers())
def test_pila(p: Pila[int], x: int) -> None:
    assert cima(apila(x, p)) == x
    assert desapila(apila(x, p)) == p
    assert esVacia(vacia())
    assert not esVacia(apila(x, p))

# La comprobación es
#    > poetry run pytest -q pilaConListas.py
#    1 passed in 0.25s
```

6.3. El tipo de datos de las pilas con librerías

6.3.1. En Haskell

```haskell
{-# OPTIONS_GHC -fno-warn-unused-top-binds #-}

module TAD.PilaConSucesiones
  (Pila,
   vacia,      -- Pila a
   apila,      -- a -> Pila a -> Pila a
   cima,       -- Pila a -> a
   desapila,   -- Pila a -> Pila a
   esVacia,    -- Pila a -> Bool
   escribePila -- Show a => Pila a -> String
  ) where

import Data.Sequence as S
import Test.QuickCheck

-- Representación de las pilas mediante sucesiones.
newtype Pila a = P (Seq a)
  deriving Eq

-- (escribePila p) es la cadena correspondiente a la pila p. Por
-- ejemplo,
--    escribePila (apila 5 (apila 2 (apila 3 vacia))) == "5 | 2 | 3"
escribePila :: Show a => Pila a -> String
```

```haskell
escribePila (P xs) = case viewl xs of
    EmptyL   -> "-"
    x :< xs' -> case viewl xs' of
        EmptyL -> show x
        _      -> show x ++ " | " ++ escribePila (P xs')

-- Procedimiento de escritura de pilas.
instance Show a => Show (Pila a) where
  show = escribePila

-- Ejemplo de pila:
--    λ> apila 1 (apila 2 (apila 3 vacia))
--    1 | 2 | 3

-- vacia es la pila vacía. Por ejemplo,
--    λ> vacia
--    -
vacia   :: Pila a
vacia = P empty

-- (apila x p) es la pila obtenida añadiendo x encima de la pila p. Por
-- ejemplo,
--    λ> apila 4 (apila 3 (apila 2 (apila 5 vacia)))
--    5 | 2 | 3 | 4
apila :: a -> Pila a -> Pila a
apila x (P xs) = P (x <| xs)

-- (cima p) es la cima de la pila p. Por ejemplo,
--    λ> cima (apila 4 (apila 3 (apila 2 (apila 5 vacia))))
--    4
cima :: Pila a -> a
cima (P xs) = case viewl xs of
  EmptyL -> error "cima de la pila vacia"
  x :< _ -> x

-- (desapila p) es la pila obtenida suprimiendo la cima de la pila
-- p. Por ejemplo,
--    λ> desapila (apila 4 (apila 3 (apila 2 (apila 5 vacia))))
--    3 | 2 | 5
desapila :: Pila a -> Pila a
```

```
desapila (P xs) = case viewl xs of
  EmptyL   -> error "desapila la pila vacia"
  _ :< xs' -> P xs'

-- (esVacia p) se verifica si p es la pila vacía. Por ejemplo,
--    esVacia (apila 1 (apila 2 (apila 3 vacia))) ==  False
--    esVacia vacia                               ==  True
esVacia :: Pila a -> Bool
esVacia (P xs) = S.null xs

-- Generador de pilas                                         --
-- ==================

-- genPila es un generador de pilas. Por ejemplo,
--    λ> sample genPila
--    -
--    -2
--    -
--    4 | -1 | 5 | 4 | -4 | 3
--    -8 | 2
--    4
--    5 | 7 | 10 | 6 | -4 | 11 | -1 | 0 | 7 | -3
--    -1 | -10
--    2 | -3 | -4 | 15 | -15 | 1 | -10 | -2 | -4 | 6 | -13 | 16 | -8 | 3 | 7
--    6
--    1 | -6 | -19 | 15 | -5 | -4 | -6 | -12 | -13 | 11 | 19 | -18 | -14 | -13 |
genPila :: (Arbitrary a, Num a) => Gen (Pila a)
genPila = do
  xs <- listOf arbitrary
  return (foldr apila vacia xs)

-- El tipo pila es una instancia del arbitrario.
instance (Arbitrary a, Num a) => Arbitrary (Pila a) where
  arbitrary = genPila

-- Propiedades
-- ===========

-- Las propiedades son
prop_pilas :: Int -> Pila Int -> Bool
```

```
prop_pilas x p =
  cima (apila x p) == x &&
  desapila (apila x p) == p &&
  esVacia vacia &&
  not (esVacia (apila x p))

-- La comprobación e:
--    λ> quickCheck prop_pilas
--    +++ OK, passed 100 tests.
```

6.3.2. En Python

```
# Se define la clase Pila con los siguientes métodos:
#    + apila(x) añade x al principio de la pila.
#    + cima() devuelve la cima de la pila.
#    + desapila() elimina la cima de la pila.
#    + esVacia() se verifica si la pila es vacía.
# Por ejemplo,
#    >>> p = Pila()
#    >>> p
#    -
#    >>> p.apila(5)
#    >>> p.apila(2)
#    >>> p.apila(3)
#    >>> p.apila(4)
#    >>> p
#    4 | 3 | 2 | 5
#    >>> p.cima()
#    4
#    >>> p.desapila()
#    >>> p
#    3 | 2 | 5
#    >>> p.esVacia()
#    False
#    >>> p = Pila()
#    >>> p.esVacia()
#    True
#
# Además se definen las correspondientes funciones. Por ejemplo,
#    >>> vacia()
```

```
#    -
#    >>> apila(4, apila(3, apila(2, apila(5, vacia()))))
#    4 | 3 | 2 | 5
#    >>> cima(apila(4, apila(3, apila(2, apila(5, vacia())))))
#    4
#    >>> desapila(apila(4, apila(3, apila(2, apila(5, vacia())))))
#    3 | 2 | 5
#    >>> esVacia(apila(4, apila(3, apila(2, apila(5, vacia())))))
#    False
#    >>> esVacia(vacia())
#    True
#
# Finalmente, se define un generador aleatorio de pilas y se comprueba
# que las pilas cumplen las propiedades de su especificación.

__all__ = [
    'Pila',
    'vacia',
    'apila',
    'esVacia',
    'cima',
    'desapila',
    'pilaAleatoria'
]

from collections import deque
from copy import deepcopy
from dataclasses import dataclass, field
from typing import Generic, TypeVar

from hypothesis import given
from hypothesis import strategies as st

A = TypeVar('A')

# Clase de las pilas mediante Listas
# ==================================

@dataclass
class Pila(Generic[A]):
```

```
    _elementos: deque[A] = field(default_factory=deque)

    def __repr__(self) -> str:
        """
        Devuelve una cadena con los elementos de la pila separados por " | ".
        Si la pila está vacía, devuelve "-".
        """
        if len(self._elementos) == 0:
            return '-'
        return ' | '.join(str(x) for x in self._elementos)

    def apila(self, x: A) -> None:
        """
        Agrega el elemento x al inicio de la pila.
        """
        self._elementos.appendleft(x)

    def esVacia(self) -> bool:
        """
        Verifica si la pila está vacía.

        Devuelve True si la pila está vacía, False en caso contrario.
        """
        return len(self._elementos) == 0

    def cima(self) -> A:
        """
        Devuelve el elemento en la cima de la pila.
        """
        return self._elementos[0]

    def desapila(self) -> None:
        """
        Elimina el elemento en la cima de la pila.
        """
        self._elementos.popleft()

# Funciones del tipo de las listas
# ================================
```

```
def vacia() -> Pila[A]:
    """
    Crea y devuelve una pila vacía de tipo A.
    """
    p: Pila[A] = Pila()
    return p

def apila(x: A, p: Pila[A]) -> Pila[A]:
    """
    Añade un elemento x al tope de la pila p y devuelve una copia de la
    pila modificada.
    """
    _aux = deepcopy(p)
    _aux.apila(x)
    return _aux

def esVacia(p: Pila[A]) -> bool:
    """
    Devuelve True si la pila está vacía, False si no lo está.
    """
    return p.esVacia()

def cima(p: Pila[A]) -> A:
    """
    Devuelve el elemento en la cima de la pila p.
    """
    return p.cima()

def desapila(p: Pila[A]) -> Pila[A]:
    """
    Elimina el elemento en la cima de la pilla p y devuelve una copia de la
    pila resultante.
    """
    _aux = deepcopy(p)
    _aux.desapila()
    return _aux

# Generador de pilas
# ==================
```

```python
def pilaAleatoria() -> st.SearchStrategy[Pila[int]]:
    """
    Genera una estrategia de búsqueda para generar pilas de enteros de
    forma aleatoria.

    Utiliza la librería Hypothesis para generar una lista de enteros y
    luego se convierte en una instancia de la clase pila.
    """
    def _creaPila(elementos: list[int]) -> Pila[int]:
        pila: Pila[int] = vacia()
        pila._elementos.extendleft(elementos)
        return pila
    return st.builds(_creaPila, st.lists(st.integers()))

# Comprobación de las propiedades de las pilas
# ============================================

# Las propiedades son
@given(p=pilaAleatoria(), x=st.integers())
def test_pila(p: Pila[int], x: int) -> None:
    assert cima(apila(x, p)) == x
    assert desapila(apila(x, p)) == p
    assert esVacia(vacia())
    assert not esVacia(apila(x, p))

# La comprobación es
#    > poetry run pytest -q pilaConQueue.py
#    1 passed in 0.25s
```

6.4. Transformación entre pilas y listas

6.4.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Utilizando el [tipo abstracto de datos de las pilas](https://bit.ly/3GTToyK),
-- definir la función
--    listaApila :: [a] -> Pila a
--    pilaAlista :: Pila a -> [a]
-- tales que
-- + (listaApila xs) es la pila formada por los elementos de xs.
```

```haskell
--   Por ejemplo,
--      λ> listaApila [3, 2, 5]
--      5 | 2 | 3
-- + (pilaAlista p) es la lista formada por los elementos de la
--   pila p. Por ejemplo,
--      λ> pilaAlista (apila 5 (apila 2 (apila 3 vacia)))
--      [3, 2, 5]
--
-- Comprobar con QuickCheck que ambas funciones son inversa; es decir,
--    pilaAlista (listaApila xs) = xs
--    listaApila (pilaAlista p)  = p
-- ---------------------------------------------------------------------

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

module Transformaciones_pilas_listas where

import TAD.Pila (Pila, vacia, apila, esVacia, cima, desapila)
import Test.QuickCheck

-- 1ª definición de listaApila
-- ===========================

listaApila :: [a] -> Pila a
listaApila ys = aux (reverse ys)
  where aux []     = vacia
        aux (x:xs) = apila x (aux xs)

-- 2ª definición de listaApila
-- ===========================

listaApila2 :: [a] -> Pila a
listaApila2 = aux . reverse
  where aux [] = vacia
        aux (x:xs) = apila x (aux xs)

-- 3ª definición de listaApila
-- ===========================

listaApila3 :: [a] -> Pila a
```

```haskell
listaApila3 = aux . reverse
  where aux = foldr apila vacia

-- 4ª definición de listaApila
-- ===========================

listaApila4 :: [a] -> Pila a
listaApila4 xs = foldr apila vacia (reverse xs)

-- 5ª definición de listaApila
-- ===========================

listaApila5 :: [a] -> Pila a
listaApila5 = foldr apila vacia . reverse

-- Comprobación de equivalencia de las definiciones de listaApila
-- ==============================================================

-- La propiedad es
prop_listaApila :: [Int] -> Bool
prop_listaApila xs =
  all (== listaApila xs)
      [listaApila2 xs,
       listaApila3 xs,
       listaApila4 xs,
       listaApila5 xs]

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

-- 1ª definición de pilaAlista
-- ===========================

pilaAlista :: Pila a -> [a]
pilaAlista p
  | esVacia p = []
  | otherwise = pilaAlista dp ++ [cp]
  where cp = cima p
        dp = desapila p
```

```
-- 2ª definición de pilaAlista
-- ===========================

pilaAlista2 :: Pila a -> [a]
pilaAlista2 = reverse . aux
  where aux p | esVacia p = []
              | otherwise = cp : aux dp
          where cp = cima p
                dp = desapila p

-- Comprobación de equivalencia de las definiciones de pilaAlista
-- ==============================================================

-- La propiedad es
prop_pilaAlista :: Pila Int -> Bool
prop_pilaAlista p =
  pilaAlista p == pilaAlista2 p

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

-- Comprobación de las propiedades
-- ===============================

-- La primera propiedad es
prop_1_listaApila :: [Int] -> Bool
prop_1_listaApila xs =
  pilaAlista (listaApila xs) == xs

-- La comprobación es
--    λ> quickCheck prop_1_listaApila
--    +++ OK, passed 100 tests.

-- La segunda propiedad es
prop_2_listaApila :: Pila Int -> Bool
prop_2_listaApila p =
  listaApila (pilaAlista p) == p
```

```
-- La comprobación es
--    λ> quickCheck prop_2_listaApila
--    +++ OK, passed 100 tests.
```

6.4.2. En Python

```
# ---------------------------------------------------------------------
# Utilizando el [tipo abstacto de datos de las pilas](https://bit.ly/3GTToyK)
# definir las funciones
#    listaApila : (list[A]) -> Pila[A]
#    pilaAlista : (Pila[A]) -> list[A]
# tales que
# + listaApila(xs) es la pila formada por los elementos de xs.
#   Por ejemplo,
#      >>> listaApila([3, 2, 5])
#      5 | 2 | 3
# + pilaAlista(p) es la lista formada por los elementos de la
#   pila p. Por ejemplo,
#      >>> ej = apila(5, apila(2, apila(3, vacia())))
#      >>> pilaAlista(ej)
#      [3, 2, 5]
#      >>> print(ej)
#      5 | 2 | 3
#
# Comprobar con Hypothesis que ambas funciones son inversas; es decir,
#    pilaAlista(listaApila(xs)) == xs
#    listaApila(pilaAlista(p))  == p
# ---------------------------------------------------------------------

from copy import deepcopy
from typing import TypeVar

from hypothesis import given
from hypothesis import strategies as st

from src.TAD.pila import (Pila, apila, cima, desapila, esVacia, pilaAleatoria,
                          vacia)

A = TypeVar('A')
```

```
# 1ª definición de listaApila
# ===========================

def listaApila(ys: list[A]) -> Pila[A]:
    def aux(xs: list[A]) -> Pila[A]:
        if not xs:
            return vacia()
        return apila(xs[0], aux(xs[1:]))

    return aux(list(reversed(ys)))

# 2ª solución de listaApila
# =========================

def listaApila2(xs: list[A]) -> Pila[A]:
    p: Pila[A] = Pila()
    for x in xs:
        p.apila(x)
    return p

# Comprobación de equivalencia de las definiciones de listaApila
# ==============================================================

# La propiedad es
@given(st.lists(st.integers()))
def test_listaApila(xs: list[int]) -> None:
    assert listaApila(xs) == listaApila2(xs)

# 1ª definición de pilaAlista
# ===========================

def pilaAlista(p: Pila[A]) -> list[A]:
    if esVacia(p):
        return []
    cp = cima(p)
    dp = desapila(p)
    return pilaAlista(dp) + [cp]

# 2ª definición de pilaAlista
# ===========================
```

```python
def pilaAlista2Aux(p: Pila[A]) -> list[A]:
    if p.esVacia():
        return []
    cp = p.cima()
    p.desapila()
    return pilaAlista2Aux(p) + [cp]

def pilaAlista2(p: Pila[A]) -> list[A]:
    p1 = deepcopy(p)
    return pilaAlista2Aux(p1)

# 3ª definición de pilaAlista
# ===========================

def pilaAlista3Aux(p: Pila[A]) -> list[A]:
    r = []
    while not p.esVacia():
        r.append(p.cima())
        p.desapila()
    return r[::-1]

def pilaAlista3(p: Pila[A]) -> list[A]:
    p1 = deepcopy(p)
    return pilaAlista3Aux(p1)

# Comprobación de equivalencia de las definiciones de pilaAlista
# ==============================================================

@given(p=pilaAleatoria())
def test_pilaAlista(p: Pila[int]) -> None:
    assert pilaAlista(p) == pilaAlista2(p)
    assert pilaAlista(p) == pilaAlista3(p)

# Comprobación de las propiedades
# ===============================

# La primera propiedad es
@given(st.lists(st.integers()))
def test_1_listaApila(xs: list[int]) -> None:
```

```
    assert pilaAlista(listaApila(xs)) == xs

# La segunda propiedad es
@given(p=pilaAleatoria())
def test_2_listaApila(p: Pila[int]) -> None:
    assert listaApila(pilaAlista(p)) == p

# La comprobación es
#      src> poetry run pytest -v transformaciones_pilas_listas.py
#      test_listaApila PASSED
#      test_pilaAlista PASSED
#      test_1_listaApila PASSED
#      test_2_listaApila PASSED
```

6.5. Filtrado de pilas según una propiedad

6.5.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Utilizando el [tipo abstracto de datos de las pilas](https://bit.ly/3GTToyK),
-- definir la función
--    filtraPila :: (a -> Bool) -> Pila a -> Pila a
-- tal que (filtraPila p q) es la pila obtenida con los elementos de
-- pila q que verifican el predicado p, en el mismo orden. Por ejemplo,
--    λ> ejPila = apila 6 (apila 3 (apila 1 (apila 4 vacia)))
--    λ> ejPila
--    6 | 3 | 1 | 4
--    λ> filtraPila even ejPila
--    6 | 4
--    λ> filtraPila odd ejPila
--    3 | 1
-- ---------------------------------------------------------------------

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

module FiltraPila where

import TAD.Pila (Pila, vacia, apila, esVacia, cima, desapila)
import Transformaciones_pilas_listas (listaApila, pilaAlista)
import Test.QuickCheck.HigherOrder
```

```haskell
-- 1ª solución
-- ===========

filtraPila1 :: (a -> Bool) -> Pila a -> Pila a
filtraPila1 p q
  | esVacia q = vacia
  | p cq      = apila cq (filtraPila1 p dq)
  | otherwise = filtraPila1 p dq
  where cq = cima q
        dq = desapila q

-- 2ª solución
-- ===========

-- Se usarán las funciones listaApila y pilaAlista del ejercicio
-- "Transformaciones entre pilas y listas" que se encuentra en
-- https://bit.ly/3ZHewQ8

filtraPila2 :: (a -> Bool) -> Pila a -> Pila a
filtraPila2 p q =
  listaApila (filter p (pilaAlista q))

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

-- La propiedad es
prop_filtraPila :: (Int -> Bool) -> [Int] -> Bool
prop_filtraPila p xs =
  filtraPila1 p q == filtraPila2 p q
  where q = listaApila xs

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

6.5.2. En Python

```python
# ---------------------------------------------------------------------
# Utilizando el [tipo abstracto de datos de las pilas](https://bit.ly/3GTToyK)
```

```python
# definir la función
#    filtraPila : (Callable[[A], bool], Pila[A]) -> Pila[A]
# tal que filtraPila(p, q) es la pila obtenida con los elementos de
# pila q que verifican el predicado p, en el mismo orden. Por ejemplo,
#    >>> ej = apila(3, apila(4, apila(6, apila(5, vacia()))))
#    >>> filtraPila(lambda x: x % 2 == 0, ej)
#    4 | 6
#    >>> filtraPila(lambda x: x % 2 == 1, ej)
#    3 | 5
#    >>> ej
#    3 | 4 | 6 | 5
# ---------------------------------------------------------------------

# pylint: disable=unused-import

from copy import deepcopy
from typing import Callable, TypeVar

from hypothesis import given

from src.TAD.pila import (Pila, apila, cima, desapila, esVacia, pilaAleatoria,
                          vacia)
from src.transformaciones_pilas_listas import listaApila, pilaAlista

A = TypeVar('A')

# 1ª solución
# ===========

def filtraPila1(p: Callable[[A], bool], q: Pila[A]) -> Pila[A]:
    if esVacia(q):
        return q
    cq = cima(q)
    dq = desapila(q)
    r = filtraPila1(p, dq)
    if p(cq):
        return apila(cq, r)
    return r

# 2ª solución
```

```
# ===========

# Se usarán las funciones listaApila y pilaAlista del ejercicio
# ”Transformaciones entre pilas y listas” que se encuentra en
# https://bit.ly/3ZHewQ8

def filtraPila2(p: Callable[[A], bool], q: Pila[A]) -> Pila[A]:
    return listaApila(list(filter(p, pilaAlista(q))))

# 3ª solución
# ===========

def filtraPila3Aux(p: Callable[[A], bool], q: Pila[A]) -> Pila[A]:
    if q.esVacia():
        return q
    cq = q.cima()
    q.desapila()
    r = filtraPila3Aux(p, q)
    if p(cq):
        r.apila(cq)
    return r

def filtraPila3(p: Callable[[A], bool], q: Pila[A]) -> Pila[A]:
    q1 = deepcopy(q)
    return filtraPila3Aux(p, q1)

# 4ª solución
# ===========

def filtraPila4Aux(p: Callable[[A], bool], q: Pila[A]) -> Pila[A]:
    r: Pila[A] = Pila()
    while not q.esVacia():
        cq = q.cima()
        q.desapila()
        if p(cq):
            r.apila(cq)
    r1: Pila[A] = Pila()
    while not r.esVacia():
        r1.apila(r.cima())
        r.desapila()
```

```python
    return r1

def filtraPila4(p: Callable[[A], bool], q: Pila[A]) -> Pila[A]:
    q1 = deepcopy(q)
    return filtraPila4Aux(p, q1)

# Comprobación de equivalencia de las definiciones
# ================================================

# La propiedad es
@given(p=pilaAleatoria())
def test_filtraPila(p: Pila[int]) -> None:
    r = filtraPila1(lambda x: x % 2 == 0, p)
    assert filtraPila2(lambda x: x % 2 == 0, p) == r
    assert filtraPila3(lambda x: x % 2 == 0, p) == r
    assert filtraPila4(lambda x: x % 2 == 0, p) == r

# La comprobación es
#    src> poetry run pytest -q filtraPila.py
#    1 passed in 0.25s
```

6.6. Aplicación de una función a los elementos de una pila

6.6.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Utilizando el [tipo abstracto de datos de las pilas](https://bit.ly/3GTToyK),
-- definir la función
--    mapPila :: (a -> a) -> Pila a -> Pila a
-- tal que (mapPila f p) es la pila formada con las imágenes por f de
-- los elementos de pila p, en el mismo orden. Por ejemplo,
--    λ> mapPila (+1) (apila 5 (apila 2 (apila 7 vacia)))
--    6 | 3 | 8
-- ---------------------------------------------------------------------

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

module MapPila where
```

```haskell
import TAD.Pila (Pila, vacia, apila, esVacia, cima, desapila)
import Transformaciones_pilas_listas (listaApila, pilaAlista)
import Test.QuickCheck.HigherOrder

-- 1ª solución
-- ===========

mapPila1 :: (a -> a) -> Pila a -> Pila a
mapPila1 f p
  | esVacia p = p
  | otherwise = apila (f cp) (mapPila1 f dp)
  where cp = cima p
        dp = desapila p

-- 2ª solución
-- ===========

-- Se usarán las funciones listaApila y pilaAlista del ejercicio
-- "Transformaciones entre pilas y listas" que se encuentra en
-- https://bit.ly/3ZHewQ8

mapPila2 :: (a -> a) -> Pila a -> Pila a
mapPila2 f p =
  listaApila (map f (pilaAlista p))

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

-- La propiedad es
prop_mapPila :: (Int -> Int) -> [Int] -> Bool
prop_mapPila f p =
  mapPila1 f q == mapPila2 f q
  where q = listaApila p

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

6.6.2. En Python

```python
# ---------------------------------------------------------------------
# Utilizando el [tipo abstracto de datos de las pilas](https://bit.ly/3GTToyK)
# definir la función
#    mapPila : (Callable[[A], A], Pila[A]) -> Pila[A]
# tal que mapPila(f, p) es la pila formada con las imágenes por f de
# los elementos de pila p, en el mismo orden. Por ejemplo,
#    >>> ej = apila(5, apila(2, apila(7, vacia())))
#    >>> mapPila1(lambda x: x + 1, ej)
#    6 | 3 | 8
#    >>> ej
#    5 | 2 | 7
# ---------------------------------------------------------------------

# pylint: disable=unused-import

from copy import deepcopy
from typing import Callable, TypeVar

from hypothesis import given

from src.TAD.pila import (Pila, apila, cima, desapila, esVacia, pilaAleatoria,
                          vacia)
from src.transformaciones_pilas_listas import listaApila, pilaAlista

A = TypeVar('A')

# 1ª solución
# ===========

def mapPila1(f: Callable[[A], A], p: Pila[A]) -> Pila[A]:
    if esVacia(p):
        return p
    cp = cima(p)
    dp = desapila(p)
    return apila(f(cp), mapPila1(f, dp))

# 2ª solución
# ===========
```

```python
# Se usarán las funciones listaApila y pilaAlista del ejercicio
# "Transformaciones entre pilas y listas" que se encuentra en
# https://bit.ly/3ZHewQ8

def mapPila2(f: Callable[[A], A], p: Pila[A]) -> Pila[A]:
    return listaApila(list(map(f, pilaAlista(p))))

# 3ª solución
# ===========

def mapPila3Aux(f: Callable[[A], A], p: Pila[A]) -> Pila[A]:
    if p.esVacia():
        return p
    cp = p.cima()
    p.desapila()
    r = mapPila3Aux(f, p)
    r.apila(f(cp))
    return r

def mapPila3(f: Callable[[A], A], p: Pila[A]) -> Pila[A]:
    p1 = deepcopy(p)
    return mapPila3Aux(f, p1)

# 4ª solución
# ===========

def mapPila4Aux(f: Callable[[A], A], p: Pila[A]) -> Pila[A]:
    r: Pila[A] = Pila()
    while not p.esVacia():
        cp = p.cima()
        p.desapila()
        r.apila(f(cp))
    r1: Pila[A] = Pila()
    while not r.esVacia():
        r1.apila(r.cima())
        r.desapila()
    return r1

def mapPila4(f: Callable[[A], A], p: Pila[A]) -> Pila[A]:
    p1 = deepcopy(p)
```

```
    return mapPila4Aux(f, p1)

# Comprobación de equivalencia de las definiciones
# ===============================================

# La propiedad es
@given(p=pilaAleatoria())
def test_mapPila(p: Pila[int]) -> None:
    r = mapPila1(lambda x: x + 1 == 0, p)
    assert mapPila2(lambda x: x + 1 == 0, p) == r
    assert mapPila3(lambda x: x + 1 == 0, p) == r
    assert mapPila4(lambda x: x + 1 == 0, p) == r

# La comprobación es
#    src> poetry run pytest -q mapPila.py
#    1 passed in 0.29s
```

6.7. Pertenencia a una pila

6.7.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Utilizando el [tipo abstracto de datos de las pilas](https://bit.ly/3GTToyK),
-- definir la función
--    pertenecePila :: Eq a => a -> Pila a -> Bool
-- tal que (pertenecePila y p) se verifica si y es un elemento de la
-- pila p. Por ejemplo,
--    pertenecePila 2 (apila 5 (apila 2 (apila 3 vacia))) == True
--    pertenecePila 4 (apila 5 (apila 2 (apila 3 vacia))) == False
-- ---------------------------------------------------------------------

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

module PertenecePila where

import TAD.Pila (Pila, vacia, apila, esVacia, cima, desapila)
import Transformaciones_pilas_listas (pilaAlista)
import Test.QuickCheck

-- 1ª solución
```

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

pertenecePila :: Eq a => a -> Pila a -> Bool
pertenecePila x p
  | esVacia p  = False
  | otherwise  = x == cp || pertenecePila x dp
  where cp = cima p
        dp = desapila p

-- 2ª solución
-- ===========

-- Se usará la función pilaAlista del ejercicio
-- "Transformaciones entre pilas y listas" que se encuentra en
-- https://bit.ly/3ZHewQ8

pertenecePila2 :: Eq a => a -> Pila a -> Bool
pertenecePila2 x p =
  x `elem` pilaAlista p

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

-- La propiedad es
prop_pertenecePila :: Int -> Pila Int -> Bool
prop_pertenecePila x p =
  pertenecePila x p == pertenecePila2 x p

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

6.7.2. En Python

```python
# ---------------------------------------------------------------------
# Utilizando el [tipo abstracto de las pilas de las pilas](https://bit.ly/3GTToyK
# definir la función
#    pertenecePila : (A, Pila[A]) -> bool
# tal que pertenecePila(x, p) se verifica si x es un elemento de la
# pila p. Por ejemplo,
```

```
#    >>> pertenecePila(2, apila(5, apila(2, apila(3, vacia()))))
#    True
#    >>> pertenecePila(4, apila(5, apila(2, apila(3, vacia()))))
#    False
# ---------------------------------------------------------------------

# pylint: disable=unused-import

from copy import deepcopy
from typing import TypeVar

from hypothesis import given
from hypothesis import strategies as st

from src.TAD.pila import (Pila, apila, cima, desapila, esVacia, pilaAleatoria,
                          vacia)
from src.transformaciones_pilas_listas import pilaAlista

A = TypeVar('A')

# 1ª solución
# ===========

def pertenecePila(x: A, p: Pila[A]) -> bool:
    if esVacia(p):
        return False
    cp = cima(p)
    dp = desapila(p)
    return x == cp or pertenecePila(x, dp)

# 2ª solución
# ===========

# Se usará la función pilaAlista del ejercicio
# "Transformaciones entre pilas y listas" que se encuentra en
# https://bit.ly/3ZHewQ8

def pertenecePila2(x: A, p: Pila[A]) -> bool:
    return x in pilaAlista(p)
```

```
# 3ª solución
# ===========

def pertenecePila3Aux(x: A, p: Pila[A]) -> bool:
    if p.esVacia():
        return False
    cp = p.cima()
    p.desapila()
    return x == cp or pertenecePila3Aux(x, p)

def pertenecePila3(x: A, p: Pila[A]) -> bool:
    p1 = deepcopy(p)
    return pertenecePila3Aux(x, p1)

# 4ª solución
# ===========

def pertenecePila4Aux(x: A, p: Pila[A]) -> bool:
    while not p.esVacia():
        cp = p.cima()
        p.desapila()
        if x == cp:
            return True
    return False

def pertenecePila4(x: A, p: Pila[A]) -> bool:
    p1 = deepcopy(p)
    return pertenecePila4Aux(x, p1)

# Comprobación de equivalencia de las definiciones
# ================================================

# La propiedad es
@given(x=st.integers(), p=pilaAleatoria())
def test_pertenecePila(x: int, p: Pila[int]) -> None:
    r = pertenecePila(x, p)
    assert pertenecePila2(x, p) == r
    assert pertenecePila3(x, p) == r
    assert pertenecePila4(x, p) == r
```

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

6.8. Inclusión de pilas

6.8.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Utilizando el [tipo abstracto de datos de las pilas](https://bit.ly/3GTToyK),
-- definir la función
--    contenidaPila :: Eq a => Pila a -> Pila a -> Bool
-- tal que (contenidaPila p1 p2) se verifica si todos los elementos de
-- de la pila p1 son elementos de la pila p2. Por ejemplo,
--    λ> ej1 = apila 3 (apila 2 vacia)
--    λ> ej2 = apila 3 (apila 4 vacia)
--    λ> ej3 = apila 5 (apila 2 (apila 3 vacia))
--    λ> contenidaPila ej1 ej3
--    True
--    λ> contenidaPila ej2 ej3
--    False
-- ---------------------------------------------------------------------

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

module ContenidaPila where

import TAD.Pila (Pila, vacia, apila, esVacia, cima, desapila)
import PertenecePila (pertenecePila)
import Test.QuickCheck

-- 1ª solución
-- ===========

-- Se usará la función pertenecePila del ejercicio
-- "Pertenencia a una pila" que se encuentra en
-- https://bit.ly/3WdM9GC

contenidaPila1 :: Eq a => Pila a -> Pila a -> Bool
contenidaPila1 p1 p2
```

```haskell
  | esVacia p1 = True
  | otherwise  = pertenecePila cp1 p2 && contenidaPila1 dp1 p2
  where cp1 = cima p1
        dp1 = desapila p1

-- 2ª solución
-- ===========

contenidaPila2 :: Eq a => Pila a -> Pila a -> Bool
contenidaPila2 p1 p2 =
  contenidaLista (pilaAlista p1) (pilaAlista p2)

contenidaLista :: Eq a => [a] -> [a] -> Bool
contenidaLista xs ys =
  all (`elem` ys) xs

-- (pilaAlista p) es la lista formada por los elementos de la
-- lista p. Por ejemplo,
--    λ> pilaAlista (apila 5 (apila 2 (apila 3 vacia)))
--    [3, 2, 5]
pilaAlista :: Pila a -> [a]
pilaAlista = reverse . aux
  where aux p | esVacia p = []
              | otherwise = cp : aux dp
          where cp = cima p
                dp = desapila p

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

-- La propiedad es
prop_contenidaPila :: Pila Int -> Pila Int -> Bool
prop_contenidaPila p1 p2 =
  contenidaPila1 p1 p2 == contenidaPila2 p1 p2

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

6.8.2. En Python

```python
# ---------------------------------------------------------------------
# Utilizando el [tipo abstracto de datos de las pilas](https://bit.ly/3GTToyK),
# definir la función
#    contenidaPila : (Pila[A], Pila[A]) -> bool
# tal que contenidaPila(p1, p2) se verifica si todos los elementos de
# de la pila p1 son elementos de la pila p2. Por ejemplo,
#    >>> ej1 = apila(3, apila(2, vacia()))
#    >>> ej2 = apila(3, apila(4, vacia()))
#    >>> ej3 = apila(5, apila(2, apila(3, vacia())))
#    >>> contenidaPila(ej1, ej3)
#    True
#    >>> contenidaPila(ej2, ej3)
#    False
# ---------------------------------------------------------------------

# pylint: disable=unused-import

from copy import deepcopy
from typing import TypeVar

from hypothesis import given

from src.pertenecePila import pertenecePila
from src.TAD.pila import (Pila, apila, cima, desapila, esVacia, pilaAleatoria,
                          vacia)
from src.transformaciones_pilas_listas import pilaAlista

A = TypeVar('A')

# 1ª solución
# ===========

# Se usará la función pertenecePila del ejercicio
# "Pertenencia a una pila" que se encuentra en
# https://bit.ly/3WdM9GC

def contenidaPila1(p1: Pila[A], p2: Pila[A]) -> bool:
    if esVacia(p1):
        return True
```

```
    cp1 = cima(p1)
    dp1 = desapila(p1)
    return pertenecePila(cp1, p2) and contenidaPila1(dp1, p2)

# 2ª solución
# ===========

# Se usará la función pilaAlista del ejercicio
# "Transformaciones entre pilas y listas" que se encuentra en
# https://bit.ly/3ZHewQ8

def contenidaPila2(p1: Pila[A], p2: Pila[A]) -> bool:
    return set(pilaAlista(p1)) <= set(pilaAlista(p2))

# 3ª solución
# ===========

def contenidaPila3Aux(p1: Pila[A], p2: Pila[A]) -> bool:
    if p1.esVacia():
        return True
    cp1 = p1.cima()
    p1.desapila()
    return pertenecePila(cp1, p2) and contenidaPila1(p1, p2)

def contenidaPila3(p1: Pila[A], p2: Pila[A]) -> bool:
    q = deepcopy(p1)
    return contenidaPila3Aux(q, p2)

# 4ª solución
# ===========

def contenidaPila4Aux(p1: Pila[A], p2: Pila[A]) -> bool:
    while not p1.esVacia():
        cp1 = p1.cima()
        p1.desapila()
        if not pertenecePila(cp1, p2):
            return False
    return True

def contenidaPila4(p1: Pila[A], p2: Pila[A]) -> bool:
```

```
    q = deepcopy(p1)
    return contenidaPila4Aux(q, p2)

# Comprobación de equivalencia de las definiciones
# ================================================

# La propiedad es
@given(p1=pilaAleatoria(), p2=pilaAleatoria())
def test_contenidaPila(p1: Pila[int], p2: Pila[int]) -> None:
    r = contenidaPila1(p1, p2)
    assert contenidaPila2(p1, p2) == r
    assert contenidaPila3(p1, p2) == r
    assert contenidaPila4(p1, p2) == r

# La comprobación es
#    src> poetry run pytest -q contenidaPila.py
#    1 passed in 0.40s
```

6.9. Reconocimiento de prefijos de pilas

6.9.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Utilizando el [tipo abstracto de datos de las pilas](https://bit.ly/3GTToyK),
-- definir la función
--    prefijoPila :: Eq a => Pila a -> Pila a -> Bool
-- tal que (prefijoPila p1 p2) se verifica si la pila p1 es justamente
-- un prefijo de la pila p2. Por ejemplo,
--    λ> ej1 = apila 4 (apila 2 vacia)
--    λ> ej2 = apila 4 (apila 2 (apila 5 vacia))
--    λ> ej3 = apila 5 (apila 4 (apila 2 vacia))
--    λ> prefijoPila ej1 ej2
--    True
--    λ> prefijoPila ej1 ej3
--    False
-- ---------------------------------------------------------------------

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

module PrefijoPila where
```

```haskell
import TAD.Pila (Pila, vacia, apila, esVacia, cima, desapila)
import Transformaciones_pilas_listas (pilaAlista)
import Data.List (isSuffixOf)
import Test.QuickCheck

-- 1ª solución
-- ===========

prefijoPila :: Eq a => Pila a -> Pila a -> Bool
prefijoPila p1 p2
  | esVacia p1 = True
  | esVacia p2 = False
  | otherwise  = cp1 == cp2 && prefijoPila dp1 dp2
  where cp1 = cima p1
        dp1 = desapila p1
        cp2 = cima p2
        dp2 = desapila p2

-- 2ª solución
-- ===========

-- Se usará la función pilaAlista del ejercicio
-- "Transformaciones entre pilas y listas" que se encuentra en
-- https://bit.ly/3ZHewQ8

prefijoPila2 :: Eq a => Pila a -> Pila a -> Bool
prefijoPila2 p1 p2 =
  pilaAlista p1 `isSuffixOf` pilaAlista p2

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

-- La propiedad es
prop_prefijoPila :: Pila Int -> Pila Int -> Bool
prop_prefijoPila p1 p2 =
  prefijoPila p1 p2 == prefijoPila2 p1 p2

-- La comprobación es
--    λ> quickCheck prop_prefijoPila
```

```
--    +++ OK, passed 100 tests.
```

6.9.2. En Python

```
# ---------------------------------------------------------------------
# Utilizando el [tipo abstracto de datos de las pilas](https://bit.ly/3GTToyK),
# definir la función
#    prefijoPila : (Pila[A], Pila[A]) -> bool
# tal que prefijoPila(p1, p2) se verifica si la pila p1 es justamente
# un prefijo de la pila p2. Por ejemplo,
#    >>> ej1 = apila(4, apila(2, vacia()))
#    >>> ej2 = apila(4, apila(2, apila(5, vacia())))
#    >>> ej3 = apila(5, apila(4, apila(2, vacia())))
#    >>> prefijoPila(ej1, ej2)
#    True
#    >>> prefijoPila(ej1, ej3)
#    False
# ---------------------------------------------------------------------

# pylint: disable=unused-import

from copy import deepcopy
from typing import TypeVar

from hypothesis import given

from src.TAD.pila import (Pila, apila, cima, desapila, esVacia, pilaAleatoria,
                          vacia)
from src.transformaciones_pilas_listas import pilaAlista

A = TypeVar('A')

# 1ª solución
# ===========

def prefijoPila(p1: Pila[A], p2: Pila[A]) -> bool:
    if esVacia(p1):
        return True
    if esVacia(p2):
        return False
```

```
    cp1 = cima(p1)
    dp1 = desapila(p1)
    cp2 = cima(p2)
    dp2 = desapila(p2)
    return cp1 == cp2 and prefijoPila(dp1, dp2)

# 2ª solución
# ===========

# Se usará la función pilaAlista del ejercicio
# "Transformaciones entre pilas y listas" que se encuentra en
# https://bit.ly/3ZHewQ8

def esSufijoLista(xs: list[A], ys: list[A]) -> bool:
    if not xs:
        return True
    return xs == ys[-len(xs):]

def prefijoPila2(p1: Pila[A], p2: Pila[A]) -> bool:
    return esSufijoLista(pilaAlista(p1), pilaAlista(p2))

# 3ª solución
# ===========

def prefijoPila3Aux(p1: Pila[A], p2: Pila[A]) -> bool:
    if p1.esVacia():
        return True
    if p2.esVacia():
        return False
    cp1 = p1.cima()
    p1.desapila()
    cp2 = p2.cima()
    p2.desapila()
    return cp1 == cp2 and prefijoPila3(p1, p2)

def prefijoPila3(p1: Pila[A], p2: Pila[A]) -> bool:
    q1 = deepcopy(p1)
    q2 = deepcopy(p2)
    return prefijoPila3Aux(q1, q2)
```

```python
# 4ª solución
# ===========

def prefijoPila4Aux(p1: Pila[A], p2: Pila[A]) -> bool:
    while not p2.esVacia() and not p1.esVacia():
        if p1.cima() != p2.cima():
            return False
        p1.desapila()
        p2.desapila()
    return p1.esVacia()

def prefijoPila4(p1: Pila[A], p2: Pila[A]) -> bool:
    q1 = deepcopy(p1)
    q2 = deepcopy(p2)
    return prefijoPila4Aux(q1, q2)

# Comprobación de equivalencia de las definiciones
# ================================================

# La propiedad es
@given(p1=pilaAleatoria(), p2=pilaAleatoria())
def test_prefijoPila(p1: Pila[int], p2: Pila[int]) -> None:
    r = prefijoPila(p1, p2)
    assert prefijoPila2(p1, p2) == r
    assert prefijoPila3(p1, p2) == r
    assert prefijoPila4(p1, p2) == r

# La comprobación es
#    src> poetry run pytest -q prefijoPila.py
#    1 passed in 0.32s
```

6.10. Reconocimiento de subpilas

6.10.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Utilizando el [tipo abstracto de datos de las pilas](https://bit.ly/3GTToyK),
-- definir la función
--    subPila :: Eq a => Pila a -> Pila a -> Bool
-- tal que (subPila p1 p2) se verifica si p1 es una subpila de p2. Por
```

```
-- ejemplo,
--    λ> ej1 = apila 2 (apila 3 vacia)
--    λ> ej2 = apila 7 (apila 2 (apila 3 (apila 5 vacia)))
--    λ> ej3 = apila 2 (apila 7 (apila 3 (apila 5 vacia)))
--    λ> subPila ej1 ej2
--    True
--    λ> subPila ej1 ej3
--    False
-- ---------------------------------------------------------------------

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

module SubPila where

import TAD.Pila (Pila, vacia, apila, esVacia, cima, desapila)
import Transformaciones_pilas_listas (pilaAlista)
import PrefijoPila (prefijoPila)
import Data.List (isPrefixOf, tails)
import Test.QuickCheck

-- 1ª solución
-- ===========

-- Se usará la función PrefijoPila del ejercicio
-- "Reconocimiento de prefijos de pilas" que se encuentra en
-- https://bit.ly/3Xqu7lo

subPila1 :: Eq a => Pila a -> Pila a -> Bool
subPila1 p1 p2
    | esVacia p1 = True
    | esVacia p2 = False
    | cp1 == cp2 = prefijoPila dp1 dp2 || subPila1 p1 dp2
    | otherwise  = subPila1 p1 dp2
    where cp1 = cima p1
          dp1 = desapila p1
          cp2 = cima p2
          dp2 = desapila p2


-- 2ª solución
```

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

-- Se usará la función pilaAlista del ejercicio
-- "Transformaciones entre pilas y listas" que se encuentra en
-- https://bit.ly/3ZHewQ8

subPila2 :: Eq a => Pila a -> Pila a -> Bool
subPila2 p1 p2 =
  sublista (pilaAlista p1) (pilaAlista p2)

-- (sublista xs ys) se verifica si xs es una sublista de ys. Por
-- ejemplo,
--    sublista [3,2] [5,3,2,7]  ==  True
--    sublista [3,2] [5,3,7,2]  ==  False
sublista :: Eq a => [a] -> [a] -> Bool
sublista xs ys =
  any (xs `isPrefixOf`) (tails ys)

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

-- La propiedad es
prop_subPila :: Pila Int -> Pila Int -> Bool
prop_subPila p1 p2 =
  subPila1 p1 p2 == subPila2 p1 p2

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

6.10.2. En Python

```python
# ---------------------------------------------------------------------
# Utilizando el [tipo abstracto de datos de las pilas](https://bit.ly/3GTToyK),
# definir la función
#    subPila : (Pila[A], Pila[A]) -> bool
# tal que subPila(p1, p2) se verifica si p1 es una subpila de p2. Por
# ejemplo,
#    >>> ej1 = apila(2, apila(3, vacia()))
#    >>> ej2 = apila(7, apila(2, apila(3, apila(5, vacia()))))
```

```
#    >>> ej3 = apila(2, apila(7, apila(3, apila(5, vacia()))))
#    >>> subPila(ej1, ej2)
#    True
#    >>> subPila(ej1, ej3)
#    False
# ---------------------------------------------------------------------

# pylint: disable=unused-import

from copy import deepcopy
from typing import TypeVar

from hypothesis import given

from src.prefijoPila import prefijoPila
from src.TAD.pila import (Pila, apila, cima, desapila, esVacia, pilaAleatoria,
                          vacia)
from src.transformaciones_pilas_listas import pilaAlista

A = TypeVar('A')

# 1ª solución
# ===========

# Se usará la función PrefijoPila del ejercicio
# ”Reconocimiento de prefijos de pilas” que se encuentra en
# https://bit.ly/3Xqu7lo

def subPila1(p1: Pila[A], p2: Pila[A]) -> bool:
    if esVacia(p1):
        return True
    if esVacia(p2):
        return False
    cp1 = cima(p1)
    dp1 = desapila(p1)
    cp2 = cima(p2)
    dp2 = desapila(p2)
    if cp1 == cp2:
        return prefijoPila(dp1, dp2) or subPila1(p1, dp2)
    return subPila1(p1, dp2)
```

```
# 2ª solución
# ===========

# Se usará la función pilaAlista del ejercicio
# "Transformaciones entre pilas y listas" que se encuentra en
# https://bit.ly/3ZHewQ8

# sublista(xs, ys) se verifica si xs es una sublista de ys. Por
# ejemplo,
#    >>> sublista([3,2], [5,3,2,7])
#    True
#    >>> sublista([3,2], [5,3,7,2])
#    False
def sublista(xs: list[A], ys: list[A]) -> bool:
    return any(xs == ys[i:i+len(xs)] for i in range(len(ys) - len(xs) + 1))

def subPila2(p1: Pila[A], p2: Pila[A]) -> bool:
    return sublista(pilaAlista(p1), pilaAlista(p2))

# 3ª solución
# ===========

def subPila3Aux(p1: Pila[A], p2: Pila[A]) -> bool:
    if p1.esVacia():
        return True
    if p2.esVacia():
        return False
    if p1.cima() != p2.cima():
        p2.desapila()
        return subPila3Aux(p1, p2)
    q1 = deepcopy(p1)
    p1.desapila()
    p2.desapila()
    return prefijoPila(p1, p2) or subPila3Aux(q1, p2)

def subPila3(p1: Pila[A], p2: Pila[A]) -> bool:
    q1 = deepcopy(p1)
    q2 = deepcopy(p2)
    return subPila3Aux(q1, q2)
```

```
# Comprobación de equivalencia de las definiciones
# ================================================

# La propiedad es
@given(p1=pilaAleatoria(), p2=pilaAleatoria())
def test_subPila(p1: Pila[int], p2: Pila[int]) -> None:
    r = subPila1(p1, p2)
    assert subPila2(p1, p2) == r
    assert subPila3(p1, p2) == r

# La comprobación es
#    src> poetry run pytest -q subPila.py
#    1 passed in 0.32s
```

6.11. Reconocimiento de ordenación de pilas

6.11.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Utilizando el [tipo abstracto de datos de las pilas](https://bit.ly/3GTToyK),
-- definir la función
--    ordenadaPila :: Ord a => Pila a -> Bool
-- tal que (ordenadaPila p) se verifica si los elementos de la pila p
-- están ordenados en orden creciente. Por ejemplo,
--    ordenadaPila (apila 1 (apila 5 (apila 6 vacia))) == True
--    ordenadaPila (apila 1 (apila 0 (apila 6 vacia))) == False
-- ---------------------------------------------------------------------

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

module OrdenadaPila where

import TAD.Pila (Pila, vacia, apila, esVacia, cima, desapila)
import Transformaciones_pilas_listas (pilaAlista)
import Test.QuickCheck

-- 1ª solución
-- ===========
```

```haskell
ordenadaPila :: Ord a => Pila a -> Bool
ordenadaPila p
  | esVacia p  = True
  | esVacia dp = True
  | otherwise  = cp <= cdp && ordenadaPila dp
  where cp  = cima p
        dp  = desapila p
        cdp = cima dp

-- 2ª solución
-- ===========

ordenadaPila2 :: Ord a => Pila a -> Bool
ordenadaPila2 =
  ordenadaLista . reverse . pilaAlista

-- (ordenadaLista xs) se verifica si la lista xs está ordenada de menor
-- a mayor. Por ejemplo,
ordenadaLista :: Ord a => [a] -> Bool
ordenadaLista xs =
  and [x <= y | (x,y) <- zip xs (tail xs)]

-- Se usará la función pilaAlista del ejercicio
-- "Transformaciones entre pilas y listas" que se encuentra en
-- https://bit.ly/3ZHewQ8

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

-- La propiedad es
prop_ordenadaPila :: Pila Int -> Bool
prop_ordenadaPila p =
  ordenadaPila p == ordenadaPila2 p

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

6.11.2. En Python

```
# ---------------------------------------------------------------------
# Utilizando el [tipo abstracto de datos de las pilas](https://bit.ly/3GTToyK),
# definir la función
#    ordenadaPila : (Pila[A]) -> bool
# tal que ordenadaPila(p) se verifica si los elementos de la pila p
# están ordenados en orden creciente. Por ejemplo,
#    >>> ordenadaPila(apila(1, apila(5, apila(6, vacia()))))
#    True
#    >>> ordenadaPila(apila(1, apila(0, apila(6, vacia()))))
#    False
# ---------------------------------------------------------------------

# pylint: disable=unused-import

from copy import deepcopy
from typing import TypeVar

from hypothesis import given

from src.TAD.pila import (Pila, apila, cima, desapila, esVacia, pilaAleatoria,
                          vacia)
from src.transformaciones_pilas_listas import pilaAlista

A = TypeVar('A', int, float, str)

# 1ª solución
# ===========

def ordenadaPila(p: Pila[A]) -> bool:
    if esVacia(p):
        return True
    cp = cima(p)
    dp = desapila(p)
    if esVacia(dp):
        return True
    cdp = cima(dp)
    return cp <= cdp and ordenadaPila(dp)

# 2ª solución
```

```
# ===========

# Se usará la función pilaAlista del ejercicio
# ”Transformaciones entre pilas y listas” que se encuentra en
# https://bit.ly/3ZHewQ8

# ordenadaLista(xs, ys) se verifica si xs es una lista ordenada. Por
# ejemplo,
#    >>> ordenadaLista([2, 5, 8])
#    True
#    >>> ordenadalista([2, 8, 5])
#    False
def ordenadaLista(xs: list[A]) -> bool:
    return all((x <= y for (x, y) in zip(xs, xs[1:])))

def ordenadaPila2(p: Pila[A]) -> bool:
    return ordenadaLista(list(reversed(pilaAlista(p))))

# 3ª solución
# ===========

def ordenadaPila3Aux(p: Pila[A]) -> bool:
    if p.esVacia():
        return True
    cp = p.cima()
    p.desapila()
    if p.esVacia():
        return True
    return cp <= p.cima() and ordenadaPila3Aux(p)

def ordenadaPila3(p: Pila[A]) -> bool:
    q = deepcopy(p)
    return ordenadaPila3Aux(q)

# 4ª solución
# ===========

def ordenadaPila4Aux(p: Pila[A]) -> bool:
    while not p.esVacia():
        cp = p.cima()
```

```
        p.desapila()
        if not p.esVacia() and cp > p.cima():
            return False
    return True

def ordenadaPila4(p: Pila[A]) -> bool:
    q = deepcopy(p)
    return ordenadaPila4Aux(q)

# Comprobación de equivalencia de las definiciones
# ================================================

# La propiedad es
@given(p=pilaAleatoria())
def test_ordenadaPila(p: Pila[int]) -> None:
    r = ordenadaPila(p)
    assert ordenadaPila2(p) == r
    assert ordenadaPila3(p) == r
    assert ordenadaPila4(p) == r

# La comprobación es
#    src> poetry run pytest -q ordenadaPila.py
#    1 passed in 0.31s
```

6.12. Ordenación de pilas por inserción

6.12.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Utilizando el [tipo abstracto de datos de las pilas](https://bit.ly/3GTToyK),
-- definir la función
--    ordenaInserPila :: Ord a => Pila a -> Pila a
-- tal que (ordenaInserPila p) es la pila obtenida ordenando por
-- inserción los los elementos de la pila p. Por ejemplo,
--    λ> ordenaInserPila (apila 4 (apila 1 (apila 3 vacia)))
--    1 | 3 | 4
--
-- Comprobar con QuickCheck que la pila (ordenaInserPila p) está
-- ordenada.
-- ---------------------------------------------------------------------
```

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

module OrdenaInserPila where

import TAD.Pila (Pila, vacia, apila, esVacia, cima, desapila)
import Transformaciones_pilas_listas (listaApila, pilaAlista)
import OrdenadaPila (ordenadaPila)
import Test.QuickCheck

-- 1ª solución
-- ===========

ordenaInserPila1 :: Ord a => Pila a -> Pila a
ordenaInserPila1 p
  | esVacia p = p
  | otherwise = insertaPila cp (ordenaInserPila1 dp)
  where cp = cima p
        dp = desapila p

insertaPila :: Ord a => a -> Pila a -> Pila a
insertaPila x p
  | esVacia p = apila x p
  | x < cp    = apila x p
  | otherwise = apila cp (insertaPila x dp)
  where cp = cima p
        dp = desapila p

-- 2ª solución
-- ===========

ordenaInserPila2 :: Ord a => Pila a -> Pila a
ordenaInserPila2  =
  listaApila . reverse . ordenaInserLista . pilaAlista

ordenaInserLista :: Ord a => [a] -> [a]
ordenaInserLista []      = []
ordenaInserLista (x: xs) = insertaLista x (ordenaInserLista xs)

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

```
insertaLista x [] = [x]
insertaLista x (y:ys) | x < y = x : y : ys
                      | otherwise = y : insertaLista x ys

-- Se usarán las funciones listaApila y pilaAlista del ejercicio
-- "Transformaciones entre pilas y listas" que se encuentra en
-- https://bit.ly/3ZHewQ8

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

-- La propiedad es
prop_ordenaInserPila :: Pila Int -> Bool
prop_ordenaInserPila p =
  ordenaInserPila1 p == ordenaInserPila2 p

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

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

-- Se usará la función ordenadaPila del ejercicio
-- "Reconocimiento de ordenación de pilas" que se encuentra en
-- https://bit.ly/3COqRbK

-- La propiedad es
prop_ordenadaOrdenaInserPila :: Pila Int -> Bool
prop_ordenadaOrdenaInserPila p =
  ordenadaPila (ordenaInserPila1 p)

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

6.12.2. En Python

```
# ---------------------------------------------------------------------
# Utilizando el [tipo abstracto de datos de las pilas](https://bit.ly/3GTToyK),
```

```
# definir la función
#    ordenaInserPila : (A, Pila[A]) -> Pila[A]
# tal que ordenaInserPila(p) es la pila obtenida ordenando por
# inserción los los elementos de la pila p. Por ejemplo,
#    >>> ordenaInserPila(apila(4, apila(1, apila(3, vacia()))))
#    1 | 3 | 4
#
# Comprobar con Hypothesis que la pila ordenaInserPila(p) está
# ordenada.
# ---------------------------------------------------------------------

# pylint: disable=unused-import

from copy import deepcopy
from typing import TypeVar

from hypothesis import given

from src.ordenadaPila import ordenadaPila
from src.TAD.pila import (Pila, apila, cima, desapila, esVacia, pilaAleatoria,
                          vacia)
from src.transformaciones_pilas_listas import listaApila, pilaAlista

A = TypeVar('A', int, float, str)

# 1ª solución
# ===========

def insertaPila(x: A, p: Pila[A]) -> Pila[A]:
    if esVacia(p):
        return apila(x, p)
    cp = cima(p)
    if x < cp:
        return apila(x, p)
    dp = desapila(p)
    return apila(cp, insertaPila(x, dp))

def ordenaInserPila1(p: Pila[A]) -> Pila[A]:
    if esVacia(p):
        return p
```

```
    cp = cima(p)
    dp = desapila(p)
    return insertaPila(cp, ordenaInserPila1(dp))

# 2ª solución
# ===========

# Se usarán las funciones listaApila y pilaAlista del ejercicio
# "Transformaciones entre pilas y listas" que se encuentra en
# https://bit.ly/3ZHewQ8

def insertaLista(x: A, ys: list[A]) -> list[A]:
    if not ys:
        return [x]
    if x < ys[0]:
        return [x] + ys
    return [ys[0]] + insertaLista(x, ys[1:])

def ordenaInserLista(xs: list[A]) -> list[A]:
    if not xs:
        return []
    return insertaLista(xs[0], ordenaInserLista(xs[1:]))

def ordenaInserPila2(p: Pila[A]) -> Pila[A]:
    return listaApila(list(reversed(ordenaInserLista(pilaAlista(p)))))

# 3ª solución
# ===========

def ordenaInserPila3Aux(p: Pila[A]) -> Pila[A]:
    if p.esVacia():
        return p
    cp = p.cima()
    p.desapila()
    return insertaPila(cp, ordenaInserPila3Aux(p))

def ordenaInserPila3(p: Pila[A]) -> Pila[A]:
    q = deepcopy(p)
    return ordenaInserPila3Aux(q)
```

```python
# Comprobación de equivalencia de las definiciones
# ================================================

# La propiedad es
@given(p=pilaAleatoria())
def test_ordenaInserPila(p: Pila[int]) -> None:
    r = ordenaInserPila1(p)
    assert ordenaInserPila2(p) == r
    assert ordenaInserPila3(p) == r

# La comprobación es
#    src> poetry run pytest -q ordenaInserPila.py
#    1 passed in 0.31s

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

# Se usará la función ordenadaPila del ejercicio
# "Reconocimiento de ordenación de pilas" que se encuentra en
# https://bit.ly/3COqRbK

# La propiedad es
@given(p=pilaAleatoria())
def test_ordenadaOrdenaInserPila(p: Pila[int]) -> None:
    ordenadaPila(ordenaInserPila1(p))

# La comprobación es
#    src> poetry run pytest -q ordenaInserPila.py
#    2 passed in 0.47s
```

6.13. Eliminación de repeticiones en una pila

6.13.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Utilizando el [tipo abstracto de datos de las pilas](https://bit.ly/3GTToyK),
-- definir la función
--    nubPila :: Eq a => Pila a -> Pila a
-- tal que (nubPila p) es la pila con los elementos de p sin repeticiones.
-- Por ejemplo,
```

```haskell
--    λ> nubPila (apila 3 (apila 1 (apila 3 (apila 5 vacia))))
--    1 | 3 | 5
-- ---------------------------------------------------------------------

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

module NubPila where

import TAD.Pila (Pila, vacia, apila, esVacia, cima, desapila)
import Transformaciones_pilas_listas (listaApila, pilaAlista)
import PertenecePila (pertenecePila)
import Data.List (nub)
import Test.QuickCheck

-- 1ª solución
-- ===========

-- Se usará la función pertenecePila del ejercicio
-- "Pertenencia a una pila" que se encuentra en
-- https://bit.ly/3WdM9GC

nubPila1 :: Eq a => Pila a -> Pila a
nubPila1 p
  | esVacia p           = vacia
  | pertenecePila cp dp = nubPila1 dp
  | otherwise           = apila cp (nubPila1 dp)
  where cp = cima p
        dp = desapila p

-- 2ª solución
-- ===========

-- Se usarán las funciones listaApila y pilaAlista del ejercicio
-- "Transformaciones entre pilas y listas" que se encuentra en
-- https://bit.ly/3ZHewQ8

nubPila2 :: Eq a => Pila a -> Pila a
nubPila2 =
  listaApila . nub . pilaAlista
```

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

-- La propiedad es
prop_nubPila :: Pila Int -> Bool
prop_nubPila p =
  nubPila1 p == nubPila2 p

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

6.13.2. En Python

```python
# ---------------------------------------------------------------------
# Utilizando el [tipo abstracto de datos de las pilas](https://bit.ly/3GTToyK),
# definir la función
#    nubPila : (Pila[A]) -> Pila[A]
# tal que nubPila(p) es la pila con los elementos de p sin repeticiones.
# Por ejemplo,
#    >>> ej = apila(3, apila(1, apila(3, apila(5, vacia()))))
#    >>> ej
#    3 | 1 | 3 | 5
#    >>> nubPila1(ej)
#    1 | 3 | 5
# ---------------------------------------------------------------------

# pylint: disable=unused-import

from copy import deepcopy
from typing import TypeVar

from hypothesis import given

from src.pertenecePila import pertenecePila
from src.TAD.pila import (Pila, apila, cima, desapila, esVacia, pilaAleatoria,
                          vacia)
from src.transformaciones_pilas_listas import listaApila, pilaAlista

A = TypeVar('A')
```

```
# 1ª solución
# ===========

# Se usará la función pertenecePila del ejercicio
# "Pertenencia a una pila" que se encuentra en
# https://bit.ly/3WdM9GC

def nubPila1(p: Pila[A]) -> Pila[A]:
    if esVacia(p):
        return p
    cp = cima(p)
    dp = desapila(p)
    if pertenecePila(cp, dp):
        return nubPila1(dp)
    return apila(cp, nubPila1(dp))

# 2ª solución
# ===========

# Se usarán las funciones listaApila y pilaAlista del ejercicio
# "Transformaciones entre pilas y listas" que se encuentra en
# https://bit.ly/3ZHewQ8

def nub(xs: list[A]) -> list[A]:
    return [x for i, x in enumerate(xs) if x not in xs[:i]]

def nubPila2(p: Pila[A]) -> Pila[A]:
    return listaApila(nub(pilaAlista(p)))

# 3ª solución
# ===========

def nubPila3Aux(p: Pila[A]) -> Pila[A]:
    if p.esVacia():
        return p
    cp = p.cima()
    p.desapila()
    if pertenecePila(cp, p):
        return nubPila3Aux(p)
```

```
    return apila(cp, nubPila3Aux(p))

def nubPila3(p: Pila[A]) -> Pila[A]:
    q = deepcopy(p)
    return nubPila3Aux(q)

# Comprobación de equivalencia de las definiciones
# ================================================

# La propiedad es
@given(p=pilaAleatoria())
def test_nubPila(p: Pila[int]) -> None:
    r = nubPila1(p)
    assert nubPila2(p) == r
    assert nubPila3(p) == r

# La comprobación es
#    src> poetry run pytest -q nubPila.py
#    1 passed in 0.27s
```

6.14. Máximo elemento de una pila

6.14.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Utilizando el [tipo abstracto de datos de las pilas](https://bit.ly/3GTToyK),
-- definir la función
--    maxPila :: Ord a => Pila a -> a
-- tal que (maxPila p) sea el mayor de los elementos de la pila p. Por
-- ejemplo,
--    λ> maxPila (apila 3 (apila 5 (apila 1 vacia)))
--    5
-- ---------------------------------------------------------------------

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

module MaxPila where

import TAD.Pila (Pila, vacia, apila, esVacia, cima, desapila)
import Transformaciones_pilas_listas (pilaAlista)
```

```haskell
import Test.QuickCheck

-- 1ª solución
-- ===========

maxPila1 :: Ord a => Pila a -> a
maxPila1 p
  | esVacia dp = cp
  | otherwise  = max cp (maxPila1 dp)
  where cp = cima p
        dp = desapila p

-- 2ª solución
-- ===========

-- Se usará la función pilaAlista del ejercicio
-- "Transformaciones entre pilas y listas" que se encuentra en
-- https://bit.ly/3ZHewQ8

maxPila2 :: Ord a => Pila a -> a
maxPila2 =
  maximum . pilaAlista

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

-- La propiedad es
prop_maxPila :: Pila Int -> Property
prop_maxPila p =
  not (esVacia p) ==> maxPila1 p == maxPila2 p

-- La comprobación es
--    λ> quickCheck prop_maxPila
--    +++ OK, passed 100 tests; 17 discarded.
```

6.14.2. En Python

```python
# ---------------------------------------------------------------------
# Utilizando el [tipo abstracto de datos de las pilas](https://bit.ly/3GTToyK),
# definir la función
```

```
#    maxPila : (Pila[A]) -> A
# tal que maxPila(p) sea el mayor de los elementos de la pila p. Por
# ejemplo,
#    >>> maxPila(apila(3, apila(5, apila(1, vacia()))))
#    5
# ---------------------------------------------------------------------

# pylint: disable=unused-import

from copy import deepcopy
from typing import TypeVar

from hypothesis import assume, given

from src.TAD.pila import (Pila, apila, cima, desapila, esVacia, pilaAleatoria,
                          vacia)
from src.transformaciones_pilas_listas import pilaAlista

A = TypeVar('A', int, float, str)

# 1ª solución
# ===========

def maxPila1(p: Pila[A]) -> A:
    cp = cima(p)
    dp = desapila(p)
    if esVacia(dp):
        return cp
    return max(cp, maxPila1(dp))

# 2ª solución
# ===========

# Se usará la función pilaAlista del ejercicio
# ”Transformaciones entre pilas y listas” que se encuentra en
# https://bit.ly/3ZHewQ8

def maxPila2(p: Pila[A]) -> A:
    return max(pilaAlista(p))
```

```
# 3ª solución
# ===========

def maxPila3Aux(p: Pila[A]) -> A:
    cp = p.cima()
    p.desapila()
    if esVacia(p):
        return cp
    return max(cp, maxPila3Aux(p))

def maxPila3(p: Pila[A]) -> A:
    q = deepcopy(p)
    return maxPila3Aux(q)

# 4ª solución
# ===========

def maxPila4Aux(p: Pila[A]) -> A:
    r = p.cima()
    while not esVacia(p):
        cp = p.cima()
        if cp > r:
            r = cp
        p.desapila()
    return r

def maxPila4(p: Pila[A]) -> A:
    q = deepcopy(p)
    return maxPila4Aux(q)

# Comprobación de equivalencia de las definiciones
# ================================================

# La propiedad es
@given(p=pilaAleatoria())
def test_maxPila(p: Pila[int]) -> None:
    assume(not esVacia(p))
    r = maxPila1(p)
    assert maxPila2(p) == r
    assert maxPila3(p) == r
```

```
    assert maxPila4(p) == r

# La comprobación es
#    src> poetry run pytest -q maxPila.py
#    1 passed in 0.25s
```

Capítulo 7

El tipo abstracto de datos de las colas

Contenido

7.1. El tipo abstracto de datos de las colas

7.1.1. En Haskell

```haskell
-- Una cola es una estructura de datos, caracterizada por ser una
-- secuencia de elementos en la que la operación de inserción se realiza
-- por un extremo (el posterior o final) y la operación de extracción
-- por el otro (el anterior o frente).
--
-- Las operaciones que definen a tipo abstracto de datos (TAD) de las
-- colas (cuyos elementos son del tipo a) son las siguientes:
--    vacia   :: Cola a
--    inserta :: a -> Cola a -> Cola a
--    primero :: Cola a -> a
--    resto   :: Cola a -> Cola a
--    esVacia :: Cola a -> Bool
-- tales que
--    + vacia es la cola vacía.
--    + (inserta x c) es la cola obtenida añadiendo x al final de c.
--    + (primero c) es el primero de la cola c.
--    + (resto c) es la cola obtenida eliminando el primero de c.
--    + (esVacia c) se verifica si c es la cola vacía.
--
-- Las operaciones tienen que verificar las siguientes propiedades:
--    + primero (inserta x vacia) == x
--    + Si c es una cola no vacía, entonces primero (inserta x c) == primero c,
--    + resto (inserta x vacia) == vacia
--    + Si c es una cola no vacía, entonces resto (inserta x c) == inserta x (res
--    + esVacia vacia
--    + not (esVacia (inserta x c))
--
-- Para usar el TAD hay que usar una implementación concreta. En
-- principio, consideraremos dos: una usando listas y otra usando
-- sucesiones. Hay que elegir la que se desee utilizar, descomentándola
```

```
-- y comentando las otras.

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

module TAD.Cola
  (Cola,
   vacia,      -- Cola a
   inserta,    -- a -> Cola a -> Cola a
   primero,    -- Cola a -> a
   resto,      -- Cola a -> Cola a
   esVacia,    -- Cola a -> Bool
  ) where

import TAD.ColaConListas
-- import TAD.ColaConDosListas
-- import TAD.ColaConSucesiones
```

7.1.2. En Python

```
# Una cola es una estructura de datos, caracterizada por ser una
# secuencia de elementos en la que la operación de inserción se realiza
# por un extremo (el posterior o final) y la operación de extracción
# por el otro (el anterior o frente).
#
# Las operaciones que definen a tipo abstracto de datos (TAD) de las
# colas (cuyos elementos son del tipo a) son las siguientes:
#    vacia   :: Cola a
#    inserta :: a -> Cola a -> Cola a
#    primero :: Cola a -> a
#    resto   :: Cola a -> Cola a
#    esVacia :: Cola a -> Bool
# tales que
#    + vacia es la cola vacía.
#    + (inserta x c) es la cola obtenida añadiendo x al final de c.
#    + (primero c) es el primero de la cola c.
#    + (resto c) es la cola obtenida eliminando el primero de c.
#    + (esVacia c) se verifica si c es la cola vacía.
#
# Las operaciones tienen que verificar las siguientes propiedades:
#    + primero (inserta x vacia) == x
```

```
#    + Si c es una cola no vacía, entonces primero (inserta x c) == primero c,
#    + resto (inserta x vacia) == vacia
#    + Si c es una cola no vacía, entonces resto (inserta x c) == inserta x (rest
#    + esVacia vacia
#    + not (esVacia (inserta x c))
#
# Para usar el TAD hay que usar una implementación concreta. En
# principio, consideraremos dos: una usando listas y otra usando
# sucesiones. Hay que elegir la que se desee utilizar, descomentándola
# y comentando las otras.

__all__ = [
    'Cola',
    'vacia',
    'inserta',
    'primero',
    'resto',
    'esVacia',
    'colaAleatoria'
]
from src.TAD.colaConListas import (Cola, colaAleatoria, esVacia, inserta,
                                   primero, resto, vacia)

# from src.TAD.colaConDosListas import (Cola, colaAleatoria, esVacia, inserta,
#                                        primero, resto, vacia)
# from src.TAD.colaConDeque import (Cola, vacia, inserta, primero, resto,
#                                   esVacia, colaAleatoria)
```

7.2. El tipo de datos de las colas mediante listas

7.2.1. En Haskell

```
{-# LANGUAGE FlexibleInstances #-}
{-# OPTIONS_GHC -fno-warn-unused-top-binds #-}

module TAD.ColaConListas
  (Cola,
   vacia,    -- Cola a
```

```
   inserta, -- a -> Cola a -> Cola a
   primero, -- Cola a -> a
   resto,   -- Cola a -> Cola a
   esVacia, -- Cola a -> Bool
  ) where

import Test.QuickCheck

-- Colas como listas:
newtype Cola a = C [a]
  deriving Eq

-- (escribeCola c) es la cadena correspondiente a la cola c. Por
-- ejemplo,
--    escribeCola (inserta 5 (inserta 2 (inserta 3 vacia))) == "3 | 2 | 5"
escribeCola :: Show a => Cola a -> String
escribeCola (C [])     = "-"
escribeCola (C [x])    = show x
escribeCola (C (x:xs)) = show x ++ " | " ++ escribeCola (C xs)

-- Procedimiento de escritura de colas.
instance Show a => Show (Cola a) where
  show = escribeCola

-- Ejemplo de cola:
--    λ> inserta 5 (inserta 2 (inserta 3 vacia))
--    3 | 2 | 5

-- vacia es la cola vacía. Por ejemplo,
--    λ> vacia
--    -
vacia :: Cola a
vacia = C []

-- (inserta x c) es la cola obtenida añadiendo x al final de la cola
-- c. Por ejemplo,
--    λ> ej = inserta 2 (inserta 3 vacia)
--    λ> ej
--    3 | 2
--    λ> inserta 5 ej
```

```
--    3 | 2 | 5
inserta :: a -> Cola a -> Cola a
inserta x (C c) = C (c ++ [x])

-- (primero c) es el primer elemento de la cola c. Por ejemplo,
--    λ> primero (inserta 5 (inserta 2 (inserta 3 vacia)))
--    3
primero :: Cola a -> a
primero (C [])    = error "primero: cola vacia"
primero (C (x:_)) = x

-- (resto c) es la cola obtenida eliminando el primer elemento de la
-- cola c. Por ejemplo,
--    λ> resto (inserta 5 (inserta 2 (inserta 3 vacia)))
--    2 | 5
resto :: Cola a -> Cola a
resto (C (_:xs)) = C xs
resto (C [])     = error "resto: cola vacia"

-- (esVacia c) se verifica si c es la cola vacía. Por ejemplo,
--    esVacia (inserta 5 (inserta 2 (inserta 3 vacia))) == False
--    esVacia vacia  == True
esVacia :: Cola a -> Bool
esVacia (C xs)  = null xs

-- Generador de colas
-- ==================

-- genCola es un generador de colas de enteros. Por ejemplo,
--    λ> sample genCola
--    -
--    -
--    -3 | 2
--    6 | 0 | 1
--    -5 | 0 | -5 | 0 | -4
--    2 | 9 | -6 | 9 | 0 | -1
--    -
--    11 | -5 | 5
--    -
--    16 | 6 | 15 | -3 | -9
```

```
--    11 | 6 | 15 | 13 | 20 | -7 | 11 | -5 | 13
genCola :: (Arbitrary a, Num a) => Gen (Cola a)
genCola = do
  xs <- listOf arbitrary
  return (foldr inserta vacia xs)

-- El tipo cola es una instancia del arbitrario.
instance (Arbitrary a, Num a) => Arbitrary (Cola a) where
  arbitrary = genCola

-- Propiedades de las colas
-- ========================

-- Las propiedades son
prop_colas1 :: Int -> Cola Int -> Bool
prop_colas1 x c =
  primero (inserta x vacia) == x &&
  resto (inserta x vacia) == vacia &&
  esVacia vacia &&
  not (esVacia (inserta x c))

prop_colas2 :: Int -> Cola Int -> Property
prop_colas2 x c =
  not (esVacia c) ==>
  primero (inserta x c) == primero c &&
  resto (inserta x c) == inserta x (resto c)

-- La comprobación es:
--    λ> quickCheck prop_colas1
--    +++ OK, passed 100 tests.
--    λ> quickCheck prop_colas2
--    +++ OK, passed 100 tests; 3 discarded.
```

7.2.2. En Python

```
# Se define la clase Cola con los siguientes métodos:
#    + inserta(x) añade x al final de la cola.
#    + primero() es el primero de la cola.
#    + resto() elimina el primero de la cola.
#    + esVacia() se verifica si la cola es vacía.
```

```
# Por ejemplo,
#    >>> c = Cola()
#    >>> c
#    -
#    >>> c.inserta(5)
#    >>> c.inserta(2)
#    >>> c.inserta(3)
#    >>> c.inserta(4)
#    >>> c
#    5 | 2 | 3 | 4
#    >>> c.primero()
#    5
#    >>> c.resto()
#    >>> c
#    2 | 3 | 4
#    >>> c.esVacia()
#    False
#    >>> c = Cola()
#    >>> c.esVacia()
#    True
#
# Además se definen las correspondientes funciones. Por ejemplo,
#    >>> vacia()
#    -
#    >>> inserta(4, inserta(3, inserta(2, inserta(5, vacia()))))
#    5 | 2 | 3 | 4
#    >>> inserta(4, inserta(3, inserta(2, inserta(5, vacia()))))
#    5
#    >>> resto(inserta(4, inserta(3, inserta(2, inserta(5, vacia())))))
#    2 | 3 | 4
#    >>> esVacia(inserta(4, inserta(3, inserta(2, inserta(5, vacia())))))
#    False
#    >>> esVacia(vacia())
#    True
#
# Finalmente, se define un generador aleatorio de colas y se comprueba
# que las colas cumplen las propiedades de su especificación.

__all__ = [
    'Cola',
```

```
    'vacia',
    'inserta',
    'primero',
    'resto',
    'esVacia',
    'colaAleatoria'
]

from copy import deepcopy
from dataclasses import dataclass, field
from typing import Generic, TypeVar

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

A = TypeVar('A')

# Clase de las colas mediante listas
# ==================================

@dataclass
class Cola(Generic[A]):
    _elementos: list[A] = field(default_factory=list)

    def __repr__(self) -> str:
        """
        Devuelve una cadena con los elementos de la cola separados por " | ".
        Si la cola está vacía, devuelve "-".
        """
        if not self._elementos:
            return '-'
        return ' | '.join(str(x) for x in self._elementos)

    def inserta(self, x: A) -> None:
        """
        Inserta el elemento x al final de la cola.
        """
        self._elementos.append(x)

    def esVacia(self) -> bool:
```

```
        """
        Comprueba si la cola está vacía.

        Devuelve True si la cola está vacía, False en caso contrario.
        """
        return not self._elementos

    def primero(self) -> A:
        """
        Devuelve el primer elemento de la cola.
        """
        return self._elementos[0]

    def resto(self) -> None:
        """
        Elimina el primer elemento de la cola
        """
        self._elementos.pop(0)

# Funciones del tipo de las listas
# ================================

def vacia() -> Cola[A]:
    """
    Crea y devuelve una cola vacía de tipo A.
    """
    c: Cola[A] = Cola()
    return c

def inserta(x: A, c: Cola[A]) -> Cola[A]:
    """
    Inserta un elemento x en la cola c y devuelve una nueva cola con
    el elemento insertado.
    """
    _aux = deepcopy(c)
    _aux.inserta(x)
    return _aux

def esVacia(c: Cola[A]) -> bool:
    """
```

```python
    Devuelve True si la cola está vacía, False si no lo está.
    """
    return c.esVacia()

def primero(c: Cola[A]) -> A:
    """
    Devuelve el primer elemento de la cola c.
    """
    return c.primero()

def resto(c: Cola[A]) -> Cola[A]:
    """
    Elimina el primer elemento de la cola c y devuelve una copia de la
    cola resultante.
    """
    _aux = deepcopy(c)
    _aux.resto()
    return _aux

# Generador de colas
# ==================

def colaAleatoria() -> st.SearchStrategy[Cola[int]]:
    """
    Genera una estrategia de búsqueda para generar colas de enteros de
    forma aleatoria.

    Utiliza la librería Hypothesis para generar una lista de enteros y
    luego se convierte en una instancia de la clase cola.
    """
    return st.lists(st.integers()).map(Cola)

# Comprobación de las propiedades de las colas
# ============================================

# Las propiedades son
@given(c=colaAleatoria(), x=st.integers())
def test_cola1(c: Cola[int], x: int) -> None:
    assert primero(inserta(x, vacia())) == x
    assert resto(inserta(x, vacia())) == vacia()
```

```
    assert esVacia(vacia())
    assert not esVacia(inserta(x, c))

@given(c=colaAleatoria(), x=st.integers())
def test_cola2(c: Cola[int], x: int) -> None:
    assume(not esVacia(c))
    assert primero(inserta(x, c)) == primero(c)
    assert resto(inserta(x, c)) == inserta(x, resto(c))

# La comprobación es
#    > poetry run pytest -q colaConListas.py
#    1 passed in 0.24s
```

7.3. El tipo de datos de las colas mediante dos listas

7.3.1. En Haskell

```
-- listas (xs,ys) de modo que los elementos de c son, en ese orden, los
-- elementos de la lista xs++(reverse ys).
--
-- Al dividir la lista en dos parte e invertir la segunda de ellas,
-- esperamos hacer más eficiente las operaciones sobre las colas.
--
-- Impondremos también una restricción adicional sobre la
-- representación: las colas serán representadas mediante pares (xs,ys)
-- tales que si xs es vacía, entonces ys será también vacía. Esta
-- restricción ha de ser conservada por los programas que crean colas.

{-# OPTIONS_GHC -fno-warn-unused-top-binds #-}

module TAD.ColaConDosListas
  (Cola,
   vacia,      -- Cola a
   inserta,    -- a -> Cola a -> Cola a
   primero,    -- Cola a -> a
   resto,      -- Cola a -> Cola a
   esVacia,    -- Cola a -> Bool
  ) where
```

```haskell
import Test.QuickCheck

-- Las colas como pares listas.
newtype Cola a = C ([a],[a])

-- (escribeCola p) es la cadena correspondiente a la cola p. Por
-- ejemplo,
--    λ> escribeCola (inserta 5 (inserta 2 (inserta 3 vacia)))
--    "3 | 2 | 5"
escribeCola :: Show a => Cola a -> String
escribeCola (C (xs,ys)) = aux (xs ++ reverse ys)
  where aux []     = "-"
        aux [z]    = show z
        aux (z:zs) = show z ++ " | " ++ aux zs

-- Procedimiento de escritura de colas.
instance Show a => Show (Cola a) where
  show = escribeCola

-- Ejemplo de cola:
--    λ> inserta 5 (inserta 2 (inserta 3 vacia))
--    3 | 2 | 5

-- vacia es la cola vacía. Por ejemplo,
--    λ>  vacia
--    -
vacia :: Cola a
vacia  = C ([],[])

-- (inserta x c) es la cola obtenida añadiendo x al final de la cola
-- c. Por ejemplo,
--    λ> inserta 5 (inserta 2 (inserta 3 vacia))
--    3 | 2 | 5
inserta :: a -> Cola a -> Cola a
inserta y (C (xs,ys)) = C (normaliza (xs,y:ys))

-- (normaliza p) es la cola obtenida al normalizar el par de listas
-- p. Por ejemplo,
--    normaliza ([],[2,5,3])   ==  ([3,5,2],[])
```

```haskell
--    normaliza ([4],[2,5,3])  ==  ([4],[2,5,3])
normaliza :: ([a],[a]) -> ([a],[a])
normaliza ([], ys) = (reverse ys, [])
normaliza p        = p

-- (primero c) es el primer elemento de la cola c. Por ejemplo,
--    λ> primero (inserta 5 (inserta 2 (inserta 3 vacia)))
--    3
primero  :: Cola a -> a
primero (C (x:_,_)) = x
primero _           = error "primero: cola vacia"

-- (resto c) es la cola obtenida eliminando el primer elemento de la
-- cola c. Por ejemplo,
--    λ> resto (inserta 5 (inserta 2 (inserta 3 vacia)))
--    2 | 5
resto  :: Cola a -> Cola a
resto (C ([],[]))   = error "resto: cola vacia"
resto (C (_:xs,ys)) = C (normaliza (xs,ys))
resto (C ([],_:_))  = error "Imposible"

-- (esVacia c) se verifica si c es la cola vacía. Por ejemplo,
--    esVacia (inserta 5 (inserta 2 (inserta 3 vacia))) == False
--    esVacia vacia == True
esVacia :: Cola a -> Bool
esVacia (C (xs,_)) = null xs

-- (valida c) se verifica si la cola c es válida; es decir, si
-- su primer elemento es vacío entonces también lo es el segundo. Por
-- ejemplo,
--    valida (C ([2],[5]))  ==  True
--    valida (C ([2],[]))   ==  True
--    valida (C ([],[5]))   ==  False
valida :: Cola a -> Bool
valida (C (xs,ys)) = not (null xs) || null ys

-- ---------------------------------------------------------------------
-- Igualdad de colas                                                  --
-- ---------------------------------------------------------------------
```

```
-- (elementos c) es la lista de los elementos de la cola c en el orden de
-- la cola. Por ejemplo,
--    λ> elementos (inserta 5 (inserta 2 (inserta 3 vacia)))
--    [3,2,5]
elementos :: Cola a -> [a]
elementos (C (xs,ys)) = xs ++ reverse ys

-- (igualColas c1 c2) se verifica si las colas c1 y c2 son iguales. Por
-- ejemplo,
--    igualColas (C ([3,2],[5,4,7])) (C ([3],[5,4,7,2]))    ==  True
--    igualColas (C ([3,2],[5,4,7])) (C ([],[5,4,7,2,3]))   ==  False
igualColas :: Eq a => Cola a -> Cola a -> Bool
igualColas c1 c2 =
  valida c1 &&
  valida c2 &&
  elementos c1 == elementos c2

instance Eq a => Eq (Cola a) where
  (==) = igualColas

-- Generador de colas                                          --
-- ==================

-- genCola es un generador de colas de enteros. Por ejemplo,
--    λ> sample genCola
--    -
--    -
--    -3 | 2
--    6 | 0 | 1
--    -5 | 0 | -5 | 0 | -4
--    2 | 9 | -6 | 9 | 0 | -1
--    -
--    11 | -5 | 5
--    -
--    16 | 6 | 15 | -3 | -9
--    11 | 6 | 15 | 13 | 20 | -7 | 11 | -5 | 13
genCola :: (Arbitrary a, Num a) => Gen (Cola a)
genCola = do
  xs <- listOf arbitrary
  return (foldr inserta vacia xs)
```

```haskell
-- El tipo pila es una instancia del arbitrario.
instance (Arbitrary a, Num a) => Arbitrary (Cola a) where
  arbitrary = genCola

-- Propiedades de las colas
-- ========================

-- Las propiedades son
prop_colas1 :: Int -> Cola Int -> Bool
prop_colas1 x c =
  primero (inserta x vacia) == x &&
  resto (inserta x vacia) == vacia &&
  esVacia vacia &&
  not (esVacia (inserta x c))

prop_colas2 :: Int -> Cola Int -> Property
prop_colas2 x c =
  not (esVacia c) ==>
  primero (inserta x c) == primero c &&
  resto (inserta x c) == inserta x (resto c)

-- La comprobación es:
--    λ> quickCheck prop_colas1
--    +++ OK, passed 100 tests.
--    λ> quickCheck prop_colas2
--    +++ OK, passed 100 tests; 3 discarded.
```

7.3.2. En Python

```python
# Se define la clase Cola con los siguientes métodos:
#    + inserta(x) añade x al final de la cola.
#    + primero() es el primero de la cola.
#    + resto() elimina el primero de la cola.
#    + esVacia() se verifica si la cola es vacía.
# Por ejemplo,
#    >>> c = Cola()
#    >>> c
#    -
#    >>> c.inserta(5)
```

```
#    >>> c.inserta(2)
#    >>> c.inserta(3)
#    >>> c.inserta(4)
#    >>> c
#    5 | 2 | 3 | 4
#    >>> c.primero()
#    5
#    >>> c.resto()
#    >>> c
#    2 | 3 | 4
#    >>> c.esVacia()
#    False
#    >>> c = Cola()
#    >>> c.esVacia()
#    True
#
# Además se definen las correspondientes funciones. Por ejemplo,
#    >>> vacia()
#    -
#    >>> inserta(4, inserta(3, inserta(2, inserta(5, vacia()))))
#    5 | 2 | 3 | 4
#    >>> primero(inserta(4, inserta(3, inserta(2, inserta(5, vacia())))))
#    5
#    >>> resto(inserta(4, inserta(3, inserta(2, inserta(5, vacia())))))
#    2 | 3 | 4
#    >>> esVacia(inserta(4, inserta(3, inserta(2, inserta(5, vacia())))))
#    False
#    >>> esVacia(vacia())
#    True
#
# Finalmente, se define un generador aleatorio de colas y se comprueba
# que las colas cumplen las propiedades de su especificación.

__all__ = [
    'Cola',
    'vacia',
    'inserta',
    'primero',
    'resto',
    'esVacia',
```

```
    'colaAleatoria'
]

from copy import deepcopy
from dataclasses import dataclass, field
from typing import Any, Generic, TypeVar

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

A = TypeVar('A')

# Clase de las colas mediante listas
# ==================================

@dataclass
class Cola(Generic[A]):
    _primera: list[A] = field(default_factory=list)
    _segunda: list[A] = field(default_factory=list)

    def _elementos(self) -> list[A]:
        """
        Devuelve una lista con los elementos de la cola en orden.
        """
        return self._primera + self._segunda[::-1]

    def __repr__(self) -> str:
        """
        Devuelve una cadena con los elementos de la cola separados por " | ".
        Si la cola está vacía, devuelve "-".
        """
        elementos = self._elementos()
        if not elementos:
            return "-"
        return " | ".join(map(str, elementos))

    def __eq__(self, c: Any) -> bool:
        """
        Comprueba si la cola actual es igual a otra cola.
        Se considera que dos colas son iguales si tienen los mismos
```

```
        elementos en el mismo orden.

        Parámetro:
        - c (Cola): La cola con la que se va a comparar.

        Devuelve True si las dos colas son iguales, False en caso
        contrario.
        """
        return self._elementos() == c._elementos()

    def inserta(self, y: A) -> None:
        """
        Inserta el elemento y en la cola.
        """
        xs = self._primera
        ys = self._segunda
        # Si no hay elementos en la primera lista, se inserta en la segunda
        if not xs:
            ys.insert(0, y)
            # Se invierte la segunda lista y se asigna a la primera
            self._primera = ys[::-1]
            self._segunda = []
        else:
            # Si hay elementos en la primera lista, se inserta en la segunda
            ys.insert(0, y)

    def esVacia(self) -> bool:
        """
        Devuelve si la cola está vacía.
        """
        return not self._primera

    def primero(self) -> A:
        """
        Devuelve el primer elemento de la cola.
        """
        return self._primera[0]

    def resto(self) -> None:
        """
```

```
        Elimina el primer elemento de la cola.
        """
        xs = self._primera
        ys = self._segunda
        del xs[0]
        if not xs:
            self._primera = ys[::-1]
            self._segunda = []

# Funciones del tipo de las listas
# ================================

def vacia() -> Cola[A]:
    """
    Crea y devuelve una cola vacía de tipo A.
    """
    c: Cola[A] = Cola()
    return c

def inserta(x: A, c: Cola[A]) -> Cola[A]:
    """
    Inserta un elemento x en la cola c y devuelve una nueva cola con
    el elemento insertado.
    """
    _aux = deepcopy(c)
    _aux.inserta(x)
    return _aux

def esVacia(c: Cola[A]) -> bool:
    """
    Devuelve True si la cola está vacía, False si no lo está.
    """
    return c.esVacia()

def primero(c: Cola[A]) -> A:
    """
    Devuelve el primer elemento de la cola c.
    """
    return c.primero()
```

```
def resto(c: Cola[A]) -> Cola[A]:
    """
    Elimina el primer elemento de la cola c y devuelve una copia de la
    cola resultante.
    """
    _aux = deepcopy(c)
    _aux.resto()
    return _aux

# Generador de colas
# ==================

def colaAleatoria() -> st.SearchStrategy[Cola[int]]:
    """
    Genera una estrategia de búsqueda para generar colas de enteros de
    forma aleatoria.

    Utiliza la librería Hypothesis para generar una lista de enteros y
    luego se convierte en una instancia de la clase cola.
    """
    return st.lists(st.integers()).map(Cola)

# Comprobación de las propiedades de las colas
# ============================================

# Las propiedades son
@given(c=colaAleatoria(), x=st.integers())
def test_cola1(c: Cola[int], x: int) -> None:
    assert primero(inserta(x, vacia())) == x
    assert resto(inserta(x, vacia())) == vacia()
    assert esVacia(vacia())
    assert not esVacia(inserta(x, c))

@given(c=colaAleatoria(), x=st.integers())
def test_cola2(c: Cola[int], x: int) -> None:
    assume(not esVacia(c))
    assert primero(inserta(x, c)) == primero(c)
    assert resto(inserta(x, c)) == inserta(x, resto(c))

# La comprobación es
```

```
#    > poetry run pytest -q colaConListas.py
#    2 passed in 0.40s
```

7.4. El tipo de datos de las colas mediante sucesiones

7.4.1. En Haskell

```haskell
{-# LANGUAGE FlexibleInstances #-}
{-# OPTIONS_GHC -fno-warn-unused-top-binds #-}

module TAD.ColaConSucesiones
  (Cola,
   vacia,   -- Cola a
   inserta, -- a -> Cola a -> Cola a
   primero, -- Cola a -> a
   resto,   -- Cola a -> Cola a
   esVacia, -- Cola a -> Bool
  ) where

import Data.Sequence as S
import Test.QuickCheck

-- Colas como sucesiones:
newtype Cola a = C (Seq a)
  deriving Eq

-- (escribeCola c) es la cadena correspondiente a la cola c. Por
-- ejemplo,
--    escribeCola (inserta 5 (inserta 2 (inserta 3 vacia))) == "3 | 2 | 5"
escribeCola :: Show a => Cola a -> String
escribeCola (C xs) = case viewl xs of
    EmptyL   -> "-"
    x :< xs' -> case viewl xs' of
        EmptyL -> show x
        _      -> show x ++ " | " ++ escribeCola (C xs')

-- Procedimiento de escritura de colas.
instance Show a => Show (Cola a) where
```

```
  show = escribeCola

-- Ejemplo de cola:
--    λ> inserta 5 (inserta 2 (inserta 3 vacia))
--    3 | 2 | 5

-- vacia es la cola vacía. Por ejemplo,
--    λ> vacia
--    C []
vacia :: Cola a
vacia = C empty

-- (inserta x c) es la cola obtenida añadiendo x al final de la cola
-- c. Por ejemplo,
--    λ> ej = inserta 2 (inserta 3 vacia)
--    λ> ej
--    3 | 2
--    λ> inserta 5 ej
--    3 | 2 | 5
inserta :: a -> Cola a -> Cola a
inserta x (C xs) = C (xs |> x )

-- (primero c) es el primer elemento de la cola c. Por ejemplo,
--    λ> primero (inserta 5 (inserta 2 (inserta 3 vacia)))
--    3
primero :: Cola a -> a
primero (C xs) = case viewl xs of
  EmptyL -> error "primero de la pila vacia"
  x :< _ -> x

-- (resto c) es la cola obtenida eliminando el primer elemento de la
-- cola c. Por ejemplo,
--    λ> resto (inserta 5 (inserta 2 (inserta 3 vacia)))
--    2 | 5
resto :: Cola a -> Cola a
resto (C xs) = case viewl xs of
  EmptyL   -> error "resto la pila vacia"
  _ :< xs' -> C xs'

-- (esVacia c) se verifica si c es la cola vacía. Por ejemplo,
```

```haskell
--    esVacia (inserta 5 (inserta 2 (inserta 3 vacia))) == False
--    esVacia vacia  == True
esVacia :: Cola a -> Bool
esVacia (C xs)  = S.null xs

-- Generador de colas                                          --
-- ==================

-- genCola es un generador de colas de enteros. Por ejemplo,
--    λ> sample genCola
--    -
--    2 | -2
--    0 | 0 | 0 | 4
--    -
--    2
--    -1 | -6 | 9
--    12 | -12 | -12 | 7 | -2 | -3 | 5 | -8 | -3 | -9 | -6
--    -11 | -5 | -7 | -8 | -10 | 8 | -9 | -7 | 6 | -12 | 8 | -9 | -1
--    -16 | -12
--    -17 | -17 | 1 | 2 | -15 | -15 | -13 | 8 | 13 | -12 | 15
--    -16 | -18
genCola :: (Arbitrary a, Num a) => Gen (Cola a)
genCola = do
  xs <- listOf arbitrary
  return (foldr inserta vacia xs)

-- El tipo pila es una instancia del arbitrario.
instance (Arbitrary a, Num a) => Arbitrary (Cola a) where
  arbitrary = genCola

-- Propiedades de las colas
-- ========================

-- Las propiedades son
prop_colas1 :: Int -> Cola Int -> Bool
prop_colas1 x c =
  primero (inserta x vacia) == x &&
  resto (inserta x vacia) == vacia &&
  esVacia vacia &&
  not (esVacia (inserta x c))
```

```haskell
prop_colas2 :: Int -> Cola Int -> Property
prop_colas2 x c =
  not (esVacia c) ==>
  primero (inserta x c) == primero c &&
  resto (inserta x c) == inserta x (resto c)

-- La comprobación es:
--    λ> quickCheck prop_colas1
--    +++ OK, passed 100 tests.
--    λ> quickCheck prop_colas2
--    +++ OK, passed 100 tests; 9 discarded.
```

7.4.2. En Python

```python
# Se define la clase Cola con los siguientes métodos:
#    + inserta(x) añade x al final de la cola.
#    + primero() es el primero de la cola.
#    + resto() elimina el primero de la cola.
#    + esVacia() se verifica si la cola es vacía.
# Por ejemplo,
#    >>> c = Cola()
#    >>> c
#    -
#    >>> c.inserta(5)
#    >>> c.inserta(2)
#    >>> c.inserta(3)
#    >>> c.inserta(4)
#    >>> c
#    5 | 2 | 3 | 4
#    >>> c.primero()
#    5
#    >>> c.resto()
#    >>> c
#    2 | 3 | 4
#    >>> c.esVacia()
#    False
#    >>> c = Cola()
#    >>> c.esVacia()
#    True
```

```
#
# Además se definen las correspondientes funciones. Por ejemplo,
#    >>> vacia()
#    -
#    >>> inserta(4, inserta(3, inserta(2, inserta(5, vacia()))))
#    5 | 2 | 3 | 4
#    >>> primero(inserta(4, inserta(3, inserta(2, inserta(5, vacia())))))
#    5
#    >>> resto(inserta(4, inserta(3, inserta(2, inserta(5, vacia())))))
#    2 | 3 | 4
#    >>> esVacia(inserta(4, inserta(3, inserta(2, inserta(5, vacia())))))
#    False
#    >>> esVacia(vacia())
#    True
#
# Finalmente, se define un generador aleatorio de colas y se comprueba
# que las colas cumplen las propiedades de su especificación.

__all__ = [
    'Cola',
    'vacia',
    'inserta',
    'primero',
    'resto',
    'esVacia',
    'colaAleatoria'
]

from collections import deque
from copy import deepcopy
from dataclasses import dataclass, field
from typing import Generic, TypeVar

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

A = TypeVar('A')

# Clase de las colas mediante deque
# ==================================
```

```
@dataclass
class Cola(Generic[A]):
    _elementos: deque[A] = field(default_factory=deque)

    def __repr__(self) -> str:
        """
        Devuelve una cadena con los elementos de la cola separados por " | ".
        Si la cola está vacía, devuelve "-".
        """
        if self.esVacia():
            return '-'
        return ' | '.join(map(str, self._elementos))

    def inserta(self, x: A) -> None:
        """
        Inserta el elemento x en la cola.
        """
        self._elementos.append(x)

    def esVacia(self) -> bool:
        """
        Devuelve si la cola está vacía.
        """
        return not self._elementos

    def primero(self) -> A:
        """
        Devuelve el primer elemento de la cola.
        """
        return self._elementos[0]

    def resto(self) -> None:
        """
        Elimina el primer elemento de la cola.
        """
        self._elementos.popleft()

# Funciones del tipo de las deque
# ================================
```

```
def vacia() -> Cola[A]:
    """
    Crea y devuelve una cola vacía de tipo A.
    """
    c: Cola[A] = Cola()
    return c

def inserta(x: A, c: Cola[A]) -> Cola[A]:
    """
    Inserta un elemento x en la cola c y devuelve una nueva cola con
    el elemento insertado.
    """
    _aux = deepcopy(c)
    _aux.inserta(x)
    return _aux

def esVacia(c: Cola[A]) -> bool:
    """
    Devuelve True si la cola está vacía, False si no lo está.
    """
    return c.esVacia()

def primero(c: Cola[A]) -> A:
    """
    Devuelve el primer elemento de la cola c.
    """
    return c.primero()

def resto(c: Cola[A]) -> Cola[A]:
    """
    Elimina el primer elemento de la cola c y devuelve una copia de la
    cola resultante.
    """
    _aux = deepcopy(c)
    _aux.resto()
    return _aux

# Generador de colas
# ==================
```

```python
def colaAleatoria() -> st.SearchStrategy[Cola[int]]:
    """
    Genera una cola aleatoria de enteros utilizando el módulo "hypothesis".

    Utiliza la función "builds" para construir una cola a partir de una lista
    de enteros generada aleatoriamente.
    """
    def _creaCola(elementos: list[int]) -> Cola[int]:
        """
        Crea una cola de enteros a partir de una lista de elementos.
        """
        cola: Cola[int] = vacia()
        for x in elementos:
            cola = inserta(x, cola)
        return cola
    return st.builds(_creaCola, st.lists(st.integers()))

# Comprobación de las propiedades de las colas
# ============================================

# Las propiedades son
@given(c=colaAleatoria(), x=st.integers())
def test_cola1(c: Cola[int], x: int) -> None:
    assert primero(inserta(x, vacia())) == x
    assert resto(inserta(x, vacia())) == vacia()
    assert esVacia(vacia())
    assert not esVacia(inserta(x, c))

@given(c=colaAleatoria(), x=st.integers())
def test_cola2(c: Cola[int], x: int) -> None:
    assume(not esVacia(c))
    assert primero(inserta(x, c)) == primero(c)
    assert resto(inserta(x, c)) == inserta(x, resto(c))

# La comprobación es
#    > poetry run pytest -q colaConDeque.py
#    1 passed in 0.24s
```

7.5. Transformaciones entre colas y listas

7.5.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Utilizando el [tipo abstracto de datos de las colas](https://bit.ly/3QWTsRL),
-- definir las funciones
--    listaAcola :: [a] -> Cola a
--    colaAlista :: Cola a -> [a]
-- tales que
-- + (listaAcola xs) es la cola formada por los elementos de xs.
--   Por ejemplo,
--      λ> listaAcola [3, 2, 5]
--      3 | 2 | 5
-- + (colaAlista c) es la lista formada por los elementos de la
--   cola c. Por ejemplo,
--      λ> colaAlista (inserta 5 (inserta 2 (inserta 3 vacia)))
--      [3, 2, 5]
--
-- Comprobar con QuickCheck que ambas funciones son inversa; es decir,
--    colaAlista (listaAcola xs) = xs
--    listaAcola (colaAlista c)  = c
-- ---------------------------------------------------------------------

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

module Transformaciones_colas_listas where

import TAD.Cola (Cola, vacia, inserta, esVacia, primero, resto)
import Test.QuickCheck

-- 1ª definición de listaAcola
-- ===========================

listaAcola :: [a] -> Cola a
listaAcola ys = aux (reverse ys)
  where aux []     = vacia
        aux (x:xs) = inserta x (aux xs)

-- 2ª definición de listaAcola
-- ===========================
```

```
listaAcola2 :: [a] -> Cola a
listaAcola2 = aux . reverse
  where aux [] = vacia
        aux (x:xs) = inserta x (aux xs)

-- 3ª definición de listaAcola
-- ===========================

listaAcola3 :: [a] -> Cola a
listaAcola3 = aux . reverse
  where aux = foldr inserta vacia

-- 4ª definición de listaAcola
-- ===========================

listaAcola4 :: [a] -> Cola a
listaAcola4 xs = foldr inserta vacia (reverse xs)

-- 5ª definición de listaAcola
-- ===========================

listaAcola5 :: [a] -> Cola a
listaAcola5 = foldr inserta vacia . reverse

-- Comprobación de equivalencia de las definiciones de listaAcola
-- ==============================================================

-- La propiedad es
prop_listaAcola :: [Int] -> Bool
prop_listaAcola xs =
  all (== listaAcola xs)
      [listaAcola2 xs,
       listaAcola3 xs,
       listaAcola4 xs,
       listaAcola5 xs]

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

```haskell
-- Definición de colaAlista
-- ========================

colaAlista :: Cola a -> [a]
colaAlista c
  | esVacia c = []
  | otherwise = pc : colaAlista rc
  where pc = primero c
        rc = resto c

-- Comprobación de las propiedades
-- ===============================

-- La primera propiedad es
prop_1_listaAcola :: [Int] -> Bool
prop_1_listaAcola xs =
  colaAlista (listaAcola xs) == xs

-- La comprobación es
--    λ> quickCheck prop_1_listaAcola
--    +++ OK, passed 100 tests.

-- La segunda propiedad es
prop_2_listaAcola :: Cola Int -> Bool
prop_2_listaAcola c =
  listaAcola (colaAlista c) == c

-- La comprobación es
--    λ> quickCheck prop_2_listaAcola
--    +++ OK, passed 100 tests.
```

7.5.2. En Python

```python
# ---------------------------------------------------------------------
# Utilizando el [tipo abstracto de datos de las colas](https://bit.ly/3GTToyK)
# definir las funciones
#    listaAcola : (list[A]) -> Cola[A]
#    colaAlista : (Cola[A]) -> list[A]
# tales que
```

```
# + listaAcola(xs) es la cola formada por los elementos de xs.
#   Por ejemplo,
#      >>> listaAcola([3, 2, 5])
#      3 | 2 | 5
# + colaAlista(c) es la lista formada por los elementos de la
#   cola c. Por ejemplo,
#      >>> ej = inserta(5, inserta(2, inserta(3, vacia())))
#      >>> colaAlista(ej)
#      [3, 2, 5]
#      >>> ej
#      3 | 2 | 5
#
# Comprobar con Hypothesis que ambas funciones son inversas; es decir,
#    colaAlista(listaAcola(xs)) == xs
#    listaAcola(colaAlista(c))  == c
# ---------------------------------------------------------------------

from copy import deepcopy
from typing import TypeVar

from hypothesis import given
from hypothesis import strategies as st

from src.TAD.cola import (Cola, colaAleatoria, esVacia, inserta, primero,
                          resto, vacia)

A = TypeVar('A')

# 1ª definición de listaAcola
# ===========================

def listaAcola(ys: list[A]) -> Cola[A]:
    def aux(xs: list[A]) -> Cola[A]:
        if not xs:
            return vacia()
        return inserta(xs[0], aux(xs[1:]))

    return aux(list(reversed(ys)))

# 2ª solución de listaAcola
```

```
# =========================

def listaAcola2(xs: list[A]) -> Cola[A]:
    p: Cola[A] = Cola()
    for x in xs:
        p.inserta(x)
    return p

# Comprobación de equivalencia de las definiciones de listaAcola
# ==============================================================

# La propiedad es
@given(st.lists(st.integers()))
def test_listaAcola(xs: list[int]) -> None:
    assert listaAcola(xs) == listaAcola2(xs)

# 1ª definición de colaAlista
# ===========================

def colaAlista(c: Cola[A]) -> list[A]:
    if esVacia(c):
        return []
    pc = primero(c)
    rc = resto(c)
    return [pc] + colaAlista(rc)

# 2ª definición de colaAlista
# ===========================

def colaAlista2Aux(c: Cola[A]) -> list[A]:
    if c.esVacia():
        return []
    pc = c.primero()
    c.resto()
    return [pc] + colaAlista2Aux(c)

def colaAlista2(c: Cola[A]) -> list[A]:
    c1 = deepcopy(c)
    return colaAlista2Aux(c1)
```

```python
# 3ª definición de colaAlista
# ===========================

def colaAlista3Aux(c: Cola[A]) -> list[A]:
    r = []
    while not c.esVacia():
        r.append(c.primero())
        c.resto()
    return r

def colaAlista3(c: Cola[A]) -> list[A]:
    c1 = deepcopy(c)
    return colaAlista3Aux(c1)

# Comprobación de equivalencia de las definiciones de colaAlista
# ==============================================================

@given(p=colaAleatoria())
def test_colaAlista(p: Cola[int]) -> None:
    assert colaAlista(p) == colaAlista2(p)
    assert colaAlista(p) == colaAlista3(p)

# Comprobación de las propiedades
# ===============================

# La primera propiedad es
@given(st.lists(st.integers()))
def test_1_listaAcola(xs: list[int]) -> None:
    assert colaAlista(listaAcola(xs)) == xs

# La segunda propiedad es
@given(c=colaAleatoria())
def test_2_listaAcola(c: Cola[int]) -> None:
    assert listaAcola(colaAlista(c)) == c

# La comprobación es
#      src> poetry run pytest -v transformaciones_colas_listas.py
#      test_listaAcola PASSED
#      test_colaAlista PASSED
#      test_1_listaAcola PASSED
```

```
#      test_2_listaAcola PASSED
```

7.6. Último elemento de una cola

7.6.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Utilizando el [tipo abstracto de datos de las colas](https://bit.ly/3QWTsRL),
-- definir la función
--    ultimoCola :: Cola a -> a
-- tal que (ultimoCola c) es el último elemento de la cola c. Por
-- ejemplo:
--    ultimoCola (inserta 3 (inserta 5 (inserta 2 vacia))) == 3
--    ultimoCola (inserta 2 vacia)                         == 2
-- ---------------------------------------------------------------------

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

module UltimoCola where

import TAD.Cola (Cola, vacia, inserta, primero, resto, esVacia)
import Transformaciones_colas_listas (colaAlista)
import Test.QuickCheck

-- 1ª solución
-- ===========

ultimoCola :: Cola a -> a
ultimoCola c
  | esVacia c  = error "cola vacia"
  | esVacia rc = pc
  | otherwise  = ultimoCola rc
  where pc = primero c
        rc = resto c

-- 2ª solución
-- ===========

-- Se usarán la función colaAlista del ejercicio
-- "Transformaciones entre colas y listas" que se encuentra en
```

```
-- https://bit.ly/3Xv0oIt

ultimoCola2 :: Cola a -> a
ultimoCola2 c
  | esVacia c  = error "cola vacia"
  | otherwise  = last (colaAlista c)

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

-- La propiedad es
prop_ultimoCola :: Cola Int -> Property
prop_ultimoCola c =
  not (esVacia c) ==> ultimoCola c == ultimoCola2 c

-- La comprobación es
--    λ> quickCheck prop_ultimoCola
--    +++ OK, passed 100 tests; 16 discarded.
```

7.6.2. En Python

```
# ---------------------------------------------------------------------
# Utilizando el [tipo abstracto de datos de las colas](https://bit.ly/3QWTsRL),
# definir la función
#    ultimoCola : (Cola[A]) -> A
# tal que ultimoCola(c) es el último elemento de la cola c. Por
# ejemplo:
#    >>> ultimoCola(inserta(3, inserta(5, inserta(2, vacia()))))
#    3
#    >>> ultimoCola(inserta(2, vacia()))
#    2
# ---------------------------------------------------------------------

# pylint: disable=unused-import

from copy import deepcopy
from typing import TypeVar

from hypothesis import assume, given
```

```
from src.TAD.cola import (Cola, colaAleatoria, esVacia, inserta, primero,
                          resto, vacia)
from src.transformaciones_colas_listas import colaAlista

A = TypeVar('A')

# 1ª solución
# ===========

def ultimoCola(c: Cola[A]) -> A:
    if esVacia(c):
        raise ValueError("cola vacia")
    pc = primero(c)
    rc = resto(c)
    if esVacia(rc):
        return pc
    return ultimoCola(rc)

# 2ª solución
# ===========

def ultimoCola2Aux(c: Cola[A]) -> A:
    if c.esVacia():
        raise ValueError("cola vacia")
    pc = primero(c)
    c.resto()
    if c.esVacia():
        return pc
    return ultimoCola2(c)

def ultimoCola2(c: Cola[A]) -> A:
    _c = deepcopy(c)
    return ultimoCola2Aux(_c)

# 3ª solución
# ===========

def ultimoCola3(c: Cola[A]) -> A:
    if esVacia(c):
        raise ValueError("cola vacia")
```

```
    while not esVacia(resto(c)):
        c = resto(c)
    return primero(c)

# 4ª solución
# ===========

def ultimoCola4Aux(c: Cola[A]) -> A:
    if c.esVacia():
        raise ValueError("cola vacia")
    r = primero(c)
    while not c.esVacia():
        c.resto()
        if not c.esVacia():
            r = primero(c)
    return r

def ultimoCola4(c: Cola[A]) -> A:
    _c = deepcopy(c)
    return ultimoCola4Aux(_c)

# 5ª solución
# ===========

# Se usarán la función colaAlista del ejercicio
# "Transformaciones entre colas y listas" que se encuentra en
# https://bit.ly/3Xv0oIt

def ultimoCola5(c: Cola[A]) -> A:
    if esVacia(c):
        raise ValueError("cola vacia")
    return colaAlista(c)[-1]

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

# La propiedad es
@given(c=colaAleatoria())
def test_ultimoCola(c: Cola[int]) -> None:
    assume(not esVacia(c))
```

```
    r = ultimoCola(c)
    assert ultimoCola2(c) == r
    assert ultimoCola3(c) == r
    assert ultimoCola4(c) == r
    assert ultimoCola5(c) == r

# La comprobación es
#      src> poetry run pytest -q ultimoCola.py
#      1 passed in 0.25s
```

7.7. Longitud de una cola

7.7.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Utilizando el [tipo abstracto de datos de las colas](https://bit.ly/3QWTsRL),
-- definir la función
--    longitudCola :: Cola a -> Int
-- tal que (longitudCola c) es el número de elementos de la cola c. Por
-- ejemplo,
--    longitudCola (inserta 4 (inserta 2 (inserta 5 vacia))) == 3
-- ---------------------------------------------------------------------

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

module LongitudCola where

import TAD.Cola (Cola, vacia, inserta, resto, esVacia)
import Transformaciones_colas_listas (colaAlista)
import Test.QuickCheck

-- 1ª solución
-- ===========

longitudCola1 :: Cola a -> Int
longitudCola1 c
  | esVacia c = 0
  | otherwise = 1 + longitudCola1 rc
  where rc = resto c
```

```haskell
-- 2ª solución
-- ===========

longitudCola2 :: Cola a -> Int
longitudCola2 = length . colaAlista

-- La función colaAlista está definida en el ejercicio
-- "Transformaciones entre colas y listas" que se encuentra en
-- https://bit.ly/3Xv0oIt

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

-- La propiedad es
prop_longitudCola :: Cola Int -> Bool
prop_longitudCola c =
  longitudCola1 c == longitudCola2 c

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

7.7.2. En Python

```python
# ---------------------------------------------------------------------
# Utilizando el [tipo abstracto de datos de las colas](https://bit.ly/3QWTsRL),
# definir la función
#    longitudCola : (Cola[A]) -> int
# tal que longitudCola(c) es el número de elementos de la cola c. Por
# ejemplo,
#    >>> longitudCola(inserta(4, inserta(2, inserta(5, vacia()))))
#    3
# ---------------------------------------------------------------------

# pylint: disable=unused-import

from copy import deepcopy
from typing import TypeVar

from hypothesis import given
```

```
from src.TAD.cola import Cola, colaAleatoria, esVacia, inserta, resto, vacia
from src.transformaciones_colas_listas import colaAlista

A = TypeVar('A')

# 1ª solución
# ===========

def longitudCola1(c: Cola[A]) -> int:
    if esVacia(c):
        return 0
    return 1 + longitudCola1(resto(c))

# 2ª solución
# ===========

def longitudCola2(c: Cola[A]) -> int:
    return len(colaAlista(c))

# 3ª solución
# ===========

def longitudCola3Aux(c: Cola[A]) -> int:
    if c.esVacia():
        return 0
    c.resto()
    return 1 + longitudCola3Aux(c)

def longitudCola3(c: Cola[A]) -> int:
    _c = deepcopy(c)
    return longitudCola3Aux(_c)

# 4ª solución
# ===========

def longitudCola4Aux(c: Cola[A]) -> int:
    r = 0
    while not esVacia(c):
        r = r + 1
```

```
        c = resto(c)
    return r

def longitudCola4(c: Cola[A]) -> int:
    _c = deepcopy(c)
    return longitudCola4Aux(_c)

# 5ª solución
# ===========

def longitudCola5Aux(c: Cola[A]) -> int:
    r = 0
    while not c.esVacia():
        r = r + 1
        c.resto()
    return r

def longitudCola5(c: Cola[A]) -> int:
    _c = deepcopy(c)
    return longitudCola5Aux(_c)

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

# La propiedad es
@given(c=colaAleatoria())
def test_longitudCola_(c: Cola[int]) -> None:
    r = longitudCola1(c)
    assert longitudCola2(c) == r
    assert longitudCola3(c) == r
    assert longitudCola4(c) == r
    assert longitudCola5(c) == r

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

7.8. Todos los elementos de la cola verifican una propiedad

7.8.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Utilizando el [tipo abstracto de datos de las colas](https://bit.ly/3QWTsRL),
-- definir la función
--    todosVerifican :: (a -> Bool) -> Cola a -> Bool
-- tal que (todosVerifican p c) se verifica si todos los elementos de la
-- cola c cumplen la propiedad p. Por ejemplo,
--    todosVerifican (>0) (inserta 3 (inserta 2 vacia))    == True
--    todosVerifican (>0) (inserta 3 (inserta (-2) vacia)) == False
-- ---------------------------------------------------------------------

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

module TodosVerifican where

import TAD.Cola (Cola, vacia, inserta, primero, resto, esVacia)
import Transformaciones_colas_listas (colaAlista, listaAcola)
import Test.QuickCheck.HigherOrder

-- 1ª solución
-- ===========

todosVerifican1 :: (a -> Bool) -> Cola a -> Bool
todosVerifican1 p c
  | esVacia c = True
  | otherwise = p pc && todosVerifican1 p rc
  where pc = primero c
        rc = resto c

-- 2ª solución
-- ===========

todosVerifican2 :: (a -> Bool) -> Cola a -> Bool
todosVerifican2 p c =
  all p (colaAlista c)
```

```haskell
-- La función colaAlista está definida en el ejercicio
-- "Transformaciones entre colas y listas" que se encuentra en
-- https://bit.ly/3Xv0oIt

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

-- La propiedad es
prop_todosVerifican :: (Int -> Bool) -> [Int] -> Bool
prop_todosVerifican p xs =
  todosVerifican1 p c == todosVerifican2 p c
  where c = listaAcola xs

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

7.8.2. En Python

```python
# ---------------------------------------------------------------------
# Utilizando el [tipo abstracto de datos de las colas](https://bit.ly/3QWTsRL),
# definir la función
#    todosVerifican : (Callable[[A], bool], Cola[A]) -> bool
# tal que todosVerifican(p, c) se verifica si todos los elementos de la
# cola c cumplen la propiedad p. Por ejemplo,
#    >>> todosVerifican(lambda x: x > 0, inserta(3, inserta(2, vacia())))
#    True
#    >>> todosVerifican(lambda x: x > 0, inserta(3, inserta(-2, vacia())))
#    False
# ---------------------------------------------------------------------

# pylint: disable=unused-import

from copy import deepcopy
from typing import Callable, TypeVar

from hypothesis import given

from src.TAD.cola import (Cola, colaAleatoria, esVacia, inserta, primero,
                          resto, vacia)
```

```
from src.transformaciones_colas_listas import colaAlista

A = TypeVar('A')

# 1ª solución
# ===========

def todosVerifican1(p: Callable[[A], bool], c: Cola[A]) -> bool:
    if esVacia(c):
        return True
    pc = primero(c)
    rc = resto(c)
    return p(pc) and todosVerifican1(p, rc)

# 2ª solución
# ===========

def todosVerifican2(p: Callable[[A], bool], c: Cola[A]) -> bool:
    return all(p(x) for x in colaAlista(c))

# La función colaAlista está definida en el ejercicio
# "Transformaciones entre colas y listas" que se encuentra en
# https://bit.ly/3Xv0oIt

# 3ª solución
# ===========

def todosVerifican3Aux(p: Callable[[A], bool], c: Cola[A]) -> bool:
    if c.esVacia():
        return True
    pc = c.primero()
    c.resto()
    return p(pc) and todosVerifican3Aux(p, c)

def todosVerifican3(p: Callable[[A], bool], c: Cola[A]) -> bool:
    _c = deepcopy(c)
    return todosVerifican3Aux(p, _c)

# 4ª solución
# ===========
```

```python
def todosVerifican4Aux(p: Callable[[A], bool], c: Cola[A]) -> bool:
    if c.esVacia():
        return True
    pc = c.primero()
    c.resto()
    return p(pc) and todosVerifican4Aux(p, c)

def todosVerifican4(p: Callable[[A], bool], c: Cola[A]) -> bool:
    _c = deepcopy(c)
    return todosVerifican4Aux(p, _c)

# 5ª solución
# ===========

def todosVerifican5Aux(p: Callable[[A], bool], c: Cola[A]) -> bool:
    while not c.esVacia():
        if not p(c.primero()):
            return False
        c.resto()
    return True

def todosVerifican5(p: Callable[[A], bool], c: Cola[A]) -> bool:
    _c = deepcopy(c)
    return todosVerifican5Aux(p, _c)

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

# La propiedad es
@given(c=colaAleatoria())
def test_filtraPila(c: Cola[int]) -> None:
    r = todosVerifican1(lambda x: x > 0, c)
    assert todosVerifican2(lambda x: x > 0, c) == r
    assert todosVerifican3(lambda x: x > 0, c) == r
    assert todosVerifican4(lambda x: x > 0, c) == r
    assert todosVerifican5(lambda x: x > 0, c) == r

# La comprobación es
#    src> poetry run pytest -q todosVerifican.py
```

```
#    1 passed in 0.25s
```

7.9. Algún elemento de la verifica una propiedad

7.9.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Utilizando el [tipo abstracto de datos de las colas](https://bit.ly/3QWTsRL),
-- definir la función
--    algunoVerifica :: (a -> Bool) -> Cola a -> Bool
-- tal que (algunoVerifica p c) se verifica si alguno de los elementos de la
-- cola c cumplen la propiedad p. Por ejemplo,
--    algunoVerifica (< 0) (inserta 3 (inserta (-2) vacia)) == True
--    algunoVerifica (< 0) (inserta 3 (inserta 2 vacia))    == False
-- ---------------------------------------------------------------------

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

module AlgunoVerifica where

import TAD.Cola (Cola, vacia, inserta, primero, resto, esVacia)
import Transformaciones_colas_listas (colaAlista, listaAcola)
import Test.QuickCheck.HigherOrder

-- 1ª solución
-- ===========

algunoVerifica1 :: (a -> Bool) -> Cola a -> Bool
algunoVerifica1 p c
  | esVacia c = False
  | otherwise = p pc || algunoVerifica1 p rc
  where pc = primero c
        rc = resto c

-- 2ª solución
-- ===========

algunoVerifica2 :: (a -> Bool) -> Cola a -> Bool
```

```haskell
algunoVerifica2 p c =
  any p (colaAlista c)

-- La función colaAlista está definida en el ejercicio
-- "Transformaciones entre colas y listas" que se encuentra en
-- https://bit.ly/3Xv0oIt

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

-- La propiedad es
prop_algunoVerifica :: (Int -> Bool) -> [Int] -> Bool
prop_algunoVerifica p xs =
  algunoVerifica1 p c == algunoVerifica2 p c
  where c = listaAcola xs

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

7.9.2. En Python

```python
# ---------------------------------------------------------------------
# Utilizando el [tipo abstracto de datos de las colas](https://bit.ly/3QWTsRL),
# definir la función
#    algunoVerifica : (Callable[[A], bool], Cola[A]) -> bool
# tal que algunoVerifica(p, c) se verifica si alguno de los elementos de la
# cola c cumplen la propiedad p. Por ejemplo,
#    >>> algunoVerifica(lambda x: x > 0, inserta(-3, inserta(2, vacia())))
#    True
#    >>> algunoVerifica(lambda x: x > 0, inserta(-3, inserta(-2, vacia())))
#    False
# ---------------------------------------------------------------------

# pylint: disable=unused-import

from copy import deepcopy
from typing import Callable, TypeVar

from hypothesis import given
```

```python
from src.TAD.cola import (Cola, colaAleatoria, esVacia, inserta, primero,
                          resto, vacia)
from src.transformaciones_colas_listas import colaAlista

A = TypeVar('A')

# 1ª solución
# ===========

def algunoVerifica1(p: Callable[[A], bool], c: Cola[A]) -> bool:
    if esVacia(c):
        return False
    pc = primero(c)
    rc = resto(c)
    return p(pc) or algunoVerifica1(p, rc)

# 2ª solución
# ===========

def algunoVerifica2(p: Callable[[A], bool], c: Cola[A]) -> bool:
    return any(p(x) for x in colaAlista(c))

# La función colaAlista está definida en el ejercicio
# "Transformaciones entre colas y listas" que se encuentra en
# https://bit.ly/3Xv0oIt

# 3ª solución
# ===========

def algunoVerifica3Aux(p: Callable[[A], bool], c: Cola[A]) -> bool:
    if c.esVacia():
        return False
    pc = c.primero()
    c.resto()
    return p(pc) or algunoVerifica3Aux(p, c)

def algunoVerifica3(p: Callable[[A], bool], c: Cola[A]) -> bool:
    _c = deepcopy(c)
    return algunoVerifica3Aux(p, _c)
```

```
# 4ª solución
# ===========

def algunoVerifica4Aux(p: Callable[[A], bool], c: Cola[A]) -> bool:
    if c.esVacia():
        return False
    pc = c.primero()
    c.resto()
    return p(pc) or algunoVerifica4Aux(p, c)

def algunoVerifica4(p: Callable[[A], bool], c: Cola[A]) -> bool:
    _c = deepcopy(c)
    return algunoVerifica4Aux(p, _c)

# 5ª solución
# ===========

def algunoVerifica5Aux(p: Callable[[A], bool], c: Cola[A]) -> bool:
    while not c.esVacia():
        if p(c.primero()):
            return True
        c.resto()
    return False

def algunoVerifica5(p: Callable[[A], bool], c: Cola[A]) -> bool:
    _c = deepcopy(c)
    return algunoVerifica5Aux(p, _c)

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

# La propiedad es
@given(c=colaAleatoria())
def test_algunoVerifica(c: Cola[int]) -> None:
    r = algunoVerifica1(lambda x: x > 0, c)
    assert algunoVerifica2(lambda x: x > 0, c) == r
    assert algunoVerifica3(lambda x: x > 0, c) == r
    assert algunoVerifica4(lambda x: x > 0, c) == r
    assert algunoVerifica5(lambda x: x > 0, c) == r
```

```
# La comprobación es
#    src> poetry run pytest -q algunoVerifica.py
#    1 passed in 0.31s
```

7.10. Extensión de colas

7.10.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Utilizando el [tipo abstracto de datos de las colas](https://bit.ly/3QWTsRL),
-- definir la función
--    extiendeCola :: Cola a -> Cola a -> Cola a
-- tal que (extiendeCola c1 c2) es la cola que resulta de poner los
-- elementos de la cola c2 a continuación de los de la cola de c1. Por
-- ejemplo,
--    λ> ej1 = inserta 3 (inserta 2 vacia)
--    λ> ej2 = inserta 5 (inserta 3 (inserta 4 vacia))
--    λ> ej1
--    2 | 3
--    λ> ej2
--    4 | 3 | 5
--    λ> extiendeCola ej1 ej2
--    2 | 3 | 4 | 3 | 5
--    λ> extiendeCola ej2 ej1
--    4 | 3 | 5 | 2 | 3
-- ---------------------------------------------------------------------

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

module ExtiendeCola where

import TAD.Cola (Cola, vacia, inserta, primero, resto, esVacia)
import Transformaciones_colas_listas (colaAlista, listaAcola)
import Test.QuickCheck

-- 1ª solución
-- ===========

extiendeCola :: Cola a -> Cola a -> Cola a
```

```haskell
extiendeCola c1 c2
  | esVacia c2 = c1
  | otherwise  = extiendeCola (inserta pc2 c1) rq2
  where pc2 = primero c2
        rq2 = resto c2

-- 2ª solución
-- ===========

extiendeCola2 :: Cola a -> Cola a -> Cola a
extiendeCola2 c1 c2 =
  listaAcola (colaAlista c1 ++ colaAlista c2)

-- Las funciones colaAlista y listaAcola están definidas en el ejercicio
-- "Transformaciones entre colas y listas" que se encuentra en
-- https://bit.ly/3Xv0oIt

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

-- La propiedad es
prop_extiendeCola :: Cola Int -> Cola Int -> Bool
prop_extiendeCola p c =
  extiendeCola p c == extiendeCola2 p c

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

7.10.2. En Python

```python
# ---------------------------------------------------------------------
# Utilizando el [tipo abstracto de datos de las colas](https://bit.ly/3QWTsRL),
# definir la función
#    extiendeCola : (Cola[A], Cola[A]) -> Cola[A]
# tal que extiendeCola(c1, c2) es la cola que resulta de poner los
# elementos de la cola c2 a continuación de los de la cola de c1. Por
# ejemplo,
#    >>> ej1 = inserta(3, inserta(2, vacia()))
#    >>> ej2 = inserta(5, inserta(3, inserta(4, vacia())))
```

```
#    >>> ej1
#    2 | 3
#    >>> ej2
#    4 | 3 | 5
#    >>> extiendeCola(ej1, ej2)
#    2 | 3 | 4 | 3 | 5
#    >>> extiendeCola(ej2, ej1)
#    4 | 3 | 5 | 2 | 3
# ---------------------------------------------------------------------

# pylint: disable=unused-import

from copy import deepcopy
from typing import TypeVar

from hypothesis import given

from src.TAD.cola import (Cola, colaAleatoria, esVacia, inserta, primero,
                          resto, vacia)
from src.transformaciones_colas_listas import colaAlista, listaAcola

A = TypeVar('A')

# 1ª solución
# ===========

def extiendeCola(c1: Cola[A], c2: Cola[A]) -> Cola[A]:
    if esVacia(c2):
        return c1
    pc2 = primero(c2)
    rc2 = resto(c2)
    return extiendeCola(inserta(pc2, c1), rc2)

# 2ª solución
# ===========

def extiendeCola2(c1: Cola[A], c2: Cola[A]) -> Cola[A]:
    return listaAcola(colaAlista(c1) + colaAlista(c2))

# Las funciones colaAlista y listaAcola están definidas en el ejercicio
```

```
# ”Transformaciones entre colas y listas” que se encuentra en
# https://bit.ly/3Xv0oIt

# 3ª solución
# ===========

def extiendeCola3Aux(c1: Cola[A], c2: Cola[A]) -> Cola[A]:
    if c2.esVacia():
        return c1
    pc2 = c2.primero()
    c2.resto()
    return extiendeCola(inserta(pc2, c1), c2)

def extiendeCola3(c1: Cola[A], c2: Cola[A]) -> Cola[A]:
    _c2 = deepcopy(c2)
    return extiendeCola3Aux(c1, _c2)

# 4ª solución
# ===========

def extiendeCola4Aux(c1: Cola[A], c2: Cola[A]) -> Cola[A]:
    r = c1
    while not esVacia(c2):
        r = inserta(primero(c2), r)
        c2 = resto(c2)
    return r

def extiendeCola4(c1: Cola[A], c2: Cola[A]) -> Cola[A]:
    _c2 = deepcopy(c2)
    return extiendeCola4Aux(c1, _c2)

# 5ª solución
# ===========

def extiendeCola5Aux(c1: Cola[A], c2: Cola[A]) -> Cola[A]:
    r = c1
    while not c2.esVacia():
        r.inserta(primero(c2))
        c2.resto()
    return r
```

```python
def extiendeCola5(c1: Cola[A], c2: Cola[A]) -> Cola[A]:
    _c1 = deepcopy(c1)
    _c2 = deepcopy(c2)
    return extiendeCola5Aux(_c1, _c2)

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

# La propiedad es
@given(c1=colaAleatoria(), c2=colaAleatoria())
def test_extiendeCola(c1: Cola[int], c2: Cola[int]) -> None:
    r = extiendeCola(c1, c2)
    assert extiendeCola2(c1, c2) == r
    assert extiendeCola3(c1, c2) == r
    assert extiendeCola4(c1, c2) == r

# La comprobación es
#    src> poetry run pytest -q extiendeCola.py
#    1 passed in 0.32s
```

7.11. Intercalado de dos colas

7.11.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Utilizando el [tipo abstracto de datos de las colas](https://bit.ly/3QWTsRL),
-- definir la función
--    intercalaColas :: Cola a -> Cola a -> Cola a
-- tal que (intercalaColas c1 c2) es la cola formada por los elementos de
-- c1 y c2 colocados en una cola, de forma alternativa, empezando por
-- los elementos de c1. Por ejemplo,
--    λ> ej1 = inserta 3 (inserta 5 vacia)
--    λ> ej2 = inserta 0 (inserta 7 (inserta 4 (inserta 9 vacia)))
--    λ> intercalaColas ej1 ej2
--    5 | 9 | 3 | 4 | 7 | 0
--    λ> intercalaColas ej2 ej1
--    9 | 5 | 4 | 3 | 7 | 0
-- ---------------------------------------------------------------------
```

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

module IntercalaColas where

import TAD.Cola (Cola, vacia, inserta, primero, resto, esVacia)
import Transformaciones_colas_listas (colaAlista, listaAcola)
import ExtiendeCola (extiendeCola)
import Test.QuickCheck

-- 1ª solución
-- ===========

intercalaColas :: Cola a -> Cola a -> Cola a
intercalaColas c1 c2
  | esVacia c1 = c2
  | esVacia c2 = c1
  | otherwise  = extiendeCola (inserta pc2 (inserta pc1 vacia))
                              (intercalaColas rc1 rc2)
  where pc1 = primero c1
        rc1 = resto c1
        pc2 = primero c2
        rc2 = resto c2

-- La función extiendeCola está definida en el ejercicio
-- ”TAD de las colas: Extensión de colas” que se encuentra en
-- https://bit.ly/3XIJJ4m

-- 2ª solución
-- ===========

intercalaColas2 :: Cola a -> Cola a -> Cola a
intercalaColas2 c1 c2 = aux c1 c2 vacia
  where
    aux d1 d2 c
      | esVacia d1 = extiendeCola c d2
      | esVacia d2 = extiendeCola c d1
      | otherwise  = aux rd1 rd2 (inserta pd2 (inserta pd1 c))
      where pd1 = primero d1
            rd1 = resto d1
            pd2 = primero d2
```

```
            rd2 = resto d2

-- 3ª solución
-- ===========

intercalaColas3 :: Cola a -> Cola a -> Cola a
intercalaColas3 c1 c2 =
  listaAcola (intercalaListas (colaAlista c1) (colaAlista c2))

-- (intercalaListas xs ys) es la lista obtenida intercalando los
-- elementos de xs e ys. Por ejemplo,
intercalaListas :: [a] -> [a] -> [a]
intercalaListas []     ys     = ys
intercalaListas xs     []     = xs
intercalaListas (x:xs) (y:ys) = x : y : intercalaListas xs ys

