Ejercicios de programación con Python

José A. Alonso Jiménez

Grupo de Lógica Computacional
Dpto. de Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, 2 de abril de 2024

Índice general

Introducción

Este libro es una colección de relaciones de ejercicios de programación con Python. Está basada en la de Ejercicios de programación funcional con Haskell que se ha usado en el curso de Informática (de 1º del Grado en Matemáticas de la Universidad de Sevilla).

Las relaciones están ordenadas según los temas del curso.

El código de los ejercicios de encuentra en el repositorio I1M–Ejercicios–Python [1] de GitHub.

[1] https://github.com/jaalonso/Ejercicios-Python

Parte I

Introducción a la programación con Python

Capítulo 1

Definiciones elementales de funciones

1.1. Definiciones por composición sobre números, listas y booleanos

```
# ---------------------------------------------------------------------
# Introducción                                                       --
# ---------------------------------------------------------------------

# En esta relación se plantean ejercicios con definiciones de funciones
# por composición sobre números, listas y booleanos.

# ---------------------------------------------------------------------
# Cabecera
# ---------------------------------------------------------------------

from math import pi
from typing import TypeVar

from hypothesis import given
from hypothesis import strategies as st

A = TypeVar('A')

# ---------------------------------------------------------------------
# Ejercicio 1. 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

# ----------------------------------------------------------------------
# Ejercicio 2. 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

# ----------------------------------------------------------------------
# Ejercicio 3. 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
# ----------------------------------------------------------------------

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

# ----------------------------------------------------------------------
# Ejercicio 4. 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
# ---------------------------------------------------------------------

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

# ---------------------------------------------------------------------
# Ejercicio 5. 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

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

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

# ---------------------------------------------------------------------
# Ejercicio 7. 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']
# ---------------------------------------------------------------------
```

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

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

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

# La equivalencia de las definiciones es
@given(st.lists(st.integers()))
def test_rota1(xs: list[int]) -> None:
    assert rota1a(xs) == rota1b(xs) == rota1c(xs)

# La comprobación está al final

# --------------------------------------------------------------------
# Ejercicio 8. 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]
# --------------------------------------------------------------------

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

```
# ---------------------------------------------------------------------
# Ejercicio 9. 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)]

# ---------------------------------------------------------------------
# Ejercicio 10. 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
# ---------------------------------------------------------------------

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

# ---------------------------------------------------------------------
# Ejercicio 11. 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]
# ---------------------------------------------------------------------

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

# 2ª solución
def interior2(xs: list[A]) -> list[A]:
    return xs[1:-1]
```

```python
# La propiedad de equivalencia es
@given(st.lists(st.integers()))
def test_interior(xs):
    assert interior1(xs) == interior2(xs)

# La comprobación está al final

# --------------------------------------------------------------------
# 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]
# --------------------------------------------------------------------

# 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, xs):
    assert finales1(n, xs) == finales2(n, xs) == finales3(n, xs)

# La comprobación está al final.
```

```
# ---------------------------------------------------------------------
# Ejercicio 13. 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])  ==  []
# ---------------------------------------------------------------------

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

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

# La propiedad de equivalencia es
@given(st.integers(), st.integers(), st.lists(st.integers()))
def test_equiv_segmento(m, n, xs):
    assert segmento1(m, n, xs) == segmento2(m, n, xs)

# La comprobación está al final.

# ---------------------------------------------------------------------
# Ejercicio 14. 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]
# ---------------------------------------------------------------------

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

# ---------------------------------------------------------------------
# Ejercicio 15. 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])

# ---------------------------------------------------------------------
# Ejercicio 16. 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
# ---------------------------------------------------------------------

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

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

# La propiedad de equivalencia es
@given(st.integers(), st.integers(), st.integers())
def test_equiv_tresIguales(x, y, z):
    assert tresIguales1(x, y, z) == tresIguales2(x, y, z)

# La comprobación está al final.

# ---------------------------------------------------------------------
# Ejercicio 17. 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
# ---------------------------------------------------------------------

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

# ---------------------------------------------------------------------
# Ejercicio 18. Definir la función
#    cuatroIguales : (int, int, int, int) -> bool
# tal que cuatroIguales(x,y,z,u) se verifica si los elementos x, y, z y
# u son iguales. Por ejemplo,
#    cuatroIguales(5, 5, 5, 5) == True
#    cuatroIguales(5, 5, 4, 5) == False
# ---------------------------------------------------------------------

# 1ª solución
def cuatroIguales1(x: int, y: int, z: int, u: int) -> bool:
    return x == y and tresIguales1(y, z, u)

# 2ª solución
def cuatroIguales2(x: int, y: int, z: int, u: int) -> bool:
    return x == y == z == u

# La propiedad de equivalencia es
@given(st.integers(), st.integers(), st.integers(), st.integers())
def test_equiv_cuatroIguales(x, y, z, u):
    assert cuatroIguales1(x, y, z, u) == cuatroIguales2(x, y, z, u)

# La comprobación está al final.

# La comprobación de las propiedades es
#    src> poetry run pytest -q definiciones_por_composicion.py
#    6 passed in 0.81s
```

1.2. Definiciones con condicionales, guardas o patrones

```
# ---------------------------------------------------------------------
# Introducción                                                       --
# ---------------------------------------------------------------------

# En esta relación se presentan ejercicios con definiciones elementales
# (no recursivas) de funciones que usan condicionales, guardas o
# patrones.
#
# Estos ejercicios se corresponden con el tema 4 del curso cuyas apuntes
# se encuentran en https://bit.ly/3x1ze0u

# ---------------------------------------------------------------------
# Cabecera
# ---------------------------------------------------------------------

from math import gcd, sqrt
from typing import TypeVar

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

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

# ---------------------------------------------------------------------
# Ejercicio 1. 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
# ---------------------------------------------------------------------

# 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, y):
    assert divisionSegura1(x, y) == divisionSegura2(x, y)

# La comprobación está al final de la relación.

# ---------------------------------------------------------------------
# Ejercicio 2. 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
# ---------------------------------------------------------------------

# 1ª solución
def xor1(x, y):
    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)

# 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, y):
    assert xor1(x, y) == xor2(x, y) == xor3(x, y) == xor4(x, y) == xor5(x, y)

# La comprobación está al final de la relación.

# -----------------------------------------------------------------------
# Ejercicio 3. 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)

# ---------------------------------------------------------------------
# Ejercicio 4. 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 esidempotente; es
# decir, si se aplica dos veces es lo mismo que no aplicarla ninguna.
# ---------------------------------------------------------------------

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):
    assert intercambia(intercambia(p)) == p

# La comprobación está al final de la relación.

# ---------------------------------------------------------------------
# Ejercicio 5. 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.
# ----------------------------------------------------------------------

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, p2, p3):
    assert distancia(p1, p3) <= distancia(p1, p2) + distancia(p2, p3)

# La comprobación está al final de la relación.

# 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
```

```
# ---------------------------------------------------------------------
# Ejercicio 6. 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.
# ---------------------------------------------------------------------

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):
    assert len(ciclo(xs)) == len(xs)

# La comprobación está al final de la relación.

# ---------------------------------------------------------------------
# Ejercicio 7. 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)
```

```python
# 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 está al final de la relación.

# ---------------------------------------------------------------------
# Ejercicio 8. 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

# ---------------------------------------------------------------------
# Ejercicio 9. 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.
# ----------------------------------------------------------------------

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, b, c):
    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 está al final de la relación.

# ----------------------------------------------------------------------
# Ejercicio 10. 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
# ----------------------------------------------------------------------

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

# ----------------------------------------------------------------------
# Ejercicio 11. 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.
# ----------------------------------------------------------------------

Rectangulo = list[float]

def interseccion(i1: Rectangulo,
                 i2: Rectangulo) -> Rectangulo:
```

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

# La propiedad es
@given(st.floats(), st.floats(), st.floats(), st.floats())
def test_prop_raices2(a1, b1, a2, b2):
    assume(a1 <= b1 and a2 <= b2)
    assert interseccion([a1, b1], [a2, b2]) == interseccion([a2, b2], [a1, b1])

# La comprobación está al final de la relación.

# ---------------------------------------------------------------------
# Ejercicio 12.1. 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 la función
#    formaReducida    : (Racional) -> Racional
# tal que formaReducida(x) es la forma reducida del número racional
# x. Por ejemplo,
#    formaReducida((4, 10))  ==  (2, 5)
#    formaReducida((0, 5))   ==  (0, 1)
# ---------------------------------------------------------------------

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)

# ---------------------------------------------------------------------
# Ejercicio 12.2. Definir la función
#    sumaRacional     : (Racional, Racional) -> Racional
# tal que 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)
# ---------------------------------------------------------------------

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

# ---------------------------------------------------------------------
# Ejercicio 12.3. Definir la función
#    productoRacional : (Racional, Racional) -> Racional
# tal que 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)
# ---------------------------------------------------------------------

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

# ---------------------------------------------------------------------
# Ejercicio 12.4. Definir la función
#    igualdadRacional : (Racional, Racional) -> bool
# tal que 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
```

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

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

# ---------------------------------------------------------------------
# Ejercicio 12.5. Comprobar con Hypothesis la propiedad distributiva del
# producto racional respecto de la suma.
# ---------------------------------------------------------------------

# 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, y, z):
    (_, 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 está al final de la relación

# ---------------------------------------------------------------------
# Comprobación de propiedades.
# ---------------------------------------------------------------------

# La comprobación de las propiedades es
#    src> poetry run pytest -q condicionales_guardas_y_patrones.py
#    9 passed in 1.85s
```

Capítulo 2

Definiciones por comprensión

2.1. Definiciones por comprensión

```
# ---------------------------------------------------------------------
# Introducción                                                       --
# ---------------------------------------------------------------------

# En esta relación se presentan ejercicios con definiciones por
# comprensión correspondientes al tema 5 que se encuentra en
#    https://jaalonso.github.io/cursos/i1m/temas/tema-5.html

# ---------------------------------------------------------------------
# Librerías auxiliares                                              --
# ---------------------------------------------------------------------

from itertools import islice
from math import ceil, e, pi, sin, sqrt, trunc
from sys import setrecursionlimit
from timeit import Timer, default_timer
from typing import Iterator, TypeVar

from hypothesis import given
from hypothesis import strategies as st

A = TypeVar('A')
setrecursionlimit(10**6)

# ---------------------------------------------------------------------
# Ejercicio 1.1. (Problema 6 del proyecto Euler) En los distintos
```

```python
# apartados de este ejercicio se definen funciones para resolver el
# problema 6 del proyecto Euler https://www.projecteuler.net/problem=6
#
# Definir, por comprensión, 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
# ---------------------------------------------------------------------

# 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

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

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

# La comprobación se hace al final.

# 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('suma1(10**8)')
#    1.55 segundos
#    >>> tiempo('suma2(10**8)')
#    0.00 segundos

# ---------------------------------------------------------------------
# Ejercicio 1.2. Definir, por comprensión, 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(sumaDeCuadrados2(10**100)))  ==  300
# ---------------------------------------------------------------------

# 1ª solución
# ===========

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

# 2ª solución
# ===========

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

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

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

# La comprobación está al final.

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

```
# La comparación es
#    >>> tiempo('sumaDeCuadrados1(10**7)')
#    2.19 segundos
#    >>> tiempo('sumaDeCuadrados2(10**7)')
#    0.00 segundos

# ---------------------------------------------------------------------
# Ejercicio 1.3. 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)  ==  250000000016666666664166666665000000000
# ---------------------------------------------------------------------

# 1ª solución
# ===========

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

# 2ª solución
# ===========

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

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

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

# La comprobación está al final

# Comparación de eficiencia
```

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

# La comparación es
#    >>> tiempo('euler6a(10**7)')
#    2.26 segundos
#    >>> tiempo('euler6b(10**7)')
#    0.00 segundos

# ---------------------------------------------------------------------
# Ejercicio 2. Definir, por comprensión, la función
#    replica : (int, A) -> list[A]
# tal que replica(n, x) es la lista formada por n copias del elemento
# x. Por ejemplo,
#    replica(4, 7)     ==  [7,7,7,7]
#    replica(3, True)  ==  [True, True, True]
# ---------------------------------------------------------------------

def replica(n: int, x: A) -> list[A]:
    return [x for _ in range(0, n)]

# ---------------------------------------------------------------------
# Ejercicio 3.1. 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 la función
#    linea : (int) -> list[int]
# tal 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
# ---------------------------------------------------------------------

# 1ª definición
# =============
```

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

# 2ª definición
# =============

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

# 3ª definición
# =============

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

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

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

# La comprobación está al final

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

# La comparación es
#    >>> tiempo('linea1(10**7)')
#    0.53 segundos
#    >>> tiempo('linea2(10**7)')
#    0.40 segundos
#    >>> tiempo('linea3(10**7)')
#    0.29 segundos
```

```python
# ---------------------------------------------------------------------
# Ejercicio 3.2. Definir la función
#    triangulo : (int) -> list[list[int]]
# tale que 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]]
# ---------------------------------------------------------------------

# 1ª definición
# =============

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

# 2ª definición
# =============

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

# 3ª definición
# =============

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

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

@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 está al final.

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

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

# ---------------------------------------------------------------------
# Ejercicio 4. Un números entero positivo es perfecto 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, por comprensión, 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]
# ---------------------------------------------------------------------

# divisores(n) es la lista de los divisores del número 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]

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

# esPerfecto(x) se verifica si x es un número perfecto. Por ejemplo,
#    esPerfecto(6)  ==  True
#    esPerfecto(8)  ==  False
def esPerfecto(x: int) -> bool:
    return sumaDivisores(x) - x == x
```

```python
def perfectos(n: int) -> list[int]:
    return [x for x in range(1, n + 1) if esPerfecto(x)]

# ---------------------------------------------------------------------
# Ejercicio 5.1. Un número natural n se denomina abundante 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
# ---------------------------------------------------------------------

def numeroAbundante(x: int) -> bool:
    return x < sumaDivisores(x) - x

# ---------------------------------------------------------------------
# Ejercicio 5.2. 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
# ---------------------------------------------------------------------

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

# ---------------------------------------------------------------------
# Ejercicio 5.3. 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
# ----------------------------------------------------------------------

def todosPares(n: int) -> bool:
    return False not in [x % 2 == 0 for x in numerosAbundantesMenores(n)]

# ----------------------------------------------------------------------
# Ejercicio 6. 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
# ----------------------------------------------------------------------

# 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 euler1(n: int) -> int:
    return sum(x for x in range(1, n)
               if (multiplo(x, 3) or multiplo(x, 5)))

# El cálculo es
#    >>> euler1(1000)
#    233168
```

```
# ----------------------------------------------------------------------
# Ejercicio 7. 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
# ----------------------------------------------------------------------

# 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)))
```

```python
# 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 está al final.

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

# 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

# ---------------------------------------------------------------------
# Ejercicio 8.1. El número e se define como el límite de la sucesión
# (1+1/n)**n; es decir,
#    e = lim (1+1/n)**n
#
# Definir la función
#    aproxE      : (int) -> list[float]
# tal 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
# ---------------------------------------------------------------------

def aproxE(k: int) -> list[float]:
    return [(1 + 1/n)**n for n in range(1, k + 1)]

# ---------------------------------------------------------------------
# Ejercicio 8.2. Definir la función
#    errorAproxE : (float) -> int
# tal que 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
# ---------------------------------------------------------------------

# naturales es el generador de los números naturales positivos, Por
# ejemplo,
#    >>> list(islice(naturales(), 5))
#    [1, 2, 3, 4, 5]
```

```python
def naturales() -> Iterator[int]:
    i = 1
    while True:
        yield i
        i += 1

def errorAproxE(x: float) -> int:
    return list(islice((n for n in naturales()
                        if abs(e - (1 + 1/n)**n) < x), 1))[0]

# ---------------------------------------------------------------------
# Ejercicio 9.1. 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 la función
#    aproxLimSeno : (int) -> list[float]
# tal 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]
# ---------------------------------------------------------------------

def aproxLimSeno(k: int) -> list[float]:
    return [sin(1/n)/(1/n) for n in range(1, k + 1)]

# ---------------------------------------------------------------------
# Ejercicio 9.2. Definir la función
#    errorLimSeno : (float) -> int
# tal que 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
# ---------------------------------------------------------------------

# 1ª definición de errorLimSeno
# =============================
```

```
def errorLimSeno(x: float) -> int:
    return list(islice((n for n in naturales()
                        if abs(1 - sin(1/n)/(1/n)) < x), 1))[0]

# ----------------------------------------------------------------------
# Ejercicio 10.1. El número π puede calcularse con la fórmula de
# Leibniz
#    π/4 = 1 - 1/3 + 1/5 - 1/7 + ...+ (-1)**n/(2*n+1) + ...
#
# Definir la función
#    calculaPi : (int) -> float
# tal 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
# ----------------------------------------------------------------------

def calculaPi(k: int) -> float:
    return 4 * sum(((-1)**n/(2*n+1) for n in range(0, k+1)))

# ----------------------------------------------------------------------
# Ejercicio 10.2. Definir la función
#    errorPi   : (float) -> int
# tal que 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
# ----------------------------------------------------------------------

def errorPi(x: float) -> int:
    return list(islice((n for n in naturales()
                        if abs(pi - calculaPi(n)) < x), 1))[0]

# ----------------------------------------------------------------------
# Ejercicio 11.1. Una terna (x,y,z) de enteros positivos es pitagórica
# si x^2 + y^2 = z^2 y x < y < z.
```

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

# 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 está al final.

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

# La comparación es
#    >>> tiempo('pitagoricas1(200)')
#    4.76 segundos
#    >>> tiempo('pitagoricas2(200)')
#    0.69 segundos
#    >>> tiempo('pitagoricas3(200)')
#    0.02 segundos

# ---------------------------------------------------------------------
# Ejercicio 11.2. Definir la función
#    numeroDePares : (int, int, int) -> int
# tal que numeroDePares(t) es el número de elementos pares de la terna
# t. Por ejemplo,
#    numeroDePares(3, 5, 7)  ==  0
#    numeroDePares(3, 6, 7)  ==  1
#    numeroDePares(3, 6, 4)  ==  2
#    numeroDePares(4, 6, 4)  ==  3
# ---------------------------------------------------------------------

def numeroDePares(x: int, y: int, z: int) -> int:
    return len([1 for n in [x, y, z] if n % 2 == 0])

# ---------------------------------------------------------------------
# Ejercicio 11.3. Definir la función
#    conjetura : (int) -> bool
# tal que conjetura(n) se verifica si todas las ternas pitagóricas
```

```
# cuyas componentes están entre 1 y n tiene un número impar de números
# pares. Por ejemplo,
#    conjetura(10)  ==  True
# ----------------------------------------------------------------------

def conjetura(n: int) -> bool:
    return False not in [numeroDePares(x, y, z) % 2 == 1
                         for (x, y, z) in pitagoricas1(n)]

# ----------------------------------------------------------------------
# Ejercicio 11.4. Demostrar la conjetura para todas las ternas
# pitagóricas.
# ----------------------------------------------------------------------
#
# Sea (x,y,z) una terna pitagórica. Entonces x^2+y^2=z^2. Pueden darse
# 4 casos:
#
# Caso 1: x e y son pares. Entonces, x^2, y^2 y z^2 también lo
# son. Luego el número de componentes pares es 3 que es impar.
#
# Caso 2: x es par e y es impar. Entonces, x^2 es par, y^2 es impar y
# z^2 es impar. Luego el número de componentes pares es 1 que es impar.
#
# Caso 3: x es impar e y es par. Análogo al caso 2.
#
# Caso 4: x e y son impares. Entonces, x^2 e y^2 también son impares y
# z^2 es par. Luego el número de componentes pares es 1 que es impar.

# ----------------------------------------------------------------------
# Ejercicio 12.1. (Problema 9 del proyecto Euler). 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)]
# ------------------------------------------------------------------------

# 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 está al final.

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

# 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

# ---------------------------------------------------------------------
# Ejercicio 12.2. Definir la función
#    euler9 : () -> int
# tal que euler9() es producto abc donde (a,b,c) es la única terna
# pitagórica tal que a+b+c=1000.
#
```

```
# Calcular el valor de euler9().
# ------------------------------------------------------------------

def euler9() -> int:
    (a, b, c) = ternasPitagoricas3(1000)[0]
    return a * b * c

# El cálculo del valor de euler9 es
#    >>> euler9()
#    31875000

# ------------------------------------------------------------------
# Ejercicio 13. 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, por comprensión, 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
# ------------------------------------------------------------------

def productoEscalar(xs: list[int], ys: list[int]) -> int:
    return sum(x * y for (x, y) in zip(xs, ys))

# ------------------------------------------------------------------
# Ejercicio 14. Definir , por comprensión,la función
#    sumaConsecutivos : (list[int]) -> list[int]
# tal que sumaConsecutivos(xs) es la suma de los pares de elementos
# consecutivos de la lista xs. Por ejemplo,
#    sumaConsecutivos([3, 1, 5, 2])           ==  [4, 6, 7]
#    sumaConsecutivos([3])                    ==  []
#    sumaConsecutivos(range(1, 1+10**8))[-1]  ==  199999999
# ------------------------------------------------------------------

def sumaConsecutivos(xs: list[int]) -> list[int]:
    return [x + y for (x, y) in zip(xs, xs[1:])]

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