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

-- La propiedad es
prop_intercalaColas :: Cola Int -> Cola Int -> Bool
prop_intercalaColas c1 c2 =
  all (== intercalaColas c1 c2)
      [intercalaColas2 c1 c2,
       intercalaColas2 c1 c2]

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

7.11.2. En Python

```
# ---------------------------------------------------------------------
# Utilizando el [tipo abstracto de datos de las colas](https://bit.ly/3QWTsRL),
# definir la función
#    intercalaColas : (Cola[A], Cola[A]) -> Cola[A]
# tal que (intercalaColas c1 c2) es la cola formada por los elementos de
# c1 y c2 colocados en una cola, de forma alternativa, empezando por
# los elementos de c1. Por ejemplo,
#    >>> ej1 = inserta(3, inserta(5, vacia()))
```

```
#    >>> ej2 = inserta(0, inserta(7, inserta(4, inserta(9, vacia()))))
#    >>> intercalaColas(ej1, ej2)
#    5 | 9 | 3 | 4 | 7 | 0
#    >>> intercalaColas(ej2, ej1)
#    9 | 5 | 4 | 3 | 7 | 0
# ---------------------------------------------------------------------

from copy import deepcopy
from typing import TypeVar

from hypothesis import given

from src.extiendeCola import extiendeCola
from src.TAD.cola import (Cola, colaAleatoria, esVacia, inserta, primero,
                          resto, vacia)
from src.transformaciones_colas_listas import colaAlista, listaAcola

A = TypeVar('A')

# 1ª solución
# ===========

def intercalaColas(c1: Cola[A], c2: Cola[A]) -> Cola[A]:
    if esVacia(c1):
        return c2
    if esVacia(c2):
        return c1
    pc1 = primero(c1)
    rc1 = resto(c1)
    pc2 = primero(c2)
    rc2 = resto(c2)
    return extiendeCola(inserta(pc2, inserta(pc1, vacia())),
                        intercalaColas(rc1, rc2))

# La función extiendeCola está definida en el ejercicio
# "TAD de las colas: Extensión de colas" que se encuentra en
# https://bit.ly/3XIJJ4m

# 2ª solución
# ===========
```

```
def intercalaColas2(c1: Cola[A], c2: Cola[A]) -> Cola[A]:
    def aux(d1: Cola[A], d2: Cola[A], d3: Cola[A]) -> Cola[A]:
        if esVacia(d1):
            return extiendeCola(d3, d2)
        if esVacia(d2):
            return extiendeCola(d3, d1)
        pd1 = primero(d1)
        rd1 = resto(d1)
        pd2 = primero(d2)
        rd2 = resto(d2)
        return aux(rd1, rd2, inserta(pd2, inserta(pd1, d3)))

    return aux(c1, c2, vacia())

# 3ª solución
# ===========

# intercalaListas(xs, ys) es la lista obtenida intercalando los
# elementos de xs e ys. Por ejemplo,
def intercalaListas(xs: list[A], ys: list[A]) -> list[A]:
    if not xs:
        return ys
    if not ys:
        return xs
    return [xs[0], ys[0]] + intercalaListas(xs[1:], ys[1:])

def intercalaColas3(c1: Cola[A], c2: Cola[A]) -> Cola[A]:
    return listaAcola(intercalaListas(colaAlista(c1), colaAlista(c2)))

# 4ª solución
# ===========

def intercalaColas4Aux(c1: Cola[A], c2: Cola[A]) -> Cola[A]:
    if c1.esVacia():
        return c2
    if c2.esVacia():
        return c1
    pc1 = c1.primero()
    c1.resto()
```

```
    pc2 = c2.primero()
    c2.resto()
    return extiendeCola(inserta(pc2, inserta(pc1, vacia())),
                        intercalaColas4Aux(c1, c2))

def intercalaColas4(c1: Cola[A], c2: Cola[A]) -> Cola[A]:
    _c1 = deepcopy(c1)
    _c2 = deepcopy(c2)
    return intercalaColas4Aux(_c1, _c2)

# 5ª solución
# ===========

def intercalaColas5Aux(c1: Cola[A], c2: Cola[A]) -> Cola[A]:
    r: Cola[A] = vacia()
    while not esVacia(c1) and not esVacia(c2):
        pc1 = primero(c1)
        c1.resto()
        pc2 = primero(c2)
        c2.resto()
        r = inserta(pc2, inserta(pc1, r))
    if esVacia(c1):
        return extiendeCola(r, c2)
    return extiendeCola(r, c1)

def intercalaColas5(c1: Cola[A], c2: Cola[A]) -> Cola[A]:
    _c1 = deepcopy(c1)
    _c2 = deepcopy(c2)
    return intercalaColas5Aux(_c1, _c2)

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

# La propiedad es
@given(c1=colaAleatoria(), c2=colaAleatoria())
def test_intercalaCola(c1: Cola[int], c2: Cola[int]) -> None:
    r = intercalaColas(c1, c2)
    assert intercalaColas2(c1, c2) == r
    assert intercalaColas3(c1, c2) == r
    assert intercalaColas4(c1, c2) == r
```

```
    assert intercalaColas5(c1, c2) == r

# La comprobación es
#    src> poetry run pytest -q intercalaColas.py
#    1 passed in 0.47s
```

7.12. Agrupación de colas

7.12.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Utilizando el [tipo abstracto de datos de las colas](https://bit.ly/3QWTsRL),
-- definir la función
--    agrupaColas :: [Cola a] -> Cola a
-- tal que (agrupaColas [c1,c2,c3,...,cn]) es la cola formada mezclando
-- las colas de la lista como sigue: mezcla c1 con c2, el resultado con
-- c3, el resultado con c4, y así sucesivamente. Por ejemplo,
--    λ> ej1 = inserta 2 (inserta 5 vacia)
--    λ> ej2 = inserta 3 (inserta 7 (inserta 4 vacia))
--    λ> ej3 = inserta 9 (inserta 0 (inserta 1 (inserta 6 vacia)))
--    λ> agrupaColas []
--    -
--    λ> agrupaColas [ej1]
--    5 | 2
--    λ> agrupaColas [ej1, ej2]
--    5 | 4 | 2 | 7 | 3
--    λ> agrupaColas [ej1, ej2, ej3]
--    5 | 6 | 4 | 1 | 2 | 0 | 7 | 9 | 3
-- ---------------------------------------------------------------------

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

module AgrupaColas where

import TAD.Cola (Cola, vacia, inserta)
import IntercalaColas (intercalaColas)
import Test.QuickCheck

-- 1ª solución
-- ===========
```

```
agrupaColas1 :: [Cola a] -> Cola a
agrupaColas1 []            = vacia
agrupaColas1 [c]           = c
agrupaColas1 (c1:c2:colas) = agrupaColas1 ((intercalaColas c1 c2) : colas)

-- La función intercalaColas está definida en el ejercicio
-- "TAD de las colas: Intercalado de dos colas" que se encuentra en
-- https://bit.ly/3XYyjsM

-- 2ª solución
-- ===========

agrupaColas2 :: [Cola a] -> Cola a
agrupaColas2 = foldl intercalaColas vacia

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

-- La propiedad es
prop_agrupaColas :: [Cola Int] -> Bool
prop_agrupaColas cs =
  agrupaColas1 cs == agrupaColas2 cs

-- La comprobación es
--    λ> quickCheckWith (stdArgs {maxSize=30}) prop_agrupaColas
--    +++ OK, passed 100 tests.
```

7.12.2. En Python

```
# ---------------------------------------------------------------------
# Utilizando el [tipo abstracto de datos de las colas](https://bit.ly/3QWTsRL),
# definir la función
#    agrupaColas : (list[Cola[A]]) -> Cola[A]
# tal que (agrupaColas [c1,c2,c3,...,cn]) es la cola formada mezclando
# las colas de la lista como sigue: mezcla c1 con c2, el resultado con
# c3, el resultado con c4, y así sucesivamente. Por ejemplo,
#    >>> ej1 = inserta(2, inserta(5, vacia()))
#    >>> ej2 = inserta(3, inserta(7, inserta(4, vacia())))
#    >>> ej3 = inserta(9, inserta(0, inserta(1, inserta(6, vacia()))))
```

```
#    >>> agrupaColas([])
#    -
#    >>> agrupaColas([ej1])
#    5 | 2
#    >>> agrupaColas([ej1, ej2])
#    5 | 4 | 2 | 7 | 3
#    >>> agrupaColas([ej1, ej2, ej3])
#    5 | 6 | 4 | 1 | 2 | 0 | 7 | 9 | 3
# ---------------------------------------------------------------------

# pylint: disable=unused-import

from functools import reduce
from typing import TypeVar

from hypothesis import given
from hypothesis import strategies as st

from src.intercalaColas import intercalaColas
from src.TAD.cola import Cola, colaAleatoria, inserta, vacia

A = TypeVar('A')

# 1ª solución
# ===========

def agrupaColas1(cs: list[Cola[A]]) -> Cola[A]:
    if not cs:
        return vacia()
    if len(cs) == 1:
        return cs[0]
    return agrupaColas1([intercalaColas(cs[0], cs[1])] + cs[2:])

# La función intercalaColas está definida en el ejercicio
# ”TAD de las colas: Intercalado de dos colas” que se encuentra en
# https://bit.ly/3XYyjsM

# 2ª solución
# ===========
```

```
def agrupaColas2(cs: list[Cola[A]]) -> Cola[A]:
    return reduce(intercalaColas, cs, vacia())

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

# La propiedad es
@given(st.lists(colaAleatoria(), max_size=4))
def test_agrupaCola(cs: list[Cola[int]]) -> None:
    assert agrupaColas1(cs) == agrupaColas2(cs)

# La comprobación es
#    src> poetry run pytest -q agrupaColas.py
#    1 passed in 0.50s
```

7.13. Pertenencia a una cola

7.13.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Utilizando el [tipo abstracto de datos de las colas](https://bit.ly/3QWTsRL),
-- definir la función
--    perteneceCola :: Eq a => a -> Cola a -> Bool
-- tal que (perteneceCola x c) se verifica si x es un elemento de la
-- cola c. Por ejemplo,
--    perteneceCola 2 (inserta 5 (inserta 2 (inserta 3 vacia))) == True
--    perteneceCola 4 (inserta 5 (inserta 2 (inserta 3 vacia))) == False
-- ---------------------------------------------------------------------

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

module PerteneceCola where

import TAD.Cola (Cola, vacia, inserta, primero, resto, esVacia)
import Transformaciones_colas_listas (colaAlista, listaAcola)
import Test.QuickCheck

-- 1ª solución
-- ===========
```

```haskell
perteneceCola :: Eq a => a -> Cola a -> Bool
perteneceCola x c
  | esVacia c = False
  | otherwise = x == primero(c) || perteneceCola x (resto c)

-- 2ª solución
-- ===========

perteneceCola2 :: Eq a => a -> Cola a -> Bool
perteneceCola2 x c =
  x `elem` colaAlista c

-- La función colaAlista está definida en el ejercicio
-- "Transformaciones entre colas y listas" que se encuentra en
-- https://bit.ly/3Xv0oIt

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

-- La propiedad es
prop_perteneceCola :: Int -> Cola Int -> Bool
prop_perteneceCola x p =
  perteneceCola x p == perteneceCola2 x p

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

7.13.2. En Python

```python
# ---------------------------------------------------------------------
# Utilizando el [tipo abstracto de datos de las colas](https://bit.ly/3QWTsRL),
# definir la función
#    perteneceCola : (A, Cola[A]) -> bool
# tal que perteneceCola(x, c) se verifica si x es un elemento de la
# cola p. Por ejemplo,
#    >>> perteneceCola(2, inserta(5, inserta(2, inserta(3, vacia()))))
#    True
#    >>> perteneceCola(4, inserta(5, inserta(2, inserta(3, vacia()))))
#    False
```

```
# ---------------------------------------------------------------------

# pylint: disable=unused-import

from copy import deepcopy
from typing import TypeVar

from hypothesis import given
from hypothesis import strategies as st

from src.TAD.cola import (Cola, colaAleatoria, esVacia, inserta, primero,
                          resto, vacia)
from src.transformaciones_colas_listas import colaAlista

A = TypeVar('A')

# 1ª solución
# ===========

def perteneceCola(x: A, c: Cola[A]) -> bool:
    if esVacia(c):
        return False
    return x == primero(c) or perteneceCola(x, resto(c))

# 2ª solución
# ===========

def perteneceCola2(x: A, c: Cola[A]) -> bool:
    return x in colaAlista(c)

# Las función colaAlista está definida en el ejercicio
# ”Transformaciones entre colas y listas” que se encuentra en
# https://bit.ly/3Xv0oIt

# 3ª solución
# ===========

def perteneceCola3Aux(x: A, c: Cola[A]) -> bool:
    if c.esVacia():
        return False
```

```
    pc = c.primero()
    c.resto()
    return x == pc or perteneceCola3Aux(x, c)

def perteneceCola3(x: A, c: Cola[A]) -> bool:
    c1 = deepcopy(c)
    return perteneceCola3Aux(x, c1)

# 4ª solución
# ===========

def perteneceCola4Aux(x: A, c: Cola[A]) -> bool:
    while not c.esVacia():
        pc = c.primero()
        c.resto()
        if x == pc:
            return True
    return False

def perteneceCola4(x: A, c: Cola[A]) -> bool:
    c1 = deepcopy(c)
    return perteneceCola4Aux(x, c1)

# Comprobación de equivalencia de las definiciones
# ================================================

# La propiedad es
@given(x=st.integers(), c=colaAleatoria())
def test_perteneceCola(x: int, c: Cola[int]) -> None:
    r = perteneceCola(x, c)
    assert perteneceCola2(x, c) == r
    assert perteneceCola3(x, c) == r
    assert perteneceCola4(x, c) == r

# La comprobación es
#    src> poetry run pytest -q perteneceCola.py
#    1 passed in 0.25s
```

7.14. Inclusión de colas

7.14.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Utilizando el [tipo abstracto de datos de las colas](https://bit.ly/3QWTsRL),
-- definir la función
--    contenidaCola :: Eq a => Cola a -> Cola a -> Bool
-- tal que (contenidaCola c1 c2) se verifica si todos los elementos de
-- la cola c1 son elementos de la cola c2. Por ejemplo,
--    λ> ej1 = inserta 3 (inserta 2 vacia)
--    λ> ej2 = inserta 3 (inserta 4 vacia)
--    λ> ej3 = inserta 5 (inserta 2 (inserta 3 vacia))
--    λ> contenidaCola ej1 ej3
--    True
--    λ> contenidaCola ej2 ej3
--    False
-- ---------------------------------------------------------------------

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

module ContenidaCola where

import TAD.Cola (Cola, vacia, inserta, esVacia, primero, resto)
import PerteneceCola (perteneceCola)
import Transformaciones_colas_listas (colaAlista)
import Test.QuickCheck

-- 1ª solución
-- ===========

contenidaCola1 :: Eq a => Cola a -> Cola a -> Bool
contenidaCola1 c1 c2
  | esVacia c1 = True
  | otherwise  = perteneceCola (primero c1) c2 &&
                 contenidaCola1 (resto c1) c2

-- La función perteneceCola está definida en el ejercicio
-- "TAD de las colas: Pertenencia a una cola" que se encuentra en
-- https://bit.ly/3RcVgqb
```

```haskell
-- 2ª solución
-- ===========

contenidaCola2 :: Eq a => Cola a -> Cola a -> Bool
contenidaCola2 c1 c2 =
  contenidaLista (colaAlista c1) (colaAlista c2)

-- La función colaAlista está definida en el ejercicio
-- "TAD de las colas: Transformaciones entre colas y listas" que se
-- encuentra en https://bit.ly/3Xv0oIt

contenidaLista :: Eq a => [a] -> [a] -> Bool
contenidaLista xs ys =
  all (`elem` ys) xs

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

-- La propiedad es
prop_contenidaCola :: Cola Int -> Cola Int -> Bool
prop_contenidaCola c1 c2 =
  contenidaCola1 c1 c2 == contenidaCola2 c1 c2

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

7.14.2. En Python

```python
# ---------------------------------------------------------------------
# Utilizando el [tipo abstracto de datos de las colas](https://bit.ly/3QWTsRL),
# definir la función
#    contenidaCola : (Cola[A], Cola[A]) -> bool
# tal que contenidaCola(c1, c2) se verifica si todos los elementos de la
# cola c1 son elementos de la cola c2. Por ejemplo,
#    >>> ej1 = inserta(3, inserta(2, vacia()))
#    >>> ej2 = inserta(3, inserta(4, vacia()))
#    >>> ej3 = inserta(5, inserta(2, inserta(3, vacia())))
#    >>> contenidaCola(ej1, ej3)
#    True
```

```python
#    >>> contenidaCola(ej2, ej3)
#    False
# ---------------------------------------------------------------------

# pylint: disable=unused-import

from copy import deepcopy
from typing import TypeVar

from hypothesis import given

from src.perteneceCola import perteneceCola
from src.TAD.cola import (Cola, colaAleatoria, esVacia, inserta, primero,
                          resto, vacia)
from src.transformaciones_colas_listas import colaAlista

A = TypeVar('A')

# 1ª solución
# ===========

def contenidaCola1(c1: Cola[A], c2: Cola[A]) -> bool:
    if esVacia(c1):
        return True
    return perteneceCola(primero(c1), c2) and contenidaCola1(resto(c1), c2)

# La función perteneceCola está definida en el ejercicio
# "Pertenencia a una cola" que se encuentra en
# https://bit.ly/3RcVgqb

# 2ª solución
# ===========

def contenidaCola2(c1: Cola[A], c2: Cola[A]) -> bool:
    return set(colaAlista(c1)) <= set(colaAlista(c2))

# La función colaAlista está definida en el ejercicio
# "Transformaciones entre colas y listas" que se encuentra en
# https://bit.ly/3Xv0oIt
```

```
# 3ª solución
# ===========

def contenidaCola3Aux(c1: Cola[A], c2: Cola[A]) -> bool:
    if c1.esVacia():
        return True
    pc1 = c1.primero()
    c1.resto()
    return perteneceCola(pc1, c2) and contenidaCola1(c1, c2)

def contenidaCola3(c1: Cola[A], c2: Cola[A]) -> bool:
    _c1 = deepcopy(c1)
    return contenidaCola3Aux(_c1, c2)

# 4ª solución
# ===========

def contenidaCola4Aux(c1: Cola[A], c2: Cola[A]) -> bool:
    while not c1.esVacia():
        pc1 = c1.primero()
        c1.resto()
        if not perteneceCola(pc1, c2):
            return False
    return True

def contenidaCola4(c1: Cola[A], c2: Cola[A]) -> bool:
    _c1 = deepcopy(c1)
    return contenidaCola4Aux(_c1, c2)

# Comprobación de equivalencia de las definiciones
# ================================================

# La propiedad es
@given(c1=colaAleatoria(), c2=colaAleatoria())
def test_contenidaCola(c1: Cola[int], c2: Cola[int]) -> None:
    r = contenidaCola1(c1, c2)
    assert contenidaCola2(c1, c2) == r
    assert contenidaCola3(c1, c2) == r
    assert contenidaCola4(c1, c2) == r
```

```
# La comprobación es
#    src> poetry run pytest -q contenidaCola.py
#    1 passed in 0.44s
```

7.15. Reconocimiento de prefijos de colas

7.15.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Utilizando el [tipo abstracto de datos de las colas](https://bit.ly/3QWTsRL),
-- definir la función
--    prefijoCola :: Eq a => Cola a -> Cola a -> Bool
-- tal que (prefijoCola c1 c2) se verifica si la cola c1 es justamente
-- un prefijo de la cola c2. Por ejemplo,
--    λ> ej1 = inserta 4 (inserta 2 vacia)
--    λ> ej2 = inserta 5 (inserta 4 (inserta 2 vacia))
--    λ> ej3 = inserta 5 (inserta 2 (inserta 4 vacia))
--    λ> prefijoCola ej1 ej2
--    True
--    λ> prefijoCola ej1 ej3
--    False
-- ---------------------------------------------------------------------

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

module PrefijoCola where

import TAD.Cola (Cola, vacia, inserta, esVacia, primero, resto)
import Transformaciones_colas_listas (colaAlista)
import Data.List (isPrefixOf)
import Test.QuickCheck

-- 1ª solución
-- ===========

prefijoCola :: Eq a => Cola a -> Cola a -> Bool
prefijoCola c1 c2
  | esVacia c1 = True
  | esVacia c2 = False
  | otherwise  = primero c1 == primero c2 &&
```

```
                 prefijoCola (resto c1) (resto c2)

-- 2ª solución
-- ===========

prefijoCola2 :: Eq a => Cola a -> Cola a -> Bool
prefijoCola2 c1 c2 =
  colaAlista c1 `isPrefixOf` colaAlista c2

-- La función colaAlista está definida en el ejercicio
-- "Transformaciones entre colas y listas" que se encuentra en
-- https://bit.ly/3Xv0oIt

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

-- La propiedad es
prop_prefijoCola :: Cola Int -> Cola Int -> Bool
prop_prefijoCola c1 c2 =
  prefijoCola c1 c2 == prefijoCola2 c1 c2

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

7.15.2. En Python

```
# ---------------------------------------------------------------------
# Utilizando el [tipo abstracto de datos de las colas](https://bit.ly/3QWTsRL),
# definir la función
#    prefijoCola : (Cola[A], Cola[A]) -> bool
# tal que prefijoCola(c1, c2) se verifica si la cola c1 es justamente
# un prefijo de la cola c2. Por ejemplo,
#    >>> ej1 = inserta(4, inserta(2, vacia()))
#    >>> ej2 = inserta(5, inserta(4, inserta(2, vacia())))
#    >>> ej3 = inserta(5, inserta(2, inserta(4, vacia())))
#    >>> prefijoCola(ej1, ej2)
#    True
#    >>> prefijoCola(ej1, ej3)
#    False
```

```python
# ---------------------------------------------------------------------

# pylint: disable=unused-import

from copy import deepcopy
from typing import TypeVar

from hypothesis import given

from src.TAD.cola import (Cola, colaAleatoria, esVacia, inserta, primero,
                          resto, vacia)
from src.transformaciones_colas_listas import colaAlista

A = TypeVar('A')

# 1ª solución
# ===========

def prefijoCola(c1: Cola[A], c2: Cola[A]) -> bool:
    if esVacia(c1):
        return True
    if esVacia(c2):
        return False
    return primero(c1) == primero(c2) and prefijoCola(resto(c1), resto(c2))

# 2ª solución
# ===========

def esPrefijoLista(xs: list[A], ys: list[A]) -> bool:
    return ys[:len(xs)] == xs

def prefijoCola2(c1: Cola[A], c2: Cola[A]) -> bool:
    return esPrefijoLista(colaAlista(c1), colaAlista(c2))

# La función colaAlista está definida en el ejercicio
# ”Transformaciones entre colas y listas” que se encuentra en
# https://bit.ly/3Xv0oIt

# 3ª solución
# ===========
```

```python
def prefijoCola3Aux(c1: Cola[A], c2: Cola[A]) -> bool:
    if c1.esVacia():
        return True
    if c2.esVacia():
        return False
    cc1 = c1.primero()
    c1.resto()
    cc2 = c2.primero()
    c2.resto()
    return cc1 == cc2 and prefijoCola3(c1, c2)

def prefijoCola3(c1: Cola[A], c2: Cola[A]) -> bool:
    q1 = deepcopy(c1)
    q2 = deepcopy(c2)
    return prefijoCola3Aux(q1, q2)

# 4ª solución
# ===========

def prefijoCola4Aux(c1: Cola[A], c2: Cola[A]) -> bool:
    while not c2.esVacia() and not c1.esVacia():
        if c1.primero() != c2.primero():
            return False
        c1.resto()
        c2.resto()
    return c1.esVacia()

def prefijoCola4(c1: Cola[A], c2: Cola[A]) -> bool:
    q1 = deepcopy(c1)
    q2 = deepcopy(c2)
    return prefijoCola4Aux(q1, q2)

# Comprobación de equivalencia de las definiciones
# ================================================

# La propiedad es
@given(c1=colaAleatoria(), c2=colaAleatoria())
def test_prefijoCola(c1: Cola[int], c2: Cola[int]) -> None:
    r = prefijoCola(c1, c2)
```

```
    assert prefijoCola2(c1, c2) == r
    assert prefijoCola3(c1, c2) == r
    assert prefijoCola4(c1, c2) == r

# La comprobación es
#    src> poetry run pytest -q prefijoCola.py
#    1 passed in 0.29s
```

7.16. Reconocimiento de subcolas

7.16.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Utilizando el [tipo abstracto de datos de las colas](https://bit.ly/3QWTsRL),
-- definir la función
--    subCola :: Eq a => Cola a -> Cola a -> Bool
-- tal que (subCola c1 c2) se verifica si c1 es una subcola de c2. Por
-- ejemplo,
--    λ> ej1 = inserta 2 (inserta 3 vacia)
--    λ> ej2 = inserta 7 (inserta 2 (inserta 3 (inserta 5 vacia)))
--    λ> ej3 = inserta 2 (inserta 7 (inserta 3 (inserta 5 vacia)))
--    λ> subCola ej1 ej2
--    True
--    λ> subCola ej1 ej3
--    False
-- ---------------------------------------------------------------------

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

module SubCola where

import TAD.Cola (Cola, vacia, inserta, esVacia, primero, resto)
import Transformaciones_colas_listas (colaAlista)
import PrefijoCola (prefijoCola)
import Data.List (isPrefixOf, tails)
import Test.QuickCheck

-- 1ª solución
-- ===========
```

```haskell
subCola1 :: Eq a => Cola a -> Cola a -> Bool
subCola1 c1 c2
    | esVacia c1 = True
    | esVacia c2 = False
    | pc1 == pc2 = prefijoCola rc1 rc2 || subCola1 c1 rc2
    | otherwise  = subCola1 c1 rc2
    where pc1 = primero c1
          rc1 = resto c1
          pc2 = primero c2
          rc2 = resto c2

-- La función PrefijoCola está definida en el ejercicio
-- "Reconocimiento de prefijos de colas" que se encuentra en
-- https://bit.ly/3HaK20x

-- 2ª solución
-- ===========

subCola2 :: Eq a => Cola a -> Cola a -> Bool
subCola2 c1 c2 =
  sublista (colaAlista c1) (colaAlista c2)

-- La función colaAlista está definida en el ejercicio
-- "Transformaciones entre colas y listas" que se encuentra en
-- https://bit.ly/3Xv0oIt

-- (sublista xs ys) se verifica si xs es una sublista de ys. Por
-- ejemplo,
--    sublista [3,2] [5,3,2,7]  ==  True
--    sublista [3,2] [5,3,7,2]  ==  False
sublista :: Eq a => [a] -> [a] -> Bool
sublista xs ys =
  any (xs `isPrefixOf`) (tails ys)

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

-- La propiedad es
prop_subCola :: Cola Int -> Cola Int -> Bool
prop_subCola c1 c2 =
```

```
  subCola1 c1 c2 == subCola2 c1 c2

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

7.16.2. En Python

```python
# ---------------------------------------------------------------------
# Utilizando el [tipo abstracto de datos de las colas](https://bit.ly/3QWTsRL),
# definir la función
#    subCola : (Cola[A], Cola[A]) -> bool
# tal que subCola(c1, c2) se verifica si c1 es una subcola de c2. Por
# ejemplo,
#    >>> ej1 = inserta(2, inserta(3, vacia()))
#    >>> ej2 = inserta(7, inserta(2, inserta(3, inserta(5, vacia()))))
#    >>> ej3 = inserta(2, inserta(7, inserta(3, inserta(5, vacia()))))
#    >>> subCola(ej1, ej2)
#    True
#    >>> subCola(ej1, ej3)
#    False
# ---------------------------------------------------------------------

# pylint: disable=unused-import

from copy import deepcopy
from typing import TypeVar

from hypothesis import given

from src.prefijoCola import prefijoCola
from src.TAD.cola import (Cola, colaAleatoria, esVacia, inserta, primero,
                          resto, vacia)
from src.transformaciones_colas_listas import colaAlista

A = TypeVar('A')

# 1ª solución
# ===========
```

```
def subCola1(c1: Cola[A], c2: Cola[A]) -> bool:
    if esVacia(c1):
        return True
    if esVacia(c2):
        return False
    pc1 = primero(c1)
    rc1 = resto(c1)
    pc2 = primero(c2)
    rc2 = resto(c2)
    if pc1 == pc2:
        return prefijoCola(rc1, rc2) or subCola1(c1, rc2)
    return subCola1(c1, rc2)

# La función prefijoCola está definida en el ejercicio
# "Reconocimiento de prefijos de colas" que se encuentra en
# https://bit.ly/3HaK20x

# 2ª solución
# ===========

# sublista(xs, ys) se verifica si xs es una sublista de ys. Por
# ejemplo,
#    >>> sublista([3,2], [5,3,2,7])
#    True
#    >>> sublista([3,2], [5,3,7,2])
#    False
def sublista(xs: list[A], ys: list[A]) -> bool:
    return any(xs == ys[i:i+len(xs)] for i in range(len(ys) - len(xs) + 1))

def subCola2(c1: Cola[A], c2: Cola[A]) -> bool:
    return sublista(colaAlista(c1), colaAlista(c2))

# La función colaAlista está definida en el ejercicio
# "Transformaciones entre colas y listas" que se encuentra en
# https://bit.ly/3Xv0oIt

# 3ª solución
# ===========

def subCola3Aux(c1: Cola[A], c2: Cola[A]) -> bool:
```

```
    if c1.esVacia():
        return True
    if c2.esVacia():
        return False
    if c1.primero() != c2.primero():
        c2.resto()
        return subCola3Aux(c1, c2)
    q1 = deepcopy(c1)
    c1.resto()
    c2.resto()
    return prefijoCola(c1, c2) or subCola3Aux(q1, c2)

def subCola3(c1: Cola[A], c2: Cola[A]) -> bool:
    q1 = deepcopy(c1)
    q2 = deepcopy(c2)
    return subCola3Aux(q1, q2)

# Comprobación de equivalencia de las definiciones
# ================================================

# La propiedad es
@given(c1=colaAleatoria(), c2=colaAleatoria())
def test_subCola(c1: Cola[int], c2: Cola[int]) -> None:
    r = subCola1(c1, c2)
    assert subCola2(c1, c2) == r
    assert subCola3(c1, c2) == r

# La comprobación es
#    src> poetry run pytest -q subCola.py
#    1 passed in 0.31s
```

7.17. Reconocimiento de ordenación de colas

7.17.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Utilizando el [tipo abstracto de datos de las colas](https://bit.ly/3QWTsRL),
-- definir la función
--    ordenadaCola :: Ord a => Cola a -> Bool
-- tal que (ordenadaCola c) se verifica si los elementos de la cola c
```

```haskell
-- están ordenados en orden creciente. Por ejemplo,
--    ordenadaCola (inserta 6 (inserta 5 (inserta 1 vacia))) == True
--    ordenadaCola (inserta 1 (inserta 0 (inserta 6 vacia))) == False
-- ---------------------------------------------------------------------

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

module OrdenadaCola where

import TAD.Cola (Cola, vacia, inserta, esVacia, primero, resto)
import Transformaciones_colas_listas (colaAlista)
import Test.QuickCheck

-- 1ª solución
-- ===========

ordenadaCola :: Ord a => Cola a -> Bool
ordenadaCola c
  | esVacia c  = True
  | esVacia rc = True
  | otherwise  = pc <= prc && ordenadaCola rc
  where pc  = primero c
        rc  = resto c
        prc = primero rc

-- 2ª solución
-- ===========

ordenadaCola2 :: Ord a => Cola a -> Bool
ordenadaCola2 =
  ordenadaLista . colaAlista

-- (ordenadaLista xs) se verifica si la lista xs está ordenada de menor
-- a mayor. Por ejemplo,
ordenadaLista :: Ord a => [a] -> Bool
ordenadaLista xs =
  and [x <= y | (x,y) <- zip xs (tail xs)]

-- La función colaAlista está definida en el ejercicio
-- "Transformaciones entre colas y listas" que se encuentra en
```

```
-- https://bit.ly/3Xv0oIt

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

-- La propiedad es
prop_ordenadaCola :: Cola Int -> Bool
prop_ordenadaCola c =
  ordenadaCola c == ordenadaCola2 c

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

7.17.2. En Python

```
# ---------------------------------------------------------------------
# Utilizando el [tipo abstracto de datos de las colas](https://bit.ly/3QWTsRL),
# definir la función
#    ordenadaCola : (Cola[A]) -> bool
# tal que ordenadaCola(c) se verifica si los elementos de la cola c
# están ordenados en orden creciente. Por ejemplo,
#    >>> ordenadaCola(inserta(6, inserta(5, inserta(1, vacia()))))
#    True
#    >>> ordenadaCola(inserta(1, inserta(0, inserta(6, vacia()))))
#    False
# ---------------------------------------------------------------------

# pylint: disable=unused-import

from copy import deepcopy
from typing import TypeVar

from hypothesis import given

from src.TAD.cola import (Cola, colaAleatoria, esVacia, inserta, primero,
                          resto, vacia)
from src.transformaciones_colas_listas import colaAlista

A = TypeVar('A', int, float, str)
```

```
# 1ª solución
# ===========

def ordenadaCola(c: Cola[A]) -> bool:
    if esVacia(c):
        return True
    pc = primero(c)
    rc = resto(c)
    if esVacia(rc):
        return True
    prc = primero(rc)
    return pc <= prc and ordenadaCola(rc)

# 2ª solución
# ===========

# ordenadaLista(xs, ys) se verifica si xs es una lista ordenada. Por
# ejemplo,
#    >>> ordenadaLista([2, 5, 8])
#    True
#    >>> ordenadalista([2, 8, 5])
#    False
def ordenadaLista(xs: list[A]) -> bool:
    return all((x <= y for (x, y) in zip(xs, xs[1:])))

def ordenadaCola2(p: Cola[A]) -> bool:
    return ordenadaLista(colaAlista(p))

# La función colaAlista  está definida en el ejercicio
# ”Transformaciones entre colas y listas” que se encuentra en
# https://bit.ly/3Xv0oIt

# 3ª solución
# ===========

def ordenadaCola3Aux(c: Cola[A]) -> bool:
    if c.esVacia():
        return True
    pc = c.primero()
```

```
    c.resto()
    if c.esVacia():
        return True
    return pc <= c.primero() and ordenadaCola3Aux(c)

def ordenadaCola3(c: Cola[A]) -> bool:
    _c = deepcopy(c)
    return ordenadaCola3Aux(_c)

# 4ª solución
# ===========

def ordenadaCola4Aux(c: Cola[A]) -> bool:
    while not c.esVacia():
        pc = c.primero()
        c.resto()
        if not c.esVacia() and pc > c.primero():
            return False
    return True

def ordenadaCola4(c: Cola[A]) -> bool:
    _c = deepcopy(c)
    return ordenadaCola4Aux(_c)

# Comprobación de equivalencia de las definiciones
# ================================================

# La propiedad es
@given(p=colaAleatoria())
def test_ordenadaCola(p: Cola[int]) -> None:
    r = ordenadaCola(p)
    assert ordenadaCola2(p) == r
    assert ordenadaCola3(p) == r
    assert ordenadaCola4(p) == r

# La comprobación es
#    src> poetry run pytest -q ordenadaCola.py
#    1 passed in 0.27s
```

)

7.18. Máximo elemento de una cola

7.18.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Utilizando el [tipo abstracto de datos de las colas](https://bit.ly/3QWTsRL),
-- definir la función
--    maxCola :: Ord a => Cola a -> a
-- tal que (maxCola c) sea el mayor de los elementos de la cola c. Por
-- ejemplo,
--    λ> maxCola (inserta 3 (inserta 5 (inserta 1 vacia)))
--    5
-- ---------------------------------------------------------------------

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

module MaxCola where

import TAD.Cola (Cola, vacia, inserta, esVacia, primero, resto)
import Transformaciones_colas_listas (colaAlista)
import Test.QuickCheck

-- 1ª solución
-- ===========

maxCola1 :: Ord a => Cola a -> a
maxCola1 c
  | esVacia rc = pc
  | otherwise  = max pc (maxCola1 rc)
  where pc = primero c
        rc = resto c

-- 2ª solución
-- ===========

maxCola2 :: Ord a => Cola a -> a
maxCola2 =
  maximum . colaAlista

-- La función colaAlista está definida en el ejercicio
-- "Transformaciones entre colas y listas" que se encuentra en
```

```
-- https://bit.ly/3Xv0oIt

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

-- La propiedad es
prop_maxCola :: Cola Int -> Property
prop_maxCola c =
  not (esVacia c) ==> maxCola1 c == maxCola2 c

-- La comprobación es
--    λ> quickCheck prop_maxCola
--    +++ OK, passed 100 tests; 16 discarded.
```

7.18.2. En Python

```
# ---------------------------------------------------------------------
# Utilizando el [tipo abstracto de datos de las colas](https://bit.ly/3QWTsRL),
# definir la función
#    maxCola : (Cola[A]) -> A
# tal que maxCola(c) sea el mayor de los elementos de la cola c. Por
# ejemplo,
#    >>> maxCola(inserta(3, inserta(5, inserta(1, vacia()))))
#    5
# ---------------------------------------------------------------------

# pylint: disable=unused-import

from copy import deepcopy
from typing import TypeVar

from hypothesis import assume, given

from src.TAD.cola import (Cola, colaAleatoria, esVacia, inserta, primero,
                          resto, vacia)
from src.transformaciones_colas_listas import colaAlista

A = TypeVar('A', int, float, str)

# 1ª solución
```

```
# ===========

def maxCola1(c: Cola[A]) -> A:
    pc = primero(c)
    rc = resto(c)
    if esVacia(rc):
        return pc
    return max(pc, maxCola1(rc))

# 2ª solución
# ===========

# Se usará la función colaAlista del ejercicio
# ”Transformaciones entre colas y listas” que se encuentra en
# https://bit.ly/3ZHewQ8

def maxCola2(c: Cola[A]) -> A:
    return max(colaAlista(c))

# 3ª solución
# ===========

def maxCola3Aux(c: Cola[A]) -> A:
    pc = c.primero()
    c.resto()
    if esVacia(c):
        return pc
    return max(pc, maxCola3Aux(c))

def maxCola3(c: Cola[A]) -> A:
    _c = deepcopy(c)
    return maxCola3Aux(_c)

# 4ª solución
# ===========

def maxCola4Aux(c: Cola[A]) -> A:
    r = c.primero()
    while not esVacia(c):
        pc = c.primero()
```

```
        if pc > r:
            r = pc
        c.resto()
    return r

def maxCola4(c: Cola[A]) -> A:
    _c = deepcopy(c)
    return maxCola4Aux(_c)

# Comprobación de equivalencia de las definiciones
# ================================================

# La propiedad es
@given(c=colaAleatoria())
def test_maxCola(c: Cola[int]) -> None:
    assume(not esVacia(c))
    r = maxCola1(c)
    assert maxCola2(c) == r
    assert maxCola3(c) == r
    assert maxCola4(c) == r

# La comprobación es
#    src> poetry run pytest -q maxCola.py
#    1 passed in 0.30s
```

Capítulo 8

El tipo abstracto de datos de los conjuntos

Contenido

8.1. El tipo abstracto de datos de los conjuntos

8.1.1. En Haskell

```
-- Un conjunto es una estructura de datos, caracterizada por ser una
-- colección de elementos en la que no importe ni el orden ni la
-- repetición de elementos.
```

```
--
-- Las operaciones que definen al tipo abstracto de datos (TAD) de los
-- conjuntos (cuyos elementos son del tipo a) son las siguientes:
--    vacio      :: Conj a
--    inserta    :: Ord a => a -> Conj a -> Conj a
--    menor      :: Ord a => Conj a -> a
--    elimina    :: Ord a => a -> Conj a -> Conj a
--    pertenece  :: Ord a => a -> Conj a -> Bool
--    esVacio    :: Conj a -> Bool
-- tales que
--    + vacio es el conjunto vacío.
--    + (inserta x c) es el conjunto obtenido añadiendo el elemento x al
--      conjunto c.
--    + (menor c) es el menor elemento del conjunto c.
--    + (elimina x c) es el conjunto obtenido eliminando el elemento x
--      del conjunto c.
--    + (pertenece x c) se verifica si x pertenece al conjunto c.
--    + (esVacio c) se verifica si c es el conjunto vacío.
--
-- Las operaciones tienen que verificar las siguientes propiedades:
--    + inserta x (inserta x c) == inserta x c
--    + inserta x (inserta y c) == inserta y (inserta x c)
--    + not (pertenece x vacio)
--    + pertenece y (inserta x c) == (x==y) || pertenece y c
--    + elimina x vacio == vacio
--    + Si x == y, entonces
--      elimina x (inserta y c) == elimina x c
--    + Si x /= y, entonces
--      elimina x (inserta y c) == inserta y (elimina x c)
--    + esVacio vacio
--    + not (esVacio (inserta x c))
--
-- Para usar el TAD hay que usar una implementación concreta. En
-- principio, consideraremos las siguientes:
--    + mediante listas no ordenadas con duplicados,
--    + mediante listas no ordenadas sin duplicados,
--    + mediante listas ordenadas sin duplicados y
--    + mediante la librería Data.Set.
-- Hay que elegir la que se desee utilizar, descomentándola y comentando
-- las otras.
```

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

module TAD.Conjunto
  (Conj,
   vacio,     -- Conj a
   inserta,   -- Ord a => a -> Conj a -> Conj a
   menor,     -- Ord a => Conj a -> a
   elimina,   -- Ord a => a -> Conj a -> Conj a
   pertenece, -- Ord a => a -> Conj a -> Bool
   esVacio    -- Conj a -> Bool
  ) where

-- import TAD.ConjuntoConListasNoOrdenadasConDuplicados
-- import TAD.ConjuntoConListasNoOrdenadasSinDuplicados
import TAD.ConjuntoConListasOrdenadasSinDuplicados
-- import TAD.ConjuntoConLibreria
```

8.1.2. En Python

```python
# Un conjunto es una estructura de datos, caracterizada por ser una
# colección de elementos en la que no importe ni el orden ni la
# repetición de elementos.
#
# Las operaciones que definen al tipo abstracto de datos (TAD) de los
# conjuntos (cuyos elementos son del tipo a) son las siguientes:
#    vacio      :: Conj a
#    inserta    :: Ord a => a -> Conj a -> Conj a
#    menor      :: Ord a => Conj a -> a
#    elimina    :: Ord a => a -> Conj a -> Conj a
#    pertenece  :: Ord a => a -> Conj a -> Bool
#    esVacio    :: Conj a -> Bool
# tales que
#    + vacio es el conjunto vacío.
#    + (inserta x c) es el conjunto obtenido añadiendo el elemento x al
#      conjunto c.
#    + (menor c) es el menor elemento del conjunto c.
#    + (elimina x c) es el conjunto obtenido eliminando el elemento x
#      del conjunto c.
#    + (pertenece x c) se verifica si x pertenece al conjunto c.
```

```
#    + (esVacio c) se verifica si c es el conjunto vacío.
#
# Las operaciones tienen que verificar las siguientes propiedades:
#    + inserta x (inserta x c) == inserta x c
#    + inserta x (inserta y c) == inserta y (inserta x c)
#    + not (pertenece x vacio)
#    + pertenece y (inserta x c) == (x==y) || pertenece y c
#    + elimina x vacio == vacio
#    + Si x == y, entonces
#      elimina x (inserta y c) == elimina x c
#    + Si x /= y, entonces
#      elimina x (inserta y c) == inserta y (elimina x c)
#    + esVacio vacio
#    + not (esVacio (inserta x c))
#
# Para usar el TAD hay que usar una implementación concreta. En
# principio, consideraremos las siguientes:
#    + mediante listas no ordenadas con duplicados,
#    + mediante listas no ordenadas sin duplicados,
#    + mediante listas ordenadas sin duplicados y
#    + mediante la librería Data.Set.
# Hay que elegir la que se desee utilizar, descomentándola y comentando
# las otras.

__all__ = [
    'Conj',
    'vacio',
    'inserta',
    'menor',
    'elimina',
    'pertenece',
    'esVacio',
    'conjuntoAleatorio'
]

# from src.TAD.conjuntoConListasNoOrdenadasConDuplicados import (
#     Conj, conjuntoAleatorio, elimina, esVacio, inserta,
#     menor, pertenece, vacio)

# from src.TAD.conjuntoConListasNoOrdenadasSinDuplicados import (
```

```
#     Conj, conjuntoAleatorio, elimina, esVacio, inserta, menor, pertenece,
#     vacio)

from src.TAD.conjuntoConListasOrdenadasSinDuplicados import (Conj,
                                                              conjuntoAleatorio,
                                                              elimina, esVacio,
                                                              inserta, menor,
                                                              pertenece, vacio)

# from src.TAD.conjuntoConLibreria import (
#     Conj, conjuntoAleatorio, elimina, esVacio, inserta, menor, pertenece,
#     vacio)
```

8.2. El tipo de datos de los conjuntos mediante listas no ordenadas con duplicados

8.2.1. En Haskell

```haskell
{-# OPTIONS_GHC -fno-warn-unused-top-binds #-}

module TAD.ConjuntoConListasNoOrdenadasConDuplicados
  (Conj,
   vacio,     -- Conj a
   inserta,   -- Ord a => a -> Conj a -> Conj a
   menor,     -- Ord a => Conj a -> a
   elimina,   -- Ord a => a -> Conj a -> Conj a
   pertenece, -- Ord a => a -> Conj a -> Bool
   esVacio    -- Conj a -> Bool
  ) where

import Data.List (intercalate, nub, sort)
import Test.QuickCheck

-- Conjuntos como listas no ordenadas con repeticiones:
newtype Conj a = Cj [a]

-- (escribeConjunto c) es la cadena correspondiente al conjunto c. Por
-- ejemplo,
--    λ> escribeConjunto (Cj [])
```

```
--    "{}"
--    λ> escribeConjunto (Cj [5])
--    "{5}"
--    λ> escribeConjunto (Cj [2, 5])
--    "{2, 5}"
--    λ> escribeConjunto (Cj [5, 2, 5])
--    "{2, 5}"
escribeConjunto :: (Show a, Ord a) => Conj a -> String
escribeConjunto (Cj xs) =
  "{" ++ intercalate ", " (map show (sort (nub xs))) ++ "}"

-- Procedimiento de escritura de conjuntos.
instance (Show a, Ord a) => Show (Conj a) where
  show = escribeConjunto

-- Nota: Aunque el conjunto no está ordenado y tenga repeticiones, al
-- escribirlo se hará sin repeticiones y ordenando sus elementos.

-- vacio es el conjunto vacío. Por ejemplo,
--    λ> vacio
--    {}
vacio :: Conj a
vacio = Cj []

-- (inserta x c) es el conjunto obtenido añadiendo el elemento x al
-- conjunto c. Por ejemplo,
--    λ> inserta 5 vacio
--    {5}
--    λ> inserta 2 (inserta 5 vacio)
--    {2, 5}
--    λ> inserta 5 (inserta 2 vacio)
--    {2, 5}
inserta :: Eq a => a -> Conj a -> Conj a
inserta x (Cj ys) = Cj (x:ys)

-- (menor c) es el menor elemento del conjunto c. Por ejemplo,
--    λ> menor (inserta 5 (inserta 2 vacio))
--    2
menor :: Ord a => Conj a -> a
menor (Cj []) = error "conjunto vacío"
```

```
menor (Cj xs) = minimum xs

-- (elimina x c) es el conjunto obtenido eliminando el elemento x
-- del conjunto c. Por ejemplo,
--    λ> elimina 2 (inserta 5 (inserta 2 vacio))
--    {5}
elimina :: Eq a => a -> Conj a -> Conj a
elimina x (Cj ys) = Cj (filter (/= x) ys)

-- (esVacio c) se verifica si c es el conjunto vacío. Por ejemplo,
--    λ> esVacio (inserta 5 (inserta 2 vacio))
--    False
--    λ> esVacio vacio
--    True
esVacio :: Conj a -> Bool
esVacio (Cj xs) = null xs

-- (pertenece x c) se verifica si x pertenece al conjunto c. Por ejemplo,
--    λ> pertenece 2 (inserta 5 (inserta 2 vacio))
--    True
--    λ> pertenece 4 (inserta 5 (inserta 2 vacio))
--    False
pertenece :: Eq a => a -> Conj a -> Bool
pertenece x (Cj xs) = x `elem` xs

-- (subconjunto c1 c2) se verifica si c1 es un subconjunto de c2. Por
-- ejemplo,
--    subconjunto (Cj [1,3,2,1]) (Cj [3,1,3,2])  ==  True
--    subconjunto (Cj [1,3,4,1]) (Cj [3,1,3,2])  ==  False
subconjunto :: Ord a => Conj a -> Conj a -> Bool
subconjunto (Cj xs) (Cj ys) = sublista xs ys
  where sublista [] _      = True
        sublista (z:zs) vs = elem z vs && sublista zs vs

-- (igualConjunto c1 c2) se verifica si los conjuntos c1 y c2 son
-- iguales. Por ejemplo,
--    igualConjunto (Cj [1,3,2,1]) (Cj [3,1,3,2])  ==  True
--    igualConjunto (Cj [1,3,4,1]) (Cj [3,1,3,2])  ==  False
igualConjunto :: Ord a => Conj a -> Conj a -> Bool
igualConjunto c c' =
```

```
  subconjunto c c' && subconjunto c' c

--- Los conjuntos son comparables por igualdad.
instance Ord a => Eq (Conj a) where
  (==) = igualConjunto

-- Generador de conjuntos                                           --
-- ======================

-- genConjunto es un generador de conjuntos. Por ejemplo,
--    λ> sample (genConjunto :: Gen (Conj Int))
--    {}
--    {1}
--    {0, 2, 3}
--    {-6, 5}
--    {2, 5}
--    {-9, -6, 4, 8}
--    {0, 1}
--    {-13, -11, -5, -2, -1, 0, 4, 6, 7, 8, 9, 14}
--    {-7, -5, -2, -1, 1, 2, 10, 13, 15}
--    {-18, -17, -16, -10, -9, 0, 1, 3, 4, 13, 16}
--    {-20, -15, -7, -1, 2, 8, 10, 15, 20}
genConjunto :: (Arbitrary a, Ord a) => Gen (Conj a)
genConjunto = do
  xs <- listOf arbitrary
  return (foldr inserta vacio xs)

-- Los conjuntos son concreciones de los arbitrarios.
instance (Arbitrary a, Ord a) => Arbitrary (Conj a) where
  arbitrary = genConjunto

-- Propiedades de los conjuntos                                         --
-- ============================

prop_conjuntos :: Int -> Int -> Conj Int -> Bool
prop_conjuntos x y c =
  inserta x (inserta x c) == inserta x c &&
  inserta x (inserta y c) == inserta y (inserta x c) &&
  not (pertenece x vacio) &&
  pertenece y (inserta x c) == (x == y) || pertenece y c &&
```

```
  elimina x vacio == vacio &&
  elimina x (inserta y c) == (if x == y
                              then elimina x c
                              else inserta y (elimina x c)) &&
  esVacio (vacio :: Conj Int) &&
  not (esVacio (inserta x c))

-- Comprobación
--    λ> quickCheck prop_conjuntos
--    +++ OK, passed 100 tests.
```

8.2.2. En Python

```
# Se define la clase Conj con los siguientes métodos:
#    + inserta(x) añade x al conjunto.
#    + menor() es el menor elemento del conjunto.
#    + elimina(x) elimina las ocurrencias de x en el conjunto.
#    + pertenece(x) se verifica si x pertenece al conjunto.
#    + esVacia() se verifica si la cola es vacía.
# Por ejemplo,
#    >>> c = Conj()
#    >>> c
#    {}
#    >>> c.inserta(5)
#    >>> c.inserta(2)
#    >>> c.inserta(3)
#    >>> c.inserta(4)
#    >>> c.inserta(5)
#    >>> c
#    {2, 3, 4, 5}
#    >>> c.menor()
#    2
#    >>> c.elimina(3)
#    >>> c
#    {2, 4, 5}
#    >>> c.pertenece(4)
#    True
#    >>> c.pertenece(3)
#    False
#    >>> c.esVacio()
```

```
#    False
#    >>> c = Conj()
#    >>> c.esVacio()
#    True
#    >>> c = Conj()
#    >>> c.inserta(2)
#    >>> c.inserta(5)
#    >>> d = Conj()
#    >>> d.inserta(5)
#    >>> d.inserta(2)
#    >>> d.inserta(5)
#    >>> c == d
#    True
#
# Además se definen las correspondientes funciones. Por ejemplo,
#    >>> vacio()
#    {}
#    >>> inserta(5, inserta(3, inserta(2, inserta(5, vacio()))))
#    {2, 3, 5}
#    >>> menor(inserta(5, inserta(3, inserta(2, inserta(5, vacio())))))
#    2
#    >>> elimina(5, inserta(5, inserta(3, inserta(2, inserta(5, vacio())))))
#    {2, 3}
#    >>> pertenece(5, inserta(5, inserta(3, inserta(2, inserta(5, vacio())))))
#    True
#    >>> pertenece(1, inserta(5, inserta(3, inserta(2, inserta(5, vacio())))))
#    False
#    >>> esVacio(inserta(5, inserta(3, inserta(2, inserta(5, vacio())))))
#    False
#    >>> esVacio(vacio())
#    True
#    >>> inserta(5, inserta(2, vacio())) == inserta(2, inserta(5, (inserta(2, vac
#    True
#
# Finalmente, se define un generador aleatorio de conjuntos y se
# comprueba que los conjuntos cumplen las propiedades de su
# especificación.

from __future__ import annotations
```

```
__all__ = [
    'Conj',
    'vacio',
    'inserta',
    'menor',
    'elimina',
    'pertenece',
    'esVacio',
    'conjuntoAleatorio'
]

from abc import abstractmethod
from copy import deepcopy
from dataclasses import dataclass, field
from typing import Any, Generic, Protocol, TypeVar

from hypothesis import given
from hypothesis import strategies as st


class Comparable(Protocol):
    @abstractmethod
    def __lt__(self: A, otro: A) -> bool:
        pass

A = TypeVar('A', bound=Comparable)

# Clase de los conjuntos mediante listas no ordenadas con duplicados
# ==================================================================

@dataclass
class Conj(Generic[A]):
    _elementos: list[A] = field(default_factory=list)

    def __repr__(self) -> str:
        """
        Devuelve una cadena con los elementos del conjunto entre llaves
        y separados por ", ".
        """
        return '{' + ', '.join(str(x) for x in sorted(list(set(self._elementos))))
```

```
    def __eq__(self, c: Any) -> bool:
        """
        Se verifica si el conjunto es igual a c; es decir, tienen los
        mismos elementos sin importar el orden ni las repeticiones.
        """
        return sorted(list(set(self._elementos))) == sorted(list(set(c._elementos

    def inserta(self, x: A) -> None:
        """
        Añade el elemento x al conjunto.
        """
        self._elementos.append(x)

    def menor(self) -> A:
        """
        Devuelve el menor elemento del conjunto
        """
        return min(self._elementos)

    def elimina(self, x: A) -> None:
        """
        Elimina el elemento x del conjunto.
        """
        while x in self._elementos:
            self._elementos.remove(x)

    def esVacio(self) -> bool:
        """
        Se verifica si el conjunto está vacío.
        """
        return not self._elementos

    def pertenece(self, x: A) -> bool:
        """
        Se verifica si x pertenece al conjunto.
        """
        return x in self._elementos

# Funciones del tipo conjunto
```

```
# ===========================

def vacio() -> Conj[A]:
    """
    Crea y devuelve un conjunto vacío de tipo A.
    """
    c: Conj[A] = Conj()
    return c

def inserta(x: A, c: Conj[A]) -> Conj[A]:
    """
    Inserta un elemento x en el conjunto c y devuelve un nuevo comjunto
    con el elemento insertado.
    """
    _aux = deepcopy(c)
    _aux.inserta(x)
    return _aux

def menor(c: Conj[A]) -> A:
    """
    Devuelve el menor elemento del conjunto c.
    """
    return c.menor()

def elimina(x: A, c: Conj[A]) -> Conj[A]:
    """
    Elimina las ocurrencias de c en c y devuelve una copia del conjunto
    resultante.
    """
    _aux = deepcopy(c)
    _aux.elimina(x)
    return _aux

def pertenece(x: A, c: Conj[A]) -> bool:
    """
    Se verifica si x pertenece a c.
    """
    return c.pertenece(x)

def esVacio(c: Conj[A]) -> bool:
```

```
    """
    Se verifica si el conjunto está vacío.
    """
    return c.esVacio()

# Generador de conjuntos
# ======================

def conjuntoAleatorio() -> st.SearchStrategy[Conj[int]]:
    """
    Genera una estrategia de búsqueda para generar conjuntos de enteros
    de forma aleatoria.

    Utiliza la librería Hypothesis para generar una lista de enteros y
    luego se convierte en una instancia de la clase cola.
    """
    return st.lists(st.integers()).map(Conj)

# Comprobación de las propiedades de los conjuntos
# ================================================

# Las propiedades son
@given(c=conjuntoAleatorio(), x=st.integers(), y=st.integers())
def test_conjuntos(c: Conj[int], x: int, y: int) -> None:
    v: Conj[int] = vacio()
    assert inserta(x, inserta(x, c)) == inserta(x, c)
    assert inserta(x, inserta(y, c)) == inserta(y, inserta(x, c))
    assert not pertenece(x, v)
    assert pertenece(y, inserta(x, c)) == (x == y) or pertenece(y, c)
    assert elimina(x, v) == v

    def relacion(x: int, y: int, c: Conj[int]) -> Conj[int]:
        if x == y:
            return elimina(x, c)
        return inserta(y, elimina(x, c))

    assert elimina(x, inserta(y, c)) == relacion(x, y, c)
    assert esVacio(vacio())
    assert not esVacio(inserta(x, c))
```

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

8.3. El tipo de datos de los conjuntos mediante listas no ordenadas sin duplicados

8.3.1. En Haskell

```haskell
{-# OPTIONS_GHC -fno-warn-unused-top-binds #-}

module TAD.ConjuntoConListasNoOrdenadasSinDuplicados
  (Conj,
   vacio,      -- Conj a
   inserta,    -- Ord a => a -> Conj a -> Conj a
   menor,      -- Ord a => Conj a -> a
   elimina,    -- Ord a => a -> Conj a -> Conj a
   pertenece,  -- Ord a => a -> Conj a -> Bool
   esVacio     -- Conj a -> Bool
  ) where

import Data.List (intercalate, sort)
import Test.QuickCheck

-- Los conjuntos como listas no ordenadas sin repeticiones.
newtype Conj a = Cj [a]

-- (escribeConjunto c) es la cadena correspondiente al conjunto c. Por
-- ejemplo,
--    λ> escribeConjunto (Cj [])
--    "{}"
--    λ> escribeConjunto (Cj [5])
--    "{5}"
--    λ> escribeConjunto (Cj [2, 5])
--    "{2, 5}"
--    λ> escribeConjunto (Cj [5, 2])
--    "{2, 5}"
escribeConjunto :: (Show a, Ord a) => Conj a -> String
escribeConjunto (Cj xs) =
```

```
  "{" ++ intercalate ", " (map show (sort xs)) ++ "}"

-- Procedimiento de escritura de conjuntos.
instance (Show a, Ord a) => Show (Conj a) where
  show = escribeConjunto

-- vacio es el conjunto vacío. Por ejemplo,
--    λ> vacio
--    {}
vacio :: Conj a
vacio = Cj []

-- (inserta x c) es el conjunto obtenido añadiendo el elemento x al
-- conjunto c. Por ejemplo,
--    λ> inserta 5 vacio
--    {5}
--    λ> inserta 2 (inserta 5 vacio)
--    {2, 5}
--    λ> inserta 5 (inserta 2 vacio)
--    {2, 5}
inserta :: Eq a => a -> Conj a -> Conj a
inserta x s@(Cj xs) | pertenece x s = s
                    | otherwise     = Cj (x:xs)

-- (menor c) es el menor elemento del conjunto c. Por ejemplo,
--    λ> menor (inserta 5 (inserta 2 vacio))
--    2
menor :: Ord a => Conj a -> a
menor (Cj []) = error "conjunto vacío"
menor (Cj xs) = minimum xs

-- (elimina x c) es el conjunto obtenido eliminando el elemento x
-- del conjunto c. Por ejemplo,
--    λ> elimina 2 (inserta 5 (inserta 2 vacio))
--    {5}
elimina :: Eq a => a -> Conj a -> Conj a
elimina x (Cj s) = Cj [y | y <- s, y /= x]

-- (esVacio c) se verifica si c es el conjunto vacío. Por ejemplo,
--    λ> esVacio (inserta 5 (inserta 2 vacio))
```

```haskell
--    False
--    λ> esVacio vacio
--    True
esVacio :: Conj a -> Bool
esVacio (Cj xs) = null xs

-- (pertenece x c) se verifica si x pertenece al conjunto c. Por ejemplo,
--    λ> pertenece 2 (inserta 5 (inserta 2 vacio))
--    True
--    λ> pertenece 4 (inserta 5 (inserta 2 vacio))
--    False
pertenece :: Eq a => a -> Conj a -> Bool
pertenece x (Cj xs) = x `elem` xs

-- (subconjunto c1 c2) se verifica si c1 es un subconjunto de c2. Por
-- ejemplo,
--    subconjunto (Cj [1,3,2]) (Cj [3,1,2])    ==  True
--    subconjunto (Cj [1,3,4,1]) (Cj [1,3,2])  ==  False
subconjunto :: Ord a => Conj a -> Conj a -> Bool
subconjunto (Cj xs) (Cj ys) = sublista xs ys
  where sublista [] _      = True
        sublista (z:zs) vs = elem z vs && sublista zs vs

-- (igualConjunto c1 c2) se verifica si los conjuntos c1 y c2 son
-- iguales. Por ejemplo,
--    igualConjunto (Cj [3,2,1]) (Cj [1,3,2])  ==  True
--    igualConjunto (Cj [1,3,4]) (Cj [1,3,2])  ==  False
igualConjunto :: Ord a => Conj a -> Conj a -> Bool
igualConjunto c c' =
  subconjunto c c' && subconjunto c' c

--- Los conjuntos son comparables por igualdad.
instance Ord a => Eq (Conj a) where
  (==) = igualConjunto

-- Generador de conjuntos                                          --
-- ======================

-- genConjunto es un generador de conjuntos. Por ejemplo,
--    λ> sample (genConjunto :: Gen (Conj Int))
```

```
--    {}
--    {-1, 0}
--    {-4, 1, 2}
--    {-3, 0, 2, 3, 4}
--    {-7}
--    {-10, -7, -5, -2, -1, 2, 5, 8}
--    {5, 7, 8, 10}
--    {-9, -6, -3, 8}
--    {-8, -6, -5, -1, 7, 9, 14}
--    {-18, -15, -14, -13, -3, -2, 1, 2, 4, 8, 12, 17}
--    {-17, -16, -13, -12, -11, -9, -6, -3, 0, 1, 3, 5, 6, 7, 16, 18}
genConjunto :: (Arbitrary a, Ord a) => Gen (Conj a)
genConjunto = do
  xs <- listOf arbitrary
  return (foldr inserta vacio xs)

-- Los conjuntos son concreciones de los arbitrarios.
instance (Arbitrary a, Ord a) => Arbitrary (Conj a) where
  arbitrary = genConjunto

-- Propiedades de los conjuntos                                          --
-- ============================

prop_conjuntos :: Int -> Int -> Conj Int -> Bool
prop_conjuntos x y c =
  inserta x (inserta x c) == inserta x c &&
  inserta x (inserta y c) == inserta y (inserta x c) &&
  not (pertenece x vacio) &&
  pertenece y (inserta x c) == (x == y) || pertenece y c &&
  elimina x vacio == vacio &&
  elimina x (inserta y c) == (if x == y
                              then elimina x c
                              else inserta y (elimina x c)) &&
  esVacio (vacio :: Conj Int) &&
  not (esVacio (inserta x c))

-- Comprobación
--    λ> quickCheck prop_conjuntos
--    +++ OK, passed 100 tests.
```

8.3.2. En Python

```
# Se define la clase Conj con los siguientes métodos:
#    + inserta(x) añade x al conjunto.
#    + menor() es el menor elemento del conjunto.
#    + elimina(x) elimina las ocurrencias de x en el conjunto.
#    + pertenece(x) se verifica si x pertenece al conjunto.
#    + esVacia() se verifica si la cola es vacía.
# Por ejemplo,
#    >>> c = Conj()
#    >>> c
#    {}
#    >>> c.inserta(5)
#    >>> c.inserta(2)
#    >>> c.inserta(3)
#    >>> c.inserta(4)
#    >>> c.inserta(5)
#    >>> c
#    {2, 3, 4, 5}
#    >>> c.menor()
#    2
#    >>> c.elimina(3)
#    >>> c
#    {2, 4, 5}
#    >>> c.pertenece(4)
#    True
#    >>> c.pertenece(3)
#    False
#    >>> c.esVacio()
#    False
#    >>> c = Conj()
#    >>> c.esVacio()
#    True
#    >>> c = Conj()
#    >>> c.inserta(2)
#    >>> c.inserta(5)
#    >>> d = Conj()
#    >>> d.inserta(5)
#    >>> d.inserta(2)
#    >>> d.inserta(5)
#    >>> c == d
```

```
#    True
#
# Además se definen las correspondientes funciones. Por ejemplo,
#    >>> vacio()
#    {}
#    >>> inserta(5, inserta(3, inserta(2, inserta(5, vacio()))))
#    {2, 3, 5}
#    >>> menor(inserta(5, inserta(3, inserta(2, inserta(5, vacio())))))
#    2
#    >>> elimina(5, inserta(5, inserta(3, inserta(2, inserta(5, vacio())))))
#    {2, 3}
#    >>> pertenece(5, inserta(5, inserta(3, inserta(2, inserta(5, vacio())))))
#    True
#    >>> pertenece(1, inserta(5, inserta(3, inserta(2, inserta(5, vacio())))))
#    False
#    >>> esVacio(inserta(5, inserta(3, inserta(2, inserta(5, vacio())))))
#    False
#    >>> esVacio(vacio())
#    True
#    >>> inserta(5, inserta(2, vacio())) == inserta(2, inserta(5, (inserta(2, vac
#    True
#
# Finalmente, se define un generador aleatorio de conjuntos y se
# comprueba que los conjuntos cumplen las propiedades de su
# especificación.

from __future__ import annotations

__all__ = [
    'Conj',
    'vacio',
    'inserta',
    'menor',
    'elimina',
    'pertenece',
    'esVacio',
    'conjuntoAleatorio'
]

from abc import abstractmethod
```

```python
from copy import deepcopy
from dataclasses import dataclass, field
from typing import Any, Generic, Protocol, TypeVar

from hypothesis import given
from hypothesis import strategies as st


class Comparable(Protocol):
    @abstractmethod
    def __lt__(self: A, otro: A) -> bool:
        pass

A = TypeVar('A', bound=Comparable)

# Clase de los conjuntos mediante listas no ordenadas sin duplicados
# ==================================================================

@dataclass
class Conj(Generic[A]):
    _elementos: list[A] = field(default_factory=list)

    def __repr__(self) -> str:
        """
        Devuelve una cadena con los elementos del conjunto entre llaves
        y separados por ", ".
        """
        return '{' + ', '.join(str(x) for x in sorted(self._elementos)) + '}'

    def __eq__(self, c: Any) -> bool:
        """
        Se verifica si el conjunto es igual a c; es decir, tienen los
        mismos elementos sin importar el orden ni las repeticiones.
        """
        return sorted(self._elementos) == sorted(c._elementos)

    def inserta(self, x: A) -> None:
        """
        Añade el elemento x al conjunto.
        """
```

```
        if x not in self._elementos:
            self._elementos.append(x)

    def menor(self) -> A:
        """
        Devuelve el menor elemento del conjunto
        """
        return min(self._elementos)

    def elimina(self, x: A) -> None:
        """
        Elimina el elemento x del conjunto.
        """
        if x in self._elementos:
            self._elementos.remove(x)

    def esVacio(self) -> bool:
        """
        Se verifica si el conjunto está vacío.
        """
        return not self._elementos

    def pertenece(self, x: A) -> bool:
        """
        Se verifica si x pertenece al conjunto.
        """
        return x in self._elementos

# Funciones del tipo conjunto
# ===========================

def vacio() -> Conj[A]:
    """
    Crea y devuelve un conjunto vacío de tipo A.
    """
    c: Conj[A] = Conj()
    return c

def inserta(x: A, c: Conj[A]) -> Conj[A]:
    """
```

```
    Inserta un elemento x en el conjunto c y devuelve un nuevo comjunto
    con el elemento insertado.
    """
    _aux = deepcopy(c)
    _aux.inserta(x)
    return _aux

def menor(c: Conj[A]) -> A:
    """
    Devuelve el menor elemento del conjunto c.
    """
    return c.menor()

def elimina(x: A, c: Conj[A]) -> Conj[A]:
    """
    Elimina las ocurrencias de c en c y devuelve una copia del conjunto
    resultante.
    """
    _aux = deepcopy(c)
    _aux.elimina(x)
    return _aux

def pertenece(x: A, c: Conj[A]) -> bool:
    """
    Se verifica si x pertenece a c.
    """
    return c.pertenece(x)

def esVacio(c: Conj[A]) -> bool:
    """
    Se verifica si el conjunto está vacío.
    """
    return c.esVacio()

# Generador de conjuntos
# ======================

def sin_duplicados(xs: list[int]) -> list[int]:
    return list(set(xs))
```

```python
def conjuntoAleatorio() -> st.SearchStrategy[Conj[int]]:
    """
    Estrategia de búsqueda para generar conjuntos de enteros de forma
    aleatoria.
    """
    xs = st.lists(st.integers()).map(sin_duplicados)
    return xs.map(Conj)

# Comprobación de las propiedades de los conjuntos
# ================================================

# Las propiedades son
@given(c=conjuntoAleatorio(), x=st.integers(), y=st.integers())
def test_conjuntos(c: Conj[int], x: int, y: int) -> None:
    assert inserta(x, inserta(x, c)) == inserta(x, c)
    assert inserta(x, inserta(y, c)) == inserta(y, inserta(x, c))
    v: Conj[int] = vacio()
    assert not pertenece(x, v)
    assert pertenece(y, inserta(x, c)) == (x == y) or pertenece(y, c)
    assert elimina(x, v) == v

    def relacion(x: int, y: int, c: Conj[int]) -> Conj[int]:
        if x == y:
            return elimina(x, c)
        return inserta(y, elimina(x, c))

    assert elimina(x, inserta(y, c)) == relacion(x, y, c)
    assert esVacio(vacio())
    assert not esVacio(inserta(x, c))

# La comprobación es
#    > poetry run pytest -q conjuntoConListasNoOrdenadasSinDuplicados.py
#    1 passed in 0.26s
```

8.4. El tipo de datos de los conjuntos mediante listas ordenadas sin duplicados

8.4.1. En Haskell

```haskell
{-# OPTIONS_GHC -fno-warn-unused-top-binds #-}

module TAD.ConjuntoConListasOrdenadasSinDuplicados
  (Conj,
   vacio,      -- Conj a
   inserta,    -- Ord a => a -> Conj a -> Conj a
   menor,      -- Ord a => Conj a -> a
   elimina,    -- Ord a => a -> Conj a -> Conj a
   pertenece,  -- Ord a => a -> Conj a -> Bool
   esVacio     -- Conj a -> Bool
  ) where

import Data.List (intercalate)
import Test.QuickCheck

-- Los conjuntos como listas ordenadas sin repeticiones.
newtype Conj a = Cj [a]
  deriving Eq

-- (escribeConjunto c) es la cadena correspondiente al conjunto c. Por
-- ejemplo,
--    λ> escribeConjunto (Cj [])
--    "{}"
--    λ> escribeConjunto (Cj [5])
--    "{5}"
--    λ> escribeConjunto (Cj [2, 5])
--    "{2, 5}"
--    λ> escribeConjunto (Cj [5, 2])
--    "{2, 5}"
escribeConjunto :: Show a => Conj a -> String
escribeConjunto (Cj xs) =
  "{" ++ intercalate ", " (map show xs) ++ "}"

-- Procedimiento de escritura de conjuntos.
instance Show a => Show (Conj a) where
```

```
  show = escribeConjunto

-- vacio es el conjunto vacío. Por ejemplo,
--    λ> vacio
--    {}
vacio :: Conj a
vacio = Cj []

-- (inserta x c) es el conjunto obtenido añadiendo el elemento x al
-- conjunto c. Por ejemplo,
--    λ> inserta 5 vacio
--    {5}
--    λ> inserta 2 (inserta 5 vacio)
--    {2, 5}
--    λ> inserta 5 (inserta 2 vacio)
--    {2, 5}
inserta :: Ord a => a -> Conj a -> Conj a
inserta x (Cj s) = Cj (agrega x s)
  where agrega z []                         = [z]
        agrega z s'@(y:ys) | z > y      = y : agrega z ys
                           | z < y      = z : s'
                           | otherwise  = s'

-- (menor c) es el menor elemento del conjunto c. Por ejemplo,
--    λ> menor (inserta 5 (inserta 2 vacio))
--    2
menor :: Ord a => Conj a -> a
menor (Cj [])    = error "conjunto vacío"
menor (Cj (x:_)) = x

-- (elimina x c) es el conjunto obtenido eliminando el elemento x
-- del conjunto c. Por ejemplo,
--    λ> elimina 2 (inserta 5 (inserta 2 vacio))
--    {5}
elimina :: Ord a => a -> Conj a -> Conj a
elimina x (Cj s) = Cj (elimina' x s)
  where elimina' _ []                       = []
        elimina' z s'@(y:ys) | z > y      = y : elimina' z ys
                             | z < y      = s'
                             | otherwise = ys
```

```haskell
-- (esVacio c) se verifica si c es el conjunto vacío. Por ejemplo,
--    λ> esVacio (inserta 5 (inserta 2 vacio))
--    False
--    λ> esVacio vacio
--    True
esVacio :: Conj a -> Bool
esVacio (Cj xs) = null xs

-- (pertenece x c) se verifica si x pertenece al conjunto c. Por ejemplo,
--    λ> pertenece 2 (inserta 5 (inserta 2 vacio))
--    True
--    λ> pertenece 4 (inserta 5 (inserta 2 vacio))
--    False
pertenece :: Ord a => a -> Conj a -> Bool
pertenece x (Cj s) = x `elem` takeWhile (<= x) s

-- Generador de conjutos                                              --
-- ======================

-- genConjunto es un generador de conjuntos. Por ejemplo,
--    λ> sample (genConjunto :: Gen (Conj Int))
--    {}
--    {1}
--    {2}
--    {}
--    {-5, -1}
--    {-9, -8, 2, 3, 10}
--    {4}
--    {-13, -7, 1, 14}
--    {-12, -10, -9, -4, 1, 2, 5, 14, 16}
--    {-18, -15, -14, -13, -10, -7, -6, -4, -1, 1, 10, 11, 12, 16}
--    {-16, -9, -6, -5, -4, -2, 3, 6, 9, 13, 17}
genConjunto :: (Arbitrary a, Ord a) => Gen (Conj a)
genConjunto = do
  xs <- listOf arbitrary
  return (foldr inserta vacio xs)

-- Los conjuntos son concreciones de los arbitrarios.
instance (Arbitrary a, Ord a) => Arbitrary (Conj a) where
```

```haskell
  arbitrary = genConjunto

-- Propiedades de los conjuntos                                        --
-- ============================

prop_conjuntos :: Int -> Int -> Conj Int -> Bool
prop_conjuntos x y c =
  inserta x (inserta x c) == inserta x c &&
  inserta x (inserta y c) == inserta y (inserta x c) &&
  not (pertenece x vacio) &&
  pertenece y (inserta x c) == (x == y) || pertenece y c &&
  elimina x vacio == vacio &&
  elimina x (inserta y c) == (if x == y
                              then elimina x c
                              else inserta y (elimina x c)) &&
  esVacio (vacio :: Conj Int) &&
  not (esVacio (inserta x c))

-- Comprobación
--    λ> quickCheck prop_conjuntos
--    +++ OK, passed 100 tests.
```

8.4.2. En Python

```python
# Se define la clase Conj con los siguientes métodos:
#    + inserta(x) añade x al conjunto.
#    + menor() es el menor elemento del conjunto.
#    + elimina(x) elimina las ocurrencias de x en el conjunto.
#    + pertenece(x) se verifica si x pertenece al conjunto.
#    + esVacia() se verifica si la cola es vacía.
# Por ejemplo,
#    >>> c = Conj()
#    >>> c
#    {}
#    >>> c.inserta(5)
#    >>> c.inserta(2)
#    >>> c.inserta(3)
#    >>> c.inserta(4)
#    >>> c.inserta(5)
#    >>> c
```

```
#    {2, 3, 4, 5}
#    >>> c.menor()
#    2
#    >>> c.elimina(3)
#    >>> c
#    {2, 4, 5}
#    >>> c.pertenece(4)
#    True
#    >>> c.pertenece(3)
#    False
#    >>> c.esVacio()
#    False
#    >>> c = Conj()
#    >>> c.esVacio()
#    True
#    >>> c = Conj()
#    >>> c.inserta(2)
#    >>> c.inserta(5)
#    >>> d = Conj()
#    >>> d.inserta(5)
#    >>> d.inserta(2)
#    >>> d.inserta(5)
#    >>> c == d
#    True
#
# Además se definen las correspondientes funciones. Por ejemplo,
#    >>> vacio()
#    {}
#    >>> inserta(5, inserta(3, inserta(2, inserta(5, vacio()))))
#    {2, 3, 5}
#    >>> menor(inserta(5, inserta(3, inserta(2, inserta(5, vacio())))))
#    2
#    >>> elimina(5, inserta(5, inserta(3, inserta(2, inserta(5, vacio())))))
#    {2, 3}
#    >>> pertenece(5, inserta(5, inserta(3, inserta(2, inserta(5, vacio())))))
#    True
#    >>> pertenece(1, inserta(5, inserta(3, inserta(2, inserta(5, vacio())))))
#    False
#    >>> esVacio(inserta(5, inserta(3, inserta(2, inserta(5, vacio())))))
#    False
```

```
#    >>> esVacio(vacio())
#    True
#    >>> inserta(5, inserta(2, vacio())) == inserta(2, inserta(5, (inserta(2, vac
#    True
#
# Finalmente, se define un generador aleatorio de conjuntos y se
# comprueba que los conjuntos cumplen las propiedades de su
# especificación.

from __future__ import annotations

__all__ = [
    'Conj',
    'vacio',
    'inserta',
    'menor',
    'elimina',
    'pertenece',
    'esVacio',
    'conjuntoAleatorio'
]

from abc import abstractmethod
from bisect import bisect_left, insort_left
from copy import deepcopy
from dataclasses import dataclass, field
from itertools import takewhile
from typing import Generic, Protocol, TypeVar

from hypothesis import given
from hypothesis import strategies as st


class Comparable(Protocol):
    @abstractmethod
    def __lt__(self: A, otro: A) -> bool:
        pass

A = TypeVar('A', bound=Comparable)
```

```
# Clase de los conjuntos mediante listas ordenadas sin duplicados
# ================================================================

@dataclass
class Conj(Generic[A]):
    _elementos: list[A] = field(default_factory=list)

    def __repr__(self) -> str:
        """
        Devuelve una cadena con los elementos del conjunto entre llaves
        y separados por ", ".
        """
        return '{' + ', '.join(str(x) for x in self._elementos) + '}'

    def inserta(self, x: A) -> None:
        """
        Añade el elemento x al conjunto.
        """
        if x not in self._elementos:
            insort_left(self._elementos, x)

    def menor(self) -> A:
        """
        Devuelve el menor elemento del conjunto
        """
        return self._elementos[0]

    def elimina(self, x: A) -> None:
        """
        Elimina el elemento x del conjunto.
        """
        pos = bisect_left(self._elementos, x)
        if pos < len(self._elementos) and self._elementos[pos] == x:
            self._elementos.pop(pos)

    def esVacio(self) -> bool:
        """
        Se verifica si el conjunto está vacío.
        """
        return not self._elementos
```

```
    def pertenece(self, x: A) -> bool:
        """
        Se verifica si x pertenece al conjunto.
        """
        return x in takewhile(lambda y: y < x or y == x, self._elementos)

# Funciones del tipo conjunto
# ===========================

def vacio() -> Conj[A]:
    """
    Crea y devuelve un conjunto vacío de tipo A.
    """
    c: Conj[A] = Conj()
    return c

def inserta(x: A, c: Conj[A]) -> Conj[A]:
    """
    Inserta un elemento x en el conjunto c y devuelve un nuevo comjunto
    con el elemento insertado.
    """
    _aux = deepcopy(c)
    _aux.inserta(x)
    return _aux

def menor(c: Conj[A]) -> A:
    """
    Devuelve el menor elemento del conjunto c.
    """
    return c.menor()

def elimina(x: A, c: Conj[A]) -> Conj[A]:
    """
    Elimina las ocurrencias de c en c y devuelve una copia del conjunto
    resultante.
    """
    _aux = deepcopy(c)
    _aux.elimina(x)
    return _aux
```

```python
def pertenece(x: A, c: Conj[A]) -> bool:
    """
    Se verifica si x pertenece a c.
    """
    return c.pertenece(x)

def esVacio(c: Conj[A]) -> bool:
    """
    Se verifica si el conjunto está vacío.
    """
    return c.esVacio()

# Generador de conjuntos
# ======================

def sin_duplicados_y_ordenado(xs: list[int]) -> list[int]:
    xs = list(set(xs))
    xs.sort()
    return xs

def conjuntoAleatorio() -> st.SearchStrategy[Conj[int]]:
    """
    Estrategia de búsqueda para generar conjuntos de enteros de forma
    aleatoria.
    """
    xs = st.lists(st.integers()).map(sin_duplicados_y_ordenado)
    return xs.map(Conj)

# Comprobación de las propiedades de los conjuntos
# ================================================

# Las propiedades son
@given(c=conjuntoAleatorio(), x=st.integers(), y=st.integers())
def test_conjuntos(c: Conj[int], x: int, y: int) -> None:
    assert inserta(x, inserta(x, c)) == inserta(x, c)
    assert inserta(x, inserta(y, c)) == inserta(y, inserta(x, c))
    v: Conj[int] = vacio()
    assert not pertenece(x, v)
    assert pertenece(y, inserta(x, c)) == (x == y) or pertenece(y, c)
```

```
    assert elimina(x, v) == v

    def relacion(x: int, y: int, c: Conj[int]) -> Conj[int]:
        if x == y:
            return elimina(x, c)
        return inserta(y, elimina(x, c))

    assert elimina(x, inserta(y, c)) == relacion(x, y, c)
    assert esVacio(vacio())
    assert not esVacio(inserta(x, c))

# La comprobación es
#    > poetry run pytest -q conjuntoConListasOrdenadasSinDuplicados.py
#    1 passed in 0.13s
```

8.5. El tipo de datos de los conjuntos mediante librería

8.5.1. En Haskell

```
{-# OPTIONS_GHC -fno-warn-unused-top-binds #-}
{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module TAD.ConjuntoConLibreria
  (Conj,
   vacio,     -- Conj a
   inserta,   -- Ord a => a -> Conj a -> Conj a
   menor,     -- Ord a => Conj a -> a
   elimina,   -- Ord a => a -> Conj a -> Conj a
   pertenece, -- Ord a => a -> Conj a -> Bool
   esVacio    -- Conj a -> Bool
  ) where

import Data.Set as S (Set, empty, insert, findMin, delete, member, null,
                      fromList, toList)
import Data.List (intercalate)
import Test.QuickCheck

-- Los conjuntos como conjuntos de la librería Data.Set
```

```
newtype Conj a = Cj (Set a)
  deriving (Eq, Ord)

-- (escribeConjunto c) es la cadena correspondiente al conjunto c. Por
-- ejemplo,
--    λ> escribeConjunto (Cj (fromList []))
--    "{}"
--    λ> escribeConjunto (Cj (fromList [5]))
--    "{5}"
--    λ> escribeConjunto (Cj (fromList [2, 5]))
--    "{2, 5}"
--    λ> escribeConjunto (Cj (fromList [5, 2]))
--    "{2, 5}"
escribeConjunto :: Show a => Conj a -> String
escribeConjunto (Cj s) =
  "{" ++ intercalate ", " (map show (toList s)) ++ "}"

-- Procedimiento de escritura de conjuntos.
instance Show a => Show (Conj a) where
  show = escribeConjunto

-- vacio es el conjunto vacío. Por ejemplo,
--    λ> vacio
--    {}
vacio :: Conj a
vacio = Cj empty

-- (inserta x c) es el conjunto obtenido añadiendo el elemento x al
-- conjunto c. Por ejemplo,
--    λ> inserta 5 vacio
--    {5}
--    λ> inserta 2 (inserta 5 vacio)
--    {2, 5}
--    λ> inserta 5 (inserta 2 vacio)
--    {2, 5}
inserta :: Ord a => a -> Conj a -> Conj a
inserta x (Cj s) = Cj (insert x s)

-- (menor c) es el menor elemento del conjunto c. Por ejemplo,
--    λ> menor (inserta 5 (inserta 2 vacio))
```

```
--    2
menor :: Ord a => Conj a -> a
menor (Cj s) = findMin s

-- (elimina x c) es el conjunto obtenido eliminando el elemento x
-- del conjunto c. Por ejemplo,
--    λ> elimina 2 (inserta 5 (inserta 2 vacio))
--    {5}
elimina :: Ord a => a -> Conj a -> Conj a
elimina x (Cj s) = Cj (delete x s)

-- (esVacio c) se verifica si c es el conjunto vacío. Por ejemplo,
--    λ> esVacio (inserta 5 (inserta 2 vacio))
--    False
--    λ> esVacio vacio
--    True
esVacio :: Conj a -> Bool
esVacio (Cj s) = S.null s

-- (pertenece x c) se verifica si x pertenece al conjunto c. Por ejemplo,
--    λ> pertenece 2 (inserta 5 (inserta 2 vacio))
--    True
--    λ> pertenece 4 (inserta 5 (inserta 2 vacio))
--    False
pertenece :: Ord a => a -> Conj a -> Bool
pertenece x (Cj s) = member x s

-- Generador de conjuntos                                          --
-- ======================

-- genConjunto es un generador de conjuntos. Por ejemplo,
--    λ> sample (genConjunto :: Gen (Conj Int))
--    {}
--    {-2, 0}
--    {-1, 3}
--    {-3, 2}
--    {-5, -4, -3, 2, 4, 6, 7}
--    {-4, 4}
--    {-9, -6, -3, 1, 5, 11, 12}
--    {-10, -8, -7, -3, 1, 2, 8, 9, 10, 13}
```

```haskell
--    {-13, -8, -7, -6, -1, 0, 1, 6, 7, 9, 11, 14, 16}
--    {-15, -12, -9, 1, 2, 9, 13, 15, 16, 18}
--    {-16}
genConjunto :: (Arbitrary a, Ord a) => Gen (Conj a)
genConjunto = do
  xs <- listOf arbitrary
  return (Cj (fromList xs))

-- Los conjuntos son concreciones de los arbitrarios.
instance (Arbitrary a, Ord a) => Arbitrary (Conj a) where
  arbitrary = genConjunto

-- Propiedades de los conjuntos                                          --
-- ============================

prop_conjuntos :: Int -> Int -> Conj Int -> Bool
prop_conjuntos x y c =
  inserta x (inserta x c) == inserta x c &&
  inserta x (inserta y c) == inserta y (inserta x c) &&
  not (pertenece x vacio) &&
  pertenece y (inserta x c) == (x == y) || pertenece y c &&
  elimina x vacio == vacio &&
  elimina x (inserta y c) == (if x == y
                               then elimina x c
                               else inserta y (elimina x c)) &&
  esVacio (vacio :: Conj Int) &&
  not (esVacio (inserta x c))

-- Comprobación
--    λ> quickCheck prop_conjuntos
--    +++ OK, passed 100 tests.
```

8.5.2. En Python

```python
# Se define la clase Conj con los siguientes métodos:
#    + inserta(x) añade x al conjunto.
#    + menor() es el menor elemento del conjunto.
#    + elimina(x) elimina las ocurrencias de x en el conjunto.
#    + pertenece(x) se verifica si x pertenece al conjunto.
#    + esVacia() se verifica si la cola es vacía.
```

```
# Por ejemplo,
#    >>> c = Conj()
#    >>> c
#    {}
#    >>> c.inserta(5)
#    >>> c.inserta(2)
#    >>> c.inserta(3)
#    >>> c.inserta(4)
#    >>> c.inserta(5)
#    >>> c
#    {2, 3, 4, 5}
#    >>> c.menor()
#    2
#    >>> c.elimina(3)
#    >>> c
#    {2, 4, 5}
#    >>> c.pertenece(4)
#    True
#    >>> c.pertenece(3)
#    False
#    >>> c.esVacio()
#    False
#    >>> c = Conj()
#    >>> c.esVacio()
#    True
#    >>> c = Conj()
#    >>> c.inserta(2)
#    >>> c.inserta(5)
#    >>> d = Conj()
#    >>> d.inserta(5)
#    >>> d.inserta(2)
#    >>> d.inserta(5)
#    >>> c == d
#    True
#
# Además se definen las correspondientes funciones. Por ejemplo,
#    >>> vacio()
#    {}
#    >>> inserta(5, inserta(3, inserta(2, inserta(5, vacio()))))
#    {2, 3, 5}
```

```
#    >>> menor(inserta(5, inserta(3, inserta(2, inserta(5, vacio())))))
#    2
#    >>> elimina(5, inserta(5, inserta(3, inserta(2, inserta(5, vacio())))))
#    {2, 3}
#    >>> pertenece(5, inserta(5, inserta(3, inserta(2, inserta(5, vacio())))))
#    True
#    >>> pertenece(1, inserta(5, inserta(3, inserta(2, inserta(5, vacio())))))
#    False
#    >>> esVacio(inserta(5, inserta(3, inserta(2, inserta(5, vacio())))))
#    False
#    >>> esVacio(vacio())
#    True
#    >>> inserta(5, inserta(2, vacio())) == inserta(2, inserta(5, (inserta(2, vac
#    True
#
# Finalmente, se define un generador aleatorio de conjuntos y se
# comprueba que los conjuntos cumplen las propiedades de su
# especificación.

from __future__ import annotations

__all__ = [
    'Conj',
    'vacio',
    'inserta',
    'menor',
    'elimina',
    'pertenece',
    'esVacio',
    'conjuntoAleatorio'
]

from abc import abstractmethod
from copy import deepcopy
from dataclasses import dataclass, field
from typing import Generic, Protocol, TypeVar

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

```
class Comparable(Protocol):
    @abstractmethod
    def __lt__(self: A, otro: A) -> bool:
        pass

A = TypeVar('A', bound=Comparable)

# Clase de los conjuntos mediante librería
# ========================================

@dataclass
class Conj(Generic[A]):
    _elementos: set[A] = field(default_factory=set)

    def __repr__(self) -> str:
        xs = [str(x) for x in self._elementos]
        return "{" + ", ".join(xs) + "}"

    def inserta(self, x: A) -> None:
        """
        Añade el elemento x al conjunto.
        """
        self._elementos.add(x)

    def menor(self) -> A:
        """
        Devuelve el menor elemento del conjunto
        """
        return min(self._elementos)

    def elimina(self, x: A) -> None:
        """
        Elimina el elemento x del conjunto.
        """
        self._elementos.discard(x)

    def esVacio(self) -> bool:
        """
        Se verifica si el conjunto está vacío.
```

```
        """
        return not self._elementos

    def pertenece(self, x: A) -> bool:
        """
        Se verifica si x pertenece al conjunto.
        """
        return x in self._elementos

# Funciones del tipo conjunto
# ===========================

def vacio() -> Conj[A]:
    """
    Crea y devuelve un conjunto vacío de tipo A.
    """
    c: Conj[A] = Conj()
    return c

def inserta(x: A, c: Conj[A]) -> Conj[A]:
    """
    Inserta un elemento x en el conjunto c y devuelve un nuevo comjunto
    con el elemento insertado.
    """
    _aux = deepcopy(c)
    _aux.inserta(x)
    return _aux

def menor(c: Conj[A]) -> A:
    """
    Devuelve el menor elemento del conjunto c.
    """
    return c.menor()

def elimina(x: A, c: Conj[A]) -> Conj[A]:
    """
    Elimina las ocurrencias de c en c y devuelve una copia del conjunto
    resultante.
    """
    _aux = deepcopy(c)
```

```
    _aux.elimina(x)
    return _aux

def pertenece(x: A, c: Conj[A]) -> bool:
    """
    Se verifica si x pertenece a c.
    """
    return c.pertenece(x)

def esVacio(c: Conj[A]) -> bool:
    """
    Se verifica si el conjunto está vacío.
    """
    return c.esVacio()

# Generador de conjuntos
# ======================

def conjuntoAleatorio() -> st.SearchStrategy[Conj[int]]:
    """
    Estrategia de búsqueda para generar conjuntos de enteros de forma
    aleatoria.
    """
    return st.builds(Conj, st.lists(st.integers()).map(set))

# Comprobación de las propiedades de los conjuntos
# ================================================

# Las propiedades son
@given(c=conjuntoAleatorio(), x=st.integers(), y=st.integers())
def test_conjuntos(c: Conj[int], x: int, y: int) -> None:
    assert inserta(x, inserta(x, c)) == inserta(x, c)
    assert inserta(x, inserta(y, c)) == inserta(y, inserta(x, c))
    v: Conj[int] = vacio()
    assert not pertenece(x, v)
    assert pertenece(y, inserta(x, c)) == (x == y) or pertenece(y, c)
    assert elimina(x, v) == v

    def relacion(x: int, y: int, c: Conj[int]) -> Conj[int]:
        if x == y:
```

```
            return elimina(x, c)
        return inserta(y, elimina(x, c))

    assert elimina(x, inserta(y, c)) == relacion(x, y, c)
    assert esVacio(vacio())
    assert not esVacio(inserta(x, c))

# La comprobación es
#    > poetry run pytest -q conjuntoConLibreria.py
#    1 passed in 0.22s
```

8.6. Transformaciones entre conjuntos y listas

8.6.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Utilizando el [tipo abstracto de datos de los conjuntos](https://bit.ly/3HbB7f
-- definir las funciones
--    listaAconjunto :: [a] -> Conj a
--    conjuntoAlista :: Conj a -> [a]
-- tales que
-- + (listaAconjunto xs) es el conjunto formado por los elementos de xs.
--   Por ejemplo,
--      λ> listaAconjunto [3, 2, 5]
--      {2, 3, 5}
-- + (conjuntoAlista c) es la lista formada por los elementos del
--   conjunto c. Por ejemplo,
--      λ> conjuntoAlista (inserta 5 (inserta 2 (inserta 3 vacio)))
--      [2,3,5]
--
-- Comprobar con QuickCheck que ambas funciones son inversa; es decir,
--    conjuntoAlista (listaAconjunto xs) = sort (nub xs)
--    listaAconjunto (conjuntoAlista c)  = c
-- ---------------------------------------------------------------------

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

module TAD_Transformaciones_conjuntos_listas where

import TAD.Conjunto (Conj, vacio, inserta, menor, elimina, pertenece, esVacio)
```

```haskell
import Data.List (sort, nub)
import Test.QuickCheck

-- 1ª definición de listaAconjunto
-- ===============================

listaAconjunto :: Ord a => [a] -> Conj a
listaAconjunto []     = vacio
listaAconjunto (x:xs) = inserta x (listaAconjunto xs)

-- 2ª definición de listaAconjunto
-- ===============================

listaAconjunto2 :: Ord a => [a] -> Conj a
listaAconjunto2 = foldr inserta vacio

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

-- La propiedad es
prop_listaAconjunto :: [Int] -> Bool
prop_listaAconjunto xs =
  listaAconjunto xs == listaAconjunto2 xs

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

-- Definición de conjuntoAlista
-- ============================

conjuntoAlista :: Ord a => Conj a -> [a]
conjuntoAlista c
  | esVacio c = []
  | otherwise = mc : conjuntoAlista rc
  where mc = menor c
        rc = elimina mc c

-- Comprobación de las propiedades
-- ===============================
```

```haskell
-- La primera propiedad es
prop_1_listaAconjunto :: [Int] -> Bool
prop_1_listaAconjunto xs =
  conjuntoAlista (listaAconjunto xs) == sort (nub xs)

-- La comprobación es
--    λ> quickCheck prop_1_listaAconjunto
--    +++ OK, passed 100 tests.

-- La segunda propiedad es
prop_2_listaAconjunto :: Conj Int -> Bool
prop_2_listaAconjunto c =
  listaAconjunto (conjuntoAlista c) == c

-- La comprobación es
--    λ> quickCheck prop_2_listaAconjunto
--    +++ OK, passed 100 tests.
```

8.6.2. En Python

```python
# ---------------------------------------------------------------------
# Utilizando el [tipo abstracto de datos de los conjuntos](https://bit.ly/3HbB7fo
# definir las funciones
#    listaAconjunto : (list[A]) -> Conj[A]
#    conjuntoAlista : (Conj[A]) -> list[A]
# tales que
# + listaAconjunto(xs) es el conjunto formado por los elementos de xs.
#   Por ejemplo,
#      >>> listaAconjunto([3, 2, 5])
#      {2, 3, 5}
# + conjuntoAlista(c) es la lista formada por los elementos del
#   conjunto c. Por ejemplo,
#      >>> conjuntoAlista(inserta(5, inserta(2, inserta(3, vacio()))))
#      [2, 3, 5]
#
# Comprobar con Hypothesis que ambas funciones son inversa; es decir,
#    conjuntoAlista (listaAconjunto xs) = sorted(list(set(xs)))
#    listaAconjunto (conjuntoAlista c)  = c
# ---------------------------------------------------------------------
```

```python
from __future__ import annotations

from abc import abstractmethod
from copy import deepcopy
from functools import reduce
from typing import Protocol, TypeVar

from hypothesis import given
from hypothesis import strategies as st

from src.TAD.conjunto import (Conj, conjuntoAleatorio, elimina, esVacio,
                              inserta, menor, vacio)


class Comparable(Protocol):
    @abstractmethod
    def __lt__(self: A, otro: A) -> bool:
        pass

A = TypeVar('A', bound=Comparable)

# 1ª definición de listaAconjunto
# ===============================

def listaAconjunto(xs: list[A]) -> Conj[A]:
    if not xs:
        return vacio()
    return inserta(xs[0], listaAconjunto(xs[1:]))

# 2ª definición de listaAconjunto
# ===============================

def listaAconjunto2(xs: list[A]) -> Conj[A]:
    return reduce(lambda ys, y: inserta(y, ys), xs, vacio())

# 3ª solución de listaAconjunto
# =============================

def listaAconjunto3(xs: list[A]) -> Conj[A]:
```

```
    c: Conj[A] = Conj()
    for x in xs:
        c.inserta(x)
    return c

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

# La propiedad es
@given(st.lists(st.integers()))
def test_listaAconjunto(xs: list[int]) -> None:
    r = listaAconjunto(xs)
    assert listaAconjunto2(xs) == r
    assert listaAconjunto3(xs) == r

# 1ª definición de conjuntoAlista
# ===============================

def conjuntoAlista(c: Conj[A]) -> list[A]:
    if esVacio(c):
        return []
    mc = menor(c)
    rc = elimina(mc, c)
    return [mc] + conjuntoAlista(rc)

# 2ª definición de conjuntoAlista
# ===============================

def conjuntoAlista2Aux(c: Conj[A]) -> list[A]:
    if c.esVacio():
        return []
    mc = c.menor()
    c.elimina(mc)
    return [mc] + conjuntoAlista2Aux(c)

def conjuntoAlista2(c: Conj[A]) -> list[A]:
    c1 = deepcopy(c)
    return conjuntoAlista2Aux(c1)

# 3ª definición de conjuntoAlista
```

```
# ================================

def conjuntoAlista3Aux(c: Conj[A]) -> list[A]:
    r = []
    while not c.esVacio():
        mc = c.menor()
        r.append(mc)
        c.elimina(mc)
    return r

def conjuntoAlista3(c: Conj[A]) -> list[A]:
    c1 = deepcopy(c)
    return conjuntoAlista3Aux(c1)

# Comprobación de equivalencia de las definiciones de conjuntoAlista
# ===============================================================

@given(c=conjuntoAleatorio())
def test_conjuntoAlista(c: Conj[int]) -> None:
    r = conjuntoAlista(c)
    assert conjuntoAlista2(c) == r
    assert conjuntoAlista3(c) == r

# Comprobación de las propiedades
# ================================

# La primera propiedad es
@given(st.lists(st.integers()))
def test_1_listaAconjunto(xs: list[int]) -> None:
    assert conjuntoAlista(listaAconjunto(xs)) == sorted(list(set(xs)))

# La segunda propiedad es
@given(c=conjuntoAleatorio())
def test_2_listaAconjunto(c: Conj[int]) -> None:
    assert listaAconjunto(conjuntoAlista(c)) == c

# La comprobación de las propiedades es
#    > poetry run pytest -v TAD_Transformaciones_conjuntos_listas.py
#       test_listaAconjunto PASSED
#       test_conjuntoAlista PASSED
```

```
#       test_1_listaAconjunto PASSED
#       test_2_listaAconjunto PASSED
```

8.7. Reconocimiento de subconjunto

8.7.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Utilizando el [tipo abstracto de datos de los conjuntos](https://bit.ly/3HbB7f
-- definir la función
--    subconjunto :: Ord a => Conj a -> Conj a -> Bool
-- tal que (subconjunto c1 c2) se verifica si todos los elementos de c1
-- pertenecen a c2. Por ejemplo,
--    λ> ej1 = inserta 5 (inserta 2 vacio)
--    λ> ej2 = inserta 3 (inserta 2 (inserta 5 vacio))
--    λ> ej3 = inserta 3 (inserta 4 (inserta 5 vacio))
--    λ> subconjunto ej1 ej2
--    True
--    λ> subconjunto ej1 ej3
--    False
-- ---------------------------------------------------------------------

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

module TAD_subconjunto where

import TAD.Conjunto (Conj, vacio, inserta, menor, elimina, pertenece, esVacio)
import TAD_Transformaciones_conjuntos_listas (conjuntoAlista)
import Test.QuickCheck

-- 1ª solución
-- ===========

subconjunto :: Ord a => Conj a -> Conj a -> Bool
subconjunto c1 c2
  | esVacio c1 = True
  | otherwise  =  pertenece mc1 c2 && subconjunto rc1 c2
  where mc1 = menor c1
        rc1 = elimina mc1 c1
```

```haskell
-- 2ª solución
-- ===========

subconjunto2 :: Ord a => Conj a -> Conj a -> Bool
subconjunto2 c1 c2 =
  and [pertenece x c2 | x <- conjuntoAlista c1]

-- La función conjuntoAlista está definida en el ejercicio
-- "Transformaciones entre conjuntos y listas" que se encuentra en
-- https://bit.ly/3RexzxH

-- 3ª solución
-- ===========

subconjunto3 :: Ord a => Conj a -> Conj a -> Bool
subconjunto3 c1 c2 =
  all (`pertenece` c2) (conjuntoAlista c1)

-- 4ª solución
-- ===========

subconjunto4 :: Ord a => Conj a -> Conj a -> Bool
subconjunto4 c1 c2 =
  sublista (conjuntoAlista c1) (conjuntoAlista c2)

-- (sublista xs ys) se verifica si xs es una sublista de ys. Por
-- ejemplo,
--    sublista [5, 2] [3, 2, 5]  ==  True
--    sublista [5, 2] [3, 4, 5]  ==  False
sublista :: Ord a => [a] -> [a] -> Bool
sublista [] _      = True
sublista (x:xs) ys = elem x ys && sublista xs ys

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

-- La propiedad es
prop_subconjunto :: Conj Int -> Conj Int -> Bool
prop_subconjunto c1 c2 =
  all (== subconjunto c1 c2)
```

```
      [subconjunto2 c1 c2,
       subconjunto3 c1 c2,
       subconjunto4 c1 c2]

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

8.7.2. En Python

```
# ---------------------------------------------------------------------
# Utilizando el [tipo abstracto de datos de los conjuntos](https://bit.ly/3HbB7fo
# definir la función
#    subconjunto :: Ord a => Conj a -> Conj a -> Bool
# tal que (subconjunto c1 c2) se verifica si todos los elementos de c1
# pertenecen a c2. Por ejemplo,
#    >>> ej1 = inserta(5, inserta(2, vacio()))
#    >>> ej2 = inserta(3, inserta(2, inserta(5, vacio())))
#    >>> ej3 = inserta(3, inserta(4, inserta(5, vacio())))
#    >>> subconjunto(ej1, ej2)
#    True
#    >>> subconjunto(ej1, ej3)
#    False
# ---------------------------------------------------------------------

# pylint: disable=unused-import

from __future__ import annotations

from abc import abstractmethod
from copy import deepcopy
from typing import Protocol, TypeVar

from hypothesis import given

from src.TAD.conjunto import (Conj, conjuntoAleatorio, elimina, esVacio,
                              inserta, menor, pertenece, vacio)
from src.TAD_Transformaciones_conjuntos_listas import conjuntoAlista
```

```python
class Comparable(Protocol):
    @abstractmethod
    def __lt__(self: A, otro: A) -> bool:
        pass

A = TypeVar('A', bound=Comparable)

# 1ª solución
# ===========

def subconjunto(c1: Conj[A], c2: Conj[A]) -> bool:
    if esVacio(c1):
        return True
    mc1 = menor(c1)
    rc1 = elimina(mc1, c1)
    return pertenece(mc1, c2) and subconjunto(rc1, c2)

# 2ª solución
# ===========

def subconjunto2(c1: Conj[A], c2: Conj[A]) -> bool:
    return all((pertenece(x, c2) for x in conjuntoAlista(c1)))

# La función conjuntoAlista está definida en el ejercicio
# ”Transformaciones entre conjuntos y listas” que se encuentra en
# https://bit.ly/3RexzxH

# 3ª solución
# ===========

# (sublista xs ys) se verifica si xs es una sublista de ys. Por
# ejemplo,
#    sublista [5, 2] [3, 2, 5]  ==  True
#    sublista [5, 2] [3, 4, 5]  ==  False
def sublista(xs: list[A], ys: list[A]) -> bool:
    if not xs:
        return True
    return xs[0] in ys and sublista(xs[1:], ys)

def subconjunto3(c1: Conj[A], c2: Conj[A]) -> bool:
```

```
    return sublista(conjuntoAlista(c1), conjuntoAlista(c2))

# 4ª solución
# ===========

def subconjunto4(c1: Conj[A], c2: Conj[A]) -> bool:
    while not esVacio(c1):
        mc1 = menor(c1)
        if not pertenece(mc1, c2):
            return False
        c1 = elimina(mc1, c1)
    return True

# 5ª solución
# ===========

def subconjunto5Aux(c1: Conj[A], c2: Conj[A]) -> bool:
    while not c1.esVacio():
        mc1 = c1.menor()
        if not c2.pertenece(mc1):
            return False
        c1.elimina(mc1)
    return True

def subconjunto5(c1: Conj[A], c2: Conj[A]) -> bool:
    _c1 = deepcopy(c1)
    return subconjunto5Aux(_c1, c2)

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

# La propiedad es
@given(c1=conjuntoAleatorio(), c2=conjuntoAleatorio())
def test_subconjunto(c1: Conj[int], c2: Conj[int]) -> None:
    r = subconjunto(c1, c2)
    assert subconjunto2(c1, c2) == r
    assert subconjunto3(c1, c2) == r
    assert subconjunto4(c1, c2) == r
    assert subconjunto5(c1, c2) == r
```

```
# La comprobación de las propiedades es
#    > poetry run pytest -q TAD_subconjunto.py
#    1 passed in 0.37s
```

8.8. Reconocimiento de subconjunto propio

8.8.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Utilizando el [tipo abstracto de datos de los conjuntos](https://bit.ly/3HbB7f
-- definir la función
--    subconjuntoPropio :: Ord a => Conj a -> Conj a -> Bool
-- tal (subconjuntoPropio c1 c2) se verifica si c1 es un subconjunto
-- propio de c2. Por ejemplo,
--    λ> ej1 = inserta 5 (inserta 2 vacio)
--    λ> ej2 = inserta 3 (inserta 2 (inserta 5 vacio))
--    λ> ej3 = inserta 3 (inserta 4 (inserta 5 vacio))
--    λ> ej4 = inserta 2 (inserta 5 vacio)
--    λ> subconjuntoPropio ej1 ej2
--    True
--    λ> subconjuntoPropio ej1 ej3
--    False
--    λ> subconjuntoPropio ej1 ej4
--    False
-- ---------------------------------------------------------------------

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

module TAD_subconjuntoPropio where

import TAD.Conjunto (Conj, vacio, inserta)
import TAD_subconjunto (subconjunto)

subconjuntoPropio :: Ord a => Conj a -> Conj a -> Bool
subconjuntoPropio c1 c2 =
  subconjunto c1 c2 && c1 /= c2

-- La función subconjunto está definida en el ejercicio
-- "Reconocimiento de subconjuntos" que se encuentra en
-- https://bit.ly/3wPBtU5
```

8.8.2. En Python

```python
# ---------------------------------------------------------------------
# Utilizando el [tipo abstracto de datos de los conjuntos](https://bit.ly/3HbB7fo
# definir la función
#    subconjuntoPropio : (Conj[A], Conj[A]) -> bool
# tal subconjuntoPropio(c1, c2) se verifica si c1 es un subconjunto
# propio de c2. Por ejemplo,
#    >>> ej1 = inserta(5, inserta(2, vacio()))
#    >>> ej2 = inserta(3, inserta(2, inserta(5, vacio())))
#    >>> ej3 = inserta(3, inserta(4, inserta(5, vacio())))
#    >>> ej4 = inserta(2, inserta(5, vacio()))
#    >>> subconjuntoPropio(ej1, ej2)
#    True
#    >>> subconjuntoPropio(ej1, ej3)
#    False
#    >>> subconjuntoPropio(ej1, ej4)
#    False
# ---------------------------------------------------------------------

# pylint: disable=unused-import

from __future__ import annotations

from abc import abstractmethod
from typing import Protocol, TypeVar

from src.TAD.conjunto import Conj, inserta, vacio
from src.TAD_subconjunto import subconjunto


class Comparable(Protocol):
    @abstractmethod
    def __lt__(self: A, otro: A) -> bool:
        pass

A = TypeVar('A', bound=Comparable)

def subconjuntoPropio(c1: Conj[A], c2: Conj[A]) -> bool:
    return subconjunto(c1, c2) and c1 != c2
```

```
# La función subconjunto está definida en el ejercicio
# "Reconocimiento de subconjuntos" que se encuentra en
# https://bit.ly/3wPBtU5
```

8.9. Conjunto unitario

8.9.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Utilizando el [tipo abstracto de datos de los conjuntos](https://bit.ly/3HbB7f
-- definir la función
--    unitario :: Ord a => a -> Conj a
-- tal que (unitario x) es el conjunto {x}. Por ejemplo,
--    unitario 5 == {5}
-- ---------------------------------------------------------------------

module TAD_Conjunto_unitario where

import TAD.Conjunto (Conj, vacio, inserta)

unitario :: Ord a => a -> Conj a
unitario x = inserta x vacio
```

8.9.2. En Python

```python
# ---------------------------------------------------------------------
# Utilizando el [tipo abstracto de datos de los conjuntos](https://bit.ly/3HbB7fo
# definir la función
#    unitario :: Ord a => a -> Conj a
# tal que (unitario x) es el conjunto {x}. Por ejemplo,
#    unitario 5 == {5}
# ---------------------------------------------------------------------

from __future__ import annotations

from abc import abstractmethod
from typing import Protocol, TypeVar

from src.TAD.conjunto import Conj, inserta, vacio
```

```
class Comparable(Protocol):
    @abstractmethod
    def __lt__(self: A, otro: A) -> bool:
        pass

A = TypeVar('A', bound=Comparable)

def unitario(x: A) -> Conj[A]:
    return inserta(x, vacio())
```

8.10. Número de elementos de un conjunto

8.10.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Utilizando el [tipo abstracto de datos de los conjuntos](https://bit.ly/3HbB7f
-- definir la función
--    cardinal :: Conj a -> Int
-- tal que (cardinal c) es el número de elementos del conjunto c. Por
-- ejemplo,
--    cardinal (inserta 4 (inserta 5 vacio))             == 2
--    cardinal (inserta 4 (inserta 5 (inserta 4 vacio))) == 2
-- ---------------------------------------------------------------------

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

module TAD_Numero_de_elementos_de_un_conjunto where

import TAD.Conjunto (Conj, vacio, inserta, menor, elimina, esVacio)
import TAD_Transformaciones_conjuntos_listas (conjuntoAlista)
import Test.QuickCheck

-- 1ª solución
-- ===========

cardinal :: Ord a => Conj a -> Int
cardinal c
  | esVacio c = 0
```

```
  | otherwise = 1 + cardinal (elimina (menor c) c)

-- 2ª solución
-- ===========

cardinal2 :: Ord a => Conj a -> Int
cardinal2 = length . conjuntoAlista

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

-- La propiedad es
prop_cardinal :: Conj Int -> Bool
prop_cardinal c =
  cardinal c == cardinal2 c

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

8.10.2. En Python

```
# ---------------------------------------------------------------------
# Utilizando el [tipo abstracto de datos de los conjuntos](https://bit.ly/3HbB7fo
# definir la función
#    cardinal : (Conj[A]) -> int
# tal que cardinal(c) es el número de elementos del conjunto c. Por
# ejemplo,
#    cardinal(inserta(4, inserta(5, vacio()))) == 2
#    cardinal(inserta(4, inserta(5, inserta(4, vacio())))) == 2
# ---------------------------------------------------------------------

# pylint: disable=unused-import

from __future__ import annotations

from abc import abstractmethod
from copy import deepcopy
from typing import Protocol, TypeVar
```

```python
from hypothesis import given

from src.TAD.conjunto import (Conj, conjuntoAleatorio, elimina, esVacio,
                              inserta, menor, vacio)
from src.TAD_Transformaciones_conjuntos_listas import conjuntoAlista


class Comparable(Protocol):
    @abstractmethod
    def __lt__(self: A, otro: A) -> bool:
        pass

A = TypeVar('A', bound=Comparable)

# 1ª solución
# ===========

def cardinal(c: Conj[A]) -> int:
    if esVacio(c):
        return 0
    return 1 + cardinal(elimina(menor(c), c))

# 2ª solución
# ===========

def cardinal2(c: Conj[A]) -> int:
    return len(conjuntoAlista(c))

# 3ª solución
# ===========

def cardinal3(c: Conj[A]) -> int:
    r = 0
    while not esVacio(c):
        r = r + 1
        c = elimina(menor(c), c)
    return r

# 4ª solución
# ===========
```

```python
def cardinal4Aux(c: Conj[A]) -> int:
    r = 0
    while not c.esVacio():
        r = r + 1
        c.elimina(menor(c))
    return r

def cardinal4(c: Conj[A]) -> int:
    _c = deepcopy(c)
    return cardinal4Aux(_c)

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

@given(c=conjuntoAleatorio())
def test_cardinal(c: Conj[int]) -> None:
    r = cardinal(c)
    assert cardinal2(c) == r
    assert cardinal3(c) == r
    assert cardinal3(c) == r

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

8.11. Unión de dos conjuntos

8.11.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Utilizando el [tipo abstracto de datos de los conjuntos](https://bit.ly/3HbB7f
-- definir la función
--    union :: Ord a => Conj a -> Conj a -> Conj a
-- tal (union c1 c2) es la unión de ambos conjuntos. Por ejemplo,
--    λ> ej1 = inserta 3 (inserta 5 vacio)
--    λ> ej2 = inserta 4 (inserta 3 vacio)
--    λ> union ej1 ej2
--    {3, 4, 5}
-- ---------------------------------------------------------------------
```

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

module TAD_Union_de_dos_conjuntos where

import TAD.Conjunto (Conj, vacio, inserta, menor, elimina, esVacio)
import TAD_Transformaciones_conjuntos_listas (conjuntoAlista, listaAconjunto)
import qualified Data.List as L (union)
import Test.QuickCheck

-- 1ª solución
-- ===========

union :: Ord a => Conj a -> Conj a -> Conj a
union c1 c2
  | esVacio c1 = c2
  | otherwise  = inserta mc1 (rc1 `union` c2)
  where mc1 = menor c1
        rc1 = elimina mc1 c1

-- 2ª solución
-- ===========

union2 :: Ord a => Conj a -> Conj a -> Conj a
union2 c1 c2 =
  foldr inserta c2 (conjuntoAlista c1)

-- La función conjuntoAlista está definida en el ejercicio
-- "Transformaciones entre conjuntos y listas" que se encuentra en
-- https://bit.ly/3RexzxH

-- 3ª solución
-- ===========

union3 :: Ord a => Conj a -> Conj a -> Conj a
union3 c1 c2 =
  listaAconjunto (conjuntoAlista c1 `L.union` conjuntoAlista c2)

-- La función listaAconjunto está definida en el ejercicio
-- "Transformaciones entre conjuntos y listas" que se encuentra en
```

```
-- https://bit.ly/3RexzxH

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

-- La propiedad es
prop_union :: Conj Int -> Conj Int -> Bool
prop_union c1 c2 =
  all (== union c1 c2)
      [union2 c1 c2,
       union3 c1 c2]

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

8.11.2. En Python

```
# ---------------------------------------------------------------------
# Utilizando el [tipo abstracto de datos de los conjuntos](https://bit.ly/3HbB7fo
# definir la función
#    union : (Conj[A], Conj[A]) -> Conj[A]
# tal (union c1 c2) es la unión de ambos conjuntos. Por ejemplo,
#    >>> ej1 = inserta(3, inserta(5, vacio()))
#    >>> ej2 = inserta(4, inserta(3, vacio()))
#    >>> union(ej1, ej2)
#    {3, 4, 5}
# ---------------------------------------------------------------------

# pylint: disable=unused-import

from __future__ import annotations

from abc import abstractmethod
from copy import deepcopy
from functools import reduce
from typing import Protocol, TypeVar

from hypothesis import given
```

```python
from src.TAD.conjunto import (Conj, conjuntoAleatorio, elimina, esVacio,
                              inserta, menor, vacio)
from src.TAD_Transformaciones_conjuntos_listas import conjuntoAlista


class Comparable(Protocol):
    @abstractmethod
    def __lt__(self: A, otro: A) -> bool:
        pass

A = TypeVar('A', bound=Comparable)

# 1ª solución
# ===========

def union(c1: Conj[A], c2: Conj[A]) -> Conj[A]:
    if esVacio(c1):
        return c2
    mc1 = menor(c1)
    rc1 = elimina(mc1, c1)
    return inserta(mc1, union(rc1, c2))

# 2ª solución
# ===========

def union2(c1: Conj[A], c2: Conj[A]) -> Conj[A]:
    return reduce(lambda c, x: inserta(x, c), conjuntoAlista(c1), c2)

# La función conjuntoAlista está definida en el ejercicio
# ”Transformaciones entre conjuntos y listas” que se encuentra en
# https://bit.ly/3RexzxH

# 3ª solución
# ===========

def union3(c1: Conj[A], c2: Conj[A]) -> Conj[A]:
    r = c2
    while not esVacio(c1):
        mc1 = menor(c1)
        r = inserta(mc1, r)
```

```
        c1 = elimina(mc1, c1)
    return r

# 4ª solución
# ===========

def union4Aux(c1: Conj[A], c2: Conj[A]) -> Conj[A]:
    while not c1.esVacio():
        mc1 = menor(c1)
        c2.inserta(mc1)
        c1.elimina(mc1)
    return c2

def union4(c1: Conj[A], c2: Conj[A]) -> Conj[A]:
    _c1 = deepcopy(c1)
    _c2 = deepcopy(c2)
    return union4Aux(_c1, _c2)

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

# La propiedad es
@given(c1=conjuntoAleatorio(), c2=conjuntoAleatorio())
def test_union(c1: Conj[int], c2: Conj[int]) -> None:
    r = union(c1, c2)
    assert union2(c1, c2) == r
    assert union3(c1, c2) == r
    assert union4(c1, c2) == r

# La comprobación de las propiedades es
#    > poetry run pytest -q TAD_Union_de_dos_conjuntos.py
#    1 passed in 0.35s
```

8.12. Unión de varios conjuntos

8.12.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Utilizando el [tipo abstracto de datos de los conjuntos](https://bit.ly/3HbB7f
-- definir la función
```

```haskell
--    unionG:: Ord a => [Conj a] -> Conj a
-- tal (unionG cs) calcule la unión de la lista de conjuntos cd. Por
-- ejemplo,
--    λ> ej1 = inserta 3 (inserta 5 vacio)
--    λ> ej2 = inserta 5 (inserta 6 vacio)
--    λ> ej3 = inserta 3 (inserta 6 vacio)
--    λ> unionG [ej1, ej2, ej3]
--    {3, 5, 6}
-- ---------------------------------------------------------------------

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

module TAD_Union_de_varios_conjuntos where

import TAD.Conjunto (Conj, vacio, inserta)
import TAD_Union_de_dos_conjuntos (union)
import Test.QuickCheck

-- 1ª solución
-- ===========

unionG :: Ord a => [Conj a] -> Conj a
unionG []          = vacio
unionG (c:cs) = c `union` unionG cs

-- La función union está definida en el ejercicio
-- "Unión de dos conjuntos" que se encuentra en
-- https://bit.ly/3Y1jBl8

-- 2ª solución
-- ===========

unionG2 :: Ord a => [Conj a] -> Conj a
unionG2 = foldr union vacio

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

-- La propiedad es
prop_unionG :: [Conj Int] -> Bool
```

```
prop_unionG cs =
  unionG cs == unionG2 cs

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

8.12.2. En Python

```
# ---------------------------------------------------------------------
# Utilizando el [tipo abstracto de datos de los conjuntos](https://bit.ly/3HbB7fo
# definir la función
#    unionG : (list[Conj[A]]) -> Conj[A]
# tal unionG(cs) calcule la unión de la lista de conjuntos cd. Por
# ejemplo,
#    >>> ej1 = inserta(3, inserta(5, vacio()))
#    >>> ej2 = inserta(5, inserta(6, vacio()))
#    >>> ej3 = inserta(3, inserta(6, vacio()))
#    >>> unionG([ej1, ej2, ej3])
#    {3, 5, 6}
# ---------------------------------------------------------------------

# pylint: disable=unused-import

from __future__ import annotations

from abc import abstractmethod
from functools import reduce
from typing import Protocol, TypeVar

from hypothesis import given
from hypothesis import strategies as st

from src.TAD.conjunto import Conj, conjuntoAleatorio, inserta, vacio
from src.TAD_Union_de_dos_conjuntos import union


class Comparable(Protocol):
    @abstractmethod
    def __lt__(self: A, otro: A) -> bool:
```

```
        pass

A = TypeVar('A', bound=Comparable)

# 1ª solución
# ===========

def unionG(cs: list[Conj[A]]) -> Conj[A]:
    if not cs:
        return vacio()
    return union(cs[0], unionG(cs[1:]))

# La función union está definida en el ejercicio
# "Unión de dos conjuntos" que se encuentra en
# https://bit.ly/3Y1jBl8

# 2ª solución
# ===========

def unionG2(cs: list[Conj[A]]) -> Conj[A]:
    return reduce(union, cs, vacio())

# 3ª solución
# ===========

def unionG3(cs: list[Conj[A]]) -> Conj[A]:
    r: Conj[A] = vacio()
    for c in cs:
        r = union(c, r)
    return r

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

# La propiedad es
@given(st.lists(conjuntoAleatorio(), max_size=10))
def test_union(cs: list[Conj[int]]) -> None:
    r = unionG(cs)
    assert unionG2(cs) == r
    assert unionG3(cs) == r
```

```
# La comprobación de las propiedades es
#    > poetry run pytest -q TAD_Union_de_varios_conjuntos.py
#    1 passed in 0.75s
```

8.13. Intersección de dos conjuntos

8.13.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Utilizando el [tipo abstracto de datos de los conjuntos](https://bit.ly/3HbB7f
-- definir la función
--    interseccion :: Ord a => Conj a -> Conj a -> Conj a
-- tal que (interseccion c1 c2) es la intersección de los conjuntos c1 y
-- c2. Por ejemplo,
--    λ> ej1 = inserta 3 (inserta 5 (inserta 2 vacio))
--    λ> ej2 = inserta 2 (inserta 4 (inserta 3 vacio))
--    λ> interseccion ej1 ej2
--    {2, 3}
-- ---------------------------------------------------------------------

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

module TAD_Interseccion_de_dos_conjuntos where

import TAD.Conjunto (Conj, vacio, inserta, menor, elimina, pertenece, esVacio)
import TAD_Transformaciones_conjuntos_listas (conjuntoAlista, listaAconjunto)
import Data.List (intersect)
import Test.QuickCheck

-- 1ª solución
-- ===========

interseccion :: Ord a => Conj a -> Conj a -> Conj a
interseccion c1 c2
  | esVacio c1        = vacio
  | pertenece mc1 c2 = inserta mc1 (interseccion rc1 c2)
  | otherwise        = interseccion rc1 c2
  where mc1 = menor c1
        rc1 = elimina mc1 c1
```

```
-- 2ª solución
-- ===========

interseccion2 :: Ord a => Conj a -> Conj a -> Conj a
interseccion2 c1 c2 =
  listaAconjunto [x | x <- conjuntoAlista c1, x `pertenece` c2]

-- Las funciones conjuntoAlista y listaAconjunto está definida en el
-- ejercicio Transformaciones entre conjuntos y listas" que se encuentra
-- en https://bit.ly/3RexzxH

-- 3ª solución
-- ===========

interseccion3 :: Ord a => Conj a -> Conj a -> Conj a
interseccion3 c1 c2 =
  listaAconjunto (conjuntoAlista c1 `intersect` conjuntoAlista c2)

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

-- La propiedad es
prop_interseccion :: Conj Int -> Conj Int -> Bool
prop_interseccion c1 c2 =
  all (== interseccion c1 c2)
      [interseccion2 c1 c2,
       interseccion3 c1 c2]

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

8.13.2. En Python

```
# ---------------------------------------------------------------------
# Utilizando el [tipo abstracto de datos de los conjuntos](https://bit.ly/3HbB7fo
# definir la función
#    interseccion : (Conj[A], Conj[A]) -> Conj[A]
# tal que interseccion(c1, c2) es la intersección de los conjuntos c1 y
```

```
# c2. Por ejemplo,
#    >>> ej1 = inserta(3, inserta(5, inserta(2, vacio())))
#    >>> ej2 = inserta(2, inserta(4, inserta(3, vacio())))
#    >>> interseccion(ej1, ej2)
#    {2, 3}
# ---------------------------------------------------------------------

from __future__ import annotations

from abc import abstractmethod
from copy import deepcopy
from typing import Protocol, TypeVar

from hypothesis import given

from src.TAD.conjunto import (Conj, conjuntoAleatorio, elimina, esVacio,
                              inserta, menor, pertenece, vacio)
from src.TAD_Transformaciones_conjuntos_listas import (conjuntoAlista,
                                                       listaAconjunto)


class Comparable(Protocol):
    @abstractmethod
    def __lt__(self: A, otro: A) -> bool:
        pass

A = TypeVar('A', bound=Comparable)

# 1ª solución
# ===========

def interseccion(c1: Conj[A], c2: Conj[A]) -> Conj[A]:
    if esVacio(c1):
        return vacio()
    mc1 = menor(c1)
    rc1 = elimina(mc1, c1)
    if pertenece(mc1, c2):
        return inserta(mc1, interseccion(rc1, c2))
    return interseccion(rc1, c2)
```

```
# 2ª solución
# ===========

def interseccion2(c1: Conj[A], c2: Conj[A]) -> Conj[A]:
    return listaAconjunto([x for x in conjuntoAlista(c1)
                           if pertenece(x, c2)])
#
# Las funciones conjuntoAlista y listaAconjunto está definida en el
# ejercicio Transformaciones entre conjuntos y listas” que se encuentra
# en https://bit.ly/3RexzxH

# 3ª solución
# ===========

def interseccion3(c1: Conj[A], c2: Conj[A]) -> Conj[A]:
    r: Conj[A] = vacio()
    while not esVacio(c1):
        mc1 = menor(c1)
        c1 = elimina(mc1, c1)
        if pertenece(mc1, c2):
            r = inserta(mc1, r)
    return r

# 4ª solución
# ===========

def interseccion4Aux(c1: Conj[A], c2: Conj[A]) -> Conj[A]:
    r: Conj[A] = vacio()
    while not c1.esVacio():
        mc1 = c1.menor()
        c1.elimina(mc1)
        if c2.pertenece(mc1):
            r.inserta(mc1)
    return r

def interseccion4(c1: Conj[A], c2: Conj[A]) -> Conj[A]:
    _c1 = deepcopy(c1)
    return interseccion4Aux(_c1, c2)

# Comprobación de equivalencia
```

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

# La propiedad es
@given(c1=conjuntoAleatorio(), c2=conjuntoAleatorio())
def test_interseccion(c1: Conj[int], c2: Conj[int]) -> None:
    r = interseccion(c1, c2)
    assert interseccion2(c1, c2) == r
    assert interseccion3(c1, c2) == r
    assert interseccion4(c1, c2) == r

# La comprobación de las propiedades es
#    > poetry run pytest -q TAD_Interseccion_de_dos_conjuntos.py
#    1 passed in 0.30s
```

8.14. Intersección de varios conjuntos

8.14.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Utilizando el tipo abstracto de datos de los conjuntos
-- (https://bit.ly/3HbB7fo) definir la función
--    interseccionG:: Ord a => [Conj a] -> Conj a
-- tal que (interseccionG cs) es la intersección de la lista de
-- conjuntos cs. Por ejemplo,
--    λ> ej1 = inserta 2 (inserta 3 (inserta 5 vacio))
--    λ> ej2 = inserta 5 (inserta 2 (inserta 7 vacio))
--    λ> ej3 = inserta 3 (inserta 2 (inserta 5 vacio))
--    λ> interseccionG [ej1, ej2, ej3]
--    {2, 5}
-- ---------------------------------------------------------------------

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

module TAD_Interseccion_de_varios_conjuntos where

import TAD.Conjunto (Conj, vacio, inserta)
import TAD_Interseccion_de_dos_conjuntos (interseccion)
import Test.QuickCheck
```

```haskell
-- 1ª solución
-- ===========

interseccionG :: Ord a => [Conj a] -> Conj a
interseccionG [c]      = c
interseccionG (cs:css) = interseccion cs (interseccionG css)

-- La función interseccion está definida en el ejercicio
-- ”Intersección de dos conjuntos” que se encuentra en
-- https://bit.ly/3jDL9xZ

-- 2ª solución
-- ===========

interseccionG2 :: Ord a => [Conj a] -> Conj a
interseccionG2 = foldr1 interseccion

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

-- La propiedad es
prop_interseccionG :: NonEmptyList (Conj Int) -> Bool
prop_interseccionG (NonEmpty cs) =
  interseccionG cs == interseccionG2 cs

-- La comprobación es
--    λ> quickCheck prop_interseccionG1
--    +++ OK, passed 100 tests.
```

8.14.2. En Python

```python
# ---------------------------------------------------------------------
# Utilizando el tipo abstracto de datos de los conjuntos
# (https://bit.ly/3HbB7fo) definir la función
#    interseccionG : (list[Conj[A]]) -> Conj[A]
# tal que interseccionG(cs) es la intersección de la lista de
# conjuntos cs. Por ejemplo,
#    >>> ej1 = inserta(2, inserta(3, inserta(5, vacio())))
#    >>> ej2 = inserta(5, inserta(2, inserta(7, vacio())))
#    >>> ej3 = inserta(3, inserta(2, inserta(5, vacio())))
```

```python
#    >>> interseccionG([ej1, ej2, ej3])
#    {2, 5}
# ---------------------------------------------------------------------

# pylint: disable=unused-import

from __future__ import annotations

from abc import abstractmethod
from functools import reduce
from typing import Protocol, TypeVar

from hypothesis import given
from hypothesis import strategies as st

from src.TAD.conjunto import Conj, conjuntoAleatorio, inserta, vacio
from src.TAD_Interseccion_de_dos_conjuntos import interseccion


class Comparable(Protocol):
    @abstractmethod
    def __lt__(self: A, otro: A) -> bool:
        pass

A = TypeVar('A', bound=Comparable)

# 1ª solución
# ===========

def interseccionG(cs: list[Conj[A]]) -> Conj[A]:
    if len(cs) == 1:
        return cs[0]
    return interseccion(cs[0], interseccionG(cs[1:]))

# 2ª solución
# ===========

def interseccionG2(cs: list[Conj[A]]) -> Conj[A]:
    return reduce(interseccion, cs[1:], cs[0])
```

```
# 3ª solución
# ===========

def interseccionG3(cs: list[Conj[A]]) -> Conj[A]:
    r = cs[0]
    for c in cs[1:]:
        r = interseccion(c, r)
    return r

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

# La propiedad es
@given(st.lists(conjuntoAleatorio(), min_size=1, max_size=10))
def test_interseccionG(cs: list[Conj[int]]) -> None:
    r = interseccionG(cs)
    assert interseccionG2(cs) == r
    assert interseccionG3(cs) == r

# La comprobación de las propiedades es
#    > poetry run pytest -q TAD_Interseccion_de_varios_conjuntos.py
#    1 passed in 0.60s
```

8.15. Conjuntos disjuntos

8.15.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Utilizando el tipo abstracto de datos de los conjuntos
-- (https://bit.ly/3HbB7fo) definir la función
--    disjuntos :: Ord a => Conj a -> Conj a -> Bool
-- tal que (disjuntos c1 c2) se verifica si los conjuntos c1 y c2 son
-- disjuntos. Por ejemplo,
--    λ> ej1 = inserta 2 (inserta 5 vacio)
--    λ> ej2 = inserta 4 (inserta 3 vacio)
--    λ> ej3 = inserta 5 (inserta 3 vacio)
--    λ> disjuntos ej1 ej2
--    True
--    λ> disjuntos ej1 ej3
--    False
```

```haskell
-- ---------------------------------------------------------------------

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

module TAD_Conjuntos_disjuntos where

import TAD.Conjunto (Conj, vacio, inserta, esVacio, menor, elimina, pertenece)
import TAD_Interseccion_de_dos_conjuntos (interseccion)
import TAD_Transformaciones_conjuntos_listas (conjuntoAlista)
import Test.QuickCheck

-- 1ª solución
-- ===========

disjuntos :: Ord a => Conj a -> Conj a -> Bool
disjuntos c1 c2 = esVacio (interseccion c1 c2)

-- La función interseccion está definida en el ejercicio
-- "Intersección de dos conjuntos" que se encuentra en
-- https://bit.ly/3jDL9xZ

-- 2ª solución
-- ===========

disjuntos2 :: Ord a => Conj a -> Conj a -> Bool
disjuntos2 c1 c2
  | esVacio c1 = True
  | pertenece mc1 c2 = False
  | otherwise        = disjuntos2 rc1 c2
  where mc1 = menor c1
        rc1 = elimina mc1 c1

-- 3ª solución
-- ===========

disjuntos3 :: Ord a => Conj a -> Conj a -> Bool
disjuntos3 c1 c2 =
  all (`notElem` ys) xs
  where xs = conjuntoAlista c1
        ys = conjuntoAlista c2
```

```haskell
-- La función conjuntoAlista está definida en el ejercicio
-- "Transformaciones entre conjuntos y listas" que se encuentra en
-- https://bit.ly/3RexzxH

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

-- La propiedad es
prop_disjuntos :: Conj Int -> Conj Int -> Bool
prop_disjuntos c1 c2 =
  all (== disjuntos c1 c2)
      [disjuntos2 c1 c2,
       disjuntos3 c1 c2]

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

8.15.2. En Python

```python
# ---------------------------------------------------------------------
# Utilizando el tipo abstracto de datos de los conjuntos
# (https://bit.ly/3HbB7fo) definir la función
#    disjuntos : (Conj[A], Conj[A]) -> bool
# tal que disjuntos(c1, c2) se verifica si los conjuntos c1 y c2 son
# disjuntos. Por ejemplo,
#    >>> ej1 = inserta(2, inserta(5, vacio()))
#    >>> ej2 = inserta(4, inserta(3, vacio()))
#    >>> ej3 = inserta(5, inserta(3, vacio()))
#    >>> disjuntos(ej1, ej2)
#    True
#    >>> disjuntos(ej1, ej3)
#    False
# ---------------------------------------------------------------------

# pylint: disable=unused-import

from __future__ import annotations
```

```python
from abc import abstractmethod
from copy import deepcopy
from typing import Protocol, TypeVar

from hypothesis import given

from src.TAD.conjunto import (Conj, conjuntoAleatorio, elimina, esVacio,
                              inserta, menor, pertenece, vacio)
from src.TAD_Interseccion_de_dos_conjuntos import interseccion
from src.TAD_Transformaciones_conjuntos_listas import conjuntoAlista


class Comparable(Protocol):
    @abstractmethod
    def __lt__(self: A, otro: A) -> bool:
        pass

A = TypeVar('A', bound=Comparable)

# 1ª solución
# ===========

def disjuntos(c1: Conj[A], c2: Conj[A]) -> bool:
    return esVacio(interseccion(c1, c2))

# 2ª solución
# ===========

def disjuntos2(c1: Conj[A], c2: Conj[A]) -> bool:
    if esVacio(c1):
        return True
    mc1 = menor(c1)
    rc1 = elimina(mc1, c1)
    if pertenece(mc1, c2):
        return False
    return disjuntos2(rc1, c2)

# 3ª solución
# ===========
```

```python
def disjuntos3(c1: Conj[A], c2: Conj[A]) -> bool:
    xs = conjuntoAlista(c1)
    ys = conjuntoAlista(c2)
    return all((x not in ys for x in xs))

# La función conjuntoAlista está definida en el ejercicio
# "Transformaciones entre conjuntos y listas" que se encuentra en
# https://bit.ly/3RexzxH

# 4ª solución
# ===========

def disjuntos4Aux(c1: Conj[A], c2: Conj[A]) -> bool:
    while not esVacio(c1):
        mc1 = menor(c1)
        if pertenece(mc1, c2):
            return False
        c1 = elimina(mc1, c1)
    return True

def disjuntos4(c1: Conj[A], c2: Conj[A]) -> bool:
    _c1 = deepcopy(c1)
    return disjuntos4Aux(_c1, c2)

# 5ª solución
# ===========

def disjuntos5Aux(c1: Conj[A], c2: Conj[A]) -> bool:
    while not c1.esVacio():
        mc1 = c1.menor()
        if c2.pertenece(mc1):
            return False
        c1.elimina(mc1)
    return True

def disjuntos5(c1: Conj[A], c2: Conj[A]) -> bool:
    _c1 = deepcopy(c1)
    return disjuntos5Aux(_c1, c2)

# Comprobación de equivalencia
```

```
# ============================

# La propiedad es
@given(c1=conjuntoAleatorio(), c2=conjuntoAleatorio())
def test_disjuntos(c1: Conj[int], c2: Conj[int]) -> None:
    r = disjuntos(c1, c2)
    assert disjuntos2(c1, c2) == r
    assert disjuntos3(c1, c2) == r
    assert disjuntos4(c1, c2) == r
    assert disjuntos5(c1, c2) == r

# La comprobación de las propiedades es
#    > poetry run pytest -q TAD_Conjuntos_disjuntos.py
#    1 passed in 0.34s
```

8.16. Diferencia de conjuntos

8.16.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Utilizando el tipo abstracto de datos de los conjuntos
-- (https://bit.ly/3HbB7fo) definir la función
--    diferencia :: Ord a => Conj a -> Conj a -> Conj a
-- tal que (diferencia c1 c2) es el conjunto de los elementos de c1 que
-- no son elementos de c2. Por ejemplo,
--    λ> ej1 = inserta 5 (inserta 3 (inserta 2 (inserta 7 vacio)))
--    λ> ej2 = inserta 7 (inserta 4 (inserta 3 vacio))
--    λ> diferencia ej1 ej2
--    {2, 5}
--    λ> diferencia ej2 ej1
--    {4}
--    λ> diferencia ej1 ej1
--    {}
-- ---------------------------------------------------------------------

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

module TAD_Diferencia_de_conjuntos where

import TAD.Conjunto (Conj, vacio, inserta, menor, elimina, pertenece, esVacio)
```

```haskell
import TAD_Transformaciones_conjuntos_listas (conjuntoAlista, listaAconjunto)
import Test.QuickCheck

-- 1ª solución
-- ===========

diferencia :: Ord a => Conj a -> Conj a -> Conj a
diferencia c1 c2
  | esVacio c1       = vacio
  | pertenece mc1 c2 = diferencia rc1 c2
  | otherwise        = inserta mc1 (diferencia rc1 c2)
  where mc1 = menor c1
        rc1 = elimina mc1 c1

-- 2ª solución
-- ===========

diferencia2 :: Ord a => Conj a -> Conj a -> Conj a
diferencia2 c1 c2 =
  listaAconjunto [x | x <- conjuntoAlista c1, not (pertenece x c2)]

-- Las funciones conjuntoAlista y listaAconjunto está definida en el
-- ejercicio Transformaciones entre conjuntos y listas” que se encuentra
-- en https://bit.ly/3RexzxH

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

-- La propiedad es
prop_diferencia :: Conj Int -> Conj Int -> Bool
prop_diferencia c1 c2 =
  diferencia c1 c2 == diferencia2 c1 c2

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

8.16.2. En Python

```
# ---------------------------------------------------------------------
# Utilizando el tipo abstracto de datos de los conjuntos
# (https://bit.ly/3HbB7fo) definir la función
#    diferencia : (Conj[A], Conj[A]) -> Conj[A]
# tal que diferencia(c1, c2) es el conjunto de los elementos de c1 que
# no son elementos de c2. Por ejemplo,
#    >>> ej1 = inserta(5, inserta(3, inserta(2, inserta(7, vacio()))))
#    >>> ej2 = inserta(7, inserta(4, inserta(3, vacio())))
#    >>> diferencia(ej1, ej2)
#    {2, 5}
#    >>> diferencia(ej2, ej1)
#    {4}
#    >>> diferencia(ej1, ej1)
#    {}
# ---------------------------------------------------------------------

from __future__ import annotations

from abc import abstractmethod
from copy import deepcopy
from typing import Protocol, TypeVar

from hypothesis import given

from src.TAD.conjunto import (Conj, conjuntoAleatorio, elimina, esVacio,
                              inserta, menor, pertenece, vacio)
from src.TAD_Transformaciones_conjuntos_listas import (conjuntoAlista,
                                                       listaAconjunto)


class Comparable(Protocol):
    @abstractmethod
    def __lt__(self: A, otro: A) -> bool:
        pass

A = TypeVar('A', bound=Comparable)

# 1ª solución
# ===========
```

```python
def diferencia(c1: Conj[A], c2: Conj[A]) -> Conj[A]:
    if esVacio(c1):
        return vacio()
    mc1 = menor(c1)
    rc1 = elimina(mc1, c1)
    if pertenece(mc1, c2):
        return diferencia(rc1, c2)
    return inserta(mc1, diferencia(rc1, c2))

# 2ª solución
# ===========

def diferencia2(c1: Conj[A], c2: Conj[A]) -> Conj[A]:
    return listaAconjunto([x for x in conjuntoAlista(c1)
                           if not pertenece(x, c2)])

# Las funciones conjuntoAlista y listaAconjunto está definida en el
# ejercicio Transformaciones entre conjuntos y listas” que se encuentra
# en https://bit.ly/3RexzxH

# 3ª solución
# ===========

def diferencia3Aux(c1: Conj[A], c2: Conj[A]) -> Conj[A]:
    r: Conj[A] = vacio()
    while not esVacio(c1):
        mc1 = menor(c1)
        if not pertenece(mc1, c2):
            r = inserta(mc1, r)
        c1 = elimina(mc1, c1)
    return r

def diferencia3(c1: Conj[A], c2: Conj[A]) -> Conj[A]:
    _c1 = deepcopy(c1)
    return diferencia3Aux(_c1, c2)

# 4ª solución
# ===========
```

```python
def diferencia4Aux(c1: Conj[A], c2: Conj[A]) -> Conj[A]:
    r: Conj[A] = Conj()
    while not c1.esVacio():
        mc1 = c1.menor()
        if not c2.pertenece(mc1):
            r.inserta(mc1)
        c1.elimina(mc1)
    return r

def diferencia4(c1: Conj[A], c2: Conj[A]) -> Conj[A]:
    _c1 = deepcopy(c1)
    return diferencia4Aux(_c1, c2)

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

# La propiedad es
@given(c1=conjuntoAleatorio(), c2=conjuntoAleatorio())
def test_diferencia(c1: Conj[int], c2: Conj[int]) -> None:
    r = diferencia(c1, c2)
    assert diferencia2(c1, c2) == r
    assert diferencia3(c1, c2) == r
    assert diferencia4(c1, c2) == r

# La comprobación de las propiedades es
#    > poetry run pytest -q TAD_Diferencia_de_conjuntos.py
#    1 passed in 0.31s
```

8.17. Diferencia simétrica

8.17.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Utilizando el tipo abstracto de datos de los conjuntos
-- (https://bit.ly/3HbB7fo) definir la función
--    diferenciaSimetrica :: Ord a => Conj a -> Conj a -> Conj a
-- tal que (diferenciaSimetrica c1 c2) es la diferencia simétrica de los
-- conjuntos c1 y c2. Por ejemplo,
--    λ> ej1 = inserta 5 (inserta 3 (inserta 2 (inserta 7 vacio)))
--    λ> ej2 = inserta 7 (inserta 4 (inserta 3 vacio))
```

```haskell
--    λ> diferenciaSimetrica ej1 ej2
--    {2, 4, 5}
-- ---------------------------------------------------------------------

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

module TAD_Diferencia_simetrica where

import TAD.Conjunto (Conj, vacio, inserta)
import TAD_Diferencia_de_conjuntos (diferencia)
import TAD_Interseccion_de_dos_conjuntos (interseccion)
import TAD_Union_de_dos_conjuntos (union)
import TAD_Transformaciones_conjuntos_listas (conjuntoAlista, listaAconjunto)

import Test.QuickCheck

-- 1ª solución
-- ===========

diferenciaSimetrica :: Ord a => Conj a -> Conj a -> Conj a
diferenciaSimetrica c1 c2 =
  diferencia (union c1 c2) (interseccion c1 c2)

-- 2ª solución
-- ===========

diferenciaSimetrica2 :: Ord a => Conj a -> Conj a -> Conj a
diferenciaSimetrica2 c1 c2 =
  listaAconjunto ([x | x <- xs, x `notElem` ys] ++
                  [y | y <- ys, y `notElem` xs])
  where xs = conjuntoAlista c1
        ys = conjuntoAlista c2

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

-- La propiedad es
prop_diferenciaSimetrica :: Conj Int -> Conj Int -> Bool
prop_diferenciaSimetrica c1 c2 =
  diferenciaSimetrica c1 c2 == diferenciaSimetrica2 c2 c1
```

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

8.17.2. En Python

```python
# ---------------------------------------------------------------------
# Utilizando el tipo abstracto de datos de los conjuntos
# (https://bit.ly/3HbB7fo) definir la función
#    diferenciaSimetrica : (Conj[A], Conj[A]) -> Conj[A]
# tal que diferenciaSimetrica(c1, c2) es la diferencia simétrica de los
# conjuntos c1 y c2. Por ejemplo,
#    >>> ej1 = inserta(5, inserta(3, inserta(2, inserta(7, vacio()))))
#    >>> ej2 = inserta(7, inserta(4, inserta(3, vacio())))
#    >>> diferenciaSimetrica(ej1, ej2)
#    {2, 4, 5}
# ---------------------------------------------------------------------

from __future__ import annotations

from abc import abstractmethod
from copy import deepcopy
from typing import Protocol, TypeVar

from hypothesis import given

from src.TAD.conjunto import (Conj, conjuntoAleatorio, elimina, esVacio,
                              inserta, menor, pertenece, vacio)
from src.TAD_Diferencia_de_conjuntos import diferencia
from src.TAD_Interseccion_de_dos_conjuntos import interseccion
from src.TAD_Transformaciones_conjuntos_listas import (conjuntoAlista,
                                                       listaAconjunto)
from src.TAD_Union_de_dos_conjuntos import union


class Comparable(Protocol):
    @abstractmethod
    def __lt__(self: A, otro: A) -> bool:
        pass
```

```
A = TypeVar('A', bound=Comparable)

# 1ª solución
# ===========

def diferenciaSimetrica(c1: Conj[A], c2: Conj[A]) -> Conj[A]:
    return diferencia(union(c1, c2), interseccion(c1, c2))

# 2ª solución
# ===========

def diferenciaSimetrica2(c1: Conj[A], c2: Conj[A]) -> Conj[A]:
    xs = conjuntoAlista(c1)
    ys = conjuntoAlista(c2)
    return listaAconjunto([x for x in xs if x not in ys] +
                          [y for y in ys if y not in xs])

# 3ª solución
# ===========

def diferenciaSimetrica3(c1: Conj[A], c2: Conj[A]) -> Conj[A]:
    r: Conj[A] = vacio()
    _c1 = deepcopy(c1)
    _c2 = deepcopy(c2)
    while not esVacio(_c1):
        mc1 = menor(_c1)
        if not pertenece(mc1, c2):
            r = inserta(mc1, r)
        _c1 = elimina(mc1, _c1)
    while not esVacio(_c2):
        mc2 = menor(_c2)
        if not pertenece(mc2, c1):
            r = inserta(mc2, r)
        _c2 = elimina(mc2, _c2)
    return r

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

```python
# La propiedad es
@given(c1=conjuntoAleatorio(), c2=conjuntoAleatorio())
def test_diferenciaSimetrica(c1: Conj[int], c2: Conj[int]) -> None:
    r = diferenciaSimetrica(c1, c2)
    assert diferenciaSimetrica2(c1, c2) == r
    assert diferenciaSimetrica3(c1, c2) == r

# La comprobación de las propiedades es
#    > poetry run pytest -q TAD_Diferencia_simetrica.py
#    1 passed in 0.30s
```

8.18. Subconjunto determinado por una propiedad

8.18.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Utilizando el tipo abstracto de datos de los conjuntos
-- (https://bit.ly/3HbB7fo) definir la función
--    filtra :: Ord a => (a -> Bool) -> Conj a -> Conj a
-- tal (filtra p c) es el conjunto de elementos de c que verifican el
-- predicado p. Por ejemplo,
--    λ> filtra even (inserta 5 (inserta 4 (inserta 7 (inserta 2 vacio))))
--    {2, 4}
--    λ> filtra odd (inserta 5 (inserta 4 (inserta 7 (inserta 2 vacio))))
--    {5, 7}
-- ---------------------------------------------------------------------

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

module TAD_Subconjunto_por_propiedad where

import TAD.Conjunto (Conj, vacio, inserta, esVacio, menor, elimina)
import TAD_Transformaciones_conjuntos_listas (conjuntoAlista, listaAconjunto)
import Test.QuickCheck.HigherOrder

-- 1ª solución
-- ===========
```

```haskell
filtra :: Ord a => (a -> Bool) -> Conj a -> Conj a
filtra p c
  | esVacio c = vacio
  | p mc      = inserta mc (filtra p rc)
  | otherwise = filtra p rc
  where mc = menor c
        rc = elimina mc c

-- 2ª solución
-- ===========

filtra2 :: Ord a => (a -> Bool) -> Conj a -> Conj a
filtra2 p c =
  listaAconjunto (filter p (conjuntoAlista c))

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

-- La propiedad es
prop_filtra :: (Int -> Bool) -> [Int] -> Bool
prop_filtra p xs =
  filtra p c == filtra2 p c
  where c = listaAconjunto xs

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

8.18.2. En Python

```python
# ---------------------------------------------------------------------
# Utilizando el tipo abstracto de datos de los conjuntos
# (https://bit.ly/3HbB7fo) definir la función
#    filtra : (Callable[[A], bool], Conj[A]) -> Conj[A]
# tal (filtra p c) es el conjunto de elementos de c que verifican el
# predicado p. Por ejemplo,
#    >>> ej = inserta(5, inserta(4, inserta(7, inserta(2, vacio()))))
#    >>> filtra(lambda x: x % 2 == 0, ej)
#    {2, 4}
#    >>> filtra(lambda x: x % 2 == 1, ej)
```

```python
#    {5, 7}
# ---------------------------------------------------------------------

# pylint: disable=unused-import

from __future__ import annotations

from abc import abstractmethod
from copy import deepcopy
from typing import Callable, Protocol, TypeVar

from hypothesis import given

from src.TAD.conjunto import (Conj, conjuntoAleatorio, elimina, esVacio,
                              inserta, menor, pertenece, vacio)
from src.TAD_Transformaciones_conjuntos_listas import (conjuntoAlista,
                                                        listaAconjunto)


class Comparable(Protocol):
    @abstractmethod
    def __lt__(self: A, otro: A) -> bool:
        pass

A = TypeVar('A', bound=Comparable)

# 1ª solución
# ===========

def filtra(p: Callable[[A], bool], c: Conj[A]) -> Conj[A]:
    if esVacio(c):
        return vacio()
    mc = menor(c)
    rc = elimina(mc, c)
    if p(mc):
        return inserta(mc, filtra(p, rc))
    return filtra(p, rc)

# 2ª solución
# ===========
```

```
def filtra2(p: Callable[[A], bool], c: Conj[A]) -> Conj[A]:
    return listaAconjunto(list(filter(p, conjuntoAlista(c))))

# 3ª solución
# ===========

def filtra3Aux(p: Callable[[A], bool], c: Conj[A]) -> Conj[A]:
    r: Conj[A] = vacio()
    while not esVacio(c):
        mc = menor(c)
        c = elimina(mc, c)
        if p(mc):
            r = inserta(mc, r)
    return r

def filtra3(p: Callable[[A], bool], c: Conj[A]) -> Conj[A]:
    _c = deepcopy(c)
    return filtra3Aux(p, _c)

# 4ª solución
# ===========

def filtra4Aux(p: Callable[[A], bool], c: Conj[A]) -> Conj[A]:
    r: Conj[A] = Conj()
    while not c.esVacio():
        mc = c.menor()
        c.elimina(mc)
        if p(mc):
            r.inserta(mc)
    return r

def filtra4(p: Callable[[A], bool], c: Conj[A]) -> Conj[A]:
    _c = deepcopy(c)
    return filtra4Aux(p, _c)

# Comprobación de equivalencia de las definiciones
# ================================================

# La propiedad es
```

```python
@given(c=conjuntoAleatorio())
def test_filtra(c: Conj[int]) -> None:
    r = filtra(lambda x: x % 2 == 0, c)
    assert filtra2(lambda x: x % 2 == 0, c) == r
    assert filtra3(lambda x: x % 2 == 0, c) == r
    assert filtra4(lambda x: x % 2 == 0, c) == r

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

8.19. Partición de un conjunto según una propiedad

8.19.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Utilizando el tipo abstracto de datos de los conjuntos
-- (https://bit.ly/3HbB7fo) definir la función
--    particion :: Ord a => (a -> Bool) -> Conj a -> (Conj a, Conj a)
-- tal que (particion c) es el par formado por dos conjuntos: el de sus
-- elementos que verifican p y el de los elementos que no lo
-- verifica. Por ejemplo,
--    λ> ej = inserta 5 (inserta 4 (inserta 7 (inserta 2 vacio)))
--    λ> particion even ej
--    ({2, 4},{5, 7})
-- ---------------------------------------------------------------------

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

module TAD_Particion_por_una_propiedad where

import TAD.Conjunto (Conj, vacio, inserta)
import TAD_Transformaciones_conjuntos_listas (conjuntoAlista, listaAconjunto)
import TAD_Subconjunto_por_propiedad (filtra)
import Data.List (partition)
import Test.QuickCheck.HigherOrder

-- 1ª solución
```

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

particion :: Ord a => (a -> Bool) -> Conj a -> (Conj a, Conj a)
particion p c = (filtra p c, filtra (not . p) c)

-- La función filtra está definida en el ejercicio
-- "Subconjunto determinado por una propiedad" que se encuentra en
-- https://bit.ly/3lplFoV

-- 2ª solución
-- ===========

particion2 :: Ord a => (a -> Bool) -> Conj a -> (Conj a, Conj a)
particion2 p c = (listaAconjunto xs, listaAconjunto ys)
  where
    (xs, ys) = partition p (conjuntoAlista c)

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

-- La propiedad es
prop_particion :: (Int -> Bool) -> [Int] -> Bool
prop_particion p xs =
  particion p c == particion2 p c
  where c = listaAconjunto xs

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

8.19.2. En Python

```
# ---------------------------------------------------------------------
# Utilizando el tipo abstracto de datos de los conjuntos
# (https://bit.ly/3HbB7fo) definir la función
#    particion : (Callable[[A], bool], Conj[A]) -> tuple[Conj[A], Conj[A]]
# tal que particion(c) es el par formado por dos conjuntos: el de sus
# elementos que verifican p y el de los elementos que no lo
# verifica. Por ejemplo,
#    >>> ej = inserta(5, inserta(4, inserta(7, inserta(2, vacio()))))
```

```python
#    >>> particion(lambda x: x % 2 == 0, ej)
#    ({2, 4}, {5, 7})
# ---------------------------------------------------------------------

from __future__ import annotations

from abc import abstractmethod
from copy import deepcopy
from typing import Callable, Protocol, TypeVar

from hypothesis import given

from src.TAD.conjunto import (Conj, conjuntoAleatorio, elimina, esVacio,
                              inserta, menor, vacio)
from src.TAD_Subconjunto_por_propiedad import filtra


class Comparable(Protocol):
    @abstractmethod
    def __lt__(self: A, otro: A) -> bool:
        pass

A = TypeVar('A', bound=Comparable)

# 1ª solución
# ===========

def particion(p: Callable[[A], bool],
              c: Conj[A]) -> tuple[Conj[A], Conj[A]]:
    return (filtra(p, c), filtra(lambda x: not p(x), c))

# La función filtra está definida en el ejercicio
# "Subconjunto determinado por una propiedad" que se encuentra en
# https://bit.ly/3lplFoV

# 2ª solución
# ===========

def particion2Aux(p: Callable[[A], bool],
                  c: Conj[A]) -> tuple[Conj[A], Conj[A]]:
```

```
    r: Conj[A] = vacio()
    s: Conj[A] = vacio()
    while not esVacio(c):
        mc = menor(c)
        c = elimina(mc, c)
        if p(mc):
            r = inserta(mc, r)
        else:
            s = inserta(mc, s)
    return (r, s)

def particion2(p: Callable[[A], bool],
               c: Conj[A]) -> tuple[Conj[A], Conj[A]]:
    _c = deepcopy(c)
    return particion2Aux(p, _c)

# 3ª solución
# ===========

def particion3Aux(p: Callable[[A], bool],
                  c: Conj[A]) -> tuple[Conj[A], Conj[A]]:
    r: Conj[A] = Conj()
    s: Conj[A] = Conj()
    while not c.esVacio():
        mc = c.menor()
        c.elimina(mc)
        if p(mc):
            r.inserta(mc)
        else:
            s.inserta(mc)
    return (r, s)

def particion3(p: Callable[[A], bool],
               c: Conj[A]) -> tuple[Conj[A], Conj[A]]:
    _c = deepcopy(c)
    return particion3Aux(p, _c)

# Comprobación de equivalencia de las definiciones
# ================================================
```

```
# La propiedad es
@given(c=conjuntoAleatorio())
def test_particion(c: Conj[int]) -> None:
    r = particion(lambda x: x % 2 == 0, c)
    assert particion2(lambda x: x % 2 == 0, c) == r
    assert particion3(lambda x: x % 2 == 0, c) == r

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

8.20. Partición según un número

8.20.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Utilizando el tipo abstracto de datos de los conjuntos
-- (https://bit.ly/3HbB7fo) definir la función
--    divide :: (Ord a) => a-> Conj a -> (Conj a, Conj a)
-- tal que (divide x c) es el par formado por dos subconjuntos de c: el
-- de los elementos menores o iguales que x y el de los mayores que x.
-- Por ejemplo,
--    λ> divide 5 (inserta 7 (inserta 2 (inserta 8 vacio)))
--    ({2},{7, 8})
-- ---------------------------------------------------------------------

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

module TAD_Particion_segun_un_numero where

import TAD.Conjunto (Conj, vacio, inserta, esVacio, menor, elimina)
import TAD_Particion_por_una_propiedad (particion)
import Test.QuickCheck

-- 1ª solución
-- ===========

divide :: Ord a => a-> Conj a -> (Conj a, Conj a)
divide x c
  | esVacio c = (vacio, vacio)
```

```
  | mc <= x    = (inserta mc c1, c2)
  | otherwise = (c1, inserta mc c2)
  where
    mc       = menor c
    rc       = elimina mc c
    (c1, c2) = divide x rc

-- 2ª solución
-- ===========

divide2 :: Ord a => a-> Conj a -> (Conj a, Conj a)
divide2 x = particion (<= x)

-- La función particion está definida en el ejercicio
-- ”Partición de un conjunto según una propiedad” que se encuentra en
-- https://bit.ly/3YCOah5

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

-- La propiedad es
prop_divide :: Int -> Conj Int -> Bool
prop_divide x c =
  divide x c == divide2 x c

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

8.20.2. En Python

```
# ---------------------------------------------------------------------
# Utilizando el tipo abstracto de datos de los conjuntos
# (https://bit.ly/3HbB7fo) definir la función
#    divide : (A, Conj[A]) -> tuple[Conj[A], Conj[A]]
# tal que (divide x c) es el par formado por dos subconjuntos de c: el
# de los elementos menores o iguales que x y el de los mayores que x.
# Por ejemplo,
#    >>> divide(5, inserta(7, inserta(2, inserta(8, vacio()))))
#    ({2}, {7, 8})
```

```
# ---------------------------------------------------------------------

from __future__ import annotations

from abc import abstractmethod
from copy import deepcopy
from typing import Protocol, TypeVar

from hypothesis import given
from hypothesis import strategies as st

from src.TAD.conjunto import (Conj, conjuntoAleatorio, elimina, esVacio,
                              inserta, menor, vacio)
from src.TAD_Particion_por_una_propiedad import particion


class Comparable(Protocol):
    @abstractmethod
    def __lt__(self: A, otro: A) -> bool:
        pass

A = TypeVar('A', bound=Comparable)

# 1ª solución
# ===========

def divide(x: A, c: Conj[A]) -> tuple[Conj[A], Conj[A]]:
    if esVacio(c):
        return (vacio(), vacio())
    mc = menor(c)
    rc = elimina(mc, c)
    (c1, c2) = divide(x, rc)
    if mc < x or mc == x:
        return (inserta(mc, c1), c2)
    return (c1, inserta(mc, c2))

# 2ª solución
# ===========

def divide2(x: A, c: Conj[A]) -> tuple[Conj[A], Conj[A]]:
```

```
    return particion(lambda y: y < x or y == x, c)

# La función particion está definida en el ejercicio
# "Partición de un conjunto según una propiedad" que se encuentra en
# https://bit.ly/3YCOah5

# 3ª solución
# ===========

def divide3Aux(x: A, c: Conj[A]) -> tuple[Conj[A], Conj[A]]:
    r: Conj[A] = vacio()
    s: Conj[A] = vacio()
    while not esVacio(c):
        mc = menor(c)
        c = elimina(mc, c)
        if mc < x or mc == x:
            r = inserta(mc, r)
        else:
            s = inserta(mc, s)
    return (r, s)

def divide3(x: A, c: Conj[A]) -> tuple[Conj[A], Conj[A]]:
    _c = deepcopy(c)
    return divide3Aux(x, _c)

# 4ª solución
# ===========

def divide4Aux(x: A, c: Conj[A]) -> tuple[Conj[A], Conj[A]]:
    r: Conj[A] = Conj()
    s: Conj[A] = Conj()
    while not c.esVacio():
        mc = c.menor()
        c.elimina(mc)
        if mc < x or mc == x:
            r.inserta(mc)
        else:
            s.inserta(mc)
    return (r, s)
```

```python
def divide4(x: A, c: Conj[A]) -> tuple[Conj[A], Conj[A]]:
    _c = deepcopy(c)
    return divide4Aux(x, _c)

# Comprobación de equivalencia de las definiciones
# ================================================

# La propiedad es
@given(x=st.integers(), c=conjuntoAleatorio())
def test_particion(x: int, c: Conj[int]) -> None:
    r = divide(x, c)
    assert divide2(x, c) == r
    assert divide3(x, c) == r
    assert divide4(x, c) == r

# La comprobación es
#    src> poetry run pytest -q TAD_Particion_segun_un_numero.py
#    1 passed in 0.30s
```

8.21. Aplicación de una función a los elementos de un conjunto

8.21.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Utilizando el tipo abstracto de datos de los conjuntos
-- (https://bit.ly/3HbB7fo) definir la función
--    mapC :: (Ord a, Ord b) => (a -> b) -> Conj a -> Conj b
-- tal que (map f c) es el conjunto formado por las imágenes de los
-- elementos de c, mediante f. Por ejemplo,
--    λ> mapC (*2) (inserta 3 (inserta 1 vacio))
--    {2, 6}
-- ---------------------------------------------------------------------

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

module TAD_mapC where

import TAD.Conjunto (Conj, vacio, inserta, esVacio, menor, elimina)
```

```haskell
import TAD_Transformaciones_conjuntos_listas (conjuntoAlista, listaAconjunto)
import Test.QuickCheck.HigherOrder

-- 1ª solución
-- ===========

mapC :: (Ord a, Ord b) => (a -> b) -> Conj a -> Conj b
mapC f c
  | esVacio c = vacio
  | otherwise = inserta (f mc) (mapC f rc)
  where mc = menor c
        rc = elimina mc c

-- 2ª solución
-- ===========

mapC2 :: (Ord a, Ord b) => (a -> b) -> Conj a -> Conj b
mapC2 f c = listaAconjunto (map f (conjuntoAlista c))

-- Las funciones conjuntoAlista y listaAconjunto está definida en el
-- ejercicio Transformaciones entre conjuntos y listas” que se encuentra
-- en https://bit.ly/3RexzxH

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

-- La propiedad es
prop_mapC :: (Int -> Int) -> [Int] -> Bool
prop_mapC f xs =
  mapC f c == mapC2 f c
  where c = listaAconjunto xs

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

8.21.2. En Python

```python
# ---------------------------------------------------------------------
```

```
# Utilizando el tipo abstracto de datos de los conjuntos
# (https://bit.ly/3HbB7fo) definir la función
#    mapC : (Callable[[A], B], Conj[A]) -> Conj[B]
# tal que map(f, c) es el conjunto formado por las imágenes de los
# elementos de c, mediante f. Por ejemplo,
#    >>> mapC(lambda x: 2 * x, inserta(3, inserta(1, vacio())))
#    {2, 6}
# ---------------------------------------------------------------------

from __future__ import annotations

from abc import abstractmethod
from copy import deepcopy
from typing import Callable, Protocol, TypeVar

from hypothesis import given

from src.TAD.conjunto import (Conj, conjuntoAleatorio, elimina, esVacio,
                              inserta, menor, vacio)
from src.TAD_Transformaciones_conjuntos_listas import (conjuntoAlista,
                                                       listaAconjunto)


class Comparable(Protocol):
    @abstractmethod
    def __lt__(self: A, otro: A) -> bool:
        pass

A = TypeVar('A', bound=Comparable)
B = TypeVar('B', bound=Comparable)

# 1ª solución
# ===========

def mapC(f: Callable[[A], B], c: Conj[A]) -> Conj[B]:
    if esVacio(c):
        return vacio()
    mc = menor(c)
    rc = elimina(mc, c)
    return inserta(f(mc), mapC(f, rc))
```

```python
# 2ª solución
# ===========

def mapC2(f: Callable[[A], B], c: Conj[A]) -> Conj[B]:
    return listaAconjunto(list(map(f, conjuntoAlista(c))))

# Las funciones conjuntoAlista y listaAconjunto está definida en el
# ejercicio Transformaciones entre conjuntos y listas” que se encuentra
# en https://bit.ly/3RexzxH

# 3ª solución
# ===========

def mapC3Aux(f: Callable[[A], B], c: Conj[A]) -> Conj[B]:
    r: Conj[B] = vacio()
    while not esVacio(c):
        mc = menor(c)
        c = elimina(mc, c)
        r = inserta(f(mc), r)
    return r

def mapC3(f: Callable[[A], B], c: Conj[A]) -> Conj[B]:
    _c = deepcopy(c)
    return mapC3Aux(f, _c)

# 4ª solución
# ===========

def mapC4Aux(f: Callable[[A], B], c: Conj[A]) -> Conj[B]:
    r: Conj[B] = Conj()
    while not c.esVacio():
        mc = c.menor()
        c.elimina(mc)
        r.inserta(f(mc))
    return r

def mapC4(f: Callable[[A], B], c: Conj[A]) -> Conj[B]:
    _c = deepcopy(c)
    return mapC4Aux(f, _c)
```

```
# Comprobación de equivalencia de las definiciones
# ================================================

# La propiedad es
@given(c=conjuntoAleatorio())
def test_mapPila(c: Conj[int]) -> None:
    r = mapC(lambda x: 2 * x, c)
    assert mapC2(lambda x: 2 * x, c) == r
    assert mapC3(lambda x: 2 * x, c) == r
    assert mapC4(lambda x: 2 * x, c) == r

# La comprobación es
#    src> poetry run pytest -q TAD_mapC.py
#    1 passed in 0.29s
```

8.22. Todos los elementos verifican una propiedad

8.22.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Utilizando el tipo abstracto de datos de los conjuntos
-- (https://bit.ly/3HbB7fo) definir la función
--    todos :: Ord a => (a -> Bool) -> Conj a -> Bool
-- tal que (todos p c) se verifica si todos los elemsntos de c
-- verifican el predicado p.  Por ejemplo,
--    todos even (inserta 4 (inserta 6 vacio))  ==  True
--    todos even (inserta 4 (inserta 7 vacio))  ==  False
-- ---------------------------------------------------------------------

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

module TAD_TodosVerificanConj where

import TAD.Conjunto (Conj, vacio, inserta, esVacio, menor, elimina)
import TAD_Transformaciones_conjuntos_listas (conjuntoAlista, listaAconjunto)
import Test.QuickCheck.HigherOrder
```

```haskell
-- 1ª solución
-- ===========

todos :: Ord a => (a -> Bool) -> Conj a -> Bool
todos p c
  | esVacio c = True
  | otherwise = p mc && todos p rc
  where mc = menor c
        rc = elimina mc c

-- 2ª solución
-- ===========

todos2 :: Ord a => (a -> Bool) -> Conj a -> Bool
todos2 p c = all p (conjuntoAlista c)

-- La función conjuntoAlista está definida en el ejercicio
-- "Transformaciones entre conjuntos y listas" que se encuentra
-- en https://bit.ly/3RexzxH

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

-- La propiedad es
prop_todos :: (Int -> Bool) -> [Int] -> Bool
prop_todos p xs =
  todos p c == todos2 p c
  where c = listaAconjunto xs

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

8.22.2. En Python

```python
# ---------------------------------------------------------------------
# Utilizando el tipo abstracto de datos de los conjuntos
# (https://bit.ly/3HbB7fo) definir la función
#    todos : (Callable[[A], bool], Conj[A]) -> bool
# tal que todos(p, c) se verifica si todos los elemsntos de c
```

```
# verifican el predicado p.  Por ejemplo,
#    >>> todos(lambda x: x % 2 == 0, inserta(4, inserta(6, vacio())))
#    True
#    >>> todos(lambda x: x % 2 == 0, inserta(4, inserta(7, vacio())))
#    False
# ---------------------------------------------------------------------

# pylint: disable=unused-import

from __future__ import annotations

from abc import abstractmethod
from copy import deepcopy
from typing import Callable, Protocol, TypeVar

from hypothesis import given

from src.TAD.conjunto import (Conj, conjuntoAleatorio, elimina, esVacio,
                              inserta, menor, vacio)
from src.TAD_Transformaciones_conjuntos_listas import conjuntoAlista


class Comparable(Protocol):
    @abstractmethod
    def __lt__(self: A, otro: A) -> bool:
        pass

A = TypeVar('A', bound=Comparable)

# 1ª solución
# ===========

def todos(p: Callable[[A], bool], c: Conj[A]) -> bool:
    if esVacio(c):
        return True
    mc = menor(c)
    rc = elimina(mc, c)
    return p(mc) and todos(p, rc)

# 2ª solución
```

```
# ===========

def todos2(p: Callable[[A], bool], c: Conj[A]) -> bool:
    return all(p(x) for x in conjuntoAlista(c))

# La función conjuntoAlista está definida en el ejercicio
# "Transformaciones entre conjuntos y listas" que se encuentra
# en https://bit.ly/3RexzxH

# 3ª solución
# ===========

def todos3Aux(p: Callable[[A], bool], c: Conj[A]) -> bool:
    while not esVacio(c):
        mc = menor(c)
        c = elimina(mc, c)
        if not p(mc):
            return False
    return True

def todos3(p: Callable[[A], bool], c: Conj[A]) -> bool:
    _c = deepcopy(c)
    return todos3Aux(p, _c)

# 4ª solución
# ===========

def todos4Aux(p: Callable[[A], bool], c: Conj[A]) -> bool:
    while not c.esVacio():
        mc = c.menor()
        c.elimina(mc)
        if not p(mc):
            return False
    return True

def todos4(p: Callable[[A], bool], c: Conj[A]) -> bool:
    _c = deepcopy(c)
    return todos4Aux(p, _c)

# Comprobación de equivalencia de las definiciones
```

```
# ================================================

# La propiedad es
@given(c=conjuntoAleatorio())
def test_todos(c: Conj[int]) -> None:
    r = todos(lambda x: x % 2 == 0, c)
    assert todos2(lambda x: x % 2 == 0, c) == r
    assert todos3(lambda x: x % 2 == 0, c) == r
    assert todos4(lambda x: x % 2 == 0, c) == r

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

8.23. Algunos elementos verifican una propiedad

8.23.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Utilizando el tipo abstracto de datos de los conjuntos
-- (https://bit.ly/3HbB7fo) definir la función
--    algunos :: Ord a => (a -> Bool) -> Conj a -> Bool
-- tal que (algunos p c) se verifica si algún elemento de c verifica el
-- predicado p. Por ejemplo,
--    algunos even (inserta 4 (inserta 7 vacio))  ==  True
--    algunos even (inserta 3 (inserta 7 vacio))  ==  False
-- ---------------------------------------------------------------------

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

module TAD_AlgunosVerificanConj where

import TAD.Conjunto (Conj, vacio, inserta, esVacio, menor, elimina)
import TAD_Transformaciones_conjuntos_listas (conjuntoAlista, listaAconjunto)
import Test.QuickCheck.HigherOrder

-- 1ª solución
-- ===========
```

```haskell
algunos :: Ord a => (a -> Bool) -> Conj a -> Bool
algunos p c
  | esVacio c = False
  | otherwise = p mc || algunos p rc
  where mc = menor c
        rc = elimina mc c

-- 2ª solución
-- ===========

algunos2 :: Ord a => (a -> Bool) -> Conj a -> Bool
algunos2 p c = any p (conjuntoAlista c)

-- La función conjuntoAlista está definida en el ejercicio
-- "Transformaciones entre conjuntos y listas" que se encuentra
-- en https://bit.ly/3RexzxH

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

-- La propiedad es
prop_algunos :: (Int -> Bool) -> [Int] -> Bool
prop_algunos p xs =
  algunos p c == algunos2 p c
  where c = listaAconjunto xs

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

8.23.2. En Python

```python
# ---------------------------------------------------------------------
# Utilizando el tipo abstracto de datos de los conjuntos
# (https://bit.ly/3HbB7fo) definir la función
#    algunos : algunos(Callable[[A], bool], Conj[A]) -> bool
# tal que algunos(p, c) se verifica si algún elemento de c verifica el
# predicado p. Por ejemplo,
#    >>> algunos(lambda x: x % 2 == 0, inserta(4, inserta(7, vacio())))
```

```python
#    True
#    >>> algunos(lambda x: x % 2 == 0, inserta(3, inserta(7, vacio())))
#    False
# ---------------------------------------------------------------------

# pylint: disable=unused-import

from __future__ import annotations

from abc import abstractmethod
from copy import deepcopy
from typing import Callable, Protocol, TypeVar

from hypothesis import given

from src.TAD.conjunto import (Conj, conjuntoAleatorio, elimina, esVacio,
                              inserta, menor, vacio)
from src.TAD_Transformaciones_conjuntos_listas import conjuntoAlista


class Comparable(Protocol):
    @abstractmethod
    def __lt__(self: A, otro: A) -> bool:
        pass

A = TypeVar('A', bound=Comparable)

# 1ª solución
# ===========

def algunos(p: Callable[[A], bool], c: Conj[A]) -> bool:
    if esVacio(c):
        return False
    mc = menor(c)
    rc = elimina(mc, c)
    return p(mc) or algunos(p, rc)

# 2ª solución
# ===========
```

```
def algunos2(p: Callable[[A], bool], c: Conj[A]) -> bool:
    return any(p(x) for x in conjuntoAlista(c))

# La función conjuntoAlista está definida en el ejercicio
# "Transformaciones entre conjuntos y listas" que se encuentra
# en https://bit.ly/3RexzxH

# 3ª solución
# ===========

def algunos3Aux(p: Callable[[A], bool], c: Conj[A]) -> bool:
    while not esVacio(c):
        mc = menor(c)
        c = elimina(mc, c)
        if p(mc):
            return True
    return False

def algunos3(p: Callable[[A], bool], c: Conj[A]) -> bool:
    _c = deepcopy(c)
    return algunos3Aux(p, _c)

# 4ª solución
# ===========

def algunos4Aux(p: Callable[[A], bool], c: Conj[A]) -> bool:
    while not c.esVacio():
        mc = c.menor()
        c.elimina(mc)
        if p(mc):
            return True
    return False

def algunos4(p: Callable[[A], bool], c: Conj[A]) -> bool:
    _c = deepcopy(c)
    return algunos4Aux(p, _c)

# Comprobación de equivalencia de las definiciones
# ================================================
```

```
# La propiedad es
@given(c=conjuntoAleatorio())
def test_algunos(c: Conj[int]) -> None:
    r = algunos(lambda x: x % 2 == 0, c)
    assert algunos2(lambda x: x % 2 == 0, c) == r
    assert algunos3(lambda x: x % 2 == 0, c) == r
    assert algunos4(lambda x: x % 2 == 0, c) == r

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

8.24. Producto cartesiano

8.24.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Utilizando el tipo abstracto de datos de los conjuntos
-- (https://bit.ly/3HbB7fo) definir la función
--    productoC :: (Ord a, Ord b) => Conj a -> Conj b -> Conj (a,b)
-- tal que (productoC c1 c2) es el producto cartesiano de los
-- conjuntos c1 y c2. Por ejemplo,
--    λ> ej1 = inserta 2 (inserta 5 vacio)
--    λ> ej2 = inserta 9 (inserta 4 (inserta 3 vacio))
--    λ> productoC ej1 ej2
--    {(2,3), (2,4), (2,9), (5,3), (5,4), (5,9)}
-- ---------------------------------------------------------------------

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

module TAD_Producto_cartesiano where

import TAD.Conjunto (Conj, vacio, inserta, esVacio, menor, elimina)
import TAD_Transformaciones_conjuntos_listas (conjuntoAlista, listaAconjunto)
import TAD_Union_de_dos_conjuntos (union)
import Test.QuickCheck

-- 1ª solución
-- ===========
```

```haskell
productoC :: (Ord a, Ord b) => Conj a -> Conj b -> Conj (a,b)
productoC c1 c2
  | esVacio c1 = vacio
  | otherwise  = agrega mc1 c2 `union` productoC rc1 c2
  where mc1 = menor c1
        rc1 = elimina mc1 c1

-- (agrega x c) es el conjunto de los pares de x con los elementos de
-- c. Por ejemplo,
--    λ> agrega 2 (inserta 9 (inserta 4 (inserta 3 vacio)))
--    {(2,3), (2,4), (2,9)}
agrega :: (Ord a, Ord b) => a -> Conj b -> Conj (a,b)
agrega x c
  | esVacio c = vacio
  | otherwise = inserta (x, mc) (agrega x rc)
  where mc = menor c
        rc = elimina mc c

-- La función union está definida en el ejercicio
-- "Unión de dos conjuntos" que se encuentra en
-- https://bit.ly/3Y1jBl8

-- 2ª solución
-- ===========

productoC2 :: (Ord a, Ord b) => Conj a -> Conj b -> Conj (a,b)
productoC2 c1 c2 =
  foldr inserta vacio [(x,y) | x <- xs, y <- ys]
  where xs = conjuntoAlista c1
        ys = conjuntoAlista c2

-- 3ª solución
-- ===========

productoC3 :: (Ord a, Ord b) => Conj a -> Conj b -> Conj (a,b)
productoC3 c1 c2 =
  listaAconjunto [(x,y) | x <- xs, y <- ys]
  where xs = conjuntoAlista c1
        ys = conjuntoAlista c2
```

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

-- La propiedad es
prop_productoC :: Conj Int -> Conj Int -> Bool
prop_productoC c1 c2 =
  all (== productoC c1 c2)
      [productoC2 c1 c2,
       productoC3 c1 c2]

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

8.24.2. En Python

```python
# ---------------------------------------------------------------------
# Utilizando el tipo abstracto de datos de los conjuntos
# (https://bit.ly/3HbB7fo) definir la función
#    productoC : (A, Conj[B]) -> Any
# tal que (productoC c1 c2) es el producto cartesiano de los
# conjuntos c1 y c2. Por ejemplo,
#    >>> ej1 = inserta(2, inserta(5, vacio()))
#    >>> ej2 = inserta(9, inserta(4, inserta(3, vacio())))
#    >>> productoC(ej1, ej2)
#    {(2, 3), (2, 4), (2, 9), (5, 3), (5, 4), (5, 9)}
# ---------------------------------------------------------------------

from __future__ import annotations

from abc import abstractmethod
from copy import deepcopy
from functools import reduce
from typing import Protocol, TypeVar

from hypothesis import given

from src.TAD.conjunto import (Conj, conjuntoAleatorio, elimina, esVacio,
                              inserta, menor, vacio)
from src.TAD_Transformaciones_conjuntos_listas import (conjuntoAlista,
```

```
                                                                   listaAconjunto)
from src.TAD_Union_de_dos_conjuntos import union


class Comparable(Protocol):
    @abstractmethod
    def __lt__(self: A, otro: A) -> bool:
        pass

A = TypeVar('A', bound=Comparable)
B = TypeVar('B', bound=Comparable)

# 1ª solución
# ===========

# (agrega x c) es el conjunto de los pares de x con los elementos de
# c. Por ejemplo,
#    >>> agrega(2, inserta(9, inserta(4, inserta(3, vacio()))))
#    {(2, 3), (2, 4), (2, 9)}
def agrega(x: A, c: Conj[B]) -> Conj[tuple[A, B]]:
    if esVacio(c):
        return vacio()
    mc = menor(c)
    rc = elimina(mc, c)
    return inserta((x, mc), agrega(x, rc))

def productoC(c1: Conj[A], c2: Conj[B]) -> Conj[tuple[A, B]]:
    if esVacio(c1):
        return vacio()
    mc1 = menor(c1)
    rc1 = elimina(mc1, c1)
    return union(agrega(mc1, c2), productoC(rc1, c2))

# La función union está definida en el ejercicio
# "Unión de dos conjuntos" que se encuentra en
# https://bit.ly/3Y1jBl8

# 2ª solución
# ===========
```

```python
def productoC2(c1: Conj[A], c2: Conj[B]) -> Conj[tuple[A, B]]:
    xs = conjuntoAlista(c1)
    ys = conjuntoAlista(c2)
    return reduce(lambda bs, a: inserta(a, bs), [(x,y) for x in xs for y in ys],

# 3ª solución
# ===========

def productoC3(c1: Conj[A], c2: Conj[B]) -> Conj[tuple[A, B]]:
    xs = conjuntoAlista(c1)
    ys = conjuntoAlista(c2)
    return listaAconjunto([(x,y) for x in xs for y in ys])

# 4ª solución
# ===========

def agrega4Aux(x: A, c: Conj[B]) -> Conj[tuple[A, B]]:
    r: Conj[tuple[A, B]] = vacio()
    while not esVacio(c):
        mc = menor(c)
        c = elimina(mc, c)
        r = inserta((x, mc), r)
    return r

def agrega4(x: A, c: Conj[B]) -> Conj[tuple[A, B]]:
    _c = deepcopy(c)
    return agrega4Aux(x, _c)

def productoC4(c1: Conj[A], c2: Conj[B]) -> Conj[tuple[A, B]]:
    r: Conj[tuple[A, B]] = vacio()
    while not esVacio(c1):
        mc1 = menor(c1)
        c1 = elimina(mc1, c1)
        r = union(agrega4(mc1, c2), r)
    return r

# 5ª solución
# ===========

def agrega5Aux(x: A, c: Conj[B]) -> Conj[tuple[A, B]]:
```

```
    r: Conj[tuple[A, B]] = Conj()
    while not c.esVacio():
        mc = c.menor()
        c.elimina(mc)
        r.inserta((x, mc))
    return r

def agrega5(x: A, c: Conj[B]) -> Conj[tuple[A, B]]:
    _c = deepcopy(c)
    return agrega5Aux(x, _c)

def productoC5(c1: Conj[A], c2: Conj[B]) -> Conj[tuple[A, B]]:
    r: Conj[tuple[A, B]] = Conj()
    while not c1.esVacio():
        mc1 = c1.menor()
        c1.elimina(mc1)
        r = union(agrega5(mc1, c2), r)
    return r

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

# La propiedad es
@given(c1=conjuntoAleatorio(), c2=conjuntoAleatorio())
def test_productoC(c1: Conj[int], c2: Conj[int]) -> None:
    r = productoC(c1, c2)
    assert productoC2(c1, c2) == r
    assert productoC3(c1, c2) == r
    assert productoC4(c1, c2) == r

# La comprobación de las propiedades es
#    > poetry run pytest -q TAD_Producto_cartesiano.py
#    1 passed in 0.35s
```

Capítulo 9

Relaciones binarias

Contenido

9.1. El tipo de las relaciones binarias

9.1.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Una relación binaria R sobre un conjunto A se puede representar
-- mediante un par (u,g) donde u es la lista de los elementos de tipo A
-- (el universo de R) y g es la lista de pares de elementos de u (el
-- grafo de R).
--
-- Definir el tipo de dato (Rel a), para representar las relaciones
-- binarias sobre a, y la función
--    esRelacionBinaria :: Eq a => Rel a -> Bool
-- tal que (esRelacionBinaria r) se verifica si r es una relación
-- binaria. Por ejemplo,
```

```
--    λ> esRelacionBinaria (R ([1, 3], [(3, 1), (3, 3)]))
--    True
--    λ> esRelacionBinaria (R ([1, 3], [(3, 1), (3, 2)]))
--    False
-- Además, definir un generador de relaciones binarias y comprobar que
-- las relaciones que genera son relaciones binarias.
-- ---------------------------------------------------------------------

module Relaciones_binarias where

import Data.List (nub)
import Test.QuickCheck

newtype Rel a = R ([a], [(a,a)])
  deriving (Eq, Show)

-- 1ª solución
-- ===========

esRelacionBinaria :: Eq a => Rel a -> Bool
esRelacionBinaria (R (u, g)) =
  and [x `elem` u && y `elem` u | (x,y) <- g]

-- 2ª solución
-- ===========

esRelacionBinaria2 :: Eq a => Rel a -> Bool
esRelacionBinaria2 (R (u, g)) =
  all (\(x,y) -> x `elem` u && y `elem` u) g

-- 3ª solución
-- ===========

esRelacionBinaria3 :: Eq a => Rel a -> Bool
esRelacionBinaria3 (R (_, []))         = True
esRelacionBinaria3 (R (u, (x,y):g)) =
  x `elem` u &&
  y `elem` u &&
  esRelacionBinaria3 (R (u, g))
```

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

-- Generador de relaciones binarias. Por ejemplo,
--    λ> sample (relacionArbitraria :: Gen (Rel Int))
--    R ([0],[])
--    R ([0,-1,1],[(0,-1),(0,1),(-1,1),(1,0)])
--    R ([1],[])
--    R ([-5,3],[(-5,-5),(-5,3),(3,-5),(3,3)])
--    R ([-2,-7],[(-7,-7)])
--    R ([11,-7],[])
--    R ([0],[])
--    R ([-13,-11],[(-13,-13)])
relacionArbitraria :: (Arbitrary a, Eq a) => Gen (Rel a)
relacionArbitraria = do
  n <- choose (0, 10)
  u1 <- vectorOf n arbitrary
  let u = nub u1
  g <- sublistOf [(x,y) | x <- u, y <- u]
  return (R (u, g))

-- Relaciones es una subclase de Arbitrary.
instance (Arbitrary a, Eq a) => Arbitrary (Rel a) where
  arbitrary = relacionArbitraria

-- La propiedad es
prop_esRelacionBinaria :: Rel Int -> Bool
prop_esRelacionBinaria r =
  esRelacionBinaria r  &&
  esRelacionBinaria2 r &&
  esRelacionBinaria3 r

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

9.1.2. En Python

```python
# ---------------------------------------------------------------------
# Una relación binaria R sobre un conjunto A se puede representar
```

```
# mediante un par (u,g) donde u es la lista de los elementos de tipo A
# (el universo de R) y g es la lista de pares de elementos de u (el
# grafo de R).
#
# Definir el tipo de dato (Rel a), para representar las relaciones
# binarias sobre a, y la función
#    esRelacionBinaria : (Rel[A]) -> bool
# tal que esRelacionBinaria(r) se verifica si r es una relación
# binaria. Por ejemplo,
#    >>> esRelacionBinaria(([1, 3], [(3, 1), (3, 3)]))
#    True
#    >>> esRelacionBinaria(([1, 3], [(3, 1), (3, 2)]))
#    False
# Además, definir un generador de relaciones binarias y comprobar que
# las relaciones que genera son relaciones binarias.
# ---------------------------------------------------------------------

from random import randint, sample
from typing import TypeVar

from hypothesis import given
from hypothesis import strategies as st

A = TypeVar('A')

Rel = tuple[list[A], list[tuple[A, A]]]

# 1ª solución
# ===========

def esRelacionBinaria(r: Rel[A]) -> bool:
    (u, g) = r
    return all((x in u and y in u for (x, y) in g))

# 2ª solución
# ===========

def esRelacionBinaria2(r: Rel[A]) -> bool:
    (u, g) = r
    if not g:
```

```
        return True
    (x, y) = g[0]
    return x in u and y in u and esRelacionBinaria2((u, g[1:]))

# 3ª solución
# ===========

def esRelacionBinaria3(r: Rel[A]) -> bool:
    (u, g) = r
    for (x, y) in g:
        if x not in u or y not in u:
            return False
    return True

# Generador de relaciones binarias
# ================================

# conjuntoArbitrario(n) es un conjunto arbitrario cuyos elementos están
# entre 0 y n-1. Por ejemplo,
#    >>> conjuntoArbitrario(10)
#    [8, 9, 4, 5]
#    >>> conjuntoArbitrario(10)
#    [1, 2, 3, 4, 5, 6, 7, 8, 9]
#    >>> conjuntoArbitrario(10)
#    [0, 1, 2, 3, 6, 7, 9]
#    >>> conjuntoArbitrario(10)
#    [8, 2, 3, 7]
def conjuntoArbitrario(n: int) -> list[int]:
    xs = sample(range(n), randint(0, n))
    return list(set(xs))

# productoCartesiano(xs, ys) es el producto cartesiano de xs e ys. Por
# ejemplo,
#    >>> productoCartesiano([2, 3], [1, 7, 5])
#    [(2, 1), (2, 7), (2, 5), (3, 1), (3, 7), (3, 5)]
def productoCartesiano(xs: list[int], ys: list[int]) -> list[tuple[int, int]]:
    return [(x, y) for x in xs for y in ys]

# sublistaArbitraria(xs) es una sublista arbitraria de xs. Por ejemplo,
#    >>> sublistaArbitraria(range(10))
```

```
#    [3, 7]
#    >>> sublistaArbitraria(range(10))
#    []
#    >>> sublistaArbitraria(range(10))
#    [4, 1, 0, 9, 8, 7, 5, 6, 2, 3]
def sublistaArbitraria(xs: list[A]) -> list[A]:
    n = len(xs)
    k = randint(0, n)
    return sample(xs, k)

# relacionArbitraria(n) es una relación arbitraria tal que los elementos
# de su universo están entre 0 y n-1. Por ejemplo,
#    >>> relacionArbitraria(3)
#    ([0, 1], [(1, 0), (1, 1)])
#    >>> relacionArbitraria(3)
#    ([], [])
#    >>> relacionArbitraria(5)
#    ([0, 2, 3, 4], [(2, 0), (3, 3), (2, 3), (4, 0), (3, 4), (4, 2)])
def relacionArbitraria(n: int) -> Rel[int]:
    u = conjuntoArbitrario(n)
    g = sublistaArbitraria(productoCartesiano(u, u))
    return (u, g)

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

# La propiedad es
@given(st.integers(min_value=0, max_value=10))
def test_esRelacionBinaria(n: int) -> None:
    r = relacionArbitraria(n)
    assert esRelacionBinaria(r)
    assert esRelacionBinaria2(r)
    assert esRelacionBinaria3(r)

# La comprobación es
#    > poetry run pytest -q Relaciones_binarias.py
#    1 passed in 0.14s
```

9.2. Universo y grafo de una relación binaria

9.2.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Usando el [tipo de las relaciones binarias](https://bit.ly/3IVVqOT),
-- definir las siguientes funciones
--    universo :: Eq a => Rel a -> [a]
--    grafo    :: Eq a => ([a],[(a,a)]) -> [(a,a)]
-- tales que
-- + (universo r) es el universo de la relación r. Por ejemplo,
--      λ> r = R ([1, 3],[(3, 1), (3, 3)])
--      λ> universo r
--      [1,3]
-- + (grafo r) es el grafo de la relación r. Por ejemplo,
--      λ> grafo r
--      [(3,1),(3,3)]
-- ---------------------------------------------------------------------

module Universo_y_grafo_de_una_relacion_binaria where

import Relaciones_binarias (Rel(R))

universo :: Eq a => Rel a -> [a]
universo (R (u,_)) = u

grafo :: Eq a => Rel a -> [(a,a)]
grafo (R (_,g)) = g
```

9.2.2. En Python

```python
# ---------------------------------------------------------------------
# Usando el [tipo de las relaciones binarias](https://bit.ly/3IVVqOT),
# definir las siguientes funciones
#    universo : (Rel[A]) -> list[A]
#    grafo    : (Rel[A]) -> list[tuple[A, A]]
# tales que
# + universo(r) es el universo de la relación r. Por ejemplo,
#      >>> r = (list(range(1, 10)), [(1, 3), (2, 6), (8, 9), (2, 7)])
#      >>> universo(r)
```

```
#      [1, 2, 3, 4, 5, 6, 7, 8, 9]
# + grafo(r) es el grafo de la relación r. Por ejemplo,
#      >>> grafo(r)
#      [(1, 3), (2, 6), (8, 9), (2, 7)]
# ---------------------------------------------------------------------

from typing import TypeVar

A = TypeVar('A')

Rel = tuple[list[A], list[tuple[A, A]]]

def universo(r: Rel[A]) -> list[A]:
    return r[0]

def grafo(r: Rel[A]) -> list[tuple[A, A]]:
    return r[1]
```

9.3. Relaciones reflexivas

9.3.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Usando el [tipo de las relaciones binarias](https://bit.ly/3IVVqOT),
-- definir la función
--    reflexiva :: Eq a => Rel a -> Bool
-- tal que (reflexiva r) se verifica si la relación r es reflexiva. Por
-- ejemplo,
--    reflexiva (R ([1,3],[(1,1),(1,3),(3,3)]))    ==  True
--    reflexiva (R ([1,2,3],[(1,1),(1,3),(3,3)]))  ==  False
-- ---------------------------------------------------------------------

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

module Relaciones_reflexivas where

import Relaciones_binarias (Rel(R))
import Test.QuickCheck

-- 1ª solución
```

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

reflexiva :: Eq a => Rel a -> Bool
reflexiva (R ([], _))    = True
reflexiva (R (x:xs, ps)) = (x, x) `elem` ps && reflexiva  (R (xs, ps))

-- 2ª solución
-- ===========

reflexiva2 :: Eq a => Rel a -> Bool
reflexiva2 (R (us,ps)) = and [(x,x) `elem` ps | x <- us]

-- 3ª solución
-- ===========

reflexiva3 :: Eq a => Rel a -> Bool
reflexiva3 (R (us,ps)) = all (`elem` ps) [(x,x) | x <- us]

-- 4ª solución
-- ===========

reflexiva4 :: Eq a => Rel a -> Bool
reflexiva4 (R (us,ps)) = all (\x -> (x,x) `elem` ps) us

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

-- La propiedad es
prop_reflexiva :: Rel Int -> Bool
prop_reflexiva r =
  all (== reflexiva r)
      [reflexiva2 r,
       reflexiva3 r,
       reflexiva4 r]

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

9.3.2. En Python

```
# ---------------------------------------------------------------------
# Usando el [tipo de las relaciones binarias](https://bit.ly/3IVVqOT),
# definir la función
#    reflexiva : (Rel) -> bool
# tal que reflexiva(r) se verifica si la relación r es reflexiva. Por
# ejemplo,
#    >>> reflexiva(([1, 3], [(1, 1),(1, 3),(3, 3)]))
#    True
#    >>> reflexiva(([1, 2, 3], [(1, 1),(1, 3),(3, 3)]))
#    False
# ---------------------------------------------------------------------

from typing import TypeVar

from hypothesis import given
from hypothesis import strategies as st

from src.Relaciones_binarias import Rel, relacionArbitraria

A = TypeVar('A')

# 1ª solución
# ===========

def reflexiva(r: Rel[A]) -> bool:
    (us, ps) = r
    if not us:
        return True
    return (us[0], us[0]) in ps and reflexiva((us[1:], ps))

# 2ª solución
# ===========

def reflexiva2(r: Rel[A]) -> bool:
    (us, ps) = r
    return all(((x,x) in ps for x in us))

# 3ª solución
# ===========
```

```python
def reflexiva3(r: Rel[A]) -> bool:
    (us, ps) = r
    for x in us:
        if (x, x) not in ps:
            return False
    return True

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

# La propiedad es
@given(st.integers(min_value=0, max_value=10))
def test_reflexiva(n: int) -> None:
    r = relacionArbitraria(n)
    res = reflexiva(r)
    assert reflexiva2(r) == res
    assert reflexiva3(r) == res

# La comprobación es
#    > poetry run pytest -q Relaciones_reflexivas.py
#    1 passed in 0.41s
```

9.4. Relaciones simétricas

9.4.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Usando el [tipo de las relaciones binarias](https://bit.ly/3IVVqOT),
-- definir la función
--    simetrica :: Eq a => Rel a -> Bool
-- tal que (simetrica r) se verifica si la relación r es simétrica. Por
-- ejemplo,
--    simetrica (R ([1,3],[(1,1),(1,3),(3,1)]))  ==  True
--    simetrica (R ([1,3],[(1,1),(1,3),(3,2)]))  ==  False
--    simetrica (R ([1,3],[]))                   ==  True
-- ---------------------------------------------------------------------

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

```haskell
module Relaciones_simetricas where

import Relaciones_binarias (Rel(R))
import Test.QuickCheck

-- 1ª solución
-- ===========

simetrica :: Eq a => Rel a -> Bool
simetrica (R (_,g)) = and [(y,x) `elem` g | (x,y) <- g]

-- 2ª solución
-- ===========

simetrica2 :: Eq a => Rel a -> Bool
simetrica2 (R (_,g)) = all (\(x,y) -> (y,x) `elem` g) g

-- 3ª solución
-- ===========

simetrica3 :: Eq a => Rel a -> Bool
simetrica3 (R (_,g)) = aux g
  where aux [] = True
        aux ((x,y):ps) = (y,x) `elem` g && aux ps

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

-- La propiedad es
prop_simetrica :: Rel Int -> Bool
prop_simetrica r =
  all (== simetrica r)
      [simetrica2 r,
       simetrica3 r]

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

9.4.2. En Python

```python
# ---------------------------------------------------------------------
# Usando el [tipo de las relaciones binarias](https://bit.ly/3IVVqOT),
# definir la función
#    simetrica : (Rel[A]) -> bool
# tal que simetrica(r) se verifica si la relación r es simétrica. Por
# ejemplo,
#    >>> simetrica(([1, 3], [(1, 1), (1, 3), (3, 1)]))
#    True
#    >>> simetrica(([1, 3], [(1, 1), (1, 3), (3, 2)]))
#    False
#    >>> simetrica(([1, 3], []))
#    True
# ---------------------------------------------------------------------

from typing import TypeVar

from hypothesis import given
from hypothesis import strategies as st

from src.Relaciones_binarias import Rel, relacionArbitraria

A = TypeVar('A')

# 1ª solución
# ===========

def simetrica(r: Rel[A]) -> bool:
    (_, g) = r
    return all(((y, x) in g for (x, y) in g))

# 2ª solución
# ===========

def simetrica2(r: Rel[A]) -> bool:
    (_, g) = r
    def aux(ps: list[tuple[A, A]]) -> bool:
        if not ps:
            return True
        (x, y) = ps[0]
```

```
        return (y, x) in g and aux(ps[1:])

    return aux(g)

# 3ª solución
# ===========

def simetrica3(r: Rel[A]) -> bool:
    (_, g) = r
    for (x, y) in g:
        if (y, x) not in g:
            return False
    return True

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

# La propiedad es
@given(st.integers(min_value=0, max_value=10))
def test_simetrica(n: int) -> None:
    r = relacionArbitraria(n)
    res = simetrica(r)
    assert simetrica2(r) == res
    assert simetrica3(r) == res

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

9.5. Composición de relaciones binarias

9.5.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Usando el [tipo de las relaciones binarias](https://bit.ly/3IVVqOT),
-- definir la función
--    composicion :: Eq a => Rel a -> Rel a -> Rel a
-- tal que (composicion r s) es la composición de las relaciones r y
-- s. Por ejemplo,
--    λ> composicion (R ([1,2],[(1,2),(2,2)])) (R ([1,2],[(2,1)]))
```

```haskell
--    R ([1,2],[(1,1),(2,1)])
-- ---------------------------------------------------------------------

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

module Composicion_de_relaciones_binarias_v2 where

import Relaciones_binarias (Rel(R))
import Test.QuickCheck

-- 1ª solución
-- ===========

composicion :: Eq a => Rel a -> Rel a -> Rel a
composicion (R (u1,g1)) (R (_,g2)) =
  R (u1,[(x,z) | (x,y) <- g1, (y',z) <- g2, y == y'])

-- 2ª solución
-- ===========

composicion2 :: Eq a => Rel a -> Rel a -> Rel a
composicion2 (R (u1,g1)) (R (_,g2)) =
  R (u1, aux g1)
  where aux [] = []
        aux ((x,y):g1') = [(x,z) | (y',z) <- g2, y == y'] ++ aux g1'

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

-- La propiedad es
prop_composicion :: Rel Int -> Rel Int -> Bool
prop_composicion r1 r2 =
  composicion r1 r2 == composicion2 r1 r2

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

9.5.2. En Python

```
# ---------------------------------------------------------------------
# Usando el [tipo de las relaciones binarias](https://bit.ly/3IVVqOT),
# definir la función
#    composicion : (Rel[A], Rel[A]) -> Rel[A]
# tal que composicion(r, s) es la composición de las relaciones r y
# s. Por ejemplo,
#    >>> composicion(([1,2],[(1,2),(2,2)]), ([1,2],[(2,1)]))
#    ([1, 2], [(1, 1), (2, 1)])
# ---------------------------------------------------------------------

from typing import TypeVar

from hypothesis import given
from hypothesis import strategies as st

from src.Relaciones_binarias import Rel, relacionArbitraria

A = TypeVar('A')

# 1ª solución
# ===========

def composicion(r1: Rel[A], r2: Rel[A]) -> Rel[A]:
    (u1, g1) = r1
    (_,  g2) = r2
    return (u1, [(x, z) for (x, y) in g1 for (u, z) in g2 if y == u])

# 2ª solución
# ===========

def composicion2(r1: Rel[A], r2: Rel[A]) -> Rel[A]:
    (u1, g1) = r1
    (_,  g2) = r2
    def aux(g: list[tuple[A, A]]) -> list[tuple[A, A]]:
        if not g:
            return []
        (x, y) = g[0]
        return [(x, z) for (u, z) in g2 if y == u] + aux(g[1:])
```

```
    return (u1, aux(g1))

# 2ª solución
# ===========

def composicion3(r1: Rel[A], r2: Rel[A]) -> Rel[A]:
    (u1, g1) = r1
    (_,  g2) = r2
    r: list[tuple[A, A]] = []
    for (x, y) in g1:
        r = r + [(x, z) for (u, z) in g2 if y == u]
    return (u1, r)

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

# La propiedad es
@given(st.integers(min_value=0, max_value=10),
       st.integers(min_value=0, max_value=10))
def test_simetrica(n: int, m: int) -> None:
    r1 = relacionArbitraria(n)
    r2 = relacionArbitraria(m)
    res = composicion(r1, r2)
    assert composicion2(r1, r2) == res
    assert composicion2(r1, r2) == res

# La comprobación es
#    > poetry run pytest -q Composicion_de_relaciones_binarias_v2.py
#    1 passed in 0.19s
```

9.6. Reconocimiento de subconjunto

9.6.1. En 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
```

```haskell
--    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 lo sucesivo, usaremos la 4ª definición
subconjunto :: Ord a => [a] -> [a] -> Bool
subconjunto = subconjunto4
```

9.6.2. 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 not xs:
        return True
    return xs[0] in ys and subconjunto2(xs[1:], ys)

# 3ª solución
# ===========

def subconjunto3(xs: list[A], ys: list[A]) -> bool:
    return all(elem in ys for elem 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_filtraAplica(xs: list[int], ys: list[int]) -> None:
    r = subconjunto1(xs, ys)
    assert subconjunto2(xs, ys) == r
    assert subconjunto3(xs, ys) == r
    assert subconjunto4(xs, ys) == r

# La comprobación es
#    src> poetry run pytest -q Reconocimiento_de_subconjunto.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
#    >>> xs = list(range(2*10**4))
#    >>> tiempo("subconjunto1(xs, xs)")
#    1.15 segundos
#    >>> tiempo("subconjunto2(xs, xs)")
#    2.27 segundos
#    >>> tiempo("subconjunto3(xs, xs)")
#    1.14 segundos
#    >>> tiempo("subconjunto4(xs, xs)")
#    0.00 segundos

# En lo sucesivo usaremos la cuarta definición
subconjunto = subconjunto4
```

9.7. Relaciones transitivas

9.7.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Usando el [tipo de las relaciones binarias](https://bit.ly/3IVVqOT),
-- definir la función
--    transitiva :: Ord a => Rel a -> Bool
-- tal que (transitiva r) se verifica si la relación r es transitiva.
-- Por ejemplo,
--    transitiva (R ([1,3,5],[(1,1),(1,3),(3,1),(3,3),(5,5)])) == True
--    transitiva (R ([1,3,5],[(1,1),(1,3),(3,1),(5,5)]))       == False
-- ---------------------------------------------------------------------

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

module Relaciones_transitivas where

import Relaciones_binarias (Rel(R))
import Reconocimiento_de_subconjunto (subconjunto)
import Universo_y_grafo_de_una_relacion_binaria (grafo)
import Composicion_de_relaciones_binarias_v2 (composicion)
import Test.QuickCheck

-- 1ª solución
-- ===========

transitiva1 :: Ord a => Rel a -> Bool
transitiva1 r@(R (_,g)) = subconjunto (grafo (composicion r r)) g

-- La función subconjunto está definida en el ejercicio
-- "Reconocimiento de subconjunto" que se encuentra en
-- https://bit.ly/427Tyeq
--
-- La función grafo está definida en el ejercicio
-- "Universo y grafo de una relación binaria" que se encuentra en
-- https://bit.ly/3J35mpC
--
-- La función composición está definida en el ejercicio
-- "Composición de relaciones binarias" que se encuentra en
-- https://bit.ly/3JyJrs7
```

```
-- 2ª solución
-- ===========

transitiva2 :: Ord a => Rel a -> Bool
transitiva2 (R (_,g)) = aux g
  where
    aux [] = True
    aux ((x,y):g') = and [(x,z) `elem` g | (u,z) <- g, u == y] && aux g'

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

-- La propiedad es
prop_transitiva :: Rel Int -> Bool
prop_transitiva r =
  transitiva1 r == transitiva2 r

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

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

-- La comparación es
--    λ> transitiva1 (R ([1..4001],[(x,x+1) | x <- [1..4000]]))
--    False
--    (3.15 secs, 898,932,776 bytes)
--    λ> transitiva2 (R ([1..4001],[(x,x+1) | x <- [1..4000]]))
--    False
--    (0.01 secs, 1,396,720 bytes)
--
--    λ> transitiva1 (R ([1..60], [(x,y) | x <- [1..60], y <- [1..60]]))
--    True
--    (2.71 secs, 852,578,456 bytes)
--    λ> transitiva2 (R ([1..60], [(x,y) | x <- [1..60], y <- [1..60]]))
--    True
--    (9.13 secs, 777,080,288 bytes)
```

```haskell
-- En lo sucesivo, usaremos la 1ª definición
transitiva :: Ord a => Rel a -> Bool
transitiva = transitiva1
```

9.7.2. En Python

```python
# ---------------------------------------------------------------------
# Usando el [tipo de las relaciones binarias](https://bit.ly/3IVVqOT),
# definir la función
#    transitiva : (Rel[A]) -> bool
# tal que transitiva(r) se verifica si la relación r es transitiva.
# Por ejemplo,
#    >>> transitiva(([1, 3, 5], [(1, 1), (1, 3), (3, 1), (3, 3), (5, 5)]))
#    True
#    >>> transitiva(([1, 3, 5], [(1, 1), (1, 3), (3, 1), (5, 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

from src.Composicion_de_relaciones_binarias_v2 import composicion
from src.Reconocimiento_de_subconjunto import subconjunto
from src.Relaciones_binarias import Rel, relacionArbitraria
from src.Universo_y_grafo_de_una_relacion_binaria import grafo

setrecursionlimit(10**6)

A = TypeVar('A')

# 1ª solución
# ===========

def transitiva1(r: Rel[A]) -> bool:
    g = grafo(r)
    return subconjunto(grafo(composicion(r, r)), g)
```

```
# La función subconjunto está definida en el ejercicio
# "Reconocimiento de subconjunto" que se encuentra en
# https://bit.ly/427Tyeq
#
# La función grafo está definida en el ejercicio
# "Universo y grafo de una relación binaria" que se encuentra en
# https://bit.ly/3J35mpC
#
# La función composición está definida en el ejercicio
# "Composición de relaciones binarias" que se encuentra en
# https://bit.ly/3JyJrs7

# 2ª solución
# ===========

def transitiva2(r: Rel[A]) -> bool:
    g = grafo(r)
    def aux(g1: list[tuple[A,A]]) -> bool:
        if not g1:
            return True
        (x, y) = g1[0]
        return all(((x, z) in g for (u,z) in g if u == y)) and aux(g1[1:])

    return aux(g)

# 3ª solución
# ===========

def transitiva3(r: Rel[A]) -> bool:
    g = grafo(r)
    g1 = list(g)
    for (x, y) in g1:
        if not all(((x, z) in g for (u,z) in g if u == y)):
            return False
    return True


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

```
# La propiedad es
@given(st.integers(min_value=0, max_value=10))
def test_simetrica(n: int) -> None:
    r = relacionArbitraria(n)
    res = transitiva1(r)
    assert transitiva2(r) == res
    assert transitiva3(r) == res

# La comprobación es
#    > poetry run pytest -q Relaciones_transitivas.py
#    1 passed in 0.12s

# 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
#    >>> u1 = range(6001)
#    >>> g1 = [(x, x+1) for x in range(6000)]
#    >>> tiempo("transitiva1((u1, g1))")
#    1.04 segundos
#    >>> tiempo("transitiva2((u1, g1))")
#    0.00 segundos
#    >>> tiempo("transitiva3((u1, g1))")
#    0.00 segundos
#
#    >>> u2 = range(60)
#    >>> g2 = [(x, y) for x in u2 for y in u2]
#    >>> tiempo("transitiva1((u2, g2))")
#    0.42 segundos
#    >>> tiempo("transitiva2((u2, g2))")
#    5.24 segundos
#    >>> tiempo("transitiva3((u2, g2))")
#    4.83 segundos
```

```
# En lo sucesivo usaremos la 1ª definición
transitiva = transitiva1
```

9.8. Relaciones irreflexivas

9.8.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Usando el [tipo de las relaciones binarias](https://bit.ly/3IVVqOT),
-- definir la función
--    irreflexiva :: Eq a => Rel a -> Bool
-- tal que (irreflexiva r) se verifica si la relación r es irreflexiva;
-- es decir, si ningún elemento de su universo está relacionado con
-- él mismo. Por ejemplo,
--    irreflexiva (R ([1,2,3],[(1,2),(2,1),(2,3)]))  ==  True
--    irreflexiva (R ([1,2,3],[(1,2),(2,1),(3,3)]))  ==  False
-- ---------------------------------------------------------------------

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

module Relaciones_irreflexivas where

import Relaciones_binarias (Rel(R))
import Test.QuickCheck

-- 1ª solución
-- ===========

irreflexiva :: Eq a => Rel a -> Bool
irreflexiva (R (u,g)) = and [(x,x) `notElem` g | x <- u]

-- 2ª solución
-- ===========

irreflexiva2 :: Eq a => Rel a -> Bool
irreflexiva2 (R(u,g)) = all (\x -> (x,x) `notElem` g) u

-- 3ª solución
-- ===========
```

```haskell
irreflexiva3 :: Eq a => Rel a -> Bool
irreflexiva3 (R(u,g)) = aux u
  where aux []     = True
        aux (x:xs) = (x,x) `notElem` g && aux xs


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

-- La propiedad es
prop_irreflexiva :: Rel Int -> Bool
prop_irreflexiva r =
  all (== irreflexiva r)
      [irreflexiva2 r,
       irreflexiva3 r]

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

9.8.2. En Python

```python
# ---------------------------------------------------------------------
# Usando el [tipo de las relaciones binarias](https://bit.ly/3IVVqOT),
# definir la función
#    irreflexiva : (Rel[A]) -> bool
# tal que irreflexiva(r) se verifica si la relación r es irreflexiva;
# es decir, si ningún elemento de su universo está relacionado con
# él mismo. Por ejemplo,
#    irreflexiva(([1, 2, 3], [(1, 2), (2, 1), (2, 3)]))  ==  True
#    irreflexiva(([1, 2, 3], [(1, 2), (2, 1), (3, 3)]))  ==  False
# ---------------------------------------------------------------------

from typing import TypeVar

from hypothesis import given
from hypothesis import strategies as st

from src.Relaciones_binarias import Rel, relacionArbitraria
```

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

# 1ª solución
# ===========

def irreflexiva(r: Rel[A]) -> bool:
    (u, g) = r
    return all(((x, x) not in g for x in u))

# 2ª solución
# ===========

def irreflexiva2(r: Rel[A]) -> bool:
    (u, g) = r
    def aux(xs: list[A]) -> bool:
        if not xs:
            return True
        return (xs[0], xs[0]) not in g and aux(xs[1:])

    return aux(u)

# 3ª solución
# ===========

def irreflexiva3(r: Rel[A]) -> bool:
    (u, g) = r
    for x in u:
        if (x, x) in g:
            return False
    return True

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

# La propiedad es
@given(st.integers(min_value=0, max_value=10))
def test_irreflexiva(n: int) -> None:
    r = relacionArbitraria(n)
    res = irreflexiva(r)
    assert irreflexiva2(r) == res
```

```
    assert irreflexiva3(r) == res

# La comprobación es
#    > poetry run pytest -q Relaciones_irreflexivas.py
#    1 passed in 0.12s
```

9.9. Relaciones antisimétricas

9.9.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Usando el [tipo de las relaciones binarias](https://bit.ly/3IVVqOT),
-- definir la función
--    antisimetrica :: Eq a => Rel a -> Bool
-- tal que (antisimetrica r) se verifica si la relación r es
-- antisimétrica; es decir, si (x,y) e (y,x) están relacionado, entonces
-- x=y. Por ejemplo,
--    antisimetrica (R ([1,2],[(1,2)]))        ==  True
--    antisimetrica (R ([1,2],[(1,2),(2,1)]))  ==  False
--    antisimetrica (R ([1,2],[(1,1),(2,1)]))  ==  True
-- ---------------------------------------------------------------------

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

module Relaciones_antisimetricas where

import Relaciones_binarias (Rel(R))
import Test.QuickCheck

-- 1ª solución
-- ===========

antisimetrica :: Eq a => Rel a -> Bool
antisimetrica (R (_,g)) =
  null [(x,y) | (x,y) <- g, x /= y, (y,x) `elem` g]

-- 2ª solución
-- ===========

antisimetrica2 :: Eq a => Rel a -> Bool
```

```haskell
antisimetrica2 (R (_,g)) =
  and [(y,x) `notElem` g | (x,y) <- g, x /= y]


-- 3ª solución
-- ===========

antisimetrica3 :: Eq a => Rel a -> Bool
antisimetrica3 (R (_,g)) =
  all (\(x, y) -> (y,x) `notElem` g || x == y) g


-- 4ª solución
-- ===========

antisimetrica4 :: Eq a => Rel a -> Bool
antisimetrica4 (R (u,g)) =
  and [((x,y) `elem` g && (y,x) `elem` g) --> (x == y)
      | x <- u, y <- u]
  where p --> q = not p || q

-- 5ª solución
-- ===========

antisimetrica5 :: Eq a => Rel a -> Bool
antisimetrica5 (R (_,g)) = aux g
  where aux []          = True
        aux ((x,y):g') = ((y,x) `notElem` g || x == y) && aux g'

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

-- La propiedad es
prop_antisimetrica :: Rel Int -> Bool
prop_antisimetrica r =
  all (== antisimetrica r)
      [antisimetrica2 r,
       antisimetrica3 r,
       antisimetrica4 r,
       antisimetrica5 r]
```

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

9.9.2. En Python

```python
# ---------------------------------------------------------------------
# Usando el [tipo de las relaciones binarias](https://bit.ly/3IVVqOT),
# definir la función
#    antisimetrica : (Rel[A]) -> bool
# tal que antisimetrica(r) se verifica si la relación r es
# antisimétrica; es decir, si (x,y) e (y,x) están relacionado, entonces
# x=y. Por ejemplo,
#    >>> antisimetrica(([1,2],[(1,2)]))
#    True
#    >>> antisimetrica(([1,2],[(1,2),(2,1)]))
#    False
#    >>> antisimetrica(([1,2],[(1,1),(2,1)]))
#    True
# ---------------------------------------------------------------------

from typing import TypeVar

from hypothesis import given
from hypothesis import strategies as st

from src.Relaciones_binarias import Rel, relacionArbitraria

A = TypeVar('A')

# 1ª solución
# ===========

def antisimetrica(r: Rel[A]) -> bool:
    (_, g) = r
    return [(x, y) for (x, y) in g if x != y and (y, x) in g] == []

# 2ª solución
# ===========
```

```
def antisimetrica2(r: Rel[A]) -> bool:
    (_, g) = r
    return all(((y, x) not in g for (x, y) in g if x != y))

# 3ª solución
# ===========

def antisimetrica3(r: Rel[A]) -> bool:
    (u, g) = r
    return all ((not ((x, y) in g and (y, x) in g) or x == y
                 for x in u for y in u))

# 4ª solución
# ===========

def antisimetrica4(r: Rel[A]) -> bool:
    (_, g) = r
    def aux(xys: list[tuple[A, A]]) -> bool:
        if not xys:
            return True
        (x, y) = xys[0]
        return ((y, x) not in g or x == y) and aux(xys[1:])

    return aux(g)

# 5ª solución
# ===========

def antisimetrica5(r: Rel[A]) -> bool:
    (_, g) = r
    for (x, y) in g:
        if (y, x) in g and x != y:
            return False
    return True

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

# La propiedad es
```

```python
@given(st.integers(min_value=0, max_value=10))
def test_antisimetrica(n: int) -> None:
    r = relacionArbitraria(n)
    res = antisimetrica(r)
    assert antisimetrica2(r) == res
    assert antisimetrica3(r) == res
    assert antisimetrica4(r) == res
    assert antisimetrica5(r) == res

# La comprobación es
#    > poetry run pytest -q Relaciones_antisimetricas.py
#    1 passed in 0.13s
```

9.10. Relaciones totales

9.10.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Usando el [tipo de las relaciones binarias](https://bit.ly/3IVVqOT),
-- definir la función
--    total :: Eq a => Rel a -> Bool
-- tal que (total r) se verifica si la relación r es total; es decir, si
-- para cualquier par x, y de elementos del universo de r, se tiene que
-- x está relacionado con y o y está relacionado con x. Por ejemplo,
--    total (R ([1,3],[(1,1),(3,1),(3,3)]))  ==  True
--    total (R ([1,3],[(1,1),(3,1)]))        ==  False
--    total (R ([1,3],[(1,1),(3,3)]))        ==  False
-- ---------------------------------------------------------------------

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

module Relaciones_totales where

import Relaciones_binarias (Rel(R))
import Test.QuickCheck (quickCheck)

-- 1ª solución
-- ===========

total :: Eq a => Rel a -> Bool
```

```
total (R (u,g)) =
  and [(x,y) `elem` g || (y,x) `elem` g | x <- u, y <- u]

-- 2ª solución
-- ===========

total2 :: Eq a => Rel a -> Bool
total2 (R (u,g)) =
  all (relacionados g) (producto u u)

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

-- (relacionados g (x,y)) se verifica si los elementos x e y están
-- relacionados por la relación de grafo g. Por ejemplo,
--    relacionados [(2,3),(3,1)] (2,3)  ==  True
--    relacionados [(2,3),(3,1)] (3,2)  ==  True
--    relacionados [(2,3),(3,1)] (1,2)  ==  False
relacionados :: Eq a => [(a,a)] -> (a,a) -> Bool
relacionados g (x,y) =
  (x,y) `elem` g || (y,x) `elem` g

-- 3ª solución
-- ===========

total3 :: Eq a => Rel a -> Bool
total3 (R (u,g)) = aux1 u
  where aux1 []       = True
        aux1 (x:xs)   = aux2 x u && aux1 xs
        aux2 _ []     = True
        aux2 x (y:ys) = relacionados g (x,y) && aux2 x ys

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

-- La propiedad es
```

```haskell
prop_total :: Rel Int -> Bool
prop_total r =
  all (== total r)
      [total2 r,
       total3 r]

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

9.10.2. En Python

```python
# ---------------------------------------------------------------------
# Usando el [tipo de las relaciones binarias](https://bit.ly/3IVVqOT),
# definir la función
#    total : (Rel[A]) -> bool
# tal que total(r) se verifica si la relación r es total; es decir, si
# para cualquier par x, y de elementos del universo de r, se tiene que
# x está relacionado con y o y está relacionado con x. Por ejemplo,
#    total (([1,3],[(1,1),(3,1),(3,3)]))  ==  True
#    total (([1,3],[(1,1),(3,1)]))        ==  False
#    total (([1,3],[(1,1),(3,3)]))        ==  False
# ---------------------------------------------------------------------

from typing import TypeVar

from hypothesis import given
from hypothesis import strategies as st

from src.Relaciones_binarias import Rel, relacionArbitraria

A = TypeVar('A')

# 1ª solución
# ===========

def total(r: Rel[A]) -> bool:
    (u, g) = r
    return all(((x, y) in g or (y, x) in g for x in u for y in u))
```

```python
# 2ª solución
# ===========

# producto(xs, ys) es el producto cartesiano de xs e ys. Por ejemplo,
#    >>> producto([2, 5], [1, 4, 6])
#    [(2, 1), (2, 4), (2, 6), (5, 1), (5, 4), (5, 6)]
def producto(xs: list[A], ys: list[A]) -> list[tuple[A,A]]:
    return [(x, y) for x in xs for y in ys]

# relacionados(g, (x, y)) se verifica si los elementos x e y están
# relacionados por la relación de grafo g. Por ejemplo,
#    relacionados([(2, 3), (3, 1)], (2, 3))  ==  True
#    relacionados([(2, 3), (3, 1)], (3, 2))  ==  True
#    relacionados([(2, 3), (3, 1)], (1, 2))  ==  False
def relacionados(g: list[tuple[A,A]], p: tuple[A,A]) -> bool:
    (x, y) = p
    return (x, y) in g or (y, x) in g

def total2(r: Rel[A]) -> bool:
    (u, g) = r
    return all(relacionados(g, p) for p in producto(u, u))

# 3ª solución
# ===========

def total3(r: Rel[A]) -> bool:
    u, g = r
    return all(relacionados(g, (x, y)) for x in u for y in u)

# 4ª solución
# ===========

def total4(r: Rel[A]) -> bool:
    (u, g) = r
    def aux2(x: A, ys: list[A]) -> bool:
        if not ys:
            return True
        return relacionados(g, (x, ys[0])) and aux2(x, ys[1:])

    def aux1(xs: list[A]) -> bool:
```

```
        if not xs:
            return True
        return aux2(xs[0], u) and aux1(xs[1:])

    return aux1(u)

# 5ª solución
# ===========

def total5(r: Rel[A]) -> bool:
    (u, g) = r
    for x in u:
        for y in u:
            if not relacionados(g, (x, y)):
                return False
    return True

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

# La propiedad es
@given(st.integers(min_value=0, max_value=10))
def test_total(n: int) -> None:
    r = relacionArbitraria(n)
    res = total(r)
    assert total2(r) == res
    assert total3(r) == res
    assert total4(r) == res
    assert total5(r) == res

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

9.11. Clausura reflexiva

9.11.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Usando el [tipo de las relaciones binarias](https://bit.ly/3IVVqOT),
```

```
-- definir la función
--    clausuraReflexiva :: Eq a => Rel a -> Rel a
-- tal que (clausuraReflexiva r) es la clausura reflexiva de r; es
-- decir, la menor relación reflexiva que contiene a r. Por ejemplo,
--    λ> clausuraReflexiva (R ([1,3],[(1,1),(3,1)]))
--    R ([1,3],[(1,1),(3,1),(3,3)])
-- ---------------------------------------------------------------------

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

module Clausura_reflexiva where

import Relaciones_binarias (Rel(R))
import Data.List (union)

clausuraReflexiva :: Eq a => Rel a -> Rel a
clausuraReflexiva (R (u,g)) =
  R (u, g `union` [(x,x) | x <- u])
```

9.11.2. En Python

```
# ---------------------------------------------------------------------
# Usando el [tipo de las relaciones binarias](https://bit.ly/3IVVqOT),
# definir la función
#    clausuraReflexiva : (Rel[A]) -> Rel[A]
# tal que clausuraReflexiva(r) es la clausura reflexiva de r; es
# decir, la menor relación reflexiva que contiene a r. Por ejemplo,
#    >>> clausuraReflexiva (([1,3],[(1,1),(3,1)]))
#    ([1, 3], [(3, 1), (1, 1), (3, 3)])
# ---------------------------------------------------------------------

from typing import TypeVar

from src.Relaciones_binarias import Rel

A = TypeVar('A')

def clausuraReflexiva(r: Rel[A]) -> Rel[A]:
    (u, g) = r
    return (u, list(set(g) | {(x, x) for x in u}))
```

9.12. Clausura simétrica

9.12.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Usando el [tipo de las relaciones binarias](https://bit.ly/3IVVqOT),
-- definir la función
--    clausuraSimetrica :: Eq a => Rel a -> Rel a
-- tal que (clausuraSimetrica r) es la clausura simétrica de r; es
-- decir, la menor relación simétrica que contiene a r. Por ejemplo,
--    λ> clausuraSimetrica (R ([1,3,5],[(1,1),(3,1),(1,5)]))
--    R ([1,3,5],[(1,1),(3,1),(1,5),(1,3),(5,1)])
--
-- Comprobar con QuickCheck que clausuraSimetrica es simétrica.
-- ---------------------------------------------------------------------

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

module Clausura_simetrica where

import Relaciones_binarias (Rel(R))
import Data.List (union)
import Relaciones_simetricas (simetrica)
import Test.QuickCheck

clausuraSimetrica :: Eq a => Rel a -> Rel a
clausuraSimetrica (R (u,g)) =
  R (u, g `union` [(y,x) | (x,y) <- g])

-- La propiedad es
prop_ClausuraSimetrica :: Rel Int -> Bool
prop_ClausuraSimetrica r =
  simetrica (clausuraSimetrica r)

-- La función simetrica está definida en el ejercicio
-- "Relaciones simétricas" que se encuentra en
-- https://bit.ly/3zlO2rH

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

9.12.2. En Python

```python
# ---------------------------------------------------------------------
# Usando el [tipo de las relaciones binarias](https://bit.ly/3IVVqOT),
# definir la función
#    clausuraSimetrica : (Rel[A]) -> Rel[A]
# tal que clausuraSimetrica(r) es la clausura simétrica de r; es
# decir, la menor relación simétrica que contiene a r. Por ejemplo,
#    >>> clausuraSimetrica(([1, 3, 5], [(1, 1), (3, 1), (1, 5)]))
#    ([1, 3, 5], [(1, 5), (3, 1), (1, 1), (1, 3), (5, 1)])
#
# Comprobar con Hypothesis que clausuraSimetrica es simétrica.
# ---------------------------------------------------------------------

from typing import TypeVar

from hypothesis import given
from hypothesis import strategies as st

from src.Relaciones_binarias import Rel, relacionArbitraria
from src.Relaciones_simetricas import simetrica

A = TypeVar('A')

def clausuraSimetrica(r: Rel[A]) -> Rel[A]:
    (u, g) = r
    return (u, list(set(g) | {(y, x) for (x,y) in g}))

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

# La propiedad es
@given(st.integers(min_value=0, max_value=10))
def test_irreflexiva(n: int) -> None:
    r = relacionArbitraria(n)
    assert simetrica(clausuraSimetrica(r))

# La función simetrica está definida en el ejercicio
# ”Relaciones simétricas” que se encuentra en
# https://bit.ly/3zlO2rH
```

```
# La comprobación es
#    > poetry run pytest -q Clausura_simetrica.py
#    1 passed in 0.12s
```

9.13. Clausura transitiva

9.13.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Usando el [tipo de las relaciones binarias](https://bit.ly/3IVVqOT),
-- definir la función
--    clausuraTransitiva :: Eq a => Rel a -> Rel a
-- tal que (clausuraTransitiva r) es la clausura transitiva de r; es
-- decir, la menor relación transitiva que contiene a r. Por ejemplo,
--    λ> clausuraTransitiva (R ([1..6],[(1,2),(2,5),(5,6)]))
--    R ([1,2,3,4,5,6],[(1,2),(2,5),(5,6),(1,5),(2,6),(1,6)])
--
-- Comprobar con QuickCheck que clausuraTransitiva es transitiva.
-- ---------------------------------------------------------------------

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

module Clausura_transitiva where

import Relaciones_binarias (Rel(R))
import Relaciones_transitivas (transitiva)
import Data.List (union)
import Test.QuickCheck

-- 1ª solución
-- ===========

clausuraTransitiva :: Ord a => Rel a -> Rel a
clausuraTransitiva (R (u,g)) = R (u, aux g)
  where
    aux u' | cerradoTr u' = u'
           | otherwise    = aux (u' `union` comp u' u')
    cerradoTr r       = subconjunto (comp r r) r
    comp r s          = [(x,z) | (x,y) <- r, (y',z) <- s, y == y']
    subconjunto xs ys = all (`elem` ys) xs
```

```haskell
-- 2ª solución
-- ===========

clausuraTransitiva2 :: Ord a => Rel a -> Rel a
clausuraTransitiva2 (R (u,g)) =
  R (u, until cerradoTr (\r -> r `union` comp r r) g)
  where
    cerradoTr r       = subconjunto (comp r r) r
    comp r s          = [(x,z) | (x,y) <- r, (y',z) <- s, y == y']
    subconjunto xs ys = all (`elem` ys) xs


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

-- La propiedad es
prop_clausuraTransitiva :: Rel Int -> Bool
prop_clausuraTransitiva r =
  clausuraTransitiva r == clausuraTransitiva2 r

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

-- Propiedad
-- =========

-- La propiedad es
prop_clausuraTransitivaEsTransitiva :: Rel Int -> Bool
prop_clausuraTransitivaEsTransitiva r =
  transitiva (clausuraTransitiva r)

-- La función transitiva está definida en el ejercicio
-- "Relaciones transitivas" que se encuentra en
-- https://bit.ly/42WRPJv

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

9.13.2. En Python

```
# ---------------------------------------------------------------------
# Usando el [tipo de las relaciones binarias](https://bit.ly/3IVVqOT),
# definir la función
#    clausuraTransitiva : (Rel[A]) -> Rel[A]
# tal que clausuraTransitiva(r) es la clausura transitiva de r; es
# decir, la menor relación transitiva que contiene a r. Por ejemplo,
#    >>> clausuraTransitiva (([1, 2, 3, 4, 5, 6], [(1, 2), (2, 5), (5, 6)]))
#    ([1, 2, 3, 4, 5, 6], [(1, 2), (2, 5), (5, 6), (2, 6), (1, 5), (1, 6)])
#
# Comprobar con Hypothesis que clausuraTransitiva es transitiva.
# ---------------------------------------------------------------------

from typing import TypeVar

from hypothesis import given
from hypothesis import strategies as st

from src.Relaciones_binarias import Rel, relacionArbitraria
from src.Relaciones_transitivas import transitiva

A = TypeVar('A')

# 1ª solución
# ===========

def clausuraTransitiva(r: Rel[A]) -> Rel[A]:
    (u, g) = r

    def subconjunto(xs: list[tuple[A, A]], ys: list[tuple[A, A]]) -> bool:
        return set(xs) <= set(ys)

    def comp(r: list[tuple[A, A]], s: list[tuple[A, A]]) -> list[tuple[A, A]]:
        return list({(x, z) for (x, y) in r for (y1, z) in s if y == y1})

    def cerradoTr(r: list[tuple[A, A]]) -> bool:
        return subconjunto(comp(r, r), r)

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

```
    def aux(u1: list[tuple[A, A]]) -> list[tuple[A, A]]:
        if cerradoTr(u1):
            return u1
        return aux(union(u1, comp(u1, u1)))

    return (u, aux(g))

# 2ª solución
# ===========

def clausuraTransitiva2(r: Rel[A]) -> Rel[A]:
    (u, g) = r

    def subconjunto(xs: list[tuple[A, A]], ys: list[tuple[A, A]]) -> bool:
        return set(xs) <= set(ys)

    def comp(r: list[tuple[A, A]], s: list[tuple[A, A]]) -> list[tuple[A, A]]:
        return list({(x, z) for (x, y) in r for (y1, z) in s if y == y1})

    def cerradoTr(r: list[tuple[A, A]]) -> bool:
        return subconjunto(comp(r, r), r)

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

    def aux(u1: list[tuple[A, A]]) -> list[tuple[A, A]]:
        if cerradoTr(u1):
            return u1
        return aux(union(u1, comp(u1, u1)))

    g1: list[tuple[A, A]] = g
    while not cerradoTr(g1):
        g1 = union(g1, comp(g1, g1))
    return (u, g1)

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

# La propiedad es
```

```
@given(st.integers(min_value=0, max_value=10))
def test_clausuraTransitiva(n: int) -> None:
    r = relacionArbitraria(n)
    assert clausuraTransitiva(r) == clausuraTransitiva2(r)

# Propiedad
# =========

# La propiedad es
@given(st.integers(min_value=0, max_value=10))
def test_cla(n: int) -> None:
    r = relacionArbitraria(n)
    assert transitiva(clausuraTransitiva(r))

# La función transitiva está definida en el ejercicio
# ”Relaciones transitivas” que se encuentra en
# https://bit.ly/42WRPJv

# La comprobación es
#    > poetry run pytest -q Clausura_transitiva.py
#    2 passed in 0.16s
```

Capítulo 10

El tipo abstracto de datos de los polinomios

Contenido

10.1. El tipo abstracto de datos de los polinomios

10.1.1. En Haskell

```
-- Un polinomio es una expresión matemática compuesta por una suma de
-- términos, donde cada término es el producto de un coeficiente y una
-- variable elevada a una potencia. Por ejemplo, el polinomio 3x^2+2x-1
-- tiene un término de segundo grado (3x^2), un término de primer grado
-- (2x) y un término constante (-1).
--
-- Las operaciones que definen al tipo abstracto de datos (TAD) de los
-- polinomios (cuyos coeficientes son del tipo a) son las siguientes:
--    polCero   :: Polinomio a
--    esPolCero :: Polinomio a -> Bool
--    consPol   :: (Num a, Eq a) => Int -> a -> Polinomio a -> Polinomio a
--    grado     :: Polinomio a -> Int
--    coefLider :: Num a => Polinomio a -> a
--    restoPol  :: (Num a, Eq a) => Polinomio a -> Polinomio a
-- tales que
--    + polCero es el polinomio cero.
--    + (esPolCero p) se verifica si p es el polinomio cero.
--    + (consPol n b p) es el polinomio bx^n+p
--    + (grado p) es el grado del polinomio p.
--    + (coefLider p) es el coeficiente líder del polinomio p.
--    + (restoPol p) es el resto del polinomio p.
--
-- Por ejemplo, el polinomio
--    3*x^4 + -5*x^2 + 3
-- se representa por
--    consPol 4 3 (consPol 2 (-5) (consPol 0 3 polCero))
--
-- Las operaciones tienen que verificar las siguientes propiedades:
```

```haskell
--    + esPolCero polCero
--    + n > grado p && b /= 0 ==> not (esPolCero (consPol n b p))
--    + consPol (grado p) (coefLider p) (restoPol p) == p
--    + n > grado p && b /= 0 ==> grado (consPol n b p) == n
--    + n > grado p && b /= 0 ==> coefLider (consPol n b p) == b
--    + n > grado p && b /= 0 ==> restoPol (consPol n b p) == p
--
-- Para usar el TAD hay que usar una implementación concreta. En
-- principio, consideraremos las siguientes:
--    + mediante tipo de dato algebraico,
--    + mediante listas densas y
--    + mediante listas dispersas.
-- Hay que elegir la que se desee utilizar, descomentándola y comentando
-- las otras.

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

module TAD.Polinomio
  ( Polinomio,
    polCero,   -- Polinomio a
    esPolCero, -- Polinomio a -> Bool
    consPol,   -- (Num a, Eq a) => Int -> a -> Polinomio a -> Polinomio a
    grado,     -- Polinomio a -> Int
    coefLider, -- Num a => Polinomio a -> a
    restoPol   -- (Num a, Eq a) => Polinomio a -> Polinomio a
  ) where

import TAD.PolRepTDA
-- import TAD.PolRepDensa
-- import TAD.PolRepDispersa
```

10.1.2. En Python

```python
# Un polinomio es una expresión matemática compuesta por una suma de
# términos, donde cada término es el producto de un coeficiente y una
# variable elevada a una potencia. Por ejemplo, el polinomio 3x^2+2x-1
# tiene un término de segundo grado (3x^2), un término de primer grado
# (2x) y un término constante (-1).
#
# Las operaciones que definen al tipo abstracto de datos (TAD) de los
```

```
# polinomios (cuyos coeficientes son del tipo a) son las siguientes:
#    polCero   :: Polinomio a
#    esPolCero :: Polinomio a -> Bool
#    consPol   :: (Num a, Eq a) => Int -> a -> Polinomio a -> Polinomio a
#    grado     :: Polinomio a -> Int
#    coefLider :: Num a => Polinomio a -> a
#    restoPol  :: (Num a, Eq a) => Polinomio a -> Polinomio a
# tales que
#    + polCero es el polinomio cero.
#    + (esPolCero p) se verifica si p es el polinomio cero.
#    + (consPol n b p) es el polinomio bx^n+p
#    + (grado p) es el grado del polinomio p.
#    + (coefLider p) es el coeficiente líder del polinomio p.
#    + (restoPol p) es el resto del polinomio p.
#
# Por ejemplo, el polinomio
#    3*x^4 + -5*x^2 + 3
# se representa por
#    consPol 4 3 (consPol 2 (-5) (consPol 0 3 polCero))
#
# Las operaciones tienen que verificar las siguientes propiedades:
#    + esPolCero polCero
#    + n > grado p && b /= 0 ==> not (esPolCero (consPol n b p))
#    + consPol (grado p) (coefLider p) (restoPol p) == p
#    + n > grado p && b /= 0 ==> grado (consPol n b p) == n
#    + n > grado p && b /= 0 ==> coefLider (consPol n b p) == b
#    + n > grado p && b /= 0 ==> restoPol (consPol n b p) == p
#
# Para usar el TAD hay que usar una implementación concreta. En
# principio, consideraremos las siguientes:
#    + mediante tipo de dato algebraico,
#    + mediante listas densas y
#    + mediante listas dispersas.
# Hay que elegir la que se desee utilizar, descomentándola y comentando
# las otras.

__all__ = [
    'Polinomio',
    'polCero',
    'esPolCero',
```

```
    'consPol',
    'grado',
    'coefLider',
    'restoPol',
    'polinomioAleatorio'
]

from src.TAD.PolRepDensa import (Polinomio, coefLider, consPol, esPolCero,
                                 grado, polCero, polinomioAleatorio, restoPol)

# from src.TAD.PolRepDispersa import (Polinomio, polCero, esPolCero,
#                                     consPol, grado, coefLider,
#                                     restoPol, polinomioAleatorio)
```

10.2. El TAD de los polinomios mediante tipos algebraicos

10.2.1. En Haskell

```haskell
{-# LANGUAGE TemplateHaskell #-}
{-# OPTIONS_GHC -fno-warn-unused-top-binds #-}

module TAD.PolRepTDA
  ( Polinomio,
    polCero,   -- Polinomio a
    esPolCero, -- Polinomio a -> Bool
    consPol,   -- (Num a, Eq a)) => Int -> a -> Polinomio a -> Polinomio a
    grado,     -- Polinomio a -> Int
    coefLider, -- Num t => Polinomio t -> t
    restoPol   -- Polinomio t -> Polinomio t
  ) where

import Test.QuickCheck

-- Representamos un polinomio mediante los constructores ConsPol y
-- PolCero. Por ejemplo, el polinomio
--    6x^4 -5x^2 + 4x -7
-- se representa por
--    ConsPol 4 6 (ConsPol 2 (-5) (ConsPol 1 4 (ConsPol 0 (-7) PolCero)))
```

```
-- Polinomio como tipo de dato algebra
data Polinomio a = PolCero
                 | ConsPol Int a (Polinomio a)
  deriving Eq

-- (escribePol p) es la cadena correspondiente al polinomio p. Por
-- ejemplo,
--    λ> escribePol (consPol 4 3 (consPol 2 (-5) (consPol 0 3 polCero)))
--    "3*x^4 + -5*x^2 + 3"
escribePol :: (Num a, Show a, Eq a) => Polinomio a -> String
escribePol PolCero               = "0"
escribePol (ConsPol 0 b PolCero) = show b
escribePol (ConsPol 0 b p)       = concat [show b, " + ", escribePol p]
escribePol (ConsPol 1 b PolCero) = show b ++ "*x"
escribePol (ConsPol 1 b p)       = concat [show b, "*x + ", escribePol p]
escribePol (ConsPol n 1 PolCero) = "x^" ++ show n
escribePol (ConsPol n b PolCero) = concat [show b, "*x^", show n]
escribePol (ConsPol n 1 p)       = concat ["x^", show n, " + ", escribePol p]
escribePol (ConsPol n b p)       = concat [show b, "*x^", show n, " + ", escribeP

-- Procedimiento de escritura de polinomios.
instance (Num a, Show a, Eq a) => Show (Polinomio a) where
  show = escribePol

-- Ejemplos de polinomios con coeficientes enteros:
ejPol1, ejPol2, ejPol3 :: Polinomio Int
ejPol1 = consPol 4 3 (consPol 2 (-5) (consPol 0 3 polCero))
ejPol2 = consPol 5 1 (consPol 2 5 (consPol 1 4 polCero))
ejPol3 = consPol 4 6 (consPol 1 2 polCero)

-- Comprobación de escritura:
--    > ejPol1
--    3*x^4 + -5*x^2 + 3
--    > ejPol2
--    x^5 + 5*x^2 + 4*x
--    > ejPol3
--    6*x^4 + 2*x

-- polCero es el polinomio cero. Por ejemplo,
```

```haskell
--    > polCero
--    0
polCero :: Polinomio a
polCero = PolCero

-- (esPolCero p) se verifica si p es el polinomio cero. Por ejemplo,
--    esPolCero polCero  ==  True
--    esPolCero ejPol1   ==  False
esPolCero :: Polinomio a -> Bool
esPolCero PolCero = True
esPolCero _       = False

-- (consPol n b p) es el polinomio bx^n+p. Por ejemplo,
--    ejPol2               ==  x^5 + 5*x^2 + 4*x
--    consPol 3 0 ejPol2   ==  x^5 + 5*x^2 + 4*x
--    consPol 3 2 polCero  ==  2*x^3
--    consPol 6 7 ejPol2   ==  7*x^6 + x^5 + 5*x^2 + 4*x
--    consPol 4 7 ejPol2   ==  x^5 + 7*x^4 + 5*x^2 + 4*x
--    consPol 5 7 ejPol2   ==  8*x^5 + 5*x^2 + 4*x
consPol :: (Num a, Eq a) => Int -> a -> Polinomio a -> Polinomio a
consPol _ 0 p = p
consPol n b PolCero = ConsPol n b PolCero
consPol n b (ConsPol m c p)
  | n > m       = ConsPol n b (ConsPol m c p)
  | n < m       = ConsPol m c (consPol n b p)
  | b+c == 0    = p
  | otherwise   = ConsPol n (b+c) p

-- (grado p) es el grado del polinomio p. Por ejemplo,
--    ejPol3        ==  6*x^4 + 2*x
--    grado ejPol3  ==  4
grado :: Polinomio a -> Int
grado PolCero           = 0
grado (ConsPol n _ _) = n

-- (coefLider p) es el coeficiente líder del polinomio p. Por ejemplo,
--    ejPol3            ==  6*x^4 + 2*x
--    coefLider ejPol3  ==  6
coefLider :: Num t => Polinomio t -> t
coefLider PolCero         = 0
```

```
coefLider (ConsPol _ b _) = b

-- (restoPol p) es el resto del polinomio p. Por ejemplo,
--    ejPol3           ==  6*x^4 + 2*x
--    restoPol ejPol3  ==  2*x
--    ejPol2           ==  x^5 + 5*x^2 + 4*x
--    restoPol ejPol2  ==  5*x^2 + 4*x
restoPol :: Polinomio t -> Polinomio t
restoPol PolCero         = PolCero
restoPol (ConsPol _ _ p) = p

-- Generador de polinomios                                              --
-- =======================

-- genPolinomio es un generador de polinomios. Por ejemplo,
--    λ> sample (genPol 1)
--    7*x^9 + 9*x^8 + 10*x^7 + -14*x^5 + -15*x^2 + -10
--    -4*x^8 + 2*x
--    -8*x^9 + 4*x^8 + 2*x^6 + 4*x^5 + -6*x^4 + 5*x^2 + -8*x
--    -9*x^9 + x^5 + -7
--    8*x^10 + -9*x^7 + 7*x^6 + 9*x^5 + 10*x^3 + -1*x^2
--    7*x^10 + 5*x^9 + -5
--    -8*x^10 + -7
--    -5*x
--    5*x^10 + 4*x^4 + -3
--    3*x^3 + -4
--    10*x
genPol :: (Num a, Arbitrary a, Eq a) => Int -> Gen (Polinomio a)
genPol 0 = return polCero
genPol _ = do
  n <- choose (0,10)
  b <- arbitrary
  p <- genPol (div n 2)
  return (consPol n b p)

instance (Num a, Arbitrary a, Eq a) => Arbitrary (Polinomio a) where
  arbitrary = sized genPol

-- Propiedades de los polinomios
-- =============================
```

```haskell
-- polCero es el polinomio cero.
prop_polCero_es_cero :: Bool
prop_polCero_es_cero =
  esPolCero polCero

-- Si n es mayor que el grado de p y b no es cero, entonces
-- (consPol n b p) es un polinomio distinto del cero.
prop_consPol_no_cero :: Int -> Int -> Polinomio Int -> Property
prop_consPol_no_cero n b p =
  n > grado p && b /= 0  ==>
  not (esPolCero (consPol n b p))

-- (consPol (grado p) (coefLider p) (restoPol p)) es igual a p.
prop_consPol :: Polinomio Int -> Bool
prop_consPol p =
  consPol (grado p) (coefLider p) (restoPol p) == p

-- Si n es mayor que el grado de p y b no es cero, entonces
-- el grado de (consPol n b p) es n.
prop_grado :: Int -> Int -> Polinomio Int -> Property
prop_grado n b p =
  n > grado p && b /= 0 ==>
  grado (consPol n b p) == n

-- Si n es mayor que el grado de p y b no es cero, entonces
-- el coeficiente líder de (consPol n b p) es b.
prop_coefLider :: Int -> Int -> Polinomio Int -> Property
prop_coefLider n b p =
  n > grado p && b /= 0 ==>
  coefLider (consPol n b p) == b

-- Si n es mayor que el grado de p y b no es cero, entonces
-- el resto de (consPol n b p) es p.
prop_restoPol :: Int -> Int -> Polinomio Int -> Property
prop_restoPol n b p =
  n > grado p && b /= 0 ==>
  restoPol (consPol n b p) == p

-- Verificación
```

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

return []

verificaPol :: IO Bool
verificaPol = $quickCheckAll

-- La verificación es
--    λ> verificaPol
--    === prop_polCero_es_cero from PolPropiedades.hs:53 ===
--    +++ OK, passed 1 test.
--
--    === prop_consPol_no_cero from PolPropiedades.hs:63 ===
--    +++ OK, passed 100 tests; 251 discarded.
--
--    === prop_consPol from PolPropiedades.hs:73 ===
--    +++ OK, passed 100 tests.
--
--    === prop_grado from PolPropiedades.hs:83 ===
--    +++ OK, passed 100 tests; 321 discarded.
--
--    === prop_coefLider from PolPropiedades.hs:94 ===
--    +++ OK, passed 100 tests; 340 discarded.
--
--    === prop_restoPol from PolPropiedades.hs:105 ===
--    +++ OK, passed 100 tests; 268 discarded.
--
--    True
```

10.3. El TAD de los polinomios mediante listas densas

10.3.1. En Haskell

```
{-# LANGUAGE TemplateHaskell #-}
{-# OPTIONS_GHC -fno-warn-unused-top-binds #-}

module TAD.PolRepDensa
  ( Polinomio,
```

```haskell
    polCero,   -- Polinomio a
    esPolCero, -- Polinomio a -> Bool
    consPol,   -- (Num a, Eq a) => Int -> a -> Polinomio a -> Polinomio a
    grado,     -- Polinomio a -> Int
    coefLider, -- Num a => Polinomio a -> a
    restoPol   -- (Num a, Eq a) => Polinomio a -> Polinomio a
  ) where

import Test.QuickCheck

-- Representaremos un polinomio por la lista de sus coeficientes ordenados
-- en orden decreciente según el grado. Por ejemplo, el polinomio
--    6x^4 -5x^2 + 4x -7
-- se representa por
--    [6,0,-2,4,-7].
--
-- En la representación se supone que, si la lista no es vacía, su
-- primer elemento es distinto de cero.

newtype Polinomio a = Pol [a]
  deriving Eq

-- (escribePol p) es la cadena correspondiente al polinomio p. Por
-- ejemplo,
--    λ> escribePol (consPol 4 3 (consPol 2 (-5) (consPol 0 3 polCero)))
--    "3*x^4 + -5*x^2 + 3"
escribePol :: (Num a, Show a, Eq a) => Polinomio a -> String
escribePol pol
  | esPolCero pol         = "0"
  | n == 0 && esPolCero p = show a
  | n == 0                = concat [show a, " + ", escribePol p]
  | n == 1 && esPolCero p = show a ++ "*x"
  | n == 1                = concat [show a, "*x + ", escribePol p]
  | a == 1 && esPolCero p = "x^" ++ show n
  | esPolCero p           = concat [show a, "*x^", show n]
  | a == 1                = concat ["x^", show n, " + ", escribePol p]
  | otherwise             = concat [show a, "*x^", show n, " + ", escribePol p]
  where n = grado pol
        a = coefLider pol
        p = restoPol pol
```

```haskell
-- Procedimiento de escritura de polinomios.
instance (Num a, Show a, Eq a) => Show (Polinomio a) where
  show = escribePol

-- Ejemplos de polinomios con coeficientes enteros:
ejPol1, ejPol2, ejPol3 :: Polinomio Int
ejPol1 = consPol 4 3 (consPol 2 (-5) (consPol 0 3 polCero))
ejPol2 = consPol 5 1 (consPol 2 5 (consPol 1 4 polCero))
ejPol3 = consPol 4 6 (consPol 1 2 polCero)

-- Comprobación de escritura:
--    > ejPol1
--    3*x^4 + -5*x^2 + 3
--    > ejPol2
--    x^5 + 5*x^2 + 4*x
--    > ejPol3
--    6*x^4 + 2*x

-- polCero es el polinomio cero. Por ejemplo,
--    λ> polCero
--    0
polCero :: Polinomio a
polCero = Pol []

-- (esPolCero p) se verifica si p es el polinomio cero. Por ejemplo,
--    esPolCero polCero  ==  True
--    esPolCero ejPol1   ==  False
esPolCero :: Polinomio a -> Bool
esPolCero (Pol []) = True
esPolCero _        = False

-- (consPol n b p) es el polinomio bx^n+p. Por ejemplo,
--    ejPol2               ==  x^5 + 5*x^2 + 4*x
--    consPol 3 0 ejPol2   ==  x^5 + 5*x^2 + 4*x
--    consPol 3 2 polCero  ==  2*x^3
--    consPol 6 7 ejPol2   ==  7*x^6 + x^5 + 5*x^2 + 4*x
--    consPol 4 7 ejPol2   ==  x^5 + 7*x^4 + 5*x^2 + 4*x
--    consPol 5 7 ejPol2   ==  8*x^5 + 5*x^2 + 4*x
consPol :: (Num a, Eq a) => Int -> a -> Polinomio a -> Polinomio a
```

```
consPol _ 0 p = p
consPol n b p@(Pol xs)
    | esPolCero p = Pol (b : replicate n 0)
    | n > m       = Pol (b : replicate (n-m-1) 0 ++ xs)
    | n < m       = consPol m c (consPol n b (restoPol p))
    | b+c == 0    = Pol (dropWhile (==0) (tail xs))
    | otherwise   = Pol ((b+c):tail xs)
    where
      c = coefLider p
      m = grado p

-- (grado p) es el grado del polinomio p. Por ejemplo,
--    ejPol3        ==  6*x^4 + 2*x
--    grado ejPol3  ==  4
grado :: Polinomio a -> Int
grado (Pol []) = 0
grado (Pol xs) = length xs - 1

-- (coefLider p) es el coeficiente líder del polinomio p. Por ejemplo,
--    ejPol3            ==  6*x^4 + 2*x
--    coefLider ejPol3  ==  6
coefLider :: Num t => Polinomio t -> t
coefLider (Pol [])    = 0
coefLider (Pol (a:_)) = a

-- (restoPol p) es el resto del polinomio p. Por ejemplo,
--    ejPol3           ==  6*x^4 + 2*x
--    restoPol ejPol3  ==  2*x
--    ejPol2           ==  x^5 + 5*x^2 + 4*x
--    restoPol ejPol2  ==  5*x^2 + 4*x
restoPol :: (Num t, Eq t) => Polinomio t -> Polinomio t
restoPol (Pol [])     = polCero
restoPol (Pol [_])    = polCero
restoPol (Pol (_:b:as))
  | b == 0    = Pol (dropWhile (==0) as)
  | otherwise = Pol (b:as)

-- Generador de polinomios
-- =======================
```

```
-- genPolinomio es un generador de polinomios. Por ejemplo,
--    λ> sample (genPol 1)
--    7*x^9 + 9*x^8 + 10*x^7 + -14*x^5 + -15*x^2 + -10
--    -4*x^8 + 2*x
--    -8*x^9 + 4*x^8 + 2*x^6 + 4*x^5 + -6*x^4 + 5*x^2 + -8*x
--    -9*x^9 + x^5 + -7
--    8*x^10 + -9*x^7 + 7*x^6 + 9*x^5 + 10*x^3 + -1*x^2
--    7*x^10 + 5*x^9 + -5
--    -8*x^10 + -7
--    -5*x
--    5*x^10 + 4*x^4 + -3
--    3*x^3 + -4
--    10*x
genPol :: (Num a, Arbitrary a, Eq a) => Int -> Gen (Polinomio a)
genPol 0 = return polCero
genPol _ = do
  n <- choose (0,10)
  b <- arbitrary
  p <- genPol (div n 2)
  return (consPol n b p)

instance (Num a, Arbitrary a, Eq a) => Arbitrary (Polinomio a) where
  arbitrary = sized genPol

-- Propiedades de los polinomios
-- =============================

-- polCero es el polinomio cero.
prop_polCero_es_cero :: Bool
prop_polCero_es_cero =
  esPolCero polCero

-- Si n es mayor que el grado de p y b no es cero, entonces
-- (consPol n b p) es un polinomio distinto del cero.
prop_consPol_no_cero :: Int -> Int -> Polinomio Int -> Property
prop_consPol_no_cero n b p =
  n > grado p && b /= 0  ==>
  not (esPolCero (consPol n b p))

-- (consPol (grado p) (coefLider p) (restoPol p)) es igual a p.
```

```haskell
prop_consPol :: Polinomio Int -> Bool
prop_consPol p =
  consPol (grado p) (coefLider p) (restoPol p) == p

-- Si n es mayor que el grado de p y b no es cero, entonces
-- el grado de (consPol n b p) es n.
prop_grado :: Int -> Int -> Polinomio Int -> Property
prop_grado n b p =
  n > grado p && b /= 0 ==>
  grado (consPol n b p) == n

-- Si n es mayor que el grado de p y b no es cero, entonces
-- el coeficiente líder de (consPol n b p) es b.
prop_coefLider :: Int -> Int -> Polinomio Int -> Property
prop_coefLider n b p =
  n > grado p && b /= 0 ==>
  coefLider (consPol n b p) == b

-- Si n es mayor que el grado de p y b no es cero, entonces
-- el resto de (consPol n b p) es p.
prop_restoPol :: Int -> Int -> Polinomio Int -> Property
prop_restoPol n b p =
  n > grado p && b /= 0 ==>
  restoPol (consPol n b p) == p

-- Verificación
-- ============

return []

verificaPol :: IO Bool
verificaPol = $quickCheckAll

-- La verificación es
--    λ> verificaPol
--    === prop_polCero_es_cero from /home/jalonso/alonso/estudio/Exercitium/Exerc
--    +++ OK, passed 1 test.
--
--    === prop_consPol_no_cero from /home/jalonso/alonso/estudio/Exercitium/Exerc
--    +++ OK, passed 100 tests; 274 discarded.
```

```
--
--    === prop_consPol from /home/jalonso/alonso/estudio/Exercitium/Exercitium/sr
--    +++ OK, passed 100 tests.
--
--    === prop_grado from /home/jalonso/alonso/estudio/Exercitium/Exercitium/src/
--    +++ OK, passed 100 tests; 297 discarded.
--
--    === prop_coefLider from /home/jalonso/alonso/estudio/Exercitium/Exercitium/
--    +++ OK, passed 100 tests; 248 discarded.
--
--    === prop_restoPol from /home/jalonso/alonso/estudio/Exercitium/Exercitium/s
--    +++ OK, passed 100 tests; 322 discarded.
--
--    True
```

10.3.2. En Python

```
# Representaremos un polinomio por la lista de sus coeficientes ordenados
# en orden decreciente según el grado. Por ejemplo, el polinomio
#    6x^4 -5x^2 + 4x -7
# se representa por
#    [6,0,-2,4,-7].
#
# En la representación se supone que, si la lista no es vacía, su
# primer elemento es distinto de cero.
#
# Se define la clase Polinomio con los siguientes métodos:
#    + esPolCero() se verifica si es el polinomio cero.
#    + consPol(n, b) es el polinomio obtenido añadiendo el térmiono bx^n
#    + grado() es el grado del polinomio.
#    + coefLider() es el coeficiente líder del polinomio.
#    + restoPol() es el resto del polinomio.
# Por ejemplo,
#    >>> Polinomio()
#    0
#    >>> ejPol1 = Polinomio().consPol(0,3).consPol(2,-5).consPol(4,3)
#    >>> ejPol1
#    3*x^4 + -5*x^2 + 3
#    >>> ejPol2 = Polinomio().consPol(1,4).consPol(2,5).consPol(5,1)
#    >>> ejPol2
```

```
#    x^5 + 5*x^2 + 4*x
#    >>> ejPol3 = Polinomio().consPol(1,2).consPol(4,6)
#    >>> ejPol3
#    6*x^4 + 2*x
#    >>> Polinomio().esPolCero()
#    True
#    >>> ejPol1.esPolCero()
#    False
#    >>> ejPol2
#    x^5 + 5*x^2 + 4*x
#    >>> ejPol2.consPol(3,0)
#    x^5 + 5*x^2 + 4*x
#    >>> Polinomio().consPol(3,2)
#    2*x^3
#    >>> ejPol2.consPol(6,7)
#    7*x^6 + x^5 + 5*x^2 + 4*x
#    >>> ejPol2.consPol(4,7)
#    x^5 + 7*x^4 + 5*x^2 + 4*x
#    >>> ejPol2.consPol(5,7)
#    8*x^5 + 5*x^2 + 4*x
#    >>> ejPol3
#    6*x^4 + 2*x
#    >>> ejPol3.grado()
#    4
#    >>> ejPol3.restoPol()
#    2*x
#    >>> ejPol2
#    x^5 + 5*x^2 + 4*x
#    >>> ejPol2.restoPol()
#    5*x^2 + 4*x
#
# Además se definen las correspondientes funciones. Por ejemplo,
#    >>> polCero()
#    0
#    >>> ejPol1a = consPol(4,3,consPol(2,-5,consPol(0,3,polCero())))
#    >>> ejPol1a
#    3*x^4 + -5*x^2 + 3
#    >>> ejPol2a = consPol(5,1,consPol(2,5,consPol(1,4,polCero())))
#    >>> ejPol2a
#    x^5 + 5*x^2 + 4*x
```

```
#    >>> ejPol3a = consPol(4,6,consPol(1,2,polCero()))
#    >>> ejPol3a
#    6*x^4 + 2*x
#    >>> esPolCero(polCero())
#    True
#    >>> esPolCero(ejPol1a)
#    False
#    >>> ejPol2a
#    x^5 + 5*x^2 + 4*x
#    >>> consPol(3,9,ejPol2a)
#    x^5 + 9*x^3 + 5*x^2 + 4*x
#    >>> consPol(3,2,polCero())
#    2*x^3
#    >>> consPol(6,7,ejPol2a)
#    7*x^6 + x^5 + 5*x^2 + 4*x
#    >>> consPol(4,7,ejPol2a)
#    x^5 + 7*x^4 + 5*x^2 + 4*x
#    >>> consPol(5,7,ejPol2a)
#    8*x^5 + 5*x^2 + 4*x
#    >>> ejPol3a
#    6*x^4 + 2*x
#    >>> grado(ejPol3a)
#    4
#    >>> restoPol(ejPol3a)
#    2*x
#    >>> ejPol2a
#    x^5 + 5*x^2 + 4*x
#    >>> restoPol(ejPol2a)
#    5*x^2 + 4*x
#
# Finalmente, se define un generador aleatorio de polinomios y se
# comprueba que los polinomios cumplen las propiedades de su
# especificación.

from __future__ import annotations

__all__ = [
    'Polinomio',
    'polCero',
    'esPolCero',
```

```
    'consPol',
    'grado',
    'coefLider',
    'restoPol',
    'polinomioAleatorio'
]

from dataclasses import dataclass, field
from itertools import dropwhile
from typing import Generic, TypeVar

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

A = TypeVar('A', int, float, complex)

# Clase de los polinomios mediante listas densas
# ==============================================

@dataclass
class Polinomio(Generic[A]):
    _coeficientes: list[A] = field(default_factory=list)

    def esPolCero(self) -> bool:
        return not self._coeficientes

    def grado(self) -> int:
        if self.esPolCero():
            return 0
        return len(self._coeficientes) - 1

    def coefLider(self) -> A:
        if self.esPolCero():
            return 0
        return self._coeficientes[0]

    def restoPol(self) -> Polinomio[A]:
        xs = self._coeficientes
        if len(xs) <= 1:
            return Polinomio([])
```

```
        if xs[1] == 0:
            return Polinomio(list(dropwhile(lambda x: x == 0, xs[2:])))
        return Polinomio(xs[1:])

    def consPol(self, n: int, b: A) -> Polinomio[A]:
        m = self.grado()
        c = self.coefLider()
        xs = self._coeficientes
        if b == 0:
            return self
        if self.esPolCero():
            return Polinomio([b] + ([0] * n))
        if n > m:
            return Polinomio([b] + ([0] * (n-m-1)) + xs)
        if n < m:
            return self.restoPol().consPol(n, b).consPol(m, c)
        if b + c == 0:
            return Polinomio(list(dropwhile(lambda x: x == 0, xs[1:])))
        return Polinomio([b + c] + xs[1:])

    def __repr__(self) -> str:
        n = self.grado()
        a = self.coefLider()
        p = self.restoPol()
        if self.esPolCero():
            return "0"
        if n == 0 and p.esPolCero():
            return str(a)
        if n == 0:
            return str(a) + " + " + str(p)
        if n == 1 and p.esPolCero():
            return str(a) + "*x"
        if n == 1:
            return str(a) + "*x + " + str(p)
        if a == 1 and p.esPolCero():
            return "x^" + str(n)
        if p.esPolCero():
            return str(a) + "*x^" + str(n)
        if a == 1:
            return "x^" + str(n) + " + " + str(p)
```

```
        return str(a) + "*x^" + str(n) + " + " + str(p)

# Funciones del tipo polinomio
# ============================

def polCero() -> Polinomio[A]:
    return Polinomio([])

def esPolCero(p: Polinomio[A]) -> bool:
    return p.esPolCero()

def grado(p: Polinomio[A]) -> int:
    return p.grado()

def coefLider(p: Polinomio[A]) -> A:
    return p.coefLider()

def restoPol(p: Polinomio[A]) -> Polinomio[A]:
    return p.restoPol()

def consPol(n: int, b: A, p: Polinomio[A]) -> Polinomio[A]:
    return p.consPol(n, b)

# Generador de polinomios
# =======================

# normal(xs) es la lista obtenida eliminando los ceros iniciales de
# xs. Por ejmplo,
#    >>> normal([0,0,5,0])
#    [5, 0]
#    >>> normal([0,0,0,0])
#    []
def normal(xs: list[A]) -> list[A]:
    return list(dropwhile(lambda x: x == 0, xs))

# polinomioAleatorio() genera polinomios aleatorios. Por ejemplo,
#    >>> polinomioAleatorio().example()
#    9*x^6 + -7*x^5 + 7*x^3 + x^2 + 7
#    >>> polinomioAleatorio().example()
#    -3*x^7 + 8*x^6 + 2*x^5 + x^4 + -1*x^3 + -6*x^2 + 8*x + -6
```

```
#    >>> polinomioAleatorio().example()
#    x^2 + 7*x + -1
def polinomioAleatorio() -> st.SearchStrategy[Polinomio[int]]:
    return st.lists(st.integers(min_value=-9, max_value=9), max_size=10)\
             .map(lambda xs: normal(xs))\
             .map(Polinomio)

# Comprobación de las propiedades de los polinomios
# =================================================

# Las propiedades son
def test_esPolCero1() -> None:
    assert esPolCero(polCero())

@given(p=polinomioAleatorio(),
       n=st.integers(min_value=0, max_value=10),
       b=st.integers())
def test_esPolCero2(p: Polinomio[int], n: int, b: int) -> None:
    assume(n > grado(p) and b != 0)
    assert not esPolCero(consPol(n, b, p))

@given(p=polinomioAleatorio())
def test_consPol(p: Polinomio[int]) -> None:
    assume(not esPolCero(p))
    assert consPol(grado(p), coefLider(p), restoPol(p)) == p

@given(p=polinomioAleatorio(),
       n=st.integers(min_value=0, max_value=10),
       b=st.integers())
def test_grado(p: Polinomio[int], n: int, b: int) -> None:
    assume(n > grado(p) and b != 0)
    assert grado(consPol(n, b, p)) == n

@given(p=polinomioAleatorio(),
       n=st.integers(min_value=0, max_value=10),
       b=st.integers())
def test_coefLider(p: Polinomio[int], n: int, b: int) -> None:
    assume(n > grado(p) and b != 0)
    assert coefLider(consPol(n, b, p)) == b
```

```
@given(p=polinomioAleatorio(),
       n=st.integers(min_value=0, max_value=10),
       b=st.integers())
def test_restoPol(p: Polinomio[int], n: int, b: int) -> None:
    assume(n > grado(p) and b != 0)
    assert restoPol(consPol(n, b, p)) == p

# La comprobación es
#    > poetry run pytest -v PolRepDensa.py
#
#    PolRepDensa.py::test_esPolCero1 PASSED
#    PolRepDensa.py::test_esPolCero2 PASSED
#    PolRepDensa.py::test_consPol PASSED
#    PolRepDensa.py::test_grado PASSED
#    PolRepDensa.py::test_coefLider PASSED
#    PolRepDensa.py::test_restoPol PASSED
#
#    === 6 passed in 1.64s ===
```