```python
# Ejercicio 15. 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)]
# ----------------------------------------------------------------------

def densa(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])]

# ----------------------------------------------------------------------
# Ejercicio 16. 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)]
# ---------------------------------------------------------------------

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)]

# ---------------------------------------------------------------------
# Ejercicio 16.1. Definir la función
#    nombres : (BD) -> list[str]
# tal 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']
# ---------------------------------------------------------------------

def nombres(bd: BD) -> list[str]:
    return [p[0] for p in bd]

# ---------------------------------------------------------------------
# Ejercicio 16.2. Definir la función
#    musicos : (BD) -> list[str]
# tal que musicos(bd) es la lista de los nombres de los músicos de la
# base de datos bd. Por ejemplo,
#    musicos(personas)  ==  ['Beethoven', 'Mozart', 'Bach']
# ---------------------------------------------------------------------
```

```python
def musicos(bd: BD) -> list[str]:
    return [p[0] for p in bd if p[1] == "Musica"]

# ---------------------------------------------------------------------
# Ejercicio 16.3. Definir la función
#    seleccion : (BD, str) -> list[str]
# tal que 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']
# ---------------------------------------------------------------------

def seleccion(bd: BD, m: str) -> list[str]:
    return [p[0] for p in bd if p[1] == m]

# ---------------------------------------------------------------------
# Ejercicio 16.4. Definir la función
#    musicos2 : (BD) -> list[str]
# tal que musicos2(bd) es la lista de los nombres de los músicos de la
# base de datos bd. Por ejemplo,
#    musicos2(personas)  ==  ['Beethoven','Mozart','Bach']
# ---------------------------------------------------------------------

def musicos2(bd: BD) -> list[str]:
    return seleccion(bd, "Musica")

# ---------------------------------------------------------------------
# Ejercicio 16.5. Definir la función
#    vivas : (BD, int) -> list[str]
# tal que 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']
# ---------------------------------------------------------------------

def vivas(bd: BD, a: int) -> list[str]:
    return [p[0] for p in bd if p[2] <= a <= p[3]]
```

```
# ----------------------------------------------------------------------
# Comprobación
# ----------------------------------------------------------------------

# La comprobación es
#    src> poetry run pytest -q definiciones_por_comprension.py
#    8 passed in 4.23s
```

Capítulo 3

Definiciones por recursión

3.1. Definiciones por recursión

```
# ----------------------------------------------------------------------
# Introducción                                                         --
# ----------------------------------------------------------------------

# En esta relación se presentan ejercicios con definiciones por
# recursión correspondientes al tema 6 que se encuentra en
#    https://jaalonso.github.io/cursos/i1m/temas/tema-6.html

# ----------------------------------------------------------------------
# Importación de librerías auxiliares                                  --
# ----------------------------------------------------------------------

from itertools import islice
from sys import setrecursionlimit
from timeit import Timer, default_timer
from typing import Iterator, TypeVar

from hypothesis import given
from hypothesis import strategies as st

setrecursionlimit(10**6)

A = TypeVar('A')

# ----------------------------------------------------------------------
# Ejercio 1. Definir, por recursión, la función
```

```
#    potencia : (int, int) -> int
# tal que potencia(x, n) es x elevado al número natural n. Por ejemplo,
#    potencia(2, 3)  ==  8
# ---------------------------------------------------------------------

def potencia(m: int, n: int) -> int:
    if n == 0:
        return 1
    return m * potencia(m, n-1)

# ---------------------------------------------------------------------
# Ejercicio 1.2. Comprobar con Hypothesis que la función potencia es
# equivalente a la predefinida (^).
# ---------------------------------------------------------------------

# 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:
    assert potencia(m, n) == m ** n

# La comprobación está al final.

# ---------------------------------------------------------------------
# Ejercicio 2. 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.
# ----------------------------------------------------------------------

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

# ----------------------------------------------------------------------
# Ejercicio 3.1, Definir por recursión la función
#    pertenece : (A, list[A]) -> bool
# tal que pertenece(x, ys) se verifica si x pertenece a la lista ys.
# Por ejemplo,
#    pertenece(3, [2, 3, 5])  ==  True
#    pertenece(4, [2, 3, 5])  ==  False
# ----------------------------------------------------------------------

def pertenece(x: A, ys: list[A]) -> bool:
    if ys:
        return x == ys[0] or pertenece(x, ys[1:])
    return False

# ----------------------------------------------------------------------
# Ejercicio 3.2. Comprobar con Hypothesis que pertenece es equivalente
# a in.
# ----------------------------------------------------------------------

# La propiedad es
```

```python
@given(st.integers(),
       st.lists(st.integers()))
def test_pertenece(x: int, ys: list[int]) -> None:
    assert pertenece(x, ys) == (x in ys)

# La comprobación está al final.

# ---------------------------------------------------------------------
# Ejercicio 4. Definir por recursión la función
#    concatenaListas :: [[a]] -> [a]
# tal que (concatenaListas xss) es la lista obtenida concatenando las
# listas de xss. Por ejemplo,
#    concatenaListas([[1, 3], [5], [2, 4, 6]]) == [1, 3, 5, 2, 4, 6]
# ---------------------------------------------------------------------

def concatenaListas(xss: list[list[A]]) -> list[A]:
    if xss:
        return xss[0] + concatenaListas(xss[1:])
    return []

# ---------------------------------------------------------------------
# Ejercicio 5.1. Definir por recursión la función
#    coge : (int, list[A]) -> list[A]
# tal que coge(n, xs) es la lista de los n primeros elementos de
# xs. Por ejemplo,
#    coge(3, range(4, 12)) == [4, 5, 6]
# ---------------------------------------------------------------------

def coge(n: int, xs: list[A]) -> list[A]:
    if n <= 0:
        return []
    if not xs:
        return []
    return [xs[0]] + coge(n - 1, xs[1:])

# ---------------------------------------------------------------------
# Ejercicio 5.2. Comprobar con Hypothesis que coge(n, xs) es equivalente
# a xs[:n], suponiendo que n >= 0.
# ---------------------------------------------------------------------
```

```python
# La propiedad es
@given(st.integers(min_value=0),
       st.lists(st.integers()))
def test_coge(n: int, xs: list[int]) -> None:
    assert coge(n, xs) == xs[:n]

# La comprobación está al final.

# ---------------------------------------------------------------------
# Ejercicio 6.1. Definir, por recursión  la función
#    sumaDeCuadradosR : (int) -> int
# tal sumaDeCuadradosR(n) es la suma de los cuadrados de los n primeros
# números naturales. Por ejemplo,
#    sumaDeCuadradosR(3)   ==  14
#    sumaDeCuadradosR(100) ==  338350
# ---------------------------------------------------------------------

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

# ---------------------------------------------------------------------
# Ejercicio 6.2. Comprobar con Hypothesis que sumaCuadradosR(n) es igual
# a n(n+1)(2n+1)/6.
# ---------------------------------------------------------------------

# La propiedad es
@given(st.integers(min_value=1, max_value=1000))
def test_sumaDeCuadrados(n: int) -> None:
    assert sumaDeCuadradosR(n) == n * (n + 1) * (2 * n + 1) // 6

# La comprobación está al final.

# ---------------------------------------------------------------------
# Ejercicio 6.3. Definir, por comprensión, la función
#    sumaDeCuadradosC : (int) -> int
# tal sumaDeCuadradosC(n) es la suma de los cuadrados de los n primeros
# números naturales. Por ejemplo,
#    sumaDeCuadradosC(3)   ==  14
```

```
#    sumaDeCuadradosC(100) ==  338350
# ---------------------------------------------------------------------

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

# ---------------------------------------------------------------------
# Ejercicio 6.4. Comprobar con Hypothesis que las funciones
# sumaCuadradosR y sumaCuadradosC son equivalentes sobre los números
# naturales.
# ---------------------------------------------------------------------

@given(st.integers(min_value=1, max_value=1000))
def test_sumaDeCuadrados2(n: int) -> None:
    assert sumaDeCuadradosR(n) == sumaDeCuadradosC(n)

# La comprobación está al final.

# ---------------------------------------------------------------------
# Ejercicio 7.1. Definir, por recursión, la función
#    digitosR : (int) -> list[int]
# tal que digitosR(n) es la lista de los dígitos del número n. Por
# ejemplo,
#    digitosR(320274)  ==  [3, 2, 0, 2, 7, 4]
# ---------------------------------------------------------------------

def digitosR(n: int) -> list[int]:
    if n < 10:
        return [n]
    return digitosR(n // 10) + [n % 10]

# ---------------------------------------------------------------------
# Ejercicio 7.2. Definir, por comprensión, la función
#    digitosC : (int) -> list[int]
# tal que digitosC(n) es la lista de los dígitos del número n. Por
# ejemplo,
#    digitosC(320274)  ==  [3, 2, 0, 2, 7, 4]
# ---------------------------------------------------------------------

def digitosC(n: int) -> list[int]:
```

```
    return [int(x) for x in str(n)]

# ---------------------------------------------------------------------
# Ejercicio 7.3. Comprobar con Hypothesis que las funciones digitosR y
# digitosC son equivalentes.
# ---------------------------------------------------------------------

# La propiedad es
@given(st.integers(min_value=1, max_value=1000))
def test_digitos(n: int) -> None:
    assert digitosR(n) == digitosC(n)

# La comprobación está al final.

# ---------------------------------------------------------------------
# Ejercicio 8.1. Definir, por recursión, la función
#    sumaDigitosR : (int) -> int
# tal que sumaDigitosR(n) es la suma de los dígitos de n. Por ejemplo,
#    sumaDigitosR(3)     ==  3
#    sumaDigitosR(2454)  == 15
#    sumaDigitosR(20045) == 11
# ---------------------------------------------------------------------

def sumaDigitosR(n: int) -> int:
    if n < 10:
        return n
    return n % 10 + sumaDigitosR(n // 10)

# ---------------------------------------------------------------------
# Ejercicio 8.2. Definir, sin usar recursión, la función
#    sumaDigitosNR : (int) -> int
# tal que sumaDigitosNR(n) es la suma de los dígitos de n. Por ejemplo,
#    sumaDigitosNR(3)     ==  3
#    sumaDigitosNR(2454)  == 15
#    sumaDigitosNR(20045) == 11
# ---------------------------------------------------------------------

def sumaDigitosNR(n: int) -> int:
    return sum(digitosC(n))
```

```python
# ---------------------------------------------------------------------
# Ejercicio 8.3. Comprobar con Hypothesis que las funciones sumaDigitosR
# y sumaDigitosNR son equivalentes.
# ---------------------------------------------------------------------

# La propiedad es
@given(st.integers(min_value=1, max_value=1000))
def test_sumaDigitos(n: int) -> None:
    assert sumaDigitosR(n) == sumaDigitosNR(n)

# La comprobación está al final.

# ---------------------------------------------------------------------
# Ejercicio 9.1. Definir, por recursión, la función
#    listaNumeroR : (list[int]) -> int
# tal que listaNumeroR(xs) es el número formado por los dígitos xs. Por
# ejemplo,
#    listaNumeroR([5])           == 5
#    listaNumeroR([1, 3, 4, 7])  == 1347
#    listaNumeroR([0, 0, 1])     == 1
# ---------------------------------------------------------------------

def listaNumeroR(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)))

# ---------------------------------------------------------------------
# Ejercicio 9.2. Definir, por comprensión, la función
#    listaNumeroC : (list[int]) -> int
# tal que listaNumeroC(xs) es el número formado por los dígitos xs. Por
# ejemplo,
#    listaNumeroC([5])           == 5
#    listaNumeroC([1, 3, 4, 7])  == 1347
#    listaNumeroC([0, 0, 1])     == 1
# ---------------------------------------------------------------------

def listaNumeroC(xs: list[int]) -> int:
```

```
    return sum((y * 10**n
                for (y, n) in zip(list(reversed(xs)), range(0, len(xs)))))

# ---------------------------------------------------------------------
# Ejercicio 9.3. Comprobar con Hypothesis que las funciones
# listaNumeroR y listaNumeroC son equivalentes.
# ---------------------------------------------------------------------

# La propiedad es
@given(st.lists(st.integers(min_value=0, max_value=9), min_size=1))
def test_listaNumero(xs: list[int]) -> None:
    print("listaNumero")
    assert listaNumeroR(xs) == listaNumeroC(xs)

# La comprobación está al final.

# ---------------------------------------------------------------------
# Ejercicio 10.1. Definir, por recursión, la función
#    mayorExponenteR : (int, int) -> int
# tal que mayorExponenteR(a, b) es el exponente de la mayor potencia de
# a que divide b. Por ejemplo,
#    mayorExponenteR(2, 8)    ==  3
#    mayorExponenteR(2, 9)    ==  0
#    mayorExponenteR(5, 100)  ==  2
#    mayorExponenteR(2, 60)   ==  2
#
# Nota: Se supone que a > 1 y b > 0.
# ---------------------------------------------------------------------

def mayorExponenteR(a: int, b: int) -> int:
    if b % a != 0:
        return 0
    return 1 + mayorExponenteR(a, b // a)

# ---------------------------------------------------------------------
# Ejercicio 10.2. Definir, por comprensión, la función
#    mayorExponenteC : (int, int) -> int
# tal que mayorExponenteC(a, b) es el exponente de la mayor potencia de
# a que divide b. Por ejemplo,
#    mayorExponenteC(2, 8)    ==  3
```

```
#    mayorExponenteC(2, 9)    ==  0
#    mayorExponenteC(5, 100)  ==  2
#    mayorExponenteC(2, 60)   ==  2
#
# Nota: Se supone que a > 1 y b > 0.
# ----------------------------------------------------------------------

# 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 mayorExponenteC(a: int, b: int) -> int:
    return list(islice((x - 1 for x in naturales() if b % (a**x) != 0), 1))[0]

# ----------------------------------------------------------------------
# Ejercicio 10.3. Comprobar con Hypothesis que las funciones
# mayorExponenteR y mayorExponenteC son equivalentes.
# ----------------------------------------------------------------------

# 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:
    assert mayorExponenteR(a, b) == mayorExponenteC(a, b)

# La comprobación está al final.

# La comprobación de las propiedades es
#    src> poetry run pytest -q definiciones_por_recursion.py
#    10 passed in 0.98s
```

3.2. Operaciones conjuntistas con listas

```
# ----------------------------------------------------------------------
# Introducción                                                       --
```

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

# En esta relación se definen operaciones conjuntistas sobre listas.

# ---------------------------------------------------------------------
# Librerías auxiliares                                               --
# ---------------------------------------------------------------------

from itertools import combinations
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
from sympy import FiniteSet

setrecursionlimit(10**6)

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

# ---------------------------------------------------------------------
# Ejercicio 1. 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
# ---------------------------------------------------------------------

# 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)

# 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

# ---------------------------------------------------------------------
# Ejercicio 2. 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
# ---------------------------------------------------------------------

# 1ª solución
# ===========

def iguales1(xs: list[Any],
             ys: list[Any]) -> bool:
    return subconjunto1(xs, ys) and subconjunto1(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)

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

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

# ---------------------------------------------------------------------
# Ejercicio 3.1. 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]
# ---------------------------------------------------------------------

# 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))

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

# 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

# ----------------------------------------------------------------------
# Nota. En los ejercicios de comprobación de propiedades, cuando se
# trata con igualdades se usa la igualdad conjuntista (definida por la
# función iguales) en lugar de la igualdad de lista (definida por ==)
# ----------------------------------------------------------------------
```

```python
# ---------------------------------------------------------------------
# Ejercicio 3.2. Comprobar con Hypothesis que la unión es conmutativa.
# ---------------------------------------------------------------------

# 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 iguales1(union1(xs1, ys1), union1(ys1, xs1))

# ---------------------------------------------------------------------
# Ejercicio 4.1. 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]) == []
# ---------------------------------------------------------------------

# 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))

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

# 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
```

```
# ---------------------------------------------------------------------
# Ejercicio 4.2. Comprobar con Hypothesis si se cumple la siguiente
# propiedad
#    A ∪ (B ∩ C) = (A ∪ B) ∩ C
# donde se considera la igualdad como conjuntos. En el caso de que no
# se cumpla verificar el contraejemplo calculado por Hypothesis.
# ---------------------------------------------------------------------

# La propiedad es
# @given(st.lists(st.integers()),
#        st.lists(st.integers()),
#        st.lists(st.integers()))
# def test_union_interseccion(xs: list[int],
#                             ys: list[int],
#                             zs: list[int]) -> None:
#     assert iguales1(union1(xs, interseccion1(ys, zs)),
#                     interseccion1(union1(xs, ys), zs))

# Al descomentar  la definición anterior y hacer la comprobación da el
# siguiente contraejemplo:
#    xs = [0], ys = [], zs = []
# ya que entonces,
#    xs ∪ (ys ∩ zs) = [0] ∪ ([] ∩ []) = [0] ∪ [] = [0]
#    (xs ∪ ys) ∩ zs = ([0] ∪ []) ∩ [] = [0] ∩ [] = []

# ---------------------------------------------------------------------
# Ejercicio 5.1. 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)]
# ---------------------------------------------------------------------

# 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]
```

```python
# 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))

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

# 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

# ---------------------------------------------------------------
# Ejercicio 5.2. Comprobar con Hypothesis que el número de
# elementos de (producto xs ys) es el producto del número de
# elementos de xs y de ys.
# ---------------------------------------------------------------

# 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)

# ---------------------------------------------------------------------
# Ejercicio 6.1. 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], []]
# ----------------------------------------------------------------------

# 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

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

# 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

# ---------------------------------------------------------------------
# Ejercicio 6.2. Comprobar con Hypothesis que el número de elementos de
# (subconjuntos xs) es 2 elevado al número de elementos de xs.
# ---------------------------------------------------------------------
```

```python
# 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)

# ---------------------------------------------------------------------
# Comprobación de las propiedades
# ---------------------------------------------------------------------

# La comprobación de las propiedades es
#    src> poetry run pytest -q operaciones_conjuntistas_con_listas.py
#    9 passed in 2.53s
```

3.3. El algoritmo de Luhn

```python
# ---------------------------------------------------------------------
# § Introducción                                                    --
# ---------------------------------------------------------------------

# El objetivo de esta relación 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 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, los llamaremos números de Luhn.

# ---------------------------------------------------------------------
# Ejercicio 1. Definir la función
```

```
#    digitosInv : : (int) -> list[int]
# tal 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]
# ----------------------------------------------------------------------

def digitosInv(n: int) -> list[int]:
    return [int(x) for x in reversed(str(n))]

# ----------------------------------------------------------------------
# Ejercicio 2. Definir la función
#    doblePosImpar : (list[int]) -> list[int]
# tal que 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]
# ----------------------------------------------------------------------

# 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)]

# ----------------------------------------------------------------------
# Ejercicio 3. Definir la función
#    sumaDigitos : (list[int]) -> int
# tal que sumaDigitos(ns) es la suma de los dígitos de ns. Por ejemplo,
#      sumaDigitos([10,5,18,4]) = 1 + 0 + 5 + 1 + 8 + 4 =
#                               = 19
# ----------------------------------------------------------------------
```

```python
def sumaDigitos(ns: list[int]) -> int:
    return sum((sum(digitosInv(n)) for n in ns))

# ---------------------------------------------------------------------
# Ejercicio 4. Definir la función
#    ultimoDigito : (int) -> int
# tal que ultimoDigito(n) es el último dígito de n. Por ejemplo,
#      ultimoDigito(123) == 3
#      ultimoDigito(0)   == 0
# ---------------------------------------------------------------------

def ultimoDigito(n: int) -> int:
    return n % 10

# ---------------------------------------------------------------------
# Ejercicio 5. Definir la función
#    luhn :: (int) -> bool
# tal que luhn(n) se verifica si n es un número de Luhn. Por ejemplo,
#      luhn(5594589764218858)  ==  True
#      luhn(1234567898765432)  ==  False
# ---------------------------------------------------------------------

def luhn(n: int) -> bool:
    return ultimoDigito(sumaDigitos(doblePosImpar(digitosInv(n)))) == 0

# ---------------------------------------------------------------------
# § Referencias                                                     --
# ---------------------------------------------------------------------

# Esta relación es una adaptación del primer trabajo del curso "CIS 194:
# Introduction to Haskell (Spring 2015)" de la Univ. de Pensilvania,
# impartido por Noam Zilberstein. El trabajo se encuentra en
# http://www.cis.upenn.edu/~cis194/hw/01-intro.pdf
#
# En el artículo [Algoritmo de Luhn](http://bit.ly/1FGGWsC) de la
# Wikipedia se encuentra información del algoritmo
```