10.4. El TAD de los polinomios mediante listas dispersas

10.4.1. En Haskell

```
{-# LANGUAGE TemplateHaskell #-}
{-# OPTIONS_GHC -fno-warn-unused-top-binds #-}

module TAD.PolRepDispersa
  ( Polinomio,
    polCero,   -- Polinomio a
    esPolCero, -- Num a =>  Polinomio a -> Bool
    consPol,   -- Num a => Int -> a -> Polinomio a -> Polinomio a
    grado,     -- Polinomio a -> Int
    coefLider, -- Num a => Polinomio a -> a
    restoPol   -- Polinomio a -> Polinomio a
  ) where

import Test.QuickCheck
```

```haskell
-- Representaremos un polinomio mediante una lista de pares (grado,coef),
-- ordenados en orden decreciente según el grado. Por ejemplo, el
-- polinomio
--    6x^4 -5x^2 + 4x -7
-- se representa por
--    [(4,6),(2,-5),(1,4),(0,-7)].
--
-- En la representación se supone que los primeros elementos de los
-- pares forman una sucesión estrictamente decreciente y que los
-- segundos elementos son distintos de cero.

newtype Polinomio a = Pol [(Int,a)]
  deriving Eq

-- (escribePol p) es la cadena correspondiente al polinomio p. Por
-- ejemplo,
--    λ> escribePol (consPol 4 3 (consPol 2 (-5) (consPol 0 3 polCero)))
--    "3*x^4 + -5*x^2 + 3"
escribePol :: (Num a, Show a, Eq a) => Polinomio a -> String
escribePol pol
  | esPolCero pol         = "0"
  | n == 0 && esPolCero p = show a
  | n == 0                = concat [show a, " + ", escribePol p]
  | n == 1 && esPolCero p = show a ++ "*x"
  | n == 1                = concat [show a, "*x + ", escribePol p]
  | a == 1 && esPolCero p = "x^" ++ show n
  | esPolCero p           = concat [show a, "*x^", show n]
  | a == 1                = concat ["x^", show n, " + ", escribePol p]
  | otherwise             = concat [show a, "*x^", show n, " + ", escribePol p]
  where n = grado pol
        a = coefLider pol
        p = restoPol pol

-- Procedimiento de escritura de polinomios.
instance (Num a, Show a, Eq a) => Show (Polinomio a) where
  show = escribePol

-- Ejemplos de polinomios con coeficientes enteros:
ejPol1, ejPol2, ejPol3 :: Polinomio Int
ejPol1 = consPol 4 3 (consPol 2 (-5) (consPol 0 3 polCero))
```

```
ejPol2 = consPol 5 1 (consPol 2 5 (consPol 1 4 polCero))
ejPol3 = consPol 4 6 (consPol 1 2 polCero)

-- Comprobación de escritura:
--    > ejPol1
--    3*x^4 + -5*x^2 + 3
--    > ejPol2
--    x^5 + 5*x^2 + 4*x
--    > ejPol3
--    6*x^4 + 2*x

-- polCero es el polinomio cero. Por ejemplo,
--    λ> polCero
--    0
polCero :: Num a => Polinomio a
polCero = Pol []

-- (esPolCero p) se verifica si p es el polinomio cero. Por ejemplo,
--    esPolCero polCero  ==  True
--    esPolCero ejPol1   ==  False
esPolCero :: Num a => Polinomio a -> Bool
esPolCero (Pol []) = True
esPolCero _        = False

-- (consPol n b p) es el polinomio bx^n+p. Por ejemplo,
--    ejPol2               ==  x^5 + 5*x^2 + 4*x
--    consPol 3 0 ejPol2   ==  x^5 + 5*x^2 + 4*x
--    consPol 3 2 polCero  ==  2*x^3
--    consPol 6 7 ejPol2   ==  7*x^6 + x^5 + 5*x^2 + 4*x
--    consPol 4 7 ejPol2   ==  x^5 + 7*x^4 + 5*x^2 + 4*x
--    consPol 5 7 ejPol2   ==  8*x^5 + 5*x^2 + 4*x
consPol :: (Num a, Eq a) => Int -> a -> Polinomio a -> Polinomio a
consPol _ 0 p = p
consPol n b p@(Pol xs)
    | esPolCero p = Pol [(n,b)]
    | n > m       = Pol ((n,b):xs)
    | n < m       = consPol m c (consPol n b (Pol (tail xs)))
    | b+c == 0    = Pol (tail xs)
    | otherwise   = Pol ((n,b+c) : tail xs)
    where
```

```
      c = coefLider p
      m = grado p

-- (grado p) es el grado del polinomio p. Por ejemplo,
--    ejPol3        ==  6*x^4 + 2*x
--    grado ejPol3  ==  4
grado :: Polinomio a -> Int
grado (Pol [])        = 0
grado (Pol ((n,_):_)) = n

-- (coefLider p) es el coeficiente líder del polinomio p. Por ejemplo,
--    ejPol3            ==  6*x^4 + 2*x
--    coefLider ejPol3  ==  6
coefLider :: Num t => Polinomio t -> t
coefLider (Pol [])        = 0
coefLider (Pol ((_,b):_)) = b

-- (restoPol p) es el resto del polinomio p. Por ejemplo,
--    ejPol3            ==  6*x^4 + 2*x
--    restoPol ejPol3   ==  2*x
--    ejPol2            ==  x^5 + 5*x^2 + 4*x
--    restoPol ejPol2   ==  5*x^2 + 4*x
restoPol :: Num t => Polinomio t -> Polinomio t
restoPol (Pol [])     = polCero
restoPol (Pol [_])    = polCero
restoPol (Pol (_:xs)) = Pol xs

-- Generador de polinomios                                             --
-- =======================

-- genPolinomio es un generador de polinomios. Por ejemplo,
--    λ> sample (genPol 1)
--    7*x^9 + 9*x^8 + 10*x^7 + -14*x^5 + -15*x^2 + -10
--    -4*x^8 + 2*x
--    -8*x^9 + 4*x^8 + 2*x^6 + 4*x^5 + -6*x^4 + 5*x^2 + -8*x
--    -9*x^9 + x^5 + -7
--    8*x^10 + -9*x^7 + 7*x^6 + 9*x^5 + 10*x^3 + -1*x^2
--    7*x^10 + 5*x^9 + -5
--    -8*x^10 + -7
--    -5*x
```

```haskell
--    5*x^10 + 4*x^4 + -3
--    3*x^3 + -4
--    10*x
genPol :: (Num a, Arbitrary a, Eq a) => Int -> Gen (Polinomio a)
genPol 0 = return polCero
genPol _ = do
  n <- choose (0,10)
  b <- arbitrary
  p <- genPol (div n 2)
  return (consPol n b p)

instance (Num a, Arbitrary a, Eq a) => Arbitrary (Polinomio a) where
  arbitrary = sized genPol

-- Propiedades de los polinomios
-- =============================

-- polCero es el polinomio cero.
prop_polCero_es_cero :: Bool
prop_polCero_es_cero =
  esPolCero polCero

-- Si n es mayor que el grado de p y b no es cero, entonces
-- (consPol n b p) es un polinomio distinto del cero.
prop_consPol_no_cero :: Int -> Int -> Polinomio Int -> Property
prop_consPol_no_cero n b p =
  n > grado p && b /= 0  ==>
  not (esPolCero (consPol n b p))

-- (consPol (grado p) (coefLider p) (restoPol p)) es igual a p.
prop_consPol :: Polinomio Int -> Bool
prop_consPol p =
  consPol (grado p) (coefLider p) (restoPol p) == p

-- Si n es mayor que el grado de p y b no es cero, entonces
-- el grado de (consPol n b p) es n.
prop_grado :: Int -> Int -> Polinomio Int -> Property
prop_grado n b p =
  n > grado p && b /= 0 ==>
  grado (consPol n b p) == n
```

```
-- Si n es mayor que el grado de p y b no es cero, entonces
-- el coeficiente líder de (consPol n b p) es b.
prop_coefLider :: Int -> Int -> Polinomio Int -> Property
prop_coefLider n b p =
  n > grado p && b /= 0 ==>
  coefLider (consPol n b p) == b

-- Si n es mayor que el grado de p y b no es cero, entonces
-- el resto de (consPol n b p) es p.
prop_restoPol :: Int -> Int -> Polinomio Int -> Property
prop_restoPol n b p =
  n > grado p && b /= 0 ==>
  restoPol (consPol n b p) == p

-- Verificación
-- ============

return []

verificaPol :: IO Bool
verificaPol = $quickCheckAll

-- La verificación es
--    λ> verificaPol
--    === prop_polCero_es_cero from /home/jalonso/alonso/estudio/Exercitium/Exerc
--    +++ OK, passed 1 test.
--
--    === prop_consPol_no_cero from /home/jalonso/alonso/estudio/Exercitium/Exerc
--    +++ OK, passed 100 tests; 264 discarded.
--
--    === prop_consPol from /home/jalonso/alonso/estudio/Exercitium/Exercitium/sr
--    +++ OK, passed 100 tests.
--
--    === prop_grado from /home/jalonso/alonso/estudio/Exercitium/Exercitium/src/
--    +++ OK, passed 100 tests; 266 discarded.
--
--    === prop_coefLider from /home/jalonso/alonso/estudio/Exercitium/Exercitium/
--    +++ OK, passed 100 tests; 251 discarded.
--
```

```
--    === prop_restoPol from /home/jalonso/alonso/estudio/Exercitium/Exercitium/s
--    +++ OK, passed 100 tests; 254 discarded.
--
--    True
```

10.4.2. En Python

```
# Representaremos un polinomio mediante una lista de pares (grado,coef),
# ordenados en orden decreciente según el grado. Por ejemplo, el
# polinomio
#    6x^4 -5x^2 + 4x -7
# se representa por
#    [(4,6),(2,-5),(1,4),(0,-7)].
#
# En la representación se supone que los primeros elementos de los
# pares forman una sucesión estrictamente decreciente y que los
# segundos elementos son distintos de cero.
#
# Se define la clase Polinomio con los siguientes métodos:
#    + esPolCero() se verifica si es el polinomio cero.
#    + consPol(n, b) es el polinomio obtenido añadiendo el térmiono bx^n
#    + grado() es el grado del polinomio.
#    + coefLider() es el coeficiente líder del polinomio.
#    + restoPol() es el resto del polinomio.
# Por ejemplo,
#    >>> Polinomio()
#    0
#    >>> ejPol1 = Polinomio().consPol(0,3).consPol(2,-5).consPol(4,3)
#    >>> ejPol1
#    3*x^4 + -5*x^2 + 3
#    >>> ejPol2 = Polinomio().consPol(1,4).consPol(2,5).consPol(5,1)
#    >>> ejPol2
#    x^5 + 5*x^2 + 4*x
#    >>> ejPol3 = Polinomio().consPol(1,2).consPol(4,6)
#    >>> ejPol3
#    6*x^4 + 2*x
#    >>> Polinomio().esPolCero()
#    True
#    >>> ejPol1.esPolCero()
#    False
```

```
#    >>> ejPol2
#    x^5 + 5*x^2 + 4*x
#    >>> ejPol2.consPol(3,0)
#    x^5 + 5*x^2 + 4*x
#    >>> Polinomio().consPol(3,2)
#    2*x^3
#    >>> ejPol2.consPol(6,7)
#    7*x^6 + x^5 + 5*x^2 + 4*x
#    >>> ejPol2.consPol(4,7)
#    x^5 + 7*x^4 + 5*x^2 + 4*x
#    >>> ejPol2.consPol(5,7)
#    8*x^5 + 5*x^2 + 4*x
#    >>> ejPol3
#    6*x^4 + 2*x
#    >>> ejPol3.grado()
#    4
#    >>> ejPol3.restoPol()
#    2*x
#    >>> ejPol2
#    x^5 + 5*x^2 + 4*x
#    >>> ejPol2.restoPol()
#    5*x^2 + 4*x
#
# Además se definen las correspondientes funciones. Por ejemplo,
#    >>> polCero()
#    0
#    >>> ejPol1a = consPol(4,3,consPol(2,-5,consPol(0,3,polCero())))
#    >>> ejPol1a
#    3*x^4 + -5*x^2 + 3
#    >>> ejPol2a = consPol(5,1,consPol(2,5,consPol(1,4,polCero())))
#    >>> ejPol2a
#    x^5 + 5*x^2 + 4*x
#    >>> ejPol3a = consPol(4,6,consPol(1,2,polCero()))
#    >>> ejPol3a
#    6*x^4 + 2*x
#    >>> esPolCero(polCero())
#    True
#    >>> esPolCero(ejPol1a)
#    False
#    >>> ejPol2a
```

```
#    x^5 + 5*x^2 + 4*x
#    >>> consPol(3,9,ejPol2a)
#    x^5 + 9*x^3 + 5*x^2 + 4*x
#    >>> consPol(3,2,polCero())
#    2*x^3
#    >>> consPol(6,7,ejPol2a)
#    7*x^6 + x^5 + 5*x^2 + 4*x
#    >>> consPol(4,7,ejPol2a)
#    x^5 + 7*x^4 + 5*x^2 + 4*x
#    >>> consPol(5,7,ejPol2a)
#    8*x^5 + 5*x^2 + 4*x
#    >>> ejPol3a
#    6*x^4 + 2*x
#    >>> grado(ejPol3a)
#    4
#    >>> restoPol(ejPol3a)
#    2*x
#    >>> ejPol2a
#    x^5 + 5*x^2 + 4*x
#    >>> restoPol(ejPol2a)
#    5*x^2 + 4*x
#
# Finalmente, se define un generador aleatorio de polinomios y se
# comprueba que los polinomios cumplen las propiedades de su
# especificación.

from __future__ import annotations

__all__ = [
    'Polinomio',
    'polCero',
    'esPolCero',
    'consPol',
    'grado',
    'coefLider',
    'restoPol',
    'polinomioAleatorio'
]

from dataclasses import dataclass, field
```

```python
from typing import Generic, TypeVar

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

A = TypeVar('A', int, float, complex)

# Clase de los polinomios mediante listas densas
# ==============================================

@dataclass
class Polinomio(Generic[A]):
    _terminos: list[tuple[int, A]] = field(default_factory=list)

    def esPolCero(self) -> bool:
        return not self._terminos

    def grado(self) -> int:
        if self.esPolCero():
            return 0
        return self._terminos[0][0]

    def coefLider(self) -> A:
        if self.esPolCero():
            return 0
        return self._terminos[0][1]

    def restoPol(self) -> Polinomio[A]:
        xs = self._terminos
        if len(xs) <= 1:
            return Polinomio([])
        return Polinomio(xs[1:])

    def consPol(self, n: int, b: A) -> Polinomio[A]:
        m = self.grado()
        c = self.coefLider()
        xs = self._terminos
        if b == 0:
            return self
        if self.esPolCero():
```

```
            return Polinomio([(n, b)])
        if n > m:
            return Polinomio([(n, b)] + xs)
        if n < m:
            return Polinomio(xs[1:]).consPol(n, b).consPol(m, c)
        if b + c == 0:
            return Polinomio(xs[1:])
        return Polinomio([(n, b + c)] + xs[1:])

    def __repr__(self) -> str:
        n = self.grado()
        a = self.coefLider()
        p = self.restoPol()
        if self.esPolCero():
            return "0"
        if n == 0 and p.esPolCero():
            return str(a)
        if n == 0:
            return str(a) + " + " + str(p)
        if n == 1 and p.esPolCero():
            return str(a) + "*x"
        if n == 1:
            return str(a) + "*x + " + str(p)
        if a == 1 and p.esPolCero():
            return "x^" + str(n)
        if p.esPolCero():
            return str(a) + "*x^" + str(n)
        if a == 1:
            return "x^" + str(n) + " + " + str(p)
        return str(a) + "*x^" + str(n) + " + " + str(p)

# Funciones del tipo polinomio
# ============================

def polCero() -> Polinomio[A]:
    return Polinomio([])

def esPolCero(p: Polinomio[A]) -> bool:
    return p.esPolCero()
```

```python
def grado(p: Polinomio[A]) -> int:
    return p.grado()

def coefLider(p: Polinomio[A]) -> A:
    return p.coefLider()

def restoPol(p: Polinomio[A]) -> Polinomio[A]:
    return p.restoPol()

def consPol(n: int, b: A, p: Polinomio[A]) -> Polinomio[A]:
    return p.consPol(n, b)

# Generador de polinomios
# =======================

# normal(ps) es la representación dispersa de un polinomio.
def normal(ps: list[tuple[int, A]]) -> list[tuple[int, A]]:
    xs = sorted(list({p[0] for p in ps}), reverse=True)
    ys = [p[1] for p in ps]
    return [(x, y) for (x, y) in zip(xs, ys) if y != 0]

# polinomioAleatorio() genera polinomios aleatorios. Por ejemplo,
#    >>> polinomioAleatorio().example()
#    -4*x^8 + -5*x^7 + -4*x^6 + -4*x^5 + -8*x^3
#    >>> polinomioAleatorio().example()
#    -7*x^9 + -8*x^6 + -8*x^3 + 2*x^2 + -1*x + 4
def polinomioAleatorio() -> st.SearchStrategy[Polinomio[int]]:
    return st.lists(st.tuples(st.integers(min_value=0, max_value=9),
                              st.integers(min_value=-9, max_value=9)))\
             .map(lambda ps: normal(ps))\
             .map(Polinomio)

# Comprobación de las propiedades de los polinomios
# =================================================

# Las propiedades son
def test_esPolCero1() -> None:
    assert esPolCero(polCero())

@given(p=polinomioAleatorio(),
```

```
       n=st.integers(min_value=0, max_value=10),
       b=st.integers())
def test_esPolCero2(p: Polinomio[int], n: int, b: int) -> None:
    assume(n > grado(p) and b != 0)
    assert not esPolCero(consPol(n, b, p))

@given(p=polinomioAleatorio())
def test_consPol(p: Polinomio[int]) -> None:
    assume(not esPolCero(p))
    assert consPol(grado(p), coefLider(p), restoPol(p)) == p

@given(p=polinomioAleatorio(),
       n=st.integers(min_value=0, max_value=10),
       b=st.integers())
def test_grado(p: Polinomio[int], n: int, b: int) -> None:
    assume(n > grado(p) and b != 0)
    assert grado(consPol(n, b, p)) == n

@given(p=polinomioAleatorio(),
       n=st.integers(min_value=0, max_value=10),
       b=st.integers())
def test_coefLider(p: Polinomio[int], n: int, b: int) -> None:
    assume(n > grado(p) and b != 0)
    assert coefLider(consPol(n, b, p)) == b

@given(p=polinomioAleatorio(),
       n=st.integers(min_value=0, max_value=10),
       b=st.integers())
def test_restoPol(p: Polinomio[int], n: int, b: int) -> None:
    assume(n > grado(p) and b != 0)
    assert restoPol(consPol(n, b, p)) == p

# La comprobación es
#    > poetry run pytest -v PolRepDispersa.py
#
#    PolRepDispersa.py::test_esPolCero1 PASSED
#    PolRepDispersa.py::test_esPolCero2 PASSED
#    PolRepDispersa.py::test_consPol PASSED
#    PolRepDispersa.py::test_grado PASSED
#    PolRepDispersa.py::test_coefLider PASSED
```

```
#    PolRepDispersa.py::test_restoPol PASSED
#
#    === 6 passed in 1.74s ===
```

10.5. Transformaciones entre las representaciones dispersa y densa

10.5.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Definir las funciones
--    densaAdispersa :: (Num a, Eq a) => [a] -> [(Int,a)]
--    dispersaAdensa :: (Num a, Eq a) => [(Int,a)] -> [a]
-- tales que
-- + (densaAdispersa xs) es la representación dispersa del polinomio
--   cuya representación densa es xs. Por ejemplo,
--      λ> densaAdispersa [9,0,0,5,0,4,7]
--      [(6,9),(3,5),(1,4),(0,7)]
-- + (dispersaAdensa ps) es la representación densa del polinomio
--   cuya representación dispersa es ps. Por ejemplo,
--      λ> dispersaAdensa [(6,9),(3,5),(1,4),(0,7)]
--      [9,0,0,5,0,4,7]
--
-- Comprobar con QuickCheck que las funciones densaAdispersa y
-- dispersaAdensa son inversas.
-- ---------------------------------------------------------------------

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

module Pol_Transformaciones_dispersa_y_densa where

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

-- 1ª definición de densaAdispersa
-- ===============================

densaAdispersa :: (Num a, Eq a) => [a] -> [(Int,a)]
densaAdispersa xs = [(m,a) | (m,a) <- zip [n-1,n-2..] xs, a /= 0]
```

```
  where n  = length xs

-- 2ª definición de densaAdispersa
-- ===============================

densaAdispersa2 :: (Num a, Eq a) => [a] -> [(Int,a)]
densaAdispersa2 xs = reverse (aux (reverse xs) 0)
  where aux [] _ = []
        aux (0:ys) n = aux ys (n+1)
        aux (y:ys) n = (n,y) : aux ys (n+1)

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

-- La propiedad es
prop_densaAdispersa :: [Int] -> Bool
prop_densaAdispersa xs =
  densaAdispersa xs == densaAdispersa2 xs

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

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

-- La comparación es
--    λ> densaAdispersa (5 : replicate (10^7) 0)
--    [(10000000,5)]
--    (4.54 secs, 3,280,572,504 bytes)
--    λ> densaAdispersa2 (5 : replicate (10^7) 0)
--    [(10000000,5)]
--    (7.35 secs, 3,696,968,576 bytes)

-- 1ª definición de dispersaAdensa
-- ===============================

dispersaAdensa :: (Num a, Eq a) => [(Int,a)] -> [a]
dispersaAdensa []      = []
dispersaAdensa [(n,a)] = a : replicate n 0
```

```
dispersaAdensa ((n,a):(m,b):ps) =
  a : replicate (n-m-1) 0 ++ dispersaAdensa ((m,b):ps)

-- 2ª definición de dispersaAdensa
-- ===============================

dispersaAdensa2 :: (Num a, Eq a) => [(Int,a)] -> [a]
dispersaAdensa2 []           = []
dispersaAdensa2 ps@((n,_):_) =
  [coeficiente ps m | m <- [n,n-1..0]]

-- (coeficiente ps n) es el coeficiente del término de grado n en el
-- polinomio cuya representación densa es ps. Por ejemplo,
--    coeficiente [(6,9),(3,5),(1,4),(0,7)] 3  ==  5
--    coeficiente [(6,9),(3,5),(1,4),(0,7)] 4  ==  0
coeficiente :: (Num a, Eq a) => [(Int,a)] -> Int -> a
coeficiente [] _                     = 0
coeficiente ((m,a):ps) n | n > m     = 0
                         | n == m    = a
                         | otherwise = coeficiente ps n

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

-- Tipo de las representaciones dispersas de polinomios.
newtype Dispersa = Dis [(Int,Int)]
  deriving Show

-- dispersaArbitraria es un generador de representaciones dispersas de
-- polinomios. Por ejemplo,
--    λ> sample dispersaArbitraria
--    Dis []
--    Dis []
--    Dis [(3,-2),(2,0),(0,3)]
--    Dis [(6,1),(4,-2),(3,4),(2,-4)]
--    Dis []
--    Dis [(5,-7)]
--    Dis [(12,5),(11,-8),(10,3),(8,-10),(7,-5),(4,12),(3,6),(2,-8),(1,11)]
--    Dis [(7,-2),(2,-8)]
--    Dis [(14,-15)]
```

```
--    Dis [(17,5),(16,1),(15,-1),(14,10),(13,5),(12,-15),(9,12),(6,14)]
--    Dis [(19,17),(12,7),(8,-3),(7,13),(5,-2),(4,7)]
dispersaArbitraria :: Gen Dispersa
dispersaArbitraria = do
  (xs, ys) <- arbitrary
  let xs' = nub (reverse (sort (map abs xs)))
      ys' = filter (/= 0) ys
  return (Dis (zip xs' ys'))

-- Dispersa está contenida en Arbitrary
instance Arbitrary Dispersa where
  arbitrary = dispersaArbitraria

-- La propiedad es
prop_dispersaAdensa :: Dispersa -> Bool
prop_dispersaAdensa (Dis xs) =
  dispersaAdensa xs == dispersaAdensa2 xs

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

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

-- La comparación es
--    λ> length (dispersaAdensa [(10^7,5)])
--    10000001
--    (0.11 secs, 560,566,848 bytes)
--    λ> length (dispersaAdensa2 [(10^7,5)])
--    10000001
--    (2.51 secs, 2,160,567,112 bytes)

-- Propiedad
-- =========

-- Tipo de las representaciones densas de polinomios.
newtype Densa = Den [Int]
  deriving Show
```

```
-- densaArbitraria es un generador de representaciones dispersas de
-- polinomios. Por ejemplo,
--    λ> sample densaArbitraria
--    Den []
--    Den []
--    Den []
--    Den [-6,6,5,-3]
--    Den []
--    Den [8,-7,-10,8,-10,-4,10,6,10]
--    Den [-6,2,11,-4,-9,-5,9,2,2,9]
--    Den [-6,9,-2]
--    Den [-1,-7,15,1,5,-2,13,16,8,7,2,16,-2,16,-7,4]
--    Den [8,13,-4,-2,-10,3,5,-4,-6,13,-9,-12,8,11,9,-18,12,10]
--    Den [-1,-2,11,17,-7,13,-12,-19,16,-10,-18,-19,1,-4,-17,10,1,10]
densaArbitraria :: Gen Densa
densaArbitraria = do
  ys <- arbitrary
  let ys' = dropWhile (== 0) ys
  return (Den ys')

-- Dispersa está contenida en Arbitrary
instance Arbitrary Densa where
  arbitrary = densaArbitraria

-- La primera propiedad es
prop_dispersaAdensa_densaAdispersa :: Densa -> Bool
prop_dispersaAdensa_densaAdispersa (Den xs) =
  dispersaAdensa (densaAdispersa xs) == xs

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

-- La segunda propiedad es
prop_densaAdispersa_dispersaAdensa :: Dispersa -> Bool
prop_densaAdispersa_dispersaAdensa (Dis ps) =
  densaAdispersa (dispersaAdensa ps) == ps

-- La comprobación es
--    λ> quickCheck prop_densaAdispersa_dispersaAdensa
```

```
--    +++ OK, passed 100 tests.
```

10.5.2. En Python

```
# ---------------------------------------------------------------------
# Definir las funciones
#    densaAdispersa : (list[A]) -> list[tuple[int, A]]
#    dispersaAdensa : (list[tuple[int, A]]) -> list[A]
# tales que
# + densaAdispersa(xs) es la representación dispersa del polinomio
#   cuya representación densa es xs. Por ejemplo,
#      >>> densaAdispersa([9, 0, 0, 5, 0, 4, 7])
#      [(6, 9), (3, 5), (1, 4), (0, 7)]
# + dispersaAdensa(ps) es la representación densa del polinomio
#   cuya representación dispersa es ps. Por ejemplo,
#      >>> dispersaAdensa([(6,9),(3,5),(1,4),(0,7)])
#      [9, 0, 0, 5, 0, 4, 7]
#
# Comprobar con Hypothesis que las funciones densaAdispersa y
# dispersaAdensa son inversas.
# ---------------------------------------------------------------------

from itertools import dropwhile
from typing import TypeVar

from hypothesis import given
from hypothesis import strategies as st

A = TypeVar('A', int, float, complex)

# 1ª definición de densaAdispersa
# ===============================

def densaAdispersa(xs: list[A]) -> list[tuple[int, A]]:
    n = len(xs)
    return [(m, a) for (m, a) in zip(range(n-1, -1, -1),  xs) if a != 0]

# 2ª definición de densaAdispersa
# ===============================
```

```
def densaAdispersa2(xs: list[A]) -> list[tuple[int, A]]:
    def aux(xs: list[A], n: int) -> list[tuple[int, A]]:
        if not xs:
            return []
        if xs[0] == 0:
            return aux(xs[1:], n + 1)
        return [(n, xs[0])] + aux(xs[1:], n + 1)

    return list(reversed(aux(list(reversed(xs)), 0)))

# 3ª definición de densaAdispersa
# ===============================

def densaAdispersa3(xs: list[A]) -> list[tuple[int, A]]:
    r = []
    n = len(xs) - 1
    for x in xs:
        if x != 0:
            r.append((n, x))
        n -= 1
    return r

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

# normalDensa(ps) es la representación dispersa de un polinomio.
def normalDensa(xs: list[A]) -> list[A]:
    return list(dropwhile(lambda x: x == 0, xs))

# densaAleatoria() genera representaciones densas de polinomios
# aleatorios. Por ejemplo,
#    >>> densaAleatoria().example()
#    [-5, 9, -6, -5, 7, -5, -1, 9]
#    >>> densaAleatoria().example()
#    [-4, 9, -3, -3, -5, 0, 6, -8, 8, 6, 0, -9]
#    >>> densaAleatoria().example()
#    [-3, -1, 2, 0, -9]
def densaAleatoria() -> st.SearchStrategy[list[int]]:
    return st.lists(st.integers(min_value=-9, max_value=9))\
             .map(normalDensa)
```

```
# La propiedad es
@given(xs=densaAleatoria())
def test_densaADispersa(xs: list[int]) -> None:
    r = densaAdispersa(xs)
    assert densaAdispersa2(xs) == r
    assert densaAdispersa3(xs) == r

# 1ª definición de dispersaAdensa
# ===============================

def dispersaAdensa(ps: list[tuple[int, A]]) -> list[A]:
    if not ps:
        return []
    if len(ps) == 1:
        return [ps[0][1]] + [0] * ps[0][0]
    (n, a) = ps[0]
    (m, _) = ps[1]
    return [a] + [0] * (n-m-1) + dispersaAdensa(ps[1:])

# 2ª definición de dispersaAdensa
# ===============================

# coeficiente(ps, n) es el coeficiente del término de grado n en el
# polinomio cuya representación densa es ps. Por ejemplo,
#    coeficiente([(6, 9), (3, 5), (1, 4), (0, 7)], 3)  ==  5
#    coeficiente([(6, 9), (3, 5), (1, 4), (0, 7)], 4)  ==  0
def coeficiente(ps: list[tuple[int, A]], n: int) -> A:
    if not ps:
        return 0
    (m, a) = ps[0]
    if n > m:
        return 0
    if n == m:
        return a
    return coeficiente(ps[1:], n)

def dispersaAdensa2(ps: list[tuple[int, A]]) -> list[A]:
    if not ps:
        return []
```

```
    n = ps[0][0]
    return [coeficiente(ps, m) for m in range(n, -1, -1)]

# 3ª definición de dispersaAdensa
# ================================

def dispersaAdensa3(ps: list[tuple[int, A]]) -> list[A]:
    if not ps:
        return []
    n = ps[0][0]
    r: list[A] = [0] * (n + 1)
    for (m, a) in ps:
        r[n-m] = a
    return r

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

# normalDispersa(ps) es la representación dispersa de un polinomio.
def normalDispersa(ps: list[tuple[int, A]]) -> list[tuple[int, A]]:
    xs = sorted(list({p[0] for p in ps}), reverse=True)
    ys = [p[1] for p in ps]
    return [(x, y) for (x, y) in zip(xs, ys) if y != 0]

# dispersaAleatoria() genera representaciones densas de polinomios
# aleatorios. Por ejemplo,
#    >>> dispersaAleatoria().example()
#    [(5, -6), (2, -1), (0, 2)]
#    >>> dispersaAleatoria().example()
#    [(6, -7)]
#    >>> dispersaAleatoria().example()
#    [(7, 2), (4, 9), (3, 3), (0, -2)]
def dispersaAleatoria() -> st.SearchStrategy[list[tuple[int, int]]]:
    return st.lists(st.tuples(st.integers(min_value=0, max_value=9),
                              st.integers(min_value=-9, max_value=9)))\
             .map(normalDispersa)

# La propiedad es
@given(ps=dispersaAleatoria())
def test_dispersaAdensa(ps: list[tuple[int, int]]) -> None:
```

```
    r = dispersaAdensa(ps)
    assert dispersaAdensa2(ps) == r
    assert dispersaAdensa3(ps) == r

# Propiedad
# =========

# La primera propiedad es
@given(xs=densaAleatoria())
def test_dispersaAdensa_densaAdispersa(xs: list[int]) -> None:
    assert dispersaAdensa(densaAdispersa(xs)) == xs

# La segunda propiedad es
@given(ps=dispersaAleatoria())
def test_densaAdispersa_dispersaAdensa(ps: list[tuple[int, int]]) -> None:
    assert densaAdispersa(dispersaAdensa(ps)) == ps

# La comprobación es
#    > poetry run pytest -v Pol_Transformaciones_dispersa_y_densa.py
#    test_densaADispersa PASSED
#    test_dispersaAdensa PASSED
#    test_dispersaAdensa_densaAdispersa PASSED
#    test_densaAdispersa_dispersaAdensa PASSED
```

10.6. Transformaciones entre polinomios y listas dispersas

10.6.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Usando el [tipo abstracto de datos de los polinomios](https://bit.ly/3KwqXYu)
-- definir las funciones
--    dispersaApolinomio :: (Num a, Eq a) => [(Int,a)] -> Polinomio a
--    polinomioAdispersa :: (Num a, Eq a) => Polinomio a -> [(Int,a)]
-- tales que
-- + (dispersaApolinomio ps) es el polinomiocuya representación dispersa
--   es ps. Por ejemplo,
--      λ> dispersaApolinomio [(6,9),(3,5),(1,4),(0,7)]
--      9*x^6 + 5*x^3 + 4*x + 7
```

```
-- + (polinomioAdispersa p) es la representación dispersa del polinomio
--   p. Por ejemplo,
--      λ> ejPol = consPol 6 9 (consPol 3 5 (consPol 1 4 (consPol 0 7 polCero)))
--      λ> ejPol
--      9*x^6 + 5*x^3 + 4*x + 7
--      λ> polinomioAdispersa ejPol
--      [(6,9),(3,5),(1,4),(0,7)]
--
-- Comprobar con QuickCheck que ambas funciones son inversas.
-- ---------------------------------------------------------------------

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

module Pol_Transformaciones_polinomios_dispersas where

import TAD.Polinomio (Polinomio, polCero, esPolCero, consPol, grado,
                      coefLider, restoPol)
import Data.List (sort, nub)
import Test.QuickCheck

-- 1ª definición de dispersaApolinomio
-- ===================================

dispersaApolinomio :: (Num a, Eq a) => [(Int,a)] -> Polinomio a
dispersaApolinomio []         = polCero
dispersaApolinomio ((n,a):ps) = consPol n a (dispersaApolinomio ps)

-- 2ª definición de dispersaApolinomio
-- ===================================

dispersaApolinomio2 :: (Num a, Eq a) => [(Int,a)] -> Polinomio a
dispersaApolinomio2 = foldr (\(x,y) -> consPol x y) polCero


-- 3ª definición de dispersaApolinomio
-- ===================================

dispersaApolinomio3 :: (Num a, Eq a) => [(Int,a)] -> Polinomio a
dispersaApolinomio3 = foldr (uncurry consPol) polCero
```

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

-- Tipo de las representaciones dispersas de polinomios.
newtype Dispersa = Dis [(Int,Int)]
  deriving Show

-- dispersaArbitraria es un generador de representaciones dispersas de
-- polinomios. Por ejemplo,
--    λ> sample dispersaArbitraria
--    Dis []
--    Dis []
--    Dis [(3,-2),(2,0),(0,3)]
--    Dis [(6,1),(4,-2),(3,4),(2,-4)]
--    Dis []
--    Dis [(5,-7)]
--    Dis [(12,5),(11,-8),(10,3),(8,-10),(7,-5),(4,12),(3,6),(2,-8),(1,11)]
--    Dis [(7,-2),(2,-8)]
--    Dis [(14,-15)]
--    Dis [(17,5),(16,1),(15,-1),(14,10),(13,5),(12,-15),(9,12),(6,14)]
--    Dis [(19,17),(12,7),(8,-3),(7,13),(5,-2),(4,7)]
dispersaArbitraria :: Gen Dispersa
dispersaArbitraria = do
  (xs, ys) <- arbitrary
  let xs' = nub (reverse (sort (map abs xs)))
      ys' = filter (/= 0) ys
  return (Dis (zip xs' ys'))

-- Dispersa está contenida en Arbitrary
instance Arbitrary Dispersa where
  arbitrary = dispersaArbitraria

-- La propiedad es
prop_dispersaApolinomio :: Dispersa -> Bool
prop_dispersaApolinomio (Dis ps) =
  all (== dispersaApolinomio ps)
      [dispersaApolinomio2 ps,
       dispersaApolinomio3 ps]

-- Definición de polinomioAdispersa
```

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

polinomioAdispersa :: (Num a, Eq a) => Polinomio a -> [(Int,a)]
polinomioAdispersa p
  | esPolCero p = []
  | otherwise   = (grado p, coefLider p) : polinomioAdispersa (restoPol p)

-- Propiedad de ser inversas
-- =========================

-- La primera propiedad es
prop_polinomioAdispersa_dispersaApolinomio :: Dispersa -> Bool
prop_polinomioAdispersa_dispersaApolinomio (Dis ps) =
  polinomioAdispersa (dispersaApolinomio ps) == ps

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

-- La segunda propiedad es
prop_dispersaApolinomio_polinomioAdispersa :: Polinomio Int -> Bool
prop_dispersaApolinomio_polinomioAdispersa p =
  dispersaApolinomio (polinomioAdispersa p) == p

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

10.6.2. En Python

```
# ---------------------------------------------------------------------
# Utilizando el [tipo abstracto de datos de los polinomios]
# (https://bit.ly/3KwqXYu) definir las funciones
#    dispersaApolinomio : (list[tuple[int, A]]) -> Polinomio[A]
#    polinomioAdispersa : (Polinomio[A]) -> list[tuple[int, A]]
# tales que
# + dispersaApolinomio(ps) es el polinomiocuya representación dispersa
#   es ps. Por ejemplo,
#      >>> dispersaApolinomio([(6, 9), (3, 5), (1, 4), (0, 7)])
#      9*x^6 + 5*x^3 + 4*x + 7
```

```
# + polinomioAdispersa(p) es la representación dispersa del polinomio
#   p. Por ejemplo,
#      >>> ejPol1 = consPol(3, 5, consPol(1, 4, consPol(0, 7, polCero())))
#      >>> ejPol = consPol(6, 9, ejPol1)
#      >>> ejPol
#      9*x^6 + 5*x^3 + 4*x + 7
#      >>> polinomioAdispersa(ejPol)
#      [(6, 9), (3, 5), (1, 4), (0, 7)]
#
# Comprobar con Hypothesis que ambas funciones son inversas.
# ---------------------------------------------------------------------

from typing import TypeVar

from hypothesis import given

from src.Pol_Transformaciones_dispersa_y_densa import dispersaAleatoria
from src.TAD.Polinomio import (Polinomio, coefLider, consPol, esPolCero, grado,
                               polCero, polinomioAleatorio, restoPol)

A = TypeVar('A', int, float, complex)


# 1ª definición de dispersaApolinomio
# ===================================

def dispersaApolinomio(ps: list[tuple[int, A]]) -> Polinomio[A]:
    if not ps:
        return polCero()
    (n, a) = ps[0]
    return consPol(n, a, dispersaApolinomio(ps[1:]))

# 2ª definición de dispersaApolinomio
# ===================================

def dispersaApolinomio2(ps: list[tuple[int, A]]) -> Polinomio[A]:
    r: Polinomio[A] = polCero()
    for (n, a) in reversed(ps):
        r = consPol(n, a, r)
    return r
```

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

# La propiedad es
@given(ps=dispersaAleatoria())
def test_dispersaApolinomio(ps: list[tuple[int, int]]) -> None:
    assert dispersaApolinomio(ps) == dispersaApolinomio2(ps)

# El generador dispersaAleatoria está definido en el ejercicio
# "Transformaciones entre las representaciones dispersa y densa" que se
# encuentra en https://bit.ly/402UpuT

# Definición de polinomioAdispersa
# ================================

def polinomioAdispersa(p: Polinomio[A]) -> list[tuple[int, A]]:
    if esPolCero(p):
        return []
    return [(grado(p), coefLider(p))] + polinomioAdispersa(restoPol(p))

# Propiedad de ser inversas
# =========================

# La primera propiedad es
@given(ps=dispersaAleatoria())
def test_polinomioAdispersa_dispersaApolinomio(ps: list[tuple[int,
                                                              int]]) -> None:
    assert polinomioAdispersa(dispersaApolinomio(ps)) == ps

# La segunda propiedad es
@given(p=polinomioAleatorio())
def test_dispersaApolinomio_polinomioAdispersa(p: Polinomio[int]) -> None:
    assert dispersaApolinomio(polinomioAdispersa(p)) == p

# La comprobación es
#    > poetry run pytest -v Pol_Transformaciones_polinomios_dispersas.py
#    test_dispersaApolinomio PASSED
#    test_polinomioAdispersa_dispersaApolinomio PASSED
#    test_dispersaApolinomio_polinomioAdispersa PASSED
```

10.7. Coeficiente del término de grado k

10.7.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Usando el [tipo abstracto de datos de los polinomios](https://bit.ly/3KwqXYu)
-- definir la función
--    coeficiente :: (Num a, Eq a) => Int -> Polinomio a -> a
-- tal que (coeficiente k p) es el coeficiente del término de grado k
-- del polinomio p. Por ejemplo,
--    λ> ejPol = consPol 5 1 (consPol 2 5 (consPol 1 4 polCero))
--    λ> ejPol
--    x^5 + 5*x^2 + 4*x
--    λ> coeficiente 2 ejPol
--    5
--    λ> coeficiente 3 ejPol
--    0
-- ---------------------------------------------------------------

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

module Pol_Coeficiente where

import TAD.Polinomio (Polinomio, coefLider, grado, restoPol,
                      consPol, polCero)

coeficiente :: (Num a, Eq a) => Int -> Polinomio a -> a
coeficiente k p | k == n                 = coefLider p
                | k > grado (restoPol p) = 0
                | otherwise              = coeficiente k (restoPol p)
  where n = grado p
```

10.7.2. En Python

```python
# ---------------------------------------------------------------------
# Utilizando el [tipo abstracto de datos de los polinomios]
# (https://bit.ly/3KwqXYu) definir la función
#    coeficiente : (int, Polinomio[A]) -> A
# tal que coeficiente(k, p) es el coeficiente del término de grado k
# del polinomio p. Por ejemplo,
```

```
#    >>> ejPol = consPol(5, 1, consPol(2, 5, consPol(1, 4, polCero())))
#    >>> ejPol
#    x^5 + 5*x^2 + 4*x
#    >>> coeficiente(2, ejPol)
#    5
#    >>> coeficiente(3, ejPol)
#    0
# ---------------------------------------------------------------------

# pylint: disable=unused-import

from typing import TypeVar

from src.TAD.Polinomio import (Polinomio, coefLider, consPol, grado, polCero,
                               restoPol)

A = TypeVar('A', int, float, complex)

def coeficiente(k: int, p: Polinomio[A]) -> A:
    if k == grado(p):
        return coefLider(p)
    if k > grado(restoPol(p)):
        return 0
    return coeficiente(k, restoPol(p))
```

10.8. Transformaciones entre polinomios y listas densas

10.8.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Utilizando el [tipo abstracto de datos de los polinomios]
-- (https://bit.ly/3KwqXYu) definir las funciones
--    densaApolinomio :: (Num a, Eq a) => [a] -> Polinomio a
--    polinomioAdensa :: (Num a, Eq a) => Polinomio a -> [a]
-- tales que
-- + (densaApolinomio xs) es el polinomio cuya representación densa es
--   xs. Por ejemplo,
--      λ> densaApolinomio [9,0,0,5,0,4,7]
```

```haskell
--      9*x^6 + 5*x^3 + 4*x + 7
-- + (polinomioAdensa c) es la representación densa del polinomio p. Por
--   ejemplo,
--      λ> ejPol = consPol 6 9 (consPol 3 5 (consPol 1 4 (consPol 0 7 polCero)))
--      λ> ejPol
--      9*x^6 + 5*x^3 + 4*x + 7
--      λ> polinomioAdensa ejPol
--      [9,0,0,5,0,4,7]
--
-- Comprobar con QuickCheck que ambas funciones son inversas.
-- ---------------------------------------------------------------------

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

module Pol_Transformaciones_polinomios_densas where

import TAD.Polinomio (Polinomio, polCero, esPolCero, consPol, grado,
                      coefLider, restoPol)
import Pol_Transformaciones_dispersa_y_densa (densaAdispersa,
                                              dispersaAdensa)
import Pol_Transformaciones_polinomios_dispersas (dispersaApolinomio,
                                                  polinomioAdispersa)
import Pol_Coeficiente (coeficiente)
import Data.List (sort, nub)
import Test.QuickCheck

-- 1ª definición de densaApolinomio
-- ================================

densaApolinomio :: (Num a, Eq a) => [a] -> Polinomio a
densaApolinomio []     = polCero
densaApolinomio (x:xs) = consPol (length xs) x (densaApolinomio xs)

-- 2ª definición de densaApolinomio
-- ================================

densaApolinomio2 :: (Num a, Eq a) => [a] -> Polinomio a
densaApolinomio2 = dispersaApolinomio . densaAdispersa

-- La función densaAdispersa está definida en el ejercicio
```

```
-- ”Transformaciones entre las representaciones dispersa y densa” que se
-- encuentra en https://bit.ly/3GTyIqe

-- La función dispersaApolinomio se encuentra en el ejercicio
-- ”Transformaciones entre polinomios y listas dispersas” que se
-- encuentra en https://bit.ly/41GgQaB

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

-- La propiedad es
prop_densaApolinomio :: [Int] -> Bool
prop_densaApolinomio xs =
  densaApolinomio xs == densaApolinomio2 xs

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

-- 1ª definición de polinomioAdensa
-- ================================

polinomioAdensa :: (Num a, Eq a) => Polinomio a -> [a]
polinomioAdensa p
  | esPolCero p = []
  | otherwise   = [coeficiente k p | k <- [n,n-1..0]]
  where n = grado p

-- La función coeficiente está definida en el ejercicio
-- ”Coeficiente del término de grado k” que se encuentra en
-- https://bit.ly/413l3oQ

-- 2ª definición de polinomioAdensa
-- ================================

polinomioAdensa2 :: (Num a, Eq a) => Polinomio a -> [a]
polinomioAdensa2 = dispersaAdensa . polinomioAdispersa

-- La función dispersaAdensa está definida en el ejercicio
-- ”Transformaciones entre las representaciones dispersa y densa” que se
```

```haskell
-- encuentra en https://bit.ly/3GTyIqe

-- La función polinomioAdispersa se encuentra en el ejercicio
-- "Transformaciones entre polinomios y listas dispersas" que se
-- encuentra en https://bit.ly/41GgQaB

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

-- La propiedad es
prop_polinomioAdensa :: Polinomio Int -> Bool
prop_polinomioAdensa p =
  polinomioAdensa p == polinomioAdensa2 p

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

-- Propiedades de inversa
-- ======================

-- La primera propiedad es
prop_polinomioAdensa_densaApolinomio :: [Int] -> Bool
prop_polinomioAdensa_densaApolinomio xs =
  polinomioAdensa (densaApolinomio xs') == xs'
  where xs' = dropWhile (== 0) xs

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

-- La segunda propiedad es
prop_densaApolinomio_polinomioAdensa :: Polinomio Int -> Bool
prop_densaApolinomio_polinomioAdensa p =
   densaApolinomio (polinomioAdensa p) == p

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

10.8.2. En Python

```
# ---------------------------------------------------------------------
# Utilizando el [tipo abstracto de datos de los polinomios]
# (https://bit.ly/3KwqXYu) definir las funciones
#    densaApolinomio : (list[A]) -> Polinomio[A]
#    polinomioAdensa : (Polinomio[A]) -> list[A]
# tales que
# + densaApolinomio(xs) es el polinomio cuya representación densa es
#   xs. Por ejemplo,
#      >>> densaApolinomio([9, 0, 0, 5, 0, 4, 7])
#      9*x^6 + 5*x^3 + 4*x + 7
# + polinomioAdensa(c) es la representación densa del polinomio p. Por
#   ejemplo,
#      >>> ejPol = consPol(6, 9, consPol(3, 5, consPol(1, 4, consPol(0, 7, polCer
#      >>> ejPol
#      9*x^6 + 5*x^3 + 4*x + 7
#      >>> polinomioAdensa(ejPol)
#      [9, 0, 0, 5, 0, 4, 7]
#
# Comprobar con Hypothesis que ambas funciones son inversas.
# ---------------------------------------------------------------------

# pylint: disable=unused-import

from typing import TypeVar

from hypothesis import given

from src.Pol_Coeficiente import coeficiente
from src.Pol_Transformaciones_dispersa_y_densa import (densaAdispersa,
                                                       densaAleatoria,
                                                       dispersaAdensa)
from src.Pol_Transformaciones_polinomios_dispersas import (dispersaApolinomio,
                                                           polinomioAdispersa)
from src.TAD.Polinomio import (Polinomio, coefLider, consPol, esPolCero, grado,
                               polCero, polinomioAleatorio, restoPol)

A = TypeVar('A', int, float, complex)

# 1ª definición de densaApolinomio
```

```
# ================================

def densaApolinomio(xs: list[A]) -> Polinomio[A]:
    if not xs:
        return polCero()
    return consPol(len(xs[1:]), xs[0], densaApolinomio(xs[1:]))

# 2ª definición de densaApolinomio
# ================================

def densaApolinomio2(xs: list[A]) -> Polinomio[A]:
    return dispersaApolinomio(densaAdispersa(xs))

# La función densaAdispersa está definida en el ejercicio
# ”Transformaciones entre las representaciones dispersa y densa” que se
# encuentra en https://bit.ly/3GTyIqe

# La función dispersaApolinomio se encuentra en el ejercicio
# ”Transformaciones entre polinomios y listas dispersas” que se
# encuentra en https://bit.ly/41GgQaB

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

# La propiedad es
@given(xs=densaAleatoria())
def test_densaApolinomio(xs: list[int]) -> None:
    assert densaApolinomio(xs) == densaApolinomio2(xs)

# La función densaAleatoria está definida en el ejercicio
# ”Transformaciones entre las representaciones dispersa y densa” que se
# encuentra en https://bit.ly/3GTyIqe

# 1ª definición de polinomioAdensa
# ================================

def polinomioAdensa(p: Polinomio[A]) -> list[A]:
    if esPolCero(p):
        return []
    n = grado(p)
```

```python
    return [coeficiente(k, p) for k in range(n, -1, -1)]

# La función coeficiente está definida en el ejercicio
# "Coeficiente del término de grado k" que se encuentra en
# https://bit.ly/413l3oQ

# 2ª definición de polinomioAdensa
# ================================

def polinomioAdensa2(p: Polinomio[A]) -> list[A]:
    return dispersaAdensa(polinomioAdispersa(p))

# La función dispersaAdensa está definida en el ejercicio
# "Transformaciones entre las representaciones dispersa y densa" que se
# encuentra en https://bit.ly/3GTyIqe

# La función polinomioAdispersa se encuentra en el ejercicio
# "Transformaciones entre polinomios y listas dispersas" que se
# encuentra en https://bit.ly/41GgQaB

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

# La propiedad es
@given(p=polinomioAleatorio())
def test_polinomioAdensa(p: Polinomio[int]) -> None:
    assert polinomioAdensa(p) == polinomioAdensa2(p)

# Propiedades de inversa
# ======================

# La primera propiedad es
@given(xs=densaAleatoria())
def test_polinomioAdensa_densaApolinomio(xs: list[int]) -> None:
    assert polinomioAdensa(densaApolinomio(xs)) == xs

# La segunda propiedad es
@given(p=polinomioAleatorio())
def test_densaApolinomio_polinomioAdensa(p: Polinomio[int]) -> None:
    assert densaApolinomio(polinomioAdensa(p)) == p
```

```
# La comprobación es
#    > poetry run pytest -v Pol_Transformaciones_polinomios_densas.py
#    test_densaApolinomio PASSED
#    test_polinomioAdensa PASSED
#    test_polinomioAdensa_densaApolinomio PASSED
#    test_densaApolinomio_polinomioAdensa PASSED
```

10.9. Construcción de términos

10.9.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Usando el [tipo abstracto de los polinomios](https://bit.ly/3KwqXYu),
-- definir la función
--    creaTermino :: (Num a, Eq a) => Int -> a -> Polinomio a
-- tal que (creaTermino n a) es el término a*x^n. Por ejemplo,
--    creaTermino 2 5  ==  5*x^2
-- ---------------------------------------------------------------------

module Pol_Crea_termino where

import TAD.Polinomio (Polinomio, polCero, consPol)

creaTermino :: (Num a, Eq a) => Int -> a -> Polinomio a
creaTermino n a = consPol n a polCero
```

10.9.2. En Python

```
# ---------------------------------------------------------------------
# Usando el [tipo abstracto de los polinomios](https://bit.ly/3KwqXYu),
# definir la función
#    creaTermino : (int, A) -> Polinomio[A]
# tal que creaTermino(n, a) es el término a*x^n. Por ejemplo,
#    >>> creaTermino(2, 5)
#    5*x^2
# ---------------------------------------------------------------------

from typing import TypeVar
```

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

from src.TAD.Polinomio import Polinomio, consPol, polCero

A = TypeVar('A', int, float, complex)

# 1ª solución
# ===========

def creaTermino(n: int, a: A) -> Polinomio[A]:
    return consPol(n, a, polCero())

# 2ª solución
# ===========

def creaTermino2(n: int, a: A) -> Polinomio[A]:
    r: Polinomio[A] = polCero()
    return r.consPol(n, a)

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

# La propiedad es
@given(st.integers(min_value=0, max_value=9),
       st.integers(min_value=-9, max_value=9))
def test_creaTermino(n: int, a: int) -> None:
    assert creaTermino(n, a) == creaTermino2(n, a)

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

10.10. Término líder de un polinomio

10.10.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Usando el [tipo abstracto de los polinomios](https://bit.ly/3KwqXYu),
```

```
-- definir la función
--    termLider :: (Num a, Eq a) => Polinomio a -> Polinomio a
-- tal que (termLider p) es el término líder del polinomio p. Por
-- ejemplo,
--    λ> ejPol = consPol 5 1 (consPol 2 5 (consPol 1 4 polCero))
--    λ> ejPol
--    x^5 + 5*x^2 + 4*x
--    λ> termLider ejPol
--    x^5
-- ---------------------------------------------------------------------

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

module Pol_Termino_lider where

import TAD.Polinomio (Polinomio, coefLider, grado, polCero, consPol)
import Pol_Crea_termino (creaTermino)

termLider :: (Num a, Eq a) => Polinomio a -> Polinomio a
termLider p = creaTermino (grado p) (coefLider p)

-- La función creaTermino está definida en el ejercicio
-- "Construcción de términos" que se encuentra en
-- https://bit.ly/3GXteuH
```

10.10.2. En Python

```
# ---------------------------------------------------------------------
# Usando el [tipo abstracto de los polinomios](https://bit.ly/3KwqXYu),
# definir la función
#    termLider : (Polinomio[A]) -> Polinomio[A]
# tal que termLider(p) es el término líder del polinomio p. Por
# ejemplo,
#    >>> ejPol = consPol(5, 1, consPol(2, 5, consPol(1, 4, polCero())))
#    >>> ejPol
#    x^5 + 5*x^2 + 4*x
#    >>> termLider(ejPol)
#    x^5
# ---------------------------------------------------------------------
```

```python
# pylint: disable=unused-import

from typing import TypeVar

from hypothesis import given

from src.Pol_Crea_termino import creaTermino
from src.TAD.Polinomio import (Polinomio, coefLider, consPol, grado, polCero,
                               polinomioAleatorio)

A = TypeVar('A', int, float, complex)

# 1ª solución
# ===========

def termLider(p: Polinomio[A]) -> Polinomio[A]:
    return creaTermino(grado(p), coefLider(p))

# 2ª solución
# ===========

def termLider2(p: Polinomio[A]) -> Polinomio[A]:
    return creaTermino(p.grado(), p.coefLider())

# La función creaTermino está definida en el ejercicio
# "Construcción de términos" que se encuentra en
# https://bit.ly/3GXteuH

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

# La propiedad es
@given(p=polinomioAleatorio())
def test_termLider(p: Polinomio[int]) -> None:
    assert termLider(p) == termLider2(p)

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

10.11. Suma de polinomios

10.11.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Usando el [tipo abstracto de los polinomios](https://bit.ly/3KwqXYu),
-- definir la función
--    sumaPol :: (Num a, Eq a) => Polinomio a -> Polinomio a -> Polinomio a
-- tal que (sumaPol p q) es la suma de los polinomios p y q. Por ejemplo,
--    λ> ejPol1 = consPol 4 3 (consPol 2 (-5) (consPol 0 3 polCero))
--    λ> ejPol2 = consPol 5 1 (consPol 2 5 (consPol 1 4 polCero))
--    λ> ejPol1
--    3*x^4 + -5*x^2 + 3
--    λ> ejPol2
--    x^5 + 5*x^2 + 4*x
--    λ> sumaPol ejPol1 ejPol2
--    x^5 + 3*x^4 + 4*x + 3
--
-- Comprobar con QuickCheck las siguientes propiedades:
-- + polCero es el elemento neutro de la suma.
-- + la suma es conmutativa.
-- ---------------------------------------------------------------------

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

module Pol_Suma_de_polinomios where

import TAD.Polinomio (Polinomio, polCero, esPolCero, consPol, grado,
                      coefLider, restoPol)
import Test.QuickCheck

sumaPol :: (Num a, Eq a) => Polinomio a -> Polinomio a -> Polinomio a
sumaPol p q
  | esPolCero p = q
  | esPolCero q = p
  | n1 > n2      = consPol n1 a1 (sumaPol r1 q)
  | n1 < n2      = consPol n2 a2 (sumaPol p r2)
  | otherwise    = consPol n1 (a1+a2) (sumaPol r1 r2)
  where (n1, a1, r1) = (grado p, coefLider p, restoPol p)
        (n2, a2, r2) = (grado q, coefLider q, restoPol q)
```

```
-- Propiedad. El polinomio cero es el elemento neutro de la suma.
prop_neutroSumaPol :: Polinomio Int -> Bool
prop_neutroSumaPol p =
  sumaPol polCero p == p

-- Comprobación con QuickCheck.
--    λ> quickCheck prop_neutroSumaPol
--    OK, passed 100 tests.

-- Propiedad. La suma es conmutativa.
prop_conmutativaSuma :: Polinomio Int -> Polinomio Int -> Bool
prop_conmutativaSuma p q =
  sumaPol p q == sumaPol q p

-- Comprobación:
--    λ> quickCheck prop_conmutativaSuma
--    OK, passed 100 tests.
```

10.11.2. En Python

```
# ---------------------------------------------------------------------
# Usando el [tipo abstracto de los polinomios](https://bit.ly/3KwqXYu),
# definir la función
#    sumaPol : (Polinomio[A], Polinomio[A]) -> Polinomio[A]
# tal que sumaPol(p, q) es la suma de los polinomios p y q. Por ejemplo,
#    >>> ejPol1 = consPol(4, 3, consPol(2, -5, consPol(0, 3, polCero())))
#    >>> ejPol2 = consPol(5, 1, consPol(2, 5, consPol(1, 4, polCero())))
#    >>> ejPol1
#    3*x^4 + -5*x^2 + 3
#    >>> ejPol2
#    x^5 + 5*x^2 + 4*x
#    >>> sumaPol(ejPol1, ejPol2)
#    x^5 + 3*x^4 + 4*x + 3
#
# Comprobar con Hypothesis las siguientes propiedades:
# + polCero es el elemento neutro de la suma.
# + la suma es conmutativa.
# ---------------------------------------------------------------------

# pylint: disable=arguments-out-of-order
```

```python
from typing import TypeVar

from hypothesis import given

from src.TAD.Polinomio import (Polinomio, coefLider, consPol, esPolCero, grado,
                               polCero, polinomioAleatorio, restoPol)

A = TypeVar('A', int, float, complex)

# 1ª solución
# ===========

def sumaPol(p: Polinomio[A], q: Polinomio[A]) -> Polinomio[A]:
    if esPolCero(p):
        return q
    if esPolCero(q):
        return p
    n1, a1, r1 = grado(p), coefLider(p), restoPol(p)
    n2, a2, r2 = grado(q), coefLider(q), restoPol(q)
    if n1 > n2:
        return consPol(n1, a1, sumaPol(r1, q))
    if n1 < n2:
        return consPol(n2, a2, sumaPol(p, r2))
    return consPol(n1, a1 + a2, sumaPol(r1, r2))

# 2ª solución
# ===========

def sumaPol2(p: Polinomio[A], q: Polinomio[A]) -> Polinomio[A]:
    if p.esPolCero():
        return q
    if q.esPolCero():
        return p
    n1, a1, r1 = p.grado(), p.coefLider(), p.restoPol()
    n2, a2, r2 = q.grado(), q.coefLider(), q.restoPol()
    if n1 > n2:
        return sumaPol(r1, q).consPol(n1, a1)
    if n1 < n2:
        return sumaPol(p, r2).consPol(n2, a2)
```

```
    return sumaPol(r1, r2).consPol(n1, a1 + a2)

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

# La propiedad es
@given(p=polinomioAleatorio(), q=polinomioAleatorio())
def test_sumaPol(p: Polinomio[int], q: Polinomio[int]) -> None:
    assert sumaPol(p, q) == sumaPol2(p,q)

# El polinomio cero es el elemento neutro de la suma.
@given(p=polinomioAleatorio())
def test_neutroSumaPol(p: Polinomio[int]) -> None:
    assert sumaPol(polCero(), p) == p
    assert sumaPol(p, polCero()) == p

# La suma es conmutativa.
@given(p=polinomioAleatorio(), q=polinomioAleatorio())
def test_conmutativaSuma(p: Polinomio[int], q: Polinomio[int]) -> None:
    assert sumaPol(p, q) == sumaPol(q, p)

# La comprobación es
#    > poetry run pytest -v Pol_Suma_de_polinomios.py
#    test_sumaPol PASSED
#    test_neutroSumaPol PASSED
#    test_conmutativaSuma PASSED
```

10.12. Producto de polinomios

10.12.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Usando el [tipo abstracto de los polinomios](https://bit.ly/3KwqXYu),
-- definir la función
--    multPol :: (Num a, Eq a) => Polinomio a -> Polinomio a -> Polinomio a
-- tal que (multPol p q) es el producto de los polinomios p y q. Por
-- ejemplo,
--    λ> ejPol1 = consPol 4 3 (consPol 2 (-5) (consPol 0 3 polCero))
--    λ> ejPol2 = consPol 5 1 (consPol 2 5 (consPol 1 4 polCero))
--    λ> ejPol1
```

```haskell
--    3*x^4 + -5*x^2 + 3
--    λ> ejPol2
--    x^5 + 5*x^2 + 4*x
--    λ> multPol ejPol1 ejPol2
--    3*x^9 + -5*x^7 + 15*x^6 + 15*x^5 + -25*x^4 + -20*x^3 + 15*x^2 + 12*x
--
-- Comprobar con QuickCheck las siguientes propiedades
-- + El producto de polinomios es conmutativo.
-- + El producto es distributivo respecto de la suma.
-- ---------------------------------------------------------------------

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

module Pol_Producto_polinomios where

import TAD.Polinomio (Polinomio, polCero, esPolCero, consPol, grado,
                      coefLider, restoPol)
import Pol_Termino_lider (termLider)
import Pol_Suma_de_polinomios (sumaPol)
import Test.QuickCheck

multPol :: (Num a, Eq a) => Polinomio a -> Polinomio a -> Polinomio a
multPol p q
  | esPolCero p = polCero
  | otherwise    = sumaPol (multPorTerm (termLider p) q)
                           (multPol (restoPol p) q)

-- (multPorTerm t p) es el producto del término t por el polinomio
-- p. Por ejemplo,
--    ejTerm                   ==  4*x
--    ejPol2                   ==  x^5 + 5*x^2 + 4*x
--    multPorTerm ejTerm ejPol2  ==  4*x^6 + 20*x^3 + 16*x^2
multPorTerm :: (Num t, Eq t) => Polinomio t -> Polinomio t -> Polinomio t
multPorTerm term pol
  | esPolCero pol = polCero
  | otherwise     = consPol (n+m) (a*b) (multPorTerm term r)
  where n = grado term
        a = coefLider term
        m = grado pol
        b = coefLider pol
```

```
        r = restoPol pol

-- El producto de polinomios es conmutativo.
prop_conmutativaProducto :: Polinomio Int -> Polinomio Int -> Bool
prop_conmutativaProducto p q =
  multPol p q == multPol q p

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

-- El producto es distributivo respecto de la suma.
prop_distributivaProductoSuma :: Polinomio Int -> Polinomio Int
                                 -> Polinomio Int -> Bool
prop_distributivaProductoSuma p q r =
  multPol p (sumaPol q r) == sumaPol (multPol p q) (multPol p r)

-- Comprobación:
--    λ> quickCheck prop_distributivaProductoSuma
--    OK, passed 100 tests.
```

10.12.2. En Python

```
# ---------------------------------------------------------------------
# Usando el [tipo abstracto de los polinomios](https://bit.ly/3KwqXYu),
# definir la función
#    multPol : (Polinomio[A], Polinomio[A]) -> Polinomio[A]
# tal que multPol(p, q) es el producto de los polinomios p y q. Por
# ejemplo,
#    >>> ejPol1 = consPol(4, 3, consPol(2, -5, consPol(0, 3, polCero())))
#    >>> ejPol2 = consPol(5, 1, consPol(2, 5, consPol(1, 4, polCero())))
#    >>> ejPol1
#    3*x^4 + -5*x^2 + 3
#    >>> ejPol2
#    x^5 + 5*x^2 + 4*x
#    >>> multPol(ejPol1, ejPol2)
#    3*x^9 + -5*x^7 + 15*x^6 + 15*x^5 + -25*x^4 + -20*x^3 + 15*x^2 + 12*x
#
# Comprobar con Hypothesis las siguientes propiedades
# + El producto de polinomios es conmutativo.
```

```
# + El producto es distributivo respecto de la suma.
# ---------------------------------------------------------------------

# pylint: disable=arguments-out-of-order

from typing import TypeVar

from hypothesis import given

from src.Pol_Suma_de_polinomios import sumaPol
from src.Pol_Termino_lider import termLider
from src.TAD.Polinomio import (Polinomio, coefLider, consPol, esPolCero, grado,
                               polCero, polinomioAleatorio, restoPol)

A = TypeVar('A', int, float, complex)

# multPorTerm(t, p) es el producto del término t por el polinomio
# p. Por ejemplo,
#    ejTerm                    ==  4*x
#    ejPol2                    ==  x^5 + 5*x^2 + 4*x
#    multPorTerm ejTerm ejPol2  ==  4*x^6 + 20*x^3 + 16*x^2
def multPorTerm(term: Polinomio[A], pol: Polinomio[A]) -> Polinomio[A]:
    n = grado(term)
    a = coefLider(term)
    m = grado(pol)
    b = coefLider(pol)
    r = restoPol(pol)
    if esPolCero(pol):
        return polCero()
    return consPol(n + m, a * b, multPorTerm(term, r))

def multPol(p: Polinomio[A], q: Polinomio[A]) -> Polinomio[A]:
    if esPolCero(p):
        return polCero()
    return sumaPol(multPorTerm(termLider(p), q),
                   multPol(restoPol(p), q))

# El producto de polinomios es conmutativo.
@given(p=polinomioAleatorio(),
       q=polinomioAleatorio())
```

```
def test_conmutativaProducto(p: Polinomio[int], q: Polinomio[int]) -> None:
    assert multPol(p, q) == multPol(q, p)

# El producto es distributivo respecto de la suma.
@given(p=polinomioAleatorio(),
       q=polinomioAleatorio(),
       r=polinomioAleatorio())
def test_distributivaProductoSuma(p: Polinomio[int],
                                  q: Polinomio[int],
                                  r: Polinomio[int]) -> None:
    assert multPol(p, sumaPol(q, r)) == sumaPol(multPol(p, q), multPol(p, r))

# La comprobación es
#    > poetry run pytest -v Pol_Producto_polinomios.py
#    test_conmutativaProducto PASSED
#    test_distributivaProductoSuma PASSED
```

10.13. Valor de un polinomio en un punto

10.13.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Usando el [tipo abstracto de los polinomios](https://bit.ly/3KwqXYu),
-- definir la función
--    valor :: (Num a, Eq a) => Polinomio a -> a -> a
-- tal que (valor p c) es el valor del polinomio p al sustituir su
-- variable por c. Por ejemplo,
--    λ> ejPol = consPol 4 3 (consPol 2 (-5) (consPol 0 3 polCero))
--    λ> ejPol
--    3*x^4 + -5*x^2 + 3
--    λ> valor ejPol 0
--    3
--    λ> valor ejPol 1
--    1
--    λ> valor ejPol (-2)
--    31
-- ---------------------------------------------------------------------

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

```haskell
module Pol_Valor_de_un_polinomio_en_un_punto where

import TAD.Polinomio (Polinomio, polCero, esPolCero, consPol, grado,
                      coefLider, restoPol)

valor :: (Num a, Eq a) => Polinomio a -> a -> a
valor p c
  | esPolCero p = 0
  | otherwise   =  b*c^n + valor r c
  where n = grado p
        b = coefLider p
        r = restoPol p
```

10.13.2. En Python

```python
# ---------------------------------------------------------------------
# Usando el [tipo abstracto de los polinomios](https://bit.ly/3KwqXYu),
# definir la función
#    valor : (Polinomio[A], A) -> A
# tal que valor(p, c) es el valor del polinomio p al sustituir su
# variable por c. Por ejemplo,
#    >>> ejPol = consPol(4, 3, consPol(2, -5, consPol(0, 3, polCero())))
#    >>> ejPol
#    3*x^4 + -5*x^2 + 3
#    >>> valor(ejPol, 0)
#    3
#    >>> valor(ejPol, 1)
#    1
#    >>> valor(ejPol, -2)
#    31
# ---------------------------------------------------------------------

# pylint: disable=unused-import

from typing import TypeVar

from src.TAD.Polinomio import (Polinomio, coefLider, consPol, esPolCero, grado,
                               polCero, restoPol)

A = TypeVar('A', int, float, complex)
```

```python
def valor(p: Polinomio[A], c: A) -> A:
    if esPolCero(p):
        return 0
    n = grado(p)
    b = coefLider(p)
    r = restoPol(p)
    return b*c**n + valor(r, c)
```

10.14. Comprobación de raíces de polinomios

10.14.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Usando el [tipo abstracto de los polinomios](https://bit.ly/3KwqXYu),
-- definir la función
--    esRaiz :: (Num a, Eq a) => a -> Polinomio a -> Bool
-- tal que (esRaiz c p) se verifica si c es una raiz del polinomio p.
-- Por ejemplo,
--    λ> ejPol = consPol 4 6 (consPol 1 2 polCero)
--    λ> ejPol
--    6*x^4 + 2*x
--    λ> esRaiz 0 ejPol
--    True
--    λ> esRaiz 1 ejPol
--    False
-- ---------------------------------------------------------------------

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

module Pol_Comprobacion_de_raices_de_polinomios where

import TAD.Polinomio (Polinomio, polCero, consPol)
import Pol_Valor_de_un_polinomio_en_un_punto (valor)

esRaiz :: (Num a, Eq a) => a -> Polinomio a -> Bool
esRaiz c p = valor p c == 0
```

10.14.2. En Python

```python
# ---------------------------------------------------------------------
# Usando el [tipo abstracto de los polinomios](https://bit.ly/3KwqXYu),
# definir la función
#    esRaiz(A, Polinomio[A]) -> bool
# tal que esRaiz(c, p) se verifica si c es una raiz del polinomio p. Por
# ejemplo,
#    >>> ejPol = consPol(4, 6, consPol(1, 2, polCero()))
#    >>> ejPol
#    6*x^4 + 2*x
#    >>> esRaiz(0, ejPol)
#    True
#    >>> esRaiz(1, ejPol)
#    False
# ---------------------------------------------------------------------

# pylint: disable=unused-import

from typing import TypeVar

from src.Pol_Valor_de_un_polinomio_en_un_punto import valor
from src.TAD.Polinomio import Polinomio, consPol, polCero

A = TypeVar('A', int, float, complex)

def esRaiz(c: A, p: Polinomio[A]) -> bool:
    return valor(p, c) == 0
```

10.15. Derivada de un polinomio

10.15.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Usando el [tipo abstracto de los polinomios](https://bit.ly/3KwqXYu),
-- definir la función
--    derivada :: (Eq a, Num a) => Polinomio a -> Polinomio a
-- tal que (derivada p) es la derivada del polinomio p. Por ejemplo,
--    λ> ejPol = consPol 5 1 (consPol 2 5 (consPol 1 4 polCero))
--    λ> ejPol
```

```
--    x^5 + 5*x^2 + 4*x
--    λ> derivada ejPol
--    5*x^4 + 10*x + 4
-- ---------------------------------------------------------------------

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

module Pol_Derivada_de_un_polinomio where

import TAD.Polinomio (Polinomio, polCero, consPol, grado, coefLider,
                      restoPol)
import Pol_Suma_de_polinomios (sumaPol)
import Test.QuickCheck

derivada :: (Eq a, Num a) => Polinomio a -> Polinomio a
derivada p
  | n == 0     = polCero
  | otherwise  = consPol (n-1) (b * fromIntegral n) (derivada r)
  where n = grado p
        b = coefLider p
        r = restoPol p

-- Propiedad. La derivada de la suma es la suma de las derivadas.
prop_derivada :: Polinomio Int -> Polinomio Int -> Bool
prop_derivada p q =
  derivada (sumaPol p q) == sumaPol (derivada p) (derivada q)

-- Comprobación
--    λ> quickCheck prop_derivada
--    OK, passed 100 tests.
```

10.15.2. En Python

```
# ---------------------------------------------------------------------
# Usando el [tipo abstracto de los polinomios](https://bit.ly/3KwqXYu),
# definir la función
#    derivada :: (Eq a, Num a) => Polinomio a -> Polinomio a
# tal que (derivada p) es la derivada del polinomio p. Por ejemplo,
#    >>> ejPol = consPol(5, 1, consPol(2, 5, consPol(1, 4, polCero())))
#    >>> ejPol
```

```python
#    x^5 + 5*x^2 + 4*x
#    >>> derivada(ejPol)
#    5*x^4 + 10*x + 4
# ---------------------------------------------------------------------

from typing import TypeVar

from hypothesis import given

from src.Pol_Suma_de_polinomios import sumaPol
from src.TAD.Polinomio import (Polinomio, coefLider, consPol, grado, polCero,
                               polinomioAleatorio, restoPol)

A = TypeVar('A', int, float, complex)


def derivada(p: Polinomio[A]) -> Polinomio[A]:
    n = grado(p)
    if n == 0:
        return polCero()
    b = coefLider(p)
    r = restoPol(p)
    return consPol(n - 1, b * n, derivada(r))

# Propiedad. La derivada de la suma es la suma de las derivadas.
@given(p=polinomioAleatorio(), q=polinomioAleatorio())
def test_derivada(p: Polinomio[int], q: Polinomio[int]) -> None:
    assert derivada(sumaPol(p, q)) == sumaPol(derivada(p), derivada(q))

# La comprobación es
#    > poetry run pytest -q Pol_Derivada_de_un_polinomio.py
#    1 passed in 0.46s
```

10.16. Resta de polinomios

10.16.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Usando el [tipo abstracto de los polinomios](https://bit.ly/3KwqXYu),
-- definir la función
```

```
--    restaPol :: (Num a, Eq a) => Polinomio a -> Polinomio a -> Polinomio a
-- tal que (restaPol p q) es el polinomio obtenido restándole a p el
-- q. Por ejemplo,
--    λ> ejPol1 = consPol 5 1 (consPol 4 5 (consPol 2 5 (consPol 0 9 polCero)))
--    λ> ejPol2 = consPol 4 3 (consPol 2 5 (consPol 0 3 polCero))
--    λ> ejPol1
--    x^5 + 5*x^4 + 5*x^2 + 9
--    λ> ejPol2
--    3*x^4 + 5*x^2 + 3
--    λ> restaPol ejPol1 ejPol2
--    x^5 + 2*x^4 + 6
-- ---------------------------------------------------------------------

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

module Pol_Resta_de_polinomios where

import TAD.Polinomio (Polinomio, polCero, consPol)
import Pol_Suma_de_polinomios (sumaPol)
import Pol_Crea_termino (creaTermino)
import Pol_Producto_polinomios (multPorTerm)

restaPol :: (Num a, Eq a) => Polinomio a -> Polinomio a -> Polinomio a
restaPol p q  =
  sumaPol p (multPorTerm (creaTermino 0 (-1)) q)
```

10.16.2. En Python

```
# ---------------------------------------------------------------------
# Usando el [tipo abstracto de los polinomios](https://bit.ly/3KwqXYu),
# definir la función
#    restaPol : (Polinomio[A], Polinomio[A]) -> Polinomio[A]
# tal que restaPol(p, q) es el polinomio obtenido restándole a p el q. Por
# ejemplo,
#    >>> ejPol1 = consPol(5,1,consPol(4,5,consPol(2,5,consPol(0,9,polCero()))))
#    >>> ejPol2 = consPol(4,3,consPol(2,5,consPol(0,3,polCero())))
#    >>> ejPol1
#    x^5 + 5*x^4 + 5*x^2 + 9
#    >>> ejPol2
#    3*x^4 + 5*x^2 + 3
```

```python
#    >>> restaPol(ejPol1, ejPol2)
#    x^5 + 2*x^4 + 6
# ---------------------------------------------------------------------

# pylint: disable=unused-import

from typing import TypeVar

from src.Pol_Crea_termino import creaTermino
from src.Pol_Producto_polinomios import multPorTerm
from src.Pol_Suma_de_polinomios import sumaPol
from src.TAD.Polinomio import Polinomio, consPol, polCero

A = TypeVar('A', int, float, complex)

def restaPol(p: Polinomio[A], q: Polinomio[A]) -> Polinomio[A]:
    return sumaPol(p, multPorTerm(creaTermino(0, -1), q))
```

10.17. Potencia de un polinomio

10.17.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Usando el [tipo abstracto de los polinomios](https://bit.ly/3KwqXYu),
-- definir la función
--    potencia :: (Num a, Eq a) => Polinomio a -> Int -> Polinomio a
-- tal que (potencia p n) es la potencia n-ésima del polinomio p. Por
-- ejemplo,
--    λ> ejPol = consPol 1 2 (consPol 0 3 polCero)
--    λ> ejPol
--    2*x + 3
--    λ> potencia ejPol 2
--    4*x^2 + 12*x + 9
--    λ> potencia ejPol 3
--    8*x^3 + 36*x^2 + 54*x + 27
-- ---------------------------------------------------------------------

module Pol_Potencia_de_un_polinomio where

import TAD.Polinomio (Polinomio, polCero, consPol)
```

```haskell
import Pol_Producto_polinomios (multPol)
import Test.QuickCheck

-- 1ª solución
-- ===========

potencia :: (Num a, Eq a) => Polinomio a -> Int -> Polinomio a
potencia _ 0 = polUnidad
potencia p n = multPol p (potencia p (n-1))

polUnidad :: (Num a, Eq a) => Polinomio a
polUnidad = consPol 0 1 polCero

-- 2ª solución
-- ===========

potencia2 :: (Num a, Eq a) => Polinomio a -> Int -> Polinomio a
potencia2 _ 0 = polUnidad
potencia2 p n
  | even n    = potencia2 (multPol p p) (n `div` 2)
  | otherwise = multPol p (potencia2 (multPol p p) ((n-1) `div` 2))

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

-- La propiedad es
prop_potencia :: Polinomio Int -> NonNegative Int -> Bool
prop_potencia p (NonNegative n) =
  potencia p n == potencia2 p n

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

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

-- La comparación es
--    λ> import TAD.Polinomio (grado)
--    λ> ejPol = consPol 1 2 (consPol 0 3 polCero)
```

```
--    λ> grado (potencia ejPol 1000)
--    1000
--    (4.57 secs, 2,409,900,720 bytes)
--    λ> grado (potencia2 ejPol 1000)
--    1000
--    (2.78 secs, 1,439,596,632 bytes)
```

10.17.2. En Python

```
# ---------------------------------------------------------------------
# Usando el [tipo abstracto de los polinomios](https://bit.ly/3KwqXYu),
# definir la función
#    potencia : (Polinomio[A], int) -> Polinomio[A]
# tal que potencia(p, n) es la potencia n-ésima del polinomio p. Por
# ejemplo,
#    >>> ejPol = consPol(1, 2, consPol(0, 3, polCero()))
#    >>> ejPol
#    2*x + 3
#    >>> potencia(ejPol, 2)
#    4*x^2 + 12*x + 9
#    >>> potencia(ejPol, 3)
#    8*x^3 + 36*x^2 + 54*x + 27
# ---------------------------------------------------------------------

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 src.Pol_Producto_polinomios import multPol
from src.TAD.Polinomio import Polinomio, consPol, polCero, polinomioAleatorio

setrecursionlimit(10**6)

A = TypeVar('A', int, float, complex)

# 1ª solución
# ===========
```

```python
def potencia(p: Polinomio[A], n: int) -> Polinomio[A]:
    if n == 0:
        return consPol(0, 1, polCero())
    return multPol(p, potencia(p, n - 1))

# 2ª solución
# ===========

def potencia2(p: Polinomio[A], n: int) -> Polinomio[A]:
    if n == 0:
        return consPol(0, 1, polCero())
    if n % 2 == 0:
        return potencia2(multPol(p, p), n // 2)
    return multPol(p, potencia2(multPol(p, p), (n - 1) // 2))

# 3ª solución
# ===========

def potencia3(p: Polinomio[A], n: int) -> Polinomio[A]:
    r: Polinomio[A] = consPol(0, 1, polCero())
    for _ in range(0, n):
        r = multPol(p, r)
    return r

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

# La propiedad es
@given(p=polinomioAleatorio(),
       n=st.integers(min_value=1, max_value=10))
def test_potencia(p: Polinomio[int], n: int) -> None:
    r = potencia(p, n)
    assert potencia2(p, n) == r
    assert potencia3(p, n) == r

# La comprobación es
#    src> poetry run pytest -q Pol_Potencia_de_un_polinomio.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
#    >>> from src.TAD.Polinomio import grado
#    >>> ejPol = consPol(1, 2, consPol(0, 3, polCero()))
#    >>> tiempo('grado(potencia(ejPol, 1000))')
#    8.58 segundos
#    >>> tiempo('grado(potencia2(ejPol, 1000))')
#    8.75 segundos
```

10.18. Integral de un polinomio

10.18.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Usando el [tipo abstracto de los polinomios](https://bit.ly/3KwqXYu),
-- definir la función
--    integral :: (Fractional a, Eq a) => Polinomio a -> Polinomio a
-- tal que (integral p) es la integral del polinomio p cuyos coefientes
-- son números racionales. Por ejemplo,
--    λ> ejPol = consPol 7 2 (consPol 4 5 (consPol 2 5 polCero))
--    λ> ejPol
--    2*x^7 + 5*x^4 + 5*x^2
--    λ> integral ejPol
--    0.25*x^8 + x^5 + 1.6666666666666667*x^3
--    λ> integral ejPol :: Polinomio Rational
--    1 % 4*x^8 + x^5 + 5 % 3*x^3
-- ---------------------------------------------------------------------

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

module Pol_Integral_de_un_polinomio where

import TAD.Polinomio (Polinomio, polCero, consPol, esPolCero, grado,
```

```
                          coefLider, restoPol)
import Data.Ratio

integral :: (Fractional a, Eq a) => Polinomio a -> Polinomio a
integral p
  | esPolCero p = polCero
  | otherwise   = consPol (n+1) (b / fromIntegral (n+1)) (integral r)
  where n = grado p
        b = coefLider p
        r = restoPol p
```

10.18.2. En Python

```python
# ---------------------------------------------------------------------
# Usando el [tipo abstracto de los polinomios](https://bit.ly/3KwqXYu),
# definir la función
#    integral : (Polinomio[float]) -> Polinomio[float]
# tal que integral(p) es la integral del polinomio p cuyos coefientes
# son números decimales. Por ejemplo,
#    >>> ejPol = consPol(7, 2, consPol(4, 5, consPol(2, 5, polCero())))
#    >>> ejPol
#    2*x^7 + 5*x^4 + 5*x^2
#    >>> integral(ejPol)
#    0.25*x^8 + x^5 + 1.6666666666666667*x^3
# ---------------------------------------------------------------------

from src.TAD.Polinomio import (Polinomio, coefLider, consPol, esPolCero, grado,
                               polCero, restoPol)


def integral(p: Polinomio[float]) -> Polinomio[float]:
    if esPolCero(p):
        return  polCero()
    n = grado(p)
    b = coefLider(p)
    r = restoPol(p)
    return consPol(n + 1, b / (n + 1), integral(r))
```

10.19. Integral definida de un polinomio

10.19.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Usando el [tipo abstracto de datos de los polinomios](https://bit.ly/3KwqXYu)
-- definir la función
--    integralDef :: (Fractional t, Eq t) => Polinomio t -> t -> t -> t
-- tal que (integralDef p a b) es la integral definida del polinomio p
-- entre a y b. Por ejemplo,
--    λ> ejPol = consPol 7 2 (consPol 4 5 (consPol 2 5 polCero))
--    λ> ejPol
--    2*x^7 + 5*x^4 + 5*x^2
--    λ> integralDef ejPol 0 1
--    2.916666666666667
--    λ> integralDef ejPol 0 1 :: Rational
--    35 % 12
-- ---------------------------------------------------------------------

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

module Pol_Integral_definida_de_un_polinomio where

import TAD.Polinomio (Polinomio, consPol, polCero)
import Pol_Valor_de_un_polinomio_en_un_punto (valor)
import Pol_Integral_de_un_polinomio (integral)

integralDef :: (Fractional t, Eq t) => Polinomio t -> t -> t -> t
integralDef p a b = valor q b - valor q a
  where q = integral p
```

10.19.2. En Python

```python
# ---------------------------------------------------------------------
# Usando el [tipo abstracto de datos de los polinomios](https://bit.ly/3KwqXYu)
# definir la función
#    integralDef : (Polinomio[float], float, float) -> float
# tal que integralDef(p, a, b) es la integral definida del polinomio p
# entre a y b. Por ejemplo,
#    >>> ejPol = consPol(7, 2, consPol(4, 5, consPol(2, 5, polCero())))
```

```
#    >>> ejPol
#    2*x^7 + 5*x^4 + 5*x^2
#    >>> integralDef(ejPol, 0, 1)
#    2.916666666666667
# ---------------------------------------------------------------------

# pylint: disable=unused-import

from src.Pol_Integral_de_un_polinomio import integral
from src.Pol_Valor_de_un_polinomio_en_un_punto import valor
from src.TAD.Polinomio import Polinomio, consPol, polCero


def integralDef(p: Polinomio[float], a: float, b: float) -> float:
    q = integral(p)
    return valor(q, b) - valor(q, a)
```

10.20. Multiplicación de un polinomio por un número

10.20.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Usando el [tipo abstracto de los polinomios](https://bit.ly/3KwqXYu),
-- definir la función
--    multEscalar :: (Num a, Eq a) => a -> Polinomio a -> Polinomio a
-- tal que (multEscalar c p) es el polinomio obtenido multiplicando el
-- número c por el polinomio p. Por ejemplo,
--    λ> ejPol = consPol 1 2 (consPol 0 3 polCero)
--    λ> ejPol
--    2*x + 3
--    λ> multEscalar 4 ejPol
--    8*x + 12
--    λ> multEscalar (1 % 4) ejPol
--    1 % 2*x + 3 % 4
-- ---------------------------------------------------------------------

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

```haskell
module Pol_Multiplicacion_de_un_polinomio_por_un_numero where

import TAD.Polinomio (Polinomio, polCero, esPolCero, consPol, grado,
                      coefLider, restoPol)
import Data.Ratio

multEscalar :: (Num a, Eq a) => a -> Polinomio a -> Polinomio a
multEscalar c p
  | esPolCero p = polCero
  | otherwise   = consPol n (c*b) (multEscalar c r)
  where n = grado p
        b = coefLider p
        r = restoPol p
```

10.20.2. En Python

```python
# ---------------------------------------------------------------------
# Usando el [tipo abstracto de los polinomios](https://bit.ly/3KwqXYu),
# definir la función
#    multEscalar : (A, Polinomio[A]) -> Polinomio[A]
# tal que multEscalar(c, p) es el polinomio obtenido multiplicando el
# número c por el polinomio p. Por ejemplo,
#    >>> ejPol = consPol(1, 2, consPol(0, 3, polCero()))
#    >>> ejPol
#    2*x + 3
#    >>> multEscalar(4, ejPol)
#    8*x + 12
#    >>> from fractions import Fraction
#    >>> multEscalar(Fraction('1/4'), ejPol)
#    1/2*x + 3/4
# ---------------------------------------------------------------------

from typing import TypeVar

from src.TAD.Polinomio import (Polinomio, coefLider, consPol, esPolCero, grado,
                               polCero, restoPol)

A = TypeVar('A', int, float, complex)

def multEscalar(c: A, p: Polinomio[A]) -> Polinomio[A]:
```

```
    if esPolCero(p):
        return polCero()
    n = grado(p)
    b = coefLider(p)
    r = restoPol(p)
    return consPol(n, c * b, multEscalar(c, r))
```

10.21. División de polinomios

10.21.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Usando el [tipo abstracto de los polinomios](https://bit.ly/3KwqXYu),
-- definir las funciones
--    cociente :: (Fractional a, Eq a) =>
--                Polinomio a -> Polinomio a -> Polinomio a
--    resto    :: (Fractional a, Eq a) =>
--                Polinomio a -> Polinomio a -> Polinomio a
-- tales que
-- + (cociente p q) es el cociente de la división de p entre q. Por
--   ejemplo,
--      λ> pol1 = consPol 3 2 (consPol 2 9 (consPol 1 10 (consPol 0 4 polCero)))
--      λ> pol1
--      2*x^3 + 9*x^2 + 10*x + 4
--      λ> pol2 = consPol 2 1 (consPol 1 3 polCero)
--      λ> pol2
--      x^2 + 3*x
--      λ> cociente pol1 pol2
--      2.0*x + 3.0
-- + (resto p q) es el resto de la división de p entre q. Por ejemplo,
--      λ> resto pol1 pol2
--      1.0*x + 4.0
-- ---------------------------------------------------------------------

module Pol_Division_de_polinomios where

import TAD.Polinomio (Polinomio, polCero, consPol, grado, coefLider)
import Pol_Crea_termino (creaTermino)
import Pol_Producto_polinomios (multPol, multPorTerm)
import Pol_Resta_de_polinomios (restaPol)
```

```haskell
import Pol_Multiplicacion_de_un_polinomio_por_un_numero (multEscalar)

cociente :: (Fractional a, Eq a) =>
            Polinomio a -> Polinomio a -> Polinomio a
cociente p q
  | n2 == 0   = multEscalar (1/a2) p
  | n1 < n2   = polCero
  | otherwise = consPol n3 a3 (cociente p3 q)
  where n1 = grado p
        a1 = coefLider p
        n2 = grado q
        a2 = coefLider q
        n3 = n1-n2
        a3 = a1/a2
        p3 = restaPol p (multPorTerm (creaTermino n3 a3) q)

resto :: (Fractional a, Eq a) =>
         Polinomio a -> Polinomio a -> Polinomio a
resto p q = restaPol p (multPol (cociente p q) q)
```

10.21.2. En Python

```python
# ---------------------------------------------------------------------
# Usando el [tipo abstracto de los polinomios](https://bit.ly/3KwqXYu),
# definir las funciones
#    cociente : (Polinomio[float], Polinomio[float]) -> Polinomio[float]
#    resto    : (Polinomio[float], Polinomio[float]) -> Polinomio[float]
# tales que
# + cociente(p, q) es el cociente de la división de p entre q. Por
#   ejemplo,
#      >>> pol1 = consPol(3, 2, consPol(2, 9, consPol(1, 10, consPol(0, 4, polCer
#      >>> pol1
#      2*x^3 + 9*x^2 + 10*x + 4
#      >>> pol2 = consPol(2, 1, consPol(1, 3, polCero()))
#      >>> pol2
#      x^2 + 3*x
#      >>> cociente(pol1, pol2)
#      2.0*x + 3.0
# + resto(p, q) es el resto de la división de p entre q. Por ejemplo,
#      >>> resto(pol1, pol2)
```

```
#      1.0*x + 4
# ---------------------------------------------------------------------

from src.Pol_Crea_termino import creaTermino
from src.Pol_Multiplicacion_de_un_polinomio_por_un_numero import multEscalar
from src.Pol_Producto_polinomios import multPol, multPorTerm
from src.Pol_Resta_de_polinomios import restaPol
from src.TAD.Polinomio import Polinomio, coefLider, consPol, grado, polCero


def cociente(p: Polinomio[float], q: Polinomio[float]) -> Polinomio[float]:
    n1 = grado(p)
    a1 = coefLider(p)
    n2 = grado(q)
    a2 = coefLider(q)
    n3 = n1 - n2
    a3 = a1 / a2
    p3 = restaPol(p, multPorTerm(creaTermino(n3, a3), q))
    if n2 == 0:
        return multEscalar(1 / a2, p)
    if n1 < n2:
        return polCero()
    return consPol(n3, a3, cociente(p3, q))

def resto(p: Polinomio[float], q: Polinomio[float]) -> Polinomio[float]:
    return restaPol(p, multPol(cociente(p, q), q))
```

10.22. Divisibilidad de polinomios

10.22.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Usando el [tipo abstracto de los polinomios](https://bit.ly/3KwqXYu),
-- definir la función
--    divisiblePol :: (Fractional a, Eq a) =>
--                    Polinomio a -> Polinomio a -> Bool
-- tal que (divisiblePol p q) se verifica si el polinomio p es divisible
-- por el polinomio q. Por ejemplo,
--    λ> pol1 = consPol 2 8 (consPol 1 14 (consPol 0 3 polCero))
--    λ> pol1
```

```
--    8*x^2 + 14*x + 3
--    λ> pol2 = consPol 1 2 (consPol 0 3 polCero)
--    λ> pol2
--    2*x + 3
--    λ> pol3 = consPol 2 6 (consPol 1 2 polCero)
--    λ> pol3
--    6*x^2 + 2*x
--    λ> divisiblePol pol1 pol2
--    True
--    λ> divisiblePol pol1 pol3
--    False
-- ---------------------------------------------------------------------

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

module Pol_Divisibilidad_de_polinomios where

import TAD.Polinomio (Polinomio, polCero, consPol, esPolCero)
import Pol_Division_de_polinomios (resto)

divisiblePol :: (Fractional a, Eq a) =>
                Polinomio a -> Polinomio a -> Bool
divisiblePol p q = esPolCero (resto p q)
```

10.22.2. En Python

```
# ---------------------------------------------------------------------
# Usando el [tipo abstracto de los polinomios](https://bit.ly/3KwqXYu),
# definir la función
#    divisiblePol : (Polinomio[float], Polinomio[float]) -> bool
# tal que divisiblePol(p, q) se verifica si el polinomio p es divisible
# por el polinomio q. Por ejemplo,
#    >>> pol1 = consPol(2, 8, consPol(1, 14, consPol(0, 3, polCero())))
#    >>> pol1
#    8*x^2 + 14*x + 3
#    >>> pol2 = consPol(1, 2, consPol(0, 3, polCero()))
#    >>> pol2
#    2*x + 3
#    >>> pol3 = consPol(2, 6, consPol(1, 2, polCero()))
#    >>> pol3
```

```python
#    6*x^2 + 2*x
#    >>> divisiblePol(pol1, pol2)
#    True
#    >>> divisiblePol(pol1, pol3)
#    False
# ---------------------------------------------------------------------

# pylint: disable=unused-import

from src.Pol_Division_de_polinomios import resto
from src.TAD.Polinomio import Polinomio, consPol, esPolCero, polCero


def divisiblePol(p: Polinomio[float], q: Polinomio[float]) -> bool:
    return esPolCero(resto(p, q))
```

10.23. Método de Horner del valor de un polinomio

10.23.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- El método de Horner para calcular el valor de un  polinomio se basa
-- en representarlo de una forma forma alernativa. Por ejemplo, para
-- calcular el valor de
--     a*x^5 + b*x^4 + c*x^3 + d*x^2 + e*x + f
-- se representa como
--    (((((0 * x + a) * x + b) * x + c) * x + d) * x + e) * x + f
-- y se evalúa de dentro hacia afuera; es decir,
--    v(0) = 0
--    v(1) = v(0)*x+a = 0*x+a = a
--    v(2) = v(1)*x+b = a*x+b
--    v(3) = v(2)*x+c = (a*x+b)*x+c = a*x^2+b*x+c
--    v(4) = v(3)*x+d = (a*x^2+b*x+c)*x+d = a*x^3+b*x^2+c*x+d
--    v(5) = v(4)*x+e = (a*x^3+b*x^2+c*x+d)*x+e = a*x^4+b*x^3+c*x^2+d*x+e
--    v(6) = v(5)*x+f = (a*x^4+b*x^3+c*x^2+d*x+e)*x+f = a*x^5+b*x^4+c*x^3+d*x^2+e*
--
-- Usando el [tipo abstracto de los polinomios](https://bit.ly/3KwqXYu),
-- definir la función
```

```haskell
--    horner :: (Num a, Eq a) => Polinomio a -> a -> a
-- tal que (horner p x) es el valor del polinomio p al sustituir su
-- variable por el número x. Por ejemplo,
--    λ> pol1 = consPol 5 1 (consPol 2 5 (consPol 1 4 polCero))
--    λ> pol1
--    x^5 + 5*x^2 + 4*x
--    λ> horner pol1 0
--    0
--    λ> horner pol1 1
--    10
--    λ> horner pol1 1.5
--    24.84375
--    λ> import Data.Ratio
--    λ> horner pol1 (3%2)
--    795 % 32
-- ---------------------------------------------------------------------

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

module Pol_Metodo_de_Horner_del_valor_de_un_polinomio where

import TAD.Polinomio (Polinomio, polCero, consPol)
import Pol_Transformaciones_polinomios_densas (polinomioAdensa)

-- 1ª solución
-- ===========

horner :: (Num a, Eq a) => Polinomio a -> a -> a
horner p x = hornerAux (polinomioAdensa p) 0
  where hornerAux [] v     = v
        hornerAux (a:as) v = hornerAux as (v*x+a)

-- El cálculo de (horner pol1 2) es el siguiente
--    horner pol1 2
--    = hornerAux [1,0,0,5,4,0] 0
--    = hornerAux   [0,0,5,4,0] ( 0*2+1) = hornerAux   [0,0,5,4,0] 1
--    = hornerAux     [0,5,4,0] ( 1*2+0) = hornerAux     [0,5,4,0] 2
--    = hornerAux       [5,4,0] ( 2*2+0) = hornerAux       [5,4,0] 4
--    = hornerAux         [4,0] ( 4*2+5) = hornerAux         [4,0] 13
--    = hornerAux           [0] (13*2+4) = hornerAux           [0] 30
```

```
--    = hornerAux             [] (30*2+0) = hornerAux             [] 60

-- 2ª solución
-- ===========

horner2 :: (Num a, Eq a) => Polinomio a -> a -> a
horner2 p x = foldl (\a b -> a*x + b) 0 (polinomioAdensa p)
```

10.23.2. En Python

```
# ---------------------------------------------------------------------
# El método de Horner para calcular el valor de un  polinomio se basa
# en representarlo de una forma forma alernativa. Por ejemplo, para
# calcular el valor de
#    a*x^5 + b*x^4 + c*x^3 + d*x^2 + e*x + f
# se representa como
#   (((((0 * x + a) * x + b) * x + c) * x + d) * x + e) * x + f
# y se evalúa de dentro hacia afuera; es decir,
#   v(0) = 0
#   v(1) = v(0)*x+a = 0*x+a = a
#   v(2) = v(1)*x+b = a*x+b
#   v(3) = v(2)*x+c = (a*x+b)*x+c = a*x^2+b*x+c
#   v(4) = v(3)*x+d = (a*x^2+b*x+c)*x+d = a*x^3+b*x^2+c*x+d
#   v(5) = v(4)*x+e = (a*x^3+b*x^2+c*x+d)*x+e = a*x^4+b*x^3+c*x^2+d*x+e
#   v(6) = v(5)*x+f = (a*x^4+b*x^3+c*x^2+d*x+e)*x+f = a*x^5+b*x^4+c*x^3+d*x^2+e*x
#
# Usando el [tipo abstracto de los polinomios](https://bit.ly/3KwqXYu),
# definir la función
#    horner : (Polinomio[float], float) -> float
# tal que horner(p, x) es el valor del polinomio p al sustituir su
# variable por el número x. Por ejemplo,
#    >>> pol1 = consPol(5, 1, consPol(2, 5, consPol(1, 4, polCero())))
#    >>> pol1
#    x^5 + 5*x^2 + 4*x
#    >>> horner(pol1, 0)
#    0
#    >>> horner(pol1, 1)
#    10
#    >>> horner(pol1, 1.5)
#    24.84375
```

```
#    >>> from fractions import Fraction
#    >>> horner(pol1, Fraction('3/2'))
#    Fraction(795, 32)
# ---------------------------------------------------------------------

# pylint: disable=unused-import

from functools import reduce

from src.Pol_Transformaciones_polinomios_densas import polinomioAdensa
from src.TAD.Polinomio import Polinomio, consPol, polCero

# 1ª solución
# ===========

def horner(p: Polinomio[float], x: float) -> float:
    def hornerAux(ys: list[float], v: float) -> float:
        if not ys:
            return v
        return hornerAux(ys[1:], v * x + ys[0])

    return hornerAux(polinomioAdensa(p), 0)

# El cálculo de horner(pol1, 2) es el siguiente
#    horner pol1 2
#    = hornerAux [1,0,0,5,4,0] 0
#    = hornerAux   [0,0,5,4,0] ( 0*2+1) = hornerAux   [0,0,5,4,0] 1
#    = hornerAux     [0,5,4,0] ( 1*2+0) = hornerAux     [0,5,4,0] 2
#    = hornerAux       [5,4,0] ( 2*2+0) = hornerAux       [5,4,0] 4
#    = hornerAux         [4,0] ( 4*2+5) = hornerAux         [4,0] 13
#    = hornerAux           [0] (13*2+4) = hornerAux           [0] 30
#    = hornerAux            [] (30*2+0) = hornerAux            [] 60

# 2ª solución
# ===========

def horner2(p: Polinomio[float], x: float) -> float:
    return reduce(lambda a, b: a * x + b, polinomioAdensa(p) , 0.0)
```

10.24. Término independiente de un polinomio

10.24.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Usando el [tipo abstracto de datos de los polinomios](https://bit.ly/3KwqXYu)
-- definir la función
--    terminoIndep :: (Num a, Eq a) => Polinomio  a -> a
-- tal que (terminoIndep p) es el término independiente del polinomio
-- p. Por ejemplo,
--    λ> ejPol1 = consPol 4 3 (consPol 2 5 (consPol 0 3 polCero))
--    λ> ejPol1
--    3*x^4 + 5*x^2 + 3
--    λ> terminoIndep ejPol1
--    3
--    λ> ejPol2 = consPol 5 1 (consPol 2 5 (consPol 1 4 polCero))
--    λ> ejPol2
--    x^5 + 5*x^2 + 4*x
--    λ> terminoIndep ejPol2
--    0
-- ---------------------------------------------------------------------

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

module Pol_Termino_independiente_de_un_polinomio where

import TAD.Polinomio (Polinomio, consPol, polCero)
import Pol_Coeficiente (coeficiente)

terminoIndep :: (Num a, Eq a) => Polinomio  a -> a
terminoIndep = coeficiente 0
```

10.24.2. En Python

```python
# ---------------------------------------------------------------------
# Usando el [tipo abstracto de los polinomios](https://bit.ly/3KwqXYu)
# definir la función
#    terminoIndep : (Polinomio[A]) -> A
# tal que terminoIndep(p) es el término independiente del polinomio
# p. Por ejemplo,
```

```
#    >>> ejPol1 = consPol(4, 3, consPol(2, 5, consPol(0, 3, polCero())))
#    >>> ejPol1
#    3*x^4 + 5*x^2 + 3
#    >>> terminoIndep(ejPol1)
#    3
#    >>> ejPol2 = consPol(5, 1, consPol(2, 5, consPol(1, 4, polCero())))
#    >>> ejPol2
#    x^5 + 5*x^2 + 4*x
#    >>> terminoIndep(ejPol2)
#    0
# ---------------------------------------------------------------------

# pylint: disable=unused-import

from typing import TypeVar

from src.Pol_Coeficiente import coeficiente
from src.TAD.Polinomio import Polinomio, consPol, polCero

A = TypeVar('A', int, float, complex)

def terminoIndep(p: Polinomio[A]) -> A:
    return coeficiente(0, p)
```

10.25. Regla de Ruffini con representación densa

10.25.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Usando el [tipo abstracto de los polinomios](https://bit.ly/3KwqXYu)
-- definir la función
--    ruffiniDensa :: Int -> [Int] -> [Int]
-- tal que (ruffiniDensa r cs) es la lista de los coeficientes del
-- cociente junto con el rsto que resulta de aplicar la regla de Ruffini
-- para dividir el polinomio cuya representación densa es cs entre
-- x-r. Por ejemplo,
--    ruffiniDensa 2 [1,2,-1,-2] == [1,4,7,12]
--    ruffiniDensa 1 [1,2,-1,-2] == [1,3,2,0]
```

```haskell
-- ya que
--       | 1   2   -1   -2           | 1   2   -1   -2
--     2 |     2    8   14         1 |     1    3    2
--     --+--------------           --+-------------
--       | 1   4    7   12           | 1   3    2    0
-- ---------------------------------------------------------------------

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

module Pol_Division_de_Ruffini_con_representacion_densa where

import Test.QuickCheck

-- 1ª solución
-- ===========

ruffiniDensa :: Int -> [Int] -> [Int]
ruffiniDensa _ [] = []
ruffiniDensa r p@(c:cs) =
  c : [x+r*y | (x,y) <- zip cs (ruffiniDensa r p)]

-- 2ª solución
-- ===========

ruffiniDensa2 :: Int -> [Int] -> [Int]
ruffiniDensa2 r =
  scanl1 (\s x -> s * r + x)

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

-- La propiedad es
prop_ruffiniDensa :: Int -> [Int] -> Bool
prop_ruffiniDensa r cs =
  ruffiniDensa r cs == ruffiniDensa2 r cs

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

10.25.2. En Python

```
# ---------------------------------------------------------------------
# Usando el [tipo abstracto de los polinomios](https://bit.ly/3KwqXYu)
# definir la función
#    ruffiniDensa : (int, list[int]) -> list[int]
# tal que ruffiniDensa(r, cs) es la lista de los coeficientes del
# cociente junto con el rsto que resulta de aplicar la regla de Ruffini
# para dividir el polinomio cuya representación densa es cs entre
# x-r. Por ejemplo,
#    ruffiniDensa(2, [1, 2, -1, -2]) == [1, 4, 7, 12]
#    ruffiniDensa(1, [1, 2, -1, -2]) == [1, 3, 2, 0]
# ya que
#      | 1  2  -1  -2           | 1  2  -1  -2
#    2 |    2   8  14         1 |    1   3   2
#    --+--------------        --+-------------
#      | 1  4   7  12           | 1  3   2   0
# ---------------------------------------------------------------------

def ruffiniDensa(r: int, p: list[int]) -> list[int]:
    if not p:
        return []
    res = [p[0]]
    for x in p[1:]:
        res.append(x + r * res[-1])
    return res
```

10.26. Regla de Ruffini

10.26.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Usando el [tipo abstracto de los polinomios](https://bit.ly/3KwqXYu),
-- definir las funciones
--    cocienteRuffini :: Int -> Polinomio Int -> Polinomio Int
--    restoRuffini    :: Int -> Polinomio Int -> Int
-- tales que
-- + (cocienteRuffini r p) es el cociente de dividir el polinomio p por
--   el polinomio x-r. Por ejemplo:
--      λ> ejPol = consPol 3 1 (consPol 2 2 (consPol 1 (-1) (consPol 0 (-2) polCe
```

```
--       λ> ejPol
--       x^3 + 2*x^2 + -1*x + -2
--       λ> cocienteRuffini 2 ejPol
--       x^2 + 4*x + 7
--       λ> cocienteRuffini (-2) ejPol
--       x^2 + -1
--       λ> cocienteRuffini 3 ejPol
--       x^2 + 5*x + 14
-- + (restoRuffini r p) es el resto de dividir el polinomio p por el
--   polinomio x-r. Por ejemplo,
--       λ> restoRuffini 2 ejPol
--       12
--       λ> restoRuffini (-2) ejPol
--       0
--       λ> restoRuffini 3 ejPol
--       40
--
-- Comprobar con QuickCheck que, dado un polinomio p y un número entero
-- r, las funciones anteriores verifican la propiedad de la división
-- euclídea.
-- ---------------------------------------------------------------------

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

module Pol_Regla_de_Ruffini where

import TAD.Polinomio (Polinomio, consPol, polCero)
import Pol_Transformaciones_polinomios_densas (densaApolinomio,
                                               polinomioAdensa)
import Pol_Division_de_Ruffini_con_representacion_densa (ruffiniDensa)
import Pol_Producto_polinomios (multPol)
import Pol_Suma_de_polinomios (sumaPol)
import Pol_Crea_termino (creaTermino)
import Test.QuickCheck

-- 1ª definición de cocienteRuffini
-- ================================

cocienteRuffini :: Int -> Polinomio Int -> Polinomio Int
cocienteRuffini r p = densaApolinomio (init (ruffiniDensa r (polinomioAdensa p)))
```

```haskell
-- 2ª definición de cocienteRuffini
-- ================================

cocienteRuffini2 :: Int -> Polinomio Int -> Polinomio Int
cocienteRuffini2 r = densaApolinomio . ruffiniDensa r . init . polinomioAdensa

-- 1ª definición de restoRuffini
-- =============================

restoRuffini :: Int -> Polinomio Int -> Int
restoRuffini r p = last (ruffiniDensa r (polinomioAdensa p))

-- 2ª definición de restoRuffini
-- =============================

restoRuffini2 :: Int -> Polinomio Int -> Int
restoRuffini2 r = last . ruffiniDensa r . polinomioAdensa

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

-- La propiedad es
prop_diviEuclidea :: Int -> Polinomio Int -> Bool
prop_diviEuclidea r p =
  p == sumaPol (multPol coci divi) rest
  where coci = cocienteRuffini r p
        divi = densaApolinomio [1,-r]
        rest = creaTermino 0 (restoRuffini r p)

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

10.26.2. En Python

```python
# ---------------------------------------------------------------------
# Usando el [tipo abstracto de los polinomios](https://bit.ly/3KwqXYu),
# definir las funciones
#    cocienteRuffini : (int, Polinomio[int]) -> Polinomio[int]
```

```
#    restoRuffini    : (int, Polinomio[int]) -> int
# tales que
# + cocienteRuffini(r, p) es el cociente de dividir el polinomio p por
#   el polinomio x-r. Por ejemplo:
#      λ> ejPol = consPol(3, 1, consPol(2, 2, consPol(1, -1, consPol(0, -2, polCe
#      λ> ejPol
#      x^3 + 2*x^2 + -1*x + -2
#      λ> cocienteRuffini(2, ejPol)
#      x^2 + 4*x + 7
#      λ> cocienteRuffini(-2, ejPol)
#      x^2 + -1
#      λ> cocienteRuffini(3, ejPol)
#      x^2 + 5*x + 14
# + restoRuffini(r, p) es el resto de dividir el polinomio p por el
#   polinomio x-r. Por ejemplo,
#      λ> restoRuffini(2, ejPol)
#      12
#      λ> restoRuffini(-2, ejPol)
#      0
#      λ> restoRuffini(3, ejPol)
#      40
#
# Comprobar con Hypothesis que, dado un polinomio p y un número entero
# r, las funciones anteriores verifican la propiedad de la división
# euclídea.
# ---------------------------------------------------------------------

# pylint: disable=unused-import

from hypothesis import given
from hypothesis import strategies as st

from src.Pol_Crea_termino import creaTermino
from src.Pol_Division_de_Ruffini_con_representacion_densa import ruffiniDensa
from src.Pol_Producto_polinomios import multPol
from src.Pol_Suma_de_polinomios import sumaPol
from src.Pol_Transformaciones_polinomios_densas import (densaApolinomio,
                                                        polinomioAdensa)
from src.TAD.Polinomio import (Polinomio, consPol, esPolCero, polCero,
                               polinomioAleatorio)
```

```
def cocienteRuffini(r: int, p: Polinomio[int]) -> Polinomio[int]:
    if esPolCero(p):
        return polCero()
    return densaApolinomio(ruffiniDensa(r, polinomioAdensa(p))[:-1])

def restoRuffini(r: int, p: Polinomio[int]) -> int:
    if esPolCero(p):
        return 0
    return ruffiniDensa(r, polinomioAdensa(p))[-1]

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

# La propiedad es
@given(r=st.integers(), p=polinomioAleatorio())
def test_diviEuclidea (r: int, p: Polinomio[int]) -> None:
    coci = cocienteRuffini(r, p)
    divi = densaApolinomio([1, -r])
    rest = creaTermino(0, restoRuffini(r, p))
    assert p == sumaPol(multPol(coci, divi), rest)

# La comprobación es
#    src> poetry run pytest -q Pol_Regla_de_Ruffini.py
#    1 passed in 0.32s
```

10.27. Reconocimiento de raíces por la regla de Ruffini

10.27.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Usando el [tipo abstracto de los polinomios](https://bit.ly/3KwqXYu),
-- definir la función
--    esRaizRuffini:: Int -> Polinomio Int -> Bool
-- tal que (esRaizRuffini r p) se verifica si r es una raiz de p, usando
-- para ello el regla de Ruffini. Por ejemplo,
--    λ> ejPol = consPol 4 6 (consPol 1 2 polCero)
```

```
--     λ> ejPol
--     6*x^4 + 2*x
--     λ> esRaizRuffini 0 ejPol
--     True
--     λ> esRaizRuffini 1 ejPol
--     False
-- ---------------------------------------------------------------------

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

module Pol_Reconocimiento_de_raices_por_la_regla_de_Ruffini where

import TAD.Polinomio (Polinomio, consPol, polCero)
import Pol_Regla_de_Ruffini (restoRuffini)

esRaizRuffini :: Int -> Polinomio Int -> Bool
esRaizRuffini r p = restoRuffini r p == 0
```

10.27.2. En Python

```
# ---------------------------------------------------------------------
# Usando el [tipo abstracto de los polinomios](https://bit.ly/3KwqXYu),
# definir la función
#    esRaizRuffini : (int, Polinomio[int]) -> bool
# tal que esRaizRuffini(r, p) se verifica si r es una raiz de p, usando
# para ello el regla de Ruffini. Por ejemplo,
#    >>> ejPol = consPol(4, 6, consPol(1, 2, polCero()))
#    >>> ejPol
#    6*x^4 + 2*x
#    >>> esRaizRuffini(0, ejPol)
#    True
#    >>> esRaizRuffini(1, ejPol)
#    False
# ---------------------------------------------------------------------

# pylint: disable=unused-import

from src.Pol_Regla_de_Ruffini import restoRuffini
from src.TAD.Polinomio import Polinomio, consPol, polCero
```

```python
def esRaizRuffini(r: int, p: Polinomio[int]) -> bool:
    return restoRuffini(r, p) == 0
```

10.28. Raíces enteras de un polinomio

10.28.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Usando el [tipo abstracto de los polinomios](https://bit.ly/3KwqXYu),
-- definir la función
--    raicesRuffini :: Polinomio Int -> [Int]
-- tal que (raicesRuffini p) es la lista de las raices enteras de p,
-- calculadas usando el regla de Ruffini. Por ejemplo,
--    λ> ejPol1 = consPol 4 3 (consPol 2 (-5) (consPol 0 3 polCero))
--    λ> ejPol1
--    3*x^4 + -5*x^2 + 3
--    λ> raicesRuffini ejPol1
--    []
--    λ> ejPol2 = consPol 5 1 (consPol 2 5 (consPol 1 4 polCero))
--    λ> ejPol2
--    x^5 + 5*x^2 + 4*x
--    λ> raicesRuffini ejPol2
--    [0,-1]
--    λ> ejPol3 = consPol 4 6 (consPol 1 2 polCero)
--    λ> ejPol3
--    6*x^4 + 2*x
--    λ> raicesRuffini ejPol3
--    [0]
--    λ> ejPol4 = consPol 3 1 (consPol 2 2 (consPol 1 (-1) (consPol 0 (-2) polCe
--    λ> ejPol4
--    x^3 + 2*x^2 + -1*x + -2
--    λ> raicesRuffini ejPol4
--    [1,-1,-2]
-- ---------------------------------------------------------------------

module Pol_Raices_enteras_de_un_polinomio where

import TAD.Polinomio (Polinomio, consPol, polCero, esPolCero)
import Pol_Termino_independiente_de_un_polinomio (terminoIndep)
```

```haskell
import Pol_Regla_de_Ruffini (cocienteRuffini)
import Pol_Reconocimiento_de_raices_por_la_regla_de_Ruffini (esRaizRuffini)
import Test.Hspec (Spec, hspec, it, shouldBe)

raicesRuffini :: Polinomio Int -> [Int]
raicesRuffini p
  | esPolCero p = []
  | otherwise   = aux (0 : divisores (terminoIndep p))
  where aux [] = []
        aux (r:rs)
          | esRaizRuffini r p = r : raicesRuffini (cocienteRuffini r p)
          | otherwise         = aux rs

-- (divisores n) es la lista de todos los divisores enteros de n. Por
-- ejemplo,
--    divisores 4     ==  [1,-1,2,-2,4,-4]
--    divisores (-6)  ==  [1,-1,2,-2,3,-3,6,-6]
divisores :: Int -> [Int]
divisores n = concat [[x,-x] | x <- [1..abs n], rem n x == 0]

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

spec :: Spec
spec = do
  it "e1" $
    raicesRuffini ejPol1 `shouldBe` []
  it "e2" $
    raicesRuffini ejPol2 `shouldBe` [0,-1]
  it "e3" $
    raicesRuffini ejPol3 `shouldBe` [0]
  it "e4" $
    raicesRuffini ejPol4 `shouldBe` [1,-1,-2]
  where
    ejPol1 = consPol 4 3 (consPol 2 (-5) (consPol 0 3 polCero))
    ejPol2 = consPol 5 1 (consPol 2 5 (consPol 1 4 polCero))
    ejPol3 = consPol 4 6 (consPol 1 2 polCero)
```

```
    ejPol4 = consPol 3 1 (consPol 2 2 (consPol 1 (-1) (consPol 0 (-2) polCero)))

-- La verificación es
--    λ> verifica
--
--    e1
--    e2
--    e3
--    e4
--
--    Finished in 0.0013 seconds
--    4 examples, 0 failures
```

10.28.2. En Python

```
# ---------------------------------------------------------------------
# Usando el [tipo abstracto de los polinomios](https://bit.ly/3KwqXYu),
# definir la función
#     raicesRuffini : (Polinomio[int]) -> list[int]
# tal que raicesRuffini(p) es la lista de las raices enteras de p,
# calculadas usando el regla de Ruffini. Por ejemplo,
#    >>> ejPol1 = consPol(4, 3, consPol(2, -5, consPol(0, 3, polCero())))
#    >>> ejPol1
#    3*x^4 + -5*x^2 + 3
#    >>> raicesRuffini(ejPol1)
#    []
#    >>> ejPol2 = consPol(5, 1, consPol(2, 5, consPol(1, 4, polCero())))
#    >>> ejPol2
#    x^5 + 5*x^2 + 4*x
#    >>> raicesRuffini(ejPol2)
#    [0, -1]
#    >>> ejPol3 = consPol(4, 6, consPol(1, 2, polCero()))
#    >>> ejPol3
#    6*x^4 + 2*x
#    >>> raicesRuffini(ejPol3)
#    [0]
#    >>> ejPol4 = consPol(3, 1, consPol(2, 2, consPol(1, -1, consPol(0, -2, polCe
#    >>> ejPol4
#    x^3 + 2*x^2 + -1*x + -2
#    >>> raicesRuffini(ejPol4)
```

```python
#    [1, -1, -2]
# ---------------------------------------------------------------------

from src.Pol_Reconocimiento_de_raices_por_la_regla_de_Ruffini import \
    esRaizRuffini
from src.Pol_Regla_de_Ruffini import cocienteRuffini
from src.Pol_Termino_independiente_de_un_polinomio import terminoIndep
from src.TAD.Polinomio import Polinomio, consPol, esPolCero, polCero


# (divisores n) es la lista de todos los divisores enteros de n. Por
# ejemplo,
#    divisores(4)  == [1, 2, 4, -1, -2, -4]
#    divisores(-6) == [1, 2, 3, 6, -1, -2, -3, -6]
def divisores(n: int) -> list[int]:
    xs = [x for x in range(1, abs(n)+1) if n % x == 0]
    return xs + [-x for x in xs]

def raicesRuffini(p: Polinomio[int]) -> list[int]:
    if esPolCero(p):
        return []
    def aux(rs: list[int]) -> list[int]:
        if not rs:
            return []
        x, *xs = rs
        if esRaizRuffini(x, p):
            return [x] + raicesRuffini(cocienteRuffini(x, p))
        return aux(xs)

    return aux([0] + divisores(terminoIndep(p)))

# Verificación
# ============

def test_raicesRuffini() -> None:
    ejPol1 = consPol(4, 3, consPol(2, -5, consPol(0, 3, polCero())))
    assert raicesRuffini(ejPol1) == []
    ejPol2 = consPol(5, 1, consPol(2, 5, consPol(1, 4, polCero())))
    assert raicesRuffini(ejPol2) == [0, -1]
    ejPol3 = consPol(4, 6, consPol(1, 2, polCero()))
```

```
    assert raicesRuffini(ejPol3) == [0]
    ejPol4 = consPol(3, 1, consPol(2, 2, consPol(1, -1, consPol(0, -2, polCero())
    assert raicesRuffini(ejPol4) == [1, -1, -2]
    print("Verificado")

# La verificación es
#    >>> test_raicesRuffini()
#    Verificado
```

10.29. Factorización de un polinomio

10.29.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Usando el [tipo abstracto de los polinomios](https://bit.ly/3KwqXYu),
-- definir la función
--    factorizacion :: Polinomio Int -> [Polinomio Int]
-- tal que (factorizacion p) es la lista de la descomposición del
-- polinomio p en factores obtenida mediante el regla de Ruffini. Por
-- ejemplo,
--    λ> ejPol1 = consPol 5 1 (consPol 2 5 (consPol 1 4 polCero))
--    λ> ejPol1
--    x^5 + 5*x^2 + 4*x
--    λ> factorizacion ejPol1
--    [1*x,1*x + 1,x^3 + -1*x^2 + 1*x + 4]
--    λ> ejPol2 = consPol 3 1 (consPol 2 2 (consPol 1 (-1) (consPol 0 (-2) polCer
--    λ> ejPol2
--    x^3 + 2*x^2 + -1*x + -2
--    λ> factorizacion ejPol2
--    [1*x + -1,1*x + 1,1*x + 2,1]
-- ---------------------------------------------------------------------

module Pol_Factorizacion_de_un_polinomio where

import TAD.Polinomio (Polinomio, consPol, polCero, esPolCero)
import Pol_Termino_independiente_de_un_polinomio (terminoIndep)
import Pol_Raices_enteras_de_un_polinomio (divisores)
import Pol_Regla_de_Ruffini (cocienteRuffini)
import Pol_Reconocimiento_de_raices_por_la_regla_de_Ruffini (esRaizRuffini)
import Pol_Transformaciones_polinomios_densas (densaApolinomio)
```

```haskell
import Test.Hspec (Spec, hspec, it, shouldBe)

factorizacion :: Polinomio Int -> [Polinomio Int]
factorizacion p
  | esPolCero p = [p]
  | otherwise   = aux (0 : divisores (terminoIndep p))
  where
    aux [] = [p]
    aux (r:rs)
        | esRaizRuffini r p =
            densaApolinomio [1,-r] : factorizacion (cocienteRuffini r p)
        | otherwise = aux rs

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

spec :: Spec
spec = do
  it "e1" $
    map show (factorizacion ejPol1)
      `shouldBe` ["1*x","1*x + 1","x^3 + -1*x^2 + 1*x + 4"]
  it "e2" $
    map show (factorizacion ejPol2)
      `shouldBe` ["1*x + -1","1*x + 1","1*x + 2","1"]
  where
    ejPol1 = consPol 5 1 (consPol 2 5 (consPol 1 4 polCero))
    ejPol2 = consPol 3 1 (consPol 2 2 (consPol 1 (-1) (consPol 0 (-2) polCero)))

-- La verificación es
--    λ> verifica
--
--    e1
--    e2
--
--    Finished in 0.0015 seconds
--    2 examples, 0 failures
```

10.29.2. En Python

```
# ---------------------------------------------------------------------
# Usando el [tipo abstracto de los polinomios](https://bit.ly/3KwqXYu),
# definir la función
#    factorizacion : (Polinomio[int]) -> list[Polinomio[int]]
# tal que factorizacion(p) es la lista de la descomposición del
# polinomio p en factores obtenida mediante el regla de Ruffini. Por
# ejemplo,
#    >>> ejPol1 = consPol(5, 1, consPol(2, 5, consPol(1, 4, polCero())))
#    >>> ejPol1
#    x^5 + 5*x^2 + 4*x
#    >>> factorizacion(ejPol1)
#    [1*x, 1*x + 1, x^3 + -1*x^2 + 1*x + 4]
#    >>> ejPol2 = consPol(3, 1, consPol(2, 2, consPol(1, -1, consPol(0, -2, polCe
#    >>> ejPol2
#    x^3 + 2*x^2 + -1*x + -2
#    >>> factorizacion(ejPol2)
#    [1*x + -1, 1*x + 1, 1*x + 2, 1]
# ---------------------------------------------------------------------

from src.Pol_Raices_enteras_de_un_polinomio import divisores
from src.Pol_Reconocimiento_de_raices_por_la_regla_de_Ruffini import \
    esRaizRuffini
from src.Pol_Regla_de_Ruffini import cocienteRuffini
from src.Pol_Termino_independiente_de_un_polinomio import terminoIndep
from src.Pol_Transformaciones_polinomios_densas import densaApolinomio
from src.TAD.Polinomio import Polinomio, consPol, esPolCero, polCero


def factorizacion(p: Polinomio[int]) -> list[Polinomio[int]]:
    def aux(xs: list[int]) -> list[Polinomio[int]]:
        if not xs:
            return [p]
        r, *rs = xs
        if esRaizRuffini(r, p):
            return [densaApolinomio([1, -r])] + factorizacion(cocienteRuffini(r,
        return aux(rs)

    if esPolCero(p):
        return [p]
```

```
    return aux([0] + divisores(terminoIndep(p)))

# Verificación
# ============

def test_factorizacion() -> None:
    ejPol1 = consPol(5, 1, consPol(2, 5, consPol(1, 4, polCero())))
    assert list(map(str, factorizacion(ejPol1))) \
        == ["1*x", "1*x + 1", "x^3 + -1*x^2 + 1*x + 4"]
    ejPol2 = consPol(3, 1, consPol(2, 2, consPol(1, -1, consPol(0, -2, polCero())
    assert list(map(str, factorizacion(ejPol2))) \
        == ["1*x + -1", "1*x + 1", "1*x + 2", "1"]
    print("Verificado")

# La verificación es
#    >>> test_factorizacion()
#    Verificado
```

Capítulo 11

El tipo abstracto de datos de los grafos

Contenido

11.1. El tipo abstracto de datos de los grafos

11.1.1. En Haskell

```haskell
-- Un grafo es una estructura que consta de un conjunto de vértices y un
-- conjunto de aristas que conectan los vértices entre sí. Cada vértice
-- representa una entidad o un elemento, y cada arista representa una
-- relación o conexión entre dos vértices.
--
-- Por ejemplo,
--
--          12
--     1 -------- 2
--     | \78     /|
--     |  \   32/ |
--     |   \   /  |
--   34|     5    |55
--     |   /   \  |
--     |  /44   \ |
--     | /     93\|
--     3 -------- 4
--          61
--
-- representa un grafo no dirigido, lo que significa que las aristas no
-- tienen una dirección específica. Cada arista tiene un peso asociado,
-- que puede representar una medida o una valoración de la relación
-- entre los vértices que conecta.
--
-- El grafo consta de cinco vértices numerados del 1 al 5. Las aristas
-- especificadas en la lista indican las conexiones entre los vértices y
-- sus respectivos pesos. Por ejemplo, la arista (1,2,12) indica que
-- existe una conexión entre el vértice 1 y el vértice 2 con un peso de
-- 12.
--
-- En el grafo representado, se pueden observar las conexiones entre los
-- vértices de la siguiente manera:
-- + El vértice 1 está conectado con el vértice 2 (peso 12), el vértice
--   3 (peso 34) y el vértice 5 (peso 78).
-- + El vértice 2 está conectado con el vértice 4 (peso 55) y el vértice
--   5 (peso 32).
-- + El vértice 3 está conectado con el vértice 4 (peso 61) y el vértice
```

```
--   5 (peso 44).
-- + El vértice 4 está conectado con el vértice 5 (peso 93).
--
-- Las operaciones del tipo abstracto de datos (TAD) de los grafos son
--    creaGrafo  :: (Ix v, Num p, Ord v, Ord p) =>
--                  Orientacion -> (v,v) -> [(v,v,p)] -> Grafo v p
--    creaGrafo' :: (Ix v, Num p, Ord v, Ord p) =>
--                  Orientacion -> (v,v) -> [(v,v)] -> Grafo v p
--    dirigido   :: (Ix v,Num p) => (Grafo v p) -> Bool
--    adyacentes :: (Ix v,Num p) => (Grafo v p) -> v -> [v]
--    nodos      :: (Ix v,Num p) => (Grafo v p) -> [v]
--    aristas    :: (Ix v,Num p) => (Grafo v p) -> [(v,v,p)]
--    aristaEn   :: (Ix v,Num p) => (Grafo v p) -> (v,v) -> Bool
--    peso       :: (Ix v,Num p) => v -> v -> (Grafo v p) -> p
-- tales que
--    + (creaGrafo o cs as) es un grafo (dirigido o no, según el valor
--       de o), con el par de cotas cs y listas de aristas as (cada
--       arista es un trío formado por los dos vértices y su peso).
--    + creaGrafo' es la versión de creaGrafo para los grafos sin pesos.
--    + (dirigido g) se verifica si g es dirigido.
--    + (nodos g) es la lista de todos los nodos del grafo g.
--    + (aristas g) es la lista de las aristas del grafo g.
--    + (adyacentes g v) es la lista de los vértices adyacentes al nodo
--      v en el grafo g.
--    + (aristaEn g a) se verifica si a es una arista del grafo g.
--    + (peso v1 v2 g) es el peso de la arista que une los vértices v1 y
--      v2 en el grafo g.
--
-- Usando el TAD de los grafos, el grafo anterior se puede definir por
--    creaGrafo ND (1,5) [(1,2,12),(1,3,34),(1,5,78),
--                        (2,4,55),(2,5,32),
--                        (3,4,61),(3,5,44),
--                        (4,5,93)]
-- con los siguientes argumentos:
--    + ND: Es un parámetro de tipo Orientacion que indica si el grafo
--      es dirigido o no. En este caso, se utiliza ND, lo que significa
--      ”no dirigido”. Por lo tanto, el grafo creado será no dirigido,
--      lo que implica que las aristas no tienen una dirección
--      específica.
--    + (1,5): Es el par de cotas que define los vértices del grafo. En
```

```
--      este caso, el grafo tiene vértices numerados desde 1 hasta 5.
--    + [(1,2,12),(1,3,34),(1,5,78),(2,4,55),(2,5,32),(3,4,61),(3,5,44),(4,5,93)]
--      Es una lista de aristas, donde cada arista está representada por
--      un trío de valores. Cada trío contiene los dos vértices que
--      están conectados por la arista y el peso de dicha arista.
--
-- Para usar el TAD hay que usar una implementación concreta. En
-- principio, consideraremos las siguientes:
--    + mediante vectores de adyacencia y
--    + mediante matriz de adyacencia.
-- Hay que elegir la que se desee utilizar, descomentándola y comentando
-- las otras.

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

module TAD.Grafo
  (Orientacion (..),
   Grafo,
   creaGrafo,
   creaGrafo',
   dirigido,
   adyacentes,
   nodos,
   aristas,
   aristaEn,
   peso
  ) where

import TAD.GrafoConListaDeAdyacencia
-- import TAD.GrafoConVectorDeAdyacencia
-- import TAD.GrafoConMatrizDeAdyacencia
```

11.1.2. En Python

```
# Un grafo es una estructura que consta de un conjunto de vértices y un
# conjunto de aristas que conectan los vértices entre sí. Cada vértice
# representa una entidad o un elemento, y cada arista representa una
# relación o conexión entre dos vértices.
#
# Por ejemplo,
```

```
#
#          12
#     1 -------- 2
#     | \78     /|
#     |  \   32/ |
#     |   \   /  |
#   34|     5    |55
#     |   /   \  |
#     |  /44   \ |
#     | /     93\|
#     3 -------- 4
#          61
#
# representa un grafo no dirigido, lo que significa que las aristas no
# tienen una dirección específica. Cada arista tiene un peso asociado,
# que puede representar una medida o una valoración de la relación
# entre los vértices que conecta.
#
# El grafo consta de cinco vértices numerados del 1 al 5. Las aristas
# especificadas en la lista indican las conexiones entre los vértices y
# sus respectivos pesos. Por ejemplo, la arista (1,2,12) indica que
# existe una conexión entre el vértice 1 y el vértice 2 con un peso de
# 12.
#
# En el grafo representado, se pueden observar las conexiones entre los
# vértices de la siguiente manera:
# + El vértice 1 está conectado con el vértice 2 (peso 12), el vértice
#   3 (peso 34) y el vértice 5 (peso 78).
# + El vértice 2 está conectado con el vértice 4 (peso 55) y el vértice
#   5 (peso 32).
# + El vértice 3 está conectado con el vértice 4 (peso 61) y el vértice
#   5 (peso 44).
# + El vértice 4 está conectado con el vértice 5 (peso 93).
#
# Las operaciones del tipo abstracto de datos (TAD) de los grafos son
#    creaGrafo
#    creaGrafo_
#    dirigido
#    adyacentes
#    nodos
```

```
#    aristas
#    aristaEn
#    peso
# tales que
#    + creaGrafo(o, cs, as) es un grafo (dirigido o no, según el valor
#       de o), con el par de cotas cs y listas de aristas as (cada
#       arista es un trío formado por los dos vértices y su peso). Ver
#       un ejemplo en el siguiente apartado.
#    + creaGrafo_ es la versión de creaGrafo para los grafos sin pesos.
#    + dirigido(g) se verifica si g es dirigido.
#    + nodos(g) es la lista de todos los nodos del grafo g.
#    + aristas(g) es la lista de las aristas del grafo g.
#    + adyacentes(g, v) es la lista de los vértices adyacentes al nodo
#      v en el grafo g.
#    + aristaEn(g, a) se verifica si a es una arista del grafo g.
#    + peso(v1, v2, g) es el peso de la arista que une los vértices v1 y
#      v2 en el grafo g.
#
# Usando el TAD de los grafos, el grafo anterior se puede definir por
#    creaGrafo ND (1,5) [(1,2,12),(1,3,34),(1,5,78),
#                        (2,4,55),(2,5,32),
#                        (3,4,61),(3,5,44),
#                        (4,5,93)]
# con los siguientes argumentos:
#    + ND: Es un parámetro de tipo Orientacion que indica si el grafo
#      es dirigido o no. En este caso, se utiliza ND, lo que significa
#      ”no dirigido”. Por lo tanto, el grafo creado será no dirigido,
#      lo que implica que las aristas no tienen una dirección
#      específica.
#    + (1,5): Es el par de cotas que define los vértices del grafo. En
#      este caso, el grafo tiene vértices numerados desde 1 hasta 5.
#    + [(1,2,12),(1,3,34),(1,5,78),(2,4,55),(2,5,32),(3,4,61),(3,5,44),(4,5,93)]:
#      Es una lista de aristas, donde cada arista está representada por
#      un trío de valores. Cada trío contiene los dos vértices que
#      están conectados por la arista y el peso de dicha arista.
#
# Para usar el TAD hay que usar una implementación concreta. En
# principio, consideraremos sólo la siguiente:
#    + mediante lista de adyacencia.
```

```python
# pylint: disable=unused-import

__all__ = [
    'Orientacion',
    'Grafo',
    'Vertice',
    'Peso',
    'creaGrafo',
    'creaGrafo_',
    'dirigido',
    'adyacentes',
    'nodos',
    'aristas',
    'aristaEn',
    'peso'
    ]

from src.TAD.GrafoConListaDeAdyacencia import (Arista, Cotas, Grafo,
                                               Orientacion, Peso, Vertice,
                                               adyacentes, aristaEn, aristas,
                                               creaGrafo, creaGrafo_, dirigido,
                                               nodos, peso)
```

11.2. El TAD de los grafos mediante listas de adyacencia

11.2.1. En Haskell

```haskell
{-# OPTIONS_GHC -fno-warn-unused-top-binds #-}

{-# LANGUAGE FlexibleInstances #-}

module TAD.GrafoConListaDeAdyacencia
    (Orientacion (..),
     Grafo,
     creaGrafo,
     creaGrafo',
     dirigido,
     adyacentes,
```

```
    nodos,
    aristas,
    aristaEn,
    peso
   ) where

-- Librerías auxiliares                                              --
import Data.Ix (Ix, range)
import Data.List (sort, nub)

-- Orientacion es D (dirigida) ó ND (no dirigida).
data Orientacion = D | ND
  deriving (Eq, Show)

-- (Grafo v p) es un grafo con vértices de tipo v y pesos de tipo p.
data Grafo v p = G Orientacion ([v],[((v,v),p)])
  deriving Eq

-- (escribeGrafo g) es la cadena correspondiente al grafo g. Por
-- ejemplo,
--    λ> escribeGrafo (creaGrafo ND (1,3) [(1,2,0),(2,3,5),(2,2,0)])
--    "G ND ([1,2,3],[((1,2),0),((2,2),0),((2,3),5)])"
--    λ> escribeGrafo (creaGrafo D (1,3) [(1,2,0),(2,3,5),(2,2,0)])
--    "G D ([1,2,3],[((1,2),0),((2,2),0),((2,3),5)])"
--    λ> escribeGrafo (creaGrafo ND (1,3) [(1,2,0),(2,3,0),(2,2,0)])
--    "G ND ([1,2,3],[(1,2),(2,2),(2,3)])"
--    λ> escribeGrafo (creaGrafo D (1,3) [(1,2,0),(2,3,0),(2,2,0)])
--    "G D ([1,2,3],[(1,2),(2,2),(2,3)])"
--    λ> escribeGrafo (creaGrafo D (1,3) [(1,2,0),(3,2,0),(2,2,0)])
--    "G D ([1,2,3],[(1,2),(2,2),(3,2)])"
--    λ> escribeGrafo (creaGrafo ND (1,3) [(1,2,0),(3,2,0),(2,2,0)])
--    "G ND ([1,2,3],[(1,2),(2,2),(2,3)])"
escribeGrafo :: (Ix v,Num p,Eq p,Show v,Show p) => Grafo v p -> String
escribeGrafo (G o (vs,as)) =
  "G " ++ show o ++ " (" ++ show vs ++ "," ++ escribeAristas ++ ")"
  where
    aristasReducidas
      | o == D    = as
      | otherwise = [((x,y),p) | ((x,y),p) <- as, x <= y]
    escribeAristas
```

```
      | ponderado = show aristasReducidas
      | otherwise = show [a | (a,_) <- aristasReducidas]
    ponderado = any (\((_,_),p) -> p /= 0) as

-- Procedimiento de escritura de grafos
instance (Ix v,Num p,Eq p,Show v,Show p) => Show (Grafo v p) where
  show = escribeGrafo

-- (creaGrafo o cs as) es un grafo (dirigido o no, según el valor de o),
-- con el par de cotas cs y listas de aristas as (cada arista es un trío
-- formado por los dos vértices y su peso). Por ejemplo,
--    λ> creaGrafo ND (1,3) [(1,2,12),(1,3,34)]
--    G ND ([1,2,3],[((1,2),12),((1,3),34),((2,1),12),((3,1),34)])
--    λ> creaGrafo D (1,3) [(1,2,12),(1,3,34)]
--    G D ([1,2,3],[((1,2),12),((1,3),34)])
--    λ> creaGrafo D (1,4) [(1,2,12),(1,3,34)]
--    G D ([1,2,3,4],[((1,2),12),((1,3),34)])
creaGrafo :: (Ix v, Num p, Ord v, Ord p) =>
             Orientacion -> (v,v) -> [(v,v,p)] -> Grafo v p
creaGrafo o cs as =
  G o (range cs,
       sort ([((x1,x2),p) | (x1,x2,p) <- as] ++
             if o == D then []
             else [((x2,x1),p) | (x1,x2,p) <- as, x1 /= x2]))

-- (creaGrafo' o cs as) es un grafo (dirigido o no, según el valor de o),
-- con el par de cotas cs y listas de aristas as (cada arista es un par
-- de vértices y se supone que su peso es 0). Por ejemplo,
--    λ> creaGrafo' ND (1,3) [(2,1),(1,3)]
--    G ND ([1,2,3],[(1,2),(1,3)])
--    λ> creaGrafo' D (1,3) [(2,1),(1,3)]
--    G D ([1,2,3],[(1,3),(2,1)])
creaGrafo' :: (Ix v, Num p, Ord v, Ord p) =>
              Orientacion -> (v,v) -> [(v,v)] -> Grafo v p
creaGrafo' o cs as =
  creaGrafo o cs [(v1,v2,0) | (v1,v2) <- as]

-- ejGrafoND es el grafo
--              12
--         1 -------- 2
```

```
--        | \78     /|
--        |  \   32/ |
--        |   \   /  |
--      34|     5    |55
--        |   /   \  |
--        |  /44   \ |
--        | /     93\|
--        3 -------- 4
--             61
-- Se define por
ejGrafoND :: Grafo Int Int
ejGrafoND = creaGrafo ND (1,5) [(1,2,12),(1,3,34),(1,5,78),
                                (2,4,55),(2,5,32),
                                (3,4,61),(3,5,44),
                                (4,5,93)]

ejGrafoD :: Grafo Int Int
ejGrafoD = creaGrafo D (1,5) [(1,2,12),(1,3,34),(1,5,78),
                              (2,4,55),(2,5,32),
                              (3,4,61),(3,5,44),
                              (4,5,93)]

-- (dirigido g) se verifica si g es dirigido. Por ejemplo,
--    dirigido ejGrafoD   ==  True
--    dirigido ejGrafoND  ==  False
dirigido :: (Ix v,Num p) => Grafo v p -> Bool
dirigido (G o _) = o == D

-- (nodos g) es la lista de todos los nodos del grafo g. Por ejemplo,
--    nodos ejGrafoND  ==  [1,2,3,4,5]
--    nodos ejGrafoD   ==  [1,2,3,4,5]
nodos :: (Ix v,Num p) => Grafo v p -> [v]
nodos (G _ (ns,_)) = ns

-- (adyacentes g v) es la lista de los vértices adyacentes al nodo v en
-- el grafo g. Por ejemplo,
--    adyacentes ejGrafoND 4  ==  [2,3,5]
--    adyacentes ejGrafoD  4  ==  [5]
adyacentes :: (Ix v, Num p) => Grafo v p -> v -> [v]
adyacentes (G _ (_,e)) v = nub [u | ((w,u),_) <- e, w == v]
```

```haskell
-- (aristaEn g a) se verifica si a es una arista del grafo g. Por
-- ejemplo,
--    aristaEn ejGrafoND (5,1)  ==  True
--    aristaEn ejGrafoND (4,1)  ==  False
--    aristaEn ejGrafoD  (5,1)  ==  False
--    aristaEn ejGrafoD  (1,5)  ==  True
aristaEn :: (Ix v,Num p) => Grafo v p -> (v,v) -> Bool
aristaEn g (x,y) = y `elem` adyacentes g x

-- (peso v1 v2 g) es el peso de la arista que une los vértices v1 y v2
-- en el grafo g. Por ejemplo,
--    peso 1 5 ejGrafoND  ==  78
--    peso 1 5 ejGrafoD   ==  78
peso :: (Ix v,Num p) => v -> v -> Grafo v p -> p
peso x y (G _ (_,gs)) = head [c | ((x',y'),c) <- gs, (x,y) == (x',y')]

-- (aristas g) es la lista de las aristas del grafo g. Por ejemplo,
--    λ> aristas ejGrafoD
--    [((1,2),12),((1,3),34),((1,5),78),
--     ((2,4),55),((2,5),32),
--     ((3,4),61),((3,5),44),
--     ((4,5),93)]
--    λ> aristas ejGrafoND
--    [((1,2),12),((1,3),34),((1,5),78),
--     ((2,1),12),((2,4),55),((2,5),32),
--     ((3,1),34),((3,4),61),((3,5),44),
--     ((4,2),55),((4,3),61),((4,5),93),
--     ((5,1),78),((5,2),32),((5,3),44),((5,4),93)]
aristas :: (Ix v,Num p) => Grafo v p -> [((v,v),p)]
aristas (G _ (_,g)) = [((v1,v2),p) | ((v1,v2),p) <- g]
```

11.2.2. En Python

```python
# Se define la clase Grafo con los siguientes métodos:
#    + dirigido() se verifica si el grafo es dirigido.
#    + nodos() es la lista de todos los nodos del grafo.
#    + aristas() es la lista de las aristas del grafo.
#    + adyacentes(v) es la lista de los vértices adyacentes al vértice
#      v en el grafo.
```

```
#    + aristaEn(a) se verifica si a es una arista del grafo.
#    + peso(v1, v2) es el peso de la arista que une los vértices v1 y
#      v2 en el grafo.
# Por ejemplo,
#    >>> Grafo(Orientacion.D, (1,3), [((1,2),0),((3,2),0),((2,2),0)])
#    G D ([1, 2, 3], [(1, 2), (2, 2), (3, 2)])
#    >>> Grafo(Orientacion.ND, (1,3), [((1,2),0),((3,2),0),((2,2),0)])
#    G ND ([1, 2, 3], [(1, 2), (2, 2), (2, 3)])
#    >>> Grafo(Orientacion.ND, (1,3), [((1,2),0),((3,2),5),((2,2),0)])
#    G ND ([1, 2, 3], [((1, 2), 0), ((2, 2), 0), ((2, 3), 5)])
#    >>> Grafo(Orientacion.D, (1,3), [((1,2),0),((3,2),5),((2,2),0)])
#    G D ([1, 2, 3], [((1, 2), 0), ((2, 2), 0), ((3, 2), 5)])
#    >>> ejGrafoND: Grafo = Grafo(Orientacion.ND,
#                                (1, 5),
#                                [((1, 2), 12), ((1, 3), 34), ((1, 5), 78),
#                                 ((2, 4), 55), ((2, 5), 32),
#                                 ((3, 4), 61), ((3, 5), 44),
#                                 ((4, 5), 93)])
#    >>> ejGrafoND
#    G ND ([1, 2, 3, 4, 5],
#          [((1, 2), 12), ((1, 3), 34), ((1, 5), 78),
#           ((2, 4), 55), ((2, 5), 32),
#           ((3, 4), 61), ((3, 5), 44),
#           ((4, 5), 93)])
#    >> ejGrafoD: Grafo = Grafo(Orientacion.D,
#                              (1,5),
#                              [((1, 2), 12), ((1, 3), 34), ((1, 5), 78),
#                               ((2, 4), 55), ((2, 5), 32),
#                               ((3, 4), 61), ((3, 5), 44),
#                               ((4, 5), 93)])
#    >>> ejGrafoD
#    G D ([1, 2, 3, 4, 5],
#         [((1, 2), 12), ((1, 3), 34), ((1, 5), 78),
#          ((2, 4), 55), ((2, 5), 32),
#          ((3, 4), 61), ((3, 5), 44),
#          ((4, 5), 93)])
#    >>> ejGrafoD.dirigido()
#    True
#    >>> ejGrafoND.dirigido()
#    False
```

```
#    >>> ejGrafoND.nodos()
#    [1, 2, 3, 4, 5]
#    >>> ejGrafoD.nodos()
#    [1, 2, 3, 4, 5]
#    >>> ejGrafoND.adyacentes(4)
#    [2, 3, 5]
#    >>> ejGrafoD.adyacentes(4)
#    [5]
#    >>> ejGrafoND.aristaEn((5, 1))
#    True
#    >>> ejGrafoND.aristaEn((4, 1))
#    False
#    >>> ejGrafoD.aristaEn((5, 1))
#    False
#    >>> ejGrafoD.aristaEn((1, 5))
#    True
#    >>> ejGrafoND.peso(1, 5)
#    78
#    >>> ejGrafoD.peso(1, 5)
#    78
#    >>> ejGrafoD._aristas
#    [((1, 2), 12), ((1, 3), 34), ((1, 5), 78),
#     ((2, 4), 55), ((2, 5), 32),
#     ((3, 4), 61), ((3, 5), 44),
#     ((4, 5), 93)]
#    >>> ejGrafoND._aristas
#    [((1, 2), 12), ((1, 3), 34), ((1, 5), 78),
#     ((2, 1), 12), ((2, 4), 55), ((2, 5), 32),
#     ((3, 1), 34), ((3, 4), 61), ((3, 5), 44),
#     ((4, 2), 55), ((4, 3), 61), ((4, 5), 93),
#     ((5, 1), 78), ((5, 2), 32), ((5, 3), 44),
#     ((5, 4), 93)]
#
# Además se definen las correspondientes funciones. Por ejemplo,
#    >>> creaGrafo(Orientacion.ND, (1,3), [((1,2),12),((1,3),34)])
#    G ND ([1, 2, 3], [((1, 2), 12), ((1, 3), 34), ((2, 1), 12), ((3, 1), 34)])
#    >>> creaGrafo(Orientacion.D, (1,3), [((1,2),12),((1,3),34)])
#    G D ([1, 2, 3], [((1, 2), 12), ((1, 3), 34)])
#    >>> creaGrafo(Orientacion.D, (1,4), [((1,2),12),((1,3),34)])
#    G D ([1, 2, 3, 4], [((1, 2), 12), ((1, 3), 34)])
```

```
#    >>> ejGrafoND2: Grafo = creaGrafo(Orientacion.ND,
#                                      (1,5),
#                                      [((1,2),12),((1,3),34),((1,5),78),
#                                       ((2,4),55),((2,5),32),
#                                       ((3,4),61),((3,5),44),
#                                       ((4,5),93)])
#    >>> ejGrafoND2
#    G ND ([1, 2, 3, 4, 5],
#          [((1, 2), 12), ((1, 3), 34), ((1, 5), 78),
#           ((2, 4), 55), ((2, 5), 32),
#           ((3, 4), 61), ((3, 5), 44),
#           ((4, 5), 93)])
#    >>> ejGrafoD2: Grafo = creaGrafo(Orientacion.D,
#                                     (1,5),
#                                     [((1,2),12),((1,3),34),((1,5),78),
#                                      ((2,4),55),((2,5),32),
#                                      ((3,4),61),((3,5),44),
#                                      ((4,5),93)])
#    >>> ejGrafoD2
#    G D ([1, 2, 3, 4, 5],
#         [((1, 2), 12), ((1, 3), 34), ((1, 5), 78),
#          ((2, 4), 55), ((2, 5), 32),
#          ((3, 4), 61), ((3, 5), 44),
#          ((4, 5), 93)])
#    >>> creaGrafo_(Orientacion.D, (1,3), [(2, 1), (1, 3)])
#    G D ([1, 2, 3], [(1, 3), (2, 1)])
#    >>> creaGrafo_(Orientacion.ND, (1,3), [(2, 1), (1, 3)])
#    G ND ([1, 2, 3], [(1, 2), (1, 3)])
#    >>> dirigido(ejGrafoD2)
#    True
#    >>> dirigido(ejGrafoND2)
#    False
#    >>> nodos(ejGrafoND2)
#    [1, 2, 3, 4, 5]
#    >>> nodos(ejGrafoD2)
#    [1, 2, 3, 4, 5]
#    >>> adyacentes(ejGrafoND2, 4)
#    [2, 3, 5]
#    >>> adyacentes(ejGrafoD2, 4)
#    [5]
```

```
#    >>> aristaEn(ejGrafoND2, (5,1))
#    True
#    >>> aristaEn(ejGrafoND2, (4,1))
#    False
#    >>> aristaEn(ejGrafoD2, (5,1))
#    False
#    >>> aristaEn(ejGrafoD2, (1,5))
#    True
#    >>> peso(1, 5, ejGrafoND2)
#    78
#    >>> peso(1, 5, ejGrafoD2)
#    78
#    >>> aristas(ejGrafoD2)
#    [((1, 2), 12), ((1, 3), 34), ((1, 5), 78),
#     ((2, 4), 55), ((2, 5), 32),
#     ((3, 4), 61), ((3, 5), 44),
#     ((4, 5), 93)]
#    >>> aristas(ejGrafoND2)
#    [((1, 2), 12), ((1, 3), 34), ((1, 5), 78),
#     ((2, 1), 12), ((2, 4), 55), ((2, 5), 32),
#     ((3, 1), 34), ((3, 4), 61), ((3, 5), 44),
#     ((4, 2), 55), ((4, 3), 61), ((4, 5), 93),
#     ((5, 1), 78), ((5, 2), 32), ((5, 3), 44), ((5, 4), 93)]

# pylint: disable=protected-access

from enum import Enum

Orientacion = Enum('Orientacion', ['D', 'ND'])

Vertice = int
Cotas = tuple[Vertice, Vertice]
Peso = float
Arista = tuple[tuple[Vertice, Vertice], Peso]

class Grafo:
    def __init__(self,
                 _orientacion: Orientacion,
                 _cotas: Cotas,
                 _aristas: list[Arista]):
```

```
        self._orientacion = _orientacion
        self._cotas = _cotas
        if _orientacion == Orientacion.ND:
            simetricas = [((v2, v1), p) for ((v1, v2), p)
                          in _aristas
                          if v1 != v2]
            self._aristas = sorted(_aristas + simetricas)
        else:
            self._aristas = sorted(_aristas)

    def nodos(self) -> list[Vertice]:
        (x, y) = self._cotas
        return list(range(x, 1 + y))

    def __repr__(self) -> str:
        o = self._orientacion
        vs = nodos(self)
        ns = self._aristas
        escribeOrientacion = "D" if o == Orientacion.D else "ND"
        ponderado = {p for ((_, _), p) in ns} != {0}
        aristasReducidas = ns if o == Orientacion.D \
            else [((x, y), p)
                  for ((x, y), p) in ns
                  if x <= y]
        escribeAristas = str(aristasReducidas) if ponderado \
            else str([a for (a, _) in aristasReducidas])
        return f"G {escribeOrientacion} ({vs}, {escribeAristas})"

    def dirigido(self) -> bool:
        return self._orientacion == Orientacion.D

    def adyacentes(self, v: int) -> list[int]:
        return list(set(u for ((w, u), _)
                        in self._aristas
                        if w == v))

    def aristaEn(self, a: tuple[Vertice, Vertice]) -> bool:
        (x, y) = a
        return y in self.adyacentes(x)
```

```
    def peso(self, v1: Vertice, v2: Vertice) -> Peso:
        return [p for ((x1, x2), p)
                in self._aristas
                if (x1, x2) == (v1, v2)][0]

def creaGrafo(o: Orientacion,
              cs: Cotas,
              as_: list[Arista]) -> Grafo:
    return Grafo(o, cs, as_)

def creaGrafo_(o: Orientacion,
               cs: Cotas,
               as_: list[tuple[Vertice, Vertice]]) -> Grafo:
    return Grafo(o, cs, [((v1, v2), 0) for (v1, v2) in as_])

def dirigido(g: Grafo) -> bool:
    return g.dirigido()

def nodos(g: Grafo) -> list[Vertice]:
    return g.nodos()

def adyacentes(g: Grafo, v: Vertice) -> list[Vertice]:
    return g.adyacentes(v)

def aristaEn(g: Grafo, a: tuple[Vertice, Vertice]) -> bool:
    return g.aristaEn(a)

def peso(v1: Vertice, v2: Vertice, g: Grafo) -> Peso:
    return g.peso(v1, v2)

def aristas(g: Grafo) -> list[Arista]:
    return g._aristas

# En los ejemplos se usarán los grafos (no dirigido y dirigido)
# correspondientes a
#             12
#        1 -------- 2
#        | \78     /|
#        |  \   32/ |
#        |   \   /  |
```

```
#      34|      5     |55
#        |    /    \   |
#        |   /44    \  |
#        | /      93\|
#        3 -------- 4
#             61
# definidos por
ejGrafoND: Grafo = Grafo(Orientacion.ND,
                         (1, 5),
                         [((1, 2), 12), ((1, 3), 34), ((1, 5), 78),
                          ((2, 4), 55), ((2, 5), 32),
                          ((3, 4), 61), ((3, 5), 44),
                          ((4, 5), 93)])
ejGrafoD: Grafo = Grafo(Orientacion.D,
                        (1,5),
                        [((1, 2), 12), ((1, 3), 34), ((1, 5), 78),
                         ((2, 4), 55), ((2, 5), 32),
                         ((3, 4), 61), ((3, 5), 44),
                         ((4, 5), 93)])

ejGrafoND2: Grafo = creaGrafo(Orientacion.ND,
                              (1,5),
                              [((1,2),12),((1,3),34),((1,5),78),
                               ((2,4),55),((2,5),32),
                               ((3,4),61),((3,5),44),
                               ((4,5),93)])

ejGrafoD2: Grafo = creaGrafo(Orientacion.D,
                             (1,5),
                             [((1,2),12),((1,3),34),((1,5),78),
                              ((2,4),55),((2,5),32),
                              ((3,4),61),((3,5),44),
                              ((4,5),93)])
```

11.3. Grafos completos

11.3.1. En Haskell

```
-- ---------------------------------------------------------------------
-- El grafo completo de orden n, K(n), es un grafo no dirigido cuyos
```

```haskell
-- conjunto de vértices es {1,..n} y tiene una arista entre cada par de
-- vértices distintos.
--
-- Usando el [tipo abstracto de datos de los grafos](https://bit.ly/45cQ3Fo),
-- definir la función,
--    completo :: Int -> Grafo Int Int
-- tal que (completo n) es el grafo completo de orden n. Por ejemplo,
--    λ> completo 4
--    G ND ([1,2,3,4],[(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)])
-- ---------------------------------------------------------------------

module Grafo_Grafos_completos where

import TAD.Grafo (Grafo, Orientacion (ND), creaGrafo')
import Test.Hspec (Spec, hspec, it, shouldBe)

completo :: Int -> Grafo Int Int
completo n =
  creaGrafo' ND (1,n) [(x,y) | x <- [1..n], y <- [x+1..n]]

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

spec :: Spec
spec = do
  it "e1" $
    show (completo 4) `shouldBe`
    "G ND ([1,2,3,4],[(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)])"

-- La verificación es
--    λ> verifica
--
--    e1
--
--    Finished in 0.0004 seconds
--    1 example, 0 failures
```

11.3.2. En Python

```python
# ---------------------------------------------------------------------
# El grafo completo de orden n, K(n), es un grafo no dirigido cuyos
# conjunto de vértices es {1,..n} y tiene una arista entre cada par de
# vértices distintos.
#
# Usando el [tipo abstracto de datos de los grafos](https://bit.ly/45cQ3Fo),
# definir la función,
#    completo : (int) -> Grafo
# tal que completo(n) es el grafo completo de orden n. Por ejemplo,
#    >>> completo(4)
#    G ND ([1, 2, 3, 4],
#          [((1, 2), 0), ((1, 3), 0), ((1, 4), 0),
#           ((2, 1), 0), ((2, 3), 0), ((2, 4), 0),
#           ((3, 1), 0), ((3, 2), 0), ((3, 4), 0),
#           ((4, 1), 0), ((4, 2), 0), ((4, 3), 0)])
# ---------------------------------------------------------------------

from src.TAD.Grafo import Grafo, Orientacion, creaGrafo_


def completo(n: int) -> Grafo:
    return creaGrafo_(Orientacion.ND,
                      (1, n),
                      [(x, y)
                       for x in range(1, n + 1)
                       for y in range(x + 1, n+1)])

# Verificación
# ============

def test_completo() -> None:
    assert str(completo(4)) == \
        "G ND ([1, 2, 3, 4], [(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)])"
    print("Verificado")

# La verificación es
#    >>> test_completo()
#    Verificado
```

11.4. Grafos ciclos

11.4.1. En Haskell

```
-- ---------------------------------------------------------------------
-- El ciclo de orden n, C(n), es un grafo no dirigido cuyo conjunto de
-- vértices es {1,...,n} y las aristas son
--    (1,2), (2,3), ..., (n-1,n), (n,1)
--
-- Usando el [tipo abstracto de datos de los grafos](https://bit.ly/45cQ3Fo),
-- definir la función,
--    grafoCiclo :: Int -> Grafo Int Int
-- tal que (grafoCiclo n) es el grafo ciclo de orden n. Por ejemplo,
--    λ> grafoCiclo 3
--    G ND ([1,2,3],[(1,2),(1,3),(2,3)])
-- ---------------------------------------------------------------------

module Grafo_Grafos_ciclos where

import TAD.Grafo (Grafo, Orientacion (ND), creaGrafo')
import Test.Hspec (Spec, hspec, it, shouldBe)

grafoCiclo :: Int -> Grafo Int Int
grafoCiclo n =
  creaGrafo' ND (1,n) ((n,1):[(x,x+1) | x <- [1..n-1]])

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

spec :: Spec
spec = do
  it "e1" $
    show (grafoCiclo 3) `shouldBe`
    "G ND ([1,2,3],[(1,2),(1,3),(2,3)])"

-- La verificación es
--    λ> verifica
--
```

```
--    e1
--
--    Finished in 0.0006 seconds
--    1 example, 0 failures
```

11.4.2. En Python

```python
# ---------------------------------------------------------------------
# El ciclo de orden n, C(n), es un grafo no dirigido cuyo conjunto de
# vértices es {1,...,n} y las aristas son
#    (1,2), (2,3), ..., (n-1,n), (n,1)
#
# Usando el [tipo abstracto de datos de los grafos](https://bit.ly/45cQ3Fo),
# definir la función,
#    grafoCiclo : (Int) -> Grafo
# tal que grafoCiclo(n) es el grafo ciclo de orden n. Por ejemplo,
#    >>> grafoCiclo(3)
#    G ND ([1, 2, 3], [(1, 2), (1, 3), (2, 3)])
# ---------------------------------------------------------------------

from src.TAD.Grafo import Grafo, Orientacion, creaGrafo_


def grafoCiclo(n: int) -> Grafo:
    return creaGrafo_(Orientacion.ND,
                      (1, n),
                      [(n,1)] + [(x, x + 1) for x in range(1, n)])

# Verificación
# ============

def test_grafoCiclo() -> None:
    assert str(grafoCiclo(3)) == \
        "G ND ([1, 2, 3], [(1, 2), (1, 3), (2, 3)])"
    print("Verificado")

# La verificación es
#    >>> test_grafoCiclo()
#    Verificado
```

11.5. Número de vértices

11.5.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Usando el [tipo abstracto de datos de los grafos](https://bit.ly/45cQ3Fo),
-- definir la función,
--    nVertices :: (Ix v, Num p) => Grafo v p ->  Int
-- tal que (nVertices g) es el número de vértices del grafo g. Por
-- ejemplo,
--    nVertices (creaGrafo' D (1,5) [(1,2),(3,1)])   ==  5
--    nVertices (creaGrafo' ND (0,5) [(1,2),(3,1)])  ==  6
-- ---------------------------------------------------------------------

module Grafo_Numero_de_vertices where

import TAD.Grafo (Grafo, Orientacion (D, ND), nodos, creaGrafo')
import Data.Ix (Ix)
import Test.Hspec (Spec, hspec, it, shouldBe)

nVertices :: (Ix v, Num p) => Grafo v p ->  Int
nVertices = length . nodos

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

spec :: Spec
spec = do
  it "e1" $
    nVertices (creaGrafo' D (1,5) [(1,2),(3,1)] :: Grafo Int Int)  `shouldBe` 5
  it "e2" $
    nVertices (creaGrafo' ND (0,5) [(1,2),(3,1)] :: Grafo Int Int) `shouldBe` 6

-- La verificación es
--    λ> verifica
--
--    e1
--    e2
```

```
--
--    Finished in 0.0002 seconds
--    2 examples, 0 failures
```

11.5.2. En Python

```python
# ---------------------------------------------------------------------
# Usando el [tipo abstracto de datos de los grafos](https://bit.ly/45cQ3Fo),
# definir la función,
#    nVertices : (Grafo) -> int
# tal que nVertices(g) es el número de vértices del grafo g. Por
# ejemplo,
#    >>> nVertices(creaGrafo_(Orientacion.D, (1,5), [(1,2),(3,1)]))
#    5
#    >>> nVertices(creaGrafo_(Orientacion.ND, (2,4), [(1,2),(3,1)]))
#    3
# ---------------------------------------------------------------------

from src.TAD.Grafo import Grafo, Orientacion, creaGrafo_, nodos


def nVertices(g: Grafo) -> int:
    return len(nodos(g))

# Verificación
# ============

def test_nVertices() -> None:
    assert nVertices(creaGrafo_(Orientacion.D, (1,5), [(1,2),(3,1)])) == 5
    assert nVertices(creaGrafo_(Orientacion.ND, (2,4), [(1,2),(3,1)])) == 3
    print("Verificado")

# La verificación es
#    >>> test_nVertices()
#    Verificado
```

11.6. Incidentes de un vértice

11.6.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- En un un grafo g, los incidentes de un vértice v es el conjuntos de
-- vértices x de g para los que hay un arco (o una arista) de x a v; es
-- decir, que v es adyacente a x.
--
-- Usando el [tipo abstracto de datos de los grafos](https://bit.ly/45cQ3Fo),
-- definir la función,
--    incidentes :: (Ix v,Num p) => (Grafo v p) -> v -> [v]
-- tal que (incidentes g v) es la lista de los vértices incidentes en el
-- vértice v. Por ejemplo,
--    λ> g1 = creaGrafo' D (1,3) [(1,2),(2,2),(3,1),(3,2)]
--    λ> incidentes g1 1
--    [3]
--    λ> incidentes g1 2
--    [1,2,3]
--    λ> incidentes g1 3
--    []
--    λ> g2 = creaGrafo' ND (1,3) [(1,2),(2,2),(3,1),(3,2)]
--    λ> incidentes g2 1
--    [2,3]
--    λ> incidentes g2 2
--    [1,2,3]
--    λ> incidentes g2 3
--    [1,2]
-- ---------------------------------------------------------------------

module Grafo_Incidentes_de_un_vertice where

import TAD.Grafo (Grafo, Orientacion (D, ND), nodos,  adyacentes, creaGrafo')
import Data.Ix
import Test.Hspec

incidentes :: (Ix v,Num p) => Grafo v p -> v -> [v]
incidentes g v = [x | x <- nodos g, v `elem` adyacentes g x]

-- Verificación
-- ============
```

```
verifica :: IO ()
verifica = hspec spec

spec :: Spec
spec = do
  it "e1" $
    incidentes g1 1 `shouldBe` [3]
  it "e2" $
    incidentes g1 2 `shouldBe` [1,2,3]
  it "e3" $
    incidentes g1 3 `shouldBe` []
  it "e4" $
    incidentes g2 1 `shouldBe` [2,3]
  it "e5" $
    incidentes g2 2 `shouldBe` [1,2,3]
  it "e6" $
    incidentes g2 3 `shouldBe` [1,2]
  where
    g1, g2 :: Grafo Int Int
    g1 = creaGrafo' D (1,3) [(1,2),(2,2),(3,1),(3,2)]
    g2 = creaGrafo' ND (1,3) [(1,2),(2,2),(3,1),(3,2)]

-- La verificación es
--    λ> verifica
--
--    e1
--    e2
--    e3
--    e4
--    e5
--    e6
--
--    Finished in 0.0005 seconds
--    6 examples, 0 failures
```

11.6.2. En Python

```
# ---------------------------------------------------------------------
# En un un grafo g, los incidentes de un vértice v es el conjuntos de
```

```
# vértices x de g para los que hay un arco (o una arista) de x a v; es
# decir, que v es adyacente a x.
#
# Usando el [tipo abstracto de datos de los grafos](https://bit.ly/45cQ3Fo),
# definir la función,
#    incidentes :: (Ix v,Num p) => (Grafo v p) -> v -> [v]
# tal que (incidentes g v) es la lista de los vértices incidentes en el
# vértice v. Por ejemplo,
#    λ> g1 = creaGrafo_(Orientacion.D, (1,3), [(1,2),(2,2),(3,1),(3,2)])
#    λ> incidentes(g1,1)
#    [3]
#    λ> incidentes g1 2
#    [1,2,3]
#    λ> incidentes g1 3
#    []
#    λ> g2 = creaGrafo_(Orientacion.ND, (1,3), [(1,2),(2,2),(3,1),(3,2)])
#    λ> incidentes g2 1
#    [2,3]
#    λ> incidentes g2 2
#    [1,2,3]
#    λ> incidentes g2 3
#    [1,2]
# ---------------------------------------------------------------------

from src.TAD.Grafo import (Grafo, Orientacion, Vertice, adyacentes, creaGrafo_,
                           nodos)


def incidentes(g: Grafo, v: Vertice) -> list[Vertice]:
    return [x for x in nodos(g) if v in adyacentes(g, x)]

# Verificación
# ============

def test_incidentes() -> None:
    g1 = creaGrafo_(Orientacion.D, (1,3), [(1,2),(2,2),(3,1),(3,2)])
    g2 = creaGrafo_(Orientacion.ND, (1,3), [(1,2),(2,2),(3,1),(3,2)])
    assert incidentes(g1,1) == [3]
    assert incidentes(g1,2) == [1, 2, 3]
    assert incidentes(g1,3) == []
```

```
    assert incidentes(g2, 1) == [2, 3]
    assert incidentes(g2, 2) == [1, 2, 3]
    assert incidentes(g2, 3) == [1, 2]
    print("Verificado")

# La verificación es
#    >>> test_incidentes()
#    Verificado
```

11.7. Contiguos de un vértice

11.7.1. En Haskell

```
-- ---------------------------------------------------------------------
-- En un un grafo g, los contiguos de un vértice v es el conjuntos de
-- vértices x de g tales que x es adyacente o incidente con v.
--
-- Usando el [tipo abstracto de datos de los grafos](https://bit.ly/45cQ3Fo),
-- definir la función,
--    contiguos :: (Ix v,Num p) => Grafo v p -> v -> [v]
-- tal que (contiguos g v) es el conjunto de los vértices de g contiguos
-- con el vértice v. Por ejemplo,
--    λ> g1 = creaGrafo' D (1,3) [(1,2),(2,2),(3,1),(3,2)]
--    λ> contiguos g1 1
--    [2,3]
--    λ> contiguos g1 2
--    [2,1,3]
--    λ> contiguos g1 3
--    [1,2]
--    λ> g2 = creaGrafo' ND (1,3) [(1,2),(2,2),(3,1),(3,2)]
--    λ> contiguos g2 1
--    [2,3]
--    λ> contiguos g2 2
--    [1,2,3]
--    λ> contiguos g2 3
--    [1,2]
-- ---------------------------------------------------------------------

module Grafo_Contiguos_de_un_vertice where
```

```haskell
import TAD.Grafo (Grafo, Orientacion (D, ND), adyacentes, creaGrafo')
import Grafo_Incidentes_de_un_vertice (incidentes)
import Data.List (nub)
import Data.Ix
import Test.Hspec

contiguos :: (Ix v,Num p) => Grafo v p -> v -> [v]
contiguos g v = nub (adyacentes g v ++ incidentes g v)

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

spec :: Spec
spec = do
  it "e1" $
    contiguos g1 1 `shouldBe` [2,3]
  it "e2" $
    contiguos g1 2 `shouldBe` [2,1,3]
  it "e3" $
    contiguos g1 3 `shouldBe` [1,2]
  it "e4" $
    contiguos g2 1 `shouldBe` [2,3]
  it "e5" $
    contiguos g2 2 `shouldBe` [1,2,3]
  it "e6" $
    contiguos g2 3 `shouldBe` [1,2]
  where
    g1, g2 :: Grafo Int Int
    g1 = creaGrafo' D (1,3) [(1,2),(2,2),(3,1),(3,2)]
    g2 = creaGrafo' ND (1,3) [(1,2),(2,2),(3,1),(3,2)]

-- La verificación es
--    λ> verifica
--
--    e1
--    e2
--    e3
```

```
--    e4
--    e5
--    e6
--
--    Finished in 0.0005 seconds
--    6 examples, 0 failures
```

11.7.2. En Python

```python
# ---------------------------------------------------------------------
# En un un grafo g, los contiguos de un vértice v es el conjuntos de
# vértices x de g tales que x es adyacente o incidente con v.
#
# Usando el [tipo abstracto de datos de los grafos](https://bit.ly/45cQ3Fo),
# definir la función,
#    contiguos : (Grafo, Vertice) -> list[Vertice]
# tal que (contiguos g v) es el conjunto de los vértices de g contiguos
# con el vértice v. Por ejemplo,
#    >>> g1 = creaGrafo_(Orientacion.D, (1,3), [(1,2),(2,2),(3,1),(3,2)])
#    >>> contiguos(g1, 1)
#    [2, 3]
#    >>> contiguos(g1, 2)
#    [1, 2, 3]
#    >>> contiguos(g1, 3)
#    [1, 2]
#    >>> g2 = creaGrafo_(Orientacion.ND, (1,3), [(1,2),(2,2),(3,1),(3,2)])
#    >>> contiguos(g2, 1)
#    [2, 3]
#    >>> contiguos(g2, 2)
#    [1, 2, 3]
#    >>> contiguos(g2, 3)
#    [1, 2]
# ---------------------------------------------------------------------


from src.Grafo_Incidentes_de_un_vertice import incidentes
from src.TAD.Grafo import Grafo, Orientacion, Vertice, adyacentes, creaGrafo_


def contiguos(g: Grafo, v: Vertice) -> list[Vertice]:
```

```
    return list(set(adyacentes(g, v) + incidentes(g, v)))

# Verificación
# ============

def test_contiguos() -> None:
    g1 = creaGrafo_(Orientacion.D, (1,3), [(1,2),(2,2),(3,1),(3,2)])
    g2 = creaGrafo_(Orientacion.ND, (1,3), [(1,2),(2,2),(3,1),(3,2)])
    assert contiguos(g1, 1) == [2, 3]
    assert contiguos(g1, 2) == [1, 2, 3]
    assert contiguos(g1, 3) == [1, 2]
    assert contiguos(g2, 1) == [2, 3]
    assert contiguos(g2, 2) == [1, 2, 3]
    assert contiguos(g2, 3) == [1, 2]
    print("Verificado")

# La verificación es
#    >>> test_contiguos()
#    Verificado
```

11.8. Lazos de un grafo

11.8.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Usando el [tipo abstracto de datos de los grafos](https://bit.ly/45cQ3Fo),
-- definir las funciones,
--    lazos  :: (Ix v,Num p) => Grafo v p -> [(v,v)]
--    nLazos :: (Ix v,Num p) => Grafo v p ->  Int
-- tales que
-- + (lazos g) es el conjunto de los lazos (es decir, aristas cuyos
--   extremos son iguales) del grafo g. Por ejemplo,
--      λ> ej1 = creaGrafo' D (1,3) [(1,1),(2,3),(3,2),(3,3)]
--      λ> ej2 = creaGrafo' ND (1,3) [(2,3),(3,1)]
--      λ> lazos ej1
--      [(1,1),(3,3)]
--      λ> lazos ej2
--      []
-- + (nLazos g) es el número de lazos del grafo g. Por ejemplo,
--      λ> nLazos ej1
```

```
--      2
--      λ> nLazos ej2
--      0
-- ---------------------------------------------------------------------

module Grafo_Lazos_de_un_grafo where

import TAD.Grafo (Grafo, Orientacion (D, ND), nodos, aristaEn, creaGrafo')
import Data.Ix (Ix)
import Test.Hspec (Spec, hspec, it, shouldBe)

lazos :: (Ix v,Num p) => Grafo v p -> [(v,v)]
lazos g = [(x,x) | x <- nodos g, aristaEn g (x,x)]

nLazos :: (Ix v,Num p) => Grafo v p ->  Int
nLazos = length . lazos

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

spec :: Spec
spec = do
  it "e1" $
    lazos ej1 `shouldBe` [(1,1),(3,3)]
  it "e2" $
    lazos ej2 `shouldBe` []
  it "e3" $
    nLazos ej1 `shouldBe` 2
  it "e4" $
    nLazos ej2 `shouldBe` 0
  where
    ej1, ej2 :: Grafo Int Int
    ej1 = creaGrafo' D (1,3) [(1,1),(2,3),(3,2),(3,3)]
    ej2 = creaGrafo' ND (1,3) [(2,3),(3,1)]

-- La verificación es
--      λ> verifica
```

```
--
--      e1
--      e2
--      e3
--      e4
--
--      Finished in 0.0005 seconds
--      4 examples, 0 failures
```

11.8.2. En Python

```python
# ---------------------------------------------------------------------
# Usando el [tipo abstracto de datos de los grafos](https://bit.ly/45cQ3Fo),
# definir las funciones,
#    lazos  : (Grafo) -> list[tuple[Vertice, Vertice]]
#    nLazos : (Grafo) -> int
# tales que
# + lazos(g) es el conjunto de los lazos (es decir, aristas cuyos
#   extremos son iguales) del grafo g. Por ejemplo,
#      >>> ej1 = creaGrafo_(Orientacion.D, (1,3), [(1,1),(2,3),(3,2),(3,3)])
#      >>> ej2 = creaGrafo_(Orientacion.ND, (1,3), [(2,3),(3,1)])
#      >>> lazos(ej1)
#      [(1,1),(3,3)]
#      >>> lazos(ej2)
#      []
# + nLazos(g) es el número de lazos del grafo g. Por ejemplo,
#      >>> nLazos(ej1)
#      2
#      >>> nLazos(ej2)
#      0
# ---------------------------------------------------------------------

from src.TAD.Grafo import (Grafo, Orientacion, Vertice, aristaEn, creaGrafo_,
                           nodos)


def lazos(g: Grafo) -> list[tuple[Vertice, Vertice]]:
    return [(x, x) for x in nodos(g) if aristaEn(g, (x, x))]

def nLazos(g: Grafo) -> int:
```

```
    return len(lazos(g))

# Verificación
# ============

def test_lazos() -> None:
    ej1 = creaGrafo_(Orientacion.D, (1,3), [(1,1),(2,3),(3,2),(3,3)])
    ej2 = creaGrafo_(Orientacion.ND, (1,3), [(2,3),(3,1)])
    assert lazos(ej1) == [(1,1),(3,3)]
    assert lazos(ej2) == []
    assert nLazos(ej1) == 2
    assert nLazos(ej2) == 0
    print("Verificado")

# La verificación es
#    >>> test_lazos()
#    Verificado
```

11.9. Número de aristas de un grafo

11.9.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Usando el [tipo abstracto de datos de los grafos](https://bit.ly/45cQ3Fo),
-- definir la función,
--    nAristas :: (Ix v,Num p) => Grafo v p ->  Int
-- tal que (nAristas g) es el número de aristas del grafo g. Si g es no
-- dirigido, las aristas de v1 a v2 y de v2 a v1 sólo se cuentan una
-- vez. Por ejemplo,
--    λ> g1 = creaGrafo' ND (1,5) [(1,2),(1,3),(1,5),(2,4),(2,5),(3,4),(3,5),(4,5
--    λ> g2 = creaGrafo' D  (1,5) [(1,2),(1,3),(1,5),(2,4),(2,5),(4,3),(4,5)]
--    λ> g3 = creaGrafo' ND (1,3) [(1,2),(1,3),(2,3),(3,3)]
--    λ> g4 = creaGrafo' ND (1,4) [(1,1),(1,2),(3,3)]
--    λ> nAristas g1
--    8
--    λ> nAristas g2
--    7
--    λ> nAristas g3
--    4
--    λ> nAristas g4
```

```haskell
--    3
--    λ> nAristas (completo 4)
--    6
--    λ> nAristas (completo 5)
--    10
--
-- Definir la función
--    prop_nAristasCompleto :: Int -> Bool
-- tal que (prop_nAristasCompleto n) se verifica si el número de aristas
-- del grafo completo de orden n es n*(n-1)/2 y, usando la función,
-- comprobar que la propiedad se cumple para n de 1 a 20.
-- ---------------------------------------------------------------------

module Grafo_Numero_de_aristas_de_un_grafo where

import TAD.Grafo (Grafo, Orientacion (D, ND), dirigido, aristas, creaGrafo')
import Data.Ix (Ix)
import Grafo_Lazos_de_un_grafo (nLazos)
import Grafo_Grafos_completos (completo)
import Test.Hspec (Spec, hspec, it, shouldBe, describe)

-- 1ª solución
-- ===========

nAristas :: (Ix v,Num p) => Grafo v p ->  Int
nAristas g | dirigido g = length (aristas g)
           | otherwise  = (length (aristas g) + nLazos g) `div` 2

-- 2ª solución
-- ===========

nAristas2 :: (Ix v,Num p) => Grafo v p ->  Int
nAristas2 g | dirigido g = length (aristas g)
            | otherwise  = length [(x,y) | ((x,y),_) <- aristas g, x <= y]

-- Propiedad
-- =========

prop_nAristasCompleto :: Int -> Bool
prop_nAristasCompleto n =
```

```
  nAristas (completo n) == n*(n-1) `div` 2

-- La comprobación es
--    λ> and [prop_nAristasCompleto n | n <- [1..20]]
--    True

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

spec :: Spec
spec = do
  describe "def. 1" $ specG nAristas
  describe "def. 2" $ specG nAristas2

specG :: (Grafo Int Int -> Int) -> Spec
specG nAristas' = do
  it "e1" $
    nAristas' g1 `shouldBe` 8
  it "e2" $
    nAristas' g2 `shouldBe` 7
  it "e3" $
    nAristas' g3 `shouldBe` 4
  it "e4" $
    nAristas' g4 `shouldBe` 3
  it "e5" $
    nAristas' (completo 4) `shouldBe` 6
  it "e6" $
    nAristas' (completo 5) `shouldBe` 10
  where
    g1, g2, g3, g4 :: Grafo Int Int
    g1 = creaGrafo' ND (1,5) [(1,2),(1,3),(1,5),(2,4),(2,5),(3,4),(3,5),(4,5)]
    g2 = creaGrafo' D  (1,5) [(1,2),(1,3),(1,5),(2,4),(2,5),(4,3),(4,5)]
    g3 = creaGrafo' ND (1,3) [(1,2),(1,3),(2,3),(3,3)]
    g4 = creaGrafo' ND (1,4) [(1,1),(1,2),(3,3)]

-- La verificación es
--    λ> verifica
```

```
--
--    def. 1
--      e1
--      e2
--      e3
--      e4
--      e5
--      e6
--    def. 2
--      e1
--      e2
--      e3
--      e4
--      e5
--      e6
--
--    Finished in 0.0013 seconds
--    12 examples, 0 failures
```

11.9.2. En Python

```
# ---------------------------------------------------------------------
# Usando el [tipo abstracto de datos de los grafos](https://bit.ly/45cQ3Fo),
# definir la función,
#    nAristas : (Grafo) -> int
# tal que nAristas(g) es el número de aristas del grafo g. Si g es no
# dirigido, las aristas de v1 a v2 y de v2 a v1 sólo se cuentan una
# vez. Por ejemplo,
#    g1 = creaGrafo_(Orientacion.ND, (1,5), [(1,2),(1,3),(1,5),(2,4),(2,5),(3,4)
#    g2 = creaGrafo_(Orientacion.D, (1,5), [(1,2),(1,3),(1,5),(2,4),(2,5),(4,3),
#    g3 = creaGrafo_(Orientacion.ND, (1,3), [(1,2),(1,3),(2,3),(3,3)])
#    g4 = creaGrafo_(Orientacion.ND, (1,4), [(1,1),(1,2),(3,3)])
#    >>> nAristas(g1)
#    8
#    >>> nAristas(g2)
#    7
#    >>> nAristas(g3)
#    4
#    >>> nAristas(g4)
#    3
```

```
#     >>> nAristas(completo(4))
#     6
#     >>> nAristas(completo(5))
#     10
#
#  Definir la función
#     prop_nAristasCompleto : (int) -> bool
#  tal que prop_nAristasCompleto(n) se verifica si el número de aristas
#  del grafo completo de orden n es n*(n-1)/2 y, usando la función,
#  comprobar que la propiedad se cumple para n de 1 a 20.
#  ---------------------------------------------------------------------

from src.Grafo_Grafos_completos import completo
from src.Grafo_Lazos_de_un_grafo import nLazos
from src.TAD.Grafo import Grafo, Orientacion, aristas, creaGrafo_, dirigido

# 1ª solución
# ===========

def nAristas(g: Grafo) -> int:
    if dirigido(g):
        return len(aristas(g))
    return (len(aristas(g)) + nLazos(g)) // 2

# 2ª solución
# ===========

def nAristas2(g: Grafo) -> int:
    if dirigido(g):
        return len(aristas(g))
    return len([(x, y) for ((x,y),_) in aristas(g) if x <= y])

# Propiedad
# =========

def prop_nAristasCompleto(n: int) -> bool:
    return nAristas(completo(n)) == n*(n-1) // 2

# La comprobación es
#    >>> all(prop_nAristasCompleto(n) for n in range(1, 21))
```

```
#    True

# Verificación
# ============

def test_nAristas() -> None:
    g1 = creaGrafo_(Orientacion.ND, (1,5),
                    [(1,2),(1,3),(1,5),(2,4),(2,5),(3,4),(3,5),(4,5)])
    g2 = creaGrafo_(Orientacion.D, (1,5),
                    [(1,2),(1,3),(1,5),(2,4),(2,5),(4,3),(4,5)])
    g3 = creaGrafo_(Orientacion.ND, (1,3), [(1,2),(1,3),(2,3),(3,3)])
    g4 = creaGrafo_(Orientacion.ND, (1,4), [(1,1),(1,2),(3,3)])
    for nAristas_ in [nAristas, nAristas2]:
        assert nAristas_(g1) == 8
        assert nAristas_(g2) == 7
        assert nAristas_(g3) == 4
        assert nAristas_(g4) == 3
        assert nAristas_(completo(4)) == 6
        assert nAristas_(completo(5)) == 10
    print("Verificado")

# La verificación es
#    >>> test_nAristas()
#    Verificado
```

11.10. Grados positivos y negativos

11.10.1. En Haskell

```
-- ---------------------------------------------------------------------
-- El grado positivo de un vértice v de un grafo g es el número de
-- vértices de g adyacentes con v y su grado negativo es el número de
-- vértices de g incidentes con v.
--
-- Usando el [tipo abstracto de datos de los grafos](https://bit.ly/45cQ3Fo),
-- definir las funciones,
--    gradoPos :: (Ix v,Num p) => Grafo v p -> v -> Int
--    gradoNeg :: (Ix v,Num p) => Grafo v p -> v -> Int
-- tales que
-- + (gradoPos g v) es el grado positivo del vértice v en el grafo g.
```

```
--   Por ejemplo,
--      λ> g1 = creaGrafo' ND (1,5) [(1,2),(1,3),(1,5),(2,4),(2,5),(3,4),(3,5),(4
--      λ> g2 = creaGrafo' D  (1,5) [(1,2),(1,3),(1,5),(2,4),(2,5),(4,3),(4,5)]
--      λ> gradoPos g1 5
--      4
--      λ> gradoPos g2 5
--      0
--      λ> gradoPos g2 1
--      3
-- + (gradoNeg g v) es el grado negativo del vértice v en el grafo g.
--   Por ejemplo,
--      λ> gradoNeg g1 5
--      4
--      λ> gradoNeg g2 5
--      3
--      λ> gradoNeg g2 1
--      0
-- ---------------------------------------------------------------------

module Grafo_Grados_positivos_y_negativos where

import TAD.Grafo (Grafo, Orientacion (D, ND), adyacentes, creaGrafo')
import Data.Ix (Ix)
import Grafo_Incidentes_de_un_vertice (incidentes)
import Test.Hspec (Spec, hspec, it, shouldBe)

-- 1ª definición de gradoPos
gradoPos :: (Ix v,Num p) => Grafo v p -> v -> Int
gradoPos g v = length (adyacentes g v)

-- 2ª definición de gradoPos
gradoPos2 :: (Ix v,Num p) => Grafo v p -> v -> Int
gradoPos2 g = length . adyacentes g

-- 1ª definición de gradoNeg
gradoNeg :: (Ix v,Num p) => Grafo v p -> v -> Int
gradoNeg g v = length (incidentes g v)

-- 2ª definición de gradoNeg
gradoNeg2 :: (Ix v,Num p) => Grafo v p -> v -> Int
```

```haskell
gradoNeg2 g = length . incidentes g

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

spec :: Spec
spec = do
  it "e1" $
    gradoPos g1 5 `shouldBe` 4
  it "e2" $
    gradoPos g2 5 `shouldBe` 0
  it "e3" $
    gradoPos g2 1 `shouldBe` 3
  it "e4" $
    gradoNeg g1 5 `shouldBe` 4
  it "e5" $
    gradoNeg g2 5 `shouldBe` 3
  it "e6" $
    gradoNeg g2 1 `shouldBe` 0
  it "e7" $
    gradoPos2 g1 5 `shouldBe` 4
  it "e8" $
    gradoPos2 g2 5 `shouldBe` 0
  it "e9" $
    gradoPos2 g2 1 `shouldBe` 3
  it "e10" $
    gradoNeg2 g1 5 `shouldBe` 4
  it "e11" $
    gradoNeg2 g2 5 `shouldBe` 3
  it "e12" $
    gradoNeg2 g2 1 `shouldBe` 0
  where
    g1, g2 :: Grafo Int Int
    g1 = creaGrafo' ND (1,5) [(1,2),(1,3),(1,5),(2,4),(2,5),(3,4),(3,5),(4,5)]
    g2 = creaGrafo' D  (1,5) [(1,2),(1,3),(1,5),(2,4),(2,5),(4,3),(4,5)]

-- La verificación es
```

```
--    λ> verifica
--
--    def. 1
--      e1
--      e2
--      e3
--      e4
--      e5
--      e6
--    def. 2
--      e1
--      e2
--      e3
--      e4
--      e5
--      e6
--
--    Finished in 0.0013 seconds
--    12 examples, 0 failures
```

11.10.2. En Python

```
# ---------------------------------------------------------------------
# El grado positivo de un vértice v de un grafo g es el número de
# vértices de g adyacentes con v y su grado negativo es el número de
# vértices de g incidentes con v.
#
# Usando el [tipo abstracto de datos de los grafos](https://bit.ly/45cQ3Fo),
# definir las funciones,
#    gradoPos : (Grafo, Vertice) -> int
#    gradoNeg : (Grafo, Vertice) -> int
# tales que
# + gradoPos(g, v) es el grado positivo del vértice v en el grafo g.
#   Por ejemplo,
#      g1 = creaGrafo_(Orientacion.ND, (1,5),
#                      [(1,2),(1,3),(1,5),(2,4),(2,5),(3,4),(3,5),(4,5)])
#      g2 = creaGrafo_(Orientacion.D, (1,5),
#                      [(1,2),(1,3),(1,5),(2,4),(2,5),(4,3),(4,5)])
#      λ> gradoPos(g1, 5)
#      4
```

```
#      λ> gradoPos(g2, 5)
#      0
#      λ> gradoPos(g2, 1)
#      3
# + gradoNeg(g, v) es el grado negativo del vértice v en el grafo g.
#   Por ejemplo,
#      λ> gradoNeg(g1, 5)
#      4
#      λ> gradoNeg(g2, 5)
#      3
#      λ> gradoNeg(g2, 1)
#      0
# ---------------------------------------------------------------------

from src.Grafo_Incidentes_de_un_vertice import incidentes
from src.TAD.Grafo import Grafo, Orientacion, Vertice, adyacentes, creaGrafo_


def gradoPos(g: Grafo, v: Vertice) -> int:
    return len(adyacentes(g, v))

def gradoNeg(g: Grafo, v: Vertice) -> int:
    return len(incidentes(g, v))

# Verificación
# ============

def test_GradoPosNeg() -> None:
    g1 = creaGrafo_(Orientacion.ND, (1,5),
                    [(1,2),(1,3),(1,5),(2,4),(2,5),(3,4),(3,5),(4,5)])
    g2 = creaGrafo_(Orientacion.D, (1,5),
                    [(1,2),(1,3),(1,5),(2,4),(2,5),(4,3),(4,5)])
    assert gradoPos(g1, 5) == 4
    assert gradoPos(g2, 5) == 0
    assert gradoPos(g2, 1) == 3
    assert gradoNeg(g1, 5) == 4
    assert gradoNeg(g2, 5) == 3
    assert gradoNeg(g2, 1) == 0
    print("Verificado")
```

```
# La verificación es
#    >>> test_GradoPosNeg()
#    Verificado
```

11.11. Generadores de grafos arbitrarios

11.11.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Definir un generador de grafos para comprobar propiedades de grafos
-- con QuickCheck y hacer el tipo de los Grafos un subtipo de
-- Arbitrary.
-- ---------------------------------------------------------------------

{-# LANGUAGE FlexibleInstances #-}
{-# OPTIONS_GHC -fno-warn-orphans #-}

module TAD.GrafoGenerador where

import TAD.Grafo (Grafo, Orientacion (D, ND), creaGrafo)
import Test.QuickCheck (Arbitrary, Gen, arbitrary, choose, vectorOf)

-- (generaGND n ps) es el grafo completo de orden n tal que los pesos
-- están determinados por ps. Por ejemplo,
--    λ> generaGND 3 [4,2,5]
--    G ND ([1,2,3],[((1,2),4),((1,3),2),((2,3),5)])
--    λ> generaGND 3 [4,-2,5]
--    G ND ([1,2,3],[((1,2),4),((2,3),5)])
generaGND :: Int -> [Int] -> Grafo Int Int
generaGND n ps  = creaGrafo ND (1,n) l3
  where l1 = [(x,y) | x <- [1..n], y <- [1..n], x < y]
        l2 = zip l1 ps
        l3 = [(x,y,z) | ((x,y),z) <- l2, z > 0]

-- (generaGD n ps) es el grafo completo de orden n tal que los pesos
-- están determinados por ps. Por ejemplo,
--    λ> generaGD 3 [4,2,5]
--    G D ([1,2,3],[((1,1),4),((1,2),2),((1,3),5)])
--    λ> generaGD 3 [4,2,5,3,7,9,8,6]
--    G D ([1,2,3],[((1,1),4),((1,2),2),((1,3),5),
```

```haskell
--                    ((2,1),3),((2,2),7),((2,3),9),
--                    ((3,1),8),((3,2),6)])
generaGD :: Int -> [Int] -> Grafo Int Int
generaGD n ps = creaGrafo D (1,n) l3
  where l1 = [(x,y) | x <- [1..n], y <- [1..n]]
        l2 = zip l1 ps
        l3 = [(x,y,z) | ((x,y),z) <- l2, z > 0]

-- genGD es un generador de grafos dirigidos. Por ejemplo,
--    λ> sample genGD
--    G D ([1],[])
--    G D ([1,2],[((1,1),5),((2,1),4)])
--    G D ([1,2],[((1,1),3),((1,2),3)])
--    G D ([1,2,3,4,5,6],[])
--    G D ([1,2],[((2,2),16)])
--    ...
genGD :: Gen (Grafo Int Int)
genGD = do
  n <- choose (1,10)
  xs <- vectorOf (n*n) arbitrary
  return (generaGD n xs)

-- genGND es un generador de grafos dirigidos. Por ejemplo,
--    λ> sample genGND
--    G ND ([1,2,3,4,5,6,7,8],[])
--    G ND ([1],[])
--    G ND ([1,2,3,4,5],[((1,2),2),((2,3),5),((3,4),5),((3,5),5)])
--    G ND ([1,2,3,4,5],[((1,2),6),((1,3),5),((1,5),1),((3,5),9),((4,5),6)])
--    G ND ([1,2,3,4],[((1,2),5),((3,4),2)])
--    G ND ([1,2,3],[])
--    G ND ([1,2,3,4],[((1,2),5),((1,4),14),((2,4),10)])
--    G ND ([1,2,3,4,5],[((1,5),8),((4,5),5)])
--    G ND ([1,2,3,4],[((1,2),1),((1,4),4),((2,3),4),((3,4),5)])
--    G ND ([1,2,3],[((1,2),8),((1,3),8),((2,3),3)])
--    ...
genGND :: Gen (Grafo Int Int)
genGND = do
  n <- choose (1,10)
  xs <- vectorOf (n*n) arbitrary
  return (generaGND n xs)
```

```
-- genG es un generador de grafos. Por ejemplo,
--    λ> sample genG
--    G ND ([1,2,3,4,5,6],[])
--    G D ([1],[((1,1),2)])
--    G D ([1,2],[((1,1),9)])
--    ...
genG :: Gen (Grafo Int Int)
genG = do
  d <- choose (True,False)
  n <- choose (1,10)
  xs <- vectorOf (n*n) arbitrary
  if d then return (generaGD n xs)
       else return (generaGND n xs)

-- Los grafos está contenido en la clase de los objetos generables
-- aleatoriamente.
instance Arbitrary (Grafo Int Int) where
  arbitrary = genG
```

11.11.2. En Python

```
# ---------------------------------------------------------------------
# Definir un generador de grafos para comprobar propiedades de grafos
# con Hypothesis.
# ---------------------------------------------------------------------

from typing import Any

from hypothesis import strategies as st
from hypothesis.strategies import composite

from src.TAD.Grafo import Grafo, Orientacion, creaGrafo_


# Generador de aristas. Por ejemplo,
#    >>> gen_aristas(5).example()
#    [(2, 5), (4, 5), (1, 2), (2, 3), (4, 1)]
#    >>> gen_aristas(5).example()
#    [(3, 4)]
```

```
#    >>> gen_aristas(5).example()
#    [(5, 3), (3, 2), (1, 3), (5, 2)]
@composite
def gen_aristas(draw: Any, n: int) -> list[tuple[int, int]]:
    as_ = draw(st.lists(st.tuples(st.integers(1,n),
                                  st.integers(1,n)),
                        unique=True))
    return as_

# Generador de grafos no dirigidos. Por ejemplo,
#    >>> gen_grafoND().example()
#    G ND ([1, 2, 3, 4, 5], [(1, 4), (5, 5)])
#    >>> gen_grafoND().example()
#    G ND ([1], [])
#    >>> gen_grafoND().example()
#    G ND ([1, 2, 3, 4, 5, 6, 7, 8], [(7, 7)])
#    >>> gen_grafoND().example()
#    G ND ([1, 2, 3, 4, 5, 6], [(1, 3), (2, 4), (3, 3), (3, 5)])
@composite
def gen_grafoND(draw: Any) -> Grafo:
    n = draw(st.integers(1,10))
    as_ = [(x, y) for (x, y ) in draw(gen_aristas(n)) if x <= y]
    return creaGrafo_(Orientacion.ND, (1,n), as_)

# Generador de grafos dirigidos. Por ejemplo,
#    >>> gen_grafoD().example()
#    G D ([1, 2, 3, 4], [(3, 3), (4, 1)])
#    >>> gen_grafoD().example()
#    G D ([1, 2], [(1, 1), (2, 1), (2, 2)])
#    >>> gen_grafoD().example()
#    G D ([1, 2], [])
@composite
def gen_grafoD(draw: Any) -> Grafo:
    n = draw(st.integers(1,10))
    as_ = draw(gen_aristas(n))
    return creaGrafo_(Orientacion.D, (1,n), as_)

# Generador de grafos. Por ejemplo,
#    >>> gen_grafo().example()
#    G ND ([1, 2, 3, 4, 5, 6, 7], [(1, 3)])
```

```
#    >>> gen_grafo().example()
#    G D ([1], [])
#    >>> gen_grafo().example()
#    G D ([1, 2, 3, 4, 5, 6, 7], [(1, 3), (3, 4), (5, 5)])
@composite
def gen_grafo(draw: Any) -> Grafo:
    o = draw(st.sampled_from([Orientacion.D, Orientacion.ND]))
    if o == Orientacion.ND:
        return draw(gen_grafoND())
    return draw(gen_grafoD())
```

11.12. Propiedades de grados positivos y negativos

11.12.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Comprobar con QuickCheck que para cualquier grafo g, las
-- sumas de los grados positivos y la de los grados negativos de los
-- vértices de g son iguales
-- ---------------------------------------------------------------------

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

module Grafo_Propiedades_de_grados_positivos_y_negativos where

import TAD.Grafo (Grafo, nodos)
import TAD.GrafoGenerador
import Grafo_Grados_positivos_y_negativos (gradoPos, gradoNeg)
import Test.QuickCheck

-- La propiedad es
prop_sumaGrados :: Grafo Int Int -> Bool
prop_sumaGrados g =
  sum [gradoPos g v | v <- vs] == sum [gradoNeg g v | v <- vs]
  where vs = nodos g

-- La comprobación es
--    λ> quickCheck prop_sumaGrados
```

```
--    +++ OK, passed 100 tests.
```

11.12.2. En Python

```python
# ---------------------------------------------------------------------
# Comprobar con Hypothesis que para cualquier grafo g, las sumas de los
# grados positivos y la de los grados negativos de los vértices de g son
# iguales
# ---------------------------------------------------------------------

from hypothesis import given

from src.Grafo_Grados_positivos_y_negativos import gradoNeg, gradoPos
from src.TAD.Grafo import Grafo, nodos
from src.TAD.GrafoGenerador import gen_grafo


# La propiedad es
@given(gen_grafo())
def test_sumaGrados(g: Grafo) -> None:
    vs = nodos(g)
    assert sum((gradoPos(g, v) for v in vs)) == sum((gradoNeg(g, v) for v in vs))

# La comprobación es
#    src> poetry run pytest -q Grafo_Propiedades_de_grados_positivos_y_negativos.
#    1 passed in 0.31s
```

11.13. Grado de un vértice

11.13.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- El grado de un vértice v de un grafo dirigido g, es el número de
-- aristas de g que contiene a v. Si g es no dirigido, el grado de un
-- vértice v es el número de aristas incidentes en v, teniendo en cuenta
-- que los lazos se cuentan dos veces.
--
-- Usando el [tipo abstracto de datos de los grafos](https://bit.ly/45cQ3Fo),
-- definir las funciones,
```

```
--    grado :: (Ix v,Num p) => Grafo v p -> v -> Int
-- tal que (grado g v) es el grado del vértice v en el grafo g. Por
-- ejemplo,
--    g1 = creaGrafo' ND (1,5) [(1,2),(1,3),(1,5),(2,4),(2,5),(3,4),(3,5),(4,5)]
--    g2 = creaGrafo' D  (1,5) [(1,2),(1,3),(1,5),(2,4),(2,5),(4,3),(4,5)]
--    g3 = creaGrafo' D  (1,3) [(1,2),(2,2),(3,1),(3,2)]
--    g4 = creaGrafo' D  (1,1) [(1,1)]
--    g5 = creaGrafo' ND (1,3) [(1,2),(1,3),(2,3),(3,3)]
--    g6 = creaGrafo' D  (1,3) [(1,2),(1,3),(2,3),(3,3)]
--    grado g1 5 ==  4
--    grado g2 5 ==  3
--    grado g2 1 ==  3
--    grado g3 2 ==  4
--    grado g3 1 ==  2
--    grado g3 3 ==  2
--    grado g4 1 ==  2
--    grado g6 3 ==  4
--    grado g6 3 ==  4
--
-- Comprobar con QuickCheck que en todo grafo, el número de nodos de
-- grado impar es par.
-- ---------------------------------------------------------------------

module Grafo_Grado_de_un_vertice where

import TAD.Grafo (Grafo, Orientacion (D, ND), dirigido, nodos, creaGrafo')
import Data.Ix (Ix)
import Grafo_Lazos_de_un_grafo (lazos)
import Grafo_Incidentes_de_un_vertice (incidentes)
import Grafo_Grados_positivos_y_negativos (gradoPos, gradoNeg)
import Test.Hspec (Spec, hspec, it, shouldBe)

grado :: (Ix v,Num p) => Grafo v p -> v -> Int
grado g v | dirigido g           = gradoNeg g v + gradoPos g v
          | (v,v) `elem` lazos g = length (incidentes g v) + 1
          | otherwise            = length (incidentes g v)

-- La propiedad es
prop_numNodosGradoImpar :: Grafo Int Int -> Bool
prop_numNodosGradoImpar g =
```

```haskell
  even (length [v | v <- nodos g, odd (grado g v)])

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

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

spec :: Spec
spec = do
  it "e1" $
    grado g1 5 `shouldBe`  4
  it "e2" $
    grado g2 5 `shouldBe`  3
  it "e3" $
    grado g2 1 `shouldBe`  3
  it "e4" $
    grado g3 2 `shouldBe`  4
  it "e5" $
    grado g3 1 `shouldBe`  2
  it "e6" $
    grado g3 3 `shouldBe`  2
  it "e7" $
    grado g4 1 `shouldBe`  2
  it "e8" $
    grado g5 3 `shouldBe`  4
  it "e9" $
    grado g6 3 `shouldBe`  4
  where
    g1, g2, g3, g4, g5, g6 :: Grafo Int Int
    g1 = creaGrafo' ND (1,5) [(1,2),(1,3),(1,5),(2,4),(2,5),(3,4),(3,5),(4,5)]
    g2 = creaGrafo' D  (1,5) [(1,2),(1,3),(1,5),(2,4),(2,5),(4,3),(4,5)]
    g3 = creaGrafo' D  (1,3) [(1,2),(2,2),(3,1),(3,2)]
    g4 = creaGrafo' D  (1,1) [(1,1)]
    g5 = creaGrafo' ND (1,3) [(1,2),(1,3),(2,3),(3,3)]
    g6 = creaGrafo' D  (1,3) [(1,2),(1,3),(2,3),(3,3)]
```

```
-- La verificación es
--    λ> verifica
--
--    e1
--    e2
--    e3
--    e4
--    e5
--    e6
--    e7
--    e8
--    e9
--
--    Finished in 0.0015 seconds
--    9 examples, 0 failures
```

11.13.2. En Python

```
# ---------------------------------------------------------------------
# El grado de un vértice v de un grafo dirigido g, es el número de
# aristas de g que contiene a v. Si g es no dirigido, el grado de un
# vértice v es el número de aristas incidentes en v, teniendo en cuenta
# que los lazos se cuentan dos veces.
#
# Usando el [tipo abstracto de datos de los grafos](https://bit.ly/45cQ3Fo),
# definir las funciones,
#    grado : (Grafo, Vertice) -> int
# tal que grado(g, v) es el grado del vértice v en el grafo g. Por
# ejemplo,
#    >>> g1 = creaGrafo_(Orientacion.ND, (1,5),
#                        [(1,2),(1,3),(1,5),(2,4),(2,5),(3,4),(3,5),(4,5)])
#    >>> g2 = creaGrafo_(Orientacion.D, (1,5),
#                        [(1,2),(1,3),(1,5),(2,4),(2,5),(4,3),(4,5)])
#    >>> g3 = creaGrafo_(Orientacion.D, (1,3),
#                        [(1,2),(2,2),(3,1),(3,2)])
#    >>> g4 = creaGrafo_(Orientacion.D, (1,1),
#                        [(1,1)])
#    >>> g5 = creaGrafo_(Orientacion.ND, (1,3),
#                        [(1,2),(1,3),(2,3),(3,3)])
```

```
#    >>> g6 = creaGrafo_(Orientacion.D, (1,3),
#                         [(1,2),(1,3),(2,3),(3,3)])
#    >>> grado(g1, 5)
#    4
#    >>> grado(g2, 5)
#    3
#    >>> grado(g2, 1)
#    3
#    >>> grado(g3, 2)
#    4
#    >>> grado(g3, 1)
#    2
#    >>> grado(g3, 3)
#    2
#    >>> grado(g4, 1)
#    2
#    >>> grado(g5, 3)
#    4
#    >>> grado(g6, 3)
#    4
#
# Comprobar con Hypothesis que en todo grafo, el número de nodos de
# grado impar es par.
# ---------------------------------------------------------------------

from hypothesis import given

from src.Grafo_Grados_positivos_y_negativos import gradoNeg, gradoPos
from src.Grafo_Incidentes_de_un_vertice import incidentes
from src.Grafo_Lazos_de_un_grafo import lazos
from src.TAD.Grafo import (Grafo, Orientacion, Vertice, creaGrafo_, dirigido,
                           nodos)
from src.TAD.GrafoGenerador import gen_grafo


def grado(g: Grafo, v: Vertice) -> int:
    if dirigido(g):
        return gradoNeg(g, v) + gradoPos(g, v)
    if (v, v) in lazos(g):
        return len(incidentes(g, v)) + 1
```

```
    return len(incidentes(g, v))

# La propiedad es
@given(gen_grafo())
def test_grado1(g: Grafo) -> None:
    assert len([v for v in nodos(g) if grado(g, v) % 2 == 1]) % 2 == 0

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

# Verificación
# ============

def test_grado() -> None:
    g1 = creaGrafo_(Orientacion.ND, (1,5),
                    [(1,2),(1,3),(1,5),(2,4),(2,5),(3,4),(3,5),(4,5)])
    g2 = creaGrafo_(Orientacion.D, (1,5),
                    [(1,2),(1,3),(1,5),(2,4),(2,5),(4,3),(4,5)])
    g3 = creaGrafo_(Orientacion.D, (1,3),
                    [(1,2),(2,2),(3,1),(3,2)])
    g4 = creaGrafo_(Orientacion.D, (1,1),
                    [(1,1)])
    g5 = creaGrafo_(Orientacion.ND, (1,3),
                    [(1,2),(1,3),(2,3),(3,3)])
    g6 = creaGrafo_(Orientacion.D, (1,3),
                    [(1,2),(1,3),(2,3),(3,3)])
    assert grado(g1, 5) == 4
    assert grado(g2, 5) == 3
    assert grado(g2, 1) == 3
    assert grado(g3, 2) == 4
    assert grado(g3, 1) == 2
    assert grado(g3, 3) == 2
    assert grado(g4, 1) == 2
    assert grado(g5, 3) == 4
    assert grado(g6, 3) == 4
    print("Verificado")

# La verificación es
#    >>> test_grado()
```

```
#    Verificado
```

11.14. Lema del apretón de manos

11.14.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- En la teoría de grafos, se conoce como "Lema del apretón de manos" la
-- siguiente propiedad: la suma de los grados de los vértices de g es el
-- doble del número de aristas de g.
--
-- Comprobar con QuickCheck que para cualquier grafo g, se verifica
-- dicha propiedad.
-- ---------------------------------------------------------------------

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

module Grafo_Lema_del_apreton_de_manos where

import TAD.Grafo (Grafo, nodos)
import TAD.GrafoGenerador
import Grafo_Grado_de_un_vertice (grado)
import Grafo_Numero_de_aristas_de_un_grafo (nAristas)
import Test.QuickCheck

prop_apretonManos :: Grafo Int Int -> Bool
prop_apretonManos g =
  sum [grado g v | v <- nodos g] == 2 * nAristas g

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

11.14.2. En Python

```python
# ---------------------------------------------------------------------
# En la teoría de grafos, se conoce como "Lema del apretón de manos" la
# siguiente propiedad: la suma de los grados de los vértices de g es el
# doble del número de aristas de g.
```

```
#
# Comprobar con Hypothesis que para cualquier grafo g, se verifica
# dicha propiedad.
# ---------------------------------------------------------------------

from hypothesis import given

from src.Grafo_Grado_de_un_vertice import grado
from src.Grafo_Numero_de_aristas_de_un_grafo import nAristas
from src.TAD.Grafo import Grafo, nodos
from src.TAD.GrafoGenerador import gen_grafo


# La propiedad es
@given(gen_grafo())
def test_apreton(g: Grafo) -> None:
    assert sum((grado(g, v) for v in nodos(g))) == 2 * nAristas(g)

# La comprobación es
#    src> poetry run pytest -q Grafo_Lema_del_apreton_de_manos.py
#    1 passed in 0.32s
```

11.15. Grafos regulares

11.15.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Un grafo es regular si todos sus vértices tienen el mismo
-- grado.
--
-- Usando el [tipo abstracto de datos de los grafos](https://bit.ly/45cQ3Fo),
-- definir la función,
--    regular :: (Ix v,Num p) => Grafo v p -> Bool
-- tal que (regular g) se verifica si el grafo g es regular. Por ejemplo,
--    λ> regular (creaGrafo' D (1,3) [(1,2),(2,3),(3,1)])
--    True
--    λ> regular (creaGrafo' ND (1,3) [(1,2),(2,3)])
--    False
--    λ> regular (completo 4)
--    True
```

```haskell
--
-- Comprobar que los grafos completos son regulares.
-- ---------------------------------------------------------------------

module Grafo_Grafos_regulares where

import TAD.Grafo (Grafo, Orientacion (D, ND), nodos, creaGrafo')
import Data.Ix (Ix)
import Grafo_Grado_de_un_vertice (grado)
import Grafo_Grafos_completos (completo)
import Test.Hspec (Spec, hspec, it, shouldBe)

regular :: (Ix v,Num p) => Grafo v p -> Bool
regular g = and [grado g v == k | v <- vs]
  where vs = nodos g
        k  = grado g (head vs)

-- La propiedad de la regularidad de todos los grafos completos de orden
-- entre m y n es
prop_CompletoRegular :: Int -> Int -> Bool
prop_CompletoRegular m n =
  and [regular (completo x) | x <- [m..n]]

-- La comprobación es
--    λ> prop_CompletoRegular 1 30
--    True

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

spec :: Spec
spec = do
  it "e1" $
    regular g1 `shouldBe` True
  it "e2" $
    regular g2 `shouldBe` False
  it "e3" $
```

```
    regular (completo 4) `shouldBe` True
  where
    g1, g2 :: Grafo Int Int
    g1 = creaGrafo' D (1,3) [(1,2),(2,3),(3,1)]
    g2 = creaGrafo' ND (1,3) [(1,2),(2,3)]

-- La verificación es
--    λ> verifica
--
--    e1
--    e2
--    e3
--
--    Finished in 0.0006 seconds
--    3 examples, 0 failures
```

11.15.2. En Python

```
# ---------------------------------------------------------------------
# Un grafo es regular si todos sus vértices tienen el mismo
# grado.
#
# Usando el [tipo abstracto de datos de los grafos](https://bit.ly/45cQ3Fo),
# definir la función,
#    regular : (Grafo) -> bool
# tal que regular(g) se verifica si el grafo g es regular. Por ejemplo,
#    >>> regular(creaGrafo_(Orientacion.D, (1,3), [(1,2),(2,3),(3,1)]))
#    True
#    >>> regular(creaGrafo_(Orientacion.ND, (1,3), [(1,2),(2,3)]))
#    False
#    >>> regular(completo(4))
#    True
#
# Comprobar que los grafos completos son regulares.
# ---------------------------------------------------------------------

from src.Grafo_Grado_de_un_vertice import grado
from src.Grafo_Grafos_completos import completo
from src.TAD.Grafo import Grafo, Orientacion, creaGrafo_, nodos
```

```
def regular(g: Grafo) -> bool:
    vs = nodos(g)
    k = grado(g, vs[0])
    return all(grado(g, v) == k for v in vs)

# La propiedad de la regularidad de todos los grafos completos de orden
# entre m y n es
def prop_CompletoRegular(m: int, n: int) -> bool:
    return all(regular(completo(x)) for x in range(m, n + 1))

# La comprobación es
#    >>> prop_CompletoRegular(1, 30)
#    True

# Verificación
# ============

def test_regular() -> None:
    g1 = creaGrafo_(Orientacion.D, (1,3), [(1,2),(2,3),(3,1)])
    g2 = creaGrafo_(Orientacion.ND, (1,3), [(1,2),(2,3)])
    assert regular(g1)
    assert not regular(g2)
    assert regular(completo(4))
    print("Verificado")

# La verificación es
#    >>> test_regular()
#    Verificado
```

11.16. Grafos k–regulares

11.16.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Un grafo es k-regular si todos sus vértices son de grado k.
-- Usando el [tipo abstracto de datos de los grafos](https://bit.ly/45cQ3Fo),
-- definir la función,
--    regularidad :: (Ix v,Num p) => Grafo v p -> Maybe Int
-- tal que (regularidad g) es la regularidad de g. Por ejemplo,
```

```haskell
--    regularidad (creaGrafo' ND (1,2) [(1,2),(2,3)]) == Just 1
--    regularidad (creaGrafo' D (1,2) [(1,2),(2,3)])  == Nothing
--    regularidad (completo 4)                        == Just 3
--    regularidad (completo 5)                        == Just 4
--    regularidad (grafoCiclo 4)                      == Just 2
--    regularidad (grafoCiclo 5)                      == Just 2
--
-- Comprobar que el grafo completo de orden n es (n-1)-regular (para
-- n de 1 a 20) y el grafo ciclo de orden n es 2-regular (para n de 3 a
-- 20).
-- ---------------------------------------------------------------------

module Grafo_Grafos_k_regulares where

import TAD.Grafo (Grafo, Orientacion (D, ND), nodos, creaGrafo')
import Data.Ix (Ix)
import Grafo_Grado_de_un_vertice (grado)
import Grafo_Grafos_regulares (regular)
import Grafo_Grafos_completos (completo)
import Grafo_Grafos_ciclos (grafoCiclo)
import Test.Hspec (Spec, hspec, it, shouldBe)

regularidad :: (Ix v,Num p) => Grafo v p -> Maybe Int
regularidad g
  | regular g = Just (grado g (head (nodos g)))
  | otherwise = Nothing

-- La propiedad de k-regularidad de los grafos completos es
prop_completoRegular :: Int -> Bool
prop_completoRegular n =
  regularidad (completo n) == Just (n-1)

-- La comprobación es
--    λ> and [prop_completoRegular n | n <- [1..20]]
--    True

-- La propiedad de k-regularidad de los grafos ciclos es
prop_cicloRegular :: Int -> Bool
prop_cicloRegular n =
  regularidad (grafoCiclo n) == Just 2
```

```
-- La comprobación es
--    λ> and [prop_cicloRegular n | n <- [3..20]]
--    True

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

spec :: Spec
spec = do
  it "e1" $
    regularidad g1              `shouldBe` Just 1
  it "e2" $
    regularidad g2              `shouldBe` Nothing
  it "e3" $
    regularidad (completo 4)    `shouldBe` Just 3
  it "e4" $
    regularidad (completo 5)    `shouldBe` Just 4
  it "e5" $
    regularidad (grafoCiclo 4) `shouldBe` Just 2
  it "e6" $
    regularidad (grafoCiclo 5) `shouldBe` Just 2
  where
    g1, g2 :: Grafo Int Int
    g1 = creaGrafo' ND (1,2) [(1,2),(2,3)]
    g2 = creaGrafo' D (1,2) [(1,2),(2,3)]

-- La verificación es
--    λ> verifica
--
--    e1
--    e2
--    e3
--    e4
--    e5
--    e6
--
```

```
--    Finished in 0.0027 seconds
--    6 examples, 0 failures
```

11.16.2. En Python

```
# ---------------------------------------------------------------------
# Un grafo es k-regular si todos sus vértices son de grado k.
# Usando el [tipo abstracto de datos de los grafos](https://bit.ly/45cQ3Fo),
# definir la función,
#    regularidad : (Grafo) -> Optional[int]
# tal que regularidad(g) es la regularidad de g. Por ejemplo,
#    regularidad(creaGrafo_(Orientacion.ND, (1,2), [(1,2),(2,3)]) == 1
#    regularidad(creaGrafo_(Orientacion.D, (1,2), [(1,2),(2,3)])  == None
#    regularidad(completo(4))                                     == 3
#    regularidad(completo(5))                                     == 4
#    regularidad(grafoCiclo(4))                                   == 2
#    regularidad(grafoCiclo(5))                                   == 2
#
# Comprobar que el grafo completo de orden n es (n-1)-regular (para
# n de 1 a 20) y el grafo ciclo de orden n es 2-regular (para n de
# 3 a 20).
# ---------------------------------------------------------------------

from typing import Optional

from src.Grafo_Grado_de_un_vertice import grado
from src.Grafo_Grafos_ciclos import grafoCiclo
from src.Grafo_Grafos_completos import completo
from src.Grafo_Grafos_regulares import regular
from src.TAD.Grafo import Grafo, Orientacion, creaGrafo_, nodos


def regularidad(g: Grafo) -> Optional[int]:
    if regular(g):
        return grado(g, nodos(g)[0])
    return None

# La propiedad de k-regularidad de los grafos completos es
def prop_completoRegular(n: int) -> bool:
    return regularidad(completo(n)) == n - 1
```

```
# La comprobación es
#    >>> all(prop_completoRegular(n) for n in range(1, 21))
#    True

# La propiedad de k-regularidad de los grafos ciclos es
def prop_cicloRegular(n: int) -> bool:
    return regularidad(grafoCiclo(n)) == 2

# La comprobación es
#    >>> all(prop_cicloRegular(n) for n in range(3, 21))
#    True

# Verificación
# ============

def test_k_regularidad() -> None:
    g1 = creaGrafo_(Orientacion.ND, (1,2), [(1,2),(2,3)])
    g2 = creaGrafo_(Orientacion.D, (1,2), [(1,2),(2,3)])
    assert regularidad(g1) == 1
    assert regularidad(g2) is None
    assert regularidad(completo(4)) == 3
    assert regularidad(completo(5)) == 4
    assert regularidad(grafoCiclo(4)) == 2
    assert regularidad(grafoCiclo(5)) == 2
    print("Verificado")

# La verificación es
#    >>> test_k_regularidad()
#    Verificado
```

11.17. Recorridos en un grafo completo

11.17.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Definir la función
--    recorridos :: [a] -> [[a]]
-- tal que (recorridos xs) es la lista de todos los posibles recorridos
-- por el grafo cuyo conjunto de vértices es xs y cada vértice se
```

```
-- encuentra conectado con todos los otros y los recorridos pasan por
-- todos los vértices una vez y terminan en el vértice inicial. Por
-- ejemplo,
--    λ> recorridos [2,5,3]
--    [[2,5,3,2],[5,2,3,5],[3,5,2,3],[5,3,2,5],[3,2,5,3],[2,3,5,2]]
-- Indicación: No importa el orden de los recorridos en la lista.
-- ---------------------------------------------------------------------

module Grafo_Recorridos_en_un_grafo_completo where

import Data.List (permutations)
import Test.Hspec (Spec, hspec, it, shouldBe)

recorridos :: [a] -> [[a]]
recorridos xs = [(y:ys) ++ [y] | y:ys <- permutations xs]

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

spec :: Spec
spec = do
  it "e1" $
    recorridos [2 :: Int,5,3] `shouldBe`
    [[2,5,3,2],[5,2,3,5],[3,5,2,3],[5,3,2,5],[3,2,5,3],[2,3,5,2]]

-- La verificación es
--    λ> verifica
--
--    e1
--
--    Finished in 0.0007 seconds
--    1 example, 0 failures
```

11.17.2. En Python

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

```
#    recorridos : (list[A]) -> list[list[A]]
# tal que recorridos(xs) es la lista de todos los posibles recorridos
# por el grafo cuyo conjunto de vértices es xs y cada vértice se
# encuentra conectado con todos los otros y los recorridos pasan por
# todos los vértices una vez y terminan en el vértice inicial. Por
# ejemplo,
#    >>> recorridos([2, 5, 3])
#    [[2, 5, 3, 2], [2, 3, 5, 2], [5, 2, 3, 5], [5, 3, 2, 5],
#     [3, 2, 5, 3], [3, 5, 2, 3]]
# ---------------------------------------------------------------------

from itertools import permutations
from typing import TypeVar

A = TypeVar('A')

def recorridos(xs: list[A]) -> list[list[A]]:
    return [(list(y) + [y[0]]) for y in permutations(xs)]

# Verificación
# ============

def test_recorridos() -> None:
    assert recorridos([2, 5, 3]) \
        == [[2, 5, 3, 2], [2, 3, 5, 2], [5, 2, 3, 5], [5, 3, 2, 5],
            [3, 2, 5, 3], [3, 5, 2, 3]]
    print("Verificado")

# La verificación es
#    >>> test_recorridos()
#    Verificado
```

11.18. Anchura de un grafo

11.18.1. En Haskell

```
-- ---------------------------------------------------------------------
-- En un grafo, la anchura de un nodo es el máximo de los valores
-- absolutos de la diferencia entre el valor del nodo y los de sus
-- adyacentes; y la anchura del grafo es la máxima anchura de sus
```

```
-- nodos. Por ejemplo, en el grafo
--    grafo1 :: Grafo Int Int
--    grafo1 = creaGrafo' D (1,5) [(1,2),(1,3),(1,5),
--                                (2,4),(2,5),
--                                (3,4),(3,5),
--                                (4,5)]
-- su anchura es 4 y el nodo de máxima anchura es el 5.
--
-- Usando el [tipo abstracto de datos de los grafos](https://bit.ly/45cQ3Fo),
-- definir la función,
--    anchura :: Grafo Int Int -> Int
-- tal que (anchuraG g) es la anchura del grafo g. Por ejemplo,
--    anchura grafo1  ==  4
--
-- Comprobar experimentalmente que la anchura del grafo ciclo de orden
-- n es n-1.
-- ---------------------------------------------------------------------

module Grafo_Anchura_de_un_grafo where

import TAD.Grafo (Grafo, Orientacion (D, ND), adyacentes, aristas,
                  creaGrafo', nodos)
import Grafo_Grafos_ciclos (grafoCiclo)
import Test.Hspec (Spec, hspec, it, shouldBe)

grafo1 :: Grafo Int Int
grafo1 = creaGrafo' D (1,5) [(1,2),(1,3),(1,5),
                             (2,4),(2,5),
                             (3,4),(3,5),
                             (4,5)]

-- 1ª solución
-- ===========

anchura :: Grafo Int Int -> Int
anchura g = maximum [anchuraN g x | x <- nodos g]

-- (anchuraN g x) es la anchura del nodo x en el grafo g. Por ejemplo,
--    anchuraN g 1  ==  4
--    anchuraN g 2  ==  3
```

```haskell
--    anchuraN g 4  ==  2
--    anchuraN g 5  ==  4
anchuraN :: Grafo Int Int -> Int -> Int
anchuraN g x = maximum (0 : [abs (x-v) | v <- adyacentes g x])

-- 2ª solución
-- ===========

anchura2 :: Grafo Int Int -> Int
anchura2 g = maximum [abs (x-y) | ((x,y),_) <- aristas g]

-- La conjetura
conjetura :: Int -> Bool
conjetura n = anchura (grafoCiclo n) == n-1

-- La comprobación es
--    λ> and [conjetura n | n <- [2..10]]
--    True

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

spec :: Spec
spec = do
  it "e1" $
    anchura grafo1 `shouldBe` 4
  it "e2" $
    anchura g2 `shouldBe` 2
  where
    g2 :: Grafo Int Int
    g2 = creaGrafo' ND (1,3) [(1,2),(1,3),(2,3),(3,3)]

-- La verificación es
--    λ> verifica
--
--    e1
--    e2
```

```
--
--    Finished in 0.0004 seconds
--    2 examples, 0 failures
```

11.18.2. En Python

```
# ---------------------------------------------------------------------
# En un grafo, la anchura de un nodo es el máximo de los valores
# absolutos de la diferencia entre el valor del nodo y los de sus
# adyacentes; y la anchura del grafo es la máxima anchura de sus
# nodos. Por ejemplo, en el grafo
#    grafo1: Grafo = creaGrafo_(Orientacion.D, (1,5), [(1,2),(1,3),(1,5),
#                                                      (2,4),(2,5),
#                                                      (3,4),(3,5),
#                                                      (4,5)])
# su anchura es 4 y el nodo de máxima anchura es el 5.
#
# Usando el [tipo abstracto de datos de los grafos](https://bit.ly/45cQ3Fo),
# definir la función,
#    anchura : (Grafo) -> int
# tal que anchuraG(g) es la anchura del grafo g. Por ejemplo,
#    anchura(grafo1)  ==  4
#
# Comprobar experimentalmente que la anchura del grafo ciclo de orden
# n es n-1.
# ---------------------------------------------------------------------

from src.Grafo_Grafos_ciclos import grafoCiclo
from src.TAD.Grafo import (Grafo, Orientacion, Vertice, adyacentes, aristas,
                           creaGrafo_, nodos)

grafo1: Grafo = creaGrafo_(Orientacion.D, (1,5), [(1,2),(1,3),(1,5),
                                                  (2,4),(2,5),
                                                  (3,4),(3,5),
                                                  (4,5)])

# 1ª solución
# ===========

def anchura(g: Grafo) -> int:
```

```
    return max(anchuraN(g, x) for x in nodos(g))

# (anchuraN g x) es la anchura del nodo x en el grafo g. Por ejemplo,
#    anchuraN g 1  ==  4
#    anchuraN g 2  ==  3
#    anchuraN g 4  ==  2
#    anchuraN g 5  ==  4
def anchuraN(g: Grafo, x: Vertice) -> int:
    return max([0] + [abs (x - v) for v in adyacentes(g, x)])

# 2ª solución
# ===========

def anchura2(g: Grafo) -> int:
    return max(abs (x-y) for ((x,y),_) in aristas(g))

# La conjetura
def conjetura(n: int) -> bool:
    return anchura(grafoCiclo(n)) == n - 1

# La comprobación es
#    >>> all(conjetura(n) for n in range(2, 11))
#    True

# Verificación
# ============

def test_anchura() -> None:
    g2 = creaGrafo_(Orientacion.ND, (1,3), [(1,2),(1,3),(2,3),(3,3)])
    assert anchura(grafo1) == 4
    assert anchura(g2) == 2
    print("Verificado")

# La verificación es
#    >>> test_anchura()
#    Verificado
```

11.19. Recorrido en profundidad

11.19.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Usando el [tipo abstracto de datos de los grafos](https://bit.ly/45cQ3Fo),
-- definir la función,
--    recorridoEnProfundidad :: (Num p, Eq p, Ix v) => v -> Grafo v p -> [v]
-- tal que (recorridoEnProfundidad i g) es el recorrido en profundidad
-- del grafo g desde el vértice i. Por ejemplo, en el grafo
--
--    +---> 2 <---+
--    |           |
--    |           |
--    1 --> 3 --> 6 --> 5
--    |                 |
--    |                 |
--    +---> 4 <---------+
--
-- definido por
--    grafo1 :: Grafo Int Int
--    grafo1 = creaGrafo' D (1,6) [(1,2),(1,3),(1,4),(3,6),(5,4),(6,2),(6,5)]
-- entonces
--    recorridoEnProfundidad 1 grafo1  ==  [1,2,3,6,5,4]
-- ---------------------------------------------------------------

module Grafo_Recorrido_en_profundidad where
import TAD.Grafo (Grafo, Orientacion (D, ND), adyacentes,
                  creaGrafo')
import Data.Ix (Ix)
import Test.Hspec (Spec, hspec, it, shouldBe)

grafo1 :: Grafo Int Int
grafo1 = creaGrafo' D (1,6) [(1,2),(1,3),(1,4),(3,6),(5,4),(6,2),(6,5)]

-- 1ª solución
-- ===========

recorridoEnProfundidad1 :: (Num p, Eq p, Ix v) => v -> Grafo v p -> [v]
recorridoEnProfundidad1 i g = rp [i] []
  where
```

```
    rp [] vis    = vis
    rp (c:cs) vis
        | c `elem` vis = rp cs vis
        | otherwise    = rp (adyacentes g c ++ cs) (vis ++ [c])

-- Traza del cálculo de (recorridoEnProfundidad1 1 grafo1)
--    recorridoEnProfundidad1 1 grafo1
--    = rp [1]     []
--    = rp [2,3,4] [1]
--    = rp [3,4]   [1,2]
--    = rp [6,4]   [1,2,3]
--    = rp [2,5,4] [1,2,3,6]
--    = rp [5,4]   [1,2,3,6]
--    = rp [4,4]   [1,2,3,6,5]
--    = rp [4]     [1,2,3,6,5,4]
--    = rp []      [1,2,3,6,5,4]
--    = [1,2,3,6,5,4]

-- 2ª solución
-- ===========

recorridoEnProfundidad :: (Num p, Eq p, Ix v) => v -> Grafo v p -> [v]
recorridoEnProfundidad i g = reverse (rp [i] [])
  where
    rp [] vis     = vis
    rp (c:cs) vis
        | c `elem` vis = rp cs vis
        | otherwise    = rp (adyacentes g c ++ cs) (c:vis)

-- Traza del cálculo de (recorridoEnProfundidad 1 grafo1)
--    RecorridoEnProfundidad 1 grafo1
--    = reverse (rp [1]     [])
--    = reverse (rp [2,3,4] [1])
--    = reverse (rp [3,4]   [2,1])
--    = reverse (rp [6,4]   [3,2,1])
--    = reverse (rp [2,5,4] [6,3,2,1])
--    = reverse (rp [5,4]   [6,3,2,1])
--    = reverse (rp [4,4]   [5,6,3,2,1])
--    = reverse (rp [4]     [4,5,6,3,2,1])
--    = reverse (rp []      [4,5,6,3,2,1])
```

```
--     = reverse [4,5,6,3,2,1]
--     = [1,2,3,6,5,4]

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

spec :: Spec
spec = do
  it "e1" $
    recorridoEnProfundidad1 1 grafo1 `shouldBe` [1,2,3,6,5,4]
  it "e2" $
    recorridoEnProfundidad 1 grafo1 `shouldBe` [1,2,3,6,5,4]
  it "e3" $
    recorridoEnProfundidad1 1 grafo2 `shouldBe` [1,2,6,3,5,4]
  it "e4" $
    recorridoEnProfundidad 1 grafo2 `shouldBe` [1,2,6,3,5,4]
  where
    grafo2 :: Grafo Int Int
    grafo2 = creaGrafo' ND (1,6) [(1,2),(1,3),(1,4),(3,6),(5,4),(6,2),(6,5)]

-- La verificación es
--    λ> verifica
--
--    e1
--    e2
--    e3
--    e4
--
--    Finished in 0.0022 seconds
--    4 examples, 0 failures
```

11.19.2. En Python

```
# ---------------------------------------------------------------------
# Usando el [tipo abstracto de datos de los grafos](https://bit.ly/45cQ3Fo),
# definir la función,
#    recorridoEnProfundidad : (Vertice, Grafo) -> list[Vertice]
```

```
# tal que recorridoEnProfundidad(i, g) es el recorrido en profundidad
# del grafo g desde el vértice i. Por ejemplo, en el grafo
#
#    +---> 2 <---+
#    |           |
#    |           |
#    1 --> 3 --> 6 --> 5
#    |                 |
#    |                 |
#    +---> 4 <---------+
#
# definido por
#    grafo1: Grafo = creaGrafo_(Orientacion.D,
#                               (1,6),
#                               [(1,2),(1,3),(1,4),(3,6),(5,4),(6,2),(6,5)])
# entonces
#    recorridoEnProfundidad(1, grafo1)  ==  [1,2,3,6,5,4]
# -----------------------------------------------------------

from src.TAD.Grafo import Grafo, Orientacion, Vertice, adyacentes, creaGrafo_

grafo1: Grafo = creaGrafo_(Orientacion.D,
                           (1,6),
                           [(1,2),(1,3),(1,4),(3,6),(5,4),(6,2),(6,5)])

# 1ª solución
# ===========

def recorridoEnProfundidad1(i: Vertice, g: Grafo) -> list[Vertice]:
    def rp(cs: list[Vertice], vis: list[Vertice]) -> list[Vertice]:
        if not cs:
            return vis
        d, *ds = cs
        if d in vis:
            return rp(ds, vis)
        return rp(adyacentes(g, d) + ds, vis + [d])
    return rp([i], [])

# Traza del cálculo de recorridoEnProfundidad1(1, grafo1)
#    recorridoEnProfundidad1(1, grafo1)
```

```
#    = rp([1],     [])
#    = rp([2,3,4], [1])
#    = rp([3,4],   [1,2])
#    = rp([6,4],   [1,2,3])
#    = rp([2,5,4], [1,2,3,6])
#    = rp([5,4],   [1,2,3,6])
#    = rp([4,4],   [1,2,3,6,5])
#    = rp([4],     [1,2,3,6,5,4])
#    = rp([],      [1,2,3,6,5,4])
#    = [1,2,3,6,5,4]

# 2ª solución
# ===========

def recorridoEnProfundidad(i: Vertice, g: Grafo) -> list[Vertice]:
    def rp(cs: list[Vertice], vis: list[Vertice]) -> list[Vertice]:
        if not cs:
            return vis
        d, *ds = cs
        if d in vis:
            return rp(ds, vis)
        return rp(adyacentes(g, d) + ds, [d] + vis)
    return list(reversed(rp([i], [])))

# Traza del cálculo de (recorridoEnProfundidad(1, grafo1)
#    recorridoEnProfundidad(1, grafo1)
#    = reverse(rp([1],     []))
#    = reverse(rp([2,3,4], [1]))
#    = reverse(rp([3,4],   [2,1]))
#    = reverse(rp([6,4],   [3,2,1]))
#    = reverse(rp([2,5,4], [6,3,2,1]))
#    = reverse(rp([5,4],   [6,3,2,1]))
#    = reverse(rp([4,4],   [5,6,3,2,1]))
#    = reverse(rp([4],     [4,5,6,3,2,1]))
#    = reverse(rp([],      [4,5,6,3,2,1]))
#    = reverse([4,5,6,3,2,1])
#    = [1,2,3,6,5,4]

# Verificación
# ============
```

```python
def test_recorridoEnProfundidad() -> None:
    grafo2 = creaGrafo_(Orientacion.ND,
                        (1,6),
                        [(1,2),(1,3),(1,4),(3,6),(5,4),(6,2),(6,5)])
    assert recorridoEnProfundidad1(1, grafo1) == [1,2,3,6,5,4]
    assert recorridoEnProfundidad1(1, grafo2) == [1,2,6,3,5,4]
    assert recorridoEnProfundidad(1, grafo1) == [1,2,3,6,5,4]
    assert recorridoEnProfundidad(1, grafo2) == [1,2,6,3,5,4]
    print("Verificado")

# La verificación es
#    >>> test_recorridoEnProfundidad()
#    Verificado
```

11.20. Recorrido en anchura

11.20.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Usando el [tipo abstracto de datos de los grafos](https://bit.ly/45cQ3Fo),
-- definir la función,
--    recorridoEnAnchura :: (Num p, Eq p, Ix v) => v -> Grafo v p -> [v]
-- tal que (recorridoEnAnchura i g) es el recorrido en anchura
-- del grafo g desde el vértice i. Por ejemplo, en el grafo
--
--    +---> 2 <---+
--    |           |
--    |           |
--    1 --> 3 --> 6 --> 5
--    |                 |
--    |                 |
--    +---> 4 <---------+
--
-- definido por
--    grafo1 :: Grafo Int Int
--    grafo1 = creaGrafo' D (1,6) [(1,2),(1,3),(1,4),(3,6),(5,4),(6,2),(6,5)]
-- entonces
--    recorridoEnAnchura 1 grafo1  ==  [1,2,3,4,6,5]
-- ---------------------------------------------------------------------
```

```haskell
module Grafo_Recorrido_en_anchura where

import TAD.Grafo (Grafo, Orientacion (D, ND), adyacentes,
                  creaGrafo')
import Data.Ix (Ix)
import Test.Hspec (Spec, hspec, it, shouldBe)

grafo1 :: Grafo Int Int
grafo1 = creaGrafo' D (1,6) [(1,2),(1,3),(1,4),(3,6),(5,4),(6,2),(6,5)]

recorridoEnAnchura :: (Num p, Eq p, Ix v) => v -> Grafo v p -> [v]
recorridoEnAnchura i g = reverse (ra [i] [])
  where
    ra [] vis    = vis
    ra (c:cs) vis
        | c `elem` vis = ra cs vis
        | otherwise    = ra (cs ++ adyacentes g c) (c:vis)

-- Traza del cálculo de (recorridoEnAnchura1 1 grafo1)
--    recorridoEnAnchura1 1 grafo1
--    = ra [1]     []
--    = ra [2,3,4] [1]
--    = ra [3,4]   [2,1]
--    = ra [4,6]   [3,2,1]
--    = ra [6]     [4,3,2,1]
--    = ra [2,5]   [6,4,3,2,1]
--    = ra [5]     [6,4,3,2,1]
--    = ra [4]     [5,6,4,3,2,1]
--    = ra []      [5,6,4,3,2,1]
--    = [1,2,3,4,6,5]

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

spec :: Spec
spec = do
```

```
  it "e1" $
    recorridoEnAnchura 1 grafo1 `shouldBe` [1,2,3,4,6,5]
  it "e2" $
    recorridoEnAnchura 1 grafo2 `shouldBe` [1,2,3,4,6,5]
  where
    grafo2 :: Grafo Int Int
    grafo2 = creaGrafo' ND (1,6) [(1,2),(1,3),(1,4),(3,6),(5,4),(6,2),(6,5)]

-- La verificación es
--    λ> verifica
--
--    e1
--    e2
--
--    Finished in 0.0010 seconds
--    2 examples, 0 failures
```

11.20.2. En Python

```
# ---------------------------------------------------------------------
# Usando el [tipo abstracto de datos de los grafos](https://bit.ly/45cQ3Fo),
# definir la función,
#    recorridoEnAnchura : (Vertice, Grafo) -> list[Vertice]
# tal que recorridoEnAnchura(i, g) es el recorrido en anchura
# del grafo g desde el vértice i. Por ejemplo, en el grafo
#
#    +---> 2 <---+
#    |           |
#    |           |
#    1 --> 3 --> 6 --> 5
#    |                 |
#    |                 |
#    +---> 4 <---------+
#
# definido por
#    grafo1: Grafo = creaGrafo_(Orientacion.D,
#                               (1,6),
#                               [(1,2),(1,3),(1,4),(3,6),(5,4),(6,2),(6,5)])
# entonces
#    recorridoEnAnchura(1, grafo1)  ==  [1,2,3,4,6,5]
```

```
# ------------------------------------------------------------

from src.TAD.Grafo import Grafo, Orientacion, Vertice, adyacentes, creaGrafo_

grafo1: Grafo = creaGrafo_(Orientacion.D,
                           (1,6),
                           [(1,2),(1,3),(1,4),(3,6),(5,4),(6,2),(6,5)])

def recorridoEnAnchura(i: Vertice, g: Grafo) -> list[Vertice]:
    def ra(cs: list[Vertice], vis: list[Vertice]) -> list[Vertice]:
        if not cs:
            return vis
        d, *ds = cs
        if d in vis:
            return ra(ds, vis)
        return ra(ds + adyacentes(g, d), [d] + vis)
    return list(reversed(ra([i], [])))

# Traza del cálculo de recorridoEnAnchura(1, grafo1)
#    recorridoEnAnchura(1, grafo1
#    = ra([1],     [])
#    = ra([2,3,4], [1])
#    = ra([3,4],   [2,1])
#    = ra([4,6],   [3,2,1])
#    = ra([6],     [4,3,2,1])
#    = ra([2,5],   [6,4,3,2,1])
#    = ra([5],     [6,4,3,2,1])
#    = ra([4],     [5,6,4,3,2,1])
#    = ra([],      [5,6,4,3,2,1])
#    = [1,2,3,4,6,5]

# Verificación
# ============

def test_recorridoEnAnchura() -> None:
    grafo2 = creaGrafo_(Orientacion.ND,
                        (1,6),
                        [(1,2),(1,3),(1,4),(3,6),(5,4),(6,2),(6,5)])
    assert recorridoEnAnchura(1, grafo1) == [1,2,3,4,6,5]
    assert recorridoEnAnchura(1, grafo2) == [1,2,3,4,6,5]
```

```
    print("Verificado")

# La verificación es
#    >>> test_recorridoEnAnchura()
#    Verificado
```

11.21. Grafos conexos

11.21.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Un grafo no dirigido G se dice conexo, si para cualquier par de
-- vértices u y v en G, existe al menos una trayectoria (una  sucesión
-- de vértices adyacentes) de u a v.
--
-- Usando el [tipo abstracto de datos de los grafos](https://bit.ly/45cQ3Fo),
-- definir la función,
--    conexo :: (Ix a, Num p, Eq p) => Grafo a p -> Bool
-- tal que (conexo g) se verifica si el grafo g es conexo. Por ejemplo,
--    conexo (creaGrafo' ND (1,3) [(1,2),(3,2)])        ==  True
--    conexo (creaGrafo' ND (1,4) [(1,2),(3,2),(4,1)])  ==  True
--    conexo (creaGrafo' ND (1,4) [(1,2),(3,4)])        ==  False
-- ---------------------------------------------------------------------

module Grafo_Grafos_conexos where

import TAD.Grafo (Grafo, Orientacion (ND), nodos, creaGrafo')
import Data.Ix (Ix)
import Grafo_Recorrido_en_anchura (recorridoEnAnchura)
import Test.Hspec (Spec, hspec, it, shouldBe)

conexo :: (Ix a, Num p, Eq p) => Grafo a p -> Bool
conexo g = length (recorridoEnAnchura i g) == n
  where xs = nodos g
        i  = head xs
        n  = length xs

-- Verificación
-- ============
```

```
verifica :: IO ()
verifica = hspec spec

spec :: Spec
spec = do
  it "e1" $
    conexo g1 `shouldBe` True
  it "e2" $
    conexo g2 `shouldBe` True
  it "e3" $
    conexo g3 `shouldBe` False
  where
    g1, g2, g3 :: Grafo Int Int
    g1 = creaGrafo' ND (1,3) [(1,2),(3,2)]
    g2 = creaGrafo' ND (1,4) [(1,2),(3,2),(4,1)]
    g3 = creaGrafo' ND (1,4) [(1,2),(3,4)]

-- La verificación es
--    λ> verifica
--
--    e1
--    e2
--    e3
--
--    Finished in 0.0003 seconds
--    3 examples, 0 failures
```

11.21.2. En Python

```
# ---------------------------------------------------------------------
# Un grafo no dirigido G se dice conexo, si para cualquier par de
# vértices u y v en G, existe al menos una trayectoria (una  sucesión
# de vértices adyacentes) de u a v.
#
# Usando el [tipo abstracto de datos de los grafos](https://bit.ly/45cQ3Fo),
# definir la función,
#    conexo :: (Grafo) -> bool
# tal que (conexo g) se verifica si el grafo g es conexo. Por ejemplo,
#    conexo (creaGrafo_(Orientacion.ND, (1,3), [(1,2),(3,2)]))       == True
#    conexo (creaGrafo_(Orientacion.ND, (1,4), [(1,2),(3,2),(4,1)])) == True
```

```
#    conexo (creaGrafo_(Orientacion.ND, (1,4), [(1,2),(3,4)]))       == False
# ---------------------------------------------------------------------

from src.Grafo_Recorrido_en_anchura import recorridoEnAnchura
from src.TAD.Grafo import Grafo, Orientacion, creaGrafo_, nodos


def conexo(g: Grafo) -> bool:
    xs = nodos(g)
    i = xs[0]
    n = len(xs)
    return len(recorridoEnAnchura(i, g)) == n

# Verificación
# ============

def test_conexo() -> None:
    g1 = creaGrafo_(Orientacion.ND, (1,3), [(1,2),(3,2)])
    g2 = creaGrafo_(Orientacion.ND, (1,4), [(1,2),(3,2),(4,1)])
    g3 = creaGrafo_(Orientacion.ND, (1,4), [(1,2),(3,4)])
    assert conexo(g1)
    assert conexo(g2)
    assert not conexo(g3)
    print("Verificado")

# La verificación es
#    >>> test_conexo()
#    Verificado
```

11.22. Coloreado correcto de un mapa

11.22.1. En Haskell

```
-- ---------------------------------------------------------------------
--  Un mapa se puede representar mediante un grafo donde los vértices
--  son las regiones del mapa y hay una arista entre dos vértices si las
--  correspondientes regiones son vecinas. Por ejemplo, el mapa siguiente
--    +----------+----------+
--    |    1     |     2    |
--    +----+-----+-----+----+
```

```
--    |    |         |    |
--    | 3  |    4    | 5  |
--    |    |         |    |
--    +----+----+----+----+
--    |    6    |    7    |
--    +---------+---------+
-- se pueden representar por
--    mapa :: Grafo Int Int
--    mapa = creaGrafo' ND (1,7)
--                        [(1,2),(1,3),(1,4),(2,4),(2,5),(3,4),
--                         (3,6),(4,5),(4,6),(4,7),(5,7),(6,7)]
--
-- Para colorear el mapa se dispone de 4 colores definidos por
--    data Color = A | B | C | D
--      deriving (Eq, Show)
--
-- Usando el [tipo abstracto de datos de los grafos](https://bit.ly/45cQ3Fo),
-- definir la función,
--    correcta :: [(Int,Color)] -> Grafo Int Int -> Bool
-- tal que (correcta ncs m) se verifica si ncs es una coloración del
-- mapa m tal que todos las regiones vecinas tienen colores distintos.
-- Por ejemplo,
--    correcta [(1,A),(2,B),(3,B),(4,C),(5,A),(6,A),(7,B)] mapa == True
--    correcta [(1,A),(2,B),(3,A),(4,C),(5,A),(6,A),(7,B)] mapa == False
-- ---------------------------------------------------------------------

module Grafo_Coloreado_correcto_de_un_mapa where

import TAD.Grafo (Grafo, Orientacion (ND), aristas, creaGrafo')
import Test.Hspec (Spec, hspec, it, shouldBe)

mapa :: Grafo Int Int
mapa = creaGrafo' ND (1,7)
                  [(1,2),(1,3),(1,4),(2,4),(2,5),(3,4),
                   (3,6),(4,5),(4,6),(4,7),(5,7),(6,7)]

data Color = A | B | C | E
  deriving (Eq, Show)

correcta :: [(Int,Color)] -> Grafo Int Int -> Bool
```

```haskell
correcta ncs g =
  and [color x /= color y | ((x,y),_) <- aristas g]
  where color x = head [c | (y,c) <- ncs, y == x]

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

spec :: Spec
spec = do
  it "e1" $
    correcta [(1,A),(2,B),(3,B),(4,C),(5,A),(6,A),(7,B)] mapa `shouldBe` True
  it "e2" $
    correcta [(1,A),(2,B),(3,A),(4,C),(5,A),(6,A),(7,B)] mapa `shouldBe` False

-- La verificación es
--    λ> verifica
--
--    e1
--    e2
--
--    Finished in 0.0004 seconds
--    2 examples, 0 failures
```

11.22.2. En Python

```python
# ---------------------------------------------------------------------
#  Un mapa se puede representar mediante un grafo donde los vértices
#  son las regiones del mapa y hay una arista entre dos vértices si las
#  correspondientes regiones son vecinas. Por ejemplo, el mapa siguiente
#    +----------+----------+
#    |    1     |     2    |
#    +----+-----+-----+----+
#    |    |           |    |
#    | 3  |     4     | 5  |
#    |    |           |    |
#    +----+-----+-----+----+
#    |    6     |     7    |
```

```
#    +----------+----------+
# se pueden representar por
#    mapa: Grafo = creaGrafo_(Orientacion.ND,
#                             (1,7),
#                             [(1,2),(1,3),(1,4),(2,4),(2,5),(3,4),
#                              (3,6),(4,5),(4,6),(4,7),(5,7),(6,7)])
#
# Para colorear el mapa se dispone de 4 colores definidos por
#    Color = Enum('Color', ['A', 'B', 'C', 'E'])
#
# Usando el [tipo abstracto de datos de los grafos](https://bit.ly/45cQ3Fo),
# definir la función,
#    correcta : (list[tuple[int, Color]], Grafo) -> bool
# tal que (correcta ncs m) se verifica si ncs es una coloración del
# mapa m tal que todos las regiones vecinas tienen colores distintos.
# Por ejemplo,
#    correcta [(1,A),(2,B),(3,B),(4,C),(5,A),(6,A),(7,B)] mapa == True
#    correcta [(1,A),(2,B),(3,A),(4,C),(5,A),(6,A),(7,B)] mapa == False
# ---------------------------------------------------------------------


from enum import Enum

from src.TAD.Grafo import Grafo, Orientacion, aristas, creaGrafo_

mapa: Grafo = creaGrafo_(Orientacion.ND,
                         (1,7),
                         [(1,2),(1,3),(1,4),(2,4),(2,5),(3,4),
                          (3,6),(4,5),(4,6),(4,7),(5,7),(6,7)])

Color = Enum('Color', ['A', 'B', 'C', 'E'])

def correcta(ncs: list[tuple[int, Color]], g: Grafo) -> bool:
    def color(x: int) -> Color:
        return [c for (y, c) in ncs if y == x][0]
    return all(color(x) != color(y) for ((x, y), _) in aristas(g))

# Verificación
# ============
```

```python
def test_correcta() -> None:
    assert correcta([(1,Color.A),
                     (2,Color.B),
                     (3,Color.B),
                     (4,Color.C),
                     (5,Color.A),
                     (6,Color.A),
                     (7,Color.B)],
                    mapa)
    assert not correcta([(1,Color.A),
                         (2,Color.B),
                         (3,Color.A),
                         (4,Color.C),
                         (5,Color.A),
                         (6,Color.A),
                         (7,Color.B)],
                        mapa)
    print("Verificado")

# La verificación es
#    >>> test_correcta()
#    Verificado
```

11.23. Nodos aislados de un grafo

11.23.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Dado un grafo dirigido G, diremos que un nodo está aislado si o bien
-- de dicho nodo no sale ninguna arista o bien no llega al nodo ninguna
-- arista. Por ejemplo, en el siguiente grafo
--    grafo1 = creaGrafo D (1,6) [(1,2,0),(1,3,0),(1,4,0),(3,6,0),
--                                (5,4,0),(6,2,0),(6,5,0)]
-- podemos ver que del nodo 1 salen 3 aristas pero no llega ninguna, por
-- lo que lo consideramos aislado. Así mismo, a los nodos 2 y 4 llegan
-- aristas pero no sale ninguna, por tanto también estarán aislados.
--
-- Usando el [tipo abstracto de datos de los grafos](https://bit.ly/45cQ3Fo),
-- definir la función,
--    aislados :: (Ix v, Num p) => Grafo v p -> [v]
```

```haskell
-- tal que (aislados g) es la lista de nodos aislados del grafo g. Por
-- ejemplo,
--    aislados grafo1 == [1,2,4]
-- ---------------------------------------------------------------------

module Grafo_Nodos_aislados_de_un_grafo where

import TAD.Grafo (Grafo, Orientacion (D), adyacentes, nodos, creaGrafo')
import Data.Ix (Ix)
import Grafo_Incidentes_de_un_vertice (incidentes)
import Test.Hspec (Spec, hspec, it, shouldBe)

grafo1 :: Grafo Int Int
grafo1 = creaGrafo' D (1,6) [(1,2),(1,3),(1,4),(3,6),
                             (5,4),(6,2),(6,5)]

aislados :: (Ix v, Num p) => Grafo v p -> [v]
aislados g =
  [n | n <- nodos g, null (adyacentes g n) || null (incidentes g n)]

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

spec :: Spec
spec = do
  it "e1" $
    aislados grafo1 `shouldBe` [1,2,4]

-- La verificación es
--    λ> verifica
--
--    e1
--
--    Finished in 0.0008 seconds
--    1 example, 0 failures
```

11.23.2. En Python

```python
# ---------------------------------------------------------------------
# Dado un grafo dirigido G, diremos que un nodo está aislado si o bien
# de dicho nodo no sale ninguna arista o bien no llega al nodo ninguna
# arista. Por ejemplo, en el siguiente grafo
#    grafo1: Grafo = creaGrafo_(Orientacion.D,
#                               (1,6),
#                               [(1,2),(1,3),(1,4),(3,6),(5,4),(6,2),(6,5)])
# podemos ver que del nodo 1 salen 3 aristas pero no llega ninguna, por
# lo que lo consideramos aislado. Así mismo, a los nodos 2 y 4 llegan
# aristas pero no sale ninguna, por tanto también estarán aislados.
#
# Usando el [tipo abstracto de datos de los grafos](https://bit.ly/45cQ3Fo),
# definir la función,
#    aislados :: (Ix v, Num p) => Grafo v p -> [v]
# tal que (aislados g) es la lista de nodos aislados del grafo g. Por
# ejemplo,
#    aislados grafo1 == [1,2,4]
# ---------------------------------------------------------------------

from src.Grafo_Incidentes_de_un_vertice import incidentes
from src.TAD.Grafo import (Grafo, Orientacion, Vertice, adyacentes, creaGrafo_,
                           nodos)

grafo1: Grafo = creaGrafo_(Orientacion.D,
                           (1,6),
                           [(1,2),(1,3),(1,4),(3,6),(5,4),(6,2),(6,5)])

def aislados(g: Grafo) -> list[Vertice]:
    return [n for n in nodos(g)
            if not adyacentes(g, n) or not incidentes(g, n)]

# Verificación
# ============

def test_aislados() -> None:
    assert aislados(grafo1) == [1, 2, 4]
    print("Verificado")

# La verificación es
```

```
#    >>> test_aislados()
#    Verificado
```

11.24. Nodos conectados en un grafo

11.24.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Usando el [tipo abstracto de datos de los grafos](https://bit.ly/45cQ3Fo),
-- definir la función,
--    conectados :: Grafo Int Int -> Int -> Int -> Bool
-- tal que (conectados g v1 v2) se verifica si los vértices v1 y v2
-- están conectados en el grafo g. Por ejemplo, si grafo1 es el grafo
-- definido por
--    grafo1 :: Grafo Int Int
--    grafo1 = creaGrafo' D (1,6) [(1,3),(1,5),(3,5),(5,1),(5,50),
--                                 (2,4),(2,6),(4,6),(4,4),(6,4)]
-- entonces,
--    conectados grafo1 1 3  ==  True
--    conectados grafo1 1 4  ==  False
--    conectados grafo1 6 2  ==  False
--    conectados grafo1 3 1  ==  True
-- ---------------------------------------------------------------------

module Grafo_Nodos_conectados_en_un_grafo where

import TAD.Grafo (Grafo, Orientacion (D, ND), adyacentes, creaGrafo')
import Data.List (union)
import Test.Hspec (Spec, hspec, it, shouldBe)

conectados :: Grafo Int Int -> Int -> Int -> Bool
conectados g v1 v2 = v2 `elem` conectadosAux g [] [v1]

conectadosAux :: Grafo Int Int -> [Int] -> [Int] -> [Int]
conectadosAux _ vs [] = vs
conectadosAux g vs (w:ws)
  | w `elem` vs = conectadosAux g vs ws
  | otherwise = conectadosAux g ([w] `union` vs) (ws `union` adyacentes g w)

-- Verificación
```

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

verifica :: IO ()
verifica = hspec spec

spec :: Spec
spec = do
  it "e1" $
    conectados grafo1 1 3 `shouldBe` True
  it "e2" $
    conectados grafo1 1 4 `shouldBe` False
  it "e3" $
    conectados grafo1 6 2 `shouldBe` False
  it "e4" $
    conectados grafo1 3 1 `shouldBe` True
  it "e5" $
    conectados grafo2 1 3 `shouldBe` True
  it "e6" $
    conectados grafo2 1 4 `shouldBe` False
  it "e7" $
    conectados grafo2 6 2 `shouldBe` True
  it "e8" $
    conectados grafo2 3 1 `shouldBe` True
  where
    grafo1, grafo2 :: Grafo Int Int
    grafo1 = creaGrafo' D (1,6) [(1,3),(1,5),(3,5),(5,1),(5,50),
                                 (2,4),(2,6),(4,6),(4,4),(6,4)]

    grafo2 = creaGrafo' ND (1,6) [(1,3),(1,5),(3,5),(5,1),(5,50),
                                  (2,4),(2,6),(4,6),(4,4),(6,4)]

-- La verificación es
--    λ> verifica
--
--    e1
--    e2
--    e3
--    e4
--    e5
--    e6
```

```
--    e7
--    e8
--
--    Finished in 0.0032 seconds
--    8 examples, 0 failures
```

11.24.2. En Python

```
# ---------------------------------------------------------------------
# Usando el [tipo abstracto de datos de los grafos](https://bit.ly/45cQ3Fo),
# definir la función,
#    conectados : (Grafo, Vertice, Vertice) -> bool
# tal que conectados(g, v1, v2) se verifica si los vértices v1 y v2
# están conectados en el grafo g. Por ejemplo, si grafo1 es el grafo
# definido por
#    grafo1 = creaGrafo_(Orientacion.D,
#                        (1,6),
#                        [(1,3),(1,5),(3,5),(5,1),(5,50),
#                         (2,4),(2,6),(4,6),(4,4),(6,4)])
# entonces,
#    conectados grafo1 1 3  ==  True
#    conectados grafo1 1 4  ==  False
#    conectados grafo1 6 2  ==  False
#    conectados grafo1 3 1  ==  True
# ----------------------------------------------------------------------------

from src.TAD.Grafo import Grafo, Orientacion, Vertice, adyacentes, creaGrafo_


def unionV(xs: list[Vertice], ys: list[Vertice]) -> list[Vertice]:
    return list(set(xs) | set(ys))

def conectadosAux(g: Grafo, vs: list[Vertice], ws: list[Vertice]) -> list[Vertice
    if not ws:
        return vs
    w, *ws = ws
    if w in vs:
        return conectadosAux(g, vs, ws)
    return conectadosAux(g, unionV([w], vs), unionV(ws, adyacentes(g, w)))
```

```python
def conectados(g: Grafo, v1: Vertice, v2: Vertice) -> bool:
    return v2 in conectadosAux(g, [], [v1])


# Verificación
# ============

def test_conectados() -> None:
    grafo1 = creaGrafo_(Orientacion.D,
                        (1,6),
                        [(1,3),(1,5),(3,5),(5,1),(5,50),
                         (2,4),(2,6),(4,6),(4,4),(6,4)])
    grafo2 = creaGrafo_(Orientacion.ND,
                        (1,6),
                        [(1,3),(1,5),(3,5),(5,1),(5,50),
                         (2,4),(2,6),(4,6),(4,4),(6,4)])
    assert conectados(grafo1, 1, 3)
    assert not conectados(grafo1, 1, 4)
    assert not conectados(grafo1, 6, 2)
    assert conectados(grafo1, 3, 1)
    assert conectados(grafo2, 1, 3)
    assert not conectados(grafo2, 1, 4)
    assert conectados(grafo2, 6, 2)
    assert conectados(grafo2, 3, 1)
    print("Verificado")