3.4. Números de Lychrel

```
# ---------------------------------------------------------------------
# Introducción                                                       --
# ---------------------------------------------------------------------

# Según la Wikipedia http://bit.ly/2X4DzMf, un número de Lychrel 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 relación vamos a buscar el primer número de Lychrel.

# ---------------------------------------------------------------------
# Librerías auxiliares                                               --
# ---------------------------------------------------------------------

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]
# ---------------------------------------------------------------------

# 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
```

3.5. Funciones sobre cadenas

```
# ----------------------------------------------------------------------
# Importación de librerías auxiliares                                --
# ----------------------------------------------------------------------
```

```python
from sys import setrecursionlimit

from hypothesis import given
from hypothesis import strategies as st

setrecursionlimit(10**6)

# ---------------------------------------------------------------------
# Ejercicio 1.1. Definir, por comprensión, la función
#    sumaDigitosC : (str) -> int
# tal que sumaDigitosC(xs) es la suma de los dígitos de la cadena
# xs. Por ejemplo,
#    sumaDigitosC("SE 2431 X")  ==  10
# ---------------------------------------------------------------------

def sumaDigitosC(xs: str) -> int:
    return sum((int(x) for x in xs if x.isdigit()))

# ---------------------------------------------------------------------
# Ejercicio 1.2. Definir, por recursión, la función
#    sumaDigitosR : (str) -> int
# tal que sumaDigitosR(xs) es la suma de los dígitos de la cadena
# xs. Por ejemplo,
#    sumaDigitosR("SE 2431 X")  ==  10
# ---------------------------------------------------------------------

def sumaDigitosR(xs: str) -> int:
    if xs:
        if xs[0].isdigit():
            return int(xs[0]) + sumaDigitosR(xs[1:])
        return sumaDigitosR(xs[1:])
    return 0

# ---------------------------------------------------------------------
# Ejercicio 1.3. Definir, por iteración, la función
#    sumaDigitosI : (str) -> int
# tal que sumaDigitosI(xs) es la suma de los dígitos de la cadena
# xs. Por ejemplo,
#    sumaDigitosI("SE 2431 X")  ==  10
```

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

def sumaDigitosI(xs: str) -> int:
    r = 0
    for x in xs:
        if x.isdigit():
            r = r + int(x)
    return r

# ---------------------------------------------------------------------
# Ejercicio 1.4. Comprobar con QuickCheck que las tres definiciones son
# equivalentes.
# ---------------------------------------------------------------------

# La propiedad es
@given(st.text(alphabet=st.characters(min_codepoint=32, max_codepoint=127)))
def test_sumaDigitos(xs: str) -> None:
    r = sumaDigitosC(xs)
    assert sumaDigitosR(xs) == r
    assert sumaDigitosI(xs) == r

# ---------------------------------------------------------------------
# Ejercicio 2.1. Definir, por comprensión, 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("")         ==  ""
# ---------------------------------------------------------------------

def mayusculaInicial(xs: str) -> str:
    if xs:
        return "".join([xs[0].upper()] + [y.lower() for y in xs[1:]])
    return ""

# ---------------------------------------------------------------------
# Ejercicio 2.2. Definir, por recursión, la función
#    mayusculaInicialRec : (str) -> str
# tal que mayusculaInicialRec(xs) es la palabra xs con la letra inicial
# en mayúscula y las restantes en minúsculas. Por ejemplo,
```

```
#    mayusculaInicialRec("sEviLLa")  ==  "Sevilla"
#    mayusculaInicialRec("")         ==  ""
# ----------------------------------------------------------------------

def mayusculaInicialRec(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 ""

# ----------------------------------------------------------------------
# Ejercicio 2.3. Comprobar con Hypothesis que ambas definiciones son
# equivalentes.
# ----------------------------------------------------------------------

# La propiedad es
@given(st.text())
def test_mayusculaInicial(xs: str) -> None:
    assert mayusculaInicial(xs) == mayusculaInicialRec(xs)

# ----------------------------------------------------------------------
# Ejercicio 3.1. 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, por comprensión, 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"]
# ----------------------------------------------------------------------


# (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 titulo(ps: list[str]) -> list[str]:
    if ps:
        return [mayusculaInicial(ps[0])] + [transforma(q) for q in ps[1:]]
    return []

# ---------------------------------------------------------------------
# Ejercicio 3.2. Definir, por recursión, la función
#    tituloRec : (list[str]) -> list[str]
# tal que tituloRec(ps) es la lista de las palabras de ps con
# las reglas de mayúsculas iniciales de los títulos. Por ejemplo,
#    >>> tituloRec(["eL", "arTE", "DE", "La", "proGraMacion"])
#    ["El", "Arte", "de", "la", "Programacion"]
# ---------------------------------------------------------------------

def tituloRec(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 []

# ---------------------------------------------------------------------
# Ejercicio 3.3. Comprobar con Hypothesis que ambas definiciones son
# equivalentes.
# ---------------------------------------------------------------------

# La propiedad es
@given(st.lists(st.text()))
```

```
def test_titulo(ps: list[str]) -> None:
    assert titulo(ps) == tituloRec(ps)

# ----------------------------------------------------------------------
# Ejercicio 4.1. Definir, por comprensión, 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]
# ----------------------------------------------------------------------

def posiciones(x: str, ys: str) -> list[int]:
    return [n for (n, y) in enumerate(ys) if y == x]

# ----------------------------------------------------------------------
# Ejercicio 4.2. Definir, por recursión, la función
#    posicionesR : (str, str) -> list[int]
# tal que posicionesR(x, ys) es la lista de la posiciones del carácter x
# en la cadena ys. Por ejemplo,
#    posicionesR('a', "Salamamca")   ==  [1,3,5,8]
# ----------------------------------------------------------------------

def posicionesR(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)

# ----------------------------------------------------------------------
# Ejercicio 4.3. Definir, por iteración, la función
#    posicionesI : (str, str) -> list[int]
# tal que posicionesI(x ,ys) es la lista de la posiciones del carácter x
# en la cadena ys. Por ejemplo,
#    posicionesI('a', "Salamamca")   ==  [1,3,5,8]
# ----------------------------------------------------------------------

def posicionesI(x: str, ys: str) -> list[int]:
```

```
    r = []
    for n, y in enumerate(ys):
        if x == y:
            r.append(n)
    return r

# ----------------------------------------------------------------------
# Ejercicio 4.3. Comprobar con Hypothesis que las tres definiciones son
# equivalentes.
# ----------------------------------------------------------------------

# La propiedad es
@given(st.text(), st.text())
def test_posiciones(x: str, ys: str) -> None:
    r = posiciones(x, ys)
    assert posicionesR(x, ys) == r
    assert posicionesI(x, ys) == r

# ----------------------------------------------------------------------
# Ejercicio 5.1. Definir, por recursión, la función
#    esSubcadenaR : (str, str) -> bool
# tal que esSubcadenaR(xs ys) se verifica si xs es una subcadena de ys.
# Por ejemplo,
#    esSubcadenaR("casa", "escasamente")   ==  True
#    esSubcadenaR("cante", "escasamente")  ==  False
#    esSubcadenaR("", "")                  ==  True
# ----------------------------------------------------------------------

def esSubcadenaR(xs: str, ys: str) -> bool:
    if not xs:
        return True
    if not ys:
        return False
    return ys.startswith(xs) or esSubcadenaR(xs, ys[1:])

# ----------------------------------------------------------------------
# Ejercicio 5.2. Definir, por comprensión, 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
# ---------------------------------------------------------------------

# 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 esSubcadena(xs: str, ys: str) -> bool:
    return any(zs.startswith(xs) for zs in sufijos(ys))

# Se puede definir por
def esSubcadena3(xs: str, ys: str) -> bool:
    return xs in ys

# ---------------------------------------------------------------------
# Ejercicio 5.3. Comprobar con Hypothesis que las tres definiciones son
# equivalentes.
# ---------------------------------------------------------------------

# La propiedad es
@given(st.text(), st.text())
def test_esSubcadena(xs: str, ys: str) -> None:
    r = esSubcadenaR(xs, ys)
    assert esSubcadena(xs, ys) == r
    assert esSubcadena3(xs, ys) == r

# ---------------------------------------------------------------------
# Comprobación de las propiedades
# ---------------------------------------------------------------------

# La comprobación es
#    src> poetry run pytest -q funciones_sobre_cadenas.py
#    1 passed in 0.41s
```

Capítulo 4

Funciones de orden superior

4.1. Funciones de orden superior y definiciones por plegado

```
# ---------------------------------------------------------------------
# Introducción                                                       --
# ---------------------------------------------------------------------

# Esta relación contiene ejercicios con funciones de orden superior y
# definiciones por plegado correspondientes al tema 7 que se encuentra
# en
#    https://jaalonso.github.io/cursos/i1m/temas/tema-7.html

# ---------------------------------------------------------------------
# Importación de librerías auxiliares                                --
# ---------------------------------------------------------------------

from functools import reduce
from itertools import dropwhile, takewhile
from operator import concat
from sys import setrecursionlimit
from timeit import Timer, default_timer
from typing import Any, Callable, TypeVar, Union

from hypothesis import given
from hypothesis import strategies as st
from more_itertools import split_at
from numpy import array, transpose
```

```
setrecursionlimit(10**6)

A = TypeVar('A')
B = TypeVar('B')
C = TypeVar('C', bound=Union[int, float, str])

# ---------------------------------------------------------------------
# Ejercicio 1. 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,
#    >>> segmentos((lambda x: x % 2 == 0), [1,2,0,4,9,6,4,5,7,2])
#    [[2, 0, 4], [6, 4], [2]]
#    >>> segmentos((lambda x: x % 2 == 1), [1,2,0,4,9,6,4,5,7,2])
#    [[1], [9], [5, 7]]
# ---------------------------------------------------------------------

# 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

# ---------------------------------------------------------------------
# Ejercicio 2.1. Definir, por comprensión, la función
#    relacionadosC : (Callable[[A, A], bool], list[A]) -> bool
# tal que relacionadosC(r, xs) se verifica si para todo par (x,y) de
# elementos consecutivos de xs se cumple la relación r. Por ejemplo,
#    >>> relacionadosC(lambda x, y: x < y, [2, 3, 7, 9])
#    True
#    >>> relacionadosC(lambda x, y: x < y, [2, 3, 1, 9])
#    False
# ---------------------------------------------------------------------

def relacionadosC(r: Callable[[A, A], bool], xs: list[A]) -> bool:
    return all((r(x, y) for (x, y) in zip(xs, xs[1:])))

# ---------------------------------------------------------------------
# Ejercicio 2.2. Definir, por recursión, la función
#    relacionadosR : (Callable[[A, A], bool], list[A]) -> bool
# tal que relacionadosR(r, xs) se verifica si para todo par (x,y) de
# elementos consecutivos de xs se cumple la relación r. Por ejemplo,
#    >>> relacionadosR(lambda x, y: x < y, [2, 3, 7, 9])
#    True
#    >>> relacionadosR(lambda x, y: x < y, [2, 3, 1, 9])
#    False
# ---------------------------------------------------------------------

def relacionadosR(r: Callable[[A, A], bool], xs: list[A]) -> bool:
    if len(xs) >= 2:
        return r(xs[0], xs[1]) and relacionadosR(r, xs[1:])
    return True

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

```python
# Ejercicio 3.1. 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]]
# ----------------------------------------------------------------------

# 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

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

# 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

# ----------------------------------------------------------------------
# Ejercicio 3.2. Comprobar con Hypothesis que la longitud de todos los
# elementos de agrupa(xs) es igual a la longitud de xs.
# ----------------------------------------------------------------------

# 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)))
```

```
# ---------------------------------------------------------------------
# Ejercicio 4.1. Definir, por comprensión, la función
#    concC : (list[list[A]]) -> list[A]
# tal que concC(xss) es la concenación de las listas de xss. Por
# ejemplo,
#    concC([[1,3],[2,4,6],[1,9]])  ==  [1,3,2,4,6,1,9]
# ---------------------------------------------------------------------

def concC(xss: list[list[A]]) -> list[A]:
    return [x for xs in xss for x in xs]

# ---------------------------------------------------------------------
# Ejercicio 4.2. Definir, por recursión, la función
#    concR : (list[list[A]]) -> list[A]
# tal que concR(xss) es la concenación de las listas de xss. Por
# ejemplo,
#    concR([[1,3],[2,4,6],[1,9]])  ==  [1,3,2,4,6,1,9]
# ---------------------------------------------------------------------

def concR(xss: list[list[A]]) -> list[A]:
    if not xss:
        return []
    return xss[0] + concR(xss[1:])

# ---------------------------------------------------------------------
# Ejercicio 4.3. Definir, usando reduce, la función
#    concP : (Any) -> Any:
# tal que concP(xss) es la concenación de las listas de xss. Por
# ejemplo,
#    concP([[1,3],[2,4,6],[1,9]])  ==  [1,3,2,4,6,1,9]
# ---------------------------------------------------------------------

def concP(xss: Any) -> Any:
    return reduce(concat, xss)

# ---------------------------------------------------------------------
# Ejercicio 4.4. Comprobar con Hypothesis que la funciones concC,
# concatR y concP son equivalentes.
# ---------------------------------------------------------------------
```

```
# La propiedad es
@given(st.lists(st.lists(st.integers()), min_size=1))
def test_conc(xss: list[list[int]]) -> None:
    r = concC(xss)
    assert concR(xss) == r
    assert concP(xss) == r

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

# La comparación es
#    >>> tiempo('concC([list(range(n)) for n in range(1500)])')
#    0.04 segundos
#    >>> tiempo('concR([list(range(n)) for n in range(1500)])')
#    6.28 segundos
#    >>> tiempo('concP([list(range(n)) for n in range(1500)])')
#    2.55 segundos

# ---------------------------------------------------------------------
# Ejercicio 4.5. Comprobar con Hypothesis que la longitud de
# concatP(xss) es la suma de las longitudes de los elementos de xss.
# ---------------------------------------------------------------------

# La propiedad es
@given(st.lists(st.lists(st.integers()), min_size=1))
def test_long_conc(xss: list[list[int]]) -> None:
    assert len(concP(xss)) == sum(map(len, xss))

# ---------------------------------------------------------------------
# Ejercicio 5.1. Definir, por comprensión, la función
#    filtraAplicaC : (Callable[[A], B], Callable[[A], bool], list[A])
#                    -> list[B]
# tal que filtraAplicaC(f, p, xs) es la lista obtenida aplicándole a los
# elementos de xs que cumplen el predicado p la función f. Por ejemplo,
#    >>> filtraAplicaC(lambda x: x + 4, lambda x: x < 3, range(1, 7))
#    [5, 6]
# ---------------------------------------------------------------------

def filtraAplicaC(f: Callable[[A], B],
                  p: Callable[[A], bool],
```

```
                  xs: list[A]) -> list[B]:
    return [f(x) for x in xs if p(x)]

# ---------------------------------------------------------------------
# Ejercicio 5.2. Definir, usando map y filter, la función
#    filtraAplicaMF : (Callable[[A], B], Callable[[A], bool], list[A])
#                     -> list[B]
# tal que filtraAplicaMF(f, p, xs) es la lista obtenida aplicándole a los
# elementos de xs que cumplen el predicado p la función f. Por ejemplo,
#    >>> filtraAplicaMF(lambda x: x + 4, lambda x: x < 3, range(1, 7))
#    [5, 6]
# ---------------------------------------------------------------------

def filtraAplicaMF(f: Callable[[A], B],
                   p: Callable[[A], bool],
                   xs: list[A]) -> list[B]:
    return list(map(f, filter(p, xs)))

# ---------------------------------------------------------------------
# Ejercicio 5.3. Definir, por recursión, la función
#    filtraAplicaR : (Callable[[A], B], Callable[[A], bool], list[A])
#                    -> list[B]
# tal que filtraAplicaR(f, p, xs) es la lista obtenida aplicándole a los
# elementos de xs que cumplen el predicado p la función f. Por ejemplo,
#    >>> filtraAplicaR(lambda x: x + 4, lambda x: x < 3, range(1, 7))
#    [5, 6]
# ---------------------------------------------------------------------

def filtraAplicaR(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])] + filtraAplicaR(f, p, xs[1:])
    return filtraAplicaR(f, p, xs[1:])

# ---------------------------------------------------------------------
# Ejercicio 5.4. Definir, por plegado, la función
#    filtraAplicaP : (Callable[[A], B], Callable[[A], bool], list[A])
```

```
#                    -> list[B]
# tal que filtraAplicaP(f, p, xs) es la lista obtenida aplicándole a los
# elementos de xs que cumplen el predicado p la función f. Por ejemplo,
#    >>> filtraAplicaP(lambda x: x + 4, lambda x: x < 3, range(1, 7))
#    [5, 6]
# ----------------------------------------------------------------------

def filtraAplicaP(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, [])

# ----------------------------------------------------------------------
# Ejercicio 5.5. Definir, por iteración, la función
#    filtraAplicaI : (Callable[[A], B], Callable[[A], bool], list[A])
#                    -> list[B]
# tal que filtraAplicaI(f, p, xs) es la lista obtenida aplicándole a los
# elementos de xs que cumplen el predicado p la función f. Por ejemplo,
#    >>> filtraAplicaI(lambda x: x + 4, lambda x: x < 3, range(1, 7))
#    [5, 6]
# ----------------------------------------------------------------------

def filtraAplicaI(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

# ----------------------------------------------------------------------
# Ejercicio 5.6. Comprobar que las definiciones de filtraAplica son
# equivalentes.
# ----------------------------------------------------------------------
```

```python
# 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 = filtraAplicaC(f, p, xs)
    assert filtraAplicaMF(f, p, xs) == r
    assert filtraAplicaR(f, p, xs) == r
    assert filtraAplicaP(f, p, xs) == r
    assert filtraAplicaI(f, p, xs) == r

# ---------------------------------------------------------------------
# Ejercicio 5.7. Comparar la eficiencia de las definiciones de
# filtraAplica.
# ---------------------------------------------------------------------

# La comparación es
#    >>> tiempo('filtraAplicaC(lambda x: x, lambda x: x % 2 == 0,
#                               range(10**5))')
#    0.02 segundos
#    >>> tiempo('filtraAplicaMF(lambda x: x, lambda x: x % 2 == 0,
#                                range(10**5))')
#    0.01 segundos
#    >>> tiempo('filtraAplicaR(lambda x: x, lambda x: x % 2 == 0,
#                               range(10**5))')
#    Process Python violación de segmento (core dumped)
#    >>> tiempo('filtraAplicaP(lambda x: x, lambda x: x % 2 == 0,
#                               range(10**5))')
#    4.07 segundos
#    >>> tiempo('filtraAplicaI(lambda x: x, lambda x: x % 2 == 0,
#                               range(10**5))')
#    0.01 segundos
#
#    >>> tiempo('filtraAplicaC(lambda x: x, lambda x: x % 2 == 0,
#                               range(10**7))')
```

```
#    1.66 segundos
#    >>> tiempo('filtraAplicaMF(lambda x: x, lambda x: x % 2 == 0,
#                               range(10**7))')
#    1.00 segundos
#    >>> tiempo('filtraAplicaI(lambda x: x, lambda x: x % 2 == 0,
#                              range(10**7))')
#    1.21 segundos

# ---------------------------------------------------------------------
# Ejercicio 6.1. Definir la función
#    maximo : (list[C]) -> C:
# 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"
# ---------------------------------------------------------------------

# 1ª solución
# ===========

def maximo1(xs: list[C]) -> C:
    if len(xs) == 1:
        return xs[0]
    return max(xs[0], maximo1(xs[1:]))

# 2ª solución
# ===========

def maximo2(xs: list[C]) -> C:
    return reduce(max, xs)

# 3ª solución
# ===========

def maximo3(xs: list[C]) -> C:
    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

# ---------------------------------------------------------------------
# Comprobación de las propiedades
# ---------------------------------------------------------------------

# La comprobación es
#    src> poetry run pytest -q \
#         funciones_de_orden_superior_y_definiciones_por_plegados.py
#    1 passed in 0.74s
```