# La verificación es
#    >>> test_conectados()
#    Verificado
```

11.25. Algoritmo de Kruskal

11.25.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- El [algoritmo de Kruskal](https://bit.ly/3N8bOOg) calcula un árbol
-- recubridor mínimo en un grafo conexo y ponderado. Es decir, busca un
-- subconjunto de aristas que, formando un árbol, incluyen todos los
-- vértices y donde el valor de la suma de todas las aristas del árbol
-- es el mínimo.
```

```
--
-- El algoritmo de Kruskal funciona de la siguiente manera:
-- + se crea un bosque B (un conjunto de árboles), donde cada vértice
--   del grafo es un árbol separado
-- + se crea un conjunto C que contenga a todas las aristas del grafo
-- + mientras C es no vacío,
--   + eliminar una arista de peso mínimo de C
--   + si esa arista conecta dos árboles diferentes se añade al bosque,
--     combinando los dos árboles en un solo árbol
--   + en caso contrario, se desecha la arista
-- Al acabar el algoritmo, el bosque tiene un solo componente, el cual
-- forma un árbol de expansión mínimo del grafo.
--
-- Usando el [tipo abstracto de datos de los grafos](https://bit.ly/45cQ3Fo),
-- definir la función,
--    kruskal :: (Ix v, Num p, Ord p) => Grafo v p -> [(p,v,v)]
-- tal que (kruskal g) es el árbol de expansión mínimo del grafo g calculado
-- mediante el algoritmo de Kruskal. Por ejemplo, si g1, g2, g3 y g4 son
-- los grafos definidos por
--    g1, g2, g3, g4 :: Grafo Int Int
--    g1 = creaGrafo ND (1,5) [(1,2,12),(1,3,34),(1,5,78),
--                             (2,4,55),(2,5,32),
--                             (3,4,61),(3,5,44),
--                             (4,5,93)]
--    g2 = creaGrafo ND (1,5) [(1,2,13),(1,3,11),(1,5,78),
--                             (2,4,12),(2,5,32),
--                             (3,4,14),(3,5,44),
--                             (4,5,93)]
--    g3 = creaGrafo ND (1,7) [(1,2,5),(1,3,9),(1,5,15),(1,6,6),
--                             (2,3,7),
--                             (3,4,8),(3,5,7),
--                             (4,5,5),
--                             (5,6,3),(5,7,9),
--                             (6,7,11)]
--    g4 = creaGrafo ND (1,7) [(1,2,5),(1,3,9),(1,5,15),(1,6,6),
--                             (2,3,7),
--                             (3,4,8),(3,5,1),
--                             (4,5,5),
--                             (5,6,3),(5,7,9),
--                             (6,7,11)]
```

```haskell
-- entonces
--    kruskal g1  ==  [(55,2,4),(34,1,3),(32,2,5),(12,1,2)]
--    kruskal g2  ==  [(32,2,5),(13,1,2),(12,2,4),(11,1,3)]
--    kruskal g3  ==  [(9,5,7),(7,2,3),(6,1,6),(5,4,5),(5,1,2),(3,5,6)]
--    kruskal g4  ==  [(9,5,7),(6,1,6),(5,4,5),(5,1,2),(3,5,6),(1,3,5)]
-- ---------------------------------------------------------------------

module Grafo_Algoritmo_de_Kruskal where
import TAD.Grafo (Grafo, Orientacion (ND), aristas, creaGrafo, nodos)
import Data.Ix (Ix)
import qualified Data.Map as M (Map, (!), fromList, keys, update)
import Data.List (sort)
import Test.Hspec (Spec, hspec, it, shouldBe)

g1, g2, g3, g4 :: Grafo Int Int
g1 = creaGrafo ND (1,5) [(1,2,12),(1,3,34),(1,5,78),
                         (2,4,55),(2,5,32),
                         (3,4,61),(3,5,44),
                         (4,5,93)]
g2 = creaGrafo ND (1,5) [(1,2,13),(1,3,11),(1,5,78),
                         (2,4,12),(2,5,32),
                         (3,4,14),(3,5,44),
                         (4,5,93)]
g3 = creaGrafo ND (1,7) [(1,2,5),(1,3,9),(1,5,15),(1,6,6),
                         (2,3,7),
                         (3,4,8),(3,5,7),
                         (4,5,5),
                         (5,6,3),(5,7,9),
                         (6,7,11)]
g4 = creaGrafo ND (1,7) [(1,2,5),(1,3,9),(1,5,15),(1,6,6),
                         (2,3,7),
                         (3,4,8),(3,5,1),
                         (4,5,5),
                         (5,6,3),(5,7,9),
                         (6,7,11)]

kruskal :: (Ix v, Num p, Ord p) => Grafo v p -> [(p,v,v)]
kruskal g = aux (sort [(p,x,y) | ((x,y),p) <- aristas g])
                (M.fromList [(x,x) | x <- nodos g])
                []
```

```
                (length (nodos g) - 1)
  where aux _  _ ae 0 = ae
        aux [] _ _  _ = error "Imposible"
        aux ((p,x,y):as) d ae n
          | actualizado = aux as d' ((p,x,y):ae) (n-1)
          | otherwise   = aux as d ae n
          where (actualizado,d') = buscaActualiza (x,y) d

-- (raiz d n) es la raíz de n en el diccionario. Por ejemplo,
--    raiz (M.fromList [(1,1),(3,1),(4,3),(5,4),(2,6),(6,6)]) 5  == 1
--    raiz (M.fromList [(1,1),(3,1),(4,3),(5,4),(2,6),(6,6)]) 2  == 6
raiz :: (Eq n, Ord n) => M.Map n n -> n -> n
raiz d x | v == x    = v
         | otherwise = raiz d v
  where v = d M.! x

-- (buscaActualiza a d) es el par formado por False y el diccionario d,
-- si los dos vértices de la arista a tienen la misma raíz en d y el par
-- formado por True y la tabla obtenida añadiéndole a d la arista
-- formada por el vértice de a de mayor raíz y la raíz del vértice de a
-- de menor raíz. Y actualizando las raices de todos los elementos
-- afectados por la raíz añadida. Por ejemplo,
--   λ> d = M.fromList [(1,1),(2,1),(3,3),(4,4),(5,5),(6,5),(7,7)]
--   λ> buscaActualiza (5,4) d
--   (True,fromList [(1,1),(2,1),(3,3),(4,4),(5,4),(6,4),(7,7)])
--   λ> d' = snd it
--   λ> buscaActualiza (6,1) d'
--   (True,fromList [(1,1),(2,1),(3,3),(4,1),(5,1),(6,1),(7,7)])
buscaActualiza :: (Eq n, Ord n) => (n,n) -> M.Map n n -> (Bool,M.Map n n)
buscaActualiza (x,y) d
  | x' == y'  = (False, d)
  | y' <  x'  = (True, modificaR x (d M.! x) y' d)
  | otherwise = (True, modificaR y (d M.! y) x' d)
  where x' = raiz d x
        y' = raiz d y

-- (modificaR x y y' d) actualiza d como sigue:
-- + el valor de todas las claves z con valor y es y'
-- + el valor de todas las claves z con (z > x) con valor x es y'
modificaR :: (Eq n, Ord n) => n -> n -> n -> M.Map n n -> M.Map n n
```

```
modificaR x y y' d = aux2 ds (aux1 cs d)
  where cs = M.keys d
        ds = filter (>x) cs
        aux1 [] tb = tb
        aux1 (a:as) tb | tb M.! a == y = aux1 as (M.update (\_ -> Just y') a tb)
                       | otherwise     = aux1 as tb
        aux2 [] tb = tb
        aux2 (b:bs) tb | tb M.! b == x = aux2 bs (M.update (\_ -> Just y') b tb)
                       | otherwise     = aux2 bs tb

-- Traza del diccionario correspondiente al grafo g3
-- ===================================================

-- Lista de aristas, ordenadas según su peso:
-- [(3,5,6),(5,1,2),(5,4,5),(6,1,6),(7,2,3),(7,3,5),(8,3,4),(9,1,3),(9,5,7),(11,6
--
-- Inicial
--   fromList [(1,1),(2,2),(3,3),(4,4),(5,5),(6,6),(7,7)]
--
-- Después de añadir la arista (5,6) de peso 3
--   fromList [(1,1),(2,2),(3,3),(4,4),(5,5),(6,5),(7,7)]
--
-- Después de añadir la arista (1,2) de peso 5
--   fromList [(1,1),(2,1),(3,3),(4,4),(5,5),(6,5),(7,7)]
--
-- Después de añadir la arista (4,5) de peso 5
--   fromList [(1,1),(2,1),(3,3),(4,4),(5,4),(6,4),(7,7)]
--
-- Después de añadir la arista (1,6) de peso 6
--   fromList [(1,1),(2,1),(3,3),(4,1),(5,1),(6,1),(7,7)]
--
-- Después de añadir la arista (2,3) de peso 7
--   fromList [(1,1),(2,1),(3,1),(4,1),(5,1),(6,1),(7,7)]
--
-- Las posibles aristas a añadir son:
-- + la (3,5) con peso 7, que no es posible pues la raíz de 3
--   coincide con la raíz de 5, por lo que formaría un ciclo
-- + la (3,4) con peso 8, que no es posible pues la raíz de 3
--   coincide con la raíz de 4, por lo que formaría un ciclo
-- + la (1,3) con peso 9, que no es posible pues la raíz de 3
```

```
--   coincide con la raíz de 1, por lo que formaría un ciclo
-- + la (5,7) con peso 9, que no forma ciclo
--
-- Después de añadir la arista (5,7) con peso 9
--    fromList [(1,1),(2,1),(3,1),(4,1),(5,1),(6,1),(7,1)]
--
-- No es posible añadir más aristas, pues formarían ciclos.

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

spec :: Spec
spec = do
  it "e1" $
    kruskal g1 `shouldBe` [(55,2,4),(34,1,3),(32,2,5),(12,1,2)]
  it "e2" $
    kruskal g2 `shouldBe` [(32,2,5),(13,1,2),(12,2,4),(11,1,3)]
  it "e3" $
    kruskal g3 `shouldBe` [(9,5,7),(7,2,3),(6,1,6),(5,4,5),(5,1,2),(3,5,6)]
  it "e4" $
    kruskal g4 `shouldBe` [(9,5,7),(6,1,6),(5,4,5),(5,1,2),(3,5,6),(1,3,5)]

-- La verificación es
--    λ> verifica
--
--    e1
--    e2
--    e3
--    e4
--
--    Finished in 0.0044 seconds
--    4 examples, 0 failures
```

11.25.2. En Python

```
# ---------------------------------------------------------------------
# El [algoritmo de Kruskal]()https://bit.ly/3N8bOOg) calcula un árbol
```

```
# recubridor mínimo en un grafo conexo y ponderado. Es decir, busca un
# subconjunto de aristas que, formando un árbol, incluyen todos los
# vértices y donde el valor de la suma de todas las aristas del árbol
# es el mínimo.
#
# El algoritmo de Kruskal funciona de la siguiente manera:
# + se crea un bosque B (un conjunto de árboles), donde cada vértice
#   del grafo es un árbol separado
# + se crea un conjunto C que contenga a todas las aristas del grafo
# + mientras C es no vacío,
#   + eliminar una arista de peso mínimo de C
#   + si esa arista conecta dos árboles diferentes se añade al bosque,
#     combinando los dos árboles en un solo árbol
#   + en caso contrario, se desecha la arista
# Al acabar el algoritmo, el bosque tiene un solo componente, el cual
# forma un árbol de expansión mínimo del grafo.
#
# Usando el [tipo abstracto de datos de los grafos](https://bit.ly/45cQ3Fo),
# definir la función,
#    kruskal : (Grafo) -> list[tuple[Peso, Vertice, Vertice]]
# tal que kruskal(g) es el árbol de expansión mínimo del grafo g calculado
# mediante el algoritmo de Kruskal. Por ejemplo, si g1, g2, g3 y g4 son
# los grafos definidos por
#    g1 = creaGrafo (Orientacion.ND,
#                    (1,5),
#                    [((1,2),12),((1,3),34),((1,5),78),
#                     ((2,4),55),((2,5),32),
#                     ((3,4),61),((3,5),44),
#                     ((4,5),93)])
#    g2 = creaGrafo (Orientacion.ND,
#                    (1,5),
#                    [((1,2),13),((1,3),11),((1,5),78),
#                     ((2,4),12),((2,5),32),
#                     ((3,4),14),((3,5),44),
#                     ((4,5),93)])
#    g3 = creaGrafo (Orientacion.ND,
#                    (1,7),
#                    [((1,2),5),((1,3),9),((1,5),15),((1,6),6),
#                     ((2,3),7),
#                     ((3,4),8),((3,5),7),
```

```
#                     ((4,5),5),
#                     ((5,6),3),((5,7),9),
#                     ((6,7),11)])
#    g4 = creaGrafo (Orientacion.ND,
#                    (1,7),
#                    [((1,2),5),((1,3),9),((1,5),15),((1,6),6),
#                     ((2,3),7),
#                     ((3,4),8),((3,5),1),
#                     ((4,5),5),
#                     ((5,6),3),((5,7),9),
#                     ((6,7),11)])
# entonces
#    kruskal(g1) == [(55,2,4),(34,1,3),(32,2,5),(12,1,2)]
#    kruskal(g2) == [(32,2,5),(13,1,2),(12,2,4),(11,1,3)]
#    kruskal(g3) == [(9,5,7),(7,2,3),(6,1,6),(5,4,5),(5,1,2),(3,5,6)]
#    kruskal(g4) == [(9,5,7),(6,1,6),(5,4,5),(5,1,2),(3,5,6),(1,3,5)]
# ---------------------------------------------------------------------

from src.TAD.Grafo import (Grafo, Orientacion, Peso, Vertice, aristas,
                           creaGrafo, nodos)

g1 = creaGrafo (Orientacion.ND,
                (1,5),
                [((1,2),12),((1,3),34),((1,5),78),
                 ((2,4),55),((2,5),32),
                 ((3,4),61),((3,5),44),
                 ((4,5),93)])
g2 = creaGrafo (Orientacion.ND,
                (1,5),
                [((1,2),13),((1,3),11),((1,5),78),
                 ((2,4),12),((2,5),32),
                 ((3,4),14),((3,5),44),
                 ((4,5),93)])
g3 = creaGrafo (Orientacion.ND,
                (1,7),
                [((1,2),5),((1,3),9),((1,5),15),((1,6),6),
                 ((2,3),7),
                 ((3,4),8),((3,5),7),
                 ((4,5),5),
                 ((5,6),3),((5,7),9),
```

```
                 ((6,7),11)])
g4 = creaGrafo (Orientacion.ND,
                (1,7),
                [((1,2),5),((1,3),9),((1,5),15),((1,6),6),
                 ((2,3),7),
                 ((3,4),8),((3,5),1),
                 ((4,5),5),
                 ((5,6),3),((5,7),9),
                 ((6,7),11)])

# raiz(d, n) es la raíz de n en el diccionario. Por ejemplo,
#    raiz({1:1, 3:1, 4:3, 5:4, 2:6, 6:6}, 5)  == 1
#    raiz({1:1, 3:1, 4:3, 5:4, 2:6, 6:6}, 2)  == 6
def raiz(d: dict[Vertice, Vertice], x: Vertice) -> Vertice:
    v = d[x]
    if v == x:
        return v
    return raiz(d, v)

# modificaR(x, y, y_, d) actualiza d como sigue:
# + el valor de todas las claves z con valor y es y_
# + el valor de todas las claves z con (z > x) con valor x es y_
def modificaR(x: Vertice,
              y: Vertice,
              y_: Vertice,
              d: dict[Vertice, Vertice]) -> dict[Vertice, Vertice]:
    def aux1(vs: list[Vertice],
             tb: dict[Vertice, Vertice],
             y: Vertice) -> dict[Vertice, Vertice]:
        for a in vs:
            if tb[a] == y:
                tb[a] = y_
        return tb

    def aux2(vs: list[Vertice],
             tb: dict[Vertice, Vertice],
             y_: Vertice) -> dict[Vertice, Vertice]:
        for b in vs:
            if tb[b] == x:
                tb[b] = y_
```

```
        return tb

    cs = list(d.keys())
    ds = [c for c in cs if c > x]

    tb = aux1(cs, d, y)
    tb = aux2(ds, tb, y_)

    return tb

# buscaActualiza(a, d) es el par formado por False y el diccionario d,
# si los dos vértices de la arista a tienen la misma raíz en d y el par
# formado por True y la tabla obtenida añadiéndole a d la arista
# formada por el vértice de a de mayor raíz y la raíz del vértice de a
# de menor raíz. Y actualizando las raices de todos los elementos
# afectados por la raíz añadida. Por ejemplo,
#    >>> buscaActualiza((5,4), {1:1, 2:1, 3:3, 4:4, 5:5, 6:5, 7:7})
#    (True, {1: 1, 2: 1, 3: 3, 4: 4, 5: 4, 6: 4, 7: 7})
#    >>> buscaActualiza((6,1), {1:1, 2:1, 3:3, 4:4, 5:4, 6:4, 7:7})
#    (True, {1: 1, 2: 1, 3: 3, 4: 1, 5: 1, 6: 1, 7: 7})
#    >>> buscaActualiza((6,2), {1:1, 2:1, 3:3, 4:1, 5:4, 6:5, 7:7})
#    (False, {1: 1, 2: 1, 3: 3, 4: 1, 5: 4, 6: 5, 7: 7})
def buscaActualiza(a: tuple[Vertice, Vertice],
                   d: dict[Vertice, Vertice]) -> tuple[bool,
                                                       dict[Vertice, Vertice]]:
    x, y = a
    x_ = raiz(d, x)
    y_ = raiz(d, y)

    if x_ == y_:
        return False, d
    if y_ < x_:
        return True, modificaR(x, d[x], y_, d)
    return True, modificaR(y, d[y], x_, d)

def kruskal(g: Grafo) -> list[tuple[Peso, Vertice, Vertice]]:
    def aux(as_: list[tuple[Peso, Vertice, Vertice]],
            d: dict[Vertice, Vertice],
            ae: list[tuple[Peso, Vertice, Vertice]],
            n: int) -> list[tuple[Peso, Vertice, Vertice]]:
```

```
        if n == 0:
            return ae
        p, x, y = as_[0]
        actualizado, d = buscaActualiza((x, y), d)
        if actualizado:
            return aux(as_[1:], d, [(p, x, y)] + ae, n - 1)
        return aux(as_[1:], d, ae, n)
    return aux(list(sorted([(p, x, y) for ((x, y), p) in aristas(g)])),
               {x: x for x in nodos(g)},
               [],
               len(nodos(g)) - 1)

# Traza del diccionario correspondiente al grafo g3
# =================================================

# Lista de aristas, ordenadas según su peso:
# [(3,5,6),(5,1,2),(5,4,5),(6,1,6),(7,2,3),(7,3,5),(8,3,4),(9,1,3),(9,5,7),(11,6,
#
# Inicial
#   {1:1, 2:2, 3:3, 4:4, 5:5, 6:6, 7:7}
#
# Después de añadir la arista (5,6) de peso 3
#   {1:1, 2:2, 3:3, 4:4, 5:5, 6:5, 7:7}
#
# Después de añadir la arista (1,2) de peso 5
#   {1:1, 2:1, 3:3, 4:4, 5:5, 6:5, 7:7}
#
# Después de añadir la arista (4,5) de peso 5
#   {1:1, 2:1, 3:3, 4:4, 5:4, 6:4, 7:7}
#
# Después de añadir la arista (1,6) de peso 6
#   {1:1, 2:1, 3:3, 4:1, 5:1, 6:1, 7:7}
#
# Después de añadir la arista (2,3) de peso 7
#   {1:1, 2:1, 3:1, 4:1, 5:1, 6:1, 7:7}
#
# Las posibles aristas a añadir son:
# + la (3,5) con peso 7, que no es posible pues la raíz de 3
#   coincide con la raíz de 5, por lo que formaría un ciclo
# + la (3,4) con peso 8, que no es posible pues la raíz de 3
```

```
#   coincide con la raíz de 4, por lo que formaría un ciclo
# + la (1,3) con peso 9, que no es posible pues la raíz de 3
#   coincide con la raíz de 1, por lo que formaría un ciclo
# + la (5,7) con peso 9, que no forma ciclo
#
# Después de añadir la arista (5,7) con peso 9
#    {1:1, 2:1, 3:1, 4:1, 5:1, 6:1, 7:1}
#
# No es posible añadir más aristas, pues formarían ciclos.

# Verificación
# ============

def test_kruskal() -> None:
    assert kruskal(g1) == [(55,2,4),(34,1,3),(32,2,5),(12,1,2)]
    assert kruskal(g2) == [(32,2,5),(13,1,2),(12,2,4),(11,1,3)]
    assert kruskal(g3) == [(9,5,7),(7,2,3),(6,1,6),(5,4,5),(5,1,2),(3,5,6)]
    assert kruskal(g4) == [(9,5,7),(6,1,6),(5,4,5),(5,1,2),(3,5,6),(1,3,5)]
    print("Vefificado")

# La verificación es
#    >>> test_kruskal()
#    Vefificado
```

11.26. Algoritmo de Prim

11.26.1. En Haskell

```
-- ---------------------------------------------------------------------
-- El [algoritmo de Prim](https://bit.ly/466fwRe) calcula un árbol
-- recubridor mínimo en un grafo conexo y ponderado. Es decir, busca un
-- subconjunto de aristas que, formando un árbol, incluyen todos los
-- vértices y donde el valor de la suma de todas las aristas del árbol
-- es el mínimo.
--
-- El algoritmo de Prim funciona de la siguiente manera:
-- + Inicializar un árbol con un único vértice, elegido arbitrariamente,
--   del grafo.
-- + Aumentar el árbol por un lado. Llamamos lado a la unión entre dos
--   vértices: de las posibles uniones que pueden conectar el árbol a los
```

```haskell
--   vértices que no están aún en el árbol, encontrar el lado de menor
--   distancia y unirlo al árbol.
-- + Repetir el paso 2 (hasta que todos los vértices pertenezcan al
--   árbol)
--
-- Usando el [tipo abstracto de datos de los grafos](https://bit.ly/45cQ3Fo),
-- definir la función,
--    prim :: (Ix v, Num p, Ord p) => Grafo v p -> [(p,v,v)]
-- tal que (prim g) es el árbol de expansión mínimo del grafo g
-- calculado mediante el algoritmo de Prim. Por ejemplo, si g1, g2, g3 y
-- g4 son los grafos definidos por
--    g1, g2, g3, g4 :: Grafo Int Int
--    g1 = creaGrafo ND (1,5) [(1,2,12),(1,3,34),(1,5,78),
--                             (2,4,55),(2,5,32),
--                             (3,4,61),(3,5,44),
--                             (4,5,93)]
--    g2 = creaGrafo ND (1,5) [(1,2,13),(1,3,11),(1,5,78),
--                             (2,4,12),(2,5,32),
--                             (3,4,14),(3,5,44),
--                             (4,5,93)]
--    g3 = creaGrafo ND (1,7) [(1,2,5),(1,3,9),(1,5,15),(1,6,6),
--                             (2,3,7),
--                             (3,4,8),(3,5,7),
--                             (4,5,5),
--                             (5,6,3),(5,7,9),
--                             (6,7,11)]
--    g4 = creaGrafo ND (1,7) [(1,2,5),(1,3,9),(1,5,15),(1,6,6),
--                             (2,3,7),
--                             (3,4,8),(3,5,1),
--                             (4,5,5),
--                             (5,6,3),(5,7,9),
--                             (6,7,11)]
-- entonces
--    prim g1  == [(55,2,4),(34,1,3),(32,2,5),(12,1,2)]
--    prim g2  == [(32,2,5),(12,2,4),(13,1,2),(11,1,3)]
--    prim g3  == [(9,5,7),(7,2,3),(5,5,4),(3,6,5),(6,1,6),(5,1,2)]
-- ---------------------------------------------------------------------

module Grafo_Algoritmo_de_Prim where
```

```haskell
import TAD.Grafo (Grafo, Orientacion (ND), aristas, creaGrafo, nodos)
import Data.Ix (Ix)
import Data.List (delete)
import Test.Hspec (Spec, hspec, it, shouldBe)

g1, g2, g3, g4 :: Grafo Int Int
g1 = creaGrafo ND (1,5) [(1,2,12),(1,3,34),(1,5,78),
                          (2,4,55),(2,5,32),
                          (3,4,61),(3,5,44),
                          (4,5,93)]
g2 = creaGrafo ND (1,5) [(1,2,13),(1,3,11),(1,5,78),
                          (2,4,12),(2,5,32),
                          (3,4,14),(3,5,44),
                          (4,5,93)]
g3 = creaGrafo ND (1,7) [(1,2,5),(1,3,9),(1,5,15),(1,6,6),
                          (2,3,7),
                          (3,4,8),(3,5,7),
                          (4,5,5),
                          (5,6,3),(5,7,9),
                          (6,7,11)]
g4 = creaGrafo ND (1,7) [(1,2,5),(1,3,9),(1,5,15),(1,6,6),
                          (2,3,7),
                          (3,4,8),(3,5,1),
                          (4,5,5),
                          (5,6,3),(5,7,9),
                          (6,7,11)]

prim :: (Ix v, Num p, Ord p) => Grafo v p -> [(p,v,v)]
prim g = prim' [n]              -- Nodos colocados
               ns               -- Nodos por colocar
               []               -- Árbol de expansión
               (aristas g)      -- Aristas del grafo
  where
    (n:ns) = nodos g
    prim' _ _ _  []   = []
    prim' _ [] ae _   = ae
    prim' t r  ae as = prim' (v':t) (delete v' r) (e:ae) as
      where e@(_,_, v') = minimum [(c,u,v)| ((u,v),c) <- as,
                                            u `elem` t,
                                            v `elem` r]
```

```
-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

spec :: Spec
spec = do
  it "e1" $
    prim g1 `shouldBe` [(55,2,4),(34,1,3),(32,2,5),(12,1,2)]
  it "e2" $
    prim g2 `shouldBe` [(32,2,5),(12,2,4),(13,1,2),(11,1,3)]
  it "e3" $
    prim g3 `shouldBe` [(9,5,7),(7,2,3),(5,5,4),(3,6,5),(6,1,6),(5,1,2)]

-- La verificación es
--    λ> verifica
--
--    e1
--    e2
--    e3
--
--    Finished in 0.0026 seconds
--    3 examples, 0 failures
```

11.26.2. En Python

```
# ---------------------------------------------------------------------
# El [algoritmo de Prim](https://bit.ly/466fwRe) calcula un árbol
# recubridor mínimo en un grafo conexo y ponderado. Es decir, busca un
# subconjunto de aristas que, formando un árbol, incluyen todos los
# vértices y donde el valor de la suma de todas las aristas del árbol
# es el mínimo.
#
# El algoritmo de Prim funciona de la siguiente manera:
# + Inicializar un árbol con un único vértice, elegido arbitrariamente,
#   del grafo.
# + Aumentar el árbol por un lado. Llamamos lado a la unión entre dos
#   vértices: de las posibles uniones que pueden conectar el árbol a los
```

```
#   vértices que no están aún en el árbol, encontrar el lado de menor
#   distancia y unirlo al árbol.
# + Repetir el paso 2 (hasta que todos los vértices pertenezcan al
#   árbol)
#
# Usando el [tipo abstracto de datos de los grafos](https://bit.ly/45cQ3Fo),
# definir la función,
#    prim : (Grafo) -> list[tuple[Peso, Vertice, Vertice]]
# tal que prim(g) es el árbol de expansión mínimo del grafo g
# calculado mediante el algoritmo de Prim. Por ejemplo, si g1, g2, g3 y
# g4 son los grafos definidos por
#    g1 = creaGrafo (Orientacion.ND,
#                    (1,5),
#                    [((1,2),12),((1,3),34),((1,5),78),
#                     ((2,4),55),((2,5),32),
#                     ((3,4),61),((3,5),44),
#                     ((4,5),93)])
#    g2 = creaGrafo (Orientacion.ND,
#                    (1,5),
#                    [((1,2),13),((1,3),11),((1,5),78),
#                     ((2,4),12),((2,5),32),
#                     ((3,4),14),((3,5),44),
#                     ((4,5),93)])
#    g3 = creaGrafo (Orientacion.ND,
#                    (1,7),
#                    [((1,2),5),((1,3),9),((1,5),15),((1,6),6),
#                     ((2,3),7),
#                     ((3,4),8),((3,5),7),
#                     ((4,5),5),
#                     ((5,6),3),((5,7),9),
#                     ((6,7),11)])
#    g4 = creaGrafo (Orientacion.ND,
#                    (1,7),
#                    [((1,2),5),((1,3),9),((1,5),15),((1,6),6),
#                     ((2,3),7),
#                     ((3,4),8),((3,5),1),
#                     ((4,5),5),
#                     ((5,6),3),((5,7),9),
#                     ((6,7),11)])
# entonces
```

```
#    prim(g1)  == [(55,2,4),(34,1,3),(32,2,5),(12,1,2)]
#    prim(g2)  == [(32,2,5),(12,2,4),(13,1,2),(11,1,3)]
#    prim(g3)  == [(9,5,7),(7,2,3),(5,5,4),(3,6,5),(6,1,6),(5,1,2)]
# ---------------------------------------------------------------------

from src.TAD.Grafo import (Grafo, Orientacion, Peso, Vertice, aristas,
                           creaGrafo, nodos)

g1 = creaGrafo (Orientacion.ND,
                (1,5),
                [((1,2),12),((1,3),34),((1,5),78),
                 ((2,4),55),((2,5),32),
                 ((3,4),61),((3,5),44),
                 ((4,5),93)])
g2 = creaGrafo (Orientacion.ND,
                (1,5),
                [((1,2),13),((1,3),11),((1,5),78),
                 ((2,4),12),((2,5),32),
                 ((3,4),14),((3,5),44),
                 ((4,5),93)])
g3 = creaGrafo (Orientacion.ND,
                (1,7),
                [((1,2),5),((1,3),9),((1,5),15),((1,6),6),
                 ((2,3),7),
                 ((3,4),8),((3,5),7),
                 ((4,5),5),
                 ((5,6),3),((5,7),9),
                 ((6,7),11)])
g4 = creaGrafo (Orientacion.ND,
                (1,7),
                [((1,2),5),((1,3),9),((1,5),15),((1,6),6),
                 ((2,3),7),
                 ((3,4),8),((3,5),1),
                 ((4,5),5),
                 ((5,6),3),((5,7),9),
                 ((6,7),11)])

def prim(g: Grafo) -> list[tuple[Peso, Vertice, Vertice]]:
    n, *ns = nodos(g)
    def prim_(t: list[Vertice],
```

```
              r: list[Vertice],
              ae: list[tuple[Peso, Vertice, Vertice]],
              as_: list[tuple[tuple[Vertice, Vertice], Peso]]) \
              -> list[tuple[Peso, Vertice, Vertice]]:
        if not as_:
            return []
        if not r:
            return ae
        e = min(((c,u,v)
                 for ((u,v),c) in as_
                 if u in t and v in r))
        (_,_, v_) = e
        return prim_([v_] + t, [x for x in r if x != v_], [e] + ae, as_)
    return prim_([n], ns, [], aristas(g))

# Verificación
# ============

def test_prim() -> None:
    assert prim(g1)  == [(55,2,4),(34,1,3),(32,2,5),(12,1,2)]
    assert prim(g2)  == [(32,2,5),(12,2,4),(13,1,2),(11,1,3)]
    assert prim(g3)  == [(9,5,7),(7,2,3),(5,5,4),(3,6,5),(6,1,6),(5,1,2)]
    print("Verificado")

# La verificación es
#    >>> test_prim()
#    Verificado
```

Capítulo 12

Divide y vencerás

Contenido

12.1. Algoritmo divide y vencerás

12.1.1. En Haskell

```
-- ---------------------------------------------------------------------
-- La técnica [divide y vencerás](https://bit.ly/46afaca) consta de
-- los siguientes pasos:
-- + Dividir el problema en subproblemas menores.
-- + Resolver por separado cada uno de los subproblemas:
--   + si los subproblemas son complejos, usar la misma técnica recursivamente;
--   + si son simples, resolverlos directamente.
-- + Combinar todas las soluciones de los subproblemas en una solución simple.
--
-- Definir la función
--    divideVenceras :: (p -> Bool)
--                   -> (p -> s)
```

```
--                    -> (p -> [p])
--                    -> (p -> [s] -> s)
--                    -> p
--                    -> s
-- tal que (divideVenceras ind resuelve divide combina pbInicial)
-- resuelve el problema pbInicial mediante la técnica de divide y
-- vencerás, donde
-- + (ind pb) se verifica si el problema pb es indivisible
-- + (resuelve pb) es la solución del problema indivisible pb
-- + (divide pb) es la lista de subproblemas de pb
-- + (combina pb ss) es la combinación de las soluciones ss de los
--   subproblemas del problema pb.
-- + pbInicial es el problema inicial
--
-- Usando la función DivideVenceras definir las funciones
--    ordenaPorMezcla :: Ord a => [a] -> [a]
--    ordenaRapida    :: Ord a => [a] -> [a]
-- tales que
-- + (ordenaPorMezcla xs) es la lista obtenida ordenando xs por el
--   procedimiento de ordenación por mezcla. Por ejemplo,
--      λ> ordenaPorMezcla [3,1,4,1,5,9,2,8]
--      [1,1,2,3,4,5,8,9]
-- + (ordenaRapida xs) es la lista obtenida ordenando xs por el
--   procedimiento de ordenación rápida. Por ejemplo,
--      λ> ordenaRapida [3,1,4,1,5,9,2,8]
--      [1,1,2,3,4,5,8,9]
-- ---------------------------------------------------------------------

{-# OPTIONS_GHC -fno-warn-unused-top-binds #-}
{-# OPTIONS_GHC -fno-warn-incomplete-patterns #-}

module DivideVenceras (divideVenceras) where

import Test.Hspec (Spec, hspec, it, shouldBe)

divideVenceras :: (p -> Bool)
               -> (p -> s)
               -> (p -> [p])
               -> (p -> [s] -> s)
               -> p
```

```haskell
               -> s
divideVenceras ind resuelve divide combina = dv'
  where
    dv' pb
      | ind pb    = resuelve pb
      | otherwise = combina pb [dv' sp | sp <- divide pb]

ordenaPorMezcla :: Ord a => [a] -> [a]
ordenaPorMezcla =
    divideVenceras ind id divide combina
    where
      ind xs             = length xs <= 1
      divide xs          = [take n xs, drop n xs]
                           where n = length xs `div` 2
      combina _ [l1,l2] = mezcla l1 l2

-- (mezcla xs ys) es la lista obtenida mezclando xs e ys. Por ejemplo,
--    mezcla [1,3] [2,4,6]  ==  [1,2,3,4,6]
mezcla :: Ord a => [a] -> [a] -> [a]
mezcla [] b = b
mezcla a [] = a
mezcla a@(x:xs) b@(y:ys) | x <= y    = x : mezcla xs b
                         | otherwise = y : mezcla a ys

ordenaRapida :: Ord a => [a] -> [a]
ordenaRapida =
    divideVenceras ind id divide combina
    where
      ind xs                 = length xs <= 1
      divide (x:xs)          = [[ y | y <- xs, y <= x],
                                 [ y | y <- xs, y > x]]
      combina (x:_) [l1,l2] = l1 ++ [x] ++ l2

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

spec :: Spec
```

```
spec = do
  it "e1" $
    ordenaPorMezcla [3,1,4,1,5,9,2,8] `shouldBe` [1,1,2,3,4,5,8,9]
  it "e2" $
    ordenaRapida [3,1,4,1,5,9,2,8] `shouldBe` [1,1,2,3,4,5,8,9]

-- La verificación es
--    λ> verifica
--
--    e1
--    e2
--
--    Finished in 0.0004 seconds
--    2 examples, 0 failures
```

12.1.2. En Python

```
# ---------------------------------------------------------------------
# La técnica [divide y vencerás](https://bit.ly/46afaca) consta de
# los siguientes pasos:
# + Dividir el problema en subproblemas menores.
# + Resolver por separado cada uno de los subproblemas:
#   + si los subproblemas son complejos, usar la misma técnica recursivamente;
#   + si son simples, resolverlos directamente.
# + Combinar todas las soluciones de los subproblemas en una solución simple.
#
# Definir la función
#    divideVenceras(Callable[[P], bool],
#                   Callable[[P], S],
#                   Callable[[P], list[P]],
#                   Callable[[P, list[S]], S],
#                   P) -> S:
# tal que divideVenceras(ind, resuelve, divide, combina, pbInicial)
# resuelve el problema pbInicial mediante la técnica de divide y
# vencerás, donde
# + ind(pb) se verifica si el problema pb es indivisible
# + resuelve(pb) es la solución del problema indivisible pb
# + divide(pb) es la lista de subproblemas de pb
# + combina(pb, ss) es la combinación de las soluciones ss de los
#   subproblemas del problema pb.
```

```python
# + pbInicial es el problema inicial
#
# Usando la función DivideVenceras definir las funciones
#    ordenaPorMezcla : (list[int]) -> list[int]
#    ordenaRapida    : (list[int]) -> list[int]
# tales que
# + ordenaPorMezcla(xs) es la lista obtenida ordenando xs por el
#   procedimiento de ordenación por mezcla. Por ejemplo,
#      >>> ordenaPorMezcla([3,1,4,1,5,9,2,8])
#      [1, 1, 2, 3, 4, 5, 8, 9]
# + ordenaRapida(xs) es la lista obtenida ordenando xs por el
#   procedimiento de ordenación rápida. Por ejemplo,
#      λ> ordenaRapida([3,1,4,1,5,9,2,8])
#      [1, 1, 2, 3, 4, 5, 8, 9]
# ---------------------------------------------------------------------

from typing import Callable, TypeVar

P = TypeVar('P')
S = TypeVar('S')

def divideVenceras(ind: Callable[[P], bool],
                   resuelve: Callable[[P], S],
                   divide: Callable[[P], list[P]],
                   combina: Callable[[P, list[S]], S],
                   p: P) -> S:
    def dv(pb: P) -> S:
        if ind(pb):
            return resuelve(pb)
        return combina(pb, [dv(sp) for sp in divide(pb)])
    return dv(p)

def ordenaPorMezcla(xs: list[int]) -> list[int]:
    def ind(xs: list[int]) -> bool:
        return len(xs) <= 1

    def divide(xs: list[int]) -> list[list[int]]:
        n = len(xs) // 2
        return [xs[:n], xs[n:]]
```

```
    def combina(_: list[int], xs: list[list[int]]) -> list[int]:
        return mezcla(xs[0], xs[1])

    return divideVenceras(ind, lambda x: x, divide, combina, xs)

# (mezcla xs ys) es la lista obtenida mezclando xs e ys. Por ejemplo,
#    mezcla([1,3], [2,4,6]) == [1,2,3,4,6]
def mezcla(a: list[int], b: list[int]) -> list[int]:
    if not a:
        return b
    if not b:
        return a
    if a[0] <= b[0]:
        return [a[0]] + mezcla(a[1:], b)
    return [b[0]] + mezcla(a, b[1:])

def ordenaRapida(xs: list[int]) -> list[int]:
    def ind(xs: list[int]) -> bool:
        return len(xs) <= 1

    def divide(xs: list[int]) -> list[list[int]]:
        x, *xs = xs
        return [[y for y in xs if y <= x],
                [y for y in xs if y > x]]

    def combina(xs: list[int], ys: list[list[int]]) -> list[int]:
        x = xs[0]
        return ys[0] + [x] + ys[1]

    return divideVenceras(ind, lambda x: x, divide, combina, xs)

# Verificación
# ============

def test_divideVenceras() -> None:
    assert ordenaPorMezcla([3,1,4,1,5,9,2,8]) == [1,1,2,3,4,5,8,9]
    assert ordenaRapida([3,1,4,1,5,9,2,8]) == [1,1,2,3,4,5,8,9]
    print("Verificado")

# La verificación es
```

```
#    >>> test_divideVenceras()
#    Verificado
```

12.2. Rompecabeza del triominó mediante divide y vencerás

12.2.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Un poliominó es una figura geométrica plana formada conectando dos o
-- más cuadrados por alguno de sus lados. Los cuadrados se conectan lado
-- con lado, pero no se pueden conectar ni por sus vértices, ni juntando
-- solo parte de un lado de un cuadrado con parte de un lado de otro. Si
-- unimos dos cuadrados se obtiene un dominó, si se juntan tres
-- cuadrados se construye un triominó.
--
-- Sólo existen dos triominós, el I-triomino (por tener forma de I) y el
-- L-triominó (por su forma de L) como se observa en las siguientes
-- figuras

--    X
---   X     X
--    X     XX
--
-- El rompecabeza del triominó consiste en cubrir un tablero cuadrado
-- con 2^n filas y 2^n columnas, en el que se ha eliminado una casilla,
-- con L-triominós de formas que cubran todas las casillas excepto la
-- eliminada y los triominós no se solapen.
--
-- La casilla eliminada se representará con -1 y los L-triominós con
-- sucesiones de tres números consecutivos en forma de L. Con esta
-- representación una solución del rompecabeza del triominó con 4 filas
-- y la fila eliminada en la posición (4,4) es
--    (  3  3  2  2 )
--    (  3  1  1  2 )
--    (  4  1  5  5 )
--    (  4  4  5 -1 )
--
-- Definir la función
```

```
--    triomino :: Int -> Posicion -> Tablero
-- tal que (triomino n p) es la solución, mediante divide y vencerás,
-- del rompecabeza del triominó en un cuadrado nxn en el que se ha
-- eliminado la casilla de la posición p. Por ejemplo,
--    λ> triomino 4 (4,4)
--    (  3  3  2  2 )
--    (  3  1  1  2 )
--    (  4  1  5  5 )
--    (  4  4  5 -1 )
--
--    λ> triomino 4 (2,3)
--    (  3  3  2  2 )
--    (  3  1 -1  2 )
--    (  4  1  1  5 )
--    (  4  4  5  5 )
--
--    λ> triomino 16 (5,6)
--    (  7  7  6  6  6  6  5  5  6  6  5  5  5  5  4  4 )
--    (  7  5  5  6  6  4  4  5  6  4  4  5  5  3  3  4 )
--    (  8  5  9  9  7  7  4  8  7  4  8  8  6  6  3  7 )
--    (  8  8  9  3  3  7  8  8  7  7  8  2  2  6  7  7 )
--    (  8  8  7  3  9 -1  8  8  7  7  6  6  2  8  7  7 )
--    (  8  6  7  7  9  9  7  8  7  5  5  6  8  8  6  7 )
--    (  9  6  6 10 10  7  7 11  8  8  5  9  9  6  6 10 )
--    (  9  9 10 10 10 10 11 11  1  8  9  9  9  9 10 10 )
--    (  8  8  7  7  7  7  6  1  1  9  8  8  8  8  7  7 )
--    (  8  6  6  7  7  5  6  6  9  9  7  8  8  6  6  7 )
--    (  9  6 10 10  8  5  5  9 10  7  7 11  9  9  6 10 )
--    (  9  9 10  4  8  8  9  9 10 10 11 11  5  9 10 10 )
--    (  9  9  8  4  4 10  9  9 10 10  9  5  5 11 10 10 )
--    (  9  7  8  8 10 10  8  9 10  8  9  9 11 11  9 10 )
--    ( 10  7  7 11 11  8  8 12 11  8  8 12 12  9  9 13 )
--    ( 10 10 11 11 11 11 12 12 11 11 12 12 12 12 13 13 )
-- ---------------------------------------------------------------------

module Rompecabeza_del_triomino_mediante_divide_y_venceras where

import DivideVenceras (divideVenceras)
import Data.Matrix
import Data.List (delete)
```

```haskell
import Test.Hspec (Spec, hspec, it, shouldBe)

-- Los tableros son matrices de números enteros donde -1 representa el
-- hueco, 0 las posiciones sin rellenar y los números mayores que 0
-- representan los triominós.
type Tablero = Matrix Int

-- Los problemas se representarán mediante pares formados por un número
-- natural mayor que 0 (que indica el número con el que se formará el
-- siguiente triominó que se coloque) y un tablero.
type Problema = (Int,Tablero)

-- Las posiciones son pares de números enteros
type Posicion = (Int,Int)

triomino :: Int -> Posicion -> Tablero
triomino n p =
  divideVenceras ind resuelve divide combina (pbInicial n p)

-- (tablero n p) es el tablero inicial del problema del triominó
-- en un cuadrado nxn en el que se ha eliminado la casilla de la
-- posición (i,j). Por ejemplo,
--    λ> tablero 4 (3,4)
--    (  0  0  0  0 )
--    (  0  0  0  0 )
--    (  0  0  0 -1 )
--    (  0  0  0  0 )
tablero :: Int -> Posicion -> Tablero
tablero n (i,j) =
  setElem (-1) (i,j) (zero n n)

-- (pbInicial n p) es el problema inicial del rompecabeza del triominó
-- en un cuadrado nxn en el que se ha eliminado la casilla de la
-- posición p. Por ejemplo,
--    λ> pbInicial 4 (4,4)
--    (1,(  0  0  0  0 )
--       (  0  0  0  0 )
--       (  0  0  0  0 )
--       (  0  0  0 -1 ))
pbInicial :: Int -> Posicion -> Problema
```

```
pbInicial n p = (1,tablero n p)

-- (ind pb) se verifica si el problema pb es indivisible. Por ejemplo,
--    ind (pbInicial 2 (1,2))  ==  True
--    ind (pbInicial 4 (1,2))  ==  False
ind :: Problema -> Bool
ind (_,p) = ncols p == 2

-- (posicionHueco t) es la posición del hueco en el tablero t. Por
-- ejemplo,
--    posicionHueco (tablero 8 (5,2))  ==  (5,2)
posicionHueco :: Tablero -> Posicion
posicionHueco p =
  head [(i,j) | i <- [1..nrows p],
                j <- [1..ncols p],
                p!(i,j) /= 0]

-- (cuadranteHueco p) es el cuadrante donde se encuentra el hueco del
-- tablero t (donde la numeración de los cuadrantes es 1 el superior
-- izquierdo, 2 el inferior izquierdo, 3 el superior derecho y 4 el
-- inferior derecho). Por ejemplo,
--    cuadranteHueco (tablero 8 (4,4))  ==  1
--    cuadranteHueco (tablero 8 (5,2))  ==  2
--    cuadranteHueco (tablero 8 (3,6))  ==  3
--    cuadranteHueco (tablero 8 (6,6))  ==  4
cuadranteHueco :: Tablero -> Int
cuadranteHueco t
  | i <= x && j <= x = 1
  | i >  x && j <= x = 2
  | i <= x && j >  x = 3
  | otherwise        = 4
  where (i,j) = posicionHueco t
        x     = nrows t `div` 2

-- (centralHueco t) es la casilla central del cuadrante del tablero t
-- donde se encuentra el hueco. Por ejemplo,
--    centralHueco (tablero 8 (5,2))  ==  (5,4)
--    centralHueco (tablero 8 (4,4))  ==  (4,4)
--    centralHueco (tablero 8 (3,6))  ==  (4,5)
--    centralHueco (tablero 8 (6,6))  ==  (5,5)
```

```haskell
centralHueco :: Tablero -> Posicion
centralHueco t =
  case cuadranteHueco t of
    1 -> (x,x)
    2 -> (x+1,x)
    3 -> (x,x+1)
    _ -> (x+1,x+1)
  where x = nrows t `div` 2

-- (centralesSinHueco t) son las posiciones centrales del tablero t de
-- los cuadrantes sin hueco. Por ejemplo,
--    centralesSinHueco (tablero 8 (5,2))  ==  [(4,4),(4,5),(5,5)]
centralesSinHueco :: Tablero -> [Posicion]
centralesSinHueco t =
  delete (i,j) [(x,x),(x+1,x),(x,x+1),(x+1,x+1)]
  where x     = nrows t `div` 2
        (i,j) = centralHueco t

-- (actualiza t ps) es la matriz obtenida cambiando en t los valores del
-- las posiciones indicadas en ps por sus correspondientes valores. Por
-- ejemplo,
--    λ> actualiza (identity 3) [((1,2),4),((3,1),5)]
--    ( 1 4 0 )
--    ( 0 1 0 )
--    ( 5 0 1 )
actualiza :: Matrix a -> [((Int,Int),a)] -> Matrix a
actualiza p []             = p
actualiza p (((i,j),x):zs) = setElem x (i,j) (actualiza p zs)

-- (triominoCentral (n,t) es el tablero obtenido colocando el triominó
-- formado por el número n en las posiciones centrales de los 3
-- cuadrantes que no contienen el hueco. Por ejemplo,
--    λ> triominoCentral (7,tablero 4 (4,4))
--    (  0  0  0  0 )
--    (  0  7  7  0 )
--    (  0  7  0  0 )
--    (  0  0  0 -1 )
triominoCentral :: Problema -> Tablero
triominoCentral (n,t) =
  actualiza t [((i,j),n) | (i,j) <- centralesSinHueco t]
```

```
-- (resuelve p) es la solución del problema indivisible p. Por ejemplo,
--    λ> tablero 2 (2,2)
--    (  0  0 )
--    (  0 -1 )
--
--    λ> resuelve (5,tablero 2 (2,2))
--    (  5  5 )
--    (  5 -1 )
resuelve :: Problema -> Tablero
resuelve = triominoCentral

-- (divide (n,t)) es la lista de de los problemas obtenidos colocando el
-- triominó n en las casillas centrales de t que no contienen el hueco y
-- dividir el tablero en sus cuatros cuadrantes y aumentar en uno el
-- número del correspondiente triominó. Por ejemplo,
--    λ> divide (3,tablero 4 (4,4))
--    [(4,(  0  0 )
--        (  3  0 )),
--     (5,(  0  0 )
--        (  0  3 )),
--     (6,(  0  3 )
--        (  0  0 )),
--     (7,(  0  0 )
--        (  0 -1 ))]
divide :: Problema -> [Problema]
divide (n,t) =
  [(n+1, submatrix 1     x (x+1) m q),
   (n+2, submatrix 1     x 1     x q),
   (n+3, submatrix (x+1) m 1     x q),
   (n+4, submatrix (x+1) m (x+1) m q)]
  where q = triominoCentral (n,t)
        m = nrows t
        x = m `div` 2

-- (combina p ts) es la combinación de las soluciones ts de los
-- subproblemas del problema p. Por ejemplo,
--    λ> let inicial = (1,tablero 4 (4,4)) :: (Int,Matrix Int)
--    λ> let [p1,p2,p3,p4] = divide inicial
--    λ> let [s1,s2,s3,s4] = map resuelve [p1,p2,p3,p4]
```

```haskell
--    λ> combina inicial [s1,s2,s3,s4]
--    (  3  3  2  2 )
--    (  3  1  1  2 )
--    (  4  1  5  5 )
--    (  4  4  5 -1 )
combina :: Problema -> [Tablero] -> Tablero
combina _ [s1,s2,s3,s4] = joinBlocks (s2,s1,s3,s4)
combina _ _             = error "Imposible"

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

spec :: Spec
spec = do
  it "e1" $
    toLists (triomino 4 (4,4)) `shouldBe`
    [[3,3,2,2],
     [3,1,1,2],
     [4,1,5,5],
     [4,4,5,-1]]
  it "e2" $
    toLists (triomino 4 (2,3)) `shouldBe`
    [[3,3,2,2],
     [3,1,-1,2],
     [4,1,1,5],
     [4,4,5,5]]
  it "e3" $
    toLists (triomino 16 (5,6)) `shouldBe`
    [[7,7,6,6,6,6,5,5,6,6,5,5,5,5,4,4],
     [7,5,5,6,6,4,4,5,6,4,4,5,5,3,3,4],
     [8,5,9,9,7,7,4,8,7,4,8,8,6,6,3,7],
     [8,8,9,3,3,7,8,8,7,7,8,2,2,6,7,7],
     [8,8,7,3,9,-1,8,8,7,7,6,6,2,8,7,7],
     [8,6,7,7,9,9,7,8,7,5,5,6,8,8,6,7],
     [9,6,6,10,10,7,7,11,8,8,5,9,9,6,6,10],
     [9,9,10,10,10,10,11,11,1,8,9,9,9,9,10,10],
     [8,8,7,7,7,7,6,1,1,9,8,8,8,8,7,7],
```

```
     [8,6,6,7,7,5,6,6,9,9,7,8,8,6,6,7],
     [9,6,10,10,8,5,5,9,10,7,7,11,9,9,6,10],
     [9,9,10,4,8,8,9,9,10,10,11,11,5,9,10,10],
     [9,9,8,4,4,10,9,9,10,10,9,5,5,11,10,10],
     [9,7,8,8,10,10,8,9,10,8,9,9,11,11,9,10],
     [10,7,7,11,11,8,8,12,11,8,8,12,12,9,9,13],
     [10,10,11,11,11,11,12,12,11,11,12,12,12,12,13,13]]

-- La verificación es
--    λ> verifica
--
--    e1
--    e2
--    e3
--
--    Finished in 0.0018 seconds
--    3 examples, 0 failures
```

12.2.2. En Python

```
# ---------------------------------------------------------------------
# Un poliominó es una figura geométrica plana formada conectando dos o
# más cuadrados por alguno de sus lados. Los cuadrados se conectan lado
# con lado, pero no se pueden conectar ni por sus vértices, ni juntando
# solo parte de un lado de un cuadrado con parte de un lado de otro. Si
# unimos dos cuadrados se obtiene un dominó, si se juntan tres
# cuadrados se construye un triominó.
#
# Sólo existen dos triominós, el I-triomino (por tener forma de I) y el
# L-triominó (por su forma de L) como se observa en las siguientes
# figuras
#
#    X
#-   X     X
#    X     XX
#
# El rompecabeza del triominó consiste en cubrir un tablero cuadrado
# con 2^n filas y 2^n columnas, en el que se ha eliminado una casilla,
# con L-triominós de formas que cubran todas las casillas excepto la
# eliminada y los triominós no se solapen.
```

```
#
# La casilla eliminada se representará con -1 y los L-triominós con
# sucesiones de tres números consecutivos en forma de L. Con esta
# representación una solución del rompecabeza del triominó con 4 filas
# y la fila eliminada en la posición (4,4) es
#    (  3  3  2  2 )
#    (  3  1  1  2 )
#    (  4  1  5  5 )
#    (  4  4  5 -1 )
#
# Definir la función
#    triomino : (int, Posicion) -> Tablero
# tal que triomino(n, p) es la solución, mediante divide y vencerás,
# del rompecabeza del triominó en un cuadrado nxn en el que se ha
# eliminado la casilla de la posición p. Por ejemplo,
#    >>> triomino(4, (4,4))
#    array([[ 3,  3,  2,  2],
#           [ 3,  1,  1,  2],
#           [ 4,  1,  5,  5],
#           [ 4,  4,  5, -1]])
#    >>> triomino(4, (2,3))
#    array([[ 3,  3,  2,  2],
#           [ 3,  1, -1,  2],
#           [ 4,  1,  1,  5],
#           [ 4,  4,  5,  5]])
#    >>> triomino(16, (5,6))
#    array([[ 7,  7,  6,  6,  6,  6,  5,  5,  6,  6,  5,  5,  5,  5,  4,  4],
#           [ 7,  5,  5,  6,  6,  4,  4,  5,  6,  4,  4,  5,  5,  3,  3,  4],
#           [ 8,  5,  9,  9,  7,  7,  4,  8,  7,  4,  8,  8,  6,  6,  3,  7],
#           [ 8,  8,  9,  3,  3,  7,  8,  8,  7,  7,  8,  2,  2,  6,  7,  7],
#           [ 8,  8,  7,  3,  9, -1,  8,  8,  7,  7,  6,  6,  2,  8,  7,  7],
#           [ 8,  6,  7,  7,  9,  9,  7,  8,  7,  5,  5,  6,  8,  8,  6,  7],
#           [ 9,  6,  6, 10, 10,  7,  7, 11,  8,  8,  5,  9,  9,  6,  6, 10],
#           [ 9,  9, 10, 10, 10, 10, 11, 11,  1,  8,  9,  9,  9,  9, 10, 10],
#           [ 8,  8,  7,  7,  7,  7,  6,  1,  1,  9,  8,  8,  8,  8,  7,  7],
#           [ 8,  6,  6,  7,  7,  5,  6,  6,  9,  9,  7,  8,  8,  6,  6,  7],
#           [ 9,  6, 10, 10,  8,  5,  5,  9, 10,  7,  7, 11,  9,  9,  6, 10],
#           [ 9,  9, 10,  4,  8,  8,  9,  9, 10, 10, 11, 11,  5,  9, 10, 10],
#           [ 9,  9,  8,  4,  4, 10,  9,  9, 10, 10,  9,  5,  5, 11, 10, 10],
#           [ 9,  7,  8,  8, 10, 10,  8,  9, 10,  8,  9,  9, 11, 11,  9, 10],
```

```
#          [10,  7,  7, 11, 11,  8,  8, 12, 11,  8,  8, 12, 12,  9,  9, 13],
#          [10, 10, 11, 11, 11, 11, 12, 12, 11, 11, 12, 12, 12, 12, 13, 13]])
# ---------------------------------------------------------------------

import numpy as np
import numpy.typing as npt

from src.DivideVenceras import divideVenceras

# Los tableros son matrices de números enteros donde -1 representa el
# hueco, 0 las posiciones sin rellenar y los números mayores que 0
# representan los triominós.
Tablero = npt.NDArray[np.int_]

# Los problemas se representarán mediante pares formados por un número
# natural mayor que 0 (que indica el número con el que se formará el
# siguiente triominó que se coloque) y un tablero.
Problema = tuple[int, Tablero]

# Las posiciones son pares de números enteros
Posicion = tuple[int, int]

# tablero(n p) es el tablero inicial del problema del triominó
# en un cuadrado nxn en el que se ha eliminado la casilla de la
# posición p. Por ejemplo,
#    >>> tablero(4, (3,4))
#    array([[ 0,  0,  0,  0],
#           [ 0,  0,  0,  0],
#           [ 0,  0,  0, -1],
#           [ 0,  0,  0,  0]])
def tablero(n: int, p: Posicion) -> Tablero:
    (i, j) = p
    q = np.zeros((n, n), dtype=int)
    q[i - 1, j - 1] = -1
    return q

# pbInicial(n, p) es el problema inicial del rompecabeza del triominó
# en un cuadrado nxn en el que se ha eliminado la casilla de la
# posición p. Por ejemplo,
#    >>> pbInicial(4, (4,4))
```

```
#    (1, array([[ 0,  0,  0,  0],
#               [ 0,  0,  0,  0],
#               [ 0,  0,  0,  0],
#               [ 0,  0,  0, -1]]))
def pbInicial(n: int, p: Posicion) -> Problema:
    return 1, tablero(n, p)

# ind(pb) se verifica si el problema pb es indivisible. Por ejemplo,
#    ind(pbInicial(2, (1,2)))  ==  True
#    ind(pbInicial(4, (1,2)))  ==  False
def ind(pb: Problema) -> bool:
    _, p = pb
    return p.shape[1] == 2

# posicionHueco(t) es la posición del hueco en el tablero t. Por
# ejemplo,
#    posicionHueco(tablero(8, (5,2)))  ==  (5,2)
def posicionHueco(t: Tablero) -> Posicion:
    indices = np.argwhere(t != 0)
    (i, j) =  tuple(indices[0])
    return (i + 1, j + 1)

# cuadranteHueco(p) es el cuadrante donde se encuentra el hueco del
# tablero t (donde la numeración de los cuadrantes es 1 el superior
# izquierdo, 2 el inferior izquierdo, 3 el superior derecho y 4 el
# inferior derecho). Por ejemplo,
#    cuadranteHueco(tablero(8, (4,4)))  ==  1
#    cuadranteHueco(tablero(8, (5,2)))  ==  2
#    cuadranteHueco(tablero(8, (3,6)))  ==  3
#    cuadranteHueco(tablero(8, (6,6)))  ==  4
def cuadranteHueco(t: Tablero) -> int:
    i, j = posicionHueco(t)
    x = t.shape[0] // 2
    if j <= x:
        if i <= x:
            return 1
        return 2
    if i <= x:
        return 3
    return 4
```

```
# centralHueco(t) es la casilla central del cuadrante del tablero t
# donde se encuentra el hueco. Por ejemplo,
#    centralHueco(tablero(8, (5,2)))  ==  (5,4)
#    centralHueco(tablero(8, (4,4)))  ==  (4,4)
#    centralHueco(tablero(8, (3,6)))  ==  (4,5)
#    centralHueco(tablero(8, (6,6)))  ==  (5,5)
def centralHueco(t: Tablero) -> Posicion:
    x = t.shape[0] // 2
    cuadrante = cuadranteHueco(t)
    if cuadrante == 1:
        return (x, x)
    if cuadrante == 2:
        return (x+1, x)
    if cuadrante == 3:
        return (x, x+1)
    return (x+1, x+1)

# centralesSinHueco(t) son las posiciones centrales del tablero t de
# los cuadrantes sin hueco. Por ejemplo,
#    centralesSinHueco(tablero(8, (5,2)))  ==  [(4,4),(4,5),(5,5)]
def centralesSinHueco(t: Tablero) -> list[Posicion]:
    x = t.shape[0] // 2
    i, j = centralHueco(t)
    ps = [(x, x), (x+1, x), (x, x+1), (x+1, x+1)]
    return [p for p in ps if p != (i, j)]

# actualiza(t, ps) es la matriz obtenida cambiando en t los valores del
# las posiciones indicadas en ps por sus correspondientes valores. Por
# ejemplo,
#    >>> actualiza(np.identity(3, dtype=int), [((1,2),4),((3,1),5)])
#    array([[1, 4, 0],
#           [0, 1, 0],
#           [5, 0, 1]])
def actualiza(p: Tablero, ps: list[tuple[Posicion, int]]) -> Tablero:
    for (i, j), x in ps:
        p[i - 1, j - 1] = x
    return p

# triominoCentral(n,t) es el tablero obtenido colocando el triominó
```

```
# formado por el número n en las posiciones centrales de los 3
# cuadrantes que no contienen el hueco. Por ejemplo,
#    >>> triominoCentral((7, tablero(4, (4,4))))
#    array([[ 0,  0,  0,  0],
#           [ 0,  7,  7,  0],
#           [ 0,  7,  0,  0],
#           [ 0,  0,  0, -1]])
def triominoCentral(p: Problema) -> Tablero:
    n, t = p
    return actualiza(t, [((i,j),n) for (i,j) in centralesSinHueco(t)])

# resuelve(p) es la solución del problema indivisible p. Por ejemplo,
#    >>> tablero(2, (2,2))
#    array([[ 0,  0],
#           [ 0, -1]])
#    >>> resuelve((5,tablero(2, (2,2))))
#    array([[ 5,  5],
#           [ 5, -1]])
def resuelve(p: Problema) -> Tablero:
    return triominoCentral(p)

# divide(n,t) es la lista de de los problemas obtenidos colocando el
# triominó n en las casillas centrales de t que no contienen el hueco y
# dividir el tablero en sus cuatros cuadrantes y aumentar en uno el
# número del correspondiente triominó. Por ejemplo,
#    >>> divide((3,tablero(4, (4,4))))
#    [(4, array([[0, 0],
#                [3, 0]])),
#     (5, array([[0, 0],
#                [0, 3]])),
#     (6, array([[0, 3],
#                [0, 0]])),
#     (7, array([[ 0,  0],
#                [ 0, -1]]))]
def divide(p: Problema) -> list[Problema]:
    q = triominoCentral(p)
    n, t = p
    m = t.shape[0]
    x = m // 2
    subproblemas = [
```

```
        (n+1, q[0:x, x:m]),
        (n+2, q[0:x, 0:x]),
        (n+3, q[x:m, 0:x]),
        (n+4, q[x:m, x:m])
    ]
    return subproblemas

# combina(p, ts) es la combinación de las soluciones ts de los
# subproblemas del problema p. Por ejemplo,
#    >>> inicial = (1, tablero(4, (4, 4)))
#    >>> p1, p2, p3, p4 = divide(inicial)
#    >>> s1, s2, s3, s4 = map(resuelve, [p1, p2, p3, p4])
#    >>> combina(inicial, [s1, s2, s3, s4])
#    array([[ 3,  3,  2,  2],
#           [ 3,  1,  1,  2],
#           [ 4,  1,  5,  5],
#           [ 4,  4,  5, -1]])
def combina(_: Problema, ts: list[Tablero]) -> Tablero:
    s1, s2, s3, s4 = ts
    combined = np.block([[s2, s1], [s3, s4]])
    return combined

def triomino(n: int, p: Posicion) -> Tablero:
    return divideVenceras(ind, resuelve, divide, combina, pbInicial(n, p))

# Verificación
# ============

def test_triomino() -> None:
    def filas(p: Tablero) -> list[list[int]]:
        return p.tolist()

    assert filas(triomino(4, (4,4))) == \
        [[3,3,2,2],[3,1,1,2],[4,1,5,5],[4,4,5,-1]]
    assert filas(triomino(4, (2,3))) == \
        [[3,3,2,2],[3,1,-1,2],[4,1,1,5],[4,4,5,5]]
    assert filas(triomino(16, (5,6))) == \
        [[7,7,6,6,6,6,5,5,6,6,5,5,5,5,4,4],
         [7,5,5,6,6,4,4,5,6,4,4,5,5,3,3,4],
         [8,5,9,9,7,7,4,8,7,4,8,8,6,6,3,7],
```

```
          [8,8,9,3,3,7,8,8,7,7,8,2,2,6,7,7],
          [8,8,7,3,9,-1,8,8,7,7,6,6,2,8,7,7],
          [8,6,7,7,9,9,7,8,7,5,5,6,8,8,6,7],
          [9,6,6,10,10,7,7,11,8,8,5,9,9,6,6,10],
          [9,9,10,10,10,10,11,11,1,8,9,9,9,9,10,10],
          [8,8,7,7,7,7,6,1,1,9,8,8,8,8,7,7],
          [8,6,6,7,7,5,6,6,9,9,7,8,8,6,6,7],
          [9,6,10,10,8,5,5,9,10,7,7,11,9,9,6,10],
          [9,9,10,4,8,8,9,9,10,10,11,11,5,9,10,10],
          [9,9,8,4,4,10,9,9,10,10,9,5,5,11,10,10],
          [9,7,8,8,10,10,8,9,10,8,9,9,11,11,9,10],
          [10,7,7,11,11,8,8,12,11,8,8,12,12,9,9,13],
          [10,10,11,11,11,11,12,12,11,11,12,12,12,12,13,13]]
    print("Verificado")

# La verificación es
#    >>> test_triomino()
#    Verificado
```

Capítulo 13

Búsqueda en espacios de estados

Contenido

13.1. Búsqueda en espacios de estados por profundidad

13.1.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Las características de los problemas de espacios de estados son:
-- + un conjunto de las posibles situaciones o nodos que constituye el
--   espacio de estados (estos son las potenciales soluciones que se
--   necesitan explorar),
-- + un conjunto de movimientos de un nodo a otros nodos, llamados los
--   sucesores del nodo,
-- + un nodo inicial y
-- + un nodo objetivo que es la solución.
--
-- Definir la función
--    buscaProfundidad :: Eq nodo => (nodo -> [nodo]) -> (nodo -> Bool)
--                                      -> nodo -> [nodo]
-- tal que (buscaProfundidad s o e) es  la lista de soluciones del
-- problema de espacio de estado definido por la función sucesores s, el
-- objetivo o y estado inicial e obtenidas mediante búsqueda en
-- profundidad.
-- ---------------------------------------------------------------------

module BusquedaEnProfundidad (buscaProfundidad) where

import TAD.Pila (apila, cima, desapila, esVacia, vacia)

buscaProfundidad :: Eq nodo => (nodo -> [nodo]) -> (nodo -> Bool)
                               -> nodo -> [nodo]
buscaProfundidad sucesores esFinal inicial =
```

```
  aux (apila inicial vacia)
  where
    aux p
      | esVacia p         = []
      | esFinal (cima p) = cima p : aux (desapila p)
      | otherwise         = aux (foldr
                                  apila
                                  (desapila p)
                                  (sucesores (cima p)))
```

13.1.2. En Python

```
# ---------------------------------------------------------------------
# Las características de los problemas de espacios de estados son:
# + un conjunto de las posibles situaciones o nodos que constituye el
#   espacio de estados (estos son las potenciales soluciones que se
#   necesitan explorar),
# + un conjunto de movimientos de un nodo a otros nodos, llamados los
#   sucesores del nodo,
# + un nodo inicial y
# + un nodo objetivo que es la solución.
#
# Definir las funciones
#    buscaProfundidad(Callable[[A], list[A]], Callable[[A], bool], A) -> list[A]
#    buscaProfundidad1(Callable[[A], list[A]], Callable[[A], bool], A) -> Optiona
# tales que
# + buscaProfundidad(s, o, e) es la lista de soluciones del
#   problema de espacio de estado definido por la función sucesores s,
#   el objetivo o y estado inicial e obtenidas mediante búsqueda en
#   profundidad.
# + buscaProfundidad1(s, o, e) es la orimera solución del
#   problema de espacio de estado definido por la función sucesores s,
#   el objetivo o y estado inicial e obtenidas mediante búsqueda en
#   profundidad.
# ---------------------------------------------------------------------

from functools import reduce
from sys import setrecursionlimit
from typing import Callable, Optional, TypeVar
```

```
from src.TAD.pila import Pila, apila, cima, desapila, esVacia, vacia

A = TypeVar('A')

setrecursionlimit(10**6)

def buscaProfundidad(sucesores: Callable[[A], list[A]],
                     esFinal: Callable[[A], bool],
                     inicial: A) -> list[A]:
    def aux(p: Pila[A]) -> list[A]:
        if esVacia(p):
            return []
        if esFinal(cima(p)):
            return [cima(p)] + aux(desapila(p))
        return aux(reduce(lambda x, y: apila(y, x),
                          sucesores(cima(p)),
                          desapila(p)))

    return aux(apila(inicial, vacia()))

def buscaProfundidad1(sucesores: Callable[[A], list[A]],
                      esFinal: Callable[[A], bool],
                      inicial: A) -> Optional[A]:
    p: Pila[A] = apila(inicial, vacia())

    while not esVacia(p):
        cp = cima(p)
        if esFinal(cp):
            return cp

        es = sucesores(cp)
        p = reduce(lambda x, y: apila(y, x), es, desapila(p))

    return None
```

13.2. El problema de las n reinas (mediante búsqueda por profundidad en espacios de estados)

13.2.1. En Haskell

```
-- ---------------------------------------------------------------------
-- El problema de las n reinas consiste en colocar n reinas en un
-- tablero cuadrado de dimensiones n por n de forma que no se encuentren
-- más de una en la misma línea: horizontal, vertical o diagonal.
--
-- Las posiciones de las reinas en el tablero se representan por su
-- columna y su fila.
--    type Columna = Int
--    type Fila    = Int
--
-- Una solución del problema de las n reinas es una lista de
-- posiciones.
--    type SolNR = [(Columna,Fila)]
--
-- Usando el procedimiento de búsqueda en profundidad, definir las
-- funciones
--    solucionesNR      :: Columna -> [SolNR]
--    primeraSolucionNR :: Columna -> SolNR
--    nSolucionesNR     :: Columna -> Int
-- tales que
-- + (solucionesNR n) es la lista de las soluciones del problema de las n
--   reinas, por búsqueda de espacio de estados en profundidad. Por
--   ejemplo,
--      take 3 (solucionesNR 8)
--      [[(1,1),(2,5),(3,8),(4,6),(5,3),(6,7),(7,2),(8,4)],
--       [(1,1),(2,6),(3,8),(4,3),(5,7),(6,4),(7,2),(8,5)],
--       [(1,1),(2,7),(3,4),(4,6),(5,8),(6,2),(7,5),(8,3)]]
-- + (primeraSolucionNR n) es la primera solución del problema de las n
--   reinas, por búsqueda en espacio de estados por profundidad. Por
--   ejemplo,
--      λ> primeraSolucionNR 8
--      [(1,1),(2,5),(3,8),(4,6),(5,3),(6,7),(7,2),(8,4)]
-- + (nSolucionesNR n) es el número de soluciones del problema de las n
```

```
--   reinas, por búsqueda en espacio de estados. Por ejemplo,
--      nSolucionesNR 8  ==  92
-- ---------------------------------------------------------------------

module BEE_Reinas_Profundidad where

import BusquedaEnProfundidad (buscaProfundidad)
import Test.Hspec (Spec, hspec, it, shouldBe)

type Columna = Int
type Fila    = Int
type SolNR = [(Columna,Fila)]

-- Los nodos del problema de las n reinas son ternas formadas por la
-- columna de la última reina colocada, el número de columnas del
-- tablero y la solución parcial de las reinas colocadas anteriormente.
type NodoNR = (Columna,Columna,SolNR)

solucionesNR :: Columna -> [SolNR]
solucionesNR n =
  map estado (buscaProfundidad sucesoresNR esFinalNR (1,n,[]))
  where
    estado (_,_,e) = e

primeraSolucionNR :: Columna -> SolNR
primeraSolucionNR =
  head . solucionesNR

nSolucionesNR :: Columna -> Int
nSolucionesNR =
  length . solucionesNR

-- (valida sp p) se verifica si la posición p es válida respecto de la
-- solución parcial sp; es decir, la reina en la posición p no amenaza a
-- ninguna de las reinas de la sp (se supone que están en distintas
-- columnas). Por ejemplo,
--    valida [(1,1)] (2,2)  ==  False
--    valida [(1,1)] (2,3)  ==  True
valida :: SolNR -> (Columna,Fila) -> Bool
valida solp (c,r) = and [test s | s <- solp]
```

```
  where test (c',r') = c'+r'/=c+r && c'-r'/=c-r && r'/=r

-- (sucesoresNR e) es la lista de los sucesores del estado e en el
-- problema de las n reinas. Por ejemplo,
--    λ> sucesoresNR (1,4,[])
--    [(2,4,[(1,1)]),(2,4,[(1,2)]),(2,4,[(1,3)]),(2,4,[(1,4)])]
sucesoresNR :: NodoNR -> [NodoNR]
sucesoresNR (c,n,solp) =
  [(c+1,n,solp ++ [(c,r)]) | r <- [1..n] , valida solp (c,r)]

-- (esFinalNR e) se verifica si e es un estado final del problema de las
-- n reinas.
esFinalNR :: NodoNR -> Bool
esFinalNR (c,n,_) = c > n

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

spec :: Spec
spec = do
  it "e1" $
    take 3 (solucionesNR 8) `shouldBe`
    [[(1,1),(2,5),(3,8),(4,6),(5,3),(6,7),(7,2),(8,4)],
     [(1,1),(2,6),(3,8),(4,3),(5,7),(6,4),(7,2),(8,5)],
     [(1,1),(2,7),(3,4),(4,6),(5,8),(6,2),(7,5),(8,3)]]
  it "e2" $
    nSolucionesNR 8 `shouldBe` 92

-- La verificación es
--    λ> verifica
--
--    e1
--    e2
--
--    Finished in 0.1173 seconds
--    2 examples, 0 failures
```

13.2.2. En Python

```python
# ---------------------------------------------------------------------
# El problema de las n reinas consiste en colocar n reinas en un
# tablero cuadrado de dimensiones n por n de forma que no se encuentren
# más de una en la misma línea: horizontal, vertical o diagonal.
#
# Las posiciones de las reinas en el tablero se representan por su
# columna y su fila.
#    Columna = int
#    Fila    = int
#
# Una solución del problema de las n reinas es una lista de
# posiciones.
#    SolNR = list[tuple[Columna, Fila]]
#
# Usando el procedimiento de búsqueda en profundidad, definir las
# funciones
#    solucionesNR      : (int) -> list[SolNR]
#    primeraSolucionNR : (int) -> SolNR
#    nSolucionesNR     : (int) -> int
# tales que
# + solucionesNR(n) es la lista de las soluciones del problema de las n
#   reinas, por búsqueda de espacio de estados en profundidad. Por
#   ejemplo,
#      >>> solucionesNR(8)[:3]
#      [[(1, 8), (2, 4), (3, 1), (4, 3), (5, 6), (6, 2), (7, 7), (8, 5)],
#       [(1, 8), (2, 3), (3, 1), (4, 6), (5, 2), (6, 5), (7, 7), (8, 4)],
#       [(1, 8), (2, 2), (3, 5), (4, 3), (5, 1), (6, 7), (7, 4), (8, 6)]]
# + primeraSolucionNR(n) es la primera solución del problema de las n
#   reinas, por búsqueda en espacio de estados por profundidad. Por
#   ejemplo,
#      >>> primeraSolucionNR(8)
#      [(1, 8), (2, 4), (3, 1), (4, 3), (5, 6), (6, 2), (7, 7), (8, 5)]
# + nSolucionesNR(n) es el número de soluciones del problema de las n
#   reinas, por búsqueda en espacio de estados. Por ejemplo,
#      >>> nSolucionesNR(8)
#      92
# ---------------------------------------------------------------------

from src.BusquedaEnProfundidad import buscaProfundidad
```

```
Columna = int
Fila = int
SolNR = list[tuple[Columna, Fila]]

# Los nodos del problema de las n reinas son ternas formadas por la
# columna de la última reina colocada, el número de columnas del
# tablero y la solución parcial de las reinas colocadas anteriormente.
NodoNR = tuple[Columna, Columna, SolNR]

# valida(sp, p) se verifica si la posición p es válida respecto de la
# solución parcial sp; es decir, la reina en la posición p no amenaza a
# ninguna de las reinas de la sp (se supone que están en distintas
# columnas). Por ejemplo,
#    valida([(1,1)], (2,2))  ==  False
#    valida([(1,1)], (2,3))  ==  True
def valida(sp: SolNR, p: tuple[Columna, Fila]) -> bool:
    c, r = p
    def test(s: tuple[Columna, Fila]) -> bool:
        c1, r1 = s
        return c1 + r1 != c + r and c1 - r1 != c - r and r1 != r

    return all(test(s) for s in sp)

# sucesoresNR(e) es la lista de los sucesores del estado e en el
# problema de las n reinas. Por ejemplo,
#    >>> sucesoresNR((1,4,[]))
#    [(2,4,[(1,1)]),(2,4,[(1,2)]),(2,4,[(1,3)]),(2,4,[(1,4)])]
def sucesoresNR (nd: NodoNR) -> list[NodoNR]:
    c,n,solp = nd
    return [(c+1,n,solp + [(c,r)]) for r in range(1, n+1) if valida(solp, (c,r))]

# esFinalNR(e) se verifica si e es un estado final del problema de las
# n reinas.
def esFinalNR(nd: NodoNR) -> bool:
    c, n, _ = nd
    return c > n

def solucionesNR(n: int) -> list[SolNR]:
    nInicial: NodoNR = (1,n,[])
```

```
    return [e for (_, _, e) in buscaProfundidad(sucesoresNR,
                                                esFinalNR,
                                                nInicial)]

def primeraSolucionNR(n: int) -> SolNR:
    return solucionesNR(n)[0]

def nSolucionesNR(n: int) -> int:
    return len(solucionesNR(n))

# Verificación
# ============

def test_nReinas() -> None:
    assert solucionesNR(8)[:3] == \
        [[(1,8),(2,4),(3,1),(4,3),(5,6),(6,2),(7,7),(8,5)],
         [(1,8),(2,3),(3,1),(4,6),(5,2),(6,5),(7,7),(8,4)],
         [(1,8),(2,2),(3,5),(4,3),(5,1),(6,7),(7,4),(8,6)]]
    assert nSolucionesNR(8) == 92
    print("Verificado")

# La verificación es
#
```

13.3. Búsqueda en espacios de estados por anchura

13.3.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Las características de los problemas de espacios de estados son:
-- + un conjunto de las posibles situaciones o nodos que constituye el
--   espacio de estados (estos son las potenciales soluciones que se
--   necesitan explorar),
-- + un conjunto de movimientos de un nodo a otros nodos, llamados los
--   sucesores del nodo,
-- + un nodo inicial y
-- + un nodo objetivo que es la solución.
--
```

```haskell
-- Definir la función
--    buscaAnchura :: Eq nodo => (nodo -> [nodo]) -> (nodo -> Bool)
--                                 -> nodo -> [nodo]
-- tal que (buscaAnchura s o e) es  la lista de soluciones del
-- problema de espacio de estado definido por la función sucesores s, el
-- objetivo o y estado inicial e obtenidas mediante búsqueda en
-- anchura.
-- ---------------------------------------------------------------------

module BusquedaEnAnchura (buscaAnchura) where

import TAD.Cola (esVacia, inserta, primero, resto, vacia)

buscaAnchura :: Eq nodo => (nodo -> [nodo]) -> (nodo -> Bool)
                               -> nodo -> [nodo]
buscaAnchura sucesores esFinal inicial =
  aux (inserta inicial vacia)
  where
    aux p
      | esVacia p        = []
      | esFinal (primero p) = primero p : aux (resto p)
      | otherwise        = aux (foldr
                                inserta
                                (resto p)
                                (sucesores (primero p)))
```

13.3.2. En Python

```python
# ---------------------------------------------------------------------
# Las características de los problemas de espacios de estados son:
# + un conjunto de las posibles situaciones o nodos que constituye el
#   espacio de estados (estos son las potenciales soluciones que se
#   necesitan explorar),
# + un conjunto de movimientos de un nodo a otros nodos, llamados los
#   sucesores del nodo,
# + un nodo inicial y
# + un nodo objetivo que es la solución.
#
# Definir las funciones
#    buscaAnchura(Callable[[A], list[A]], Callable[[A], bool], A) -> list[A]
```

```
#    buscaAnchura1(Callable[[A], list[A]], Callable[[A], bool], A) -> Optional[A]
# tales que
# + buscaAnchura(s, o, e) es la lista de soluciones del
#   problema de espacio de estado definido por la función sucesores s,
#   el objetivo o y estado inicial e obtenidas mediante búsqueda en
#   anchura.
# + buscaAnchura1(s, o, e) es la orimera solución del
#   problema de espacio de estado definido por la función sucesores s,
#   el objetivo o y estado inicial e obtenidas mediante búsqueda en
#   anchura.
# ---------------------------------------------------------------------

from functools import reduce
from sys import setrecursionlimit
from typing import Callable, Optional, TypeVar

from src.TAD.cola import Cola, esVacia, inserta, primero, resto, vacia

A = TypeVar('A')

setrecursionlimit(10**6)

def buscaAnchura(sucesores: Callable[[A], list[A]],
                 esFinal: Callable[[A], bool],
                 inicial: A) -> list[A]:
    def aux(p: Cola[A]) -> list[A]:
        if esVacia(p):
            return []
        if esFinal(primero(p)):
            return [primero(p)] + aux(resto(p))
        return aux(reduce(lambda x, y: inserta(y, x),
                          sucesores(primero(p)),
                          resto(p)))

    return aux(inserta(inicial, vacia()))

def buscaAnchura1(sucesores: Callable[[A], list[A]],
                  esFinal: Callable[[A], bool],
                  inicial: A) -> Optional[A]:
    c: Cola[A] = inserta(inicial, vacia())
```

```python
    while not esVacia(c):
        pc = primero(c)
        if esFinal(pc):
            return pc

        es = sucesores(pc)
        c = reduce(lambda x, y: inserta(y, x), es, resto(c))

    return None
```

13.4. El problema de las n reinas (mediante búsqueda en espacios de estados por anchura)

13.4.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- El problema de las n reinas consiste en colocar n reinas en un
-- tablero cuadrado de dimensiones n por n de forma que no se encuentren
-- más de una en la misma línea: horizontal, vertical o diagonal.
--
-- Las posiciones de las reinas en el tablero se representan por su
-- columna y su fila.
--    type Columna = Int
--    type Fila    = Int
--
-- Una solución del problema de las n reinas es una lista de
-- posiciones.
--    type SolNR = [(Columna,Fila)]
--
-- Usando el procedimiento de búsqueda en anchura, definir las
-- funciones
--    solucionesNR      :: Columna -> [SolNR]
--    primeraSolucionNR :: Columna -> SolNR
--    nSolucionesNR     :: Columna -> Int
-- tales que
-- + (solucionesNR n) es la lista de las soluciones del problema de las n
--   reinas, por búsqueda de espacio de estados en anchura. Por
```

```haskell
--   ejemplo,
--      take 3 (solucionesNR 8)
--      [[(1,8),(2,4),(3,1),(4,3),(5,6),(6,2),(7,7),(8,5)],
--       [(1,8),(2,3),(3,1),(4,6),(5,2),(6,5),(7,7),(8,4)],
--       [(1,8),(2,2),(3,5),(4,3),(5,1),(6,7),(7,4),(8,6)]]
-- + (primeraSolucionNR n) es la primera solución del problema de las n
--   reinas, por búsqueda en espacio de estados por anchura. Por
--   ejemplo,
--      λ> primeraSolucionNR 8
--      [(1,8),(2,4),(3,1),(4,3),(5,6),(6,2),(7,7),(8,5)]
-- + (nSolucionesNR n) es el número de soluciones del problema de las n
--   reinas, por búsqueda en espacio de estados. Por ejemplo,
--      nSolucionesNR 8  ==  92
-- ---------------------------------------------------------------------

module BEE_Reinas_Anchura where

import BusquedaEnAnchura (buscaAnchura)
import Test.Hspec (Spec, hspec, it, shouldBe)

type Columna = Int
type Fila    = Int
type SolNR = [(Columna,Fila)]

-- Los nodos del problema de las n reinas son ternas formadas por la
-- columna de la última reina colocada, el número de columnas del
-- tablero y la solución parcial de las reinas colocadas anteriormente.
type NodoNR = (Columna,Columna,SolNR)

solucionesNR :: Columna -> [SolNR]
solucionesNR n =
  map estado (buscaAnchura sucesoresNR esFinalNR (1,n,[]))
  where
    estado (_,_,e) = e

primeraSolucionNR :: Columna -> SolNR
primeraSolucionNR =
  head . solucionesNR

nSolucionesNR :: Columna -> Int
```

```
nSolucionesNR =
  length . solucionesNR

-- (valida sp p) se verifica si la posición p es válida respecto de la
-- solución parcial sp; es decir, la reina en la posición p no amenaza a
-- ninguna de las reinas de la sp (se supone que están en distintas
-- columnas). Por ejemplo,
--    valida [(1,1)] (2,2)  ==  False
--    valida [(1,1)] (2,3)  ==  True
valida :: SolNR -> (Columna,Fila) -> Bool
valida solp (c,r) = and [test s | s <- solp]
  where test (c',r') = c'+r'/=c+r && c'-r'/=c-r && r'/=r

-- (sucesoresNR e) es la lista de los sucesores del estado e en el
-- problema de las n reinas. Por ejemplo,
--    λ> sucesoresNR (1,4,[])
--    [(2,4,[(1,1)]),(2,4,[(1,2)]),(2,4,[(1,3)]),(2,4,[(1,4)])]
sucesoresNR :: NodoNR -> [NodoNR]
sucesoresNR (c,n,solp) =
  [(c+1,n,solp ++ [(c,r)]) | r <- [1..n] , valida solp (c,r)]

-- (esFinalNR e) se verifica si e es un estado final del problema de las
-- n reinas.
esFinalNR :: NodoNR -> Bool
esFinalNR (c,n,_) = c > n

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

spec :: Spec
spec = do
  it "e1" $
    take 3 (solucionesNR 8) `shouldBe`
    [[(1,8),(2,4),(3,1),(4,3),(5,6),(6,2),(7,7),(8,5)],
     [(1,8),(2,3),(3,1),(4,6),(5,2),(6,5),(7,7),(8,4)],
     [(1,8),(2,2),(3,5),(4,3),(5,1),(6,7),(7,4),(8,6)]]
  it "e2" $
```

```
    nSolucionesNR 8 `shouldBe` 92

-- La verificación es
--    λ> verifica
--
--    e1
--    e2
--
--    Finished in 0.2116 seconds
--    2 examples, 0 failures
```

13.4.2. En Python

```
# ---------------------------------------------------------------------
# El problema de las n reinas consiste en colocar n reinas en un
# tablero cuadrado de dimensiones n por n de forma que no se encuentren
# más de una en la misma línea: horizontal, vertical o diagonal.
#
# Las posiciones de las reinas en el tablero se representan por su
# columna y su fila.
#    Columna = int
#    Fila    = int
#
# Una solución del problema de las n reinas es una lista de
# posiciones.
#    SolNR = list[tuple[Columna, Fila]]
#
# Usando el procedimiento de búsqueda en profundidad, definir las
# funciones
#    solucionesNR      : (int) -> list[SolNR]
#    primeraSolucionNR : (int) -> SolNR
#    nSolucionesNR     : (int) -> int
# tales que
# + solucionesNR(n) es la lista de las soluciones del problema de las n
#   reinas, por búsqueda de espacio de estados en profundidad. Por
#   ejemplo,
#      >>> solucionesNR(8)[:3]
#      [[(1,1),(2,5),(3,8),(4,6),(5,3),(6,7),(7,2),(8,4)],
#       [(1,1),(2,6),(3,8),(4,3),(5,7),(6,4),(7,2),(8,5)],
#       [(1,1),(2,7),(3,4),(4,6),(5,8),(6,2),(7,5),(8,3)]]
```

```
# + primeraSolucionNR(n) es la primera solución del problema de las n
#   reinas, por búsqueda en espacio de estados por profundidad. Por
#   ejemplo,
#      >>> primeraSolucionNR(8)
#      [(1,1),(2,5),(3,8),(4,6),(5,3),(6,7),(7,2),(8,4)]
# + nSolucionesNR(n) es el número de soluciones del problema de las n
#   reinas, por búsqueda en espacio de estados. Por ejemplo,
#      >>> nSolucionesNR(8)
#      92
# ---------------------------------------------------------------------

from src.BusquedaEnAnchura import buscaAnchura

Columna = int
Fila = int
SolNR = list[tuple[Columna, Fila]]

# Los nodos del problema de las n reinas son ternas formadas por la
# columna de la última reina colocada, el número de columnas del
# tablero y la solución parcial de las reinas colocadas anteriormente.
NodoNR = tuple[Columna, Columna, SolNR]

# valida(sp, p) se verifica si la posición p es válida respecto de la
# solución parcial sp; es decir, la reina en la posición p no amenaza a
# ninguna de las reinas de la sp (se supone que están en distintas
# columnas). Por ejemplo,
#    valida([(1,1)], (2,2))  ==  False
#    valida([(1,1)], (2,3))  ==  True
def valida(sp: SolNR, p: tuple[Columna, Fila]) -> bool:
    c, r = p
    def test(s: tuple[Columna, Fila]) -> bool:
        c1, r1 = s
        return c1 + r1 != c + r and c1 - r1 != c - r and r1 != r

    return all(test(s) for s in sp)

# sucesoresNR(e) es la lista de los sucesores del estado e en el
# problema de las n reinas. Por ejemplo,
#    >>> sucesoresNR((1,4,[]))
#    [(2,4,[(1,1)]),(2,4,[(1,2)]),(2,4,[(1,3)]),(2,4,[(1,4)])]
```

```python
def sucesoresNR (nd: NodoNR) -> list[NodoNR]:
    c,n,solp = nd
    return [(c+1,n,solp + [(c,r)]) for r in range(1, n+1) if valida(solp, (c,r))]

# esFinalNR(e) se verifica si e es un estado final del problema de las
# n reinas.
def esFinalNR(nd: NodoNR) -> bool:
    c, n, _ = nd
    return c > n

def solucionesNR(n: int) -> list[SolNR]:
    nInicial: NodoNR = (1,n,[])
    return [e for (_, _, e) in buscaAnchura(sucesoresNR,
                                            esFinalNR,
                                            nInicial)]

def primeraSolucionNR(n: int) -> SolNR:
    return solucionesNR(n)[0]

def nSolucionesNR(n: int) -> int:
    return len(solucionesNR(n))

# Verificación
# ============

def test_nReinas() -> None:
    assert solucionesNR(5)[:3] == \
        [[(1,1),(2,3),(3,5),(4,2),(5,4)],
         [(1,1),(2,4),(3,2),(4,5),(5,3)],
         [(1,2),(2,4),(3,1),(4,3),(5,5)]]
    assert nSolucionesNR(5) == 10
    print("Verificado")

# La verificación es
#    >>> test_nReinas()
#    Verificado
```

13.5. El problema de la mochila (mediante espacio de estados)

13.5.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Se tiene una mochila de capacidad de peso p y una lista de n objetos
-- para colocar en la mochila. Cada objeto i tiene un peso w(i) y un
-- valor v(i). Considerando la posibilidad de colocar el mismo objeto
-- varias veces en la mochila, el problema consiste en determinar la
-- forma de colocar los objetos en la mochila sin sobrepasar la
-- capacidad de la mochila colocando el máximo valor posible.
--
-- Para solucionar el problema se definen los siguientes tipos:
-- + Una solución del problema de la mochila es una lista de objetos.
--      type SolMoch = [Objeto]
-- + Los objetos son pares formado por un peso y un valor
--      type Objeto = (Peso,Valor)
-- + Los pesos son número enteros
--      type Peso = Int
-- + Los valores son números reales.
--      type Valor = Float
-- + Los estados del problema de la mochila son 5-tupla de la forma
--   (v,p,l,o,s) donde v es el valor de los objetos colocados, p es el
--   peso de los objetos colocados, l es el límite de la capacidad de la
--   mochila, o es la lista de los objetos colocados (ordenados de forma
--   creciente según sus pesos) y s es la solución parcial.
--      type NodoMoch = (Valor,Peso,Peso,[Objeto],SolMoch)
--
-- Usando el procedimiento de [búsqueda en profundidad](http://bit.ly/2sqPtGs),
-- definir la función
--    mochila :: [Objeto] -> Peso -> (SolMoch,Valor)
-- tal que (mochila os l) es la solución del problema de la mochila para
-- la lista de objetos os y el límite de capacidad l. Por ejemplo,
--    > mochila [(2,3),(3,5),(4,6),(5,10)] 8
--    ([(5,10.0),(3,5.0)],15.0)
--    > mochila [(2,3),(3,5),(5,6)] 10
--    ([(3,5.0),(3,5.0),(2,3.0),(2,3.0)],16.0)
--    > mochila [(8,15),(15,10),(3,6),(6,13),(2,4),(4,8),(5,6),(7,7)] 35
--    ([(6,13.0),(6,13.0),(6,13.0),(6,13.0),(6,13.0),(3,6.0),(2,4.0)],75.0)
```

```haskell
--    > mochila [(2,2.8),(3,4.4),(5,6.1)] 10
--    ([(3,4.4),(3,4.4),(2,2.8),(2,2.8)],14.4)
-- ---------------------------------------------------------------------

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

module BEE_Mochila where

import BusquedaEnProfundidad (buscaProfundidad)
import Data.List (sort)
import Test.Hspec (Spec, hspec, it, shouldBe)

type Peso     = Int
type Valor    = Float
type Objeto   = (Peso,Valor)
type SolMoch  = [Objeto]
type NodoMoch = (Valor,Peso,Peso,[Objeto],SolMoch)

mochila :: [Objeto] -> Peso -> (SolMoch,Valor)
mochila os l = (sol,v)
  where
    (v,_,_,_,sol) =
      maximum (buscaProfundidad sucesoresMoch
                                esObjetivoMoch
                                (inicial os l))

-- (inicial os l) es el estado inicial del problema de la mochila
-- para la lista de objetos os y el límite de capacidad l
inicial :: [Objeto] -> Peso -> NodoMoch
inicial os l =
  (0,0,l,sort os,[])

-- (sucesoresMoch e) es la lista de los sucesores del estado e en el
-- problema de la mochila para la lista de objetos os y el límite de
-- capacidad l.
sucesoresMoch :: NodoMoch -> [NodoMoch]
sucesoresMoch (v,p,l,os,solp) =
  [( v+v',
     p+p',
     l,
```

```
     [o | o@(p'',_) <- os, p''>=p'],
     (p',v'):solp )
  | (p',v') <- os,
    p+p' <= l]

-- (esObjetivoMoch e) se verifica si e es un estado final el problema de
-- la mochila para la lista de objetos os y el límite de capacidad l .
esObjetivoMoch :: NodoMoch -> Bool
esObjetivoMoch (_,p,l,(p',_):_,_) = p+p'>l

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

spec :: Spec
spec = do
  it ”e1” $
    mochila [(2,3),(3,5),(4,6),(5,10)] 8
    `shouldBe` ([(5,10.0),(3,5.0)],15.0)
  it ”e2” $
    mochila [(2,3),(3,5),(5,6)] 10
    `shouldBe` ([(3,5.0),(3,5.0),(2,3.0),(2,3.0)],16.0)
  it ”e3” $
    mochila [(8,15),(15,10),(3,6),(6,13),(2,4),(4,8),(5,6),(7,7)] 35
    `shouldBe` ([(6,13.0),(6,13.0),(6,13.0),(6,13.0),(6,13.0),(3,6.0),(2,4.0)],75
  it ”e4” $
    mochila [(2,2.8),(3,4.4),(5,6.1)] 10
    `shouldBe` ([(3,4.4),(3,4.4),(2,2.8),(2,2.8)],14.4)

-- La verificación es
--    λ> verifica
--
--    e1
--    e2
--    e3
--    e4
--
--    Finished in 0.0424 seconds
```

```
--    4 examples, 0 failures
```

13.5.2. En Python

```
# ---------------------------------------------------------------------
# Se tiene una mochila de capacidad de peso p y una lista de n objetos
# para colocar en la mochila. Cada objeto i tiene un peso w(i) y un
# valor v(i). Considerando la posibilidad de colocar el mismo objeto
# varias veces en la mochila, el problema consiste en determinar la
# forma de colocar los objetos en la mochila sin sobrepasar la
# capacidad de la mochila colocando el máximo valor posible.
#
# Para solucionar el problema se definen los siguientes tipos:
# + Una solución del problema de la mochila es una lista de objetos.
#      SolMoch = list[Objeto]
# + Los objetos son pares formado por un peso y un valor
#      Objeto = tuple[Peso, Valor]
# + Los pesos son número enteros
#      Peso = int
# + Los valores son números reales.
#      Valor = float
# + Los estados del problema de la mochila son 5-tupla de la forma
#   (v,p,l,o,s) donde v es el valor de los objetos colocados, p es el
#   peso de los objetos colocados, l es el límite de la capacidad de la
#   mochila, o es la lista de los objetos colocados (ordenados de forma
#   creciente según sus pesos) y s es la solución parcial.
#      NodoMoch = tuple[Valor, Peso, Peso, list[Objeto], SolMoch]
#
# Usando el procedimiento de [búsqueda en profundidad](http://bit.ly/2sqPtGs),
# definir la función
#    mochila : (list[Objeto], Peso) -> tuple[SolMoch, Valor]
# tal que mochila(os, l) es la solución del problema de la mochila para
# la lista de objetos os y el límite de capacidad l. Por ejemplo,
#    >>> mochila([(2,3),(3,5),(4,6),(5,10)], 8)
#    ([(5, 10), (3, 5)], 15)
#    >>> mochila([(2,3),(3,5),(5,6)], 10)
#    ([(3, 5), (3, 5), (2, 3), (2, 3)], 16)
#    >>> mochila([(8,15),(15,10),(3,6),(6,13), (2,4),(4,8),(5,6),(7,7)], 35)
#    ([(6, 13), (6, 13), (6, 13), (6, 13), (6, 13), (3, 6), (2, 4)], 75)
#    >>> mochila([(2,2.8),(3,4.4),(5,6.1)], 10)
```

```
#    ([(3, 4.4), (3, 4.4), (2, 2.8), (2, 2.8)], 14.4)
# ---------------------------------------------------------------------

from src.BusquedaEnProfundidad import buscaProfundidad

Peso = int
Valor = float
Objeto = tuple[Peso, Valor]
SolMoch = list[Objeto]
NodoMoch = tuple[Valor, Peso, Peso, list[Objeto], SolMoch]

# inicial(os, l) es el estado inicial del problema de la mochila
# para la lista de objetos os y el límite de capacidad l
def inicial(os: list[Objeto], l: Peso) -> NodoMoch:
    return (0,0,l,sorted(os),[])

# sucesoresMoch(e) es la lista de los sucesores del estado e en el
# problema de la mochila para la lista de objetos os y el límite de
# capacidad l.
def sucesoresMoch(n: NodoMoch) -> list[NodoMoch]:
    (v,p,l,os,solp) = n
    return [( v+v1,
              p+p1,
              l,
              [(p2,v2) for (p2,v2) in os if p2 >= p1],
              [(p1,v1)] + solp )
            for (p1,v1) in os if p + p1 <= l]

# esObjetivoMoch(e) se verifica si e es un estado final el problema de
# la mochila para la lista de objetos os y el límite de capacidad l .
def esObjetivoMoch(e: NodoMoch) -> bool:
    (_, p, l, os, _) = e
    (p_, _) = os[0]
    return p + p_ > l

def mochila(os: list[Objeto], l: Peso) -> tuple[SolMoch, Valor]:
    (v,_,_,_,sol) = max(buscaProfundidad(sucesoresMoch,
                                         esObjetivoMoch,
                                         inicial(os, l)))
    return (sol, v)
```

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

def test_Mochila() -> None:
    assert mochila([(2,3),(3,5),(4,6),(5,10)], 8) == \
        ([(5,10.0),(3,5.0)],15.0)
    assert mochila([(2,3),(3,5),(5,6)], 10) == \
        ([(3,5.0),(3,5.0),(2,3.0),(2,3.0)],16.0)
    assert mochila([(2,2.8),(3,4.4),(5,6.1)], 10) == \
        ([(3,4.4),(3,4.4),(2,2.8),(2,2.8)],14.4)
    print("Verificado")

# La verificación es
#    >>> test_Mochila()
#    Verificado
```

13.6. El tipo abstracto de datos de las colas de prioridad

13.6.1. En Haskell

```haskell
-- Una cola de prioridad es una cola en la que cada elemento tiene
-- asociada una prioridad. La operación de extracción siempre elige el
-- elemento de menor prioridad.
--
-- Las operaciones que definen a tipo abstracto de datos (TAD) de las
-- colas de prioridad (cuyos elementos son del tipo a) son las
-- siguientes:
--    vacia   :: Ord a => CPrioridad a
--    inserta :: Ord a => a -> CPrioridad a -> CPrioridad a
--    primero :: Ord a => CPrioridad a -> a
--    resto   :: Ord a => CPrioridad a -> CPrioridad a
--    esVacia :: Ord a => CPrioridad a -> Bool
-- tales que
-- + vacia es la cola de prioridad vacía.
-- + (inserta x c) añade el elemento x a la cola de prioridad c.
-- + (primero c) es el primer elemento de la cola de prioridad c.
-- + (resto c) es el resto de la cola de prioridad c.
```

```
-- + (esVacia c) se verifica si la cola de prioridad c es vacía.
--
-- Las operaciones tienen que verificar las siguientes propiedades:
-- + inserta x (inserta y c) == inserta y (inserta x c)
-- + primero (inserta x vacia) == x
-- + Si x <= y, entonces primero (inserta y (inserta x c)) == primero (inserta x
-- + resto (inserta x vacia) == vacia
-- + Si x <= y, entonces resto (inserta y (inserta x c)) == inserta y (resto (ins
-- + esVacia vacia
-- + not (esVacia (inserta x c))
--
-- Para usar el TAD hay que usar una implementación concreta. En
-- principio, consideraremos su representación mediante listas.

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

module TAD.ColaDePrioridad
  (CPrioridad,
   vacia,   -- Ord a => CPrioridad a
   inserta, -- Ord a => a -> CPrioridad a -> CPrioridad a
   primero, -- Ord a => CPrioridad a -> a
   resto,   -- Ord a => CPrioridad a -> CPrioridad a
   esVacia, -- Ord a => CPrioridad a -> Bool
  ) where

import TAD.ColaDePrioridadConListas
```

13.6.2. En Python

```
# Una cola de prioridad es una cola en la que cada elemento tiene
# asociada una prioridad. La operación de extracción siempre elige el
# elemento de menor prioridad.
#
# Las operaciones que definen a tipo abstracto de datos (TAD) de las
# colas de prioridad (cuyos elementos son del tipo a) son las
# siguientes:
#    vacia   :: Ord a => CPrioridad a
#    inserta :: Ord a => a -> CPrioridad a -> CPrioridad a
#    primero :: Ord a => CPrioridad a -> a
#    resto   :: Ord a => CPrioridad a -> CPrioridad a
```

```
#    esVacia :: Ord a => CPrioridad a -> Bool
# tales que
# + vacia es la cola de prioridad vacía.
# + (inserta x c) añade el elemento x a la cola de prioridad c.
# + (primero c) es el primer elemento de la cola de prioridad c.
# + (resto c) es el resto de la cola de prioridad c.
# + (esVacia c) se verifica si la cola de prioridad c es vacía.
#
# Las operaciones tienen que verificar las siguientes propiedades:
# + inserta x (inserta y c) == inserta y (inserta x c)
# + primero (inserta x vacia) == x
# + Si x <= y, entonces primero (inserta y (inserta x c)) == primero (inserta x c
# + resto (inserta x vacia) == vacia
# + Si x <= y, entonces resto (inserta y (inserta x c)) == inserta y (resto (inse
# + esVacia vacia
# + not (esVacia (inserta x c))
#
# Para usar el TAD hay que usar una implementación concreta. En
# principio, consideraremos su representación mediante listas.

__all__ = [
    'CPrioridad',
    'vacia',
    'inserta',
    'primero',
    'resto',
    'esVacia',
     ]

from src.TAD.ColaDePrioridadConListas import (CPrioridad, esVacia, inserta,
                                              primero, resto, vacia)
```

13.7. El tipo de datos de las colas de prioridad mediante listas

13.7.1. En Haskell

```
{-# LANGUAGE FlexibleInstances #-}
{-# OPTIONS_GHC -fno-warn-unused-top-binds #-}
```

```
module TAD.ColaDePrioridadConListas
  (CPrioridad,
   vacia,   -- Ord a => CPrioridad a
   inserta, -- Ord a => a -> CPrioridad a -> CPrioridad a
   primero, -- Ord a => CPrioridad a -> a
   resto,   -- Ord a => CPrioridad a -> CPrioridad a
   esVacia, -- Ord a => CPrioridad a -> Bool
  ) where

import Test.QuickCheck

-- Colas de prioridad mediante listas.
newtype CPrioridad a = CP [a]
  deriving Eq

-- (escribeColaDePrioridad c) es la cadena correspondiente a la cola de
-- prioridad c. Por ejemplo,
--    λ> escribeColaDePrioridad (inserta 5 (inserta 2 (inserta 3 vacia)))
--    "2 | 3 | 5"
escribeColaDePrioridad :: Show a => CPrioridad a -> String
escribeColaDePrioridad (CP [])     = "-"
escribeColaDePrioridad (CP [x])    = show x
escribeColaDePrioridad (CP (x:xs)) = show x ++ " | " ++ escribeColaDePrioridad (C

-- Procedimiento de escritura de colas de prioridad.
instance Show a => Show (CPrioridad a) where
  show = escribeColaDePrioridad

-- Ejemplo de cola de prioridad
--    λ> inserta 5 (inserta 2 (inserta 3 vacia))
--    2 | 3 | 5

-- vacia es la cola de prioridad vacía. Por ejemplo,
--    λ> vacia
--    CP []
vacia :: Ord a => CPrioridad a
vacia = CP []

-- (inserta x c) es la cola obtenida añadiendo el elemento x a la cola
```

```
-- de prioridad c. Por ejemplo,
--    λ> inserta 5 (foldr inserta vacia [3,1,7,2,9])
--    1 | 2 | 3 | 5 | 7 | 9
inserta :: Ord a => a -> CPrioridad a -> CPrioridad a
inserta x (CP q) = CP (ins x q)
  where ins y []                   = [y]
        ins y r@(e:r') | y < e     = y:r
                       | otherwise = e:ins y r'

-- (primero c) es el primer elemento de la cola de prioridad c. Por
-- ejemplo,
--    primero (foldr inserta vacia [3,1,7,2,9])  ==  1
primero :: Ord a => CPrioridad a -> a
primero (CP(x:_)) = x
primero _         = error "primero: cola de prioridad vacia"

-- (resto c) es la cola de prioridad obtenida eliminando el primer
-- elemento de la cola de prioridad c. Por ejemplo,
--    λ> resto (foldr inserta vacia [3,1,7,2,9])
--    2 | 3 | 7 | 9
resto :: Ord a => CPrioridad a -> CPrioridad a
resto (CP (_:xs)) = CP xs
resto _           = error "resto: cola de prioridad vacia"

-- (esVacia c) se verifica si la cola de prioridad c es vacía. Por
-- ejemplo,
--    esVacia (foldr inserta vacia [3,1,7,2,9]) ==  False
--    esVacia vacia                             ==  True
esVacia :: Ord a => CPrioridad a -> Bool
esVacia (CP xs) = null xs

-- Generador de colas de prioridad
-- ===============================

-- genCPrioridad es un generador de colas de enteros. Por ejemplo,
--    λ> sample genCPrioridad
--    -
--    0 | 0
--    4
--    -4 | -3 | 6 | 6
```

```
--    -7 | -6 | -2 | 0
--    -10 | -10 | -5 | 1 | 4 | 6 | 6 | 9 | 10
--    -
--    -13 | -11 | -9 | -5 | -2 | -1 | 0 | 1 | 2 | 2 | 13 | 14
--    -15 | -13 | -13 | -5 | -3 | -1 | 3 | 5 | 7 | 9 | 9 | 14 | 16
--    -
--    -17 | -15 | -14 | -5 | -2 | 1 | 1 | 2 | 5 | 7
genCPrioridad :: (Arbitrary a, Num a, Ord a) =>  Gen (CPrioridad a)
genCPrioridad = do
  xs <- listOf arbitrary
  return (foldr inserta vacia xs)

-- El tipo cola de prioridad es una instancia del arbitrario.
instance (Arbitrary a, Num a, Ord a) => Arbitrary (CPrioridad a) where
  arbitrary = genCPrioridad

-- Propiedades de las colas de prioridad
-- =====================================

-- Propiedad. Si se añade dos elementos a una cola de prioridad se
-- obtiene la misma cola de prioridad idependientemente del orden en
-- que se añadan los elementos.
prop_inserta_conmuta :: Int -> Int -> CPrioridad Int -> Bool
prop_inserta_conmuta x y c =
  inserta x (inserta y c) == inserta y (inserta x c)

-- Comprobación.
--    λ> quickCheck prop_inserta_conmuta
--    +++ OK, passed 100 tests.

-- Propiedad. La cabeza de la cola de prioridad obtenida añadiendo un
-- elemento x a la cola de prioridad vacía es x.
prop_primero_inserta_vacia :: Int -> CPrioridad Int -> Bool
prop_primero_inserta_vacia x _ =
  primero (inserta x vacia) == x

-- Comprobación.
--    λ> quickCheck prop_primero_inserta_vacia
--    +++ OK, passed 100 tests.
```

```haskell
-- Propiedad. El primer elemento de una cola de prioridad c no cambia
-- cuando se le añade un elemento mayor o igual que algún elemento de c.
prop_primero_inserta :: Int -> Int -> CPrioridad Int -> Property
prop_primero_inserta x y c =
  x <= y ==> primero (inserta y c') == primero c'
  where c' = inserta x c

-- Comprobación.
--    λ> quickCheck prop_primero_inserta
--    +++ OK, passed 100 tests.

-- Propiedad. El resto de añadir un elemento a la cola de prioridad
-- vacía es la cola vacía.
prop_resto_inserta_vacia :: Int -> Bool
prop_resto_inserta_vacia x =
  resto (inserta x vacia) == vacia

-- Comprobación.
--    λ> quickCheck prop_resto_inserta_vacia
--    +++ OK, passed 100 tests.

-- Propiedad. El resto de la cola de prioridad obtenida añadiendo un
-- elemento y a una cola c' (que tiene algún elemento menor o igual que
-- y) es la cola que se obtiene añadiendo y al resto de c'.
prop_resto_inserta :: Int -> Int -> CPrioridad Int -> Property
prop_resto_inserta x y c =
  x <= y ==> resto (inserta y c') == inserta y (resto c')
  where c' = inserta x c

-- Comprobación:
--    λ> quickCheck prop_resto_inserta
--    +++ OK, passed 100 tests.

-- Propiedad. vacia es una cola vacía.
prop_vacia_es_vacia :: Bool
prop_vacia_es_vacia = esVacia (vacia :: CPrioridad Int)

-- Comprobación.
--    λ> quickCheck prop_vacia_es_vacia
--    +++ OK, passed 100 tests.
```

```haskell
-- Propiedad. Si se añade un elemento a una cola de prioridad se obtiene
-- una cola no vacía.
prop_inserta_no_es_vacia :: Int -> CPrioridad Int -> Bool
prop_inserta_no_es_vacia x c =
  not (esVacia (inserta x c))

-- Comprobación.
--    λ> quickCheck prop_inserta_no_es_vacia
--    +++ OK, passed 100 tests.
```

13.7.2. En Python

```python
# Se define la clase CPrioridad con los siguientes métodos:
#    + inserta(x) añade x a la cola.
#    + primero() es el primero de la cola.
#    + resto() elimina el primero de la cola.
#    + esVacia() se verifica si la cola es vacía.
# Por ejemplo,
#    >>> c = CPrioridad()
#    >>> c
#    -
#    >>> c.inserta(5)
#    >>> c.inserta(2)
#    >>> c.inserta(3)
#    >>> c.inserta(4)
#    >>> c
#    2 | 3 | 4 | 5
#    >>> c.primero()
#    2
#    >>> c.resto()
#    >>> c
#    3 | 4 | 5
#    >>> c.esVacia()
#    False
#    >>> c = CPrioridad()
#    >>> c.esVacia()
#    True
#
# Además se definen las correspondientes funciones. Por ejemplo,
```

```
#    >>> vacia()
#    -
#    >>> inserta(4, inserta(3, inserta(2, inserta(5, vacia()))))
#    2 | 3 | 4 | 5
#    >>> primero (inserta(4, inserta(3, inserta(2, inserta(5, vacia())))))
#    2
#    >>> resto (inserta(4, inserta(3, inserta(2, inserta(5, vacia())))))
#    3 | 4 | 5
#    >>> esVacia(inserta(4, inserta(3, inserta(2, inserta(5, vacia())))))
#    False
#    >>> esVacia(vacia())
#    True
#
# Finalmente, se define un generador aleatorio de colas de prioridad y
# se comprueba que las colas de prioridad cumplen las propiedades de su
# especificación.

from __future__ import annotations

__all__ = [
   'CPrioridad',
   'vacia',
   'inserta',
   'primero',
   'resto',
   'esVacia',
]

from abc import abstractmethod
from copy import deepcopy
from dataclasses import dataclass, field
from typing import Generic, Protocol, TypeVar

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


class Comparable(Protocol):
    @abstractmethod
    def __lt__(self: A, otro: A) -> bool:
```

```
        pass

A = TypeVar('A', bound=Comparable)

# Clase de las colas de prioridad mediante listas
# ===============================================

@dataclass
class CPrioridad(Generic[A]):
    _elementos: list[A] = field(default_factory=list)

    def __repr__(self) -> str:
        """
        Devuelve una cadena con los elementos de la cola separados por " | ".
        Si la cola está vacía, devuelve "-".
        """
        if not self._elementos:
            return '-'
        return ' | '.join(str(x) for x in self._elementos)

    def esVacia(self) -> bool:
        """
        Comprueba si la cola está vacía.

        Devuelve True si la cola está vacía, False en caso contrario.
        """
        return not self._elementos

    def inserta(self, x: A) -> None:
        """
        Inserta el elemento x en la cola de prioridad.
        """
        self._elementos.append(x)
        self._elementos.sort()

    def primero(self) -> A:
        """
        Devuelve el primer elemento de la cola.
        """
        return self._elementos[0]
```

```
    def resto(self) -> None:
        """
        Elimina el primer elemento de la cola
        """
        self._elementos.pop(0)

# Funciones del tipo de las listas
# ================================

def vacia() -> CPrioridad[A]:
    """
    Crea y devuelve una cola vacía de tipo A.
    """
    c: CPrioridad[A] = CPrioridad()
    return c

def inserta(x: A, c: CPrioridad[A]) -> CPrioridad[A]:
    """
    Inserta un elemento x en la cola c y devuelve una nueva cola con
    el elemento insertado.
    """
    _aux = deepcopy(c)
    _aux.inserta(x)
    return _aux

def esVacia(c: CPrioridad[A]) -> bool:
    """
    Devuelve True si la cola está vacía, False si no lo está.
    """
    return c.esVacia()

def primero(c: CPrioridad[A]) -> A:
    """
    Devuelve el primer elemento de la cola c.
    """
    return c.primero()

def resto(c: CPrioridad[A]) -> CPrioridad[A]:
    """
```

```python
    Elimina el primer elemento de la cola c y devuelve una copia de la
    cola resultante.
    """
    _aux = deepcopy(c)
    _aux.resto()
    return _aux

# Generador de colas de prioridad
# ===============================

def colaAleatoria() -> st.SearchStrategy[CPrioridad[int]]:
    """
    Genera una estrategia de búsqueda para generar colas de enteros de
    forma aleatoria.

    Utiliza la librería Hypothesis para generar una lista de enteros y
    luego se convierte en una instancia de la clase cola.
    """
    return st.lists(st.integers()).map(CPrioridad)

# Comprobación de las propiedades de las colas
# ============================================

# Las propiedades son
@given(c=colaAleatoria(), x=st.integers(), y=st.integers())
def test_cola1(c: CPrioridad[int], x: int, y: int) -> None:
    assert inserta(x, inserta(y, c)) == inserta(y, inserta(x, c))
    assert primero(inserta(x, vacia())) == x
    assert resto(inserta(x, vacia())) == vacia()
    assert esVacia(vacia())
    assert not esVacia(inserta(x, c))

@given(c=colaAleatoria(), x=st.integers(), y=st.integers())
def test_cola2(c: CPrioridad[int], x: int, y: int) -> None:
    assume(not y < x)
    assert primero(inserta(y, (inserta(x, c)))) == \
        primero(inserta(x,c))
    assert resto(inserta(y, (inserta(x, c)))) == \
        inserta(y, resto(inserta(x, c)))
```

```
# La comprobación es
#    > poetry run pytest -q ColaDePrioridadConListas.py
#    2 passed in 0.54s
```

13.8. Búsqueda por primero el mejor

13.8.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- En la búsqueda por primero el mejor se supone que los estados están
-- ordenados mediante una función, la heurística, que es una estimación
-- de su coste para llegar a un estado final.
--
-- Definir la función
--    buscaPM :: Ord n => (n -> [n]) -> (n -> Bool) -> n -> [n]
-- tal que (buscaPM s o e) es la lista de soluciones del problema de
-- espacio de estado definido por la función sucesores s, el objetivo
-- o y estado inicial e, obtenidas buscando por primero el mejor.
-- ---------------------------------------------------------------------

module BusquedaPrimeroElMejor (buscaPM)  where

import TAD.ColaDePrioridad (esVacia, inserta, primero, resto, vacia)

buscaPM :: Ord n => (n -> [n]) -> (n -> Bool) -> n -> [n]
buscaPM sucesores esFinal x = busca' (inserta x vacia) where
  busca' c
    | esVacia c = []
    | esFinal (primero c) = primero c : busca' (resto c)
    | otherwise = busca' (foldr inserta (resto c) (sucesores (primero c)))
```

13.8.2. En Python

```python
# ---------------------------------------------------------------------
# En la búsqueda por primero el mejos se supone que los estados están
# ordenados mediante una función, la heurística, que es una rstimación
# de su coste para llegar a un estado final.
#
# Definir la función
```

```python
#    buscaPM : (Callable[[A], list[A]], Callable[[A], bool], A) -> Optional[A]
# tal que buscaPM(s, o, e) es la primera de las soluciones del problema de
# espacio de estado definido por la función sucesores s, el objetivo
# o y estado inicial e, obtenidas buscando por primero el mejor.
# ---------------------------------------------------------------------

from __future__ import annotations

from abc import abstractmethod
from functools import reduce
from typing import Callable, Optional, Protocol, TypeVar

from src.TAD.ColaDePrioridad import (CPrioridad, esVacia, inserta, primero,
                                     resto, vacia)


class Comparable(Protocol):
    @abstractmethod
    def __lt__(self: A, otro: A) -> bool:
        pass

A = TypeVar('A', bound=Comparable)

def buscaPM(sucesores: Callable[[A], list[A]],
            esFinal: Callable[[A], bool],
            inicial: A) -> Optional[A]:
    c: CPrioridad[A] = inserta(inicial, vacia())

    while not esVacia(c):
        if esFinal(primero(c)):
            return primero(c)

        es = sucesores(primero(c))
        c = reduce(lambda x, y: inserta(y, x), es, resto(c))

    return None
```

13.9. El problema del 8 puzzle

13.9.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Para el 8-puzzle se usa un cajón cuadrado en el que hay situados 8
-- bloques cuadrados. El cuadrado restante está sin rellenar. Cada
-- bloque tiene un número. Un bloque adyacente al hueco puede deslizarse
-- hacia él. El juego consiste en transformar la posición inicial en la
-- posición final mediante el deslizamiento de los bloques. En
-- particular, consideramos el estado inicial y final siguientes:
--
--    +---+---+---+                   +---+---+---+
--    |   | 1 | 3 |                   | 1 | 2 | 3 |
--    +---+---+---+                   +---+---+---+
--    | 8 | 2 | 4 |                   | 8 |   | 4 |
--    +---+---+---+                   +---+---+---+
--    | 7 | 5 | 5 |                   | 7 | 6 | 5 |
--    +---+---+---+                   +---+---+---+
--    Estado inicial                  Estado final
--
-- Para solucionar el problema se definen los siguientes tipos:
-- + Tablero es matriz de número enteros (que representan las piezas en
--   cada posición y el 0 representa el hueco):
--      type Tablero  = Matrix Int
-- + Estado es una listas de tableros [t_n,...,t_1] tal que t_i es un
--   sucesor de t_(i-1).
--      newtype Estado = Est [Tablero]
--        deriving Show
--
-- Usando el procedimiento de [búsqueda por primero el mejor](???),
-- definir la función
--    solucion_8puzzle :: Tablero -> [Tablero]
-- tal que (solucion_8puzzle t) es la solución del problema del problema
-- del 8 puzzle a partir del tablero0 t. Por ejemplo,
--    λ> solucion_8puzzle (fromLists [[0,1,3],[8,2,4],[7,6,5]])
--    [┌       ┐  ┌       ┐  ┌       ┐
--     │ 0 1 3 │  │ 1 0 3 │  │ 1 2 3 │
--     │ 8 2 4 │  │ 8 2 4 │  │ 8 0 4 │
--     │ 7 6 5 │  │ 7 6 5 │  │ 7 6 5 │
--     └       ┘, └       ┘, └       ┘]
```

```haskell
--    λ> length (solucion_8puzzle (fromLists [[2,6,3],[5,0,4],[1,7,8]]))
--    21
-- ---------------------------------------------------------------------

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

module BPM_8Puzzle where

import BusquedaPrimeroElMejor (buscaPM)
import Data.Matrix (Matrix, (!), fromLists, setElem, toLists)
import Test.Hspec (Spec, hspec, it, shouldBe)

type Tablero  = Matrix Int

newtype Estado = Est [Tablero]
  deriving (Eq, Show)

solucion_8puzzle :: Tablero -> [Tablero]
solucion_8puzzle t = reverse ts
  where (Est ts) = head (buscaPM sucesores
                                 esFinal
                                 (inicial t))

-- Estado inicial
-- ==============

-- (inicial t) es el estado inicial del problema del 8 puzzle a partir del
-- tablero t.
inicial :: Tablero -> Estado
inicial t = Est [t]

-- Estado final
-- ============

-- (esFinal e) se verifica si e es un estado final.
esFinal :: Estado -> Bool
esFinal (Est (n:_)) = n == tableroFinal

-- tableroFinal es el estado tablero final del 8 puzzle.
tableroFinal :: Tablero
```

```
tableroFinal = fromLists [[1,2,3],
                          [8,0,4],
                          [7,6,5]]

-- Sucesores
-- =========

-- (sucesores e) es la lista de sucesores del estado e. Por ejemplo,
--    λ> sucesores (Est [fromLists [[2,1,3],[8,0,4],[7,6,5]]])
--    [Est [┌       ┐  ┌       ┐
--          │ 2 0 3 │  │ 2 1 3 │
--          │ 8 1 4 │  │ 8 0 4 │
--          │ 7 6 5 │  │ 7 6 5 │
--          └       ┘, └       ┘],
--     Est [┌       ┐  ┌       ┐
--          │ 2 1 3 │  │ 2 1 3 │
--          │ 8 6 4 │  │ 8 0 4 │
--          │ 7 0 5 │  │ 7 6 5 │
--          └       ┘, └       ┘],
--     Est [┌       ┐  ┌       ┐
--          │ 2 1 3 │  │ 2 1 3 │
--          │ 0 8 4 │  │ 8 0 4 │
--          │ 7 6 5 │  │ 7 6 5 │
--          └       ┘, └       ┘],
--     Est [┌       ┐  ┌       ┐
--          │ 2 1 3 │  │ 2 1 3 │
--          │ 8 4 0 │  │ 8 0 4 │
--          │ 7 6 5 │  │ 7 6 5 │
--          └       ┘, └       ┘]]
sucesores :: Estado -> [Estado]
sucesores (Est e@(t:_)) =
  [Est (t':e) | t' <- tablerosSucesores t,
                t' `notElem` e]

-- (tablerosSucesores t) es la lista de los tableros sucesores del
-- tablero t. Por ejemplo,
--    λ> tablerosSucesores (fromLists [[2,1,3],[8,0,4],[7,6,5]])
--    [┌       ┐  ┌       ┐  ┌       ┐  ┌       ┐
--     │ 2 0 3 │  │ 2 1 3 │  │ 2 1 3 │  │ 2 1 3 │
--     │ 8 1 4 │  │ 8 6 4 │  │ 0 8 4 │  │ 8 4 0 │
```

```
--     │ 7 6 5 │  │ 7 0 5 │  │ 7 6 5 │  │ 7 6 5 │
--     └       ┘, └       ┘, └       ┘, └       ┘]
tablerosSucesores :: Tablero -> [Tablero]
tablerosSucesores t =
  [intercambia t p q | q <- posicionesVecinas p]
  where p = posicionHueco t

-- Una posición es un par de enteros.
type Posicion = (Int,Int)

-- (posicionesVecinas p) son las posiciones de la matriz cuadrada de
-- dimensión 3 que se encuentran encima, abajo, a la izquierda y a la
-- derecha de los posición p. Por ejemplo,
--    λ> posicionesVecinas (2,2)
--    [(1,2),(3,2),(2,1),(2,3)]
--    λ> posicionesVecinas (1,2)
--    [(2,2),(1,1),(1,3)]
--    λ> posicionesVecinas (1,1)
--    [(2,1),(1,2)]
posicionesVecinas :: Posicion -> [Posicion]
posicionesVecinas (i,j) =
  [(i-1,j) | i > 1] ++
  [(i+1,j) | i < 3] ++
  [(i,j-1) | j > 1] ++
  [(i,j+1) | j < 3]

-- (posicionHueco t) es la posición del hueco en el tablero t. Por
-- ejemplo,
--    λ> posicionHueco (fromLists [[2,1,3],[8,0,4],[7,6,5]])
--    (2,2)
posicionHueco :: Tablero -> Posicion
posicionHueco t =
  posicionElemento t 0

-- (posicionElemento t a) es la posición de elemento a en el tablero
-- t. Por ejemplo,
--    λ> posicionElemento (fromLists [[2,1,3],[8,0,4],[7,6,5]]) 4
--    (2,3)
posicionElemento :: Tablero -> Int -> Posicion
posicionElemento t a =
```

```
  head [(i,j) | i <- [1..3],
                j <- [1..3],
                t ! (i,j) == a]

-- (intercambia t p1 p2) es el tablero obtenido intercambiando en t los
-- elementos que se encuentran en las posiciones p1 y p2. Por ejemplo,
--    λ> intercambia (fromLists [[2,1,3],[8,0,4],[7,6,5]]) (1,2) (2,2)
--    ┌       ┐
--    │ 2 0 3 │
--    │ 8 1 4 │
--    │ 7 6 5 │
--    └       ┘
intercambia :: Tablero -> Posicion -> Posicion -> Tablero
intercambia t p1 p2 =
  setElem a2 p1 (setElem a1 p2 t)
  where a1 = t ! p1
        a2 = t ! p2

-- Heurística
-- ==========

-- (heuristica t) es la suma de la distancia Manhatan desde la posición de
-- cada objeto del tablero a su posición en el tablero final. Por
-- ejemplo,
--    λ> heuristica (fromLists [[0,1,3],[8,2,4],[7,6,5]])
--    4
heuristica :: Tablero  -> Int
heuristica t =
  sum [distancia (posicionElemento t i)
                 (posicionElemento tableroFinal i)
      | i <- [0..8]]

-- (distancia p1 p2) es la distancia Manhatan entre las posiciones p1 y
-- p2. Por ejemplo,
--    distancia (2,7) (4,1)  ==  8
distancia :: Posicion -> Posicion -> Int
distancia (x1,y1) (x2,y2) = abs (x1-x2) + abs (y1-y2)

-- Comparación de estados
-- ======================
```

```haskell
-- Un estado es menor o igual que otro si tiene la heurística de su
-- primer tablero es menor o que la del segundo o so iguales y el
-- primero es más corto.
instance Ord Estado where
  Est (t1:ts1) <= Est (t2:ts2) = (heuristica t1 < heuristica t2) ||
                                 ((heuristica t1 == heuristica t2) &&
                                  (length ts1 <= length ts2))

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

spec :: Spec
spec = do
  it "e1" $
    map toLists (solucion_8puzzle (fromLists [[0,1,3],[8,2,4],[7,6,5]]))
   `shouldBe` [[[0,1,3],
                [8,2,4],
                [7,6,5]],
               [[1,0,3],
                [8,2,4],
                [7,6,5]],
               [[1,2,3],
                [8,0,4],
                [7,6,5]]]
  it "e2" $
    length (solucion_8puzzle (fromLists [[2,6,3],[5,0,4],[1,7,8]]))
    `shouldBe` 21

-- La verificación es
--    λ> verifica
--
--    e1
--    e2
--
--    Finished in 0.1361 seconds
--    2 examples, 0 failures
```

13.9.2. En Python

```
# ---------------------------------------------------------------------
# Para el 8-puzzle se usa un cajón cuadrado en el que hay situados 8
# bloques cuadrados. El cuadrado restante está sin rellenar. Cada
# bloque tiene un número. Un bloque adyacente al hueco puede deslizarse
# hacia él. El juego consiste en transformar la posición inicial en la
# posición final mediante el deslizamiento de los bloques. En
# particular, consideramos el estado inicial y final siguientes:
#
#    +---+---+---+                  +---+---+---+
#    |   | 1 | 3 |                  | 1 | 2 | 3 |
#    +---+---+---+                  +---+---+---+
#    | 8 | 2 | 4 |                  | 8 |   | 4 |
#    +---+---+---+                  +---+---+---+
#    | 7 | 5 | 5 |                  | 7 | 6 | 5 |
#    +---+---+---+                  +---+---+---+
#    Estado inicial                 Estado final
#
# Para solucionar el problema se definen los siguientes tipos:
# + Tablero es lista de listas de números enteros (que representan las
#   piezas en  cada posición y el 0 representa el hueco):
#      Tablero = list[list[int]]
# + Estado es una tupla (h, n, ts), donde ts es una listas de tableros
#   [t_n,...,t_1] tal que t_i es un sucesor de t_(i-1) y h es la
#   heurística de t_n.
#      Estado = tuple[int, int, list[Tablero]]
#
# Usando el procedimiento de [búsqueda por primero el mejor](???),
# definir la función
#    solucion_8puzzle : (Tablero) -> Tablero
# tal que solucion_8puzzle(t) es la solución del problema del problema
# del 8 puzzle a partir del tablero t. Por ejemplo,
#    >>> solucion_8puzzle([[0,1,3],[8,2,4],[7,6,5]])
#    [[[0, 1, 3],
#      [8, 2, 4],
#      [7, 6, 5]],
#     [[1, 0, 3],
#      [8, 2, 4],
#      [7, 6, 5]],
#     [[1, 2, 3],
```

```
#       [8, 0, 4],
#       [7, 6, 5]]]
#     >>> solucion_8puzzle([[8,1,3],[0,2,4],[7,6,5]])
#     [[[8, 1, 3],
#       [0, 2, 4],
#       [7, 6, 5]],
#      [[0, 1, 3],
#       [8, 2, 4],
#       [7, 6, 5]],
#      [[1, 0, 3],
#       [8, 2, 4],
#       [7, 6, 5]],
#      [[1, 2, 3],
#       [8, 0, 4],
#       [7, 6, 5]]]
#     >>> len(solucion_8puzzle([[2,6,3],[5,0,4],[1,7,8]]))
#     21
# ---------------------------------------------------------------------

from copy import deepcopy
from typing import Optional

from src.BusquedaPrimeroElMejor import buscaPM

Tablero = list[list[int]]

# Tablero final
# =============

# tableroFinal es el tablero final del 8 puzzle.
tableroFinal: Tablero = [[1,2,3],
                         [8,0,4],
                         [7,6,5]]

# Posiciones
# ==========

# Una posición es un par de enteros.
Posicion = tuple[int,int]
```

```python
# Heurística
# ==========

# distancia(p1, p2) es la distancia Manhatan entre las posiciones p1 y
# p2. Por ejemplo,
#    >>> distancia((2,7), (4,1))
#    8
def distancia(p1: Posicion, p2: Posicion) -> int:
    (x1, y1) = p1
    (x2, y2) = p2
    return abs(x1-x2) + abs (y1-y2)

# posicionElemento(t, a) es la posición de elemento a en el tablero
# t. Por ejemplo,
#    λ> posicionElemento([[2,1,3],[8,0,4],[7,6,5]], 4)
#    (1, 2)
def posicionElemento(t: Tablero, a: int) -> Posicion:
    for i in range(0, 3):
        for j in range(0, 3):
            if t[i][j] == a:
                return (i, j)
    return (4, 4)

# posicionHueco(t) es la posición del hueco en el tablero t. Por
# ejemplo,
#    >>> posicionHueco([[2,1,3],[8,0,4],[7,6,5]])
#    (1, 1)
def posicionHueco(t: Tablero) -> Posicion:
    return posicionElemento(t, 0)

# heuristica(t) es la suma de la distancia Manhatan desde la posición de
# cada objeto del tablero a su posición en el tablero final. Por
# ejemplo,
#    >>> heuristica([[0,1,3],[8,2,4],[7,6,5]])
#    4
def heuristica(t: Tablero) -> int:
    return sum((distancia(posicionElemento(t, i),
                          posicionElemento(tableroFinal, i))
                for i in range(0, 10)))
```

```
# Estados
# =======

# Un estado es una tupla (h, n, ts), donde ts es una listas de tableros
# [t_n,...,t_1] tal que t_i es un sucesor de t_(i-1) y h es la
# heurística de t_n.
Estado = tuple[int, int, list[Tablero]]

# Estado inicial
# ==============

# inicial(t) es el estado inicial del problema del 8 puzzle a partir del
# tablero t.
def inicial(t: Tablero) -> Estado:
    return (heuristica(t), 1, [t])

# Estado final
# ============

# esFinal(e) se verifica si e es un estado final.
def esFinal(e: Estado) -> bool:
    (_, _, ts) = e
    return ts[0] == tableroFinal

# Sucesores
# =========

# posicionesVecinas(p) son las posiciones de la matriz cuadrada de
# dimensión 3 que se encuentran encima, abajo, a la izquierda y a la
# derecha de los posición p. Por ejemplo,
#    >>> posicionesVecinas((1,1))
#    [(0, 1), (2, 1), (1, 0), (1, 2)]
#    >>> posicionesVecinas((0,1))
#    [(1, 1), (0, 0), (0, 2)]
#    >>> posicionesVecinas((0,0))
#    [(1, 0), (0, 1)]
def posicionesVecinas(p: Posicion) -> list[Posicion]:
    (i, j) = p
    vecinas = []
    if i > 0:
```

```
        vecinas.append((i - 1, j))
    if i < 2:
        vecinas.append((i + 1, j))
    if j > 0:
        vecinas.append((i, j - 1))
    if j < 2:
        vecinas.append((i, j + 1))
    return vecinas

# intercambia(t,p1, p2) es el tablero obtenido intercambiando en t los
# elementos que se encuentran en las posiciones p1 y p2. Por ejemplo,
#    >>> intercambia([[2,1,3],[8,0,4],[7,6,5]], (0,1), (1,1))
#    [[2, 0, 3], [8, 1, 4], [7, 6, 5]]
def intercambia(t: Tablero, p1: Posicion, p2: Posicion) -> Tablero:
    (i1, j1) = p1
    (i2, j2) = p2
    t1 = deepcopy(t)
    a1 = t1[i1][j1]
    a2 = t1[i2][j2]
    t1[i1][j1] = a2
    t1[i2][j2] = a1
    return t1

# tablerosSucesores(t) es la lista de los tablrtos sucesores del
# tablero t. Por ejemplo,
#    >>> tablerosSucesores([[2,1,3],[8,0,4],[7,6,5]])
#    [[[2, 0, 3], [8, 1, 4], [7, 6, 5]],
#     [[2, 1, 3], [8, 6, 4], [7, 0, 5]],
#     [[2, 1, 3], [0, 8, 4], [7, 6, 5]],
#     [[2, 1, 3], [8, 4, 0], [7, 6, 5]]]
def tablerosSucesores(t: Tablero) -> list[Tablero]:
    p = posicionHueco(t)
    return [intercambia(t, p, q) for q in posicionesVecinas(p)]

# (sucesores e) es la lista de sucesores del estado e. Por ejemplo,
#    >>> t1 = [[0,1,3],[8,2,4],[7,6,5]]
#    >>> es = sucesores((heuristica(t1), 1, [t1]))
#    >>> es
#    [(4, 2, [[[8, 1, 3],
#              [0, 2, 4],
```

```
#                [7, 6, 5]],
#               [[0, 1, 3],
#                [8, 2, 4],
#                [7, 6, 5]]]),
#      (2, 2, [[[1, 0, 3],
#                [8, 2, 4],
#                [7, 6, 5]],
#               [[0, 1, 3],
#                [8, 2, 4],
#                [7, 6, 5]]])]
#     >>> sucesores(es[1])
#     [(0, 3, [[[1, 2, 3],
#                [8, 0, 4],
#                [7, 6, 5]],
#               [[1, 0, 3],
#                [8, 2, 4],
#                [7, 6, 5]],
#               [[0, 1, 3],
#                [8, 2, 4],
#                [7, 6, 5]]]),
#      (4, 3, [[[1, 3, 0],
#                [8, 2, 4],
#                [7, 6, 5]],
#               [[1, 0, 3],
#                [8, 2, 4],
#                [7, 6, 5]],
#               [[0, 1, 3],
#                [8, 2, 4],
#                [7, 6, 5]]])]
def sucesores(e: Estado) -> list[Estado]:
    (_, n, ts) = e
    return [(heuristica(t1), n+1, [t1] + ts)
            for t1 in tablerosSucesores(ts[0])
            if t1 not in ts]

# Solución
# ========

def solucion_8puzzle(t: Tablero) -> Optional[list[Tablero]]:
    r = buscaPM(sucesores, esFinal, inicial(t))
```

```python
    if r is None:
        return None
    (_, _, ts) = r
    ts.reverse()
    return ts

# Verificación
# ============

def test_8puzzle() -> None:
    assert solucion_8puzzle([[8,1,3],[0,2,4],[7,6,5]]) == \
        [[[8, 1, 3], [0, 2, 4], [7, 6, 5]],
         [[0, 1, 3], [8, 2, 4], [7, 6, 5]],
         [[1, 0, 3], [8, 2, 4], [7, 6, 5]],
         [[1, 2, 3], [8, 0, 4], [7, 6, 5]]]

# La verificación es
#    src> poetry run pytest -q BPM_8Puzzle.py
#    1 passed in 0.10s
```

13.10. Búsqueda en escalada

13.10.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- En la búsqueda en escalada se supone que los estados están ordenados
-- mediante una función, la heurística, que es una estimación de su
-- coste para llegar a un estado final.
--
-- Definir la función
--    buscaEscalada :: Ord n => (n -> [n]) -> (n -> Bool) -> n -> [n]
-- tal que (buscaEscalada s o e) es la lista de soluciones del problema de
-- espacio de estado definido por la función sucesores s, el objetivo
-- o y estado inicial e, obtenidas buscando en escalada.
-- ---------------------------------------------------------------------

module BusquedaEnEscalada (buscaEscalada)  where

import TAD.ColaDePrioridad (esVacia, inserta, primero, vacia)
```

```haskell
buscaEscalada :: Ord n => (n -> [n]) -> (n -> Bool) -> n -> [n]
buscaEscalada sucesores esFinal x = busca' (inserta x vacia) where
  busca' c
    | esVacia c           = []
    | esFinal (primero c) = [primero c]
    | otherwise           = busca' (foldr inserta vacia (sucesores (primero c)))
```

13.10.2. En Python

```python
# ---------------------------------------------------------------------
# En la búsqueda en escalada se supone que los estados están ordenados
# mediante una función, la heurística, que es una estimación de su
# coste para llegar a un estado final.
#
# Definir la función
#    buscaEscalada(Callable[[A], list[A]], Callable[[A], bool], A) -> list[A]
# tal que buscaEscalada(s, o, e) es la lista de soluciones del problema de
# espacio de estado definido por la función sucesores s, el objetivo
# o y estado inicial e, obtenidas buscando en escalada.
# ---------------------------------------------------------------------

from __future__ import annotations

from abc import abstractmethod
from functools import reduce
from typing import Callable, Optional, Protocol, TypeVar

from src.TAD.ColaDePrioridad import (CPrioridad, esVacia, inserta, primero,
                                     vacia)


class Comparable(Protocol):
    @abstractmethod
    def __lt__(self: A, otro: A) -> bool:
        pass

A = TypeVar('A', bound=Comparable)

def buscaEscalada(sucesores: Callable[[A], list[A]],
                  esFinal: Callable[[A], bool],
```

```
                 inicial: A) -> Optional[A]:
    c: CPrioridad[A] = inserta(inicial, vacia())

    while not esVacia(c):
        x = primero(c)
        if esFinal(x):
            return x

        c = reduce(lambda x, y: inserta(y, x), sucesores(x), vacia())

    return None
```

13.11. El algoritmo de Prim del árbol de expansión mínimo por escalada

13.11.1. En Haskell

```
-- ---------------------------------------------------------------------
-- El [algoritmo de Prim](https://bit.ly/466fwRe) calcula un árbol
-- recubridor mínimo en un grafo conexo y ponderado. Es decir, busca un
-- subconjunto de aristas que, formando un árbol, incluyen todos los
-- vértices y donde el valor de la suma de todas las aristas del árbol
-- es el mínimo.
--
-- El algoritmo de Prim funciona de la siguiente manera:
-- + Inicializar un árbol con un único vértice, elegido arbitrariamente,
--   del grafo.
-- + Aumentar el árbol por un lado. Llamamos lado a la unión entre dos
--   vértices: de las posibles uniones que pueden conectar el árbol a los
--   vértices que no están aún en el árbol, encontrar el lado de menor
--   distancia y unirlo al árbol.
-- + Repetir el paso 2 (hasta que todos los vértices pertenezcan al
--   árbol)
--
-- Usando la [búsqueda en escalada](https://bit.ly/3Kk4A99) el [tipo
-- abstracto de datos de los grafos](https://bit.ly/45cQ3Fo), definir la
-- función
--    prim :: (Ix v, Num p, Ord p) => Grafo v p -> [(p,v,v)]
-- tal que (prim g) es el árbol de expansión mínimo del grafo g
```

```
-- calculado mediante el algoritmo de Prim con bñusqueda en
-- escalada. Por ejemplo, si g1, g2, g3 y g4 son los grafos definidos
-- por
--    g1, g2, g3, g4 :: Grafo Int Int
--    g1 = creaGrafo ND (1,5) [(1,2,12),(1,3,34),(1,5,78),
--                              (2,4,55),(2,5,32),
--                              (3,4,61),(3,5,44),
--                              (4,5,93)]
--    g2 = creaGrafo ND (1,5) [(1,2,13),(1,3,11),(1,5,78),
--                              (2,4,12),(2,5,32),
--                              (3,4,14),(3,5,44),
--                              (4,5,93)]
--    g3 = creaGrafo ND (1,7) [(1,2,5),(1,3,9),(1,5,15),(1,6,6),
--                              (2,3,7),
--                              (3,4,8),(3,5,7),
--                              (4,5,5),
--                              (5,6,3),(5,7,9),
--                              (6,7,11)]
--    g4 = creaGrafo ND (1,7) [(1,2,5),(1,3,9),(1,5,15),(1,6,6),
--                              (2,3,7),
--                              (3,4,8),(3,5,1),
--                              (4,5,5),
--                              (5,6,3),(5,7,9),
--                              (6,7,11)]
-- entonces
--    prim g1 == [(2,4,55),(1,3,34),(2,5,32),(1,2,12)]
--    prim g2 == [(2,5,32),(2,4,12),(1,2,13),(1,3,11)]
--    prim g3 == [(5,7,9),(2,3,7),(5,4,5),(6,5,3),(1,6,6),(1,2,5)]
--    prim g4 == [(5,7,9),(5,4,5),(5,3,1),(6,5,3),(1,6,6),(1,2,5)]
-- ---------------------------------------------------------------------

module Escalada_Prim where

import BusquedaEnEscalada (buscaEscalada)
import TAD.Grafo (Grafo, Orientacion (ND), aristaEn, creaGrafo, nodos, peso)
import Data.Ix (Ix)
import Data.List (delete)
import Test.Hspec (Spec, hspec, it, shouldBe)

g1, g2, g3, g4 :: Grafo Int Int
```

```haskell
g1 = creaGrafo ND (1,5) [(1,2,12),(1,3,34),(1,5,78),
                         (2,4,55),(2,5,32),
                         (3,4,61),(3,5,44),
                         (4,5,93)]
g2 = creaGrafo ND (1,5) [(1,2,13),(1,3,11),(1,5,78),
                         (2,4,12),(2,5,32),
                         (3,4,14),(3,5,44),
                         (4,5,93)]
g3 = creaGrafo ND (1,7) [(1,2,5),(1,3,9),(1,5,15),(1,6,6),
                         (2,3,7),
                         (3,4,8),(3,5,7),
                         (4,5,5),
                         (5,6,3),(5,7,9),
                         (6,7,11)]
g4 = creaGrafo ND (1,7) [(1,2,5),(1,3,9),(1,5,15),(1,6,6),
                         (2,3,7),
                         (3,4,8),(3,5,1),
                         (4,5,5),
                         (5,6,3),(5,7,9),
                         (6,7,11)]

-- Una arista esta formada por dos vértices junto con su peso.
type Arista a b = (a,a,b)

-- Un estado (Estado (p,t,r,aem)) está formado por el peso p de la
-- última arista añadida el árbol de expansión mínimo (aem), la lista t
-- de nodos del grafo que están en el aem, la lista r de nodos del
-- grafo que no están en el aem y el aem.
type Estado a b = (b,[a],[a],[Arista a b])

-- (inicial g) es el estado inicial correspondiente al grafo g.
inicial :: (Ix a, Num b, Ord b) => Grafo a b -> Estado a b
inicial g = (0,[n],ns,[])
  where (n:ns) = nodos g

-- (esFinal e) se verifica si e es un estado final; es decir, si no
-- queda ningún elemento en la lista de nodos sin colocar en el árbol de
-- expansión mínimo.
esFinal :: Estado a b -> Bool
esFinal (_,_,[],_) = True
```

```
esFinal _           = False

-- (sucesores g e) es la lista de los sucesores del estado e en el
-- grafo g. Por ejemplo,
--    λ> sucesores g1 (0,[1],[2..5],[])
--    [(12,[2,1],[3,4,5],[(1,2,12)]),
--     (34,[3,1],[2,4,5],[(1,3,34)]),
--     (78,[5,1],[2,3,4],[(1,5,78)])]
sucesores
  :: (Ix a, Num b, Eq b) => Grafo a b -> Estado a b -> [Estado a b]
sucesores g (_,t,r,aem) =
  [(peso x y g, y:t, delete y r, (x,y,peso x y g):aem)
   | x <- t , y <- r, aristaEn g (x,y)]

prim :: (Ix a, Num b, Ord b) => Grafo a b -> [Arista a b]
prim g = sol
  where [(_,_,_,sol)] = buscaEscalada (sucesores g)
                                      esFinal
                                      (inicial g)

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

spec :: Spec
spec = do
  it "e1" $
    prim g1 `shouldBe` [(2,4,55),(1,3,34),(2,5,32),(1,2,12)]
  it "e2" $
    prim g2 `shouldBe` [(2,5,32),(2,4,12),(1,2,13),(1,3,11)]
  it "e3" $
    prim g3 `shouldBe` [(5,7,9),(2,3,7),(5,4,5),(6,5,3),(1,6,6),(1,2,5)]
  it "e4" $
    prim g4 `shouldBe` [(5,7,9),(5,4,5),(5,3,1),(6,5,3),(1,6,6),(1,2,5)]

-- La verificación es
--    λ> verifica
--
```

```
--    e1
--    e2
--    e3
--    e4
--
--    Finished in 0.0043 seconds
--    4 examples, 0 failures
```

13.11.2. En Python

```
# ---------------------------------------------------------------------
# El [algoritmo de Prim](https://bit.ly/466fwRe) calcula un árbol
# recubridor mínimo en un grafo conexo y ponderado. Es decir, busca un
# subconjunto de aristas que, formando un árbol, incluyen todos los
# vértices y donde el valor de la suma de todas las aristas del árbol
# es el mínimo.
#
# El algoritmo de Prim funciona de la siguiente manera:
# + Inicializar un árbol con un único vértice, elegido arbitrariamente,
#   del grafo.
# + Aumentar el árbol por un lado. Llamamos lado a la unión entre dos
#   vértices: de las posibles uniones que pueden conectar el árbol a los
#   vértices que no están aún en el árbol, encontrar el lado de menor
#   distancia y unirlo al árbol.
# + Repetir el paso 2 (hasta que todos los vértices pertenezcan al
#   árbol)
#
# Usando la [búsqueda en escalada](https://bit.ly/3Kk4A99) el [tipo
# abstracto de datos de los grafos](https://bit.ly/45cQ3Fo), definir la
# función
#    prim : (Grafo) -> list[tuple[Peso, Vertice, Vertice]]
# tal que prim(g) es el árbol de expansión mínimo del grafo g
# calculado mediante el algoritmo de Prim con bñusqueda en
# escalada. Por ejemplo, si g1, g2, g3 y g4 son los grafos definidos
# por
#    g1 = creaGrafo (Orientacion.ND,
#                    (1,5),
#                    [((1,2),12),((1,3),34),((1,5),78),
#                     ((2,4),55),((2,5),32),
#                     ((3,4),61),((3,5),44),
```

```
#                      ((4,5),93)])
#    g2 = creaGrafo (Orientacion.ND,
#                     (1,5),
#                     [((1,2),13),((1,3),11),((1,5),78),
#                      ((2,4),12),((2,5),32),
#                      ((3,4),14),((3,5),44),
#                      ((4,5),93)])
#    g3 = creaGrafo (Orientacion.ND,
#                     (1,7),
#                     [((1,2),5),((1,3),9),((1,5),15),((1,6),6),
#                      ((2,3),7),
#                      ((3,4),8),((3,5),7),
#                      ((4,5),5),
#                      ((5,6),3),((5,7),9),
#                      ((6,7),11)])
#    g4 = creaGrafo (Orientacion.ND,
#                     (1,7),
#                     [((1,2),5),((1,3),9),((1,5),15),((1,6),6),
#                      ((2,3),7),
#                      ((3,4),8),((3,5),1),
#                      ((4,5),5),
#                      ((5,6),3),((5,7),9),
#                      ((6,7),11)])
# entonces
#    prim(g1) == [((2,4),55),((1,3),34),((2,5),32),((1,2),12)]
#    prim(g2) == [((2,5),32),((2,4),12),((1,2),13),((1,3),11)]
#    prim(g3) == [((5,7),9),((2,3),7),((5,4),5),((6,5),3),((1,6),6),((1,2),5)]
#    prim(g4) == [((5,7),9),((5,4),5),((5,3),1),((6,5),3),((1,6),6),((1,2),5)]
# ---------------------------------------------------------------------

from typing import Optional

from src.BusquedaEnEscalada import buscaEscalada
from src.TAD.Grafo import (Grafo, Orientacion, Peso, Vertice, aristaEn,
                           creaGrafo, nodos, peso)

g1 = creaGrafo (Orientacion.ND,
                (1,5),
                [((1,2),12),((1,3),34),((1,5),78),
                 ((2,4),55),((2,5),32),
```

```
                 ((3,4),61),((3,5),44),
                 ((4,5),93)])
g2 = creaGrafo (Orientacion.ND,
                (1,5),
                [((1,2),13),((1,3),11),((1,5),78),
                 ((2,4),12),((2,5),32),
                 ((3,4),14),((3,5),44),
                 ((4,5),93)])
g3 = creaGrafo (Orientacion.ND,
                (1,7),
                [((1,2),5),((1,3),9),((1,5),15),((1,6),6),
                 ((2,3),7),
                 ((3,4),8),((3,5),7),
                 ((4,5),5),
                 ((5,6),3),((5,7),9),
                 ((6,7),11)])
g4 = creaGrafo (Orientacion.ND,
                (1,7),
                [((1,2),5),((1,3),9),((1,5),15),((1,6),6),
                 ((2,3),7),
                 ((3,4),8),((3,5),1),
                 ((4,5),5),
                 ((5,6),3),((5,7),9),
                 ((6,7),11)])

Arista = tuple[tuple[Vertice, Vertice], Peso]

# Un nodo (Estado (p,t,r,aem)) está formado por el peso p de la última
# arista añadida el árbol de expansión mínimo (aem), la lista t
# de nodos del grafo que están en el aem, la lista r de nodos del
# grafo que no están en el aem y el aem.
Estado = tuple[Peso, list[Vertice], list[Vertice], list[Arista]]

# inicial(g) es el estado inicial correspondiente al grafo g.
def inicial(g: Grafo) -> Estado:
    n, *ns = nodos(g)
    return (0, [n], ns, [])

# esFinal(e) se verifica si e es un estado final; es decir, si no
# queda ningún elemento en la lista de nodos sin colocar en el árbol de
```

```
# expansión mínimo.
def esFinal(e: Estado) -> bool:
    return e[2] == []

# sucesores(g, e) es la lista de los sucesores del estado e en el
# grafo g. Por ejemplo,
#    λ> sucesores(g1, (0,[1],[2,3,4,5],[]))
#    [(12,[2,1],[3,4,5],[(1,2,12)]),
#     (34,[3,1],[2,4,5],[(1,3,34)]),
#     (78,[5,1],[2,3,4],[(1,5,78)])]
def sucesores(g: Grafo, e: Estado) -> list[Estado]:
    (_,t,r,aem) = e
    return [(peso(x, y, g),
             [y] + t,
             [x for x in r if x != y],
             [((x,y),peso(x, y, g))] + aem)
            for x in t for y in r if aristaEn(g, (x, y))]

def prim(g: Grafo) -> Optional[list[Arista]]:
    r = buscaEscalada(lambda e: sucesores(g, e), esFinal, inicial(g))
    if r is None:
        return None
    return r[3]

# Verificación
# ============

def test_prim() -> None:
    assert prim(g1) == [((2,4),55),((1,3),34),((2,5),32),((1,2),12)]
    assert prim(g2) == [((2,5),32),((2,4),12),((1,2),13),((1,3),11)]
    assert prim(g3) == [((5,7),9),((2,3),7),((5,4),5),((6,5),3),((1,6),6),((1,2),
    assert prim(g4) == [((5,7),9),((5,4),5),((5,3),1),((6,5),3),((1,6),6),((1,2),
    print("Verificado")

# La verificación es
#    >>> test_prim()
#    Verificado
```

13.12. El problema del granjero mediante búsqueda en espacio de estado

13.12.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Un granjero está parado en un lado del río y con él tiene un lobo,
-- una cabra y una repollo. En el río hay un barco pequeño. El granjero
-- desea cruzar el río con sus tres posesiones. No hay puentes y en el
-- barco hay solamente sitio para el granjero y un artículo. Si deja
-- la cabra con la repollo sola en un lado del río la cabra comerá la
-- repollo. Si deja el lobo y la cabra en un lado, el lobo se comerá a
-- la cabra. ¿Cómo puede cruzar el granjero el río con los tres
-- artículos, sin que ninguno se coma al otro?
--
-- Para representar el problema se definen los siguientes tipos de dato:
-- + Orilla con dos constructores I y D que representan las orillas
--   izquierda y derecha, respectivamente.
-- + Estado que es una tupla que representa en qué orilla se encuentra
--   cada uno de los elementos (granjero, lobo, cabra, repollo). Por
--   ejemplo, (I,D,D,I) representa que el granjero está en la izquierda,
--   que el lobo está en la derecha, que la cabra está en la derecha y
--   el repollo está en la izquierda.
--
-- Usando el [procedimiento de búsqueda en profundidad](https://bit.ly/3NPI4qV),
-- definir la función
--    granjero :: [[Estado]]
-- tal que granjero son las soluciones del problema del granjero
-- mediante el patrón de búsqueda en espacio de estados. Por ejemplo,
--    λ> head granjero
--    [(I,I,I,I),(D,I,D,I),(I,I,D,I),(D,D,D,I),
--     (I,D,I,I),(D,D,I,D),(I,D,I,D),(D,D,D,D)]
--    λ> length granjero
--    2
-- ---------------------------------------------------------------------

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

module BEE_El_problema_del_granjero where
```

```
import BusquedaEnProfundidad (buscaProfundidad)
import Test.Hspec (Spec, hspec, it, shouldBe)

data Orilla = I | D
  deriving (Eq, Show)

type Estado = (Orilla,Orilla,Orilla,Orilla)

-- (seguro e) se verifica si el estado e es seguro; es decir, que no
-- puede estar en una orilla el lobo con la cabra sin el granjero ni la
-- cabra con el repollo sin el granjero. Por ejemplo,
--    seguro (I,D,D,I)  ==  False
--    seguro (D,D,D,I)  ==  True
--    seguro (D,D,I,I)  ==  False
--    seguro (I,D,I,I)  ==  True
seguro :: Estado -> Bool
seguro (g,l,c,r) = not (g /= c && (c == l || c == r))

-- (opuesta x) es la opuesta de la orilla x. Por ejemplo
--    opuesta I = D
opuesta :: Orilla -> Orilla
opuesta I = D
opuesta D = I

-- (sucesoresE e) es la lista de los sucesores seguros del estado e. Por
-- ejemplo,
--    sucesoresE (I,I,I,I)  ==  [(D,I,D,I)]
--    sucesoresE (D,I,D,I)  ==  [(I,I,D,I),(I,I,I,I)]
sucesoresE :: Estado -> [Estado]
sucesoresE e = [mov e | mov <- [m1,m2,m3,m4], seguro (mov e)]
  where m1 (g,l,c,r) = (opuesta g, l, c, r)
        m2 (g,l,c,r) = (opuesta g, opuesta l, c, r)
        m3 (g,l,c,r) = (opuesta g, l, opuesta c, r)
        m4 (g,l,c,r) = (opuesta g, l, c, opuesta r)

-- Nodo es el tipo de los nodos del espacio de búsqueda, donde un nodo
-- es una lista de estados
--    [e_n, ..., e_2, e_1]
-- tal que e_1 es el estado inicial y para cada i (2 <= i <= n), e_i es un
-- sucesor de e_(i-1).
```

```haskell
newtype Nodo = Nodo [Estado]
  deriving (Eq, Show)

-- inicial es el nodo inicial en el que todos están en la orilla
-- izquierda.
inicial :: Nodo
inicial = Nodo [(I,I,I,I)]

-- (esFinal n) se verifica si n es un nodo final; es decir, su primer
-- elemento es el estado final. Por ejemplo,
--    esFinal (Nodo [(D,D,D,D),(I,I,I,I)])  ==  True
--    esFinal (Nodo [(I,I,D,I),(I,I,I,I)])  ==  False
esFinal :: Nodo -> Bool
esFinal (Nodo (n:_)) = n == (D,D,D,D)

-- (sucesores n) es la lista de los sucesores del nodo n. Por ejemplo,
--    λ> sucesores (Nodo [(I,I,D,I),(D,I,D,I),(I,I,I,I)])
--    [Nodo [(D,D,D,I),(I,I,D,I),(D,I,D,I),(I,I,I,I)],
--     Nodo [(D,I,D,D),(I,I,D,I),(D,I,D,I),(I,I,I,I)]]
sucesores :: Nodo -> [Nodo]
sucesores (Nodo n@(e:es)) =
  [Nodo (e':n) | e' <- sucesoresE e, e' `notElem` es]

granjero :: [[Estado]]
granjero =
  [reverse es | (Nodo es) <- buscaProfundidad sucesores esFinal inicial]

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

spec :: Spec
spec = do
  it "e1" $
    head granjero `shouldBe`
    [(I,I,I,I),(D,I,D,I),(I,I,D,I),(D,D,D,I),
     (I,D,I,I),(D,D,I,D),(I,D,I,D),(D,D,D,D)]
  it "e2" $
```

```
    length granjero `shouldBe` 2

-- La verificación es
--    λ> verifica
--
--    e1
--    e2
--
--    Finished in 0.0008 seconds
--    2 examples, 0 failures
```

13.12.2. En Python

```
# ---------------------------------------------------------------------
# Un granjero está parado en un lado del río y con él tiene un lobo,
# una cabra y una repollo. En el río hay un barco pequeño. El granjero
# desea cruzar el río con sus tres posesiones. No hay puentes y en el
# barco hay solamente sitio para el granjero y un artículo. Si deja
# la cabra con la repollo sola en un lado del río la cabra comerá la
# repollo. Si deja el lobo y la cabra en un lado, el lobo se comerá a
# la cabra. ¿Cómo puede cruzar el granjero el río con los tres
# artículos, sin que ninguno se coma al otro?
#
# Para representar el problema se definen los siguientes tipos de dato:
# + Orilla con dos constructores I y D que representan las orillas
#   izquierda y derecha, respectivamente.
# + Estado que es una tupla que representa en qué orilla se encuentra
#   cada uno de los elementos (granjero, lobo, cabra, repollo). Por
#   ejemplo, (I,D,D,I) representa que el granjero está en la izquierda,
#   que el lobo está en la derecha, que la cabra está en la derecha y
#   el repollo está en la izquierda.
#
# Usando el [procedimiento de búsqueda en profundidad](https://bit.ly/3NPI4qV),
# definir la función
#    granjero : () -> list[list[Estado]]
# tal que granjero() son las soluciones del problema del granjero
# mediante el patrón de búsqueda en espacio de estados. Por ejemplo,
#    >>> granjero()
#    [[(I,I,I,I),(D,I,D,I),(I,I,D,I),(D,I,D,D),(I,I,I,D),(D,D,I,D),(I,D,I,D),(D,D
#     [(I,I,I,I),(D,I,D,I),(I,I,D,I),(D,D,D,I),(I,D,I,I),(D,D,I,D),(I,D,I,D),(D,D
```

```
# ---------------------------------------------------------------------

from enum import Enum

from src.BusquedaEnProfundidad import buscaProfundidad


class Orilla(Enum):
    I = 0
    D = 1

    def __repr__(self) -> str:
        return self.name

I = Orilla.I
D = Orilla.D

Estado = tuple[Orilla, Orilla, Orilla, Orilla]

# seguro(e) se verifica si el estado e es seguro; es decir, que no
# puede estar en una orilla el lobo con la cabra sin el granjero ni la
# cabra con el repollo sin el granjero. Por ejemplo,
#    seguro((I,D,D,I))  ==  False
#    seguro((D,D,D,I))  ==  True
#    seguro((D,D,I,I))  ==  False
#    seguro((I,D,I,I))  ==  True
def seguro(e: Estado) -> bool:
    (g,l,c,r) = e
    return not (g != c and c in {l, r})

# (opuesta x) es la opuesta de la orilla x. Por ejemplo
#    opuesta(I) == D
def opuesta(o: Orilla) -> Orilla:
    if o == I:
        return D
    return I

# sucesoresE(e) es la lista de los sucesores seguros del estado e. Por
# ejemplo,
#    sucesoresE((I,I,I,I))  ==  [(D,I,D,I)]
```

```python
#    sucesoresE((D,I,D,I))  ==  [(I,I,D,I),(I,I,I,I)]
def sucesoresE(e: Estado) -> list[Estado]:
    def mov(n: int, e: Estado) -> Estado:
        (g,l,c,r) = e
        if n == 1:
            return (opuesta(g), l, c, r)
        if n == 2:
            return (opuesta(g), opuesta(l), c, r)
        if n == 3:
            return (opuesta(g), l, opuesta(c), r)
        return (opuesta(g), l, c, opuesta(r))
    return [mov(n, e) for n in range(1, 5) if seguro(mov(n, e))]

# Nodo es el tipo de los nodos del espacio de búsqueda, donde un nodo
# es una lista de estados
#    [e_n, ..., e_2, e_1]
# tal que e_1 es el estado inicial y para cada i (2 <= i <= n), e_i es un
# sucesor de e_(i-1).
Nodo = list[Estado]

# inicial es el nodo inicial en el que todos están en la orilla
# izquierda.
inicial: Nodo = [(I,I,I,I)]

# esFinal(n) se verifica si n es un nodo final; es decir, su primer
# elemento es el estado final. Por ejemplo,
#    esFinal([(D,D,D,D),(I,I,I,I)])  ==  True
#    esFinal([(I,I,D,I),(I,I,I,I)])  ==  False
def esFinal(n: Nodo) -> bool:
    return n[0] == (D,D,D,D)

# sucesores(n) es la lista de los sucesores del nodo n. Por ejemplo,
#    >>> sucesores([(I,I,D,I),(D,I,D,I),(I,I,I,I)])
#    [[(D, D, D, I), (I, I, D, I), (D, I, D, I), (I, I, I, I)],
#     [(D, I, D, D), (I, I, D, I), (D, I, D, I), (I, I, I, I)]]
def sucesores(n: Nodo) -> list[Nodo]:
    e, *es = n
    return [[e1] + n for e1 in sucesoresE(e) if e1 not in es]

def granjero() -> list[list[Estado]]:
```

```
    return [list(reversed(es)) for es in buscaProfundidad(sucesores, esFinal, ini

# # Verificación
# # ============

def test_granjero() -> None:
    assert granjero() == \
        [[(I,I,I,I),(D,I,D,I),(I,I,D,I),(D,I,D,D),(I,I,I,D),(D,D,I,D),(I,D,I,D),(
         [(I,I,I,I),(D,I,D,I),(I,I,D,I),(D,D,D,I),(I,D,I,I),(D,D,I,D),(I,D,I,D),(
    print("Verificado")

# La verificación es
#    >>> test_granjero()
#    Verificado
```

13.13. El problema de las fichas mediante búsqueda en espacio de estado

13.13.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Para el problema de las fichas de orden (m,n) se considera un tablero
-- con m+n+1 cuadrados consecutivos.
--
-- Inicialmente, en cada uno de los m primeros cuadrados hay una ficha
-- blanca, a continuación un hueco y  en cada uno de los n últimos
-- cuadrados hay una ficha verde. El objetivo consiste en tener las
-- fichas verdes al principio y las blancas al final.
--
-- Por ejemplo, en el problema de las fichas de orden (3,3) el tablero
-- inicial es
--       +---+---+---+---+---+---+---+
--       | B | B | B |   | V | V | V |
--       +---+---+---+---+---+---+---+
-- y el final es
--       +---+---+---+---+---+---+---+
--       | V | V | V |   | B | B | B |
--       +---+---+---+---+---+---+---+
--
```

```
-- Los movimientos permitidos consisten en desplazar una ficha al hueco
-- saltando, como máximo, sobre otras dos.
--
-- Para representar el problema se definen los siguientes tipos de
-- datos:
-- + Ficha con tres constructores B, V y H que representan las fichas
--   blanca, verde y hueco, respectivamente.
--      data Ficha = B | V | H
--        deriving (Eq, Show)
-- + Tablero que es una lista de fichas que representa las fichas
--   colocadas en el tablero.
--      type Tablero = [Ficha]
-- + Estado representa los estados del espacio de búsqueda, donde un
--   estado es una lista de tableros [t_n, ..., t_2, t_1] tal que t_1 es
--   el tablero inicial y para cada i (2 <= i <= n), t_i es un sucesor
--   de t_(i-1).
--      newtype Estado = E [Tablero]
--        deriving (Eq, Show)
-- + Busqueda es un procedimiento de búsqueda
--      type Busqueda = (Estado -> [Estado]) ->
--                      (Estado -> Bool) ->
--                      Estado ->
--                      [Estado]
--
-- Además, se considera la heurística que para cada tablero vale la suma
-- de piezas blancas situadas a la izquierda de cada una de las piezas
-- verdes. Por  ejemplo, para el estado
--       +---+---+---+---+---+---+---+
--       | B | V | B |   | V | V | B |
--       +---+---+---+---+---+---+---+
-- su valor es 1+2+2 = 5. La heurística de un estado es la del primero
-- de sus tableros.
--
-- Usando los métodos de búsqueda estudiado en los ejercicios
-- anteriores, definir la función
--    fichas :: Busqueda -> Int -> Int -> [[Tablero]]
-- tal que (fichas b m n) es la lista de las soluciones del problema de
-- las fichas de orden (m,n) obtenidas mediante el procedimiento de
-- búsqueda b. Por ejemplo,
--    λ> head (fichas buscaProfundidad 2 2)
```

```haskell
--    [[B,B,H,V,V],[B,H,B,V,V],[H,B,B,V,V],[V,B,B,H,V],[V,B,H,B,V],[V,H,B,B,V],
--     [H,V,B,B,V],[B,V,H,B,V],[B,H,V,B,V],[H,B,V,B,V],[B,B,V,H,V],[B,B,V,V,H],
--     [B,H,V,V,B],[H,B,V,V,B],[V,B,H,V,B],[V,H,B,V,B],[H,V,B,V,B],[B,V,H,V,B],
--     [B,V,V,H,B],[H,V,V,B,B],[V,H,V,B,B],[V,V,H,B,B]]
--    λ> head (fichas buscaAnchura 2 2)
--    [[B,B,H,V,V],[B,B,V,V,H],[B,H,V,V,B],[B,V,V,H,B],[H,V,V,B,B],
--     [V,V,H,B,B]]
--    λ> head (fichas buscaPM 2 2)
--    [[B,B,H,V,V],[B,H,B,V,V],[B,V,B,H,V],[H,V,B,B,V],[V,H,B,B,V],
--     [V,V,B,B,H],[V,V,B,H,B],[V,V,H,B,B]]
--    λ> head (fichas buscaEscalada 2 2)
--    [[B,B,H,V,V],[B,H,B,V,V],[B,V,B,H,V],[H,V,B,B,V],[V,H,B,B,V],
--     [V,V,B,B,H],[V,V,B,H,B],[V,V,H,B,B]]
-- ---------------------------------------------------------------------

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

module BEE_El_problema_de_las_fichas where

import BusquedaEnProfundidad (buscaProfundidad)
import BusquedaEnAnchura (buscaAnchura)
import BusquedaPrimeroElMejor (buscaPM)
import BusquedaEnEscalada (buscaEscalada)
import Test.Hspec (Spec, hspec, it, shouldBe)

-- Representación del problema
-- ============================

data Ficha = B | V | H
  deriving (Eq, Show)

type Tablero = [Ficha]

-- (tableroInicial m n) representa el tablero inicial del problema de las fichas
-- de orden (m,n). Por ejemplo,
--    tableroInicial 2 3  ==  [B,B,H,V,V,V]
--    tableroInicial 3 2  ==  [B,B,B,H,V,V]
tableroInicial ::  Int -> Int -> Tablero
tableroInicial m n = replicate m B ++ [H] ++ replicate n V
```

```
-- (tableroFinal m n) representa el tablero final del problema de las fichas de
-- orden (m,n). Por ejemplo,
--    tableroFinal 2 3  ==  [V,V,V,H,B,B]
--    tableroFinal 3 2  ==  [V,V,H,B,B,B]
tableroFinal ::  Int -> Int -> Tablero
tableroFinal m n = replicate n V ++ [H] ++ replicate m B

-- (tablerosSucesores t) es la lista de los sucesores del tablero t. Por
-- ejemplo,
--    λ> tablerosSucesores [V,B,H,V,V,B]
--    [[V,H,B,V,V,B],[H,B,V,V,V,B],[V,B,V,H,V,B],[V,B,V,V,H,B],
--     [V,B,B,V,V,H]]
--    λ> tablerosSucesores [B,B,B,H,V,V,V]
--    [[B,B,H,B,V,V,V],[B,H,B,B,V,V,V],[H,B,B,B,V,V,V],
--     [B,B,B,V,H,V,V],[B,B,B,V,V,H,V],[B,B,B,V,V,V,H]]
tablerosSucesores :: Tablero -> [Tablero]
tablerosSucesores t =
  [intercambia i j t | i <- [j-1,j-2,j-3,j+1,j+2,j+3]
                     , 0 <= i, i < n]
  where j = posicionHueco t
        n = length t

-- (posicionHueco t) es la posición del hueco en el tablero t. Por
-- ejemplo,
--    posicionHueco (tableroInicial 3 2)  ==  3
posicionHueco :: Tablero -> Int
posicionHueco t = length (takeWhile (/=H) t)

-- (intercambia xs i j) es la lista obtenida intercambiando los
-- elementos de xs en las posiciones i y j. Por ejemplo,
--    intercambia 2 6 [0..9]  ==  [0,1,6,3,4,5,2,7,8,9]
--    intercambia 6 2 [0..9]  ==  [0,1,6,3,4,5,2,7,8,9]
intercambia :: Int -> Int -> [a] -> [a]
intercambia i j xs = concat [xs1,[x2],xs2,[x1],xs3]
  where (xs1,x1,xs2,x2,xs3) = divide (min i j) (max i j) xs

-- (divide xs i j) es la tupla (xs1,x1,xs2,x2,xs3) tal que xs1 son los
-- elementos de xs cuya posición es menor que i, x1 es el elemento de xs
-- en la posición i, xs2 son los elementos de xs cuya posición es mayor
-- que i y menor que j, x2 es el elemento de xs en la posición j y xs3
```

```haskell
-- son los elementos de xs cuya posición es mayor que j (suponiendo que
-- i < j). Por ejemplo,
--    divide 2 6 [0..9]  ==  ([0,1],2,[3,4,5],6,[7,8,9])
divide :: Int -> Int -> [a] -> ([a],a,[a],a,[a])
divide i j xs = (xs1,x1,xs2,x2,xs3)
  where (xs1,x1:ys)  = splitAt i xs
        (xs2,x2:xs3) = splitAt (j - i - 1) ys

newtype Estado = E [Tablero]
  deriving (Eq, Show)

-- (inicial m n) representa el estado inicial del problema de las fichas
-- de orden (m,n). Por ejemplo,
--    inicial 2 3  ==  E [[B,B,H,V,V,V]]
--    inicial 3 2  ==  E [[B,B,B,H,V,V]]
inicial :: Int -> Int -> Estado
inicial m n = E [tableroInicial m n]

-- (esFinal m n e) se verifica si e es un estado final del problema de las
-- fichas de orden (m,n). Por ejemplo,
--    λ> esFinal 2 1 (E [[V,H,B,B],[V,B,B,H],[H,B,B,V],[B,B,H,V]])
--    True
--    λ> esFinal 2 1 (E [[V,B,B,H],[H,B,B,V],[B,B,H,V]])
--    False
esFinal :: Int -> Int -> Estado -> Bool
esFinal m n (E (e:_)) = e == tableroFinal m n

-- (sucesores n) es la lista de los sucesores del estado n. Por ejemplo,
--    λ> sucesores (E [[H,B,B,V],[B,B,H,V]])
--    [E [[B,H,B,V],[H,B,B,V],[B,B,H,V]],
--     E [[V,B,B,H],[H,B,B,V],[B,B,H,V]]]
--    λ> sucesores (E [[B,H,B,V],[H,B,B,V],[B,B,H,V]])
--    [E [[B,V,B,H],[B,H,B,V],[H,B,B,V],[B,B,H,V]]]
sucesores :: Estado -> [Estado]
sucesores (E e@(t:ts)) =
  [E (t':e) | t' <- tablerosSucesores t,
              t' `notElem` ts]

-- Heurística
-- ==========
```

```
-- (heuristicaT t) es la heurística del tablero t. Por ejemplo,
--    heuristicaT [B,V,B,H,V,V,B] == 5
heuristicaT :: Tablero -> Int
heuristicaT []     = 0
heuristicaT (V:xs) = heuristicaT xs
heuristicaT (H:xs) = heuristicaT xs
heuristicaT (B:xs) = heuristicaT xs + length (filter (==V) xs)

-- (heuristica e) es la heurística del primer tablero del estado e. Por
-- ejemplo,
--    heuristica (E [[H,B,B,V],[B,B,H,V]])            ==  2
--    heuristica (E [[V,B,B,H],[H,B,B,V],[B,B,H,V]])  ==  0
heuristica :: Estado -> Int
heuristica (E (t:_)) = heuristicaT t

-- Estado es un subtipo de Ord de forma que un estado es menor o igual
-- que otro si su heurística lo es.
instance Ord Estado where
  e1 <= e2 = heuristica e1 <= heuristica e2

-- Solución por búsqueda
-- =====================

type Busqueda = (Estado -> [Estado]) ->
                (Estado -> Bool) ->
                Estado ->
                [Estado]

fichas :: Busqueda -> Int -> Int -> [[Tablero]]
fichas b m n =
  [reverse es | E es <- b sucesores (esFinal m n) (inicial m n)]

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

spec :: Spec
```

```haskell
spec = do
  it "e1" $
    head (fichas buscaProfundidad 2 2) `shouldBe`
    [[B,B,H,V,V],[B,H,B,V,V],[H,B,B,V,V],[V,B,B,H,V],[V,B,H,B,V],[V,H,B,B,V],
     [H,V,B,B,V],[B,V,H,B,V],[B,H,V,B,V],[H,B,V,B,V],[B,B,V,H,V],[B,B,V,V,H],
     [B,H,V,V,B],[H,B,V,V,B],[V,B,H,V,B],[V,H,B,V,B],[H,V,B,V,B],[B,V,H,V,B],
     [B,V,V,H,B],[H,V,V,B,B],[V,H,V,B,B],[V,V,H,B,B]]
  it "e2" $
    head (fichas buscaAnchura 2 2) `shouldBe`
    [[B,B,H,V,V],[B,B,V,V,H],[B,H,V,V,B],[B,V,V,H,B],[H,V,V,B,B],[V,V,H,B,B]]
  it "e3" $
    head (fichas buscaPM 2 2) `shouldBe`
    [[B,B,H,V,V],[B,H,B,V,V],[B,V,B,H,V],[H,V,B,B,V],[V,H,B,B,V],[V,V,B,B,H],
     [V,V,B,H,B],[V,V,H,B,B]]
  it "e4" $
    head (fichas buscaEscalada 2 2) `shouldBe`
    [[B,B,H,V,V],[B,H,B,V,V],[B,V,B,H,V],[H,V,B,B,V],[V,H,B,B,V],[V,V,B,B,H],
     [V,V,B,H,B],[V,V,H,B,B]]

-- La verificación es
--    λ> verifica
--
--    e1
--    e2
--    e3
--    e4
--
--    Finished in 0.0055 seconds
--    4 examples, 0 failures
```

13.13.2. En Python

```python
# ---------------------------------------------------------------------
# Para el problema de las fichas de orden (m,n) se considera un tablero
# con m+n+1 cuadrados consecutivos.
#
# Inicialmente, en cada uno de los m primeros cuadrados hay una ficha
# blanca, a continuación un hueco y  en cada uno de los n últimos
# cuadrados hay una ficha verde. El objetivo consiste en tener las
# fichas verdes al principio y las blancas al final.
```

```
#
# Por ejemplo, en el problema de las fichas de orden (3,3) el tablero
# inicial es
#       +---+---+---+---+---+---+---+
#       | B | B | B |   | V | V | V |
#       +---+---+---+---+---+---+---+
# y el final es
#       +---+---+---+---+---+---+---+
#       | V | V | V |   | B | B | B |
#       +---+---+---+---+---+---+---+
#
# Los movimientos permitidos consisten en desplazar una ficha al hueco
# saltando, como máximo, sobre otras dos.
#
# Además, se considera la heurística que para cada tablero vale la suma
# de piezas blancas situadas a la izquierda de cada una de las piezas
# verdes. Por  ejemplo, para el estado
#       +---+---+---+---+---+---+---+
#       | B | V | B |   | V | V | B |
#       +---+---+---+---+---+---+---+
# su valor es 1+2+2 = 5. La heurística de un estado es la del primero
# de sus tableros.
#
# Para representar el problema se definen los siguientes tipos de
# datos:
# + Ficha con tres constructores B, V y H que representan las fichas
#   blanca, verde y hueco, respectivamente.
#      class Ficha(Enum):
#          B = 0
#          V = 1
#          H = 2
#
#          def __repr__(self) -> str:
#              return self.name
#
#      B = Ficha.B
#      V = Ficha.V
#      H = Ficha.H
# + Tablero que es una lista de fichas que representa las fichas
#   colocadas en el tablero.
```

```
#       Tablero = list[Ficha]
# + Estado representa los estados del espacio de búsqueda, donde un
#   estado es una lista de tableros [t_n, ..., t_2, t_1] tal que t_1 es
#   el tablero inicial y para cada i (2 <= i <= n), t_i es un sucesor
#   de t_(i-1).
#       class Estado(list[Tablero]):
#           def __lt__(self, other):
#               return heuristicaT(self[0]) < heuristicaT(other[0])
# + Busqueda es un procedimiento de búsqueda
#       Busqueda = Callable[[Callable[[Estado], list[Estado]],
#                            Callable[[Estado], bool],
#                            Estado],
#                           Optional[Estado]]
#
# Usando los métodos de búsqueda estudiado en los ejercicios
# anteriores, definir la función
#    fichas :: Busqueda -> Int -> Int -> [[Tablero]]
# tal que fichas(b, m, n) es la lista de las soluciones del problema de
# las fichas de orden (m,n) obtenidas mediante el procedimiento de
# búsqueda b. Por ejemplo,
#    >>> fichas(buscaProfundidad1, 2, 2)
#    [[B,B,H,V,V],[H,B,B,V,V],[B,H,B,V,V],[B,V,B,H,V],[H,V,B,B,V].
#     [B,V,H,B,V],[B,V,V,B,H],[B,H,V,B,V],[B,B,V,H,V],[H,B,V,B,V],
#     [V,B,H,B,V],[V,B,V,B,H],[V,H,V,B,B],[V,B,V,H,B],[H,B,V,V,B],
#     [B,H,V,V,B],[B,V,V,H,B],[H,V,V,B,B],[V,V,H,B,B]]
#    >>> fichas(buscaAnchura1, 2, 2)
#    [[B,B,H,V,V],[H,B,B,V,V],[V,B,B,H,V],[V,H,B,B,V],[V,V,B,B,H],
#     [V,V,H,B,B]]
#    >>> fichas(buscaPM, 2, 2)
#    [[B,B,H,V,V],[H,B,B,V,V],[V,B,B,H,V],[V,B,B,V,H],[V,H,B,V,B],
#     [V,V,B,H,B],[V,V,H,B,B]]
#    >>> fichas(buscaEscalada, 2, 2)
#    [[B,B,H,V,V],[B,H,B,V,V],[H,B,B,V,V],[V,B,B,H,V],[V,B,B,V,H],
#     [V,H,B,V,B],[V,V,B,H,B],[V,V,H,B,B]]
# ---------------------------------------------------------------------

from enum import Enum
from functools import partial
from typing import Callable, Optional
```

```python
from src.BusquedaEnAnchura import buscaAnchura1
from src.BusquedaEnEscalada import buscaEscalada
from src.BusquedaEnProfundidad import buscaProfundidad1
from src.BusquedaPrimeroElMejor import buscaPM

# Representación del problema
# ===========================

class Ficha(Enum):
    B = 0
    V = 1
    H = 2

    def __repr__(self) -> str:
        return self.name

B = Ficha.B
V = Ficha.V
H = Ficha.H

Tablero = list[Ficha]

# tableroInicial(m, n) representa el tablero inicial del problema de las fichas
# de orden (m,n). Por ejemplo,
#    tableroInicial(2, 3)  ==  [B,B,H,V,V,V]
#    tableroInicial(3, 2)  ==  [B,B,B,H,V,V]
def tableroInicial(m: int, n: int) -> Tablero:
    return [B]*m + [H] + [V]*n

# tableroFinal(m, n) representa el tablero final del problema de las fichas de
# orden (m,n). Por ejemplo,
#    tableroFinal(2, 3)  ==  [V,V,V,H,B,B]
#    tableroFinal(3, 2)  ==  [V,V,H,B,B,B]
def tableroFinal(m: int, n: int) -> Tablero:
    return [V]*n + [H] + [B]*m

# posicionHueco(t) es la posición del hueco en el tablero t. Por
# ejemplo,
#    posicionHueco(tableroInicial(3, 2))  ==  3
def posicionHueco(t: Tablero) -> int:
```

```
    return t.index(H)

# intercambia(xs, i, j) es la lista obtenida intercambiando los
# elementos de xs en las posiciones i y j. Por ejemplo,
#    intercambia(1, 3, tableroInicial(3, 2))  ==  [B, H, B, B, V, V]
def intercambia(i: int, j: int, t: Tablero) -> Tablero:
    t1 = t.copy()
    t1[i], t1[j] = t1[j], t1[i]
    return t1

# tablerosSucesores(t) es la lista de los sucesores del tablero t. Por
# ejemplo,
#    >>> tablerosSucesores([V,B,H,V,V,B])
#    [[V,H,B,V,V,B],[H,B,V,V,V,B],[V,B,V,H,V,B],[V,B,V,V,H,B],
#     [V,B,B,V,V,H]]
#    >>> tablerosSucesores([B,B,B,H,V,V,V])
#    [[B,B,H,B,V,V,V],[B,H,B,B,V,V,V],[H,B,B,B,V,V,V],
#     [B,B,B,V,H,V,V],[B,B,B,V,V,H,V],[B,B,B,V,V,V,H]]
def tablerosSucesores(t: Tablero) -> list[Tablero]:
    j = posicionHueco(t)
    n = len(t)
    return [intercambia(i, j, t)
            for i in [j-1,j-2,j-3,j+1,j+2,j+3]
            if 0 <= i < n]

# Heurística
# ==========

# heuristicaT(t) es la heurística del tablero t. Por ejemplo,
#    heuristicaT([B,V,B,H,V,V,B]) == 5
def heuristicaT(t: Tablero) -> int:
    if not t:
        return 0
    f, *fs = t
    if f in {V, H}:
        return heuristicaT(fs)
    return heuristicaT(fs) + len([x for x in fs if x == V])

class Estado(list[Tablero]):
    def __lt__(self, e: list[Tablero]) -> bool:
```

```python
        return heuristicaT(self[0]) < heuristicaT(e[0])

# inicial(m, n) representa el estado inicial del problema de las fichas
# de orden (m,n). Por ejemplo,
#    inicial(2, 3)  ==  [[B,B,H,V,V,V]]
#    inicial(3, 2)  ==  [[B,B,B,H,V,V]]
def inicial(m: int, n: int) -> Estado:
    return Estado([tableroInicial(m, n)])

# esFinal(m, n, e) se verifica si e es un estado final del problema de las
# fichas de orden (m,n). Por ejemplo,
#    >>> esFinal(2, 1, [[V,H,B,B],[V,B,B,H],[H,B,B,V],[B,B,H,V]])
#    True
#    >>> esFinal(2, 1, [[V,B,B,H],[H,B,B,V],[B,B,H,V]])
#    False
def esFinal(m: int, n: int, e: Estado) -> bool:
    return e[0] == tableroFinal(m, n)

# (sucesores n) es la lista de los sucesores del estado n. Por ejemplo,
#    >>> sucesores([[H,B,B,V],[B,B,H,V]])
#    [[[B,H,B,V],[H,B,B,V],[B,B,H,V]],
#     [[V,B,B,H],[H,B,B,V],[B,B,H,V]]]
#    >>> sucesores([[B,H,B,V],[H,B,B,V],[B,B,H,V]])
#    [[[B,V,B,H],[B,H,B,V],[H,B,B,V],[B,B,H,V]]]
def sucesores(e: Estado) -> list[Estado]:
    t, *ts = e
    return [Estado([t1] + e) for t1 in tablerosSucesores(t) if t1 not in ts]

# Solución por búsqueda
# =====================

Busqueda = Callable[[Callable[[Estado], list[Estado]],
                     Callable[[Estado], bool],
                     Estado],
                    Optional[Estado]]

def fichas(b: Busqueda, m: int, n: int) -> Optional[list[Tablero]]:
    r = partial(b, sucesores, lambda e: esFinal(m, n, e), inicial(m, n))()
    if r is None:
        return None
```

```
    return [list(reversed(es)) for es in r]

# Verificación
# ============

def test_fichas() -> None:
    assert fichas(buscaProfundidad1, 1, 2) == \
        [[B, H, V, V], [B, V, V, H], [H, V, V, B], [V, V, H, B]]
    assert fichas(buscaAnchura1, 1, 2) == \
        [[B, H, V, V], [B, V, V, H], [H, V, V, B], [V, V, H, B]]
    assert fichas(buscaPM, 1, 2) == \
        [[B, H, V, V], [B, V, H, V], [H, V, B, V], [V, V, B, H],
         [V, V, H, B]]
    assert fichas(buscaEscalada, 1, 2) == \
        [[B, H, V, V], [H, B, V, V], [V, B, H, V], [V, H, B, V],
         [V, V, B, H], [V, V, H, B]]
    print("Verificado")

# La verificación es
#    >>> test_fichas()
#    Verificado
```

13.14. El problema del calendario mediante búsqueda en espacio de estado

13.14.1. En Haskell

```
-- ---------------------------------------------------------------------
-- El problema del calendario, para una competición deportiva en la que
-- se enfrentan n participantes, consiste en elaborar un calendario de
-- forma que:
--    + el campeonato dure n-1 días,
--    + cada participante juegue exactamente un partido diario y
--    + cada participante juegue exactamente una vez con cada adversario.
-- Por ejemplo, con 8 participantes una posible solución es
--      | 1 2 3 4 5 6 7
--    --+--------------
--    1 | 2 3 4 5 6 7 8
--    2 | 1 4 3 6 5 8 7
```

```
--    3 | 4 1 2 7 8 5 6
--    4 | 3 2 1 8 7 6 5
--    5 | 6 7 8 1 2 3 4
--    6 | 5 8 7 2 1 4 3
--    7 | 8 5 6 3 4 1 2
--    8 | 7 6 5 4 3 2 1
-- donde las filas indican los jugadores y las columnas los días; es
-- decir, el elemento (i,j) indica el adversario del jugador i el día j;
-- por ejemplo, el adversario del jugador 2 el 4ª día es el jugador 6.
--
-- Para representar el problema se define el tipo Calendario como
-- matrices de enteros
--    type Calendario = Matrix Int
--
-- Usando el [procedimiento de búsqueda en profundidad](https://bit.ly/3NPI4qV),
-- definir la función
--    calendario :: Int -> [Calendario]
-- tal que (calendario n) son las soluciones del problema del calendario,
-- con n participantes, mediante el patrón de búsqueda em
-- profundidad. Por ejemplo,
--    λ> head (calendario 6)
--    ┌           ┐
--    │ 2 3 4 5 6 │
--    │ 1 4 5 6 3 │
--    │ 5 1 6 4 2 │
--    │ 6 2 1 3 5 │
--    │ 3 6 2 1 4 │
--    │ 4 5 3 2 1 │
--    └           ┘
--
--    λ> length (calendario 6)
--    720
--    λ> length (calendario 5)
--    0
-- ---------------------------------------------------------------------

module El_problema_del_calendario_mediante_busqueda_en_espacio_de_estado where

import BusquedaEnProfundidad (buscaProfundidad)
import Data.Matrix (Matrix, (!), nrows, zero, setElem, toLists)
```

```haskell
import Data.List ((\\))
import Test.Hspec (Spec, hspec, it, shouldBe)

type Calendario = Matrix Int

-- (inicial n) es el estado inicial para el problema del calendario con
-- n participantes; es decir, una matriz de n fila y n-1 columnas con
-- todos sus elementos iguales a 0. Por ejemplo,
--    λ> inicial 4
--    ┌       ┐
--    │ 0 0 0 │
--    │ 0 0 0 │
--    │ 0 0 0 │
--    │ 0 0 0 │
--    └       ┘
inicial :: Int -> Calendario
inicial n = zero n (n-1)

-- (huecos c) es la lista de las posiciones de c cuyo valor es 0.
huecos :: Calendario -> [(Int, Int)]
huecos c = [(i,j) | i <- [1..n], j <- [1..n-1], c!(i,j) == 0]
  where n = nrows c

-- (sucesores c) es la lista de calendarios obtenidos poniendo en el
-- lugar del primer elemento nulo de c uno de los posibles jugadores de
-- forma que se cumplan las condiciones del problema. Por ejemplo,
--    λ> sucesores (inicial 4)
--    [┌       ┐  ┌       ┐  ┌       ┐
--     │ 2 0 0 │  │ 3 0 0 │  │ 4 0 0 │
--     │ 1 0 0 │  │ 0 0 0 │  │ 0 0 0 │
--     │ 0 0 0 │  │ 1 0 0 │  │ 0 0 0 │
--     │ 0 0 0 │  │ 0 0 0 │  │ 1 0 0 │
--     └       ┘, └       ┘, └       ┘]
--    λ> sucesores (fromLists [[2,3,0],[1,0,0],[0,1,0],[0,0,0]])
--    [┌       ┐
--     │ 2 3 4 │
--     │ 1 0 0 │
--     │ 0 1 0 │
--     │ 0 0 1 │
--     └       ┘]
```

```
--    λ> sucesores (fromLists [[2,3,4],[1,0,0],[0,1,0],[0,0,1]])
--    [┌       ┐
--     │ 2 3 4 │
--     │ 1 4 0 │
--     │ 0 1 0 │
--     │ 0 2 1 │
--     └       ┘]
sucesores :: Calendario -> [Calendario]
sucesores c =
  [setElem i (k,j) (setElem k (i,j) c) |
   k <- [1..n] \\ (i : [c!(k,j) | k <- [1..i-1]] ++
                       [c!(i,k) | k <- [1..j-1]]),
   c!(k,j) == 0]
  where
    n = nrows c
    (i,j) = head (huecos c)

-- (esFinal c) se verifica si c un estado final para el problema
-- del calendario con n participantes; es decir, no queda en c ningún
-- elemento igual a 0. Por ejemplo,
--    λ> esFinal (fromLists [[2,3,4],[1,4,3],[4,1,2],[3,2,1]])
--    True
--    λ> esFinal (fromLists [[2,3,4],[1,4,3],[4,1,2],[3,2,0]])
--    False
esFinal :: Calendario -> Bool
esFinal c = null (huecos c)

calendario :: Int -> [Calendario]
calendario n = buscaProfundidad sucesores esFinal (inicial n)

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

spec :: Spec
spec = do
  it "e1" $
    toLists (head (calendario 6)) `shouldBe`
```

```
    [[2,3,4,5,6],[1,4,5,6,3],[5,1,6,4,2],[6,2,1,3,5],[3,6,2,1,4],[4,5,3,2,1]]
  it "e2" $
    length (calendario 6) `shouldBe` 720
  it "e3" $
    length (calendario 5) `shouldBe` 0

-- La verificación es
--    λ> verifica
--
--    e1
--    e2
--    e3
--
--    Finished in 0.2580 seconds
--    3 examples, 0 failures
```

13.14.2. En Python

```
# ---------------------------------------------------------------------
# El problema del calendario, para una competición deportiva en la que
# se enfrentan n participantes, consiste en elaborar un calendario de
# forma que:
#    + el campeonato dure n-1 días,
#    + cada participante juegue exactamente un partido diario y
#    + cada participante juegue exactamente una vez con cada adversario.
# Por ejemplo, con 8 participantes una posible solución es
#      | 1 2 3 4 5 6 7
#    --+--------------
#    1 | 2 3 4 5 6 7 8
#    2 | 1 4 3 6 5 8 7
#    3 | 4 1 2 7 8 5 6
#    4 | 3 2 1 8 7 6 5
#    5 | 6 7 8 1 2 3 4
#    6 | 5 8 7 2 1 4 3
#    7 | 8 5 6 3 4 1 2
#    8 | 7 6 5 4 3 2 1
# donde las filas indican los jugadores y las columnas los días; es
# decir, el elemento (i,j) indica el adversario del jugador i el día j;
# por ejemplo, el adversario del jugador 2 el 4ª día es el jugador 6.
#
```

```
# Para representar el problema se define el tipo Calendario como
# matrices de enteros.
#
# Usando el [procedimiento de búsqueda en profundidad](https://bit.ly/3NPI4qV),
# definir la función
#    calendario : (int) -> [Calendario]
# tal que calendario(n) son las soluciones del problema del calendario,
# con n participantes, mediante el patrón de búsqueda em
# profundidad. Por ejemplo,
#    >>> calendario(6)[0]
#    array([[6, 5, 4, 3, 2],
#           [5, 4, 3, 6, 1],
#           [4, 6, 2, 1, 5],
#           [3, 2, 1, 5, 6],
#           [2, 1, 6, 4, 3],
#           [1, 3, 5, 2, 4]])
#    >>> len(calendario(6))
#    720
#    >>> len(calendario(5))
#    0
# ---------------------------------------------------------------------

from copy import deepcopy
from typing import Optional

import numpy as np
import numpy.typing as npt

from src.BusquedaEnProfundidad import buscaProfundidad

Calendario = npt.NDArray[np.int_]

# inicial(n) es el estado inicial para el problema del calendario con
# n participantes; es decir, una matriz de n fila y n-1 columnas con
# todos sus elementos iguales a 0. Por ejemplo,
#    >>> inicial(4)
#    array([[0, 0, 0],
#           [0, 0, 0],
#           [0, 0, 0],
#           [0, 0, 0]])
```

```python
def inicial(n: int) -> Calendario:
    return np.zeros((n, n - 1), dtype=int)

# primerHueco(c) es la posición del primer elemento cuyo valor es 0. Si
# todos los valores son distintos de 0, devuelve (-1,-1). Por ejemplo,
#    primerHueco(np.array([[1,2,3],[4,5,0],[7,0,0]])) == (1, 2)
#    primerHueco(np.array([[1,2,3],[4,5,6],[7,8,0]])) == (2, 2)
#    primerHueco(np.array([[1,2,3],[4,5,6],[7,8,9]])) == (-1, -1)
def primerHueco(c: Calendario) -> tuple[int, int]:
    (n, m) = c.shape
    for i in range(0, n):
        for j in range(0, m):
            if c[i,j] == 0:
                return (i, j)
    return (-1, -1)

# libres(c, i, j) es la lista de valores que que pueden poner en la
# posición (i,j) del calendario c. Por ejemplo,
#    libres(np.array([[0,0,0],[0,0,0],[0,0,0],[0,0,0]]),0,0) == [2, 3, 4]
#    libres(np.array([[2,0,0],[1,0,0],[0,0,0],[0,0,0]]),0,1) == [3, 4]
#    libres(np.array([[2,3,0],[1,0,0],[0,1,0],[0,0,0]]),0,2) == [4]
#    libres(np.array([[2,3,4],[1,0,0],[0,1,0],[0,0,1]]),1,1) == [4]
#    libres(np.array([[2,3,4],[1,4,0],[0,1,0],[0,2,1]]),1,2) == [3]
def libres(c: Calendario, i: int, j: int) -> list[int]:
    n = c.shape[0]
    return list(set(range(1, n + 1))
                - {i + 1}
                - set(c[i])
                - set(c[:,j]))

# setElem(k, i, j, c) es el calendario obtenido colocando en c el valor
# k en la posición (i,j).
#    >>> setElem(7,1,2,np.array([[1,2,3],[4,5,0],[0,0,0]]))
#    array([[1, 2, 3],
#           [4, 5, 7],
#           [0, 0, 0]])
def setElem(k: int, i: int, j: int, c: Calendario) -> Calendario:
    _c = deepcopy(c)
    _c[i, j] = k
    return _c
```

```python
# sucesores(c) es la lista de calendarios obtenidos poniendo en el
# lugar del primer elemento nulo de c uno de los posibles jugadores de
# forma que se cumplan las condiciones del problema. Por ejemplo,
#    >>> sucesores(np.array([[0,0,0],[0,0,0],[0,0,0],[0,0,0]]))
#    [array([[2,0,0], [1,0,0], [0,0,0], [0,0,0]]),
#     array([[3,0,0], [0,0,0], [1,0,0], [0,0,0]]),
#     array([[4,0,0], [0,0,0], [0,0,0], [1,0,0]])]
#    >>> sucesores(np.array([[2,0,0],[1,0,0],[0,0,0],[0,0,0]]))
#    [array([[2,3,0], [1,0,0], [0,1,0], [0,0,0]]),
#     array([[2,4,0], [1,0,0], [0,0,0], [0,1,0]])]
#    >>> sucesores(np.array([[2,3,0],[1,0,0],[0,1,0],[0,0,0]]))
#    [array([[2,3,4], [1,0,0], [0,1,0], [0,0,1]])]
#    >>> sucesores(np.array([[2,3,4],[1,0,0],[0,1,0],[0,0,1]]))
#    [array([[2,3,4], [1,4,0], [0,1,0], [0,2,1]])]
#    >>> sucesores(np.array([[2,3,4],[1,4,0],[0,1,0],[0,2,1]]))
#    [array([[2,3,4], [1,4,3], [0,1,2], [0,2,1]])]
#    >>> sucesores(np.array([[2,3,4],[1,4,3],[0,1,2],[0,2,1]]))
#    [array([[2,3,4], [1,4,3], [4,1,2], [3,2,1]])]
#    >>> sucesores(np.array([[2,3,4],[1,4,3],[4,1,2],[3,2,1]]))
#    []
def sucesores(c: Calendario) -> list[Calendario]:
    n = c.shape[0]
    (i, j) = primerHueco(c)
    return [setElem(i+1, k-1, j, setElem(k, i, j, c))
            for k in libres(c, i, j)]

# esFinal(c) se verifica si c un estado final para el problema
# del calendario con n participantes; es decir, no queda en c ningún
# elemento igual a 0. Por ejemplo,
#    >>> esFinal(np.array([[2,3,4],[1,4,3],[4,1,2],[3,2,1]]))
#    True
#    >>> esFinal(np.array([[2,3,4],[1,4,3],[4,1,2],[3,2,0]]))
#    False
def esFinal(c: Calendario) -> bool:
    return primerHueco(c) == (-1, -1)

def calendario(n: int) -> list[Calendario]:
    return buscaProfundidad(sucesores, esFinal, inicial(n))
```

```
# Verificación
# ============

def test_calendario() -> None:
    def filas(p: Calendario) -> list[list[int]]:
        return p.tolist()

    assert filas(calendario(6)[0]) == \
        [[6, 5, 4, 3, 2],
         [5, 4, 3, 6, 1],
         [4, 6, 2, 1, 5],
         [3, 2, 1, 5, 6],
         [2, 1, 6, 4, 3],
         [1, 3, 5, 2, 4]]
    assert len(calendario(6)) == 720
    assert len(calendario(5)) == 0
    print("Verificado")
```

13.15. El problema del dominó

13.15.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Las fichas del dominó se pueden representar por pares de números
-- enteros. El problema del dominó consiste en colocar todas las fichas
-- de una lista dada de forma que el segundo número de cada ficha
-- coincida con el primero de la siguiente.
--
-- Definir, mediante búsqueda en espacio de estados, la función
--    domino :: [(Int,Int)] -> [[(Int,Int)]]
-- tal que (domino fs) es la lista de las soluciones del problema del
-- dominó correspondiente a las fichas fs. Por ejemplo,
--    λ> domino [(1,2),(2,3),(1,4)]
--    [[(4,1),(1,2),(2,3)],[(3,2),(2,1),(1,4)]]
--    λ> domino [(1,2),(1,1),(1,4)]
--    [[(4,1),(1,1),(1,2)],[(2,1),(1,1),(1,4)]]
--    λ> domino [(1,2),(3,4),(2,3)]
--    [[(1,2),(2,3),(3,4)],[(4,3),(3,2),(2,1)]]
--    λ> domino [(1,2),(2,3),(5,4)]
--    []
```

```haskell
-- ---------------------------------------------------------------------

module El_problema_del_domino where

import BusquedaEnProfundidad (buscaProfundidad)
import Data.List (delete)
import Test.Hspec (Spec, hspec, it, shouldBe)

-- Las fichas son pares de números enteros.
type Ficha  = (Int,Int)

-- Un problema está definido por la lista de fichas que hay que colocar
type Problema = [Ficha]

-- Los estados son los pares formados por la listas sin colocar y las
-- colocadas.
type Estado = ([Ficha],[Ficha])

-- (inicial p) es el estado inicial del problema p. Por ejemplo,
--    λ> inicial [(1,2),(2,3),(1,4)]
--    ([(1,2),(2,3),(1,4)],[])
inicial :: Problema -> Estado
inicial p = (p,[])

-- (esFinal e) se verifica si e es un estado final. Por ejemplo,
--    λ> esFinal ([], [(4,1),(1,2),(2,3)])
--    True
--    λ> esFinal ([(2,3)], [(4,1),(1,2)])
--    False
esFinal :: Estado -> Bool
esFinal = null . fst

-- (sucesores e) es la lista de los sucesores del estado e. Por ejemplo,
--    λ> sucesores ([(1,2),(2,3),(1,4)],[])
--    [([(2,3),(1,4)],[(1,2)]),
--     ([(1,2),(1,4)],[(2,3)]),
--     ([(1,2),(2,3)],[(1,4)]),
--     ([(2,3),(1,4)],[(2,1)]),
--     ([(1,2),(1,4)],[(3,2)]),
--     ([(1,2),(2,3)],[(4,1)])]
```

```haskell
--    λ> sucesores ([(2,3),(1,4)],[(1,2)])
--    [([(2,3)],[(4,1),(1,2)])]
--    λ> sucesores ([(2,3),(1,4)],[(2,1)])
--    [([(1,4)],[(3,2),(2,1)])]
sucesores :: Estado -> [Estado]
sucesores (fs,[]) =
  [(delete (a,b) fs, [(a,b)]) | (a,b) <- fs, a /= b] ++
  [(delete (a,b) fs, [(b,a)]) | (a,b) <- fs]
sucesores (fs,e@((x,_):_)) =
  [(delete (u,v) fs,(u,v):e) | (u,v) <- fs, u /= v, v == x] ++
  [(delete (u,v) fs,(v,u):e) | (u,v) <- fs, u /= v, u == x] ++
  [(delete (u,v) fs,(u,v):e) | (u,v) <- fs, u == v, u == x]

-- (soluciones p) es la lista de las soluciones del problema p. Por
-- ejemplo,
--    λ> soluciones [(1,2),(2,3),(1,4)]
--    [([],[(4,1),(1,2),(2,3)]),([],[(3,2),(2,1),(1,4)])]
soluciones :: Problema -> [Estado]
soluciones p = buscaProfundidad sucesores esFinal (inicial p)

domino :: Problema -> [[Ficha]]
domino p = map snd (soluciones p)

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

spec :: Spec
spec = do
  it "e1" $
    domino [(1,2),(2,3),(1,4)] `shouldBe`
    [[(4,1),(1,2),(2,3)],[(3,2),(2,1),(1,4)]]
  it "e2" $
    domino [(1,2),(1,1),(1,4)] `shouldBe`
    [[(4,1),(1,1),(1,2)],[(2,1),(1,1),(1,4)]]
  it "e3" $
    domino [(1,2),(3,4),(2,3)] `shouldBe`
    [[(1,2),(2,3),(3,4)],[(4,3),(3,2),(2,1)]]
```

```
  it "e4" $
    domino [(1,2),(2,3),(5,4)] `shouldBe`
    []

-- La verificación es
--    λ> verifica
--
--    e1
--    e2
--    e3
--    e4
--
--    Finished in 0.0013 seconds
--    4 examples, 0 failures
```

13.15.2. En Python

```
# ---------------------------------------------------------------------
# Las fichas del dominó se pueden representar por pares de números
# enteros. El problema del dominó consiste en colocar todas las fichas
# de una lista dada de forma que el segundo número de cada ficha
# coincida con el primero de la siguiente.
#
# Definir, mediante búsqueda en espacio de estados, la función
#    domino : (Problema) -> list[list[Ficha]]
# tal que domino(fs) es la lista de las soluciones del problema del
# dominó correspondiente a las fichas fs. Por ejemplo,
#    >>> domino([(1,2),(2,3),(1,4)])
#    [[(3, 2), (2, 1), (1, 4)], [(4, 1), (1, 2), (2, 3)]]
#    >>> domino([(1,2),(1,1),(1,4)])
#    [[(2, 1), (1, 1), (1, 4)], [(4, 1), (1, 1), (1, 2)]]
#    >>> domino([(1,2),(3,4),(2,3)])
#    [[(4, 3), (3, 2), (2, 1)], [(1, 2), (2, 3), (3, 4)]]
#    >>> domino([(1,2),(2,3),(5,4)])
#    []
# ---------------------------------------------------------------------

from src.BusquedaEnProfundidad import buscaProfundidad

# Las fichas son pares de números enteros.
```

```
Ficha  = tuple[int, int]

# Un problema está definido por la lista de fichas que hay que colocar
Problema = list[Ficha]

# Los estados son los pares formados por la listas sin colocar y las
# colocadas.
Estado = tuple[list[Ficha], list[Ficha]]

# inicial(p) es el estado inicial del problema p. Por ejemplo,
#    >>> inicial([(1,2),(2,3),(1,4)])
#    ([(1, 2), (2, 3), (1, 4)], [])
def inicial(p: Problema) -> Estado:
    return (p, [])

# esFinal(e) se verifica si e es un estado final. Por ejemplo,
#    >>> esFinal(([], [(4,1),(1,2),(2,3)]))
#    True
#    >>> esFinal(([(2,3)], [(4,1),(1,2)]))
#    False
def esFinal(e: Estado) -> bool:
    return not e[0]

# elimina(f, fs) es la lista obtenida eliminando la ficha f de la lista
# fs. Por ejemplo,
#    >>> elimina((1,2),[(4,1),(1,2),(2,3)])
#    [(4, 1), (2, 3)]
def elimina(f: Ficha, fs: list[Ficha]) -> list[Ficha]:
    return [g for g in fs if g != f]

# sucesores(e) es la lista de los sucesores del estado e. Por ejemplo,
#    >>> sucesores(([(1,2),(2,3),(1,4)],[]))
#    [([(2,3),(1,4)],[(1,2)]),
#     ([(1,2),(1,4)],[(2,3)]),
#     ([(1,2),(2,3)],[(1,4)]),
#     ([(2,3),(1,4)],[(2,1)]),
#     ([(1,2),(1,4)],[(3,2)]),
#     ([(1,2),(2,3)],[(4,1)])]
#    >>> sucesores(([(2,3),(1,4)],[(1,2)]))
#    [([(2,3)],[(4,1),(1,2)])]
```

```
#    >>> sucesores(([(2,3),(1,4)],[(2,1)]))
#    [([(1,4)],[(3,2),(2,1)])]
def sucesores(e: Estado) -> list[Estado]:
    if not e[1]:
        return [(elimina((a,b), e[0]), [(a,b)]) for (a,b) in e[0] if a != b] + \
               [(elimina((a,b), e[0]), [(b,a)]) for (a,b) in e[0]]
    return [(elimina((u,v),e[0]),[(u,v)]+e[1]) for (u,v) in e[0] if u != v and v
           [(elimina((u,v),e[0]),[(v,u)]+e[1]) for (u,v) in e[0] if u != v and u
           [(elimina((u,v),e[0]),[(u,v)]+e[1]) for (u,v) in e[0] if u == v and u

# soluciones(p) es la lista de las soluciones del problema p. Por
# ejemplo,
#    >>> soluciones([(1,2),(2,3),(1,4)])
#    [([], [(3, 2), (2, 1), (1, 4)]), ([], [(4, 1), (1, 2), (2, 3)])]
def soluciones(p: Problema) -> list[Estado]:
    return buscaProfundidad(sucesores, esFinal, inicial(p))

def domino(p: Problema) -> list[list[Ficha]]:
    return [s[1] for s in soluciones(p)]

# # Verificación
# # ============

def test_domino() -> None:
    assert domino([(1,2),(2,3),(1,4)]) == \
        [[(3, 2), (2, 1), (1, 4)], [(4, 1), (1, 2), (2, 3)]]
    assert domino([(1,2),(1,1),(1,4)]) == \
        [[(2, 1), (1, 1), (1, 4)], [(4, 1), (1, 1), (1, 2)]]
    assert domino([(1,2),(3,4),(2,3)]) == \
        [[(4, 3), (3, 2), (2, 1)], [(1, 2), (2, 3), (3, 4)]]
    assert domino([(1,2),(2,3),(5,4)]) == \
        []
    print("Verificado")

# La verificación es
#    >>> test_domino()
#    Verificado
```

13.16. Problema de suma cero

13.16.1. En Haskell

```
-- ---------------------------------------------------------------------
-- El problema de suma cero consiste en, dado el conjunto de números
-- enteros, encontrar sus subconjuntos no vacío cuyos elementos sumen
-- cero.
--
-- Usando el [procedimiento de búsqueda en profundidad](https://bit.ly/3NPI4qV),
-- definir la función
--    suma0 :: [Int] -> [[Int]]
-- tal que (suma0 ns) es la lista de las soluciones del problema de suma
-- cero para ns. Por ejemplo,
--    λ> suma0 [-7,-3,-2,5,8]
--    [[-3,-2,5]]
--    λ> suma0 [-7,-3,-2,5,8,-1]
--    [[-7,-3,-2,-1,5,8],[-7,-1,8],[-3,-2,5]]
--    λ> suma0 [-7,-3,1,5,8]
--    []
-- ---------------------------------------------------------------------

module Problema_de_suma_cero where

import BusquedaEnProfundidad (buscaProfundidad)
import Data.List (delete, nub, sort)
import Test.Hspec (Spec, hspec, it, shouldBe)

-- Los estados son ternas formadas por los números seleccionados, su
-- suma y los restantes números.
type Estado = ([Int], Int, [Int])

inicial :: [Int] -> Estado
inicial ns = ([],0,ns)

esFinal :: Estado -> Bool
esFinal (xs,s,_) =
  not (null xs) && s == 0

sucesores :: Estado -> [Estado]
sucesores (xs,s,ns) =
```

```
  [(n:xs, n+s, delete n ns) | n <- ns]

soluciones :: [Int] -> [Estado]
soluciones ns =
  buscaProfundidad sucesores esFinal (inicial ns)

suma0 :: [Int] -> [[Int]]
suma0 ns =
  nub [sort xs | (xs,_,_) <- soluciones ns]

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

spec :: Spec
spec = do
  it "e1" $
    suma0 [-7,-3,-2,5,8] `shouldBe`
    [[-3,-2,5]]
  it "e2" $
    suma0 [-7,-3,-2,5,8,-1] `shouldBe`
    [[-7,-3,-2,-1,5,8],[-7,-1,8],[-3,-2,5]]
  it "e3" $
    suma0 [-7,-3,1,5,8] `shouldBe`
    []

-- La verificación es
--    λ> verifica
--
--    e1
--    e2
--    e3
--
--    Finished in 0.0098 seconds
--    3 examples, 0 failures
```

13.16.2. En Python

```python
# ---------------------------------------------------------------------
# El problema de suma cero consiste en, dado el conjunto de números
# enteros, encontrar sus subconjuntos no vacío cuyos elementos sumen
# cero.
#
# Usando el [procedimiento de búsqueda en profundidad](https://bit.ly/3NPI4qV),
# definir la función
#    suma0 : (list[int]) -> list[list[int]]
# tal que suma0(ns) es la lista de las soluciones del problema de suma
# cero para ns. Por ejemplo,
#    λ> suma0([-7,-3,-2,5,8])
#    [[-3,-2,5]]
#    λ> suma0([-7,-3,-2,5,8,-1])
#    [[-7,-3,-2,-1,5,8],[-7,-1,8],[-3,-2,5]]
#    λ> suma0([-7,-3,1,5,8])
#    []
# ---------------------------------------------------------------------

from src.BusquedaEnProfundidad import buscaProfundidad

# Los estados son ternas formadas por los números seleccionados, su
# suma y los restantes números.
Estado = tuple[list[int], int, list[int]]

def inicial(ns: list[int]) -> Estado:
    return ([], 0, ns)

def esFinal(e: Estado) -> bool:
    (xs,s,_) = e
    return xs != [] and s == 0

def sucesores(e: Estado) -> list[Estado]:
    (xs, s, ns) = e
    return [([n] + xs, n + s, [m for m in ns if m !=n])
            for n in ns]

def soluciones(ns: list[int]) -> list[Estado]:
    return buscaProfundidad(sucesores, esFinal, inicial(ns))
```

```
def suma0(ns: list[int]) -> list[list[int]]:
    xss = [list(sorted(s[0])) for s in soluciones(ns)]
    r = []
    for xs in xss:
        if xs not in r:
            r.append(xs)
    return r

# Verificación
# ============

def test_suma0() -> None:
    assert suma0([-7,-3,-2,5,8]) == \
        [[-3,-2,5]]
    assert suma0([-7,-3,-2,5,8,-1]) == \
        [[-7,-3,-2,-1,5,8],[-7,-1,8],[-3,-2,5]]
    assert not suma0([-7,-3,1,5,8])
    print("Verificado")

# La verificación es
#    >>> test_suma0()
#    Verificado
```

13.17. Problema de las jarras

13.17.1. En Haskell

```
-- ---------------------------------------------------------------------
-- En el problema de las jarras (A,B,C) se tienen dos jarras sin marcas
-- de medición, una de A litros de capacidad y otra de B. También se
-- dispone de una bomba que permite llenar las jarras de agua.
--
-- El problema de las jarras (A,B,C) consiste en determinar cómo se
-- puede lograr tener exactamente C litros de agua en la jarra de A
-- litros de capacidad.
--
-- Usando el [procedimiento de búsqueda en anchura](https://bit.ly/3XBlqG7),
-- definir la función
--    jarras :: (Int,Int,Int) -> [[(Int,Int)]]
-- tal (jarras (a,b,c)) es la lista de las soluciones del problema de las
```

```
-- jarras (a,b,c). Por ejemplo,
--    λ> take 3 (jarras (4,3,2))
--    [[(0,0),(0,3),(3,0),(3,3),(4,2),(0,2),(2,0)],
--     [(0,0),(4,0),(1,3),(1,0),(0,1),(4,1),(2,3)],
--     [(0,0),(0,3),(3,0),(4,0),(1,3),(1,0),(0,1),(4,1),(2,3)]]
--
-- La interpretación [(0,0),(4,0),(1,3),(1,0),(0,1),(4,1),(2,3)] es:
--    (0,0) se inicia con las dos jarras vacías,
--    (4,0) se llena la jarra de 4 con el grifo,
--    (1,3) se llena la de 3 con la de 4,
--    (1,0) se vacía la de 3,
--    (0,1) se pasa el contenido de la primera a la segunda,
--    (4,1) se llena la primera con el grifo,
--    (2,3) se llena la segunda con la primera.
--
-- Otros ejemplos
--    λ> length (jarras (15,10,5))
--    8
--    λ> map length (jarras (15,10,5))
--    [3,5,5,7,7,7,8,9]
--    λ> jarras (15,10,4)
--    []
-- ---------------------------------------------------------------------

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

module Problema_de_las_jarras where

import BusquedaEnAnchura (buscaAnchura)
import Test.Hspec (Spec, hspec, it, shouldBe)

-- Un problema es una lista de 3 números enteros (a,b,c) tales que a es
-- la capacidad de la primera jarra, b es la capacidad de la segunda
-- jarra y c es el número de litros que se desea obtener en la primera
-- jarra.
type Problema = (Int,Int,Int)

-- Una configuracion es una lista de dos números. El primero es el
-- contenido de la primera jarra y el segundo el de la segunda.
type Configuracion = (Int,Int)
```

```haskell
-- Inicialmente, las dos jarras están vacías.
configuracionInicial :: Configuracion
configuracionInicial = (0,0)

-- (esConfiguracionFinal p e) se verifica si e es un configuracion final
-- del problema p.
esConfiguracionFinal :: Problema -> Configuracion -> Bool
esConfiguracionFinal (_,_,c) (x,_) = x == c

-- (sucesorasConfiguracion p c) son las sucesoras de la configuración c
-- del problema p. Por ejemplo,
--    sucesorasConfiguracion (4,3,2) (0,0)  ==  [(4,0),(0,3)]
--    sucesorasConfiguracion (4,3,2) (4,0)  ==  [(4,3),(0,0),(1,3)]
--    sucesorasConfiguracion (4,3,2) (4,3)  ==  [(0,3),(4,0)]
sucesorasConfiguracion :: Problema -> Configuracion -> [Configuracion]
sucesorasConfiguracion (a,b,_) (x,y) =
    [(a,y) | x < a] ++
    [(x,b) | y < b] ++
    [(0,y) | x > 0] ++
    [(x,0) | y > 0] ++
    [(a,y-(a-x)) | x < a, y > 0, x + y > a] ++
    [(x-(b-y),b) | x > 0, y < b, x + y > b] ++
    [(x+y,0) | y > 0, x + y <= a] ++
    [(0,x+y) | x > 0, x + y <= b]

-- Los estados son listas de configuraciones [c_n,...c_2,c_1] tales que
-- c_1 es la configuración inicial y, para 2 <= i <= n, c_i es una
-- sucesora de c_(i-1).
type Estado = [Configuracion]

-- inicial es el estado cuyo único elemento es la configuración
-- inicial.
inicial :: Estado
inicial = [configuracionInicial]

-- (esFinal p e) se verifica si e es un estado final; es decir, su
-- primer elemento es una configuración final.
esFinal :: Problema -> Estado -> Bool
esFinal p (e:_) = esConfiguracionFinal p e
```

```haskell
-- (sucesores p e) es la lista de los sucesores del estado e en el
-- problema p. Por ejemplo,
--    λ> sucesores (4,3,2) [(0,0)]
--    [[(4,0),(0,0)],[(0,3),(0,0)]]
--    λ> sucesores (4,3,2) [(4,0),(0,0)]
--    [[(4,3),(4,0),(0,0)],[(1,3),(4,0),(0,0)]]
--    λ> sucesores (4,3,2) [(4,3),(4,0),(0,0)]
--    [[(0,3),(4,3),(4,0),(0,0)]]
sucesores :: Problema -> Estado -> [Estado]
sucesores p e@(c:_) =
    [c':e | c' <- sucesorasConfiguracion p c,
            c' `notElem` e]

jarras :: Problema -> [Estado]
jarras p = map reverse soluciones
  where
     soluciones = buscaAnchura (sucesores p) (esFinal p) inicial

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

spec :: Spec
spec = do
  it "e1" $
    take 3 (jarras (4,3,2)) `shouldBe`
    [[(0,0),(0,3),(3,0),(3,3),(4,2),(0,2),(2,0)],
     [(0,0),(4,0),(1,3),(1,0),(0,1),(4,1),(2,3)],
     [(0,0),(0,3),(3,0),(4,0),(1,3),(1,0),(0,1),(4,1),(2,3)]]
  it "e2" $
    length (jarras (15,10,5)) `shouldBe` 8
  it "e3" $
    map length (jarras (15,10,5)) `shouldBe`
    [3,5,5,7,7,7,8,9]
  it "e4" $
    jarras (15,10,4) `shouldBe` []
```

```
-- La verificación es
--    λ> verifica
--
--    e1
--    e2
--    e3
--    e4
--
--    Finished in 0.0080 seconds
--    4 examples, 0 failures
```

13.17.2. En Python

```
# ---------------------------------------------------------------------
# En el problema de las jarras (A,B,C) se tienen dos jarras sin marcas
# de medición, una de A litros de capacidad y otra de B. También se
# dispone de una bomba que permite llenar las jarras de agua.
#
# El problema de las jarras (A,B,C) consiste en determinar cómo se
# puede lograr tener exactamente C litros de agua en la jarra de A
# litros de capacidad.
#
# Usando el [procedimiento de búsqueda en anchura](https://bit.ly/3XBlqG7),
# definir la función
#    jarras: tuple[int,int,int] -> list[list[tuple[int,int]]]
# tal jarras((a,b,c)) es la lista de las soluciones del problema de las
# jarras (a,b,c). Por ejemplo,
#    >>> jarras((4,3,2))[:3]
#    [[(0, 0), (4, 0), (1, 3), (1, 0), (0, 1), (4, 1), (2, 3)],
#     [(0, 0), (0, 3), (3, 0), (3, 3), (4, 2), (0, 2), (2, 0)],
#     [(0, 0), (4, 0), (4, 3), (0, 3), (3, 0), (3, 3), (4, 2), (0, 2), (2, 0)]]
#
# La interpretación [(0,0),(4,0),(1,3),(1,0),(0,1),(4,1),(2,3)] es:
#    (0,0) se inicia con las dos jarras vacías,
#    (4,0) se llena la jarra de 4 con el grifo,
#    (1,3) se llena la de 3 con la de 4,
#    (1,0) se vacía la de 3,
#    (0,1) se pasa el contenido de la primera a la segunda,
#    (4,1) se llena la primera con el grifo,
#    (2,3) se llena la segunda con la primera.
```

```
#
# Otros ejemplos
#    >>> len(jarras((15,10,5)))
#    8
#    >>> [len(e) for e in jarras((15,10,5))]
#    [3, 5, 5, 7, 7, 7, 8, 9]
#    >>> jarras((15,10,4))
#    []
# ---------------------------------------------------------------------

from src.BusquedaEnAnchura import buscaAnchura

# Un problema es una lista de 3 números enteros (a,b,c) tales que a es
# la capacidad de la primera jarra, b es la capacidad de la segunda
# jarra y c es el número de litros que se desea obtener en la primera
# jarra.
Problema = tuple[int, int, int]

# Una configuracion es una lista de dos números. El primero es el
# contenido de la primera jarra y el segundo el de la segunda.
Configuracion = tuple[int, int]

# Inicialmente, las dos jarras están vacías.
configuracionInicial: Configuracion = (0,0)

# esConfiguracionFinal(p, e) se verifica si e es un configuracion final
# del problema p.
def esConfiguracionFinal(p: Problema, c: Configuracion) -> bool:
    return p[2] == c[0]

# sucesorasConfiguracion(p, c) son las sucesoras de la configuración c
# del problema p. Por ejemplo,
#    sucesorasConfiguracion((4,3,2), (0,0))  ==  [(4,0),(0,3)]
#    sucesorasConfiguracion((4,3,2), (4,0))  ==  [(4,3),(0,0),(1,3)]
#    sucesorasConfiguracion((4,3,2), (4,3))  ==  [(0,3),(4,0)]
def sucesorasConfiguracion(p: Problema, c: Configuracion) -> list[Configuracion]:
    (a, b, _) = p
    (x, y) = c
    r = []
    if x < a:
```

```
        r.append((a, y))
    if y < b:
        r.append((x, b))
    if x > 0:
        r.append((0, y))
    if y > 0:
        r.append((x, 0))
    if x < a and y > 0 and x + y > a:
        r.append((a, y - (a - x)))
    if x > 0 and y < b and x + y > b:
        r.append((x - (b - y), b))
    if y > 0 and x + y <= a:
        r.append((x + y, 0))
    if x > 0 and x + y <= b:
        r.append((0, x + y))
    return r

# Los estados son listas de configuraciones [c_n,...c_2,c_1] tales que
# c_1 es la configuración inicial y, para 2 <= i <= n, c_i es una
# sucesora de c_(i-1).
Estado = list[Configuracion]

# inicial es el estado cuyo único elemento es la configuración
# inicial.
inicial: Estado = [configuracionInicial]

# esFinal(p, e) se verifica si e es un estado final; es decir, su
# primer elemento es una configuración final.
def esFinal(p: Problema, e: Estado) -> bool:
    return esConfiguracionFinal(p, e[0])

# sucesores(p, e) es la lista de los sucesores del estado e en el
# problema p. Por ejemplo,
#    λ> sucesores((4,3,2), [(0,0)])
#    [[(4,0),(0,0)],[(0,3),(0,0)]]
#    λ> sucesores((4,3,2), [(4,0),(0,0)])
#    [[(4,3),(4,0),(0,0)],[(1,3),(4,0),(0,0)]]
#    λ> sucesores((4,3,2), [(4,3),(4,0),(0,0)])
#    [[(0,3),(4,3),(4,0),(0,0)]]
def sucesores(p: Problema, e: Estado) -> list[Estado]:
```

```
    return [[c] + e
            for c in sucesorasConfiguracion(p, e[0])
            if c not in e]

def jarras(p: Problema) -> list[Estado]:
    soluciones = buscaAnchura(lambda e: sucesores(p, e),
                              lambda e: esFinal(p, e),
                              inicial)
    return [list(reversed(e)) for e in soluciones]

# Verificación
# ============

def test_jarras() -> None:
    assert jarras((4,3,2))[:3] == \
        [[(0, 0), (4, 0), (1, 3), (1, 0), (0, 1), (4, 1), (2, 3)],
         [(0, 0), (0, 3), (3, 0), (3, 3), (4, 2), (0, 2), (2, 0)],
         [(0, 0), (4, 0), (4, 3), (0, 3), (3, 0), (3, 3), (4, 2), (0, 2), (2, 0)]
    assert len(jarras((15,10,5))) == 8
    assert [len(e) for e in jarras((15,10,5))] == [3, 5, 5, 7, 7, 7, 8, 9]
    assert jarras((15,10,4)) == []
    print("Verificado")

# La verificación es
#    >>> test_jarras()
#    Verificado
```

Capítulo 14

Programación dinámica

Contenido

14.1. La función de Fibonacci

14.1.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Los primeros términos de la sucesión de Fibonacci son
--    0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, ...
--
-- Escribir dos definiciones (una recursiva y otra con programación
-- dinámica) de la función
--    fib :: Integer -> Integer
-- tal que (fib n) es el n-ésimo término de la sucesión de Fibonacci. Por ejemplo
--    fib 6 == 8
--
-- Comparar la eficiencia de las dos definiciones.
-- ---------------------------------------------------------------------

module La_funcion_de_Fibonacci_por_programacion_dinamica where

import Data.Array
import Test.Hspec (Spec, hspec, it, shouldBe)

-- 1ª definición (por recursión)
-- ============================

fib1 :: Integer -> Integer
fib1 0 = 0
fib1 1 = 1
fib1 n = fib1 (n-1) + fib1 (n-2)
```

```haskell
-- 2ª definición (con programación dinámica)
-- =========================================

fib2 :: Integer -> Integer
fib2 n = vectorFib2 n ! n

-- (vectorFib2 n) es el vector con índices de 0 a n tal que el valor
-- de la posición i es el i-ésimo número de Finonacci. Por ejemplo,
--    λ> vectorFib2 7
--    array (0,7) [(0,0),(1,1),(2,1),(3,2),(4,3),(5,5),(6,8),(7,13)]
vectorFib2 :: Integer -> Array Integer Integer
vectorFib2 n = v where
  v = array (0,n) [(i,f i) | i <- [0..n]]
  f 0 = 0
  f 1 = 1
  f m = v!(m-1) + v!(m-2)

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

-- La comparación es
--    λ> fib1 34
--    5702887
--    (11.82 secs, 3,504,944,704 bytes)
--    λ> fib2 34
--    5702887
--    (0.01 secs, 587,808 bytes)

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

spec :: Spec
spec = do
  it "e1" $
    fib1 6 `shouldBe` 8
  it "e2" $
```

```
    fib2 6 `shouldBe` 8
  it "e3" $
    map fib1 [0..9] `shouldBe` map fib2 [0..9]

-- La verificación es
--    λ> verifica
--
--    e1
--    e2
--    e3
--
--    Finished in 0.0007 seconds
--    3 examples, 0 failures
--    (0.01 secs, 788,952 bytes)
```

14.1.2. En Python

```
# ---------------------------------------------------------------------
# Los primeros términos de la sucesión de Fibonacci son
#    0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, ...
#
# Escribir dos definiciones (una recursiva y otra con programación
# dinámica) de la función
#    fib :: Integer -> Integer
# tal que (fib n) es el n-ésimo término de la sucesión de Fibonacci. Por
# ejemplo,
#    fib(6) == 8
#
# Comparar la eficiencia de las dos definiciones.
# ---------------------------------------------------------------------

from sys import setrecursionlimit
from timeit import Timer, default_timer

import numpy as np
import numpy.typing as npt

setrecursionlimit(10**6)

# 1ª definición (por recursión)
```

```
# ==============================

def fib1(n: int) -> int:
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fib1(n - 1) + fib1(n - 2)

# 2ª definición (con programación dinámica)
# =========================================

def fib2(n: int) -> int:
    return vectorFib2(n)[n]

# (vectorFib2 n) es el vector con índices de 0 a n tal que el valor
# de la posición i es el i-ésimo número de Finonacci. Por ejemplo,
#    >>> vectorFib2(7)
#    [0, 1, 1, 2, 3, 5, 8, 13]
def vectorFib2(n: int) -> list[int]:
    v = [0] * (n + 1)
    v[0] = 0
    v[1] = 1
    for i in range(2, n + 1):
        v[i] = v[i - 1] + v[i - 2]
    return v

# 3ª definición (con programación dinámica y array)
# =================================================

def fib3(n: int) -> int:
    return vectorFib3(n)[n]

# (vectorFib3 n) es el vector con índices de 0 a n tal que el valor
# de la posición i es el i-ésimo número de Finonacci. Por ejemplo,
#    >>> vectorFib3(7)
#    array([ 0,  1,  1,  2,  3,  5,  8, 13])
def vectorFib3(n: int) -> npt.NDArray[np.int_]:
    v = np.zeros(n + 1, dtype=int)
    v[0] = 0
```

```
    v[1] = 1
    for i in range(2, n + 1):
        v[i] = v[i - 1] + v[i - 2]
    return v

# 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('fib1(34)')
#    2.10 segundos
#    >>> tiempo('fib2(34)')
#    0.00 segundos
#    >>> tiempo('fib3(34)')
#    0.00 segundos
#
#    >>> tiempo('fib2(100000)')
#    0.37 segundos
#    >>> tiempo('fib3(100000)')
#    0.08 segundos

# Verificación
# ============

def test_fib() -> None:
    assert fib1(6) == 8
    assert fib2(6) == 8
    assert fib3(6) == 8
    print("Verificado")

# La verificación es
#    >>> test_fib()
#    Verificado
```

14.2. Coeficientes binomiales

14.2.1. En Haskell

```
-- ---------------------------------------------------------------------
-- El coeficiente binomial n sobre k es el número de subconjuntos de k
-- elementos escogidos de un conjunto con n elementos.
--
-- Definir la función
--    binomial :: Integer -> Integer -> Integer
-- tal que (binomial n k) es el coeficiente binomial n sobre k. Por
-- ejemplo,
--    binomial 6 3 == 20
--    binomial 5 2 == 10
--    binomial 5 3 == 10
-- ---------------------------------------------------------------------

module Coeficientes_binomiales where

import Data.Array (Array, (!), array)
import Test.Hspec (Spec, hspec, it, shouldBe)

-- 1ª definición (por recursión)
-- =============================

binomial1 :: Integer -> Integer -> Integer
binomial1 _ 0 = 1
binomial1 n k
  | n == k    = 1
  | otherwise = binomial1 (n-1) (k-1) + binomial1 (n-1) k

-- 2ª definición (con programación dinámica)
-- =========================================

binomial2 :: Integer -> Integer -> Integer
binomial2 n k = matrizBinomial2 n k ! (n,k)

-- (matrizBinomial2 n k) es la matriz de orden (n+1)x(k+1) tal que el
-- valor en la posición (i,j) (con j <= i) es el coeficiente binomial i
-- sobre j. Por ejemplo,
--    λ> [[(matrizBinomial2 3 3)!(i,j) | j <- [0..i]] | i <- [0..3]]
```

```
--    [[1],[1,1],[1,2,1],[1,3,3,1]]
matrizBinomial2 :: Integer -> Integer -> Array (Integer,Integer) Integer
matrizBinomial2 n k = q where
  q = array ((0,0),(n,k)) [((i,j),f i j) | i <- [0..n], j <- [0..k]]
  f _ 0 = 1
  f i j
    | i == j    = 1
    | otherwise = q!(i-1,j-1) + q!(i-1,j)

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

-- La comparación es
--    λ> binomial1 25 12
--    5200300
--    (6.45 secs, 2,654,797,776 bytes)
--    λ> binomial2 25 12
--    5200300
--    (0.00 secs, 826,272 bytes)

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

spec :: Spec
spec = do
  it "e1" $
    binomial1 6 3 `shouldBe` 20
  it "e2" $
    binomial1 5 2 `shouldBe` 10
  it "e3" $
    binomial1 5 3 `shouldBe` 10
  it "e4" $
    binomial2 6 3 `shouldBe` 20
  it "e5" $
    binomial2 5 2 `shouldBe` 10
  it "e6" $
    binomial2 5 3 `shouldBe` 10
```

```
-- La verificación es
--    λ> verifica
--
--    e1
--    e2
--    e3
--    e4
--    e5
--    e6
--
--    Finished in 0.0006 seconds
--    6 examples, 0 failures
```

14.2.2. En Python

```
# ---------------------------------------------------------------------
# El coeficiente binomial n sobre k es el número de subconjuntos de k
# elementos escogidos de un conjunto con n elementos.
#
# Definir la función
#    binomial : (int, int) -> int
# tal que binomial(n, k) es el coeficiente binomial n sobre k. Por
# ejemplo,
#    binomial(6, 3) == 20
#    binomial(5, 2) == 10
#    binomial(5, 3) == 10
# ---------------------------------------------------------------------

from sys import setrecursionlimit
from timeit import Timer, default_timer

import numpy as np
import numpy.typing as npt

setrecursionlimit(10**6)

# 1ª definición (por recursión)
# =============================
```

```python
def binomial1(n: int, k: int) -> int:
    if k == 0:
        return 1
    if n == k:
        return 1
    return binomial1(n-1, k-1) + binomial1(n-1, k)

# 2ª definición (con programación dinámica)
# =========================================

def binomial2(n: int, k: int) -> int:
    return matrizBinomial2(n, k)[n][k]

# (matrizBinomial2 n k) es la matriz de orden (n+1)x(k+1) tal que el
# valor en la posición (i,j) (con j <= i) es el coeficiente binomial i
# sobre j. Por ejemplo,
#    >>> matrizBinomial2(3, 3)
#    [[1, 0, 0, 0], [1, 1, 0, 0], [1, 2, 1, 0], [1, 3, 3, 1]]
def matrizBinomial2(n: int, k: int) -> list[list[int]]:
    q = [[0 for i in range(k + 1)] for j in range(n + 1)]

    for i in range(n + 1):
        for j in range(min(i, k) + 1):
            if j == 0:
                q[i][j] = 1
            elif i == j:
                q[i][j] = 1
            else:
                q[i][j] = q[i - 1][j - 1] + q[i - 1][j]

    return q

# 3ª definición (con programación dinámica y numpy)
# ================================================

def binomial3(n: int, k: int) -> int:
    return matrizBinomial3(n, k)[n][k]

# (matrizBinomial3 n k) es la matriz de orden (n+1)x(k+1) tal que el
# valor en la posición (i,j) (con j <= i) es el coeficiente binomial i
```

```python
# sobre j. Por ejemplo,
#    >>> matrizBinomial3(3, 3)
#    array([[1, 0, 0, 0],
#           [1, 1, 0, 0],
#           [1, 2, 1, 0],
#           [1, 3, 3, 1]])
def matrizBinomial3(n: int, k: int) -> npt.NDArray[np.int_]:
    q = np.zeros((n + 1, k + 1), dtype=object)

    for i in range(n + 1):
        for j in range(min(i, k) + 1):
            if j == 0:
                q[i, j] = 1
            elif i == j:
                q[i, j] = 1
            else:
                q[i, j] = q[i - 1, j - 1] + q[i - 1, j]

    return q

# 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('binomial1(27, 12)')
#    4.26 segundos
#    >>> tiempo('binomial2(27, 12)')
#    0.00 segundos
#    >>> tiempo('binomial3(27, 12)')
#    0.00 segundos
#
# >>> tiempo('binomial2(50000, 12)')
# 0.18 segundos
# >>> tiempo('binomial3(50000, 12)')
# 0.26 segundos
```

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

def test_binomial() -> None:
    assert binomial1(6, 3) == 20
    assert binomial1(5, 2) == 10
    assert binomial1(5, 3) == 10
    assert binomial2(6, 3) == 20
    assert binomial2(5, 2) == 10
    assert binomial2(5, 3) == 10
    assert binomial3(6, 3) == 20
    assert binomial3(5, 2) == 10
    assert binomial3(5, 3) == 10
    print("Verificado")

# La verificación es
#    >>> test_binomial()
#    Verificado
```

14.3. Longitud de la subsecuencia común máxima

14.3.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Si a una secuencia X de elementos (pongamos por ejemplo, caracteres)
-- le quitamos algunos de ellos y dejamos los que quedan en el orden en
-- el que aparecían originalmente tenemos lo que se llama una
-- subsecuencia de X. Por ejemplo, "aaoa" es una subsecuencia de la
-- secuencia "amapola".
--
-- El término también se aplica cuando quitamos todos los elementos (es
-- decir, la secuencia vacía es siempre subsecuencia de cualquier
-- secuencia) o cuando no quitamos ninguno (lo que significa que
-- cualquier secuencia es siempre subsecuencia de sí misma).
--
-- Dadas dos secuencias X e Y, decimos que Z es una subsecuencia común
-- de X e Y si Z es subsecuencia de X y de Y. Por ejemplo, si X =
```

```
-- ”amapola” e Y = ”matamoscas”, la secuencia ”aaoa” es una de las
-- subsecuencias comunes de X e Y más larga, con longitud 4, ya que no
-- hay ninguna subsecuencia común a X e Y de longitud mayor que
-- 4. También son subsecuencias comunes de longitud 4 ”maoa” o ”amoa”.
--
-- Se desea encontrar la longitud de las subsecuencias comunes más
-- largas de dos secuencias de caracteres dadas.
--
-- Definir la función
--    longitudSCM :: Eq a => [a] -> [a] -> Int
-- tal que (longitudSCM xs ys) es la longitud de la subsecuencia máxima
-- de xs e ys. Por ejemplo,
--    longitudSCM ”amapola” ”matamoscas” == 4
--    longitudSCM ”atamos” ”matamoscas”  == 6
--    longitudSCM ”aaa” ”bbbb”           == 0
-- ---------------------------------------------------------------------

module Longitud_SCM where

import Data.Array (Array,(!), array, listArray)
import Test.Hspec (Spec, hspec, it, shouldBe)

-- 1ª definición (por recursión)
-- =============================

longitudSCM1 :: Eq a => [a] -> [a] -> Int
longitudSCM1 [] _ = 0
longitudSCM1 _ [] = 0
longitudSCM1 (x:xs) (y:ys)
  | x == y    = 1 + longitudSCM1 xs ys
  | otherwise = max (longitudSCM1 (x:xs) ys) (longitudSCM1 xs (y:ys))

-- 2ª definición (con programación dinámica)
-- =========================================

longitudSCM2 :: Eq a => [a] -> [a] -> Int
longitudSCM2 xs ys = matrizLongitudSCM2 xs ys ! (n,m)
  where n = length xs
        m = length ys
```

```
-- (matrizLongitudSCM2 xs ys) es la matriz de orden (n+1)x(m+1) (donde n
-- y m son los números de elementos de xs e ys, respectivamente) tal que
-- el valor en la posición (i,j) es la longitud de la SCM de los i
-- primeros elementos de xs y los j primeros elementos de ys. Por ejemplo,
--    λ> elems (matrizLongitudSCM2 "amapola" "matamoscas")
--    [0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,1,0,1,1,1,1,2,2,2,2,2,2,
--     0,1,2,2,2,2,2,2,2,3,3,0,1,2,2,2,2,2,2,2,3,3,0,1,2,2,2,2,3,3,3,3,3,
--     0,1,2,2,2,2,3,3,3,3,3,0,1,2,2,3,3,3,3,3,4,4]
-- Gráficamente,
--       m a t a m o s c a s
--    [0,0,0,0,0,0,0,0,0,0,0,
-- a   0,0,1,1,1,1,1,1,1,1,1,
-- m   0,1,1,1,1,2,2,2,2,2,2,
-- a   0,1,2,2,2,2,2,2,2,3,3,
-- p   0,1,2,2,2,2,2,2,2,3,3,
-- o   0,1,2,2,2,2,3,3,3,3,3,
-- l   0,1,2,2,2,2,3,3,3,3,3,
-- a   0,1,2,2,3,3,3,3,3,4,4]
matrizLongitudSCM2 :: Eq a => [a] -> [a] -> Array (Int,Int) Int
matrizLongitudSCM2 xs ys = q
  where
    n = length xs
    m = length ys
    v = listArray (1,n) xs
    w = listArray (1,m) ys
    q = array ((0,0),(n,m)) [((i,j), f i j) | i <- [0..n], j <- [0..m]]
      where f 0 _ = 0
            f _ 0 = 0
            f i j | v ! i == w ! j = 1 + q ! (i-1,j-1)
                  | otherwise      = max (q ! (i-1,j)) (q ! (i,j-1))

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

-- La comparación es
--    λ> longitudSCM1 (take 18 (cycle [1,3])) (take 18 (cycle [2,3]))
--    9
--    (13.90 secs, 8,883,660,048 bytes)
--    λ> longitudSCM2 (take 18 (cycle [1,3])) (take 18 (cycle [2,3]))
--    9
```

```
--    (0.01 secs, 953,880 bytes)

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

spec :: Spec
spec = do
  it "e1" $
    longitudSCM1 "amapola" "matamoscas" `shouldBe` 4
  it "e2" $
    longitudSCM1 "atamos" "matamoscas"  `shouldBe` 6
  it "e3" $
    longitudSCM1 "aaa" "bbbb"           `shouldBe` 0
  it "e4" $
    longitudSCM2 "amapola" "matamoscas" `shouldBe` 4
  it "e5" $
    longitudSCM2 "atamos" "matamoscas"  `shouldBe` 6
  it "e6" $
    longitudSCM2 "aaa" "bbbb"           `shouldBe` 0

-- La verificación es
--    λ> verifica
--
--    e1
--    e2
--    e3
--    e4
--    e5
--    e6
--
--    Finished in 0.0025 seconds
--    6 examples, 0 failures
```

14.3.2. En Python

```
# ---------------------------------------------------------------------
# Si a una secuencia X de elementos (pongamos por ejemplo, caracteres)
```

```
# le quitamos algunos de ellos y dejamos los que quedan en el orden en
# el que aparecían originalmente tenemos lo que se llama una
# subsecuencia de X. Por ejemplo, "aaoa" es una subsecuencia de la
# secuencia "amapola".
#
# El término también se aplica cuando quitamos todos los elementos (es
# decir, la secuencia vacía es siempre subsecuencia de cualquier
# secuencia) o cuando no quitamos ninguno (lo que significa que
# cualquier secuencia es siempre subsecuencia de sí misma).
#
# Dadas dos secuencias X e Y, decimos que Z es una subsecuencia común
# de X e Y si Z es subsecuencia de X y de Y. Por ejemplo, si X =
# "amapola" e Y = "matamoscas", la secuencia "aaoa" es una de las
# subsecuencias comunes de X e Y más larga, con longitud 4, ya que no
# hay ninguna subsecuencia común a X e Y de longitud mayor que
# 4. También son subsecuencias comunes de longitud 4 "maoa" o "amoa".
#
# Se desea encontrar la longitud de las subsecuencias comunes más
# largas de dos secuencias de caracteres dadas.
#
# Definir la función
#    longitudSCM : (str, str) -> int
# tal que longitudSCM(xs, ys) es la longitud de la subsecuencia máxima
# de xs e ys. Por ejemplo,
#    longitudSCM("amapola", "matamoscas") == 4
#    longitudSCM("atamos", "matamoscas")  == 6
#    longitudSCM("aaa", "bbbb")           == 0
# ---------------------------------------------------------------------

from timeit import Timer, default_timer

# 1ª definición (por recursión)
# =============================

def longitudSCM1(xs: str, ys: str) -> int:
    if not xs:
        return 0
    if not ys:
        return 0
    if xs[0] == ys[0]:
```

```
        return 1 + longitudSCM1(xs[1:], ys[1:])
    return max(longitudSCM1(xs, ys[1:]), longitudSCM1(xs[1:], ys))

# 2ª definición (con programación dinámica)
# =========================================

# matrizLongitudSCM2(xs, ys) es la matriz de orden (n+1)x(m+1) (donde n
# y m son los números de elementos de xs e ys, respectivamente) tal que
# el valor en la posición (i,j) es la longitud de la SCM de los i
# primeros elementos de xs y los j primeros elementos de ys. Por ejemplo,
#    >>> matrizLongitudSCM2("amapola", "matamoscas")
#    [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
#     [0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1],
#     [0, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2],
#     [0, 1, 2, 2, 2, 2, 2, 2, 2, 3, 3],
#     [0, 1, 2, 2, 2, 2, 2, 2, 2, 3, 3],
#     [0, 1, 2, 2, 2, 2, 3, 3, 3, 3, 3],
#     [0, 1, 2, 2, 2, 2, 3, 3, 3, 3, 3],
#     [0, 1, 2, 2, 3, 3, 3, 3, 3, 4, 4]]
# Gráficamente,
#       m a t a m o s c a s
#    [0,0,0,0,0,0,0,0,0,0,0,
# a   0,0,1,1,1,1,1,1,1,1,1,
# m   0,1,1,1,1,2,2,2,2,2,2,
# a   0,1,2,2,2,2,2,2,2,3,3,
# p   0,1,2,2,2,2,2,2,2,3,3,
# o   0,1,2,2,2,2,3,3,3,3,3,
# l   0,1,2,2,2,2,3,3,3,3,3,
# a   0,1,2,2,3,3,3,3,3,4,4]
def matrizLongitudSCM2(xs: str, ys: str) -> list[list[int]]:
    n = len(xs)
    m = len(ys)
    q = [[0 for _ in range(m + 1)] for _ in range(n + 1)]
    for i in range(1, n + 1):
        for j in range(1, m + 1):
            if xs[i - 1] == ys[j - 1]:
                q[i][j] = 1 + q[i - 1][j - 1]
            else:
                q[i][j] = max(q[i - 1][j], q[i][j - 1])
    return q
```

```
def longitudSCM2(xs: str, ys: str) -> int:
    n = len(xs)
    m = len(ys)
    return matrizLongitudSCM2(xs, ys)[n][m]

# 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('longitudSCM1([1,3]*9, [2,3]*9)')
#    8.04 segundos
#    >>> tiempo('longitudSCM2([1,3]*9, [2,3]*9)')
#    0.00 segundos

# Verificación
# ============

def test_longitudSCM() -> None:
    assert longitudSCM1("amapola", "matamoscas") == 4
    assert longitudSCM1("atamos", "matamoscas")  == 6
    assert longitudSCM1("aaa", "bbbb")           == 0
    assert longitudSCM2("amapola", "matamoscas") == 4
    assert longitudSCM2("atamos", "matamoscas")  == 6
    assert longitudSCM2("aaa", "bbbb")           == 0
    print("Verificado")

# La verificación es
#    >>> test_longitudSCM()
#    Verificado
```

14.4. Subsecuencia común máxima

14.4.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Si a una secuencia X de elementos (pongamos por ejemplo, caracteres)
-- le quitamos algunos de ellos y dejamos los que quedan en el orden en
-- el que aparecían originalmente tenemos lo que se llama una
-- subsecuencia de X. Por ejemplo, "aaoa" es una subsecuencia de la
-- secuencia "amapola".
--
-- El término también se aplica cuando quitamos todos los elementos (es
-- decir, la secuencia vacía es siempre subsecuencia de cualquier
-- secuencia) o cuando no quitamos ninguno (lo que significa que
-- cualquier secuencia es siempre subsecuencia de sí misma).
--
-- Dadas dos secuencias X e Y, decimos que Z es una subsecuencia común
-- de X e Y si Z es subsecuencia de X y de Y. Por ejemplo, si X =
-- "amapola" e Y = "matamoscas", la secuencia "aaoa" es una de las
-- subsecuencias comunes de X e Y más larga, con longitud 4, ya que no
-- hay ninguna subsecuencia común a X e Y de longitud mayor que
-- 4. También son subsecuencias comunes de longitud 4 "maoa" o "amoa".
--
-- Definir la función
--    scm :: Eq a => [a] -> [a] -> [a]
-- tal que (scm xs ys) es una de las subsecuencias comunes de longitud
-- máxima de xs e ys. Por ejemplo,
--    scm "amapola" "matamoscas" == "amoa"
--    scm "atamos" "matamoscas"  == "atamos"
--    scm "aaa" "bbbb"           == ""
-- ---------------------------------------------------------------------

module Subsecuencia_comun_maxima where

import Data.Array (Array, (!), array, listArray)
import Test.Hspec (Spec, hspec, it, shouldBe)

-- 1ª definición (por recursión)
-- =============================

scm1 :: Eq a => [a] -> [a] -> [a]
```

```
scm1 [] _ = []
scm1 _ [] = []
scm1 (x:xs) (y:ys)
  | x == y    = x : scm1 xs ys
  | otherwise = mayor (scm1 (x:xs) ys) (scm1 xs (y:ys))

-- (mayor xs ys) es la cadena más larga de xs e ys.
--    mayor "hola" "buenas"  ==  "buenas"
--    mayor "hola" "pera"    ==  "hola"
mayor :: [a] -> [a] -> [a]
mayor xs ys
  | length xs >= length ys = xs
  | otherwise              = ys

-- 2ª definición (con programación dinámica)
-- =========================================

scm2 :: Eq a => [a] -> [a] -> [a]
scm2 xs ys = reverse (matrizSCM2 xs ys ! (n,m))
  where n = length xs
        m = length ys

-- (matrizSCM2 xs ys) es la matriz de orden (n+1)x(m+1) (donde n
-- y m son los números de elementos de xs e ys, respectivamente) tal que
-- el valor en la posición (i,j) es una SCM de los i primeros
-- elementos de xs y los j primeros elementos de ys. Por ejemplo,
--    λ> elems (matrizSCM2 "amapola" "matamoscas")
--    ["","","","","","","","","","","","","","a","a","a","a","a","a",
--     "a","a","a","","m","a","a","a","ma","ma","ma","ma","ma","ma","",
--     "m","am","am","aa","ma","ma","ma","ma","ama","ama","","m","am",
--     "am","aa","ma","ma","ma","ma","ama","ama","","m","am","am","aa",
--     "ma","oma","oma","oma","ama","ama","","m","am","am","aa","ma",
--     "oma","oma","oma","ama","ama","","m","am","am","aam","aam","oma",
--     "oma","oma","aoma","aoma"]
-- Gráficamente,
--        m    a     t    a     m     o     s     c     a      s
--    ["","" ,""   ,""  ,""    ,""    ,""    ,""    ,""    ,""     ,"",
-- a   "","" ,"a"  ,"a" ,"a"   ,"a"   ,"a"   ,"a"   ,"a"   ,"a"    ,"a",
-- m   "","m","a"  ,"a" ,"a"   ,"ma"  ,"ma"  ,"ma"  ,"ma"  ,"ma"   ,"ma",
-- a   "","m","am","am","aa"  ,"ma"  ,"ma"  ,"ma"  ,"ma"  ,"ama"  ,"ama",
```

```haskell
-- p   "","m","am","am","aa" ,"ma" ,"ma" ,"ma" ,"ma" ,"ama" ,"ama",
-- o   "","m","am","am","aa" ,"ma" ,"oma","oma","oma","ama" ,"ama",
-- l   "","m","am","am","aa" ,"ma" ,"oma","oma","oma","ama" ,"ama",
-- a   "","m","am","am","aam","aam","oma","oma","oma","aoma","aoma"]
matrizSCM2 :: Eq a => [a] -> [a] -> Array (Int,Int) [a]
matrizSCM2 xs ys = q where
  q = array ((0,0),(n,m)) [((i,j), f i j) | i <- [0..n], j <- [0..m]]
  n = length xs
  m = length ys
  v = listArray (1,n) xs
  w = listArray (1,m) ys
  f 0 _ = []
  f _ 0 = []
  f i j | v ! i == w ! j = (v!i) : (q ! (i-1,j-1))
        | otherwise      = mayor (q ! (i-1,j)) (q ! (i,j-1))

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

-- La comparación es
--    λ> length (scm1 (take 18 (cycle [1,3])) (take 18 (cycle [2,3])))
--    9
--    (20.17 secs, 11,436,759,992 bytes)
--    λ> length (scm2 (take 18 (cycle [1,3])) (take 18 (cycle [2,3])))
--    9
--    (0.00 secs, 1,013,624 bytes)

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

spec :: Spec
spec = do
  it "e1" $
    scm1 "amapola" "matamoscas" `shouldBe` "amoa"
  it "e2" $
    scm1 "atamos" "matamoscas"  `shouldBe` "atamos"
  it "e3" $
```

```
    scm1 "aaa" "bbbb"           `shouldBe` ""
  it "e4" $
    scm2 "amapola" "matamoscas" `shouldBe` "amoa"
  it "e5" $
    scm2 "atamos" "matamoscas"  `shouldBe` "atamos"
  it "e6" $
    scm2 "aaa" "bbbb"           `shouldBe` ""

-- La verificación es
--    λ> verifica
--
--    e1
--    e2
--    e3
--    e4
--    e5
--    e6
--
--    Finished in 0.0026 seconds
--    6 examples, 0 failures
```

14.4.2. En Python

```
# ---------------------------------------------------------------------
# Si a una secuencia X de elementos (pongamos por ejemplo, caracteres)
# le quitamos algunos de ellos y dejamos los que quedan en el orden en
# el que aparecían originalmente tenemos lo que se llama una
# subsecuencia de X. Por ejemplo, "aaoa" es una subsecuencia de la
# secuencia "amapola".
#
# El término también se aplica cuando quitamos todos los elementos (es
# decir, la secuencia vacía es siempre subsecuencia de cualquier
# secuencia) o cuando no quitamos ninguno (lo que significa que
# cualquier secuencia es siempre subsecuencia de sí misma).
#
# Dadas dos secuencias X e Y, decimos que Z es una subsecuencia común
# de X e Y si Z es subsecuencia de X y de Y. Por ejemplo, si X =
# "amapola" e Y = "matamoscas", la secuencia "aaoa" es una de las
# subsecuencias comunes de X e Y más larga, con longitud 4, ya que no
# hay ninguna subsecuencia común a X e Y de longitud mayor que
```

```
# 4. También son subsecuencias comunes de longitud 4 "maoa" o "amoa".
#
# Definir la función
#    scm : (str, str) -> str
# tal que scm(xs, ys) es una de las subsecuencias comunes de longitud
# máxima de xs e ys. Por ejemplo,
#    scm("amapola", "matamoscas") == "amoa"
#    scm("atamos", "matamoscas")  == "atamos"
#    scm("aaa", "bbbb")           == ""
# ---------------------------------------------------------------------

from sys import setrecursionlimit
from timeit import Timer, default_timer

setrecursionlimit(10**6)

# 1ª definición (por recursión)
# =============================

# (mayor xs ys) es la cadena más larga de xs e ys.
#    mayor "hola" "buenas"  ==  "buenas"
#    mayor "hola" "pera"    ==  "hola"
def mayor(xs: str, ys: str) -> str:
    if len(xs) >= len(ys):
        return xs
    return ys

def scm1(xs: str, ys: str) -> str:
    if not xs:
        return ""
    if not ys:
        return ""
    if xs[0] == ys[0]:
        return xs[0] + scm1(xs[1:], ys[1:])
    return mayor(scm1(xs, ys[1:]), scm1(xs[1:], ys))

# 2ª definición (con programación dinámica)
# =========================================

def scm2(xs: str, ys: str) -> str:
```

```
    n = len(xs)
    m = len(ys)
    return (matrizSCM2(xs, ys)[n][m])[::-1]

# matrizSCM2(xs, ys) es la matriz de orden (n+1)x(m+1) (donde n
# y m son los números de elementos de xs e ys, respectivamente) tal que
# el valor en la posición (i,j) es una SCM de los i primeros
# elementos de xs y los j primeros elementos de ys. Por ejemplo,
#    >>> matrizSCM2("amapola", "matamoscas")
#    [['', '', '', '', '', '', '', '', '', '', ''],
#     ['', '', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a'],
#     ['', 'm', 'a', 'a', 'a', 'ma', 'ma', 'ma', 'ma', 'ma', 'ma'],
#     ['', 'm', 'am', 'am', 'aa', 'ma', 'ma', 'ma', 'ma', 'ama', 'ama'],
#     ['', 'm', 'am', 'am', 'aa', 'ma', 'ma', 'ma', 'ma', 'ama', 'ama'],
#     ['', 'm', 'am', 'am', 'aa', 'ma', 'oma', 'oma', 'oma', 'ama', 'ama'],
#     ['', 'm', 'am', 'am', 'aa', 'ma', 'oma', 'oma', 'oma', 'ama', 'ama'],
#     ['', 'm', 'am', 'am', 'aam', 'aam', 'oma', 'oma', 'oma', 'aoma', 'aoma']]
# Gráficamente,
#         m   a    t    a     m     o     s     c     a      s
#    ["",""  ,""  ,""  ,""   ,""   ,""   ,""   ,""   ,""    ,"",
# a   "",""  ,"a" ,"a" ,"a"  ,"a"  ,"a"  ,"a"  ,"a"  ,"a"   ,"a",
# m   "","m","a"  ,"a" ,"a"  ,"ma" ,"ma" ,"ma" ,"ma" ,"ma"  ,"ma",
# a   "","m","am","am","aa"  ,"ma" ,"ma" ,"ma" ,"ma" ,"ama" ,"ama",
# p   "","m","am","am","aa"  ,"ma" ,"ma" ,"ma" ,"ma" ,"ama" ,"ama",
# o   "","m","am","am","aa"  ,"ma" ,"oma","oma","oma","ama" ,"ama",
# l   "","m","am","am","aa"  ,"ma" ,"oma","oma","oma","ama" ,"ama",
# a   "","m","am","am","aam","aam","oma","oma","oma","aoma","aoma"]
def matrizSCM2(xs: str, ys: str) -> list[list[str]]:
    n = len(xs)
    m = len(ys)
    q = [["" for _ in range(m + 1)] for _ in range(n + 1)]
    for i in range(1, n + 1):
        for j in range(1, m + 1):
            if xs[i - 1] == ys[j - 1]:
                q[i][j] = xs[i - 1] + q[i - 1][j - 1]
            else:
                q[i][j] = mayor(q[i - 1][j], q[i][j - 1])
    return q

# # 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('scm1(["1","3"]*9, ["2","3"]*9)')
#    8.44 segundos
#    >>> tiempo('scm2(["1","3"]*9, ["2","3"]*9)')
#    0.00 segundos

# Verificación
# ============

def test_scm() -> None:
    assert scm1("amapola", "matamoscas") == "amoa"
    assert scm1("atamos", "matamoscas")  == "atamos"
    assert scm1("aaa", "bbbb")           == ""
    assert scm2("amapola", "matamoscas") == "amoa"
    assert scm2("atamos", "matamoscas")  == "atamos"
    assert scm2("aaa", "bbbb")           == ""
    print("Verificado")

# La verificación es
#    >>> test_scm()
#    Verificado
```

14.5. La distancia Levenshtein (con programación dinámica)

14.5.1. En Haskell

```
-- ---------------------------------------------------------------------
-- La distancia de Levenshtein (o distancia de edición) es el número
-- mínimo de operaciones requeridas para transformar una cadena de
-- caracteres en otra. Las operaciones de edición que se pueden hacer
-- son:
```

```
-- + insertar un carácter (por ejemplo, de "abc" a "abca")
-- + eliminar un carácter (por ejemplo, de "abc" a "ac")
-- + sustituir un carácter (por ejemplo, de "abc" a "adc")
--
-- Por ejemplo, la distancia de Levenshtein entre "casa" y "calle" es de
-- 3 porque se necesitan al menos tres ediciones elementales para
-- cambiar uno en el otro:
--    "casa"  --> "cala"  (sustitución de 's' por 'l')
--    "cala"  --> "calla" (inserción de 'l' entre 'l' y 'a')
--    "calla" --> "calle" (sustitución de 'a' por 'e')
--
-- Definir la función
--    levenshtein :: String -> String -> Int
-- tal que (levenshtein xs ys) es la distancia de Levenshtein entre xs e
-- ys. Por ejemplo,
--    levenshtein "casa"  "calle"    ==  3
--    levenshtein "calle" "casa"     ==  3
--    levenshtein "casa"  "casa"     ==  0
--    levenshtein "ana" "maria"      ==  3
--    levenshtein "agua" "manantial" ==  7
-- ---------------------------------------------------------------------

module Levenshtein where

import Data.Array(Array, (!), array)
import Test.Hspec (Spec, hspec, it, shouldBe)

-- 1ª definición (por recursión)
-- =============================

levenshtein1 :: String -> String -> Int
levenshtein1 "" ys = length ys
levenshtein1 xs "" = length xs
levenshtein1 c1@(x:xs) c2@(y:ys)
  | x == y    = levenshtein1 xs ys
  | otherwise = 1 + minimum [ levenshtein1 xs c2
                            , levenshtein1 c1 ys
                            , levenshtein1 xs ys]

-- 2ª definición (con programación dinámica)
```

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

levenshtein2 :: String -> String -> Int
levenshtein2 xs ys = matrizLevenshtein xs ys ! (m,n)
  where  m = length xs
         n = length ys

-- (matrizLevenshtein xs ys) es la matriz cuyo número de filas es la
-- longitud de xs, cuyo número de columnas es la longitud de ys y en
-- valor en la posición (i,j) es la distancia de Levenshtein entre los
-- primeros i caracteres de xs y los j primeros caracteres de ys. Por
-- ejemplo,
--    λ> elems (matrizLevenshtein "casa" "calle")
--    [0,1,2,3,4,5,1,0,1,2,3,4,2,1,0,1,2,3,3,2,1,1,2,3,4,3,2,2,2,3]
-- Gráficamente,
--       c a l l e
--     0,1,2,3,4,5,
--  c  1,0,1,2,3,4,
--  a  2,1,0,1,2,3,
--  s  3,2,1,1,2,3,
--  a  4,3,2,2,2,3
matrizLevenshtein :: String -> String -> Array (Int,Int) Int
matrizLevenshtein xs ys = q where
  q = array ((0,0),(m,n)) [((i,j), f i j) | i <- [0..m], j <- [0..n]]
  m = length xs
  n = length ys
  f 0 j = j
  f i 0 = i
  f i j | xs !! (i-1) == ys !! (j-1) = q ! (i-1,j-1)
        | otherwise                  = 1 + minimum [ q ! (i-1,j)
                                                    , q ! (i,j-1)
                                                    , q ! (i-1,j-1)]

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

-- La comparación es
--    λ> levenshtein1 (show (2^33)) (show (3^33))
--    12
--    (16.19 secs, 11,766,254,536 bytes)
```

```
--    λ> levenshtein2 (show (2^33)) (show (3^33))
--    12
--    (0.02 secs, 0 bytes)

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

spec :: Spec
spec = do
  it "ej1" $
    levenshtein1 "casa"  "calle"    `shouldBe`  3
  it "ej2" $
    levenshtein1 "calle" "casa"     `shouldBe`  3
  it "ej3" $
    levenshtein1 "casa"  "casa"     `shouldBe`  0
  it "ej4" $
    levenshtein1 "ana" "maria"      `shouldBe`  3
  it "ej5" $
    levenshtein1 "agua" "manantial" `shouldBe`  7
  it "ej6" $
    levenshtein2 "casa"  "calle"    `shouldBe`  3
  it "ej7" $
    levenshtein2 "calle" "casa"     `shouldBe`  3
  it "ej8" $
    levenshtein2 "casa"  "casa"     `shouldBe`  0
  it "ej9" $
    levenshtein2 "ana" "maria"      `shouldBe`  3
  it "ej10" $
    levenshtein2 "agua" "manantial" `shouldBe`  7

-- La verificación es
--    λ> verifica
--
--    ej1
--    ej2
--    ej3
--    ej4
```

```
--    ej5
--    ej6
--    ej7
--    ej8
--    ej9
--    ej10
--
--    Finished in 0.0024 seconds
--    10 examples, 0 failures
```

14.5.2. En Python

```
# ---------------------------------------------------------------------
# La distancia de Levenshtein (o distancia de edición) es el número
# mínimo de operaciones requeridas para transformar una cadena de
# caracteres en otra. Las operaciones de edición que se pueden hacer
# son:
# + insertar un carácter (por ejemplo, de "abc" a "abca")
# + eliminar un carácter (por ejemplo, de "abc" a "ac")
# + sustituir un carácter (por ejemplo, de "abc" a "adc")
#
# Por ejemplo, la distancia de Levenshtein entre "casa" y "calle" es de
# 3 porque se necesitan al menos tres ediciones elementales para
# cambiar uno en el otro:
#    "casa"  --> "cala"  (sustitución de 's' por 'l')
#    "cala"  --> "calla" (inserción de 'l' entre 'l' y 'a')
#    "calla" --> "calle" (sustitución de 'a' por 'e')
#
# Definir la función
#    levenshtein : (str, str) -> int
# tal que levenshtein(xs, ys) es la distancia de Levenshtein entre xs e
# ys. Por ejemplo,
#    levenshtein("casa",  "calle")     ==  3
#    levenshtein("calle", "casa")      ==  3
#    levenshtein("casa",  "casa")      ==  0
#    levenshtein("ana",   "maria")     ==  3
#    levenshtein("agua",  "manantial") ==  7
# ---------------------------------------------------------------------

from sys import setrecursionlimit
```

```
from timeit import Timer, default_timer

setrecursionlimit(10**6)

# 1ª definición (por recursión)
# =============================

def levenshtein1(xs: str, ys: str) -> int:
    if not xs:
        return len(ys)
    if not ys:
        return len(xs)
    if xs[0] == ys[0]:
        return levenshtein1(xs[1:], ys[1:])
    return 1 + min([levenshtein1(xs[1:], ys),
                    levenshtein1(xs, ys[1:]),
                    levenshtein1(xs[1:], ys[1:])])


# 2ª definición (con programación dinámica)
# =========================================

# matrizLevenshtein(xs, ys) es la matriz cuyo número de filas es la
# longitud de xs, cuyo número de columnas es la longitud de ys y en
# valor en la posición (i,j) es la distancia de Levenshtein entre los
# primeros i caracteres de xs y los j primeros caracteres de ys. Por
# ejemplo,
#    >>> matrizLevenshtein("casa", "calle")
#    [[0, 1, 2, 3, 4, 5],
#     [1, 0, 1, 2, 3, 4],
#     [2, 1, 0, 1, 2, 3],
#     [3, 2, 1, 1, 2, 3],
#     [4, 3, 2, 2, 2, 3]]
# Gráficamente,
#       c a l l e
#     0,1,2,3,4,5,
#  c  1,0,1,2,3,4,
#  a  2,1,0,1,2,3,
#  s  3,2,1,1,2,3,
#  a  4,3,2,2,2,3
```

```python
def matrizLevenshtein(xs: str, ys: str) -> list[list[int]]:
    n = len(xs)
    m = len(ys)
    q = [[0 for _ in range(m + 1)] for _ in range(n + 1)]
    for i in range(n + 1):
        q[i][0] = i
    for j in range(m + 1):
        q[0][j] = j
    for i in range(1,n + 1):
        for j in range(1, m + 1):
            if xs[i - 1] == ys[j - 1]:
                q[i][j] = q[i - 1][j - 1]
            else:
                q[i][j] = 1 + min([q[i-1][j],  q[i][j-1], q[i-1][j-1]])
    return q

def levenshtein2(xs: str, ys: str) -> int:
    m = len(xs)
    n = len(ys)
    return matrizLevenshtein(xs, ys)[m][n]

# 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('levenshtein1(str(2**33), str(3**33))')
#    13.78 segundos
#    >>> tiempo('levenshtein2(str(2**33), str(3**33))')
#    0.00 segundos

# Verificación
# ============

def test_levenshtein() -> None:
    assert levenshtein1("casa",  "calle")     ==  3
```