Capítulo 5

Tipos definidos y de datos algebraicos

5.1. Tipos de datos algebraicos: Árboles binarios

```
# ----------------------------------------------------------------------
# Introducción                                                       --
# ----------------------------------------------------------------------

# En esta relación se presenta ejercicios sobre árboles binarios
# definidos como tipos de datos algebraicos.
#
# Los ejercicios corresponden al tema 9 que se encuentran en
#    https://jaalonso.github.io/cursos/i1m/temas/tema-9.html

# ----------------------------------------------------------------------
# Librerías auxiliares
# ----------------------------------------------------------------------

from dataclasses import dataclass
from random import choice, randint
from typing import Callable, Generic, TypeVar

from hypothesis import given
from hypothesis import strategies as st

A = TypeVar("A")
```

```
B = TypeVar("B")

# ---------------------------------------------------------------------
# Nota 1. En los siguientes ejercicios se trabajará con los árboles
# binarios definidos como sigue
#    @dataclass
#    class Arbol(Generic[A]):
#        pass
#
#    @dataclass
#    class H(Arbol[A]):
#        x: A
#
#    @dataclass
#    class N(Arbol[A]):
#        x: A
#        i: Arbol[A]
#        d: Arbol[A]
# Por ejemplo, el árbol
#          9
#        / \
#       /   \
#      3     7
#     / \
#    2   4
# se representa por
#    N(9, N(3, H(2), H(4)), H(7))
# ---------------------------------------------------------------------

@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]

# ---------------------------------------------------------------------
# Nota 2. En las comprobación de propiedades se usará el generador
#    arbolArbitrario(int) -> Arbol[int]
# tal que (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))])

# ---------------------------------------------------------------------
# Ejercicio 1.1. Definir la función
#    nHojas : (Arbol[A]) -> int
# tal 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
# ---------------------------------------------------------------------

def nHojas(a: Arbol[A]) -> int:
    match a:
        case H(_):
            return 1
        case N(_, i, d):
            return nHojas(i) + nHojas(d)
    assert False

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

```python
# Ejercicio 1.2. Definir la función
#    nNodos : (Arbol[A]) -> int
# tal que nNodos(x) es el número de nodos del árbol x. Por ejemplo,
#      nNodos(N(9, N(3, H(2), H(4)), H(7)))  ==  2
# ----------------------------------------------------------------------

def nNodos(a: Arbol[A]) -> int:
    match a:
        case H(_):
            return 0
        case N(_, i, d):
            return 1 + nNodos(i) + nNodos(d)
    assert False

# ----------------------------------------------------------------------
# Ejercicio 1.3. Comprobar con Hypothesis que en todo árbol binario el
# número de sus hojas es igual al número de sus nodos más uno.
# ----------------------------------------------------------------------

# 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

# ----------------------------------------------------------------------
# Ejercicio 2.1. 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
# ----------------------------------------------------------------------

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

# ---------------------------------------------------------------------
# Ejercicio 2.2. Comprobar con Hypothesis que para todo árbol biario
# x, se tiene que
#    nNodos(x) <= 2^profundidad(x) - 1
# ---------------------------------------------------------------------

# La propiedad es
@given(st.integers(min_value=1, max_value=10))
def test_nNodos(n: int) -> None:
    a = arbolArbitrario(n)
    assert nNodos(a) <= 2 ** profundidad(a) - 1

# ---------------------------------------------------------------------
# Ejercicio 3.1. Definir la función
#    preorden  : (Arbol[A]) -> list[A]
# tal 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]
# ---------------------------------------------------------------------

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

# ---------------------------------------------------------------------
# Ejercicio 3.2. Comprobar con Hypothesis que la longitud de la lista
# obtenida recorriendo un árbol en sentido preorden es igual al número
# de nodos del árbol más el número de hojas.
# ---------------------------------------------------------------------

# La propiedad es
```

```python
@given(st.integers(min_value=1, max_value=10))
def test_recorrido(n: int) -> None:
    a = arbolArbitrario(n)
    assert len(preorden(a)) == nNodos(a) + nHojas(a)

# ----------------------------------------------------------------------
# Ejercicio 3.3. Definir la función
#    postorden : (Arbol[A]) -> list[A]
# tal que (postorden x) es la lista correspondiente al recorrido postorden
# del árbol x; es decir, primero recorre el subárbol izquierdo, a
# continuación el subárbol derecho y, finalmente, la raíz del
# árbol. Por ejemplo,
#    >>> postorden(N(9, N(3, H(2), H(4)), H(7)))
#    [2, 4, 3, 7, 9]
# ----------------------------------------------------------------------

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

# ----------------------------------------------------------------------
# Ejercicio 4.1. 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)))
# ----------------------------------------------------------------------

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

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

```python
# Ejercicio 4.2. Comprobar con Hypothesis que para todo árbol x,
#    espejo(espejo(x)) = x
# ----------------------------------------------------------------------

@given(st.integers(min_value=1, max_value=10))
def test_espejo1(n: int) -> None:
    x = arbolArbitrario(n)
    assert espejo(espejo(x)) == x

# ----------------------------------------------------------------------
# Ejercicio 4.3. Comprobar con Hypothesis que para todo árbol binario
# x, se tiene que
#    reversed(preorden(espejo(x))) = postorden(x)
# ----------------------------------------------------------------------

@given(st.integers(min_value=1, max_value=10))
def test_espejo2(n: int) -> None:
    x = arbolArbitrario(n)
    assert list(reversed(preorden(espejo(x)))) == postorden(x)

# ----------------------------------------------------------------------
# Ejercicio 4.4. Comprobar con Hypothesis que para todo árbol x,
#    postorden(espejo(x)) = reversed(preorden(x))
# ----------------------------------------------------------------------

@given(st.integers(min_value=1, max_value=10))
def test_espejo(n: int) -> None:
    x = arbolArbitrario(n)
    assert postorden(espejo(x)) == list(reversed(preorden(x)))

# ----------------------------------------------------------------------
# Ejercicio 5.1. 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))
# ----------------------------------------------------------------------

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

# ----------------------------------------------------------------------
# Ejercicio 5.2. 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.
# ----------------------------------------------------------------------

# 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

# ----------------------------------------------------------------------
# Ejercicio 6.2. 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)))
# ----------------------------------------------------------------------
```

```python
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

# ---------------------------------------------------------------------
# Ejercicio 6.2. Comprobar con Hypothesis que el número de hojas de
# replicateArbol(n,x) es 2^n, para todo número natural n
# ---------------------------------------------------------------------

# La propiedad es
@given(st.integers(min_value=1, max_value=10),
       st.integers(min_value=1, max_value=10))
def test_replicateArbol(n: int, x: int) -> None:
    assert nHojas(replicateArbol(n, x)) == 2**n

# ---------------------------------------------------------------------
# Ejercicio 7.1. Definir la función
#    mapArbol : (Callable[[A], B], Arbol[A]) -> Arbol[B]
# tal que mapArbol(f, x) es el árbol obtenido aplicándole a cada nodo de
# x la función f. Por ejemplo,
#    >>> mapArbol(lambda x: 2 * x, N(9, N(3, H(2), H(4)), H(7)))
#    N(18, N(6, H(4), H(8)), H(14))
# ---------------------------------------------------------------------

def mapArbol(f: Callable[[A], B], a: Arbol[A]) -> Arbol[B]:
    match a:
        case H(x):
            return H(f(x))
        case N(x, i, d):
            return N(f(x), mapArbol(f, i), mapArbol(f, d))
    assert False

# ---------------------------------------------------------------------
# Ejercicio 7.2. Comprobar con Hypothesis que
```

```
#    list(map(lambda n: 1 + n, preorden(x))) ==
#    list(preorden(mapArbol(lambda n: 1 + n, x)))
# ----------------------------------------------------------------------

@given(st.integers(min_value=1, max_value=10))
def test_map_preorden(n: int) -> None:
    x = arbolArbitrario(n)
    print(x)
    assert list(map(lambda n: 1 + n, preorden(x))) == \
        list(preorden(mapArbol(lambda n: 1 + n, x)))

# ----------------------------------------------------------------------
# Comprobación de las propiedades
# ----------------------------------------------------------------------

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

5.2. Tipos de datos algebraicos: Árboles

```
# ----------------------------------------------------------------------
# Introducción                                                       --
# ----------------------------------------------------------------------

# En esta relación se presenta ejercicios sobre distintos tipos de
# datos algebraicos. Concretamente,
# + Árboles binarios:
#   + Árboles binarios con valores en los nodos.
#   + Árboles binarios con valores en las hojas.
#   + Árboles binarios con valores en las hojas y en los nodos.
#   + Árboles booleanos.
# + Árboles generales
#
# Los ejercicios corresponden al tema 9 que se encuentran en
#    https://jaalonso.github.io/cursos/i1m/temas/tema-9.html

# ----------------------------------------------------------------------
# Librerías auxiliares
# ----------------------------------------------------------------------
```

```
from dataclasses import dataclass
from math import ceil, sqrt
from typing import Callable, Generic, TypeVar

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

# ---------------------------------------------------------------------
# Ejercicio 1.1. Los árboles binarios con valores en los nodos se pueden
# definir por
#    @dataclass
#    class Arbol1(Generic[A]):
#        pass
#
#    @dataclass
#    class H1(Arbol1[A]):
#        pass
#
#    @dataclass
#    class N1(Arbol1[A]):
#        x: A
#        i: Arbol1
#        d: Arbol1
# Por ejemplo, el árbol
#         9
#        / \
#       /   \
#      8     6
#     / \   / \
#    3   2 4   5
# se puede representar por
#    N1(9,
#      N1(8, N1(3, H1(), H1()), N1(2, H1(), H1())),
#      N1(6, N1(4, H1(), H1()), N1(5, H1(), H1())))
#
# Definir la función
#    sumaArbol : (Arbol1) -> int
# tal sumaArbol(x) es la suma de los valores que hay en el árbol x.
# Por ejemplo,
```

```
#    >>> sumaArbol(N1(2,
#                      N1(5, N1(3, H1(), H1()), N1(7, H1(), H1())),
#                      N1(4, H1(), H1())))
#    21
# ----------------------------------------------------------------------

@dataclass
class Arbol1(Generic[A]):
    pass

@dataclass
class H1(Arbol1[A]):
    pass

@dataclass
class N1(Arbol1[A]):
    x: A
    i: Arbol1
    d: Arbol1

def sumaArbol(a: Arbol1[int]) -> int:
    match a:
        case H1():
            return 0
        case N1(x, i, d):
            return x + sumaArbol(i) + sumaArbol(d)
    assert False

# ----------------------------------------------------------------------
# Ejercicio 1.2. Definir la función
#    mapArbol : (Callable[[A], B], Arbol1[A]) -> Arbol1[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,
#                 N1(2,
#                   N1(5, N1(3, H1(), H1()), N1(7, H1(), H1())),
#                   N1(4, H1(), H1())))
#    N1(3, N1(6, N1(4, H1(), H1()), N1(8, H1(), H1())), N1(5, H1(), H1()))
# ----------------------------------------------------------------------
```

```
def mapArbol(f: Callable[[A], B], a: Arbol1[A]) -> Arbol1[B]:
    match a:
        case H1():
            return H1()
        case N1(x, i, d):
            return N1(f(x), mapArbol(f, i), mapArbol(f, d))
    assert False

# ----------------------------------------------------------------------
# Ejercicio 1.3. Definir la función
#    ramaIzquierda : (Arbol1[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(N1(2,
#                         N1(5, N1(3, H1(), H1()), N1(7, H1(), H1())),
#                         N1(4, H1(), H1())))
#    [2, 5, 3]
# ----------------------------------------------------------------------

def ramaIzquierda(a: Arbol1[A]) -> list[A]:
    match a:
        case H1():
            return []
        case N1(x, i, _):
            return [x] + ramaIzquierda(i)
    assert False

# ----------------------------------------------------------------------
# Ejercicio 1.4. Diremos que un árbol está balanceado si para cada nodo
# la diferencia entre el número de nodos de sus subárboles izquierdo y
# derecho es menor o igual que uno.
#
# Definir la función
#    balanceado : (Arbol1[A]) -> bool
# tal que balanceado(a) se verifica si el árbol a está balanceado. Por
# ejemplo,
#    >>> balanceado(N1(5, H1(), N1(3, H1(), H1())))
#    True
#    >>> balanceado(N1(4,
#                       N1(3, N1(2, H1(), H1()), H1()),
```

```
#                     N1(5, H1(), N1(6, H1(), N1(7, H1(), H1())))
#    False
# ----------------------------------------------------------------------

def numeroNodos(a: Arbol1[A]) -> int:
    match a:
        case H1():
            return 0
        case N1(_, i, d):
            return 1 + numeroNodos(i) + numeroNodos(d)
    assert False

def balanceado(a: Arbol1[A]) -> bool:
    match a:
        case H1():
            return True
        case N1(_, i, d):
            return abs(numeroNodos(i) - numeroNodos(d)) <= 1 \
                and balanceado(i) and balanceado(d)
    assert False

# ----------------------------------------------------------------------
# Ejercicio 2. Los árboles binarios con valores en las hojas se pueden
# definir por
#    @dataclass
#    class Arbol2(Generic[A]):
#        pass
#
#    @dataclass
#    class H2(Arbol2[A]):
#        x: A
#
#    @dataclass
#    class N2(Arbol2[A]):
#        i: Arbol2[A]
#        d: Arbol2[A]
# Por ejemplo, los árboles
#    árbol1          árbol2       árbol3     árbol4
#       o              o            o          o
#      / \            / \          / \        / \
```

```
#     1   o          o   3       o   3       o   1
#        / \        / \         / \         / \
#       2   3      1   2       1   4       2   3
# se representan por
#    arbol1: Arbol2[int] = N2(H2(1), N2(H2(2), H2(3)))
#    arbol2: Arbol2[int] = N2(N2(H2(1), H2(2)), H2(3))
#    arbol3: Arbol2[int] = N2(N2(H2(1), H2(4)), H2(3))
#    arbol4: Arbol2[int] = N2(N2(H2(2), H2(3)), H2(1))
#
# Definir la función
#    igualBorde : (Arbol2[A], Arbol2[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
# ----------------------------------------------------------------------

@dataclass
class Arbol2(Generic[A]):
    pass

@dataclass
class H2(Arbol2[A]):
    x: A

@dataclass
class N2(Arbol2[A]):
    i: Arbol2[A]
    d: Arbol2[A]

arbol1: Arbol2[int] = N2(H2(1), N2(H2(2), H2(3)))
arbol2: Arbol2[int] = N2(N2(H2(1), H2(2)), H2(3))
arbol3: Arbol2[int] = N2(N2(H2(1), H2(4)), H2(3))
arbol4: Arbol2[int] = N2(N2(H2(2), H2(3)), H2(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: Arbol2[A]) -> list[A]:
```

```
    match a:
        case H2(x):
            return [x]
        case N2(i, d):
            return borde(i) + borde(d)
    assert False

def igualBorde(t1: Arbol2[A], t2: Arbol2[A]) -> bool:
    return borde(t1) == borde(t2)

# ----------------------------------------------------------------------
# Ejercicio 3.1. Los árboles binarios con valores en las hojas y en los
# nodos se definen por
#    @dataclass
#    class Arbol3(Generic[A]):
#        pass
#
#    @dataclass
#    class H3(Arbol3[A]):
#        x: A
#
#    @dataclass
#    class N3(Arbol3[A]):
#        x: A
#        i: Arbol3[A]
#        d: Arbol3[A]
# Por ejemplo, los árboles
#         5              8             5           5
#        / \            / \           / \         / \
#       /   \          /   \         /   \       /   \
#      9     7        9     3       9     2     4     7
#     / \   / \      / \   / \     / \               / \
#    1   4 6   8    1   4 6   2   1   4             6   2
# se pueden representar por
#     ej3arbol1: Arbol3[int] = N3(5, N3(9, H3(1), H3(4)), N3(7, H3(6), H3(8)))
#     ej3arbol2: Arbol3[int] = N3(8, N3(9, H3(1), H3(4)), N3(3, H3(6), H3(2)))
#     ej3arbol3: Arbol3[int] = N3(5, N3(9, H3(1), H3(4)), H3(2))
#     ej3arbol4: Arbol3[int] = N3(5, H3(4), N3(7, H3(6), H3(2)))
#
# Definir la función
```

```
#    igualEstructura : (Arbol3[A], Arbol3[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
# ----------------------------------------------------------------------

@dataclass
class Arbol3(Generic[A]):
    pass

@dataclass
class H3(Arbol3[A]):
    x: A

@dataclass
class N3(Arbol3[A]):
    x: A
    i: Arbol3[A]
    d: Arbol3[A]

ej3arbol1: Arbol3[int] = N3(5, N3(9, H3(1), H3(4)), N3(7, H3(6), H3(8)))
ej3arbol2: Arbol3[int] = N3(8, N3(9, H3(1), H3(4)), N3(3, H3(6), H3(2)))
ej3arbol3: Arbol3[int] = N3(5, N3(9, H3(1), H3(4)), H3(2))
ej3arbol4: Arbol3[int] = N3(5, H3(4), N3(7, H3(6), H3(2)))

def igualEstructura(a: Arbol3[A], b: Arbol3[A]) -> bool:
    match (a, b):
        case (H3(_), H3(_)):
            return True
        case (N3(_, i1, d1), N3(_, i2, d2)):
            return igualEstructura(i1, i2) and igualEstructura(d1, d2)
        case (_, _):
            return False
    assert False

# ----------------------------------------------------------------------
# Ejercicio 3.2. Definir la función
#    algunoArbol3 : (Arbol3[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(N3(5, N3(3, H3(1), H3(4)), H3(2)), lambda x: x > 4)
#    True
#    >>> algunoArbol(N3(5, N3(3, H3(1), H3(4)), H3(2)), lambda x: x > 7)
#    False
# ----------------------------------------------------------------------

def algunoArbol(a: Arbol3[A], p: Callable[[A], bool]) -> bool:
    match a:
        case H3(x):
            return p(x)
        case N3(x, i, d):
            return p(x) or algunoArbol(i, p) or algunoArbol(d, p)
    assert False

# ----------------------------------------------------------------------
# Ejercicio 3.3. Un elemento de un árbol se dirá de nivel k si aparece
# en el árbol a distancia k  de la raíz.
#
# Definir la función
#    nivel : (int, Arbol3[A]) -> list[A]
# tal que nivel(k, a) es la lista de los elementos de nivel k del árbol
# a. Por ejemplo,
#     >>> nivel(0, N3(7, N3(2, H3(5), H3(4)), H3(9)))
#     [7]
#     >>> nivel(1, N3(7, N3(2, H3(5), H3(4)), H3(9)))
#     [2, 9]
#     >>> nivel(2, N3(7, N3(2, H3(5), H3(4)), H3(9)))
#     [5, 4]
#     >>> nivel(3, N3(7, N3(2, H3(5), H3(4)), H3(9)))
#     []
# ----------------------------------------------------------------------

def nivel(k: int, a: Arbol3[A]) -> list[A]:
    match (k, a):
        case (0, H3(x)):
            return [x]
        case (0, N3(x, _, _)):
            return [x]
```

```
        case (_, H3(_)):
            return []
        case (_, N3(_, i, d)):
            return nivel(k - 1, i) + nivel(k - 1, d)
    assert False

# ----------------------------------------------------------------------
# Ejercicio 3.4. 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) -> Arbol3[int]
# tal que arbolFactorizacion(n) es el árbol de factorización de n. Por
# ejemplo,
#    arbolFactorizacion(60) == N3(60,
#                                 N3(6, H3(2), H3(3)),
#                                 N3(10, H3(2), H3(5)))
#    arbolFactorizacion(45) == N3(45, H3(5), N3(9, H3(3), H3(3)))
#    arbolFactorizacion(7)  == H3(7)
#    arbolFactorizacion(9)  == N3(9, H3(3), H3(3))
#    arbolFactorizacion(14) == N3(14, H3(2), H3(7))
```

```
#    arbolFactorizacion(28) == N3(28, N3(4, H3(2), H3(2)), H3(7))
#    arbolFactorizacion(84) == N3(84,
#                                 H3(7),
#                                 N3(12, H3(3), N3(4, H3(2), H3(2))))
# ----------------------------------------------------------------------

# 1ª solución
# ===========

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

# divisoresMedio(n) es el par formado por los divisores medios de
# n. Por ejemplo,
#    divisoresMedio(30)  ==  (5,6)
#    divisoresMedio(7)   ==  (1,7)
#    divisoresMedio(16)  ==  (4,4)
def divisoresMedio(n: int) -> tuple[int, int]:
    xs = divisores(n)
    x = xs[len(xs) // 2]
    return (n // x, x)