```
assert levenshtein1("calle", "casa")      ==  3
assert levenshtein1("casa",  "casa")      ==  0
assert levenshtein1("ana",   "maria")     ==  3
assert levenshtein1("agua",  "manantial") ==  7
assert levenshtein2("casa",  "calle")     ==  3
assert levenshtein2("calle", "casa")      ==  3
assert levenshtein2("casa",  "casa")      ==  0
assert levenshtein2("ana",   "maria")     ==  3
assert levenshtein2("agua",  "manantial") ==  7
print("Verificado")
```

14.6. Caminos en una retícula (con programación dinámica)

14.6.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Se considera una retícula con sus posiciones numeradas, desde el
-- vértice superior izquierdo, hacia la derecha y hacia abajo. Por
-- ejemplo, la retícula de dimensión 3x4 se numera como sigue:
--    |-------+-------+-------+-------|
--    | (1,1) | (1,2) | (1,3) | (1,4) |
--    | (2,1) | (2,2) | (2,3) | (2,4) |
--    | (3,1) | (3,2) | (3,3) | (3,4) |
--    |-------+-------+-------+-------|
--
-- Definir la función
--    caminos :: (Int,Int) -> [[(Int,Int)]]
-- tal que (caminos (m,n)) es la lista de los caminos en la retícula de
-- dimensión mxn desde (1,1) hasta (m,n). Por ejemplo,
--    λ> caminos (2,3)
--    [[(1,1),(1,2),(1,3),(2,3)],
--     [(1,1),(1,2),(2,2),(2,3)],
--     [(1,1),(2,1),(2,2),(2,3)]]
--    λ> mapM_ print (caminos (3,4))
--    [(1,1),(1,2),(1,3),(1,4),(2,4),(3,4)]
--    [(1,1),(1,2),(1,3),(2,3),(2,4),(3,4)]
--    [(1,1),(1,2),(2,2),(2,3),(2,4),(3,4)]
--    [(1,1),(2,1),(2,2),(2,3),(2,4),(3,4)]
```

```haskell
--     [(1,1),(1,2),(1,3),(2,3),(3,3),(3,4)]
--     [(1,1),(1,2),(2,2),(2,3),(3,3),(3,4)]
--     [(1,1),(2,1),(2,2),(2,3),(3,3),(3,4)]
--     [(1,1),(1,2),(2,2),(3,2),(3,3),(3,4)]
--     [(1,1),(2,1),(2,2),(3,2),(3,3),(3,4)]
--     [(1,1),(2,1),(3,1),(3,2),(3,3),(3,4)]
-- ---------------------------------------------------------------------

module Programacion_dinamica_Caminos_en_una_reticula where

import Data.Array (Array, (!), array)
import Test.Hspec (Spec, hspec, it, shouldBe)

-- 1ª solución (por recursión)
-- ===========================

caminos1 :: (Int,Int) -> [[(Int,Int)]]
caminos1 p = map reverse (caminos1Aux p)
  where
    caminos1Aux (1,y) = [[(1,z) | z <- [y,y-1..1]]]
    caminos1Aux (x,1) = [[(z,1) | z <- [x,x-1..1]]]
    caminos1Aux (x,y) = [(x,y) : cs | cs <- caminos1Aux (x-1,y) ++
                                            caminos1Aux (x,y-1)]

-- 2ª solución (con programación dinámica)
-- =======================================

caminos2 :: (Int,Int) -> [[(Int,Int)]]
caminos2 p = map reverse (matrizCaminos p ! p)

matrizCaminos :: (Int,Int) -> Array (Int,Int) [[(Int,Int)]]
matrizCaminos (m,n) = q
  where
    q = array ((1,1),(m,n)) [((i,j),f i j) | i <- [1..m], j <- [1..n]]
    f 1 y = [[(1,z) | z <- [y,y-1..1]]]
    f x 1 = [[(z,1) | z <- [x,x-1..1]]]
    f x y = [(x,y) : cs | cs <- q!(x-1,y) ++ q!(x,y-1)]

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

```
-- La comparación es
--    λ> maximum (head (caminos1 (2000,2000)))
--    (2000,2000)
--    (0.01 secs, 3,459,576 bytes)
--    λ> maximum (head (caminos2 (2000,2000)))
--    (2000,2000)
--    (2.79 secs, 1,507,636,688 bytes)

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

spec :: Spec
spec = do
  it "e1" $
    caminos1 (2,3) `shouldBe`
    [[(1,1),(1,2),(1,3),(2,3)],
     [(1,1),(1,2),(2,2),(2,3)],
     [(1,1),(2,1),(2,2),(2,3)]]
  it "e2" $
    caminos2 (2,3) `shouldBe`
    [[(1,1),(1,2),(1,3),(2,3)],
     [(1,1),(1,2),(2,2),(2,3)],
     [(1,1),(2,1),(2,2),(2,3)]]

-- La verificación es
--    λ> verifica
--
--    e1
--    e2
--
--    Finished in 0.0010 seconds
--    2 examples, 0 failures
```

14.6.2. En Python

```
# ---------------------------------------------------------------------
# Se considera una retícula con sus posiciones numeradas, desde el
# vértice superior izquierdo, hacia la derecha y hacia abajo. Por
# ejemplo, la retícula de dimensión 3x4 se numera como sigue:
#    |-------+-------+-------+-------|
#    | (1,1) | (1,2) | (1,3) | (1,4) |
#    | (2,1) | (2,2) | (2,3) | (2,4) |
#    | (3,1) | (3,2) | (3,3) | (3,4) |
#    |-------+-------+-------+-------|
#
# Definir la función
#    caminos : (tuple[int, int]) -> list[list[tuple[int, int]]]
# tal que caminos((m,n)) es la lista de los caminos en la retícula de
# dimensión mxn desde (1,1) hasta (m,n). Por ejemplo,
#    >>> caminos((2,3))
#    [[(1, 1), (1, 2), (1, 3), (2, 3)],
#     [(1, 1), (1, 2), (2, 2), (2, 3)],
#     [(1, 1), (2, 1), (2, 2), (2, 3)]]
#    >>> for c in caminos1((3,4)):
#    ...     print(c)
#    ...
#    [(1, 1), (1, 2), (1, 3), (1, 4), (2, 4), (3, 4)]
#    [(1, 1), (1, 2), (1, 3), (2, 3), (2, 4), (3, 4)]
#    [(1, 1), (1, 2), (2, 2), (2, 3), (2, 4), (3, 4)]
#    [(1, 1), (2, 1), (2, 2), (2, 3), (2, 4), (3, 4)]
#    [(1, 1), (1, 2), (1, 3), (2, 3), (3, 3), (3, 4)]
#    [(1, 1), (1, 2), (2, 2), (2, 3), (3, 3), (3, 4)]
#    [(1, 1), (2, 1), (2, 2), (2, 3), (3, 3), (3, 4)]
#    [(1, 1), (1, 2), (2, 2), (3, 2), (3, 3), (3, 4)]
#    [(1, 1), (2, 1), (2, 2), (3, 2), (3, 3), (3, 4)]
#    [(1, 1), (2, 1), (3, 1), (3, 2), (3, 3), (3, 4)]
# ---------------------------------------------------------------------

from collections import defaultdict
from sys import setrecursionlimit
from timeit import Timer, default_timer

setrecursionlimit(10**6)
```

```
# 1ª solución (por recursión)
# ===========================

def caminos1(p: tuple[int, int]) -> list[list[tuple[int, int]]]:
    def aux(p: tuple[int, int]) -> list[list[tuple[int, int]]]:
        (x, y) = p
        if x == 1:
            return [[(1,z) for z in range(y, 0, -1)]]
        if y == 1:
            return [[(z,1) for z in range(x, 0, -1)]]
        return [[(x,y)] + cs for cs in aux((x-1,y)) + aux((x,y-1))]

    return [list(reversed(ps)) for ps in aux(p)]

# 2ª solución (con programación dinámica)
# =======================================

def caminos2(p: tuple[int, int]) -> list[list[tuple[int, int]]]:
    return [list(reversed(ps)) for ps in diccionarioCaminos(p)[p]]

# diccionarioCaminos((m,n)) es el diccionario cuyas claves son los
# puntos de la retícula mxn y sus valores son los caminos a dichos
# puntos. Por ejemplo,
#    >>> diccionarioCaminos((2,3))
#    defaultdict(<class 'list'>,
#                {(1,1): [[(1,1)]],
#                 (1,2): [[(1,2),(1,1)]],
#                 (1,3): [[(1,3),(1,2),(1,1)]],
#                 (2,1): [[(2,1),(1,1)]],
#                 (2,2): [[(2,2),(1,2),(1,1)],
#                         [(2,2),(2,1),(1,1)]],
#                 (2,3): [[(2,3),(1,3),(1,2),(1,1)],
#                         [(2,3),(2,2),(1,2),(1,1)],
#                         [(2,3),(2,2),(2,1),(1,1)]]})
def diccionarioCaminos(p: tuple[int, int]) -> dict[tuple[int, int], list[list[tup
    m, n = p
    q = defaultdict(list)
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if i == 1:
```

```
                q[(i, j)] = [[(1, z) for z in range(j, 0, -1)]]
            elif j == 1:
                q[(i, j)] = [[(z, 1) for z in range(i, 0, -1)]]
            else:
                q[(i, j)] = [[(i, j)] + cs for cs in q[(i-1, j)] + q[(i, j-1)]]
    return q

# 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('max(caminos1((13,13))[0])')
#    26.75 segundos
#    >>> tiempo('max(caminos2((13,13))[0])')
#    7.40 segundos

# Verificación
# ============

def test_caminos() -> None:
    assert caminos1((2,3)) == \
        [[(1,1),(1,2),(1,3),(2,3)],
         [(1,1),(1,2),(2,2),(2,3)],
         [(1,1),(2,1),(2,2),(2,3)]]
    assert caminos2((2,3)) == \
        [[(1,1),(1,2),(1,3),(2,3)],
         [(1,1),(1,2),(2,2),(2,3)],
         [(1,1),(2,1),(2,2),(2,3)]]
    print("Verificado")

# La verificación es
#    >>> test_caminos()
#    Verificado
```

14.7. Caminos en una matriz (con programación dinámica)

14.7.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Los caminos desde el extremo superior izquierdo (posición (1,1))
-- hasta el extremo inferior derecho (posición (3,4)) en la matriz
--    (  1  6 11  2 )
--    (  7 12  3  8 )
--    (  3  8  4  9 )
-- moviéndose en cada paso una casilla hacia la derecha o abajo, son los
-- siguientes:
--    [1,6,11,2,8,9]
--    [1,6,11,3,8,9]
--    [1,6,12,3,8,9]
--    [1,7,12,3,8,9]
--    [1,6,11,3,4,9]
--    [1,6,12,3,4,9]
--    [1,7,12,3,4,9]
--    [1,6,12,8,4,9]
--    [1,7,12,8,4,9]
--    [1,7, 3,8,4,9]
--
-- Definir la función
--    caminos :: Matrix Int -> [[Int]]
-- tal que (caminos m) es la lista de los caminos en la matriz m desde
-- el extremo superior izquierdo hasta el extremo inferior derecho,
-- moviéndose en cada paso una casilla hacia abajo o hacia la
-- derecha. Por ejemplo,
--    λ> caminos (fromLists [[1,6,11,2],[7,12,3,8],[3,8,4,9]])
--    [[1,6,11,2,8,9],
--     [1,6,11,3,8,9],
--     [1,6,12,3,8,9],
--     [1,7,12,3,8,9],
--     [1,6,11,3,4,9],
--     [1,6,12,3,4,9],
--     [1,7,12,3,4,9],
--     [1,6,12,8,4,9],
--     [1,7,12,8,4,9],
```

```
--     [1,7, 3,8,4,9]]
--    λ> length (caminos (fromList 12 12 [1..]))
--    705432
-- ---------------------------------------------------------------------

module Caminos_en_una_matriz where

import Data.Matrix (Matrix, (!), fromLists, matrix, nrows, ncols)
import Test.Hspec (Spec, hspec, it, shouldBe)

-- 1ª definición (por recursión)
-- =============================

caminos1 :: Matrix Int -> [[Int]]
caminos1 m =
  reverse (map reverse (caminos1Aux m (nf,nc)))
  where nf = nrows m
        nc = ncols m

-- (caminos1Aux m p) es la lista de los caminos invertidos en la matriz m
-- desde la posición (1,1) hasta la posición p. Por ejemplo,
caminos1Aux :: Matrix Int -> (Int,Int) -> [[Int]]
caminos1Aux m (1,1) = [[m!(1,1)]]
caminos1Aux m (1,j) = [[m!(1,k) | k <- [j,j-1..1]]]
caminos1Aux m (i,1) = [[m!(k,1) | k <- [i,i-1..1]]]
caminos1Aux m (i,j) = [m!(i,j) : xs
                      | xs <- caminos1Aux m (i,j-1) ++
                              caminos1Aux m (i-1,j)]

-- 2ª solución (mediante programación dinámica)
-- ============================================

caminos2 :: Matrix Int -> [[Int]]
caminos2 m =
  map reverse (matrizCaminos m ! (nrows m, ncols m))

matrizCaminos :: Matrix Int -> Matrix [[Int]]
matrizCaminos m = q
  where
    q = matrix (nrows m) (ncols m) f
```

```
    f (1,y) = [[m!(1,z) | z <- [y,y-1..1]]]
    f (x,1) = [[m!(z,1) | z <- [x,x-1..1]]]
    f (x,y) = [m!(x,y) : cs | cs <- q!(x-1,y) ++ q!(x,y-1)]

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

-- La comparación es
--    λ> length (caminos1 (fromList 11 11 [1..]))
--    184756
--    (3.64 secs, 667,727,568 bytes)
--    λ> length (caminos2 (fromList 11 11 [1..]))
--    184756
--    (0.82 secs, 129,181,072 bytes)

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

spec :: Spec
spec = do
  it "e1" $
    caminos1 (fromLists [[1,6,11,2],[7,12,3,8],[3,8,4,9]]) `shouldBe` r
  it "e2" $
    caminos2 (fromLists [[1,6,11,2],[7,12,3,8],[3,8,4,9]]) `shouldBe` r
  where r = [[1,6,11,2,8,9],
             [1,6,11,3,8,9],
             [1,6,12,3,8,9],
             [1,7,12,3,8,9],
             [1,6,11,3,4,9],
             [1,6,12,3,4,9],
             [1,7,12,3,4,9],
             [1,6,12,8,4,9],
             [1,7,12,8,4,9],
             [1,7, 3,8,4,9]]

-- La verificación es
--    λ> verifica
```

```
--
--    e1
--    e2
--
--    Finished in 0.0010 seconds
--    2 examples, 0 failures
```

14.7.2. En Python

```
# ---------------------------------------------------------------------
# Los caminos desde el extremo superior izquierdo (posición (1,1))
# hasta el extremo inferior derecho (posición (3,4)) en la matriz
#    (  1  6 11  2 )
#    (  7 12  3  8 )
#    (  3  8  4  9 )
# moviéndose en cada paso una casilla hacia la derecha o abajo, son los
# siguientes:
#    [1,6,11,2,8,9]
#    [1,6,11,3,8,9]
#    [1,6,12,3,8,9]
#    [1,7,12,3,8,9]
#    [1,6,11,3,4,9]
#    [1,6,12,3,4,9]
#    [1,7,12,3,4,9]
#    [1,6,12,8,4,9]
#    [1,7,12,8,4,9]
#    [1,7, 3,8,4,9]
#
# Definir la función
#    caminos : (list[list[int]]) -> list[list[int]]
# tal que caminos (m) es la lista de los caminos en la matriz m desde
# el extremo superior izquierdo hasta el extremo inferior derecho,
# moviéndose en cada paso una casilla hacia abajo o hacia la
# derecha. Por ejemplo,
#    >>> caminos([[1,6,11,2],[7,12,3,8],[3,8,4,9]])
#    [[1, 6, 11, 2, 8, 9],
#     [1, 6, 11, 3, 8, 9],
#     [1, 6, 12, 3, 8, 9],
#     [1, 7, 12, 3, 8, 9],
#     [1, 6, 11, 3, 4, 9],
```

```
#     [1, 6, 12, 3, 4, 9],
#     [1, 7, 12, 3, 4, 9],
#     [1, 6, 12, 8, 4, 9],
#     [1, 7, 12, 8, 4, 9],
#     [1, 7,  3, 8, 4, 9]]
#    >>> len(caminos([list(range(12*n+1, 12*(n+1)+1)) for n in range(12)]))
#    705432
# ---------------------------------------------------------------------

from collections import defaultdict
from sys import setrecursionlimit
from timeit import Timer, default_timer

setrecursionlimit(10**6)

# 1ª definición (por recursión)
# ===============================

def caminos1(m: list[list[int]]) -> list[list[int]]:
    nf = len(m)
    nc = len(m[0])
    return list(reversed([list(reversed(xs)) for xs in caminos1Aux(m, (nf,nc))]))

# caminos1Aux(m, p) es la lista de los caminos invertidos en la matriz m
# desde la posición (1,1) hasta la posición p. Por ejemplo,
def caminos1Aux(m: list[list[int]], p: tuple[int, int]) -> list[list[int]]:
    (i, j) = p
    if p == (1,1):
        return [[m[0][0]]]
    if i == 1:
        return [[m[0][k-1] for k in range(j, 0, -1)]]
    if j == 1:
        return [[m[k-1][0] for k in range(i, 0, -1)]]
    return [[m[i-1][j-1]] + xs
            for xs in caminos1Aux(m, (i,j-1)) + caminos1Aux(m, (i-1,j))]

# 2ª solución (mediante programación dinámica)
# ==============================================

def caminos2(p: list[list[int]]) -> list[list[int]]:
```

```
    m = len(p)
    n = len(p[0])
    return [list(reversed(xs)) for xs in diccionarioCaminos(p)[(m, n)]]

# diccionarioCaminos(p) es el diccionario cuyas claves son los
# puntos de la matriz p y sus valores son los caminos a dichos
# puntos. Por ejemplo,
#    >>> diccionarioCaminos([[1,6,11,2],[7,12,3,8],[3,8,4,9]])
#    defaultdict(<class 'list'>,
#                {(1, 1): [[1]],
#                 (1, 2): [[6, 1]],
#                 (1, 3): [[11, 6, 1]],
#                 (1, 4): [[2, 11, 6, 1]],
#                 (2, 1): [[7, 1]],
#                 (2, 2): [[12, 6, 1], [12, 7, 1]],
#                 (2, 3): [[3, 11, 6, 1], [3, 12, 6, 1], [3, 12, 7, 1]],
#                 (2, 4): [[8, 2, 11, 6, 1], [8, 3, 11, 6, 1],
#                          [8, 3, 12, 6, 1], [8, 3, 12, 7, 1]],
#                 (3, 1): [[3, 7, 1]],
#                 (3, 2): [[8, 12, 6, 1], [8, 12, 7, 1], [8, 3, 7, 1]],
#                 (3, 3): [[4, 3, 11, 6, 1], [4, 3, 12, 6, 1],
#                          [4, 3, 12, 7, 1], [4, 8, 12, 6, 1],
#                          [4, 8, 12, 7, 1], [4, 8, 3, 7, 1]],
#                 (3, 4): [[9, 8, 2, 11, 6, 1], [9, 8, 3, 11, 6, 1],
#                          [9, 8, 3, 12, 6, 1], [9, 8, 3, 12, 7, 1],
#                          [9, 4, 3, 11, 6, 1], [9, 4, 3, 12, 6, 1],
#                          [9, 4, 3, 12, 7, 1], [9, 4, 8, 12, 6, 1],
#                          [9, 4, 8, 12, 7, 1], [9, 4, 8, 3, 7, 1]]})
def diccionarioCaminos(p: list[list[int]]) -> dict[tuple[int, int], list[list[int
    m = len(p)
    n = len(p[0])
    q = defaultdict(list)
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if i == 1:
                q[(i, j)] = [[p[0][z-1] for z in range(j, 0, -1)]]
            elif j == 1:
                q[(i, j)] = [[p[z-1][0] for z in range(i, 0, -1)]]
            else:
                q[(i, j)] = [[p[i-1][j-1]] + cs for cs in q[(i-1, j)] + q[(i, j-1
```

```
    return q

# 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('caminos1([list(range(11*n+1, 11*(n+1)+1)) for n in range(12)])')
#    2.20 segundos
#    >>> tiempo('caminos2([list(range(11*n+1, 11*(n+1)+1)) for n in range(12)])')
#    0.64 segundos

# Verificación
# ============

def test_caminos() -> None:
    r = [[1, 6, 11, 2, 8, 9],
         [1, 6, 11, 3, 8, 9],
         [1, 6, 12, 3, 8, 9],
         [1, 7, 12, 3, 8, 9],
         [1, 6, 11, 3, 4, 9],
         [1, 6, 12, 3, 4, 9],
         [1, 7, 12, 3, 4, 9],
         [1, 6, 12, 8, 4, 9],
         [1, 7, 12, 8, 4, 9],
         [1, 7,  3, 8, 4, 9]]
    assert caminos1([[1,6,11,2],[7,12,3,8],[3,8,4,9]]) == r
    assert caminos2([[1,6,11,2],[7,12,3,8],[3,8,4,9]]) == r
    print("Verificado")

# La verificación es
#    >>> test_caminos()
#    Verificado
```

14.8. Máxima suma de los caminos en una matriz

14.8.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Los caminos desde el extremo superior izquierdo (posición (1,1))
-- hasta el extremo inferior derecho (posición (3,4)) en la matriz
--    (  1  6 11  2 )
--    (  7 12  3  8 )
--    (  3  8  4  9 )
-- moviéndose en cada paso una casilla hacia la derecha o hacia abajo,
-- son los siguientes:
--    [1,6,11,2,8,9]
--    [1,6,11,3,8,9]
--    [1,6,12,3,8,9]
--    [1,7,12,3,8,9]
--    [1,6,11,3,4,9]
--    [1,6,12,3,4,9]
--    [1,7,12,3,4,9]
--    [1,6,12,8,4,9]
--    [1,7,12,8,4,9]
--    [1,7, 3,8,4,9]
-- La suma de los caminos son 37, 38, 39, 40, 34, 35, 36, 40, 41 y 32,
-- respectivamente. El camino de máxima suma es el penúltimo (1, 7, 12, 8,
-- 4, 9) que tiene una suma de 41.
--
-- Definir la función
--    maximaSuma :: Matrix Int -> Int
-- tal que (maximaSuma m) es el máximo de las sumas de los caminos en la
-- matriz m desde el extremo superior izquierdo hasta el extremo
-- inferior derecho, moviéndose en cada paso una casilla hacia abajo o
-- hacia la derecha. Por ejemplo,
--    λ> maximaSuma (fromLists [[1,6,11,2],[7,12,3,8],[3,8,4,9]])
--    41
--    λ> maximaSuma (fromList 800 800 [1..])
--    766721999
-- ---------------------------------------------------------------------

module Maxima_suma_de_los_caminos_en_una_matriz where
```

```haskell
import Data.Matrix (Matrix, (!), fromList, fromLists, matrix, nrows, ncols)
import Test.Hspec (Spec, hspec, it, shouldBe)
import Caminos_en_una_matriz (caminos1, caminos2)

-- 1ª definicion de maximaSuma (con caminos1)
-- ==========================================

maximaSuma1 :: Matrix Int -> Int
maximaSuma1 =
  maximum . map sum . caminos1

-- Se usará la función caminos1 del ejercicio
-- "Caminos en una matriz" que se encuentra en
-- https://bit.ly/45bYoYE


-- 2ª definición de maximaSuma (con caminos2)
-- ==========================================

maximaSuma2 :: Matrix Int -> Int
maximaSuma2 =
  maximum . map sum . caminos2

-- Se usará la función caminos2 del ejercicio
-- "Caminos en una matriz" que se encuentra en
-- https://bit.ly/45bYoYE

-- 3ª definicion de maximaSuma (por recursión)
-- ===========================================

maximaSuma3 :: Matrix Int -> Int
maximaSuma3 m = maximaSuma3Aux m (nf,nc)
  where nf = nrows m
        nc = ncols m

-- (maximaSuma3Aux m p) calcula la suma máxima de un camino hasta la
-- posición p. Por ejemplo,
--    λ> maximaSuma3Aux (fromLists [[1,6,11,2],[7,12,3,8],[3,8,4,9]]) (3,4)
--    41
```

```haskell
--    λ> maximaSuma3Aux (fromLists [[1,6,11,2],[7,12,3,8],[3,8,4,9]]) (3,3)
--    32
--    λ> maximaSuma3Aux (fromLists [[1,6,11,2],[7,12,3,8],[3,8,4,9]]) (2,4)
--    31
maximaSuma3Aux :: Matrix Int -> (Int,Int) -> Int
maximaSuma3Aux m (1,1) = m ! (1,1)
maximaSuma3Aux m (1,j) = maximaSuma3Aux m (1,j-1) + m ! (1,j)
maximaSuma3Aux m (i,1) = maximaSuma3Aux m (i-1,1) + m ! (i,1)
maximaSuma3Aux m (i,j) =
  max (maximaSuma3Aux m (i,j-1)) (maximaSuma3Aux m (i-1,j)) + m ! (i,j)

-- 4ª solución (mediante programación dinámica)
-- ===============================================

maximaSuma4 :: Matrix Int -> Int
maximaSuma4 m = q ! (nf,nc)
  where nf = nrows m
        nc = ncols m
        q  = matrizMaximaSuma m

-- (matrizMaximaSuma m) es la matriz donde en cada posición p se
-- encuentra el máxima de las sumas de los caminos desde (1,1) a p en la
-- matriz m. Por ejemplo,
--    λ> matrizMaximaSuma (fromLists [[1,6,11,2],[7,12,3,8],[3,8,4,9]])
--    (  1  7 18 20 )
--    (  8 20 23 31 )
--    ( 11 28 32 41 )
matrizMaximaSuma :: Matrix Int -> Matrix Int
matrizMaximaSuma m = q
  where nf = nrows m
        nc = ncols m
        q  = matrix nf nc f
          where  f (1,1) = m ! (1,1)
                 f (1,j) = q ! (1,j-1) + m ! (1,j)
                 f (i,1) = q ! (i-1,1) + m ! (i,1)
                 f (i,j) = max (q ! (i,j-1)) (q ! (i-1,j)) + m ! (i,j)

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

```
-- La comparación es
--    λ> maximaSuma1 (fromList 11 11 [1..])
--    1781
--    (3.88 secs, 1,525,812,680 bytes)
--    λ> maximaSuma2 (fromList 11 11 [1..])
--    1781
--    (1.08 secs, 546,144,264 bytes)
--    λ> maximaSuma3 (fromList 11 11 [1..])
--    1781
--    (0.55 secs, 217,712,280 bytes)
--    λ> maximaSuma4 (fromList 11 11 [1..])
--    1781
--    (0.01 secs, 643,832 bytes)

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

spec :: Spec
spec = do
  it "e1" $
    maximaSuma1 (fromLists [[1,6,11,2],[7,12,3,8],[3,8,4,9]]) `shouldBe` 41
  it "e2" $
    maximaSuma1 (fromList 4 4 [1..]) `shouldBe` 73
  it "e3" $
    maximaSuma2 (fromLists [[1,6,11,2],[7,12,3,8],[3,8,4,9]]) `shouldBe` 41
  it "e4" $
    maximaSuma2 (fromList 4 4 [1..]) `shouldBe` 73
  it "e5" $
    maximaSuma3 (fromLists [[1,6,11,2],[7,12,3,8],[3,8,4,9]]) `shouldBe` 41
  it "e6" $
    maximaSuma3 (fromList 4 4 [1..]) `shouldBe` 73
  it "e7" $
    maximaSuma4 (fromLists [[1,6,11,2],[7,12,3,8],[3,8,4,9]]) `shouldBe` 41
  it "e8" $
    maximaSuma4 (fromList 4 4 [1..]) `shouldBe` 73

-- La verificación es
```

```
--    λ> verifica
--
--    e1
--    e2
--    e3
--    e4
--    e5
--    e6
--    e7
--    e8
--
--    Finished in 0.0034 seconds
--    8 examples, 0 failures
```

14.8.2. En Python

```
# ---------------------------------------------------------------------
# Los caminos desde el extremo superior izquierdo (posición (1,1))
# hasta el extremo inferior derecho (posición (3,4)) en la matriz
#    (  1  6 11  2 )
#    (  7 12  3  8 )
#    (  3  8  4  9 )
# moviéndose en cada paso una casilla hacia la derecha o hacia abajo,
# son los siguientes:
#    [1,6,11,2,8,9]
#    [1,6,11,3,8,9]
#    [1,6,12,3,8,9]
#    [1,7,12,3,8,9]
#    [1,6,11,3,4,9]
#    [1,6,12,3,4,9]
#    [1,7,12,3,4,9]
#    [1,6,12,8,4,9]
#    [1,7,12,8,4,9]
#    [1,7, 3,8,4,9]
# La suma de los caminos son 37, 38, 39, 40, 34, 35, 36, 40, 41 y 32,
# respectivamente. El camino de máxima suma es el penúltimo (1, 7, 12, 8,
# 4, 9) que tiene una suma de 41.
#
# Definir la función
#    maximaSuma : (list[list[int]]) -> int
```

```
# tal que maximaSuma(m) es el máximo de las sumas de los caminos en la
# matriz m desde el extremo superior izquierdo hasta el extremo
# inferior derecho, moviéndose en cada paso una casilla hacia abajo o
# hacia la derecha. Por ejemplo,
#    >>> maximaSuma([[1,6,11,2],[7,12,3,8],[3,8,4,9]])
#    41
#    >>> maximaSuma4([list(range(800*n+1, 800*(n+1)+1)) for n in range(800)])
#    766721999
# ---------------------------------------------------------------------

from collections import defaultdict
from sys import setrecursionlimit
from timeit import Timer, default_timer

from src.Caminos_en_una_matriz import caminos1, caminos2

setrecursionlimit(10**6)

# 1ª definicion de maximaSuma (con caminos1)
# ==========================================

def maximaSuma1(m: list[list[int]]) -> int:
    return max((sum(xs) for xs in caminos1(m)))

# Se usará la función caminos1 del ejercicio
# "Caminos en una matriz" que se encuentra en
# https://bit.ly/45bYoYE

# 2ª definición de maximaSuma (con caminos2)
# ==========================================

def maximaSuma2(m: list[list[int]]) -> int:
    return max((sum(xs) for xs in caminos2(m)))

# Se usará la función caminos2 del ejercicio
# "Caminos en una matriz" que se encuentra en
# https://bit.ly/45bYoYE

# 3ª definicion de maximaSuma (por recursión)
# ===========================================
```

```
def maximaSuma3(m: list[list[int]]) -> int:
    nf = len(m)
    nc = len(m[0])
    return maximaSuma3Aux(m, (nf,nc))

# (maximaSuma3Aux m p) calcula la suma máxima de un camino hasta la
# posición p. Por ejemplo,
#    λ> maximaSuma3Aux (fromLists [[1,6,11,2],[7,12,3,8],[3,8,4,9]]) (3,4)
#    41
#    λ> maximaSuma3Aux (fromLists [[1,6,11,2],[7,12,3,8],[3,8,4,9]]) (3,3)
#    32
#    λ> maximaSuma3Aux (fromLists [[1,6,11,2],[7,12,3,8],[3,8,4,9]]) (2,4)
#    31
def maximaSuma3Aux(m: list[list[int]], p: tuple[int, int]) -> int:
    (i, j) = p
    if (i, j) == (1, 1):
        return m[0][0]
    if i == 1:
        return maximaSuma3Aux(m, (1,j-1)) + m[0][j-1]
    if j == 1:
        return maximaSuma3Aux(m, (i-1,1)) + m[i-1][0]
    return max(maximaSuma3Aux(m, (i,j-1)), maximaSuma3Aux(m, (i-1,j))) + m[i-1][j

# 4ª solución (mediante programación dinámica)
# ============================================

def maximaSuma4(p: list[list[int]]) -> int:
    m = len(p)
    n = len(p[0])
    return diccionarioMaxSuma(p)[(m,n)]

# diccionarioMaxSuma(p) es el diccionario cuyas claves son los
# puntos de la matriz p y sus valores son las máximas sumas de los
# caminos a dichos puntos. Por ejemplo,
#    diccionarioMaxSuma([[1,6,11,2],[7,12,3,8],[3,8,4,9]])
#    defaultdict(<class 'int'>,
#                {(1, 0): 0,
#                 (1, 1): 1,  (1, 2): 7,  (1, 3): 18, (1, 4): 20,
#                 (2, 1): 8,  (2, 2): 20, (2, 3): 23, (2, 4): 31,
```

```
#                  (3, 1): 11, (3, 2): 28, (3, 3): 32, (3, 4): 41})
def diccionarioMaxSuma(p: list[list[int]]) -> dict[tuple[int, int], int]:
    m = len(p)
    n = len(p[0])
    q: dict[tuple[int, int], int] = defaultdict(int)
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if i == 1:
                q[(i, j)] = q[(1,j-1)] + p[0][j-1]
            elif j == 1:
                q[(i, j)] = q[(i-1,1)] + p[i-1][0]
            else:
                q[(i, j)] = max(q[(i,j-1)], q[(i-1,j)]) + p[i-1][j-1]
    return q

# 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('maximaSuma1([list(range(12*n+1, 12*(n+1)+1)) for n in range(12)]
#    4.95 segundos
#    >>> tiempo('maximaSuma2([list(range(12*n+1, 12*(n+1)+1)) for n in range(12)]
#    1.49 segundos
#    >>> tiempo('maximaSuma3([list(range(12*n+1, 12*(n+1)+1)) for n in range(12)]
#    0.85 segundos
#    >>> tiempo('maximaSuma4([list(range(12*n+1, 12*(n+1)+1)) for n in range(12)]
#    0.00 segundos

# Verificación
# ============

def test_maximaSuma() -> None:
    assert maximaSuma1([[1,6,11,2],[7,12,3,8],[3,8,4,9]]) == 41
    assert maximaSuma2([[1,6,11,2],[7,12,3,8],[3,8,4,9]]) == 41
    assert maximaSuma3([[1,6,11,2],[7,12,3,8],[3,8,4,9]]) == 41
```

```
    assert maximaSuma4([[1,6,11,2],[7,12,3,8],[3,8,4,9]]) == 41
    print("Verificado")

# La verificación es
#    >>> test_maximaSuma()
#    Verificado
```

14.9. Camino de máxima suma en una matriz

14.9.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Los caminos desde el extremo superior izquierdo (posición (1,1))
-- hasta el extremo inferior derecho (posición (3,4)) en la matriz
--    (  1  6 11  2 )
--    (  7 12  3  8 )
--    (  3  8  4  9 )
-- moviéndose en cada paso una casilla hacia la derecha o hacia abajo,
-- son los siguientes:
--    [1,6,11,2,8,9]
--    [1,6,11,3,8,9]
--    [1,6,12,3,8,9]
--    [1,7,12,3,8,9]
--    [1,6,11,3,4,9]
--    [1,6,12,3,4,9]
--    [1,7,12,3,4,9]
--    [1,6,12,8,4,9]
--    [1,7,12,8,4,9]
--    [1,7, 3,8,4,9]
-- La suma de los caminos son 37, 38, 39, 40, 34, 35, 36, 40, 41 y 32,
-- respectivamente. El camino de máxima suma es el penúltimo (1, 7, 12, 8,
-- 4, 9) que tiene una suma de 41.
--
-- Definir la función
--    caminoMaxSuma :: Matrix Int -> [Int]
-- tal que (caminoMaxSuma m) es un camino de máxima suma en la matriz m
-- desde el extremo superior izquierdo hasta el extremo inferior derecho,
-- moviéndose en cada paso una casilla hacia abajo o hacia la
-- derecha. Por ejemplo,
--    λ> caminoMaxSuma (fromLists [[1,6,11,2],[7,12,3,8],[3,8,4,9]])
```

```
--    [1,7,12,8,4,9]
--    λ> sum (caminoMaxSuma (fromList 500 500 [1..]))
--    187001249
-- ---------------------------------------------------------------------

module Camino_de_maxima_suma_en_una_matriz where

import Data.Matrix (Matrix, (!), fromLists, matrix, nrows, ncols)
import Test.Hspec (Spec, hspec, it, shouldBe)

-- 1ª definición de caminoMaxSuma (con caminos1)
-- ==============================================

caminoMaxSuma1 :: Matrix Int -> [Int]
caminoMaxSuma1 m =
  head [c | c <- cs, sum c == k]
  where cs = caminos1 m
        k  = maximum (map sum cs)

caminos1 :: Matrix Int -> [[Int]]
caminos1 m =
  reverse (map reverse (caminos1Aux m (nf,nc)))
  where nf = nrows m
        nc = ncols m

-- (caminos1Aux m p) es la lista de los caminos invertidos en la matriz m
-- desde la posición (1,1) hasta la posición p. Por ejemplo,
caminos1Aux :: Matrix Int -> (Int,Int) -> [[Int]]
caminos1Aux m (1,1) = [[m!(1,1)]]
caminos1Aux m (1,j) = [[m!(1,k) | k <- [j,j-1..1]]]
caminos1Aux m (i,1) = [[m!(k,1) | k <- [i,i-1..1]]]
caminos1Aux m (i,j) = [m!(i,j) : xs
                      | xs <- caminos1Aux m (i,j-1) ++
                              caminos1Aux m (i-1,j)]

-- 2ª definición de caminoMaxSuma (con caminos2)
-- ==============================================

caminoMaxSuma2 :: Matrix Int -> [Int]
caminoMaxSuma2 m =
```

```haskell
  head [c | c <- cs, sum c == k]
  where cs = caminos2 m
        k  = maximum (map sum cs)

caminos2 :: Matrix Int -> [[Int]]
caminos2 m =
  map reverse (matrizCaminos m ! (nrows m, ncols m))

matrizCaminos :: Matrix Int -> Matrix [[Int]]
matrizCaminos m = q
  where
    q = matrix (nrows m) (ncols m) f
    f (1,y) = [[m!(1,z) | z <- [y,y-1..1]]]
    f (x,1) = [[m!(z,1) | z <- [x,x-1..1]]]
    f (x,y) = [m!(x,y) : cs | cs <- q!(x-1,y) ++ q!(x,y-1)]

-- 3ª definición de caminoMaxSuma (con programación dinámica)
-- ============================================================

caminoMaxSuma3 :: Matrix Int -> [Int]
caminoMaxSuma3 m = reverse (snd (q ! (nf,nc)))
  where nf = nrows m
        nc = ncols m
        q  = caminoMaxSumaAux m

caminoMaxSumaAux :: Matrix Int -> Matrix (Int,[Int])
caminoMaxSumaAux m = q
  where
    nf = nrows m
    nc = ncols m
    q  = matrix nf nc f
      where
        f (1,1) = (m!(1,1),[m!(1,1)])
        f (1,j) = (k + m!(1,j), m!(1,j):xs)
          where (k,xs) = q!(1,j-1)
        f (i,1) = (k + m!(i,1), m!(i,1):xs)
          where (k,xs) = q!(i-1,1)
        f (i,j) | k1 > k2   = (k1 + m!(i,j), m!(i,j):xs)
                | otherwise = (k2 + m!(i,j), m!(i,j):ys)
          where (k1,xs) = q!(i,j-1)
```

```
                (k2,ys) = q!(i-1,j)

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

-- La comparación es
--    λ> length (caminoMaxSuma1 (fromList 11 11 [1..]))
--    21
--    (3.92 secs, 1,778,557,904 bytes)
--    λ> length (caminoMaxSuma2 (fromList 11 11 [1..]))
--    21
--    (1.16 secs, 798,889,488 bytes)
--    λ> length (caminoMaxSuma3 (fromList 11 11 [1..]))
--    21
--    (0.00 secs, 680,256 bytes)

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

spec :: Spec
spec = do
  it "e1" $
    caminoMaxSuma1 (fromLists [[1,6,11,2],[7,12,3,8],[3,8,4,9]])
    `shouldBe` [1,7,12,8,4,9]
  it "e2" $
    caminoMaxSuma2 (fromLists [[1,6,11,2],[7,12,3,8],[3,8,4,9]])
    `shouldBe` [1,7,12,8,4,9]
  it "e3" $
    caminoMaxSuma3 (fromLists [[1,6,11,2],[7,12,3,8],[3,8,4,9]])
    `shouldBe` [1,7,12,8,4,9]

-- La verificación es
--    λ> verifica
--
--    e1
--    e2
--    e3
```

```
--
--    Finished in 0.0007 seconds
--    3 examples, 0 failures
```

14.9.2. En Python

```
# ---------------------------------------------------------------------
# Los caminos desde el extremo superior izquierdo (posición (1,1))
# hasta el extremo inferior derecho (posición (3,4)) en la matriz
#    (  1  6 11  2 )
#    (  7 12  3  8 )
#    (  3  8  4  9 )
# moviéndose en cada paso una casilla hacia la derecha o hacia abajo,
# son los siguientes:
#    [1,6,11,2,8,9]
#    [1,6,11,3,8,9]
#    [1,6,12,3,8,9]
#    [1,7,12,3,8,9]
#    [1,6,11,3,4,9]
#    [1,6,12,3,4,9]
#    [1,7,12,3,4,9]
#    [1,6,12,8,4,9]
#    [1,7,12,8,4,9]
#    [1,7, 3,8,4,9]
# La suma de los caminos son 37, 38, 39, 40, 34, 35, 36, 40, 41 y 32,
# respectivamente. El camino de máxima suma es el penúltimo (1, 7, 12, 8,
# 4, 9) que tiene una suma de 41.
#
# Definir la función
#    caminoMaxSuma : (list[list[int]]) -> list[int]
# tal que caminoMaxSuma(m) es un camino de máxima suma en la matriz m
# desde el extremo superior izquierdo hasta el extremo inferior derecho,
# moviéndose en cada paso una casilla hacia abajo o hacia la
# derecha. Por ejemplo,
#    >>> caminoMaxSuma1([[1,6,11,2],[7,12,3,8],[3,8,4,9]])
#    [1, 7, 12, 8, 4, 9]
#    >>> sum(caminoMaxSuma3([list(range(500*n+1, 500*(n+1)+1)) for n in range(500
#    187001249
# ---------------------------------------------------------------------
```

```
from collections import defaultdict
from sys import setrecursionlimit
from timeit import Timer, default_timer

from src.Caminos_en_una_matriz import caminos1, caminos2

setrecursionlimit(10**6)

# 1ª definición de caminoMaxSuma (con caminos1)
# =============================================

def caminoMaxSuma1(m: list[list[int]]) -> list[int]:
    cs = caminos1(m)
    k = max((sum(c) for c in cs))
    return [c for c in cs if sum(c) == k][0]

# Se usa la función caminos1 del ejercicio
# "Caminos en una matriz" que se encuentra en
# https://bit.ly/45bYoYE

# 2ª definición de caminoMaxSuma (con caminos2)
# =============================================

def caminoMaxSuma2(m: list[list[int]]) -> list[int]:
    cs = caminos2(m)
    k = max((sum(c) for c in cs))
    return [c for c in cs if sum(c) == k][0]

# Se usa la función caminos2 del ejercicio
# "Caminos en una matriz" que se encuentra en
# https://bit.ly/45bYoYE

# 3ª definición de caminoMaxSuma (con programación dinámica)
# ==========================================================

def caminoMaxSuma3(m: list[list[int]]) -> list[int]:
    nf = len(m)
    nc = len(m[0])
    return list(reversed(diccionarioCaminoMaxSuma(m)[(nf, nc)][1]))
```

```
# diccionarioCaminoMaxSuma(p) es el diccionario cuyas claves son los
# puntos de la matriz p y sus valores son los pares formados por la
# máxima suma de los caminos hasta dicho punto y uno de los caminos con
# esa suma. Por ejemplo,
#    >>> diccionarioCaminoMaxSuma([[1,6,11,2],[7,12,3,8],[3,8,4,9]])
#    {(1, 1): (1, [1]),
#     (1, 2): (7, [6, 1]),
#     (1, 3): (18, [11, 6, 1]),
#     (1, 4): (20, [2, 11, 6, 1]),
#     (2, 1): (8, [7, 1]),
#     (3, 1): (11, [3, 7, 1]),
#     (2, 2): (20, [12, 7, 1]),
#     (2, 3): (23, [3, 12, 7, 1]),
#     (2, 4): (31, [8, 3, 12, 7, 1]),
#     (3, 2): (28, [8, 12, 7, 1]),
#     (3, 3): (32, [4, 8, 12, 7, 1]),
#     (3, 4): (41, [9, 4, 8, 12, 7, 1])}
def diccionarioCaminoMaxSuma(p: list[list[int]]) -> dict[tuple[int, int], tuple[i
    m = len(p)
    n = len(p[0])
    q: dict[tuple[int, int], tuple[int, list[int]]] = {}
    q[(1, 1)] = (p[0][0], [p[0][0]])
    for j in range(2, n + 1):
        (k, xs) = q[(1, j-1)]
        q[(1, j)] = (k + p[0][j-1], [p[0][j-1]] + xs)
    for i in range(2, m + 1):
        (k,xs) = q[(i-1,1)]
        q[(i, 1)] =  (k + p[i-1][0], [p[i-1][0]] + xs)
    for i in range(2, m + 1):
        for j in range(2, n + 1):
            (k1,xs) = q[(i,j-1)]
            (k2,ys) = q[(i-1,j)]
            if k1 > k2:
                q[(i,j)] = (k1 + p[i-1][j-1], [p[i-1][j-1]] + xs)
            else:
                q[(i,j)] = (k2 + p[i-1][j-1], [p[i-1][j-1]] + ys)
    return q

# 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('caminoMaxSuma1([list(range(11*n+1, 11*(n+1)+1)) for n in range(1
#    1.92 segundos
#    >>> tiempo('caminoMaxSuma2([list(range(11*n+1, 11*(n+1)+1)) for n in range(1
#    0.65 segundos
#    >>> tiempo('caminoMaxSuma3([list(range(11*n+1, 11*(n+1)+1)) for n in range(1
#    0.00 segundos

# Verificación
# ============

def test_caminoMaxSuma() -> None:
    assert caminoMaxSuma1([[1,6,11,2],[7,12,3,8],[3,8,4,9]]) == \
        [1, 7, 12, 8, 4, 9]
    assert caminoMaxSuma2([[1,6,11,2],[7,12,3,8],[3,8,4,9]]) == \
        [1, 7, 12, 8, 4, 9]
    assert caminoMaxSuma3([[1,6,11,2],[7,12,3,8],[3,8,4,9]]) == \
        [1, 7, 12, 8, 4, 9]
    print("Verificado")

# La verificación es
#    >>> test_caminoMaxSuma()
#    Verificado
```

Parte III

Aplicaciones a las matemáticas

Capítulo 15

Cálculo numérico

Contenido

15.1. Método de Herón para calcular la raíz cuadrada

15.1.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- El método de Herón para calcular la raíz cuadrada de un número se
-- basa en las siguientes propiedades:
-- + Si y es una aproximación de la raíz cuadrada de x, entonces
--   (y+x/y)/2 es una aproximación mejor.
-- + El límite de la sucesión definida por
--       x_0     = 1
--       x_{n+1} = (x_n+x/x_n)/2
--   es la raíz cuadrada de x.
--
-- Definir la función
--    raiz :: Double -> Double
-- tal que (raiz x) es la raíz cuadrada de x calculada usando la
-- propiedad anterior con una aproximación de 0.00001 y tomando como
-- valor inicial 1. Por ejemplo,
--    raiz 9  ==  3.000000001396984
-- ---------------------------------------------------------------------

module Metodo_de_Heron_para_calcular_la_raiz_cuadrada where

import Test.QuickCheck
import Test.Hspec (Spec, hspec, it, shouldBe)

-- 1ª solución
-- ===========

raiz :: Double -> Double
raiz x = raizAux 1
```

```haskell
  where raizAux y | aceptable y = y
                  | otherwise   = raizAux (mejora y)
        aceptable y = abs(y*y-x) < 0.00001
        mejora y    = 0.5*(y+x/y)

-- 2ª solución
-- ===========

raiz2 :: Double -> Double
raiz2 x = until aceptable mejora 1
  where aceptable y = abs(y*y-x) < 0.00001
        mejora y    = 0.5*(y+x/y)

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

-- La propiedad es
prop_raiz :: Positive Double -> Bool
prop_raiz (Positive x) =
  raiz x ~= sqrt x &&
  raiz2 x ~= sqrt x
  where
    a ~= b = abs (a-b) < 0.001

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

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

spec :: Spec
spec = do
  it "e1" $
    raiz 9 `shouldBe`  3.000000001396984
  it "e2" $
    raiz2 9 `shouldBe`  3.000000001396984
```

```
-- La verificación es
--    λ> verifica
--
--    e1
--    e2
--
--    Finished in 0.0008 seconds
--    2 examples, 0 failures
```

15.1.2. En Python

```
# ---------------------------------------------------------------------
# El método de Herón para calcular la raíz cuadrada de un número se
# basa en las siguientes propiedades:
# + Si y es una aproximación de la raíz cuadrada de x, entonces
#   (y+x/y)/2 es una aproximación mejor.
# + El límite de la sucesión definida por
#       x_0     = 1
#       x_{n+1} = (x_n+x/x_n)/2
#   es la raíz cuadrada de x.
#
# Definir la función
#    raiz : (float) -> float
# tal que raiz(x) es la raíz cuadrada de x calculada usando la
# propiedad anterior con una aproximación de 0.00001 y tomando como
# valor inicial 1. Por ejemplo,
#    raiz(9)  ==  3.000000001396984
# ---------------------------------------------------------------------

# 1ª solución
# ===========

def raiz(x : float) -> float:
    def aceptable(y: float) -> bool:
        return abs(y*y-x) < 0.00001
    def mejora(y: float) -> float:
        return 0.5*(y+x/y)
    def raizAux(y: float) -> float:
        if aceptable(y):
```

```
            return y
        return raizAux(mejora(y))
    return raizAux(1)

# 2ª solución
# ===========

def raiz2(x: float) -> float:
    def aceptable(y: float) -> bool:
        return abs(y*y-x) < 0.00001
    def mejora(y: float) -> float:
        return 0.5*(y+x/y)
    y = 1.0
    while not aceptable(y):
        y = mejora(y)
    return y

# Verificación
# ============

def test_raiz() -> None:
    assert raiz(9)  ==  3.000000001396984
    assert raiz2(9)  ==  3.000000001396984
    print("Verificado")

# La verificación es
#    >>> test_raiz()
#    Verificado
```

15.2. Método de Newton para calcular raíces

15.2.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Los ceros de una función pueden calcularse mediante el método de
-- Newton basándose en las siguientes propiedades:
-- + Si b es una aproximación para el punto cero de f, entonces
--   b-f(b)/f'(b) es una mejor aproximación.
-- + el límite de la sucesión x_n definida por
--      x_0     = 1
```

```
--      x_{n+1} = x_n-f(x_n)/f'(x_n)
--   es un cero de f.
--
-- Definir la función
--    puntoCero :: (Double -> Double) -> Double
-- tal que (puntoCero f) es un cero de la función f calculado usando la
-- propiedad anterior. Por ejemplo,
--    puntoCero cos  ==  1.5707963267949576
-- ---------------------------------------------------------------------

module Metodo_de_Newton_para_calcular_raices where

import Test.Hspec (Spec, hspec, it, shouldBe)

-- 1ª solución
-- ===========

puntoCero :: (Double -> Double) -> Double
puntoCero f = puntoCeroAux f 1
  where puntoCeroAux f' x | aceptable x = x
                          | otherwise   = puntoCeroAux f' (mejora x)
        aceptable b = abs (f b) < 0.00001
        mejora b    = b - f b / derivada f b

-- (derivada f x) es el valor de la derivada de la función f en el punto
-- x con aproximación 0.0001. Por ejemplo,
--    derivada sin pi == -0.9999999983354435
--    derivada cos pi == 4.999999969612645e-5
derivada :: (Double -> Double) -> Double -> Double
derivada f x = (f (x+a) - f x)/a
  where a = 0.0001

-- 2ª solución
-- ===========

puntoCero2 :: (Double -> Double) -> Double
puntoCero2 f = until aceptable mejora 1
  where aceptable b = abs (f b) < 0.00001
        mejora b    = b - f b / derivada f b
```

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

verifica :: IO ()
verifica = hspec spec

spec :: Spec
spec = do
  it "e1" $
    puntoCero cos `shouldBe` 1.5707963267949576
  it "e2" $
    puntoCero2 cos `shouldBe` 1.5707963267949576

-- La verificación es
--    λ> verifica
--
--    e1
--    e2
--
--    Finished in 0.0002 seconds
--    2 examples, 0 failures
```

15.2.2. En Python

```python
# ---------------------------------------------------------------------
# Los ceros de una función pueden calcularse mediante el método de
# Newton basándose en las siguientes propiedades:
# + Si b es una aproximación para el punto cero de f, entonces
#   b-f(b)/f'(b) es una mejor aproximación.
# + el límite de la sucesión x_n definida por
#      x_0     = 1
#      x_{n+1} = x_n-f(x_n)/f'(x_n)
#   es un cero de f.
#
# Definir la función
#    puntoCero : (Callable[[float], float]) -> float
# tal que puntoCero(f) es un cero de la función f calculado usando la
# propiedad anterior. Por ejemplo,
#    puntoCero(cos)  ==  1.5707963267949576
# ---------------------------------------------------------------------
```

```python
from math import cos, pi, sin
from typing import Callable

# 1ª solución
# ===========

# derivada(f, x) es el valor de la derivada de la función f en el punto
# x con aproximación 0.0001. Por ejemplo,
#    derivada(sin, pi) == -0.9999999983354435
#    derivada(cos, pi) == 4.999999969612645e-5
def derivada(f: Callable[[float], float], x: float) -> float:
    a = 0.0001
    return (f(x+a) - f(x)) / a

def puntoCero(f: Callable[[float], float]) -> float:
    def aceptable(b: float) -> bool:
        return abs(f(b)) < 0.00001
    def mejora(b: float) -> float:
        return b - f(b) / derivada(f, b)
    def aux(g: Callable[[float], float], x: float) -> float:
        if aceptable(x):
            return x
        return aux(g, mejora(x))
    return aux(f, 1)

# 2ª solución
# ===========

def puntoCero2(f: Callable[[float], float]) -> float:
    def aceptable(b: float) -> bool:
        return abs(f(b)) < 0.00001
    def mejora(b: float) -> float:
        return b - f(b) / derivada(f, b)
    y = 1.0
    while not aceptable(y):
        y = mejora(y)
    return y

# Verificación
```

```
# ============

def test_puntoCero () -> None:
    assert puntoCero(cos) == 1.5707963267949576
    assert puntoCero(cos) - pi/2 == 6.106226635438361e-14
    assert puntoCero(sin) == -5.8094940533562345e-15
    assert puntoCero2(cos) == 1.5707963267949576
    assert puntoCero2(cos) - pi/2 == 6.106226635438361e-14
    assert puntoCero2(sin) == -5.8094940533562345e-15
    print("Verificado")

# La comprobación es
#    >>> test_puntoCero()
#    Verificado
```

15.3. Funciones inversas por el método de Newton

15.3.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Definir, usando puntoCero, la función
--    inversa :: (Double -> Double) -> Double -> Double
-- tal que (inversa g x) es el valor de la inversa de g en x. Por
-- ejemplo,
--    inversa (^2) 9  ==  3.000000002941184
--
-- Definir, usando inversa, las funciones raizCuadrada, raizCubica,
-- arcoseno y arcocoseno que calculen la raíz cuadrada, la raíz cúbica,
-- el arco seno y el arco coseno, respectivamente. Por ejemplo,
--    raizCuadrada 9  ==  3.000000002941184
--    raizCubica 27   ==  3.0000000000196048
--    arcoseno 1      ==  1.5665489428306574
--    arcocoseno 0    ==  1.5707963267949576
-- ---------------------------------------------------------------------

module Funciones_inversas_por_el_metodo_de_Newton where

import Metodo_de_Newton_para_calcular_raices (puntoCero)
```

```haskell
import Test.Hspec (Spec, hspec, it, shouldBe)

inversa :: (Double -> Double) -> Double -> Double
inversa g a = puntoCero f
  where f x = g x - a

raizCuadrada, raizCubica, arcoseno, arcocoseno :: Double -> Double
raizCuadrada = inversa (^2)
raizCubica   = inversa (^3)
arcoseno     = inversa sin
arcocoseno   = inversa cos

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

spec :: Spec
spec = do
  it "e1" $
    inversa (^2) 9  `shouldBe`  3.000000002941184
  it "e2" $
    raizCuadrada 9  `shouldBe`  3.000000002941184
  it "e3" $
    raizCubica 27   `shouldBe`  3.0000000000196048
  it "e4" $
    arcoseno 1      `shouldBe`  1.5665489428306574
  it "e5" $
    arcocoseno 0    `shouldBe`  1.5707963267949576

-- La verificación es
--    λ> verifica
--
--    e1
--    e2
--    e3
--    e4
--    e5
--
```

```
--    Finished in 0.0006 seconds
--    5 examples, 0 failures
```

15.3.2. En Python

```python
# ---------------------------------------------------------------------
# Definir, usando puntoCero, la función
#    inversa : (Callable[[float],float], float) -> float
# tal que inversa(g, x) es el valor de la inversa de g en x. Por
# ejemplo,
#    inversa(lambda x: x**2, 9)  ==  3.000000002941184
#
# Definir, usando inversa, las funciones raizCuadrada, raizCubica,
# arcoseno y arcocoseno que calculen la raíz cuadrada, la raíz cúbica,
# el arco seno y el arco coseno, respectivamente. Por ejemplo,
#    raizCuadrada(9)  ==  3.000000002941184
#    raizCubica(27)   ==  3.0000000000196048
#    arcoseno(1)      ==  1.5665489428306574
#    arcocoseno(0)    ==  1.5707963267949576
# ---------------------------------------------------------------------

from math import cos, sin
from typing import Callable

from src.Metodo_de_Newton_para_calcular_raices import puntoCero


def inversa(g: Callable[[float],float], a: float) -> float:
    def f(x: float) -> float:
        return g(x) - a
    return puntoCero(f)

def raizCuadrada(x: float) -> float:
    return inversa(lambda y: y**2, x)

def raizCubica(x: float) -> float:
    return inversa(lambda y: y**3, x)

def arcoseno(x: float) -> float:
    return inversa(sin, x)
```

```python
def arcocoseno(x: float) -> float:
    return inversa(cos, x)

# Verificación
# ============

def test_inversa() -> None:
    assert inversa(lambda x: x**2, 9) == 3.000000002941184
    assert raizCuadrada(9) == 3.000000002941184
    assert raizCubica(27) == 3.0000000000196048
    assert arcoseno(1) == 1.5665489428306574
    assert arcocoseno(0) == 1.5707963267949576
    print("Verificado")

# La comprobación es
#    >>> test_inversa()
#    Verificado
```

15.4. Límites de sucesiones

15.4.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- Definir la función
--    limite :: (Double -> Double) -> Double -> Double
-- tal que (limite f a) es el valor de f en el primer término x tal que,
-- para todo y entre x+1 y x+100, el valor absoluto de la diferencia
-- entre f(y) y f(x) es menor que a. Por ejemplo,
--    limite (\n -> (2*n+1)/(n+5)) 0.001  ==  1.9900110987791344
--    limite (\n -> (1+1/n)**n) 0.001     ==  2.714072874546881
-- ---------------------------------------------------------------------

module Limites_de_sucesiones where

import Test.Hspec (Spec, hspec, it, shouldBe)

limite :: (Double -> Double) -> Double -> Double
limite f a =
  head [f x | x <- [1..],
```

```
              maximum [abs (f y - f x) | y <- [x+1..x+100]] < a]

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

spec :: Spec
spec = do
  it "e1" $
    limite (\n -> (2*n+1)/(n+5)) 0.001  `shouldBe`  1.9900110987791344
  it "e2" $
    limite (\n -> (1+1/n)**n) 0.001     `shouldBe`  2.714072874546881

-- La verificación es
--    λ> verifica
--
--    e1
--    e2
--
--    Finished in 0.1927 seconds
--    2 examples, 0 failures
```

15.4.2. En Python

```
# ---------------------------------------------------------------------
# Definir la función
#    limite : (Callable[[float], float], float) -> float
# tal que limite(f, a) es el valor de f en el primer término x tal que,
# para todo y entre x+1 y x+100, el valor absoluto de la diferencia
# entre f(y) y f(x) es menor que a. Por ejemplo,
#    limite(lambda n :  (2*n+1)/(n+5), 0.001) ==  1.9900110987791344
#    limite(lambda n : (1+1/n)**n, 0.001)     ==  2.714072874546881
# ---------------------------------------------------------------------

from itertools import count
from typing import Callable

# 1ª solución
```

```
# ===========

def limite(f: Callable[[float], float], a: float) -> float:
    x = 1
    while True:
        maximum_diff = max(abs(f(y) - f(x)) for y in range(x+1, x+101))
        if maximum_diff < a:
            return f(x)
        x += 1

# 2ª solución
# ===========

def limite2(f: Callable[[float], float], a: float) -> float:
    x = 1
    while True:
        y = f(x)
        if max(abs(y - f(x + i)) for i in range(1, 101)) < a:
            break
        x += 1
    return y

# 3ª solución
# ===========

def limite3(f: Callable[[float], float], a: float) -> float:
    for x in count(1):
        if max(abs(f(y) - f(x)) for y in range(x + 1, x + 101)) < a:
            r = f(x)
            break
    return r

# Verificación
# ============

def test_limite() -> None:
    assert limite(lambda n :  (2*n+1)/(n+5), 0.001) ==  1.9900110987791344
    assert limite(lambda n : (1+1/n)**n, 0.001)     ==  2.714072874546881
    assert limite2(lambda n :  (2*n+1)/(n+5), 0.001) ==  1.9900110987791344
    assert limite2(lambda n : (1+1/n)**n, 0.001)     ==  2.714072874546881
```

```
    assert limite3(lambda n :  (2*n+1)/(n+5), 0.001) ==  1.9900110987791344
    assert limite3(lambda n : (1+1/n)**n, 0.001)     ==  2.714072874546881
    print("Verificado")

# La comprobación es
#    >>> test_limite()
#    Verificado
```

15.5. Método de bisección para calcular ceros de una función

15.5.1. En Haskell

```
-- ---------------------------------------------------------------------
-- El método de bisección para calcular un cero de una función en el
-- intervalo [a,b] se basa en el teorema de Bolzano:
--    "Si f(x) es una función continua en el intervalo [a, b], y si,
--    además, en los extremos del intervalo la función f(x) toma valores
--    de signo opuesto (f(a) * f(b) < 0), entonces existe al menos un
--    valor c en (a, b) para el que f(c) = 0".
--
-- El método para calcular un cero de la función f en el intervalo [a,b]
-- con un error menor que e consiste en tomar el punto medio del
-- intervalo c = (a+b)/2 y considerar los siguientes casos:
-- + Si |f(c)| < e, hemos encontrado una aproximación del punto que
--   anula f en el intervalo con un error aceptable.
-- + Si f(c) tiene signo distinto de f(a), repetir el proceso en el
--   intervalo [a,c].
-- + Si no, repetir el proceso en el intervalo [c,b].
--
-- Definir la función
--    biseccion :: (Double -> Double) -> Double -> Double -> Double -> Double
-- tal que (biseccion f a b e) es una aproximación del punto del
-- intervalo [a,b] en el que se anula la función f, con un error menor
-- que e, calculada mediante el método de la bisección. Por ejemplo,
--    biseccion (\x -> x^2 - 3) 0 5 0.01             ==  1.7333984375
--    biseccion (\x -> x^3 - x - 2) 0 4 0.01         ==  1.521484375
--    biseccion cos 0 2 0.01                         ==  1.5625
--    biseccion (\x -> log (50-x) - 4) (-10) 3 0.01  ==  -5.125
```

```haskell
-- ---------------------------------------------------------------------

module Metodo_de_biseccion_para_calcular_ceros_de_una_funcion where

import Test.Hspec (Spec, hspec, it, shouldBe)

-- 1ª solución
-- ===========

biseccion :: (Double -> Double) -> Double -> Double -> Double -> Double
biseccion f a b e
  | abs (f c) < e = c
  | f a * f c < 0 = biseccion f a c e
  | otherwise     = biseccion f c b e
  where c = (a+b)/2

-- 2ª solución
-- ===========

biseccion2 :: (Double -> Double) -> Double -> Double -> Double -> Double
biseccion2 f a b e = aux a b
  where aux a' b' | abs (f c) < e   = c
                  | f a' * f c < 0  = aux a' c
                  | otherwise       = aux c b'
          where c = (a'+b')/2

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

spec :: Spec
spec = do
  it "e1" $
    biseccion (\x -> x^2 - 3) 0 5 0.01          `shouldBe`  1.7333984375
  it "e2" $
    biseccion (\x -> x^3 - x - 2) 0 4 0.01      `shouldBe`  1.521484375
  it "e3" $
    biseccion cos 0 2 0.01                      `shouldBe`  1.5625
```

```
  it "e4" $
    biseccion (\x -> log (50-x) - 4) (-10) 3 0.01  `shouldBe`  -5.125
  it "e5" $
    biseccion2 (\x -> x^2 - 3) 0 5 0.01            `shouldBe`  1.7333984375
  it "e6" $
    biseccion2 (\x -> x^3 - x - 2) 0 4 0.01        `shouldBe`  1.521484375
  it "e7" $
    biseccion2 cos 0 2 0.01                        `shouldBe`  1.5625
  it "e8" $
    biseccion2 (\x -> log (50-x) - 4) (-10) 3 0.01  `shouldBe`  -5.125

-- La verificación es
--    λ> verifica
--
--    e1
--    e2
--    e3
--    e4
--    e5
--    e6
--    e7
--    e8
--
--    Finished in 0.0008 seconds
--    8 examples, 0 failures
```

15.5.2. En Python

```
# ---------------------------------------------------------------------
# El método de bisección para calcular un cero de una función en el
# intervalo [a,b] se basa en el teorema de Bolzano:
#    "Si f(x) es una función continua en el intervalo [a, b], y si,
#    además, en los extremos del intervalo la función f(x) toma valores
#    de signo opuesto (f(a) * f(b) < 0), entonces existe al menos un
#    valor c en (a, b) para el que f(c) = 0".
#
# El método para calcular un cero de la función f en el intervalo [a,b]
# con un error menor que e consiste en tomar el punto medio del
# intervalo c = (a+b)/2 y considerar los siguientes casos:
# + Si |f(c)| < e, hemos encontrado una aproximación del punto que
```

```
#   anula f en el intervalo con un error aceptable.
# + Si f(c) tiene signo distinto de f(a), repetir el proceso en el
#   intervalo [a,c].
# + Si no, repetir el proceso en el intervalo [c,b].
#
# Definir la función
#    biseccion : (Callable[[float], float], float, float, float) -> float
# tal que biseccion(f, a, b, e) es una aproximación del punto del
# intervalo [a,b] en el que se anula la función f, con un error menor
# que e, calculada mediante el método de la bisección. Por ejemplo,
#    biseccion(lambda x : x**2 - 3, 0, 5, 0.01)          == 1.7333984375
#    biseccion(lambda x : x**3 - x - 2, 0, 4, 0.01)      == 1.521484375
#    biseccion(cos, 0, 2, 0.01)                          == 1.5625
#    biseccion(lambda x : log(50-x) - 4, -10, 3, 0.01) == -5.125
# ---------------------------------------------------------------------

from math import cos, log
from typing import Callable

# 1ª solución
# ===========

def biseccion(f: Callable[[float], float],
              a: float,
              b: float,
              e: float) -> float:
    c = (a+b)/2
    if abs(f(c)) < e:
        return c
    if f(a) * f(c) < 0:
        return biseccion(f, a, c, e)
    return biseccion(f, c, b, e)

# 2ª solución
# ===========

def biseccion2(f: Callable[[float], float],
               a: float,
               b: float,
               e: float) -> float:
```

```
    def aux(a1: float,  b1: float) -> float:
        c = (a1+b1)/2
        if abs(f(c)) < e:
            return c
        if f(a1) * f(c) < 0:
            return aux(a1, c)
        return aux(c, b1)
    return aux(a, b)

# Verificación
# ============

def test_biseccion() -> None:
    assert biseccion(lambda x : x**2 - 3, 0, 5, 0.01) == 1.7333984375
    assert biseccion(lambda x : x**3 - x - 2, 0, 4, 0.01) == 1.521484375
    assert biseccion(cos, 0, 2, 0.01) == 1.5625
    assert biseccion(lambda x : log(50-x) - 4, -10, 3, 0.01) == -5.125
    assert biseccion2(lambda x : x**2 - 3, 0, 5, 0.01) == 1.7333984375
    assert biseccion2(lambda x : x**3 - x - 2, 0, 4, 0.01) == 1.521484375
    assert biseccion2(cos, 0, 2, 0.01) == 1.5625
    assert biseccion2(lambda x : log(50-x) - 4, -10, 3, 0.01) == -5.125
    print("Verificado")

# La comprobación es
#    >>> test_biseccion()
#    Verificado
```

15.6. Raíces enteras

15.6.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Definir la función
--    raizEnt :: Integer -> Integer -> Integer
-- tal que (raizEnt x n) es la raíz entera n-ésima de x; es decir, el
-- mayor número entero y tal que y^n <= x. Por ejemplo,
--    raizEnt  8 3      ==  2
--    raizEnt  9 3      ==  2
--    raizEnt 26 3      ==  2
--    raizEnt 27 3      ==  3
```

```haskell
--    raizEnt (10^50) 2 ==  10000000000000000000000000
--
-- Comprobar con QuickCheck que para todo número natural n,
--     raizEnt (10^(2*n)) 2 == 10^n
-- ---------------------------------------------------------------------

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

module Raices_enteras where

import Test.Hspec (Spec, hspec, it, shouldBe)
import Test.QuickCheck (quickCheck)

-- 1ª solución
-- ===========

raizEnt1 :: Integer -> Integer -> Integer
raizEnt1 x n =
  last (takeWhile (\y -> y^n <= x) [0..])

-- 2ª solución
-- ===========

raizEnt2 :: Integer -> Integer -> Integer
raizEnt2 x n =
  floor ((fromIntegral x)**(1 / fromIntegral n))

-- Nota. La solución anterior falla para números grandes. Por ejemplo,
--    λ> raizEnt2 (10^50) 2 == 10^25
--    False

-- 3ª solución
-- ===========

raizEnt3 :: Integer -> Integer -> Integer
raizEnt3 x n = aux (1,x)
  where aux (a,b) | d == x    = c
                  | c == a    = c
                  | d < x     = aux (c,b)
                  | otherwise = aux (a,c)
```

```haskell
          where c = (a+b) `div` 2
                d = c^n

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

--    λ> raizEnt1 (10^14) 2
--    10000000
--    (6.15 secs, 6,539,367,976 bytes)
--    λ> raizEnt2 (10^14) 2
--    10000000
--    (0.00 secs, 0 bytes)
--    λ> raizEnt3 (10^14) 2
--    10000000
--    (0.00 secs, 25,871,944 bytes)
--
--    λ> raizEnt2 (10^50) 2
--    9999999999999998758486016
--    (0.00 secs, 0 bytes)
--    λ> raizEnt3 (10^50) 2
--    10000000000000000000000000
--    (0.00 secs, 0 bytes)

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

-- La propiedad es
prop_raizEnt :: Integer -> Bool
prop_raizEnt n =
  raizEnt3 (10^(2*m)) 2 == 10^m
  where m = abs n

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

-- Verificación
-- ============

verifica :: IO ()
```

```
verifica = hspec spec

spec :: Spec
spec = do
  it "e1" $
    raizEnt1  8 3 `shouldBe` 2
  it "e2" $
    raizEnt1  9 3 `shouldBe` 2
  it "e3" $
    raizEnt1 26 3 `shouldBe` 2
  it "e4" $
    raizEnt1 27 3 `shouldBe` 3
  it "e5" $
    raizEnt2  8 3 `shouldBe` 2
  it "e6" $
    raizEnt2  9 3 `shouldBe` 2
  it "e7" $
    raizEnt2 26 3 `shouldBe` 2
  it "e8" $
    raizEnt2 27 3 `shouldBe` 3
  it "e9" $
    raizEnt3  8 3 `shouldBe` 2
  it "e10" $
    raizEnt3  9 3 `shouldBe` 2
  it "e11" $
    raizEnt3 26 3 `shouldBe` 2
  it "e12" $
    raizEnt3 27 3 `shouldBe` 3

-- La verificación es
--    λ> verifica
--
--    e1
--    e2
--    e3
--    e4
--    e5
--    e6
--    e7
--    e8
```

```
--     e9
--     e10
--     e11
--     e12
--
--     Finished in 0.0007 seconds
--     12 examples, 0 failures
```

15.6.2. En Python

```python
# ---------------------------------------------------------------------
# Definir la función
#    raizEnt : (int, int) -> int
# tal que raizEnt(x, n) es la raíz entera n-ésima de x; es decir, el
# mayor número entero y tal que y^n <= x. Por ejemplo,
#    raizEnt(8, 3)      ==  2
#    raizEnt(9, 3)      ==  2
#    raizEnt(26, 3)     ==  2
#    raizEnt(27, 3)     ==  3
#    raizEnt(10**50, 2) ==  10000000000000000000000000
#
# Comprobar con Hypothesis que para todo número natural n,
#     raizEnt(10**(2*n), 2) == 10**n
# ---------------------------------------------------------------------

from itertools import count, takewhile
from math import floor
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 raizEnt(x: int, n: int) -> int:
    return list(takewhile(lambda y : y ** n <= x, count(0)))[-1]
```

```python
# 2ª solución
# ===========

def raizEnt2(x: int, n: int) -> int:
    return floor(x ** (1 / n))

# Nota. La solución anterior falla para números grandes. Por ejemplo,
#    >>> raizEnt2(10**50, 2) == 10 **25
#    False

# 3ª solución
# ===========

def raizEnt3(x: int, n: int) -> int:
    def aux(a: int, b: int) -> int:
        c = (a + b) // 2
        d = c ** n
        if d == x:
            return c
        if c == a:
            return c
        if d < x:
            return aux(c, b)
        return aux(a, c)
    return aux(1, 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('raizEnt(10**14, 2)')
#    2.71 segundos
#    >>> tiempo('raizEnt2(10**14, 2)')
#    0.00 segundos
```

```python
#    >>> tiempo('raizEnt3(10**14, 2)')
#    0.00 segundos
#
#    >>> raizEnt2(10**50, 2)
#    10000000000000000905969664
#    >>> raizEnt3(10**50, 2)
#    10000000000000000000000000

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

# La propiedad es
@given(st.integers(min_value=0, max_value=1000))
def test_raizEntP(n: int) -> None:
    assert raizEnt3(10**(2*n), 2) == 10**n

# La comprobación es
#    >>> test_raizEntP()
#    >>>

# Verificación
# ============

def test_raizEnt() -> None:
    assert raizEnt(8, 3) == 2
    assert raizEnt(9, 3) == 2
    assert raizEnt(26, 3) == 2
    assert raizEnt(27, 3) == 3
    assert raizEnt2(8, 3) == 2
    assert raizEnt2(9, 3) == 2
    assert raizEnt2(26, 3) == 2
    assert raizEnt2(27, 3) == 3
    assert raizEnt3(8, 3) == 2
    assert raizEnt3(9, 3) == 2
    assert raizEnt3(26, 3) == 2
    assert raizEnt3(27, 3) == 3
    print("Verificado")

# La comprobación es
#    >>> test_raizEnt()
```

```
#    Verificado
```

15.7. Integración por el método de los rectángulos

15.7.1. En Haskell

```haskell
-- ---------------------------------------------------------------------
-- La integral definida de una función f entre los límites a y b puede
-- calcularse mediante la regla del rectángulo (ver en
-- http://bit.ly/1FDhZ1z) usando la fórmula
--    h * (f(a+h/2) + f(a+h+h/2) + f(a+2h+h/2) + ... + f(a+n*h+h/2))
-- con a+n*h+h/2 <= b < a+(n+1)*h+h/2 y usando valores pequeños para h.
--
-- Definir la función
--    integral :: (Fractional a, Ord a) => a -> a -> (a -> a) -> a -> a
-- tal que (integral a b f h) es el valor de dicha expresión. Por
-- ejemplo, el cálculo de la integral de f(x) = x^3 entre 0 y 1, con
-- paso 0.01, es
--    integral 0 1 (^3) 0.01  ==  0.24998750000000042
-- Otros ejemplos son
--    integral 0 1 (^4) 0.01                        ==  0.19998333362500048
--    integral 0 1 (\x -> 3*x^2 + 4*x^3) 0.01       ==  1.9999250000000026
--    log 2 - integral 1 2 (\x -> 1/x) 0.01         ==  3.124931644782336e-6
--    pi - 4 * integral 0 1 (\x -> 1/(x^2+1)) 0.01  ==  -8.333333331389525e-6
-- ---------------------------------------------------------------------

module Integracion_por_rectangulos where

import Test.Hspec (Spec, hspec, it, shouldBe, shouldSatisfy)

-- 1ª solución
-- ===========

integral :: (Fractional a, Ord a) => a -> a -> (a -> a) -> a -> a
integral a b f h = h * suma (a+h/2) b (+h) f

-- (suma a b s f) es l valor de
--    f(a) + f(s(a)) + f(s(s(a)) + ... + f(s(...(s(a))...))
```

```
-- hasta que s(s(...(s(a))...)) > b. Por ejemplo,
--    suma 2 5 (1+) (^3)  ==  224
suma :: (Ord t, Num a) => t -> t -> (t -> t) -> (t -> a) -> a
suma a b s f = sum [f x | x <- sucesion a b s]

-- (sucesion x y s) es la lista
--    [a, s(a), s(s(a), ..., s(...(s(a))...)]
-- hasta que s(s(...(s(a))...)) > b. Por ejemplo,
--    sucesion 3 20 (+2)  ==  [3,5,7,9,11,13,15,17,19]
sucesion :: Ord a => a -> a -> (a -> a) -> [a]
sucesion a b s = takeWhile (<=b) (iterate s a)

-- 2ª solución
-- ===========

integral2 :: (Fractional a, Ord a) => a -> a -> (a -> a) -> a -> a
integral2 a b f h
  | a+h/2 > b = 0
  | otherwise = h * f (a+h/2) + integral2 (a+h) b f h

-- 3ª solución
-- ===========

integral3 :: (Fractional a, Ord a) => a -> a -> (a -> a) -> a -> a
integral3 a b f h = aux a where
  aux x | x+h/2 > b = 0
        | otherwise = h * f (x+h/2) + aux (x+h)

-- Comparación de eficiencia
--    λ> integral 0 10 (^3) 0.00001
--    2499.9999998811422
--    (4.62 secs, 1084774336 bytes)
--    λ> integral2 0 10 (^3) 0.00001
--    2499.999999881125
--    (7.90 secs, 1833360768 bytes)
--    λ> integral3 0 10 (^3) 0.00001
--    2499.999999881125
--    (7.27 secs, 1686056080 bytes)

-- Verificación
```

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

verifica :: IO ()
verifica = hspec spec

spec :: Spec
spec = do
  it "e1" $
    integral' 0 1 (^(3::Int)) 0.01 `shouldBe` 0.24998750000000042
  it "e2" $
    integral' 0 1 (^(4::Int)) 0.01 `shouldBe` 0.19998333362500048
  it "e3" $
    integral' 0 1 (\x -> 3*x^(2::Int) + 4*x^(3::Int)) 0.01 `shouldBe` 1.999925000
  it "e4" $
    log 2 - integral' 1 2 (1 /) 0.01 `shouldBe` 3.124931644782336e-6
  it "e5" $
    pi - 4 * integral' 0 1 (\x -> 1/(x^(2::Int)+1)) 0.01 `shouldBe` -8.333333313
  it "e1b" $
    integral2' 0 1 (^(3::Int)) 0.01 `shouldSatisfy` (~= 0.24998750000000042)
  it "e2b" $
    integral2' 0 1 (^(4::Int)) 0.01 `shouldSatisfy` (~= 0.19998333362500048)
  it "e3b" $
    integral2' 0 1 (\x -> 3*x^(2::Int) + 4*x^(3::Int)) 0.01 `shouldSatisfy` (~= 1
  it "e4b" $
    log 2 - integral2' 1 2 (1 /) 0.01 `shouldSatisfy` (~= 3.124931644782336e-6)
  it "e5b" $
    pi - 4 * integral2' 0 1 (\x -> 1/(x^(2::Int)+1)) 0.01 `shouldSatisfy` (~= (-8
  it "e1c" $
    integral3' 0 1 (^(3::Int)) 0.01 `shouldSatisfy` (~= 0.24998750000000042)
  it "e2c" $
    integral3' 0 1 (^(4::Int)) 0.01 `shouldSatisfy` (~= 0.19998333362500048)
  it "e3c" $
    integral3' 0 1 (\x -> 3*x^(2::Int) + 4*x^(3::Int)) 0.01 `shouldSatisfy` (~= 1
  it "e4c" $
    log 2 - integral3' 1 2 (1 /) 0.01 `shouldSatisfy` (~= 3.124931644782336e-6)
  it "e5c" $
    pi - 4 * integral3' 0 1 (\x -> 1/(x^(2::Int)+1)) 0.01 `shouldSatisfy` (~= (-8
  where
    integral', integral2', integral3' :: Double -> Double -> (Double -> Double) -
    integral'  = integral
```

```
    integral2' = integral2
    integral3' = integral3
    a ~= b = abs (a - b) < 0.00001

-- La verificación es
--    λ> verifica
--
--    Finished in 0.0058 seconds
--    15 examples, 0 failures
```

15.7.2. En Python

```
# ---------------------------------------------------------------------
# La integral definida de una función f entre los límites a y b puede
# calcularse mediante la regla del rectángulo (ver en
# http://bit.ly/1FDhZ1z) usando la fórmula
#    h * (f(a+h/2) + f(a+h+h/2) + f(a+2h+h/2) + ... + f(a+n*h+h/2))
# con a+n*h+h/2 <= b < a+(n+1)*h+h/2 y usando valores pequeños para h.
#
# Definir la función
#    integral : (float, float, Callable[[float], float], float) -> float
# tal que integral(a, b, f, h) es el valor de dicha expresión. Por
# ejemplo, el cálculo de la integral de f(x) = x^3 entre 0 y 1, con
# paso 0.01, es
#    integral(0, 1, lambda x : x**3, 0.01) == 0.24998750000000042
# Otros ejemplos son
#    integral(0, 1, lambda x : x**4, 0.01) == 0.19998333362500054
#    integral(0, 1, lambda x : 3*x**2 + 4*x**3, 0.01) == 1.9999250000000026
#    log(2) - integral(1, 2, lambda x : 1/x, 0.01) == 3.124931644782336e-6
#    pi - 4 * integral(0, 1, lambda x : 1/(x**2+1), 0.01) == -8.333333331389525e-
# ---------------------------------------------------------------------

from math import log, pi
from typing import Callable

# 1ª solución
# ===========

# sucesion(x, y, s) es la lista
#    [a, s(a), s(s(a), ..., s(...(s(a))...)]
```

```python
# hasta que s(s(...(s(a))...)) > b. Por ejemplo,
#    sucesion(3, 20, lambda x : x+2)  ==  [3,5,7,9,11,13,15,17,19]
def sucesion(a: float, b: float, s: Callable[[float], float]) -> list[float]:
    xs = []
    while a <= b:
        xs.append(a)
        a = s(a)
    return xs

# suma(a, b, s, f) es el valor de
#    f(a) + f(s(a)) + f(s(s(a)) + ... + f(s(...(s(a))...))
# hasta que s(s(...(s(a))...)) > b. Por ejemplo,
#    suma(2, 5, lambda x: x+1, lambda x: x**3)  ==  224
def suma(a: float,
         b: float,
         s: Callable[[float], float],
         f: Callable[[float], float]) -> float:
    return sum(f(x) for x in sucesion(a, b, s))

def integral(a: float,
             b: float,
             f: Callable[[float], float],
             h: float) -> float:
    return h * suma(a+h/2, b, lambda x: x+h, f)

# 2ª solución
# ===========

def integral2(a: float,
              b: float,
              f: Callable[[float], float],
              h: float) -> float:
    if a+h/2 > b:
        return 0
    return h * f(a+h/2) + integral2(a+h, b, f, h)

# 3ª solución
# ===========

def integral3(a: float,
```

```
              b: float,
              f: Callable[[float], float],
              h: float) -> float:
    def aux(x: float) -> float:
        if x+h/2 > b:
            return 0
        return h * f(x+h/2) + aux(x+h)
    return aux(a)

# Verificación
# ============

def test_integral() -> None:
    def aproximado(a: float, b: float) -> bool:
        return abs(a - b) < 0.00001
    assert integral(0, 1, lambda x : x**3, 0.01) == 0.24998750000000042
    assert integral(0, 1, lambda x : x**4, 0.01) == 0.19998333362500054
    assert integral(0, 1, lambda x : 3*x**2 + 4*x**3, 0.01) == 1.9999250000000026
    assert log(2) - integral(1, 2, lambda x : 1/x, 0.01) == 3.124931644782336e-6
    assert pi - 4 * integral(0, 1, lambda x : 1/(x**2+1), 0.01) == -8.33333333138
    assert aproximado(integral2(0, 1, lambda x : x**3, 0.01),
                      0.24998750000000042)
    assert aproximado(integral2(0, 1, lambda x : x**4, 0.01),
                      0.19998333362500054)
    assert aproximado(integral2(0, 1, lambda x : 3*x**2 + 4*x**3, 0.01),
                      1.9999250000000026)
    assert aproximado(log(2) - integral2(1, 2, lambda x : 1/x, 0.01),
                      3.124931644782336e-6)
    assert aproximado(pi - 4 * integral2(0, 1, lambda x : 1/(x**2+1), 0.01),
                      -8.333333331389525e-6)
    assert aproximado(integral3(0, 1, lambda x : x**3, 0.01),
                      0.24998750000000042)
    assert aproximado(integral3(0, 1, lambda x : x**4, 0.01),
                      0.19998333362500054)
    assert aproximado(integral3(0, 1, lambda x : 3*x**2 + 4*x**3, 0.01),
                      1.9999250000000026)
    assert aproximado(log(2) - integral3(1, 2, lambda x : 1/x, 0.01),
                      3.124931644782336e-6)
    assert aproximado(pi - 4 * integral3(0, 1, lambda x : 1/(x**2+1), 0.01),
                      -8.333333331389525e-6)
```

```
    print("Verificado")

# La verificación es
#    >>> test_integral()
#    Verificado
```

15.8. Algoritmo de bajada para resolver un sistema triangular inferior

15.8.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Un sistema de ecuaciones lineales Ax = b es triangular inferior si
-- todos los elementos de la matriz A que están por encima de la
-- diagonal principal son nulos; es decir, es de la forma
--    a(1,1)*x(1)                                               = b(1)
--    a(2,1)*x(1) + a(2,2)*x(2)                                 = b(2)
--    a(3,1)*x(1) + a(3,2)*x(2) + a(3,3)*x(3)                   = b(3)
--    ...
--    a(n,1)*x(1) + a(n,2)*x(2) + a(n,3)*x(3) +...+ a(x,x)*x(n) = b(n)
--
-- El sistema es compatible si, y sólo si, el producto de los elementos
-- de la diagonal principal es distinto de cero. En este caso, la
-- solución se puede calcular mediante el algoritmo de bajada:
--    x(1) = b(1) / a(1,1)
--    x(2) = (b(2) - a(2,1)*x(1)) / a(2,2)
--    x(3) = (b(3) - a(3,1)*x(1) - a(3,2)*x(2)) / a(3,3)
--    ...
--    x(n) = (b(n) - a(n,1)*x(1) - a(n,2)*x(2) -...- a(n,n-1)*x(n-1)) / a(n,n)
--
-- Definir la función
--    bajada :: Matrix Double -> Matrix Double -> Matrix Double
-- tal que (bajada a b) es la solución, mediante el algoritmo de bajada,
-- del sistema compatible triangular superior ax = b. Por ejemplo,
--    λ> let a = fromLists [[2,0,0],[3,1,0],[4,2,5.0]]
--    λ> let b = fromLists [[3],[6.5],[10]]
--    λ> bajada a b
--    ( 1.5 )
--    ( 2.0 )
```

```haskell
--     ( 0.0 )
-- Es decir, la solución del sistema
--     2x             = 3
--     3x + y         = 6.5
--     4x + 2y + 5 z = 10
-- es x=1.5, y=2 y z=0.
-- ---------------------------------------------------------------------

module Algoritmo_de_bajada where

import Data.Matrix (Matrix, (!), fromLists, nrows, toLists)
import Test.Hspec (Spec, hspec, it, shouldBe)

bajada :: Matrix Double -> Matrix Double -> Matrix Double
bajada a b = fromLists [[x i] | i <- [1..m]]
  where m   = nrows a
        x k = (b!(k,1) - sum [a!(k,j) * x j | j <- [1..k-1]]) / a!(k,k)

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

spec :: Spec
spec = do
  it "ej1" $
    toLists (bajada a b) `shouldBe` [[1.5],[2.0],[0.0]]
    where
      a = fromLists [[2,0,0],[3,1,0],[4,2,5.0]]
      b = fromLists [[3],[6.5],[10]]

-- La verificación es
--     λ> verifica
--
--     Finished in 0.0007 seconds
--     1 example, 0 failures
```

15.8.2. En Python

```
# ---------------------------------------------------------------------
# Un sistema de ecuaciones lineales Ax = b es triangular inferior si
# todos los elementos de la matriz A que están por encima de la
# diagonal principal son nulos; es decir, es de la forma
#    a(1,1)*x(1)                                               = b(1)
#    a(2,1)*x(1) + a(2,2)*x(2)                                 = b(2)
#    a(3,1)*x(1) + a(3,2)*x(2) + a(3,3)*x(3)                   = b(3)
#    ...
#    a(n,1)*x(1) + a(n,2)*x(2) + a(n,3)*x(3) +...+ a(x,x)*x(n) = b(n)
#
# El sistema es compatible si, y sólo si, el producto de los elementos
# de la diagonal principal es distinto de cero. En este caso, la
# solución se puede calcular mediante el algoritmo de bajada:
#    x(1) = b(1) / a(1,1)
#    x(2) = (b(2) - a(2,1)*x(1)) / a(2,2)
#    x(3) = (b(3) - a(3,1)*x(1) - a(3,2)*x(2)) / a(3,3)
#    ...
#    x(n) = (b(n) - a(n,1)*x(1) - a(n,2)*x(2) -...- a(n,n-1)*x(n-1)) / a(n,n)
#
# Definir la función
#    bajada : (list[list[float]], list[list[float]]) -> list[list[float]]
# tal que bajada(a, b) es la solución, mediante el algoritmo de bajada,
# del sistema compatible triangular superior ax = b. Por ejemplo,
#    >>> bajada([[2,0,0],[3,1,0],[4,2,5.0]], [[3],[6.5],[10]])
#    [[1.5], [2.0], [0.0]]
# Es decir, la solución del sistema
#    2x            = 3
#    3x + y        = 6.5
#    4x + 2y + 5 z = 10
# es x=1.5, y=2 y z=0.
# ---------------------------------------------------------------------

def bajada(a: list[list[float]], b: list[list[float]]) -> list[list[float]]:
    n = len(a)
    def x(k: int) -> float:
        return (b[k][0] - sum((a[k][j] * x(j) for j in range(0, k)))) / a[k][k]
    return [[x(i)] for i in range(0, n)]

# Verificación
```

```
# ============

def test_bajada() -> None:
    assert bajada([[2,0,0],[3,1,0],[4,2,5.0]], [[3],[6.5],[10]]) == \
        [[1.5], [2.0], [0.0]]
    print("Verificado")

# La verificación es
#    >>> test_bajada()
#    Verificado
```

Parte IV

Miscelánea

Capítulo 16

Miscelánea

Contenido

16.1. Números de Pentanacci

16.1.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Los números de Fibonacci se definen mediante las ecuaciones
--    F(0) = 0
--    F(1) = 1
```

```
--    F(n) = F(n-1) + F(n-2), si n > 1
-- Los primeros números de Fibonacci son
--    0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, ...
--
-- Una generalización de los anteriores son los números de Pentanacci
-- definidos por las siguientes ecuaciones
--    P(0) = 0
--    P(1) = 1
--    P(2) = 1
--    P(3) = 2
--    P(4) = 4
--    P(n) = P(n-1) + P(n-2) + P(n-3) + P(n-4) + P(n-5), si n > 4
-- Los primeros números de Pentanacci son
--   0, 1, 1, 2, 4, 8, 16, 31, 61, 120, 236, 464, 912, 1793, 3525, ...
--
-- Definir las funciones
--    pentanacci  :: Integer -> Integer
--    pentanaccis :: [Integer]
-- tales que
-- + (pentanacci n) es el n-ésimo número de Pentanacci. Por ejemplo,
--      λ> pentanacci 14
--      3525
--      λ> pentanacci (10^5) `mod` 10^30
--      482929150584077921552549215816
--      λ> length (show (pentanacci (10^5)))
--      29357
-- + pentanaccis es la lista de los números de Pentanacci. Por ejemplo,
--      λ> take 15 pentanacci
--      [0,1,1,2,4,8,16,31,61,120,236,464,912,1793,3525]
-- ---------------------------------------------------------------------

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

module Numeros_de_Pentanacci where

import Data.List (genericIndex, zipWith5)
import Test.Hspec (Spec, hspec, it, shouldBe)
import Test.QuickCheck (NonNegative (NonNegative), quickCheckWith, maxSize, stdAr
```

```haskell
-- 1ª solución
-- ===========

pentanacci1  :: Integer -> Integer
pentanacci1  0 = 0
pentanacci1  1 = 1
pentanacci1  2 = 1
pentanacci1  3 = 2
pentanacci1  4 = 4
pentanacci1  n = sum [pentanacci1 (n-k) | k <- [1..5]]

pentanaccis1 :: [Integer]
pentanaccis1 = [pentanacci1 n | n <- [0..]]

-- 2ª solución
-- ===========

pentanaccis2 :: [Integer]
pentanaccis2 =
  0 : 1 : 1 : 2 : 4 : zipWith5 f (r 0) (r 1) (r 2) (r 3) (r 4)
  where f a b c d e = a+b+c+d+e
        r n         = drop n pentanaccis2

pentanacci2  :: Integer -> Integer
pentanacci2 n = pentanaccis2 `genericIndex` n

-- 3ª solución
-- ===========

pentanaccis3 :: [Integer]
pentanaccis3 = p (0, 1, 1, 2, 4)
  where p (a, b, c, d, e) = a : p (b, c, d, e, a + b + c + d + e)

pentanacci3  :: Integer -> Integer
pentanacci3 n = pentanaccis3 `genericIndex` n

-- 4ª solución
-- ===========

pentanaccis4 :: [Integer]
```

```
pentanaccis4 = 0: 1: 1: 2: 4: p pentanaccis4
  where p (a:b:c:d:e:xs) = (a+b+c+d+e): p (b:c:d:e:xs)

pentanacci4  :: Integer -> Integer
pentanacci4 n = pentanaccis4 `genericIndex` n

-- Verificación
-- ============

verifica :: IO ()
verifica = hspec spec

spec :: Spec
spec = do
  it "ej1" $
    take 15 pentanaccis1 `shouldBe`
    [0,1,1,2,4,8,16,31,61,120,236,464,912,1793,3525]
  it "ej2" $
    take 15 pentanaccis2 `shouldBe`
    [0,1,1,2,4,8,16,31,61,120,236,464,912,1793,3525]
  it "ej3" $
    take 15 pentanaccis3 `shouldBe`
    [0,1,1,2,4,8,16,31,61,120,236,464,912,1793,3525]
  it "ej4" $
    take 15 pentanaccis4 `shouldBe`
    [0,1,1,2,4,8,16,31,61,120,236,464,912,1793,3525]

-- La verificación es
--    λ> verifica
--
--    4 examples, 0 failures

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

-- La propiedad es
prop_pentanaccis :: NonNegative Int -> Bool
prop_pentanaccis (NonNegative n) =
  all (== pentanaccis1 !! n)
      [pentanaccis1 !! n,
```

```
       pentanaccis2 !! n,
       pentanaccis3 !! n]

-- La comprobación es
--    λ> quickCheckWith (stdArgs {maxSize=25}) prop_pentanaccis
--    +++ OK, passed 100 tests.

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

-- La comparación es
--    λ> pentanacci1 25
--    5976577
--    (4.64 secs, 1,865,626,496 bytes)
--    λ> pentanacci2 25
--    5976577
--    (0.00 secs, 578,584 bytes)
--
--    λ> length (show (pentanacci2 (10^5)))
--    29357
--    (1.16 secs, 2,543,272,136 bytes)
--    λ> length (show (pentanacci3 (10^5)))
--    29357
--    (1.00 secs, 2,560,881,608 bytes)
--    λ> length (show (pentanacci4 (10^5)))
--    29357
--    (1.03 secs, 2,592,078,744 bytes)

-- Referencias
-- ===========

-- + Tito III Piezas y Eric Weisstein, [Pentanacci number](https://bit.ly/3cPJGkF
-- + N. J. A. Sloane, [Sucesión A001591 de la OEIS](https://oeis.org/A001591).
```

16.1.2. En Python

```
# ---------------------------------------------------------------------
# Los números de Fibonacci se definen mediante las ecuaciones
#    F(0) = 0
#    F(1) = 1
```

```
#    F(n) = F(n-1) + F(n-2), si n > 1
# Los primeros números de Fibonacci son
#    0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, ...
#
# Una generalización de los anteriores son los números de Pentanacci
# definidos por las siguientes ecuaciones
#    P(0) = 0
#    P(1) = 1
#    P(2) = 1
#    P(3) = 2
#    P(4) = 4
#    P(n) = P(n-1) + P(n-2) + P(n-3) + P(n-4) + P(n-5), si n > 4
# Los primeros números de Pentanacci son
#   0, 1, 1, 2, 4, 8, 16, 31, 61, 120, 236, 464, 912, 1793, 3525, ...
#
# Definir las funciones
#    pentanacci  : (int) -> int
#    pentanaccis : () -> Iterator[int]
# tales que
# + pentanacci(n) es el n-ésimo número de Pentanacci. Por ejemplo,
#      >>> pentanacci(14)
#      3525
#      >>> pentanacci(10**5) % 10**30
#      482929150584077921552549215816
#      >>> len(str(pentanacci(10**5)))
#      29357
# + pentanaccis() genera los números de Pentanacci. Por ejemplo,
#      >>> list(islice(pentanaccis(), 15))
#      [0, 1, 1, 2, 4, 8, 16, 31, 61, 120, 236, 464, 912, 1793, 3525]
# ---------------------------------------------------------------------

from itertools import count, islice
from sys import set_int_max_str_digits
from timeit import Timer, default_timer
from typing import Iterator

set_int_max_str_digits(30000)

# 1ª solución
# ===========
```

```python
def pentanacci1(n: int) -> int:
    if n == 0:
        return 0
    if n == 1:
        return 1
    if n == 2:
        return 1
    if n == 3:
        return 2
    if n == 4:
        return 4
    return sum((pentanacci1(n-k) for k in range(1, 6)))

def pentanaccis1() -> Iterator[int]:
    return (pentanacci1(n) for n in count())

# 2ª solución
# ===========

def pentanaccis2() -> Iterator[int]:
    seq = [0, 1, 1, 2, 4]
    while True:
        yield seq[0]
        seq.append(sum(seq))
        seq.pop(0)

def nth(i: Iterator[int], n: int) -> int:
    return list(islice(i, n, n+1))[0]

def pentanacci2(n: int) -> int:
    return nth(pentanaccis2(), n)

# Verificación
# ============

def test_pentanacci() -> None:
    assert pentanacci1(14) == 3525
    assert list(islice(pentanaccis1(), 15)) == \
        [0, 1, 1, 2, 4, 8, 16, 31, 61, 120, 236, 464, 912, 1793, 3525]
```

```
    assert pentanacci2(14) == 3525
    assert list(islice(pentanaccis2(), 15)) == \
        [0, 1, 1, 2, 4, 8, 16, 31, 61, 120, 236, 464, 912, 1793, 3525]
    print("Verificado")

# La verificación es
#    >>> test_pentanacci()
#    Verificado

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

# La propiedad es
def test_pentanacci_equiv() -> bool:
    return list(islice(pentanaccis1(), 25)) == list(islice(pentanaccis2(), 25))

# La comprobación es
#    >>> test_pentanacci_equiv()
#    True

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

# 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('pentanacci1(28)')
#    8.24 segundos
#    >>> tiempo('pentanacci2(28)')
#    0.00 segundos

# Referencias
# ===========
```

```
# + Tito III Piezas y Eric Weisstein, [Pentanacci number](https://bit.ly/3cPJGkF)
# + N. J. A. Sloane, [Sucesión A001591 de la OEIS](https://oeis.org/A001591).
```

16.2. El teorema de Navidad de Fermat

16.2.1. En Haskell

```
-- ---------------------------------------------------------------------
-- El 25 de diciembre de 1640, en una carta a Mersenne, Fermat demostró
-- la conjetura de Girard: todo primo de la forma 4n+1 puede expresarse
-- de manera única como suma de dos cuadrados. Por eso es conocido como
-- el [teorema de Navidad de Fermat](http://bit.ly/2Roso1o).
--
-- Definir las funciones
--    representaciones :: Integer -> [(Integer,Integer)]
--    primosImparesConRepresentacionUnica :: [Integer]
--    primos4nM1 :: [Integer]
-- tales que
-- + (representaciones n) es la lista de pares de números naturales
--   (x,y) tales que n = x^2 + y^2 con x <= y. Por ejemplo.
--      representaciones  20           ==  [(2,4)]
--      representaciones  25           ==  [(0,5),(3,4)]
--      representaciones 325           ==  [(1,18),(6,17),(10,15)]
--      representaciones 100000147984  ==  [(0,316228)]
--      length (representaciones (10^10))    ==  6
--      length (representaciones (4*10^12))  ==  7
-- + primosImparesConRepresentacionUnica es la lista de los números
--   primos impares que se pueden escribir exactamente de una manera
--   como suma de cuadrados de pares de números naturales (x,y) con
--   x <= y. Por ejemplo,
--      λ> take 20 primosImparesConRepresentacionUnica
--      [5,13,17,29,37,41,53,61,73,89,97,101,109,113,137,149,157,173,181,193]
-- + primos4nM1 es la lista de los números primos que se pueden escribir
--   como uno más un múltiplo de 4 (es decir, que son congruentes con 1
--   módulo 4). Por ejemplo,
--      λ> take 20 primos4nM1
--      [5,13,17,29,37,41,53,61,73,89,97,101,109,113,137,149,157,173,181,193]
--
-- El teorema de Navidad de Fermat afirma que un número primo impar p se
-- puede escribir exactamente de una manera como suma de dos cuadrados
```

```
-- de números naturales p = x² + y² (con x <= y) si, y sólo si, p se
-- puede escribir como uno más un múltiplo de 4 (es decir, que es
-- congruente con 1 módulo 4).

-- Comprobar con QuickCheck el teorema de Navidad de Fermat; es decir,
-- que para todo número n, los n-ésimos elementos de
-- primosImparesConRepresentacionUnica y de primos4nM1 son iguales.
-- ---------------------------------------------------------------------

module El_teorema_de_Navidad_de_Fermat where

import Data.Numbers.Primes (primes)
import Test.Hspec (Spec, hspec, it, shouldBe)
import Test.QuickCheck

-- 1ª definición de representaciones
-- =================================

representaciones :: Integer -> [(Integer,Integer)]
representaciones n =
  [(x,y) | x <- [0..n], y <- [x..n], n == x*x + y*y]

-- 2ª definición de representaciones
-- =================================

representaciones2 :: Integer -> [(Integer,Integer)]
representaciones2 n =
  [(x,raiz z) | x <- [0..raiz (n `div` 2)]
              , let z = n - x*x
              , esCuadrado z]

-- (esCuadrado x) se verifica si x es un número al cuadrado. Por
-- ejemplo,
--    esCuadrado 25  ==  True
--    esCuadrado 26  ==  False
esCuadrado :: Integer -> Bool
esCuadrado x = x == y * y
  where y = raiz x

-- (raiz x) es la raíz cuadrada entera de x. Por ejemplo,
```

```
--    raiz 25  ==  5
--    raiz 24  ==  4
--    raiz 26  ==  5
raiz :: Integer -> Integer
raiz 0 = 0
raiz 1 = 1
raiz x = aux (0,x)
    where aux (a,b) | d == x    = c
                    | c == a    = a
                    | d < x     = aux (c,b)
                    | otherwise = aux (a,c)
              where c = (a+b) `div` 2
                    d = c^2

-- 3ª definición de representaciones
-- =================================

representaciones3 :: Integer -> [(Integer,Integer)]
representaciones3 n =
  [(x,raiz3 z) | x <- [0..raiz3 (n `div` 2)]
               , let z = n - x*x
               , esCuadrado3 z]

-- (esCuadrado3 x) se verifica si x es un número al cuadrado. Por
-- ejemplo,
--    esCuadrado3 25  ==  True
--    esCuadrado3 26  ==  False
esCuadrado3 :: Integer -> Bool
esCuadrado3 x = x == y * y
  where y = raiz3 x

-- (raiz3 x) es la raíz cuadrada entera de x. Por ejemplo,
--    raiz3 25  ==  5
--    raiz3 24  ==  4
--    raiz3 26  ==  5
raiz3 :: Integer -> Integer
raiz3 x = floor (sqrt (fromIntegral x))

-- 4ª definición de representaciones
-- =================================
```

```haskell
representaciones4 :: Integer -> [(Integer, Integer)]
representaciones4 n = aux 0 (floor (sqrt (fromIntegral n)))
  where aux x y
          | x > y     = []
          | otherwise = case compare (x*x + y*y) n of
                          LT -> aux (x + 1) y
                          EQ -> (x, y) : aux (x + 1) (y - 1)
                          GT -> aux x (y - 1)

-- Equivalencia de las definiciones de representaciones
-- ====================================================

-- La propiedad es
prop_representaciones_equiv :: Positive Integer -> Bool
prop_representaciones_equiv (Positive n) =
  representaciones  n == representaciones2 n &&
  representaciones2 n == representaciones3 n &&
  representaciones3 n == representaciones4 n

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

-- Comparación de eficiencia de las definiciones de representaciones
-- =================================================================

--    λ> representaciones 3025
--    [(0,55),(33,44)]
--    (2.86 secs, 1,393,133,528 bytes)
--    λ> representaciones2 3025
--    [(0,55),(33,44)]
--    (0.00 secs, 867,944 bytes)
--    λ> representaciones3 3025
--    [(0,55),(33,44)]
--    (0.00 secs, 173,512 bytes)
--    λ> representaciones4 3025
--    [(0,55),(33,44)]
--    (0.00 secs, 423,424 bytes)
--
```

```haskell
--    λ> length (representaciones2 (10^10))
--    6
--    (3.38 secs, 2,188,903,544 bytes)
--    λ> length (representaciones3 (10^10))
--    6
--    (0.10 secs, 62,349,048 bytes)
--    λ> length (representaciones4 (10^10))
--    6
--    (0.11 secs, 48,052,360 bytes)
--
--    λ> length (representaciones3 (4*10^12))
--    7
--    (1.85 secs, 1,222,007,176 bytes)
--    λ> length (representaciones4 (4*10^12))
--    7
--    (1.79 secs, 953,497,480 bytes)

-- Definición de primosImparesConRepresentacionUnica
-- =================================================

primosImparesConRepresentacionUnica :: [Integer]
primosImparesConRepresentacionUnica =
  [x | x <- tail primes
     , length (representaciones4 x) == 1]

-- Definición de primos4nM1
-- ========================

primos4nM1 :: [Integer]
primos4nM1 = [x | x <- primes
                , x `mod` 4 == 1]

-- Teorema de Navidad de Fermat
-- ============================

-- La propiedad es
prop_teoremaDeNavidadDeFermat :: Positive Int -> Bool
prop_teoremaDeNavidadDeFermat (Positive n) =
  primosImparesConRepresentacionUnica !! n == primos4nM1 !! n
```

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

-- ---------------------------------------------------------------------
-- § Verificación                                                     --
-- ---------------------------------------------------------------------

verifica :: IO ()
verifica = hspec spec

spec :: Spec
spec = do
  it "e1" $
    representaciones  20 `shouldBe`  [(2,4)]
  it "e2" $
    representaciones  25 `shouldBe`  [(0,5),(3,4)]
  it "e3" $
    representaciones 325 `shouldBe`  [(1,18),(6,17),(10,15)]
  it "e4" $
    take 20 primosImparesConRepresentacionUnica `shouldBe`
     [5,13,17,29,37,41,53,61,73,89,97,101,109,113,137,149,157,173,181,193]
  it "e5" $
    take 20 primos4nM1 `shouldBe`
     [5,13,17,29,37,41,53,61,73,89,97,101,109,113,137,149,157,173,181,193]

-- La verificación es
--    λ> verifica
--
--    5 examples, 0 failures
```

16.2.2. En Python

```
# ---------------------------------------------------------------------
# El 25 de diciembre de 1640, en una carta a Mersenne, Fermat demostró
# la conjetura de Girard: todo primo de la forma 4n+1 puede expresarse
# de manera única como suma de dos cuadrados. Por eso es conocido como
# el [teorema de Navidad de Fermat](http://bit.ly/2Roso1o).
#
# Definir las funciones
```

```
#    representaciones : (int) -> list[tuple[int, int]]
#    primosImparesConRepresentacionUnica : () -> Iterator[int]
#    primos4nM1 : () -> Iterator[int]
# tales que
# + representaciones(n) es la lista de pares de números naturales
#   (x,y) tales que n = x^2 + y^2 con x <= y. Por ejemplo.
#      representaciones(20)           ==  [(2,4)]
#      representaciones(25)           ==  [(0,5),(3,4)]
#      representaciones(325)          ==  [(1,18),(6,17),(10,15)]
#      representaciones(100000147984) ==  [(0,316228)]
#      length (representaciones (10^10))    ==  6
#      length (representaciones (4*10^12))  ==  7
# + primosImparesConRepresentacionUnica() genera los números primos
#   impares que se pueden escribir exactamente de una manera como suma
#   de cuadrados de pares de números naturales (x,y) con x <= y. Por
#   ejemplo,
#      >>> list(islice(primosImparesConRepresentacionUnica(), 20))
#      [5,13,17,29,37,41,53,61,73,89,97,101,109,113,137,149,157,173,181,193]
# + primos4nM1() genera los números primos que se pueden escribir  como
#   uno más un múltiplo de 4 (es decir, que son congruentes con 1 módulo
#   4). Por ejemplo,
#      >>> list(islice(primos4nM1(), 20))
#      [5,13,17,29,37,41,53,61,73,89,97,101,109,113,137,149,157,173,181,193]
#
# El teorema de Navidad de Fermat afirma que un número primo impar p se
# puede escribir exactamente de una manera como suma de dos cuadrados de
# números naturales p = x² + y² (con x <= y) si, y sólo si, p se puede
# escribir como uno más un múltiplo de 4 (es decir, que es congruente
# con 1 módulo 4).
#
# Comprobar con Hypothesis el teorema de Navidad de Fermat; es decir,
# que para todo número n, los n-ésimos elementos de
# primosImparesConRepresentacionUnica y de primos4nM1 son iguales.
# ---------------------------------------------------------------------

from itertools import count, islice
from math import floor, sqrt
from timeit import Timer, default_timer
from typing import Iterator
```

```python
from hypothesis import given
from hypothesis import strategies as st
from sympy import isprime

# 1ª definición de representaciones
# =================================

def representaciones(n: int) -> list[tuple[int, int]]:
    return [(x,y) for x in range(n+1) for y in range(x,n+1) if n == x*x + y*y]

# 2ª definición de representaciones
# =================================

# raiz(x) es la raíz cuadrada entera de x. Por ejemplo,
#    raiz(25)     ==  5
#    raiz(24)     ==  4
#    raiz(26)     ==  5
#    raiz(10**46) == 100000000000000000000000
def raiz(x: int) -> int:
    def aux(a: int, b: int) -> int:
        c = (a+b) // 2
        d = c**2
        if d == x:
            return c
        if c == a:
            return a
        if d < x:
            return aux (c,b)
        return aux (a,c)
    if x == 0:
        return 0
    if x == 1:
        return 1
    return aux(0, x)

# Nota. La siguiente definición de raíz cuadrada entera falla para
# números grandes. Por ejemplo,
#    >>> raiz2(10**46)
#    99999999999999991611392
def raiz2(x: int) -> int:
```

```
    return floor(sqrt(x))

# esCuadrado(x) se verifica si x es un número al cuadrado. Por
# ejemplo,
#    esCuadrado(25)     == True
#    esCuadrado(26)     == False
#    esCuadrado(10**46) == True
#    esCuadrado(10**47) == False
def esCuadrado(x: int) -> bool:
    y = raiz(x)
    return x == y * y

def representaciones2(n: int) -> list[tuple[int, int]]:
    r: list[tuple[int, int]] = []
    for x in range(1 + raiz(n // 2)):
        z = n - x*x
        if esCuadrado(z):
            r.append((x, raiz(z)))
    return r

# 3ª definición de representaciones
# =================================

def representaciones3(n: int) -> list[tuple[int, int]]:
    r: list[tuple[int, int]] = []
    for x in range(1 + raiz(n // 2)):
        y = n - x*x
        z = raiz(y)
        if y == z * z:
            r.append((x, z))
    return r

# Verificación
# ============

def test_representaciones() -> None:
    assert representaciones(20) == [(2,4)]
    assert representaciones(25) == [(0,5),(3,4)]
    assert representaciones(325) == [(1,18),(6,17),(10,15)]
    assert representaciones2(20) == [(2,4)]
```

```
    assert representaciones2(25) == [(0,5),(3,4)]
    assert representaciones2(325) == [(1,18),(6,17),(10,15)]
    assert representaciones3(20) == [(2,4)]
    assert representaciones3(25) == [(0,5),(3,4)]
    assert representaciones3(325) == [(1,18),(6,17),(10,15)]
    print("Verificado")

# La comprobación es
#    >>> test_representaciones()
#    Verificado

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

# La propiedad es
@given(st.integers(min_value=1, max_value=1000))
def test_representaciones_equiv(x: int) -> None:
    xs = representaciones(x)
    assert representaciones2(x) == xs
    assert representaciones3(x) == xs

# La comprobación es
#    >>> test_representaciones_equiv()
#    >>>

# Comparación de eficiencia de las definiciones de representaciones
# =================================================================

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('representaciones(5000)')
#    1.27 segundos
#    >>> tiempo('representaciones2(5000)')
#    0.00 segundos
#    >>> tiempo('representaciones3(5000)')
#    0.00 segundos
```

```
#
#    >>> tiempo('len(representaciones2(10**12))')
#    11.54 segundos
#    >>> tiempo('len(representaciones3(10**12))')
#    12.08 segundos

# Definición de primosImparesConRepresentacionUnica
# =================================================

# primos() genera la lista de los primos. Por ejemplo,
#    >>> list(islice(primos(), 10))
#    [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
def primos() -> Iterator[int]:
    return (n for n in count() if isprime(n))

def primosImparesConRepresentacionUnica() -> Iterator[int]:
    return (x for x in islice(primos(), 1, None)
            if len(representaciones2(x)) == 1)

# Verificación de primosImparesConRepresentacionUnica
# ===================================================

def test_primosImparesConRepresentacionUnica() -> None:
    assert list(islice(primosImparesConRepresentacionUnica(), 20)) == \
        [5,13,17,29,37,41,53,61,73,89,97,101,109,113,137,149,157,173,181,193]
    print("Verificado")

# La comprobación es
#    >>> test_primosImparesConRepresentacionUnica()
#    Verificado

# Definición de primos4nM1
# ========================

def primos4nM1() -> Iterator[int]:
    return (x for x in primos() if x % 4 == 1)

# La comprobación es
#    >>> test_primos4nM1()
#    Verificado
```

```
# Verificación de primos4nM1
# ==========================

def test_primos4nM1() -> None:
    assert list(islice(primos4nM1(), 20)) == \
        [5,13,17,29,37,41,53,61,73,89,97,101,109,113,137,149,157,173,181,193]
    print("Verificado")

# Teorema de Navidad de Fermat
# ============================

# nth(i, n) es el n-ésimo elemento del iterador i. Por ejemplo,
#    nth(primos(), 4) == 11
def nth(i: Iterator[int], n: int) -> int:
    return list(islice(i, n, n+1))[0]

# La propiedad es
@given(st.integers(min_value=1, max_value=300))
def test_teoremaDeNavidadDeFermat(n: int) -> None:
    assert nth(primosImparesConRepresentacionUnica(), n) == nth(primos4nM1(), n)

# La comprobación es
#    >>> test_teoremaDeNavidadDeFermat()
#    >>>
```

16.3. Números primos de Hilbert

16.3.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Un [número de Hilbert](http://bit.ly/204SW1p) es un entero positivo
-- de la forma 4n+1. Los primeros números de Hilbert son 1, 5, 9, 13,
-- 17, 21, 25, 29, 33, 37, 41, 45, 49, 53, 57, 61, 65, 69, 73, 77, 81,
-- 85, 89, 93, 97, ...
--
-- Un primo de Hilbert es un número de Hilbert n que no es divisible
-- por ningún número de Hilbert menor que n (salvo el 1). Los primeros
-- primos de Hilbert son 5, 9, 13, 17, 21, 29, 33, 37, 41, 49, 53, 57,
-- 61, 69, 73, 77, 89, 93, 97, 101, 109, 113, 121, 129, 133, 137, 141,
```

```haskell
-- 149, 157, 161, 173, 177, 181, 193, 197, ...
--
-- Definir la sucesión
--    primosH :: [Integer]
-- tal que sus elementos son los primos de Hilbert. Por ejemplo,
--    take 15 primosH     == [5,9,13,17,21,29,33,37,41,49,53,57,61,69,73]
--    primosH !! (3*10^4) == 313661
-- ---------------------------------------------------------------------

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

module Numeros_primos_de_Hilbert where

import Data.Numbers.Primes (isPrime, primeFactors)
import Test.QuickCheck (NonNegative (NonNegative), quickCheck)
import Test.Hspec (Spec, hspec, it, shouldBe)

-- 1ª solución
-- ===========

primosH1 :: [Integer]
primosH1 = [n | n <- tail numerosH,
                divisoresH n == [1,n]]

-- numerosH es la sucesión de los números de Hilbert. Por ejemplo,
--    take 15 numerosH  ==  [1,5,9,13,17,21,25,29,33,37,41,45,49,53,57]
numerosH :: [Integer]
numerosH = [1,5..]

-- (divisoresH n) es la lista de los números de Hilbert que dividen a
-- n. Por ejemplo,
--   divisoresH 117  ==  [1,9,13,117]
--   divisoresH  21  ==  [1,21]
divisoresH :: Integer -> [Integer]
divisoresH n = [x | x <- takeWhile (<=n) numerosH,
                    n `mod` x == 0]

-- 2ª solución
-- ===========
```

```
primosH2 :: [Integer]
primosH2 = filter esPrimoH (tail numerosH)
  where esPrimoH n = all noDivideAn [5,9..m]
          where noDivideAn x = n `mod` x /= 0
                m            = ceiling (sqrt (fromIntegral n))

-- 3ª solución
-- ===========

-- Basada en la siguiente propiedad: Un primo de Hilbert es un primo
-- de la forma 4n + 1 o un semiprimo de la forma (4a + 3) × (4b + 3)
-- (ver en https://bit.ly/3zq7h4e ).

primosH3 :: [Integer]
primosH3 = [ n | n <- numerosH, isPrime n || semiPrimoH n ]
  where semiPrimoH n = length xs == 2 && all (\x -> (x-3) `mod` 4 == 0) xs
          where xs = primeFactors n

-- Verificación                                                     --
-- ============

verifica :: IO ()
verifica = hspec spec

spec :: Spec
spec = do
  it "e1" $
    take 15 primosH1 `shouldBe` [5,9,13,17,21,29,33,37,41,49,53,57,61,69,73]
  it "e2" $
    take 15 primosH2 `shouldBe` [5,9,13,17,21,29,33,37,41,49,53,57,61,69,73]
  it "e3" $
    take 15 primosH3 `shouldBe` [5,9,13,17,21,29,33,37,41,49,53,57,61,69,73]

-- La verificación es
--    λ> verifica
--
--    3 examples, 0 failures

-- Comprobación de equivalencia
```

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

-- La propiedad es
prop_primosH :: NonNegative Int -> Bool
prop_primosH (NonNegative n) =
  all (== primosH1 !! n)
      [primosH2 !! n,
       primosH3 !! n]

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

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

-- La comparación es
--    λ> primosH1 !! 2000
--    16957
--    (2.16 secs, 752,085,752 bytes)
--    λ> primosH2 !! 2000
--    16957
--    (0.03 secs, 19,771,008 bytes)
--    λ> primosH3 !! 2000
--    16957
--    (0.07 secs, 152,029,168 bytes)
--
--    λ> primosH2 !! (3*10^4)
--    313661
--    (1.44 secs, 989,761,888 bytes)
--    λ> primosH3 !! (3*10^4)
--    313661
--    (2.06 secs, 6,554,068,992 bytes)
```

16.3.2. En Python

```
# ---------------------------------------------------------------------
# Un [número de Hilbert](http://bit.ly/204SW1p) es un entero positivo
# de la forma 4n+1. Los primeros números de Hilbert son 1, 5, 9, 13,
# 17, 21, 25, 29, 33, 37, 41, 45, 49, 53, 57, 61, 65, 69, 73, 77, 81,
```

```
# 85, 89, 93, 97, ...
#
# Un primo de Hilbert es un número de Hilbert n que no es divisible
# por ningún número de Hilbert menor que n (salvo el 1). Los primeros
# primos de Hilbert son 5, 9, 13, 17, 21, 29, 33, 37, 41, 49, 53, 57,
# 61, 69, 73, 77, 89, 93, 97, 101, 109, 113, 121, 129, 133, 137, 141,
# 149, 157, 161, 173, 177, 181, 193, 197, ...
#
# Definir la sucesión
#    primosH :: [Integer]
# tal que sus elementos son los primos de Hilbert. Por ejemplo,
#    >>> list(islice(primosH1(), 15))
#    [5, 9, 13, 17, 21, 29, 33, 37, 41, 49, 53, 57, 61, 69, 73]
#    >>> nth(primosH1(), 5000)
#    45761
# ---------------------------------------------------------------------

from itertools import count, islice, takewhile
from math import ceil, sqrt
from timeit import Timer, default_timer
from typing import Iterator

from hypothesis import given
from hypothesis import strategies as st

# 1ª solución
# ===========

# numerosH es la sucesión de los números de Hilbert. Por ejemplo,
#    >>> list(islice(numerosH(), 15))
#    [1, 5, 9, 13, 17, 21, 25, 29, 33, 37, 41, 45, 49, 53, 57]
def numerosH() -> Iterator[int]:
    return count(1, 4)

# (divisoresH n) es la lista de los números de Hilbert que dividen a
# n. Por ejemplo,
#   divisoresH(117)  ==  [1,9,13,117]
#   divisoresH(21)   ==  [1,21]
def divisoresH(n: int) -> list[int]:
    return [x for x in takewhile(lambda x: x <= n, numerosH())
```

```
            if n % x == 0]

def primosH1() -> Iterator[int]:
    return (n for n in islice(numerosH(), 1, None)
            if divisoresH(n) == [1, n])

# 2ª solución
# ===========

def primosH2() -> Iterator[int]:
    def esPrimoH(n: int) -> bool:
        m = ceil(sqrt(n))
        def noDivideAn(x: int) -> bool:
            return n % x != 0
        return all(noDivideAn(x) for x in range(5, m+1, 4))
    return filter(esPrimoH, islice(numerosH(), 1, None))

# Verificación                                                          --
# ============

def test_primosH() -> None:
    assert list(islice(primosH1(), 15)) == \
        [5, 9, 13, 17, 21, 29, 33, 37, 41, 49, 53, 57, 61, 69, 73]
    assert list(islice(primosH2(), 15)) == \
        [5, 9, 13, 17, 21, 29, 33, 37, 41, 49, 53, 57, 61, 69, 73]
    print ("Verificado")

# La verificación es
#    >>> test_primosH()
#    Verificado

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

# nth(i, n) es el n-ésimo elemento del iterador i. Por ejemplo,
#    nth(primos(), 4) == 11
def nth(i: Iterator[int], n: int) -> int:
    return list(islice(i, n, n+1))[0]

# La propiedad es
```

```
@given(st.integers(min_value=1, max_value=1000))
def test_primosH_equiv(n: int) -> None:
    assert nth(primosH1(), n) == nth(primosH2(), n)

# La comprobación es
#    >>> test_primosH_equiv()
#    >>>

# 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('nth(primosH1(), 3000)')
#    2.51 segundos
#    >>> tiempo('nth(primosH2(), 3000)')
#    0.05 segundos
```

16.4. Factorizaciones de números de Hilbert

16.4.1. En Haskell

```
-- ---------------------------------------------------------------------
-- Un [**número de Hilbert**](http://bit.ly/204SW1p) es un entero
-- positivo de la forma 4n+1. Los primeros números de Hilbert son 1, 5,
-- 9, 13, 17, 21, 25, 29, 33, 37, 41, 45, 49, 53, 57, 61, 65, 69, ...
--
-- Un **primo de Hilbert** es un número de Hilbert n que no es divisible
-- por ningún número de Hilbert menor que n (salvo el 1). Los primeros
-- primos de Hilbert son 5, 9, 13, 17, 21, 29, 33, 37, 41, 49, 53, 57,
-- 61, 69, 73, 77, 89, 93, 97, 101, 109, 113, 121, 129, 133, 137, ...
--
-- Definir la función
--    factorizacionesH :: Integer -> [[Integer]]
-- tal que (factorizacionesH n) es la listas de primos de Hilbert cuyo
-- producto es el número de Hilbert n. Por ejemplo,
```

```haskell
--   factorizacionesH  25    ==  [[5,5]]
--   factorizacionesH  45    ==  [[5,9]]
--   factorizacionesH 441    ==  [[9,49],[21,21]]
--   factorizacionesH 80109  ==  [[9,9,989],[9,69,129]]
--
-- Comprobar con QuickCheck que todos los números de Hilbert son
-- factorizables como producto de primos de Hilbert (aunque la
-- factorización, como para el 441, puede no ser única).
-- ---------------------------------------------------------------------

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

module Factorizaciones_de_numeros_de_Hilbert where

import Data.Numbers.Primes (isPrime, primeFactors)
import Test.QuickCheck (Positive (Positive), quickCheck)
import Test.Hspec (Spec, describe, hspec, it, shouldBe)
import Numeros_primos_de_Hilbert (primosH1, primosH2, primosH3)

import Data.Traversable (forM)

-- 1ª solución
-- ===========

factorizacionesH1 :: Integer -> [[Integer]]
factorizacionesH1 = aux primosH1
  where
    aux (x:xs) n
      | x == n          = [[n]]
      | x > n           = []
      | n `mod` x == 0 = map (x:) (aux (x:xs) (n `div` x) ) ++ aux xs n
      | otherwise       = aux xs n

-- 2ª solución
-- ===========

factorizacionesH2 :: Integer -> [[Integer]]
factorizacionesH2 = aux primosH2
```

```
  where
    aux (x:xs) n
      | x == n           = [[n]]
      | x > n            = []
      | n `mod` x == 0 = map (x:) (aux (x:xs) (n `div` x) ) ++ aux xs n
      | otherwise        = aux xs n

-- 3ª solución
-- ===========

-- Basada en la siguiente propiedad: Un primo de Hilbert es un primo
-- de la forma 4n + 1 o un semiprimo de la forma (4a + 3) × (4b + 3)
-- (ver en https://bit.ly/3zq7h4e ).

factorizacionesH3 :: Integer -> [[Integer]]
factorizacionesH3 = aux primosH3
  where
    aux (x:xs) n
      | x == n           = [[n]]
      | x > n            = []
      | n `mod` x == 0 = map (x:) (aux (x:xs) (n `div` x) ) ++ aux xs n
      | otherwise        = aux xs n

-- Verificación                                                      --
-- ============

verifica :: IO ()
verifica = hspec spec

specG :: (Integer -> [[Integer]]) -> Spec
specG factorizacionesH = do
  it "e1" $
    factorizacionesH  25 `shouldBe` [[5,5]]
  it "e2" $
    factorizacionesH  45 `shouldBe` [[5,9]]
  it "e3" $
    factorizacionesH 441 `shouldBe` [[9,49],[21,21]]

spec :: Spec
spec = do
```

```
  describe "def. 1" $ specG factorizacionesH1
  describe "def. 2" $ specG factorizacionesH2
  describe "def. 3" $ specG factorizacionesH3

-- La verificación es
--    λ> verifica
--
--    9 examples, 0 failures

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

-- La propiedad es
prop_factorizacionesH :: Positive Integer -> Bool
prop_factorizacionesH (Positive n) =
  all (== factorizacionesH1 m)
      [factorizacionesH2 m,
       factorizacionesH3 m]
  where m = 1 + 4 * n

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

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

-- La comparación es
--    λ> factorizacionesH1 80109
--    [[9,9,989],[9,69,129]]
--    (42.77 secs, 14,899,787,640 bytes)
--    λ> factorizacionesH2 80109
--    [[9,9,989],[9,69,129]]
--    (0.26 secs, 156,051,104 bytes)
--    λ> factorizacionesH3 80109
--    [[9,9,989],[9,69,129]]
--    (0.35 secs, 1,118,236,536 bytes)

-- Propiedad de factorización
-- ==========================
```

```haskell
-- La propiedad es
prop_factorizable :: Positive Integer -> Bool
prop_factorizable (Positive n) =
  not (null (factorizacionesH1 (1 + 4 * n)))

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

-- ---------------------------------------------------------------------
-- § Referencias                                                      --
-- ---------------------------------------------------------------------

-- Basado en el artículo [Failure of unique factorization (A simple
-- example of the failure of the fundamental theorem of
-- arithmetic)](http://bit.ly/20A2Nyc) de R.J. Lipton en el blog [Gödel's
-- Lost Letter and P=NP](https://rjlipton.wordpress.com).
--
-- Otras  referencias
--
-- + Wikipedia, [Hilbert number](http://bit.ly/204SW1p).
-- + E.W. Weisstein, [Hilbert number](http://bit.ly/204T8O4) en MathWorld.
-- + N.J.A. Sloane, [Sucesión A057948](https://oeis.org/A057948) en la
--   OEIS.
-- + N.J.A. Sloane, [Sucesión A057949](https://oeis.org/A057949) en la
--   OEIS.
```

16.4.2. En Python

```python
# ---------------------------------------------------------------------
# Un [**número de Hilbert**](http://bit.ly/204SW1p) es un entero
# positivo de la forma 4n+1. Los primeros números de Hilbert son 1, 5,
# 9, 13, 17, 21, 25, 29, 33, 37, 41, 45, 49, 53, 57, 61, 65, 69, ...
#
# Un **primo de Hilbert** es un número de Hilbert n que no es divisible
# por ningún número de Hilbert menor que n (salvo el 1). Los primeros
# primos de Hilbert son 5, 9, 13, 17, 21, 29, 33, 37, 41, 49, 53, 57,
# 61, 69, 73, 77, 89, 93, 97, 101, 109, 113, 121, 129, 133, 137, ...
#
```

```
# Definir la función
#    factorizacionesH : (int) -> list[list[int]]
# tal que factorizacionesH(n) es la listas de primos de Hilbert cuyo
# producto es el número de Hilbert n. Por ejemplo,
#   factorizacionesH(25)    ==  [[5,5]]
#   factorizacionesH(45)    ==  [[5,9]]
#   factorizacionesH(441)   ==  [[9,49],[21,21]]
#   factorizacionesH(80109) ==  [[9,9,989],[9,69,129]]
#
# Comprobar con Hypothesis que todos los números de Hilbert son
# factorizables como producto de primos de Hilbert (aunque la
# factorización, como para el 441, puede no ser única).
# ---------------------------------------------------------------------

from itertools import takewhile
from sys import setrecursionlimit
from timeit import Timer, default_timer

from hypothesis import given
from hypothesis import strategies as st

from src.Numeros_primos_de_Hilbert import primosH1, primosH2

setrecursionlimit(10**6)

# 1ª solución
# ===========

def factorizacionesH1(m: int) -> list[list[int]]:
    ys = list(takewhile(lambda y: y <= m, primosH1()))
    def aux(zs: list[int], n: int) -> list[list[int]]:
        if not zs:
            return []
        x, *xs = zs
        if x == n:
            return [[n]]
        if x > n:
            return []
        if n % x == 0:
            return [[x] + ns for ns in aux(zs, n // x)] + aux(xs, n)
```

```
        return aux(xs, n)
    return aux(ys, m)

# 2ª solución
# ===========

def factorizacionesH2(m: int) -> list[list[int]]:
    ys = list(takewhile(lambda y: y <= m, primosH2()))
    def aux(zs: list[int], n: int) -> list[list[int]]:
        if not zs:
            return []
        x, *xs = zs
        if x == n:
            return [[n]]
        if x > n:
            return []
        if n % x == 0:
            return [[x] + ns for ns in aux(zs, n // x)] + aux(xs, n)
        return aux(xs, n)
    return aux(ys, m)

# Verificación
# ============

def test_factorizacionesH() -> None:
    for factorizacionesH in [factorizacionesH1, factorizacionesH2]:
        assert factorizacionesH(25) == [[5,5]]
        assert factorizacionesH(45) == [[5,9]]
        assert factorizacionesH(441) == [[9,49],[21,21]]
    print("Verificado")

# La verificación es
#    >>> test_factorizacionesH()
#    Verificado

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

# La propiedad es
@given(st.integers(min_value=1, max_value=1000))
```

```python
def test_factorizacionesH_equiv(n: int) -> None:
    m = 1 + 4 * n
    assert factorizacionesH1(m) == factorizacionesH2(m)

# La comprobación es
#    >>> test_factorizacionesH_equiv()
#    >>>

# 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('factorizacionesH1(80109)')
#    25.40 segundos
#    >>> tiempo('factorizacionesH2(80109)')
#    0.27 segundos

# Propiedad de factorización
# ==========================

# La propiedad es
@given(st.integers(min_value=1, max_value=1000))
def test_factorizable(n: int) -> None:
    assert factorizacionesH2(1 + 4 * n) != []

# La comprobación es
#    >>> test_factorizable()
#    >>>

# Comprobación de todas las propiedades
# =====================================

# La comprobación es
#    src> poetry run pytest -v Factorizaciones_de_numeros_de_Hilbert.py
#    ===== test session starts =====
```

```
#    test_factorizacionesH PASSED
#    test_factorizacionesH_equiv PASSED
#    test_factorizable PASSED
#    ===== 3 passed in 2.25s =====

# ---------------------------------------------------------------------
# § Referencias                                                      --
# ---------------------------------------------------------------------

# Basado en el artículo [Failure of unique factorization (A simple
# example of the failure of the fundamental theorem of
# arithmetic)](http://bit.ly/20A2Nyc) de R.J. Lipton en el blog [Gödel's
# Lost Letter and P=NP](https://rjlipton.wordpress.com).
#
# Otras  referencias
#
# + Wikipedia, [Hilbert number](http://bit.ly/204SW1p).
# + E.W. Weisstein, [Hilbert number](http://bit.ly/204T8O4) en MathWorld.
# + N.J.A. Sloane, [Sucesión A057948](https://oeis.org/A057948) en la
#   OEIS.
# + N.J.A. Sloane, [Sucesión A057949](https://oeis.org/A057949) en la
#   OEIS.
```

Apéndices

Apéndice A

Método de Pólya para la resolución de problemas

A.1. Método de Pólya para resolver 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.