# esPrimo(n) se verifica si n es primo. Por ejemplo,
#    esPrimo(7)  ==  True
#    esPrimo(9)  ==  False
def esPrimo(n: int) -> bool:
    return divisores(n) == [1, n]

def arbolFactorizacion1(n: int) -> Arbol3[int]:
    if esPrimo(n):
        return H3(n)
    (x, y) = divisoresMedio(n)
    return N3(n, arbolFactorizacion1(x), arbolFactorizacion1(y))

# 2ª solución
# ===========

# divisoresMedio2(n) es el par formado por los divisores medios de
```

```python
# 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) -> Arbol3[int]:
    if esPrimo(n):
        return H3(n)
    (x, y) = divisoresMedio2(n)
    return N3(n, arbolFactorizacion2(x), arbolFactorizacion2(y))

# ---------------------------------------------------------------------
# Ejercicio 4. Se consideran los árboles con operaciones booleanas
# definidos por
#    @dataclass
#    class ArbolB:
#        pass
#
#    @dataclass
#    class H(ArbolB):
#        x: bool
#
#    @dataclass
#    class Conj(ArbolB):
#        i: ArbolB
#        d: ArbolB
#
#    @dataclass
#    class Disy(ArbolB):
#        i: ArbolB
#        d: ArbolB
#
#    @dataclass
#    class Neg(ArbolB):
#        a: ArbolB
#
```

```
# 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: ArbolB = Conj(Disy(Conj(H(True), H(False)),
#                            (Neg(H(False)))),
#                       (Conj(Neg(H(False)),
#                             (H(True)))))
#
#    ej2: ArbolB = Conj(Disy(Conj(H(True), H(False)),
#                            (Neg(H(True)))),
#                       (Conj(Neg(H(False)),
#                             (H(True)))))
#
# Definir la función
#    valorB : (ArbolB) -> bool
# tal que valorB(a) es el resultado de procesar el árbol a realizando
# las operaciones booleanas especificadas en los nodos. Por ejemplo,
#    valorB(ej1) == True
#    valorB(ej2) == False
# ----------------------------------------------------------------------

@dataclass
class ArbolB:
    pass

@dataclass
class H(ArbolB):
    x: bool

@dataclass
class Conj(ArbolB):
    i: ArbolB
```

```
    d: ArbolB

@dataclass
class Disy(ArbolB):
    i: ArbolB
    d: ArbolB

@dataclass
class Neg(ArbolB):
    a: ArbolB

ej1: ArbolB = Conj(Disy(Conj(H(True), H(False)),
                        (Neg(H(False)))),
                   (Conj(Neg(H(False)),
                         (H(True)))))

ej2: ArbolB = Conj(Disy(Conj(H(True), H(False)),
                        (Neg(H(True)))),
                   (Conj(Neg(H(False)),
                         (H(True)))))

def valorB(a: ArbolB) -> bool:
    match a:
        case H(x):
            return x
        case Neg(b):
            return not valorB(b)
        case Conj(i, d):
            return valorB(i) and valorB(d)
        case Disy(i, d):
            return valorB(i) or valorB(d)
    assert False

# ------------------------------------------------------------------
# Ejercicio 5. Los árboles generales se pueden representar mediante el
# siguiente tipo de dato
#    @dataclass
#    class ArbolG(Generic[A]):
#        pass
#
```

```
#    @dataclass
#    class NG(ArbolG[A]):
#        x: A
#        y: list[ArbolG[A]]
# Por ejemplo, los árboles
#      1               3               3
#     / \             /|\             / | \
#    2   3           / | \           /  |  \
#        |          5  4  7         5   4   7
#        4          |     /\        |   |  / \
#                   6    2  1       6   1 2   1
#                                       / \
#                                      2   3
#                                          |
#                                          4
# se representan por
#    ejG1: ArbolG[int] = NG(1, [NG(2, []), NG(3, [NG(4, [])])])
#    ejG2: ArbolG[int] = NG(3, [NG(5, [NG(6, [])]),
#                               NG(4, []),
#                               NG(7, [NG(2, []), NG(1, [])])])
#    ejG3: ArbolG[int] = NG(3, [NG(5, [NG(6, [])]),
#                               NG(4, [NG(1, [NG(2, []),
#                                             NG(3, [NG(4, [])])])]),
#                               NG(7, [NG(2, []), NG(1, [])])])
#
# Definir la función
#     ramifica : (ArbolG[A], ArbolG[A], Callable[[A], bool]) -> ArbolG[A]
# tal que ramifica(a1, a2, p) es el árbol que resulta de añadir una copia
# del árbol a2 a los nodos de a1 que cumplen un predicado p. Por
# ejemplo,
#    >>> ramifica(ejG1, NG(8, []), lambda x: x > 4)
#    NG(1, [NG(2, []), NG(3, [NG(4, [])])])
#    >>> ramifica(ejG1, NG(8, []), lambda x: x > 3)
#    NG(1, [NG(2, []), NG(3, [NG(4, [NG(8, [])])])])
#    >>> ramifica(ejG1, NG(8, []), lambda x: x > 2)
#    NG(1, [NG(2, []), NG(3, [NG(4, [NG(8, [])]), NG(8, [])])])
#    >>> ramifica(ejG1, NG(8, []), lambda x: x > 1)
#    NG(1, [NG(2, [NG(8, [])]), NG(3, [NG(4, [NG(8, [])]), NG(8, [])])])
#    >>> ramifica(ejG1, NG(8, []), lambda x: x > 0)
#    NG(1, [NG(2, [NG(8, [])]),
```

```
#           NG(3, [NG(4, [NG(8, [])]), NG(8, [])]),
#           NG(8, [])])
# -- ------------------------------------------------------------------

@dataclass
class ArbolG(Generic[A]):
    pass

@dataclass
class NG(ArbolG[A]):
    x: A
    y: list[ArbolG[A]]

ejG1: ArbolG[int] = NG(1, [NG(2, []), NG(3, [NG(4, [])])])
ejG2: ArbolG[int] = NG(3, [NG(5, [NG(6, [])]),
                           NG(4, []),
                           NG(7, [NG(2, []), NG(1, [])])])
ejG3: ArbolG[int] = NG(3, [NG(5, [NG(6, [])]),
                           NG(4, [NG(1, [NG(2, []), NG(3, [NG(4, [])])])]),
                           NG(7, [NG(2, []), NG(1, [])])])

def ramifica(a1: ArbolG[A],
             a2: ArbolG[A],
             p: Callable[[A], bool]) -> ArbolG[A]:
    match a1:
        case NG(x, xs):
            if p(x):
                return NG(x, [ramifica(a, a2, p) for a in xs] + [a2])
            return NG(x, [ramifica(a, a2, p) for a in xs])
    assert False
```

5.3. Tipos de datos algebraicos: Expresiones

```
# ------------------------------------------------------------------
# Introducción                                                    --
# ------------------------------------------------------------------

# En esta relación se continúan con ejercicios sobre distintos tipos de
# datos algebraicos de la relación anterior. Concretamente,
# + Expresiones aritméticas
```

```
#   + Expresiones aritméticas básicas.
#   + Expresiones aritméticas con una variable.
#   + Expresiones aritméticas con varias variables.
#   + Expresiones aritméticas generales.
#   + Expresiones aritméticas con tipo de operaciones.
# + Expresiones vectoriales
#
# Los ejercicios corresponden al tema 9 que se encuentran en
#    https://jaalonso.github.io/cursos/i1m/temas/tema-9.html

# ---------------------------------------------------------------------
# Librerías auxiliares
# ---------------------------------------------------------------------

from dataclasses import dataclass
from enum import Enum
from typing import Callable, TypeVar

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

# ---------------------------------------------------------------------
# Ejercicio 6.1. Las expresiones aritméticas básicas pueden
# representarse usando el siguiente tipo de datos
#    @dataclass
#    class Expr1:
#        pass
#
#    @dataclass
#    class C1(Expr1):
#        x: int
#
#    @dataclass
#    class S1(Expr1):
#        x: Expr1
#        y: Expr1
#
#    @dataclass
#    class P1(Expr1):
#        x: Expr1
```

```
#        y: Expr1
# Por ejemplo, la expresión 2*(3+7) se representa por
#    P1(C1(2), S1(C1(3), C1(7)))
#
# Definir la función
#    valor : (Expr1) -> int:
# tal que valor(e) es el valor de la expresión aritmética e. Por
# ejemplo,
#    valor(P1(C1(2), S1(C1(3), C1(7))))  ==  20
# ----------------------------------------------------------------------

@dataclass
class Expr1:
    pass

@dataclass
class C1(Expr1):
    x: int

@dataclass
class S1(Expr1):
    x: Expr1
    y: Expr1

@dataclass
class P1(Expr1):
    x: Expr1
    y: Expr1

def valor(e: Expr1) -> int:
    match e:
        case C1(x):
            return x
        case S1(x, y):
            return valor(x) + valor(y)
        case P1(x, y):
            return valor(x) * valor(y)
    assert False

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

```python
# Ejercicio 6.2. Definir la función
#    aplica : (Callable[[int], int], Expr1) -> Expr1
# 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, S1(P1(C1(3), C1(5)), P1(C1(6), C1(7))))
#    S1(P1(C1(5), C1(7)), P1(C1(8), C1(9)))
#    >>> aplica(lambda x: 2 * x, S1(P1(C1(3), C1(5)), P1(C1(6), C1(7))))
#    S1(P1(C1(6), C1(10)), P1(C1(12), C1(14)))
# ----------------------------------------------------------------------

def aplica(f: Callable[[int], int], e: Expr1) -> Expr1:
    match e:
        case C1(x):
            return C1(f(x))
        case S1(x, y):
            return S1(aplica(f, x), aplica(f, y))
        case P1(x, y):
            return P1(aplica(f, x), aplica(f, y))
    assert False

# ----------------------------------------------------------------------
# Ejercicio 7.1. Las expresiones aritméticas construidas con una
# variable (denotada por X), los números enteros y las operaciones de
# sumar y multiplicar se pueden representar mediante el tipo de datos
# Expr2 definido por
#    @dataclass
#    class Expr2:
#        pass
#
#    @dataclass
#    class X(Expr2):
#        pass
#
#    @dataclass
#    class C2(Expr2):
#        x: int
#
#    @dataclass
#    class S2(Expr2):
#        x: Expr2
```

```
#         y: Expr2
#
#     @dataclass
#     class P2(Expr2):
#         x: Expr2
#         y: Expr2
# Por ejemplo, la expresión X*(13+X) se representa por
#     P2(X(), S2(C2(13), X()))
#
# Definir la función
#     valorE : (Expr2, int) -> int
# tal que valorE(e, n) es el valor de la expresión e cuando se
# sustituye su variable por n. Por ejemplo,
#     valorE(P2(X(), S2(C2(13), X())), 2)  ==  30
# ----------------------------------------------------------------------

@dataclass
class Expr2:
    pass

@dataclass
class X(Expr2):
    pass

@dataclass
class C2(Expr2):
    x: int

@dataclass
class S2(Expr2):
    x: Expr2
    y: Expr2

@dataclass
class P2(Expr2):
    x: Expr2
    y: Expr2

def valorE(e: Expr2, n: int) -> int:
    match e:
```

```
        case X():
            return n
        case C2(a):
            return a
        case S2(e1, e2):
            return valorE(e1, n) + valorE(e2, n)
        case P2(e1, e2):
            return valorE(e1, n) * valorE(e2, n)
    assert False

# ---------------------------------------------------------------------
# Ejercicio 7.2. Definir la función
#    numVars : (Expr2) -> int
# tal que numVars(e) es el número de variables en la expresión e. Por
# ejemplo,
#    numVars(C2(3))                    ==  0
#    numVars(X())                      ==  1
#    numVars(P2(X(), S2(C2(13), X())))  ==  2
# ---------------------------------------------------------------------

def numVars(e: Expr2) -> int:
    match e:
        case X():
            return 1
        case C2(_):
            return 0
        case S2(e1, e2):
            return numVars(e1) + numVars(e2)
        case P2(e1, e2):
            return numVars(e1) + numVars(e2)
    assert False


# ---------------------------------------------------------------------
# Ejercicio 8.1. Las expresiones aritméticas con variables pueden
# representarse usando el siguiente tipo de datos
#    @dataclass
#    class Expr3:
#        pass
#
```

```
#    @dataclass
#    class C3(Expr3):
#        x: int
#
#    @dataclass
#    class V3(Expr3):
#        x: str
#
#    @dataclass
#    class S3(Expr3):
#        x: Expr3
#        y: Expr3
#
#    @dataclass
#    class P3(Expr3):
#        x: Expr3
#        y: Expr3
#
# Por ejemplo, la expresión 2*(a+5) se representa por
#    P3(C3(2), S3(V3('a'), C3(5)))
#
# Definir la función
#    valor3 : (Expr3, list[tuple[str, int]]) -> int
# tal que valor3(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> valor3(P3(C3(2), S3(V3('a'), V3('b'))), [('a', 2), ('b', 5)])
#    14
# ----------------------------------------------------------------------

@dataclass
class Expr3:
    pass

@dataclass
class C3(Expr3):
    x: int

@dataclass
class V3(Expr3):
```

```
    x: str

@dataclass
class S3(Expr3):
    x: Expr3
    y: Expr3

@dataclass
class P3(Expr3):
    x: Expr3
    y: Expr3

def valor3(e: Expr3, xs: list[tuple[str, int]]) -> int:
    match e:
        case C3(a):
            return a
        case V3(x):
            return [y for (z, y) in xs if z == x][0]
        case S3(e1, e2):
            return valor3(e1, xs) + valor3(e2, xs)
        case P3(e1, e2):
            return valor3(e1, xs) * valor3(e2, xs)
    assert False

# ---------------------------------------------------------------------
# Ejercicio 8.2. Definir la función
#    sumas : (Expr3) -> int
# tal que sumas(e) es el número de sumas en la expresión e. Por
# ejemplo,
#    sumas(P3(V3('z'), S3(C3(3), V3('x')))) == 1
#    sumas(S3(V3('z'), S3(C3(3), V3('x')))) == 2
#    sumas(P3(V3('z'), P3(C3(3), V3('x')))) == 0
# ---------------------------------------------------------------------

def sumas(e: Expr3) -> int:
    match e:
        case C3(_):
            return 0
        case V3(_):
            return 0
```

```
        case S3(e1, e2):
            return 1 + sumas(e1) + sumas(e2)
        case P3(e1, e2):
            return sumas(e1) + sumas(e2)
    assert False

# ----------------------------------------------------------------------
# Ejercicio 8.3. Definir la función
#    sustitucion : (Expr3, list[tuple[str, int]]) -> Expr3
# 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(P3(V3('z'), S3(C3(3), V3('x'))), [('x', 7), ('z', 9)])
#    P3(C3(9), S3(C3(3), C3(7)))
#    >>> sustitucion(P3(V3('z'), S3(C3(3), V3('y'))), [('x', 7), ('z', 9)])
#    P3(C3(9), S3(C3(3), V3('y')))
# ----------------------------------------------------------------------

def sustitucion(e: Expr3, ps: list[tuple[str, int]]) -> Expr3:
    match (e, ps):
        case(e, []):
            return e
        case (V3(c), ps):
            if c == ps[0][0]:
                return C3(ps[0][1])
            return sustitucion(V3(c), ps[1:])
        case (C3(n), _):
            return C3(n)
        case (S3(e1, e2), ps):
            return S3(sustitucion(e1, ps), sustitucion(e2, ps))
        case (P3(e1, e2), ps):
            return P3(sustitucion(e1, ps), sustitucion(e2, ps))
    assert False

# ----------------------------------------------------------------------
# Ejercicio 8.4. Definir la función
#    reducible : (Expr3) -> 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(S3(C3(3), C3(4)))              == True
#    reducible(S3(C3(3), V3('x')))            == False
#    reducible(S3(C3(3), P3(C3(4), C3(5))))   == True
#    reducible(S3(V3('x'), P3(C3(4), C3(5)))) == True
#    reducible(S3(C3(3), P3(V3('x'), C3(5)))) == False
#    reducible(C3(3))                         == False
#    reducible(V3('x'))                       == False
# ----------------------------------------------------------------------

def reducible(e: Expr3) -> bool:
    match e:
        case C3(_):
            return False
        case V3(_):
            return False
        case S3(C3(_), C3(_)):
            return True
        case S3(a, b):
            return reducible(a) or reducible(b)
        case P3(C3(_), C3(_)):
            return True
        case P3(a, b):
            return reducible(a) or reducible(b)
    assert False

# ----------------------------------------------------------------------
# Ejercicio 9. Las expresiones aritméticas generales se pueden definir
# usando el siguiente tipo de datos
#    @dataclass
#    class Expr4:
#        pass
#
#    @dataclass
#    class C4(Expr4):
#        x: int
#
#    @dataclass
#    class Y(Expr4):
#        pass
#
```

```
#    @dataclass
#    class S4(Expr4):
#        x: Expr4
#        y: Expr4
#
#    @dataclass
#    class R4(Expr4):
#        x: Expr4
#        y: Expr4
#
#    @dataclass
#    class P4(Expr4):
#        x: Expr4
#        y: Expr4
#
#    @dataclass
#    class E4(Expr4):
#        x: Expr4
#        y: int
# Por ejemplo, la expresión
#    3*y - (y+2)^7
# se puede definir por
#    R4(P4(C4(3), Y()), E4(S4(Y(), C4(2)), 7))
#
# Definir la función
#    maximo : (Expr4, 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(E4(S4(C4(10), P4(R4(C4(1), Y()), Y())), 2), list(range(-3, 4)))
#    (100, [0, 1])
# ----------------------------------------------------------------------

@dataclass
class Expr4:
    pass

@dataclass
class C4(Expr4):
    x: int
```

```
@dataclass
class Y(Expr4):
    pass

@dataclass
class S4(Expr4):
    x: Expr4
    y: Expr4

@dataclass
class R4(Expr4):
    x: Expr4
    y: Expr4

@dataclass
class P4(Expr4):
    x: Expr4
    y: Expr4

@dataclass
class E4(Expr4):
    x: Expr4
    y: int

def valor4(e: Expr4, n: int) -> int:
    match e:
        case C4(a):
            return a
        case Y():
            return n
        case S4(e1, e2):
            return valor4(e1, n) + valor4(e2, n)
        case R4(e1, e2):
            return valor4(e1, n) - valor4(e2, n)
        case P4(e1, e2):
            return valor4(e1, n) * valor4(e2, n)
        case E4(e1, m):
            return valor4(e1, n) ** m
    assert False
```

```python
def maximo(e: Expr4, ns: list[int]) -> tuple[int, list[int]]:
    m = max((valor4(e, n) for n in ns))
    return (m, [n for n in ns if valor4(e, n) == m])

# ----------------------------------------------------------------------
# Ejercicio 10. 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 Expr5:
#        pass
#
#    @dataclass
#    class C5(Expr5):
#        x: int
#
#    @dataclass
#    class Ap(Expr5):
#        o: Op
#        x: Expr5
#        y: Expr5
# Por ejemplo, la expresión
#    (7-3)+(2*5)
# se representa por
#    Ap(Op.S, Ap(Op.R, C5(7), C5(3)), Ap(Op.M, C5(2), C5(5)))
#
# Definir la función
#    valorEG : (Expr5) -> int
# tal que valorEG(e) es el valor de la expresión e. Por ejemplo,
#    >>> valorEG(Ap(Op.S, Ap(Op.R, C5(7), C5(3)), Ap(Op.M, C5(2), C5(5))))
#    14
#    >>> valorEG(Ap(Op.M, Ap(Op.R, C5(7), C5(3)), Ap(Op.S, C5(2), C5(5))))
#    28
# ----------------------------------------------------------------------

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

```
@dataclass
class Expr5:
    pass

@dataclass
class C5(Expr5):
    x: int

@dataclass
class Ap(Expr5):
    o: Op
    x: Expr5
    y: Expr5

def aplica5(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 valorEG(e: Expr5) -> int:
    match e:
        case C5(x):
            return x
        case Ap(o, e1, e2):
            return aplica5(o, valorEG(e1), valorEG(e2))
    assert False

# ------------------------------------------------------------------
# Ejercicio 11. 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)
# ----------------------------------------------------------------------

@dataclass
class ExpV:
    pass

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

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

```python
@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

6.1. El tipo abstracto de datos (TAD) de las pilas

```
# 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. Implementación del TAD de las pilas mediante listas

```
# 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()
#    >>> print(p)
#    -
#    >>> p.apila(5)
#    >>> p.apila(2)
#    >>> p.apila(3)
#    >>> p.apila(4)
#    >>> print(p)
```

```
#    4 | 3 | 2 | 5
#    >>> p.cima()
#    4
#    >>> p.desapila()
#    >>> print(p)
#    3 | 2 | 5
#    >>> p.esVacia()
#    False
#    >>> p = Pila()
#    >>> p.esVacia()
#    True
#
# Además se definen las correspondientes funciones. Por ejemplo,
#    >>> print(vacia())
#    -
#    >>> print(apila(4, apila(3, apila(2, apila(5, vacia())))))
#    4 | 3 | 2 | 5
#    >>> print(cima(apila(4, apila(3, apila(2, apila(5, vacia()))))))
#    4
#    >>> print(desapila(apila(4, apila(3, apila(2, apila(5, vacia()))))))
#    3 | 2 | 5
#    >>> print(esVacia(apila(4, apila(3, apila(2, apila(5, vacia()))))))
#    False
#    >>> print(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
```

```python
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 __str__(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.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
# ============================================

# 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. Implementación del TAD de las pilas mediante deque

```
# 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()
#    >>> print(p)
#    -
#    >>> p.apila(5)
#    >>> p.apila(2)
#    >>> p.apila(3)
#    >>> p.apila(4)
#    >>> print(p)
#    4 | 3 | 2 | 5
#    >>> p.cima()
#    4
#    >>> p.desapila()
#    >>> print(p)
#    3 | 2 | 5
#    >>> p.esVacia()
#    False
#    >>> p = Pila()
#    >>> p.esVacia()
#    True
#
# Además se definen las correspondientes funciones. Por ejemplo,
#    >>> print(vacia())
#    -
#    >>> print(apila(4, apila(3, apila(2, apila(5, vacia())))))
#    4 | 3 | 2 | 5
#    >>> print(cima(apila(4, apila(3, apila(2, apila(5, vacia())))))
#    4
#    >>> print(desapila(apila(4, apila(3, apila(2, apila(5, vacia()))))))
#    3 | 2 | 5
#    >>> print(esVacia(apila(4, apila(3, apila(2, apila(5, vacia()))))))
#    False
```

```
#    >>> print(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 __str__(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
# ==================

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. Ejercicios con el TAD de las pilas

```
# ---------------------------------------------------------------------
# Introducción                                                       --
# ---------------------------------------------------------------------

# El objetivo de esta relación de ejercicios es definir funciones sobre
# el tipo abstracto de datos de las pilas, utilizando la
# implementación del ejercicio anterior, cuyo código es
# pilaConListas.py que se encuentra en https://bit.ly/3VVt8by

# ---------------------------------------------------------------------
# Importación de librerías                                           --
# ---------------------------------------------------------------------

from copy import deepcopy
from typing import Callable, TypeVar

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

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

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

# ---------------------------------------------------------------------
# Ejercicio 1. Definir la función
#    listaApila : (list[A]) -> Pila[A]
# tal que listaApila(xs) es la pila formada por los elementos de xs.
# Por ejemplo,
#    >>> print(listaApila([3, 2, 5]))
#    5 | 2 | 3
# ---------------------------------------------------------------------

# 1ª solución
# ===========

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

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

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

# La propiedad es
@given(st.lists(st.integers()))
```

```
def test_listaApila(xs: list[int]) -> None:
    assert listaApila(xs) == listaApila2(xs)

# ---------------------------------------------------------------------
# Ejercicio 2. Definir la función
#    pilaALista : (Pila[A]) -> list[A]
# tal que pilaAlista(p) es la lista formada por los elementos de la
# lista p. Por ejemplo,
#    >>> ej = apila(5, apila(2, apila(3, vacia())))
#    >>> pilaAlista(ej)
#    [3, 2, 5]
#    >>> print(ej)
#    5 | 2 | 3
# ---------------------------------------------------------------------

# 1ª solución
# ===========

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

# 2ª solución
# ===========

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 solución
```

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

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

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

# ---------------------------------------------------------------------
# Ejercicio 3. Comprobar con Hypothesis que ambas funciones son
# inversas; es decir,
#    pilaAlista(listaApila(xs)) == xs
#    listaApila(pilaAlista(p))  == p
# ---------------------------------------------------------------------

# 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

# ---------------------------------------------------------------------
# Ejercicio 4. 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()))))
#    >>> print(filtraPila(lambda x: x % 2 == 0, ej))
#    4 | 6
#    >>> print(filtraPila(lambda x: x % 2 == 1, ej))
#    3 | 5
#    >>> print(ej)
#    3 | 4 | 6 | 5
# ----------------------------------------------------------------------

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

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()
    return r1

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

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

# 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

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

```
# Ejercicio 5. 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())))
#    >>> print(mapPila(lambda x: x + 1, ej))
#    6 | 3 | 8
#    >>> print(ej)
#    5 | 2 | 7
# ---------------------------------------------------------------------

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

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

# ---------------------------------------------------------------------
# Ejercicio 6. 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
# ----------------------------------------------------------------------

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

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

# ---------------------------------------------------------------------
# Ejercicio 7. 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
# ---------------------------------------------------------------------

# 1ª solución
# ===========

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

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

# ---------------------------------------------------------------------
# Ejercicio 8. 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
# ---------------------------------------------------------------------

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

```
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)

# 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

# ---------------------------------------------------------------------
# Ejercicio 9. 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
# ---------------------------------------------------------------------

# 1ª solución
# ===========

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

# 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)
```

```python
# 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

# ---------------------------------------------------------------------
# Ejercicio 10. 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
# ---------------------------------------------------------------------

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

# 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)
```

```python
# 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

# ---------------------------------------------------------------------
# Ejercicio 11.1. 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,
#    >>> print(ordenaInserPila(apila(4, apila(1, apila(3, vacia())))))
#    1 | 3 | 4
# ---------------------------------------------------------------------

# 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))
```

```python
# 2ª solución
# ===========

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)

# 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

# ---------------------------------------------------------------------
# Ejercicio 11.2. Comprobar con Hypothesis que la pila
# ordenaInserPila(p) está ordenada.
# ---------------------------------------------------------------------

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

# ---------------------------------------------------------------------
# Ejercicio 12. 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()))))
#    >>> print(ej)
#    3 | 1 | 3 | 5
#    >>> print(nubPila1(ej))
#    1 | 3 | 5
# ---------------------------------------------------------------------

# 1ª solución
# ===========

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

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

# ---------------------------------------------------------------------
# Ejercicio 13. 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
# ---------------------------------------------------------------------
```

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

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

# ------------------------------------------------------------------------
# Comprobación de las propiedades
# ------------------------------------------------------------------------

# La comprobación es
#    src> poetry run pytest -v el_TAD_de_las_pilas.py
#       test_listaApila PASSED
#       test_pilaAlista PASSED
#       test_1_listaApila PASSED
#       test_2_listaApila PASSED
#       test_filtraPila PASSED
#       test_mapPila PASSED
#       test_pertenecePila PASSED
#       test_contenidaPila PASSED
#       test_prefijoPila PASSED
#       test_subPila PASSED
#       test_ordenadaPila PASSED
#       test_ordenaInserPila PASSED
#       test_ordenadaOrdenaInserPila PASSED
#       test_nubPila PASSED
#       test_maxPila PASSED
#   15 passed in 2.91s
```

Capítulo 7

El tipo abstracto de datos de las colas

7.1. El tipo abstracto de datos (TAD) de las colas

```
# 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, colaAleatoria, esVacia, inserta,
                                  primero, resto, vacia)
```

7.2. Implementación del TAD de las colas mediante listas

```
# 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()
#    >>> print(c)
#    -
```

```
#    >>> c.inserta(5)
#    >>> c.inserta(2)
#    >>> c.inserta(3)
#    >>> c.inserta(4)
#    >>> print(c)
#    5 | 2 | 3 | 4
#    >>> c.primero()
#    5
#    >>> c.resto()
#    >>> print(c)
#    2 | 3 | 4
#    >>> c.esVacia()
#    False
#    >>> c = Cola()
#    >>> c.esVacia()
#    True
#
# Además se definen las correspondientes funciones. Por ejemplo,
#    >>> print(vacia())
#    -
#    >>> print(inserta(4, inserta(3, inserta(2, inserta(5, vacia())))))
#    5 | 2 | 3 | 4
#    >>> primero(inserta(4, inserta(3, inserta(2, inserta(5, vacia())))))
#    5
#    >>> print(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 __str__(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:
    """
    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. Implementación del TAD de las colas mediante dos listas

```
# 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()
#    >>> print(c)
#    -
#    >>> c.inserta(5)
#    >>> c.inserta(2)
#    >>> c.inserta(3)
#    >>> c.inserta(4)
#    >>> print(c)
#    5 | 2 | 3 | 4
#    >>> c.primero()
#    5
#    >>> c.resto()
#    >>> print(c)
#    2 | 3 | 4
#    >>> c.esVacia()
#    False
#    >>> c = Cola()
#    >>> c.esVacia()
#    True
#
# Además se definen las correspondientes funciones. Por ejemplo,
```

```
#    >>> print(vacia())
#    -
#    >>> print(inserta(4, inserta(3, inserta(2, inserta(5, vacia()))))
#    5 | 2 | 3 | 4
#    >>> primero(inserta(4, inserta(3, inserta(2, inserta(5, vacia()))))
#    5
#    >>> print(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 __str__(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. Implementación del TAD de las colas mediante deque

```
# 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()
#    >>> print(c)
#    -
#    >>> c.inserta(5)
```

```
#    >>> c.inserta(2)
#    >>> c.inserta(3)
#    >>> c.inserta(4)
#    >>> print(c)
#    5 | 2 | 3 | 4
#    >>> c.primero()
#    5
#    >>> c.resto()
#    >>> print(c)
#    2 | 3 | 4
#    >>> c.esVacia()
#    False
#    >>> c = Cola()
#    >>> c.esVacia()
#    True
#
# Además se definen las correspondientes funciones. Por ejemplo,
#    >>> print(vacia())
#    -
#    >>> print(inserta(4, inserta(3, inserta(2, inserta(5, vacia())))))
#    5 | 2 | 3 | 4
#    >>> primero(inserta(4, inserta(3, inserta(2, inserta(5, vacia())))))
#    5
#    >>> print(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 __str__(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
# ==================

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. Ejercicios con el TAD de las colas

```
# -----------------------------------------------------------------------
# Introducción                                                          --
# -----------------------------------------------------------------------

# El objetivo de esta relación de ejercicios es definir funciones sobre
# el TAD de las colas, utilizando sus implementaciones estudiadas en las
# secciones anteriores.

# -----------------------------------------------------------------------
# Importación de librerías                                              --
# -----------------------------------------------------------------------

from copy import deepcopy
from functools import reduce
from typing import Callable, TypeVar

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

from src.TAD.cola import (Cola, colaAleatoria, esVacia, inserta, primero,
                          resto, vacia)

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

```python
# ---------------------------------------------------------------------
# Ejercicio 1.1 Definir la función
#    listaAcola : (list[A]) -> Cola[A]
# tal que listaAcola(xs) es la cola formada por los elementos de xs. Por
# ejemplo,
#    >>> print(listaAcola([3, 2, 5]))
#    3 | 2 | 5
# ---------------------------------------------------------------------

# 1ª solución
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
def listaAcola2(xs: list[A]) -> Cola[A]:
    p: Cola[A] = Cola()
    for x in xs:
        p.inserta(x)
    return p

# Comprobación de equivalencia
@given(st.lists(st.integers()))
def test_listaAcola(xs: list[int]) -> None:
    assert listaAcola(xs) == listaAcola2(xs)

# ---------------------------------------------------------------------
# Ejercicio 1.2. Definir la función
#    colaAlista : (Cola[A]) -> list[A]
# tal que 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]
#    >>> print(ej)
#    3 | 2 | 5
```

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

# 1ª solución
def colaAlista(c: Cola[A]) -> list[A]:
    if esVacia(c):
        return []
    pc = primero(c)
    rc = resto(c)
    return [pc] + colaAlista(rc)

# 2ª solución
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)

# 3ª solución
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
@given(p=colaAleatoria())
def test_colaAlista(p: Cola[int]) -> None:
    assert colaAlista(p) == colaAlista2(p)
    assert colaAlista(p) == colaAlista3(p)
```

```
# ---------------------------------------------------------------------
# Ejercicio 1.3. Comprobar con Hypothesis que ambas funciones son
# inversas; es decir,
#    colaAlista(listaAcola(xs)) == xs
#    listaAcola(colaAlista(c))  == c
# ---------------------------------------------------------------------

@given(st.lists(st.integers()))
def test_1_listaAcola(xs: list[int]) -> None:
    assert colaAlista(listaAcola(xs)) == xs

@given(c=colaAleatoria())
def test_2_listaAcola(c: Cola[int]) -> None:
    assert listaAcola(colaAlista(c)) == c

# ---------------------------------------------------------------------
# Ejercicio 2. 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
# ---------------------------------------------------------------------

# 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
def ultimoCola5(c: Cola[A]) -> A:
    if esVacia(c):
        raise ValueError("cola vacia")
    return colaAlista(c)[-1]

# Comprobación de equivalencia
```

```
@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

# ---------------------------------------------------------------------
# Ejercicio 3. 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
# ---------------------------------------------------------------------

# 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
@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

# ---------------------------------------------------------------------
# Ejercicio 4. 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
# ----------------------------------------------------------------------

# 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))

# 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
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
@given(c=colaAleatoria())
def test_todosVerifican(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

# ---------------------------------------------------------------------
# Ejercicio 5. 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
# ---------------------------------------------------------------------

# 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))

# 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
```

```
@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

# ---------------------------------------------------------------------
# Ejercicio 6. 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())))
#    >>> print(ej1)
#    2 | 3
#    >>> print(ej2)
#    4 | 3 | 5
#    >>> print(extiendeCola(ej1, ej2))
#    2 | 3 | 4 | 3 | 5
#    >>> print(extiendeCola(ej2, ej1))
#    4 | 3 | 5 | 2 | 3
# ---------------------------------------------------------------------

# 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))

# 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

def extiendeCola5(c1: Cola[A], c2: Cola[A]) -> Cola[A]:
    _c1 = deepcopy(c1)
    _c2 = deepcopy(c2)
    return extiendeCola5Aux(_c1, _c2)

# Comprobación de equivalencia
@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

# ---------------------------------------------------------------------
# Ejercicio 7. 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()))))
#    >>> print(intercalaColas(ej1, ej2))
#    5 | 9 | 3 | 4 | 7 | 0
#    >>> print(intercalaColas(ej2, ej1))
#    9 | 5 | 4 | 3 | 7 | 0
# ---------------------------------------------------------------------

# 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))

# 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
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
@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

# ------------------------------------------------------------------
# Ejercicio 8. 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()))))
#    >>> print(agrupaColas([ej1]))
#    5 | 2
#    >>> print(agrupaColas([ej1, ej2]))
#    5 | 4 | 2 | 7 | 3
#    >>> print(agrupaColas([ej1, ej2, ej3]))
#    5 | 6 | 4 | 1 | 2 | 0 | 7 | 9 | 3
# ------------------------------------------------------------------

# 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:])

# 2ª solución
def agrupaColas2(cs: list[Cola[A]]) -> Cola[A]:
    return reduce(intercalaColas, cs, vacia())

# Comprobación de equivalencia
@given(st.lists(colaAleatoria(), max_size=4))
def test_agrupaCola(cs: list[Cola[int]]) -> None:
    assert agrupaColas1(cs) == agrupaColas2(cs)

# --------------------------------------------------------------------
# Ejercicio 9. 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
# --------------------------------------------------------------------

# 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)

# 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
@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

# ---------------------------------------------------------------------
# Ejercicio 10. 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
#    >>> contenidaCola(ej2, ej3)
```

```
#    False
# ----------------------------------------------------------------------

# 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)

# 2ª solución
def contenidaCola2(c1: Cola[A], c2: Cola[A]) -> bool:
    return set(colaAlista(c1)) <= set(colaAlista(c2))

# 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
@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

# ---------------------------------------------------------------------
# Ejercicio 11. 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
# ---------------------------------------------------------------------

# 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))

# 3ª solución
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
@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

# --------------------------------------------------------------------
# Ejercicio 12. 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
# ---------------------------------------------------------------------

# 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)

# 2ª solución
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))

# 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
@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

# --------------------------------------------------------------------
# Ejercicio 13. 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
# --------------------------------------------------------------------

# 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
def ordenadaLista(xs: list[A]) -> bool:
    return all((x <= y for (x, y) in zip(xs, xs[1:])))
```

```python
def ordenadaCola2(p: Cola[A]) -> bool:
    return ordenadaLista(colaAlista(p))

# 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
@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

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

```
# Ejercicio 14. 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
# ----------------------------------------------------------------------

# 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
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
@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

# ---------------------------------------------------------------------
# Comprobaciones
# ---------------------------------------------------------------------

# Las comprobación de las propiedades es
#    > poetry run pytest -v el_TAD_de_las_colas.py
#         test_listaAcola PASSED
#         test_colaAlista PASSED
#         test_1_listaAcola PASSED
#         test_2_listaAcola PASSED
#         test_ultimoCola PASSED
#         test_longitudCola_ PASSED
#         test_todosVerifican PASSED
#         test_algunoVerifica PASSED
#         test_extiendeCola PASSED
#         test_intercalaCola PASSED
#         test_agrupaCola PASSED
#         test_perteneceCola PASSED
#         test_contenidaCola PASSED
#         test_prefijoCola PASSED
#         test_subCola PASSED
#         test_ordenadaCola PASSED
#         test_maxCola PASSED
```

Capítulo 8

El tipo abstracto de datos de las colas de prioridad

8.1. El tipo abstracto de datos (TAD) de las colas de prioridad

```
# 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)
```

8.2. Implementación del TAD de las colas de prioridad mediante listas

```
# 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',
]
```

```python
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]:
    """
    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
```

Capítulo 9

El tipo abstracto de datos de los conjuntos

9.1. El tipo abstracto de datos (TAD) de los conjuntos

```
# 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)
```

9.2. Implementación del TAD de los conjuntos mediante listas no ordenadas con duplicados

```
# 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.
```

```python
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
```

9.3. Implementación del TAD de los conjuntos mediante listas no ordenadas sin duplicados

```
# 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 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))

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
```

9.4. Implementación del TAD de los conjuntos mediante listas ordenadas sin duplicados

```
# 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

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

```python
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
```

9.5. Implementación del TAD de los conjuntos mediante librería

```
# 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'
]
```

```python
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
```

9.6. Operaciones con conjuntos

```
# ======================================================================
# Introducción
# ======================================================================

# El objetivo de esta relación de ejercicios es definir operaciones
# entre conjuntos utilizando el tipo abstracto de datos de los
# conjuntos.

# ======================================================================
# Librerías auxiliares
# ======================================================================

from __future__ import annotations

from abc import abstractmethod
from copy import deepcopy
from functools import reduce
from typing import Callable, Protocol, TypeVar
```

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

from src.TAD.conjunto import (Conj, conjuntoAleatorio, elimina, esVacio,
                              inserta, menor, pertenece, vacio)


class Comparable(Protocol):
    @abstractmethod
    def __lt__(self: A, otro: A) -> bool:
        pass

A = TypeVar('A', bound=Comparable)
B = TypeVar('B', bound=Comparable)

# ============================================================================
# Ejercicios
# ============================================================================

# ----------------------------------------------------------------------
# Ejercicio 1. Definir la función
#    listaAconjunto : (list[A]) -> Conj[A]
# tal que
# listaAconjunto(xs) es el conjunto formado por los elementos de xs.
# Por ejemplo,
#    >>> listaAconjunto([3, 2, 5])
#    {2, 3, 5}
# ----------------------------------------------------------------------

# 1ª solución
# ===========

def listaAconjunto(xs: list[A]) -> Conj[A]:
    if not xs:
        return vacio()
    return inserta(xs[0], listaAconjunto(xs[1:]))

# 2ª solución
# ===========
```

```
def listaAconjunto2(xs: list[A]) -> Conj[A]:
    return reduce(lambda ys, y: inserta(y, ys), xs, vacio())

# 3ª solución
# ===========

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

# ---------------------------------------------------------------------
# Ejercicio 2. Definir la función
#    conjuntoAlista : (Conj[A]) -> list[A]
# tal que 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]
# ---------------------------------------------------------------------

# 1ª solución
# ===========

def conjuntoAlista(c: Conj[A]) -> list[A]:
    if esVacio(c):
        return []
    mc = menor(c)
    rc = elimina(mc, c)
```

```
    return [mc] + conjuntoAlista(rc)

# 2ª solución
# ===========

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 solución
# ===========

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

@given(c=conjuntoAleatorio())
def test_conjuntoAlista(c: Conj[int]) -> None:
    r = conjuntoAlista(c)
    assert conjuntoAlista2(c) == r
    assert conjuntoAlista3(c) == r
```

```
# ---------------------------------------------------------------------
# Ejercicio 3. Comprobar con Hypothesis que ambas funciones son inversa;
# es decir,
#    conjuntoAlista (listaAconjunto xs) = sorted(list(set(xs)))
#    listaAconjunto (conjuntoAlista c)  = c
# ---------------------------------------------------------------------

# 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

# ---------------------------------------------------------------------
# Ejercicio 4. 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
# ---------------------------------------------------------------------

# 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)))

# 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

# ---------------------------------------------------------------------
# Ejercicio 5. 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
# ---------------------------------------------------------------------

def subconjuntoPropio(c1: Conj[A], c2: Conj[A]) -> bool:
    return subconjunto(c1, c2) and c1 != c2
```

```
# ---------------------------------------------------------------------
# Ejercicio 6. Definir la función
#    unitario :: Ord a => a -> Conj a
# tal que (unitario x) es el conjunto {x}. Por ejemplo,
#    unitario 5 == {5}
# ---------------------------------------------------------------------

def unitario(x: A) -> Conj[A]:
    return inserta(x, vacio())

# ---------------------------------------------------------------------
# Ejercicio 7. 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
# ---------------------------------------------------------------------

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

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

# ---------------------------------------------------------------------
# Ejercicio 8. 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}
# ---------------------------------------------------------------------

# 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)

# 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

# ---------------------------------------------------------------------
# Ejercicio 9. 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}
# ---------------------------------------------------------------------

# 1ª solución
# ===========

def unionG(cs: list[Conj[A]]) -> Conj[A]:
    if not cs:
        return vacio()
    return union(cs[0], unionG(cs[1:]))

# 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_unionG(cs: list[Conj[int]]) -> None:
    r = unionG(cs)
    assert unionG2(cs) == r
    assert unionG3(cs) == r

# ---------------------------------------------------------------------
# Ejercicio 10. 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}
# ---------------------------------------------------------------------

# 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)])

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

# 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

# ---------------------------------------------------------------------
# Ejercicio 11. 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())))
#    >>> interseccionG([ej1, ej2, ej3])
#    {2, 5}
# ---------------------------------------------------------------------

# 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

# ---------------------------------------------------------------------
# Ejercicio 12. 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
# ---------------------------------------------------------------------

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

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))

# 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

# ---------------------------------------------------------------------
# Ejercicio 13. 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)
#    {}
# ---------------------------------------------------------------------

# 1ª solución
# ===========

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)])

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

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

# ---------------------------------------------------------------------
# Ejercicio 14. 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}
# ---------------------------------------------------------------------

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

# 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

# ---------------------------------------------------------------------
# Ejercicio 15. 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)
#    {5, 7}
# ---------------------------------------------------------------------

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

# La propiedad es
@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

# ---------------------------------------------------------------------
# Ejercicio 16. 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()))))
#    >>> particion(lambda x: x % 2 == 0, ej)
#    ({2, 4}, {5, 7})
# ---------------------------------------------------------------------

# 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))

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

# 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

# ---------------------------------------------------------------------
# Ejercicio 17. 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})
# ---------------------------------------------------------------------

# 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)

# 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)

def divide4(x: A, c: Conj[A]) -> tuple[Conj[A], Conj[A]]:
    _c = deepcopy(c)
    return divide4Aux(x, _c)

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

# La propiedad es
@given(x=st.integers(), c=conjuntoAleatorio())
def test_divide(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

# ---------------------------------------------------------------------
# Ejercicio 18. 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}
# ---------------------------------------------------------------------

# 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))

# 2ª solución
# ===========

def mapC2(f: Callable[[A], B], c: Conj[A]) -> Conj[B]:
    return listaAconjunto(list(map(f, conjuntoAlista(c))))

# 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
```

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

# 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

# ---------------------------------------------------------------------
# Ejercicio 19. 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
# ---------------------------------------------------------------------
```

```python
# 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))

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

# 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

# ---------------------------------------------------------------------
# Ejercicio 20. 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())))
#    True
#    >>> algunos(lambda x: x % 2 == 0, inserta(3, inserta(7, vacio())))
#    False
# ---------------------------------------------------------------------

# 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))

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

# 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

# ---------------------------------------------------------------------
# Ejercicio 21. 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)}
# ---------------------------------------------------------------------

# 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))

# 2ª solución
# ===========
```

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

```python
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 -v operaciones_con_conjuntos.py
```

Capítulo 10

Relaciones binarias homogéneas

10.1. Relaciones binarias homogéneas

```
# ------------------------------------------------------------------
# Introducción                                                     --
# ------------------------------------------------------------------

# El objetivo de esta relación de ejercicios es definir propiedades y
# operaciones sobre las relaciones binarias (homogéneas).
#
# Como referencia se puede usar el artículo de la wikipedia
# http://bit.ly/HVHOPS

# ------------------------------------------------------------------
# § Librerías auxiliares                                           --
# ------------------------------------------------------------------

from random import randint, sample
from sys import setrecursionlimit
from timeit import Timer, default_timer
from typing import TypeVar

from hypothesis import given
from hypothesis import strategies as st

A = TypeVar('A')

setrecursionlimit(10**6)
```

```python
# ---------------------------------------------------------------------
# Ejercicio 1. 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.
# ---------------------------------------------------------------------

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]
```

```python
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 está al final
#    > poetry run pytest -q Relaciones_binarias.py
#    1 passed in 0.14s

# ---------------------------------------------------------------------
# Ejercicio 2. Definir la función
#    universo : (Rel[A]) -> list[A]
# tal que universo(r) es el universo de la relación r. Por ejemplo,
#    >>> universo(([3, 2, 5], [(2, 3), (3, 5)]))
#    [3, 2, 5]
# ---------------------------------------------------------------------
```

```
def universo(r: Rel[A]) -> list[A]:
    return r[0]

# ---------------------------------------------------------------------
# Ejercicio 3. Definir la función
#    grafo    : (Rel[A]) -> list[tuple[A, A]]
# tal que grafo(r) es el grafo de la relación r. Por ejemplo,
#    >>> grafo(([3, 2, 5], [(2, 3), (3, 5)]))
#    [(2, 3), (3, 5)]
# ---------------------------------------------------------------------

def grafo(r: Rel[A]) -> list[tuple[A, A]]:
    return r[1]

# ---------------------------------------------------------------------
# Ejercicio 4. 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
# ---------------------------------------------------------------------

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

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

# ---------------------------------------------------------------------
# Ejercicio 5. 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
# ---------------------------------------------------------------------

# 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

# ---------------------------------------------------------------------
# Ejercicio 6. Definir la función
#    subconjunto : (list[A], list[A]) -> bool
```

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

# 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

# ---------------------------------------------------------------------
# Ejercicio 7. 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)])
# ---------------------------------------------------------------------

# 1ª solución
```

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

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_composicion(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

# ---------------------------------------------------------------------
# Ejercicio 8. 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
# ---------------------------------------------------------------------

# 1ª solución
# ===========

def transitiva1(r: Rel[A]) -> bool:
    g = grafo(r)
    return subconjunto(grafo(composicion(r, r)), g)

# 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_transitiva(n: int) -> None:
    r = relacionArbitraria(n)
    res = transitiva1(r)
    assert transitiva2(r) == res
    assert transitiva3(r) == res

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

# 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

# ---------------------------------------------------------------------
# Ejercicio 9. Definir la función
#    esEquivalencia : (Rel[A]) -> bool
# tal que esEquivalencia(r) se verifica si la relación r es de
# equivalencia. Por ejemplo,
#    >>> esEquivalencia (([1,3,5],[(1,1),(1,3),(3,1),(3,3),(5,5)]))
#    True
#    >>> esEquivalencia (([1,2,3,5],[(1,1),(1,3),(3,1),(3,3),(5,5)]))
#    False
#    >>> esEquivalencia (([1,3,5],[(1,1),(1,3),(3,3),(5,5)]))
#    False
# ---------------------------------------------------------------------

def esEquivalencia(r: Rel[A]) -> bool:
    return reflexiva(r) and simetrica(r) and transitiva(r)

# ---------------------------------------------------------------------
# Ejercicio 10. 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
# ---------------------------------------------------------------------

# 1ª solución
# ===========

def irreflexiva(r: Rel[A]) -> bool:
    (u, g) = r
    return all(((x, x) not in g for x in u))

# 2ª solución
# ===========
```

```python
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

# ---------------------------------------------------------------------
# Ejercicio 11. 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
# ---------------------------------------------------------------------

# 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
@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

# ---------------------------------------------------------------------
# Ejercicio 12. 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
# ---------------------------------------------------------------------

# 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

# ---------------------------------------------------------------------
# § Clausuras                                                       --
# ---------------------------------------------------------------------

# ---------------------------------------------------------------------
# Ejercicio 13. 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)])
# ----------------------------------------------------------------------

def clausuraReflexiva(r: Rel[A]) -> Rel[A]:
    (u, g) = r
    return (u, list(set(g) | {(x, x) for x in u}))

# ----------------------------------------------------------------------
# Ejercicio 14. 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)])
# ----------------------------------------------------------------------

def clausuraSimetrica(r: Rel[A]) -> Rel[A]:
    (u, g) = r
    return (u, list(set(g) | {(y, x) for (x,y) in g}))

# ----------------------------------------------------------------------
# Ejercicio 15. Comprobar con Hipothesis que clausuraSimetrica es
# simétrica.
# ----------------------------------------------------------------------

# La propiedad es
@given(st.integers(min_value=0, max_value=10))
def test_clausuraSimetrica(n: int) -> None:
    r = relacionArbitraria(n)
    assert simetrica(clausuraSimetrica(r))

# ----------------------------------------------------------------------
# Ejercicio 16. 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)])
# ----------------------------------------------------------------------
```

```python
# 1ª solución
# ===========

def clausuraTransitiva(r: Rel[A]) -> Rel[A]:
    (u, g) = r

    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 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)

# ---------------------------------------------------------------------
# Ejercicio 17. Comprobar con QuickCheck que clausuraTransitiva es
# transitiva.
# ---------------------------------------------------------------------

# 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 comprobación de las propiedades es
#    > poetry run pytest -q relaciones_binarias_homogeneas.py
#    12 passed in 0.66s
```

Capítulo 11

El tipo abstracto de datos de los polinomios

11.1. El tipo abstracto de datos (TAD) de los polinomios

```
# 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)
```

11.2. Implementación del TAD de los polinomios mediante listas densas

```
# 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 ===
```

11.3. Implementación del TAD de los polinomios mediante listas dispersas

```
# 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 término 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'
]
```

```python
from dataclasses import dataclass, field
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()

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 ===
```

11.4. Operaciones con el tipo abstracto de datos de los polinomios

```
# ------------------------------------------------------------------
# Introducción                                                     --
# ------------------------------------------------------------------

# El objetivo de esta relación es ampliar el conjunto de operaciones
# sobre polinomios definidas utilizando las implementaciones de los
# ejercicios anteriores-

# ------------------------------------------------------------------
# Importación de librerías                                         --
# ------------------------------------------------------------------

from functools import reduce
from itertools import dropwhile
from sys import setrecursionlimit
from typing import TypeVar

from hypothesis import given
from hypothesis import strategies as st

from src.TAD.Polinomio import (Polinomio, coefLider, consPol, esPolCero, grado,
                               polCero, polinomioAleatorio, restoPol)

setrecursionlimit(10**6)

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

# ------------------------------------------------------------------
# Ejercicio 1. Definir la función
#    densaAdispersa : (list[A]) -> list[tuple[int, A]]
```

```python
# tal 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)]
# ---------------------------------------------------------------------

# 1ª solución
# ===========

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 solución
# ===========

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 solución
# ===========

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

```
# 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

# La comprobación es
#    >>> test_densaADispersa()
#    >>>

# ---------------------------------------------------------------------
# Ejercicio 2. Definir la función
#    dispersaAdensa : (list[tuple[int, A]]) -> list[A]
# tal que 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]
# ---------------------------------------------------------------------

# 1ª solución
# ===========
```

```python
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 solución
# ===========

# coeficienteDensa(ps, n) es el coeficiente del término de grado n en el
# polinomio cuya representación densa es ps. Por ejemplo,
#    coeficienteDensa([(6, 9), (3, 5), (1, 4), (0, 7)], 3)  ==  5
#    coeficienteDensa([(6, 9), (3, 5), (1, 4), (0, 7)], 4)  ==  0
def coeficienteDensa(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 coeficienteDensa(ps[1:], n)

def dispersaAdensa2(ps: list[tuple[int, A]]) -> list[A]:
    if not ps:
        return []
    n = ps[0][0]
    return [coeficienteDensa(ps, m) for m in range(n, -1, -1)]

# 3ª solución
# ===========

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

# 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

# La comprobación es
#    >>> test_dispersaAdensa()
#    >>>

# ----------------------------------------------------------------------
# Ejercicio 3. Comprobar con Hypothesis que las funciones densaAdispersa
# y dispersaAdensa son inversas.
```

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

# La primera propiedad es
@given(xs=densaAleatoria())
def test_dispersaAdensa_densaAdispersa(xs: list[int]) -> None:
    assert dispersaAdensa(densaAdispersa(xs)) == xs

# La comprobación es
#    >>> test_dispersaAdensa_densaAdispersa()
#    >>>

# 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
#    >>> test_densaAdispersa_dispersaAdensa()
#    >>>

# ---------------------------------------------------------------------
# Ejercicio 4. Definir la función
#    dispersaApolinomio : (list[tuple[int, A]]) -> Polinomio[A]
# tal 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
# ---------------------------------------------------------------------

# 1ª solución
# ===========

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 solución
# ===========
```

```python
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

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

# La propiedad es
@given(ps=dispersaAleatoria())
def test_dispersaApolinomio(ps: list[tuple[int, int]]) -> None:
    assert dispersaApolinomio(ps) == dispersaApolinomio2(ps)

# La comprobación es
#    >>> test_dispersaApolinomio()
#    >>>

# ---------------------------------------------------------------------
# Ejercicio 5. Definir la función
#    polinomioAdispersa : (Polinomio[A]) -> list[tuple[int, A]]
# tal 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)]
# ---------------------------------------------------------------------

def polinomioAdispersa(p: Polinomio[A]) -> list[tuple[int, A]]:
    if esPolCero(p):
        return []
    return [(grado(p), coefLider(p))] + polinomioAdispersa(restoPol(p))

# ---------------------------------------------------------------------
# Ejercicio 6. Comprobar con Hypothesis que ambas funciones son
# inversas.
```

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

# La primera propiedad es
@given(ps=dispersaAleatoria())
def test_polinomioAdispersa_dispersaApolinomio(ps: list[tuple[int,
                                                              int]]) -> None:
    assert polinomioAdispersa(dispersaApolinomio(ps)) == ps

# La comprobación es
#    >>> test_polinomioAdispersa_dispersaApolinomio()
#    >>>

# La segunda propiedad es
@given(p=polinomioAleatorio())
def test_dispersaApolinomio_polinomioAdispersa(p: Polinomio[int]) -> None:
    assert dispersaApolinomio(polinomioAdispersa(p)) == p

# La comprobación es
#    >>> test_dispersaApolinomio_polinomioAdispersa()
#    >>>

# ---------------------------------------------------------------------
# Ejercicio 7. 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
# ---------------------------------------------------------------------

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))

# ---------------------------------------------------------------------
# Ejercicio 8. Definir la función
#    densaApolinomio : (list[A]) -> Polinomio[A]
# tal 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
# ---------------------------------------------------------------------

# 1ª solución
# ===========

def densaApolinomio(xs: list[A]) -> Polinomio[A]:
    if not xs:
        return polCero()
    return consPol(len(xs[1:]), xs[0], densaApolinomio(xs[1:]))

# 2ª solución
# ===========

def densaApolinomio2(xs: list[A]) -> Polinomio[A]:
    return dispersaApolinomio(densaAdispersa(xs))

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

# La propiedad es
@given(xs=densaAleatoria())
def test_densaApolinomio(xs: list[int]) -> None:
    assert densaApolinomio(xs) == densaApolinomio2(xs)

# La comprobación es
#    >>> test_densaApolinomio()
#    >>>

# ---------------------------------------------------------------------
# Ejercicio 9. Definir la función
#    polinomioAdensa : (Polinomio[A]) -> list[A]
```

```
# tal que polinomioAdensa(c) es la representación densa del polinomio
# p. Por ejemplo,
#    >>> ejPol = consPol(6, 9, consPol(3, 5, consPol(1, 4, consPol(0, 7, polCero(
#    >>> ejPol
#    9*x^6 + 5*x^3 + 4*x + 7
#    >>> polinomioAdensa(ejPol)
#    [9, 0, 0, 5, 0, 4, 7]
# ---------------------------------------------------------------------

# 1ª solución
# ===========

def polinomioAdensa(p: Polinomio[A]) -> list[A]:
    if esPolCero(p):
        return []
    n = grado(p)
    return [coeficiente(k, p) for k in range(n, -1, -1)]

# 2ª solución
# ===========

def polinomioAdensa2(p: Polinomio[A]) -> list[A]:
    return dispersaAdensa(polinomioAdispersa(p))

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

# La propiedad es
@given(p=polinomioAleatorio())
def test_polinomioAdensa(p: Polinomio[int]) -> None:
    assert polinomioAdensa(p) == polinomioAdensa2(p)

# La comprobación es
#    >>> test_polinomioAdensa()
#    >>>

# ---------------------------------------------------------------------
# Ejercicio 10. Comprobar con Hypothesis que ambas funciones son
# inversas.
# ---------------------------------------------------------------------
```

```python
# La primera propiedad es
@given(xs=densaAleatoria())
def test_polinomioAdensa_densaApolinomio(xs: list[int]) -> None:
    assert polinomioAdensa(densaApolinomio(xs)) == xs

# La comprobacion es
#    >>> test_polinomioAdensa_densaApolinomio()
#    >>>

# La segunda propiedad es
@given(p=polinomioAleatorio())
def test_densaApolinomio_polinomioAdensa(p: Polinomio[int]) -> None:
    assert densaApolinomio(polinomioAdensa(p)) == p

# La comprobación es
#    >>> test_densaApolinomio_polinomioAdensa()
#    >>>

# ---------------------------------------------------------------------
# Ejercicio 11. 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
# ---------------------------------------------------------------------

# 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
#    >>> test_creaTermino()
#    >>>

# ---------------------------------------------------------------------
# Ejercicio 12. 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
# ---------------------------------------------------------------------

# 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())

# 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
#    >>> test_termLider()
#    >>>

# ---------------------------------------------------------------------
# Ejercicio 13. 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.
# ---------------------------------------------------------------------

# 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)

# La comprobación es
#    >>> test_sumaPol()
#    >>>

# ----------------------------------------------------------------------
# Ejercicio 14. Comprobar con Hypothesis las siguientes propiedades:
# + polCero es el elemento neutro de la suma.
# + la suma es conmutativa.
# ----------------------------------------------------------------------

# 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 comprobación es
#    >>> test_neutroSumaPol()
#    >>>

# La suma es conmutativa.
@given(p=polinomioAleatorio(), q=polinomioAleatorio())
def test_conmutativaSuma(p: Polinomio[int], q: Polinomio[int]) -> None:
    p1 = p
    q1 = q
    assert sumaPol(p, q) == sumaPol(q1, p1)

# La comprobación es
#    >>> test_conmutativaSuma()
#    >>>

# ---------------------------------------------------------------------
# Ejercicio 15. 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
# ---------------------------------------------------------------------

# 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))

# ---------------------------------------------------------------------
# Ejercicio 16. Comprobar con Hypothesis las siguientes propiedades
# + El producto de polinomios es conmutativo.
# + El producto es distributivo respecto de la suma.
# ---------------------------------------------------------------------

# El producto de polinomios es conmutativo.
@given(p=polinomioAleatorio(),
       q=polinomioAleatorio())
def test_conmutativaProducto(p: Polinomio[int], q: Polinomio[int]) -> None:
    p1 = p
    q1 = q
    assert multPol(p, q) == multPol(q1, p1)

# La comprobación es
#    >>> test_conmutativaProducto()
#    >>>

# 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
#    >>> test_distributivaProductoSuma()
#    >>>

# ----------------------------------------------------------------------
# Ejercicio 17. 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
# ----------------------------------------------------------------------

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)

# ----------------------------------------------------------------------
# Ejercicio 18. 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
# ----------------------------------------------------------------------

def esRaiz(c: A, p: Polinomio[A]) -> bool:
    return valor(p, c) == 0

# ----------------------------------------------------------------------
# Ejercicio 19. 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
#    x^5 + 5*x^2 + 4*x
#    >>> derivada(ejPol)
#    5*x^4 + 10*x + 4
# ----------------------------------------------------------------------

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))

# ----------------------------------------------------------------------
# Ejercicio 20. Comprobar con Hypothesis que la derivada de la suma es
# la suma de las derivadas.
# ----------------------------------------------------------------------

# La propiedad es
@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
#    >>> test_derivada()
#    >>>

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

```
# Ejercicio 21. 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
#    >>> restaPol(ejPol1, ejPol2)
#    x^5 + 2*x^4 + 6
# ---------------------------------------------------------------------

def restaPol(p: Polinomio[A], q: Polinomio[A]) -> Polinomio[A]:
    return sumaPol(p, multPorTerm(creaTermino(0, -1), q))

# ---------------------------------------------------------------------
# Ejercicio 22. 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
# ---------------------------------------------------------------------

# 1ª solución
# ===========

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
```

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

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
#    >>> test_potencia()
#    >>>

# ---------------------------------------------------------------------
# Ejercicio 23. 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
# ----------------------------------------------------------------------

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))

# ----------------------------------------------------------------------
# Ejercicio 24. 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
# ----------------------------------------------------------------------

def integralDef(p: Polinomio[float], a: float, b: float) -> float:
    q = integral(p)
    return valor(q, b) - valor(q, a)

# ----------------------------------------------------------------------
# Ejercicio 25. 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
# ---------------------------------------------------------------------

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))

# ---------------------------------------------------------------------
# Ejercicio 26. Definir la función
#    cociente : (Polinomio[float], Polinomio[float]) -> Polinomio[float]
# tal 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, polCero(
#    >>> 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
# ---------------------------------------------------------------------

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))
```

```
# ---------------------------------------------------------------------
# Ejercicio 27. Definir la función
#    resto : (Polinomio[float], Polinomio[float]) -> Polinomio[float]
# tal que resto(p, q) es el resto de la división de p entre q. Por ejemplo,
#    >>> resto(pol1, pol2)
#    1.0*x + 4
# ---------------------------------------------------------------------

def resto(p: Polinomio[float], q: Polinomio[float]) -> Polinomio[float]:
    return restaPol(p, multPol(cociente(p, q), q))

# ---------------------------------------------------------------------
# Ejercicio 28. 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
#    6*x^2 + 2*x
#    >>> divisiblePol(pol1, pol2)
#    True
#    >>> divisiblePol(pol1, pol3)
#    False
# ---------------------------------------------------------------------

def divisiblePol(p: Polinomio[float], q: Polinomio[float]) -> bool:
    return esPolCero(resto(p, q))

# ---------------------------------------------------------------------
# Ejercicio 29. 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)
# ----------------------------------------------------------------------

# 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)

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

# La comprobación es
#    > poetry run pytest El_TAD_de_polinomios_operaciones.py
#    ======= 20 passed in 5.51s =======
```

11.5. División y factorización de polinomios mediante la regla de Ruffini

```
# ---------------------------------------------------------------------
# Introducción                                                        --
# ---------------------------------------------------------------------

# El objetivo de esta relación de ejercicios es implementar la regla de
# Ruffini y sus aplicaciones utilizando las implementaciones del TAD de
# polinomio estudiadas en los ejercicios anteriores.

# ---------------------------------------------------------------------
# Importación de librerías                                            --
# ---------------------------------------------------------------------
```

```
# pylint: disable=unused-import

from typing import TypeVar

from hypothesis import given
from hypothesis import strategies as st

from src.El_TAD_de_polinomios_operaciones import (coeficiente, creaTermino,
                                                  densaApolinomio, multPol,
                                                  polinomioAdensa, sumaPol)
from src.TAD.Polinomio import (Polinomio, consPol, esPolCero, polCero,
                               polinomioAleatorio)

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

# ------------------------------------------------------------------
# Ejercicio 1. 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
# ------------------------------------------------------------------

def terminoIndep(p: Polinomio[A]) -> A:
    return coeficiente(0, p)

# ------------------------------------------------------------------
# Ejercicio 2. 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

# ----------------------------------------------------------------------
# Ejercicio 3. Definir la función
#    cocienteRuffini : (int, Polinomio[int]) -> Polinomio[int]
# tal que cocienteRuffini(r, p) es el cociente de dividir el polinomio p
# por el polinomio x-r. Por ejemplo:
#      >>> ejPol = consPol(3, 1, consPol(2, 2, consPol(1, -1, consPol(0, -2, polC
#      >>> ejPol
#      x^3 + 2*x^2 + -1*x + -2
#      >>> cocienteRuffini(2, ejPol)
#      x^2 + 4*x + 7
#      >>> cocienteRuffini(-2, ejPol)
#      x^2 + -1
#      >>> cocienteRuffini(3, ejPol)
#      x^2 + 5*x + 14
# ----------------------------------------------------------------------

def cocienteRuffini(r: int, p: Polinomio[int]) -> Polinomio[int]:
    if esPolCero(p):
        return polCero()
    return densaApolinomio(ruffiniDensa(r, polinomioAdensa(p))[:-1])
```

```
# ---------------------------------------------------------------------
# Ejercicio 4. Definir la función
#    restoRuffini    : (int, Polinomio[int]) -> int
# tal que restoRuffini(r, p) es el resto de dividir el polinomio p por
# el polinomio x-r. Por ejemplo,
#    >>> restoRuffini(2, ejPol)
#    12
#    >>> restoRuffini(-2, ejPol)
#    0
#    >>> restoRuffini(3, ejPol)
#    40
# ---------------------------------------------------------------------

def restoRuffini(r: int, p: Polinomio[int]) -> int:
    if esPolCero(p):
        return 0
    return ruffiniDensa(r, polinomioAdensa(p))[-1]

# ---------------------------------------------------------------------
# Ejercicio 5. Comprobar con Hypothesis que, dado un polinomio p y un
# número entero r, las funciones anteriores verifican la propiedad de la
# división euclídea.
# ---------------------------------------------------------------------

# 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
#    >>> test_diviEuclidea()
#    >>>

# ---------------------------------------------------------------------
# Ejercicio 6. 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
# ---------------------------------------------------------------------

def esRaizRuffini(r: int, p: Polinomio[int]) -> bool:
    return restoRuffini(r, p) == 0

# ---------------------------------------------------------------------
# Ejercicio 6. Definir la función
#    divisores : (int) -> list[int]
# tal que 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]

# ---------------------------------------------------------------------
# Ejercicio 7. 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)
#    [1, -1, -2]
# ----------------------------------------------------------------------

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)))

# ----------------------------------------------------------------------
# Ejercicio 8. 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]
# ----------------------------------------------------------------------

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)))

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

# La comprobación es
#    src> poetry run pytest -v Division_y_factorizacion_de_polinomios.py
#    test_diviEuclidea PASSED
#    ========= 1 passed in 0.32s ==========
```

Capítulo 12

El tipo abstracto de datos de los grafos

12.1. El tipo abstracto de datos (TAD) de los grafos

```
# 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.

# 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)
```

12.2. Implementación del TAD de los grafos mediante listas

```
# 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)]
```

```python
# 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)])
```

12.3. Problemas básicos con el TAD de los grafos

```
# ---------------------------------------------------------------------
# Introducción                                                       --
# ---------------------------------------------------------------------

# El objetivo de esta relación de ejercicios es definir funciones sobre
# el TAD de los grafos usando las implementaciones de los ejercicios
# anteriores.

# ---------------------------------------------------------------------
# Importación de librerías                                           --
# ---------------------------------------------------------------------

from typing import Any, Optional

from hypothesis import given
from hypothesis import strategies as st
from hypothesis.strategies import composite

from src.TAD.Grafo import (Grafo, Orientacion, Vertice, adyacentes, aristaEn,
                           aristas, creaGrafo_, dirigido, nodos)

# ---------------------------------------------------------------------
# Ejercicio 1. 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)])
# ----------------------------------------------------------------------

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

# ----------------------------------------------------------------------
# Ejercicio 2. 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)
#
# 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)])
# ----------------------------------------------------------------------

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

# ----------------------------------------------------------------------
# Ejercicio 3. 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
# ----------------------------------------------------------------------

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

# ----------------------------------------------------------------------
# Ejercicio 4. 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.
#
# 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]
# ----------------------------------------------------------------------

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

# ---------------------------------------------------------------------
# Ejercicio 5. 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.
#
# 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]
# ---------------------------------------------------------------------

def contiguos(g: Grafo, v: Vertice) -> list[Vertice]:
    return list(set(adyacentes(g, v) + incidentes(g, v)))
```

```python
# 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

# ---------------------------------------------------------------------
# Ejercicio 6. Definir la función
#    lazos  : (Grafo) -> list[tuple[Vertice, Vertice]]
# tal 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)
#    []
# ---------------------------------------------------------------------

def lazos(g: Grafo) -> list[tuple[Vertice, Vertice]]:
    return [(x, x) for x in nodos(g) if aristaEn(g, (x, x))]

# 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) == []
    print("Verificado")

# La verificación es
#    >>> test_lazos()
#    Verificado

# ---------------------------------------------------------------------
# Ejercicio 7. Definir la función
#    nLazos : (Grafo) -> int
# tal que nLazos(g) es el número de lazos del grafo g. Por ejemplo,
#    >>> nLazos(ej1)
#    2
#    >>> nLazos(ej2)
#    0
# ---------------------------------------------------------------------

def nLazos(g: Grafo) -> int:
    return len(lazos(g))

# Verificación
# ============

def test_nLazos() -> 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 nLazos(ej1) == 2
    assert nLazos(ej2) == 0
    print("Verificado")

# La verificación es
#    >>> test_nLazos()
#    Verificado

#  ---------------------------------------------------------------------
#  Ejercicio 8. 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
#  ----------------------------------------------------------------------

# 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])

# 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

# ---------------------------------------------------------------------
#  Ejercicio 9. 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.
# ---------------------------------------------------------------------

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

# ---------------------------------------------------------------------
# Ejercicio 10. El grado positivo de un vértice v de un grafo g es el
# número de vértices de g adyacentes con v.
#
# Definir la función
#    gradoPos : (Grafo, Vertice) -> int
# tal 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
# ---------------------------------------------------------------------

def gradoPos(g: Grafo, v: Vertice) -> int:
    return len(adyacentes(g, v))

# Verificación
# ============

def test_GradoPos() -> 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
    print("Verificado")

# La verificación es
#    >>> test_GradoPos()
#    Verificado

# ---------------------------------------------------------------------
# Ejercicio 11. El grado negativo de un vértice v de un grafo g es el
# número de vértices de g incidentes con v.
#
# Definir la función
#    gradoNeg : (Grafo, Vertice) -> int
# tal que gradoNeg(g, v) es el grado negativo 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> gradoNeg(g1, 5)
#      4
#      λ> gradoNeg(g2, 5)
#      3
#      λ> gradoNeg(g2, 1)
#      0
# ----------------------------------------------------------------------

def gradoNeg(g: Grafo, v: Vertice) -> int:
    return len(incidentes(g, v))

# Verificación
# ============

def test_GradoNeg() -> 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 gradoNeg(g1, 5) == 4
    assert gradoNeg(g2, 5) == 3
    assert gradoNeg(g2, 1) == 0
    print("Verificado")

# La verificación es
#    >>> test_GradoPosNeg()
#    Verificado

# ----------------------------------------------------------------------
# Ejercicio 12. Definir un generador de grafos para comprobar
# propiedades de grafos con Hypothesis.
# ----------------------------------------------------------------------

# 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())

# ---------------------------------------------------------------------
# Ejercicio 13. 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
# ---------------------------------------------------------------------

# 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
#    >>> test_sumaGrados()
#    >>>

# ---------------------------------------------------------------------
# Ejercicio 14. 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.
#
# 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
# ----------------------------------------------------------------------

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))

# 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

# ----------------------------------------------------------------------
# Ejercicio 15. Comprobar con Hypothesis que en todo grafo, el número de
# nodos de grado impar es par.
# ----------------------------------------------------------------------

# 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
#    >>> test_grado1()
#    >>>

# ----------------------------------------------------------------------
#  Ejercicio 16. 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.
# ----------------------------------------------------------------------

# 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
#    >>> test_apreton()
#    >>>

# ----------------------------------------------------------------------
# Ejercicio 17. Comprobar con QuickCheck que en todo grafo, el número
# de nodos de grado impar es par.
# ----------------------------------------------------------------------

# La propiedad es
@given(gen_grafo())
def test_numNodosGradoImpar(g: Grafo) -> None:
    vs = nodos(g)
    m = len([v for v in vs if grado(g, v) % 2 == 1])
    assert m % 2 == 0

# La comprobación es
#    >>> test_numNodosGradoImpar()
#    >>>

# ----------------------------------------------------------------------
# Ejercicio 18. Definir la propiedad
```

```
#   prop_GradoCompleto :: Int -> Bool
# tal que (prop_GradoCompleto n) se verifica si todos los vértices del
# grafo completo K(n) tienen grado n-1. Usarla para comprobar que dicha
# propiedad se verifica para los grafos completos de grados 1 hasta 30.
# ----------------------------------------------------------------------

# La propiedad es
@given(st.integers(min_value=1, max_value=20))
def test_GradoCompleto(n: int) -> None:
    g = completo(n)
    assert all(grado(g, v) == (n - 1) for v in nodos(g))

# La comprobación es
#    >>> test_GradoCompleto()
#    >>>

# ----------------------------------------------------------------------
# Ejercicio 19. Un grafo es regular si todos sus vértices tienen el
# mismo grado.
#
# 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
# ----------------------------------------------------------------------

def regular(g: Grafo) -> bool:
    vs = nodos(g)
    k = grado(g, vs[0])
    return all(grado(g, v) == k for v in vs)

# 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

# ---------------------------------------------------------------------
# Ejercicio 20. Comprobar que los grafos completos son regulares.
# ---------------------------------------------------------------------

# 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

# ---------------------------------------------------------------------
# Ejercicio 21. Un grafo es k-regular si todos sus vértices son de
# grado k.
#
# 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
# ---------------------------------------------------------------------

def regularidad(g: Grafo) -> Optional[int]:
```

```
    if regular(g):
        return grado(g, nodos(g)[0])
    return None

# 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

# ---------------------------------------------------------------------
# Ejercicio 22. Comprobar que el grafo completo de orden n es
# (n-1)-regular (para n de 1 a 20).
# ---------------------------------------------------------------------

# La propiedad 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

# ---------------------------------------------------------------------
# Ejercicio 23. Comprobar que el grafo ciclo de orden n es 2-regular
# (para n de 3 a 20).
# ---------------------------------------------------------------------
```

```
# La propiedad 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
# ============

# La comprobación de las propiedades es
#    src> poetry run pytest -v Problemas_basicos_de_grafos.py
#    test_completo PASSED
#    test_grafoCiclo PASSED
#    test_nVertices PASSED
#    test_incidentes PASSED
#    test_contiguos PASSED
#    test_lazos PASSED
#    test_nLazos PASSED
#    test_nAristas PASSED
#    test_GradoPos PASSED
#    test_GradoNeg PASSED
#    test_sumaGrados PASSED
#    test_grado PASSED
#    test_grado1 PASSED
#    test_apreton PASSED
#    test_numNodosGradoImpar PASSED
#    test_GradoCompleto PASSED
#    test_regular PASSED
#    test_k_regularidad PASSED
#    ====== passed in 1.17s ======
```

12.4. Algoritmos sobre grafos

```
# ----------------------------------------------------------------------
# Introducción                                                        --
# ----------------------------------------------------------------------

# En esta relación se presentan los algoritmos fundamentales sobre
```

```
# grafos.

# ---------------------------------------------------------------------
# Librerías auxiliares                                               --
# ---------------------------------------------------------------------

from typing import TypeVar

from src.TAD.Grafo import (Grafo, Orientacion, Peso, Vertice, adyacentes,
                           aristas, creaGrafo, creaGrafo_, nodos)

A = TypeVar('A')

# ---------------------------------------------------------------------
# Ejercicio 1. 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
#    grafo2: Grafo = creaGrafo_(Orientacion.D,
#                               (1,6),
#                               [(1,2),(1,3),(1,4),(3,6),(5,4),(6,2),(6,5)])
# entonces
#    recorridoEnProfundidad(1, grafo2)  ==  [1,2,3,6,5,4]
# ---------------------------------------------------------------------

grafo2: 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
# ============

def test_recorridoEnProfundidad() -> None:
    grafo3 = creaGrafo_(Orientacion.ND,
                        (1,6),
                        [(1,2),(1,3),(1,4),(3,6),(5,4),(6,2),(6,5)])
    assert recorridoEnProfundidad1(1, grafo2) == [1,2,3,6,5,4]
    assert recorridoEnProfundidad1(1, grafo3) == [1,2,6,3,5,4]
    assert recorridoEnProfundidad(1, grafo2) == [1,2,3,6,5,4]
    assert recorridoEnProfundidad(1, grafo3) == [1,2,6,3,5,4]
    print("Verificado")

# La verificación es
#    >>> test_recorridoEnProfundidad()
#    Verificado

# ---------------------------------------------------------------------
# Ejercicio 2. 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
#    grafo4: Grafo = creaGrafo_(Orientacion.D,
#                               (1,6),
#                               [(1,2),(1,3),(1,4),(3,6),(5,4),(6,2),(6,5)])
# entonces
#    recorridoEnAnchura(1, grafo4)  ==  [1,2,3,4,6,5]
# ------------------------------------------------------------

grafo4: 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, grafo4)
#    recorridoEnAnchura(1, grafo4
#    = 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:
    grafo5 = creaGrafo_(Orientacion.ND,
                        (1,6),
                        [(1,2),(1,3),(1,4),(3,6),(5,4),(6,2),(6,5)])
    assert recorridoEnAnchura(1, grafo4) == [1,2,3,4,6,5]
    assert recorridoEnAnchura(1, grafo5) == [1,2,3,4,6,5]
    print("Verificado")

# La verificación es
#    >>> test_recorridoEnAnchura()
#    Verificado

# ---------------------------------------------------------------------
# Ejercicio 3. 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.
#
# 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)]
# ----------------------------------------------------------------------

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("Verificado")

# La verificación es
#    >>> test_kruskal()
#    Verificado

# ----------------------------------------------------------------------
# Ejercicio 4. 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 en el ejercicio anterior,
# 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)]
# ----------------------------------------------------------------------

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

# Verificación
# ============

# La comprobación es
#    src> poetry run pytest -v Algoritmos_sobre_grafos.py
#    === session starts ==================================================
#    test_recorridoEnProfundidad PASSED
#    test_recorridoEnAnchura PASSED
#    test_kruskal PASSED
#    test_prim PASSED
#    === passed in 0.12s =================================================
```

12.5. Ejercicios sobre grafos

```
# ----------------------------------------------------------------------
# Introducción                                                         --
# ----------------------------------------------------------------------

# En esta relación se presenta una recopilación de ejercicios sobre
# grafos propuestos en exámenes.

# ----------------------------------------------------------------------
# Librerías auxiliares                                                 --
# ----------------------------------------------------------------------
```

```
from enum import Enum
from itertools import permutations
from typing import TypeVar

from src.Algoritmos_sobre_grafos import recorridoEnAnchura
from src.Problemas_basicos_de_grafos import grafoCiclo, incidentes
from src.TAD.Grafo import (Grafo, Orientacion, Vertice, adyacentes, aristas,
                           creaGrafo_, nodos)

A = TypeVar('A')

# ---------------------------------------------------------------------
# Ejercicio 1. 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]]
# ---------------------------------------------------------------------

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
```

```
# ----------------------------------------------------------------------
# Ejercicio 2.1. 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.
#
# Definir la función,
#    anchura : (Grafo) -> int
# tal que anchuraG(g) es la anchura del grafo g. Por ejemplo,
#    anchura(grafo1)  ==  4
# ----------------------------------------------------------------------

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))

# 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

# ---------------------------------------------------------------------
# Ejercicio 2.2. Comprobar experimentalmente que la anchura del grafo
# ciclo de orden n es n-1.
# ---------------------------------------------------------------------

# 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

# ---------------------------------------------------------------------
# Ejercicio 3. 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.
#
# 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
# ---------------------------------------------------------------------
```

```python
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

# ------------------------------------------------------------------
# Ejercicio 4. 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
# ----------------------------------------------------------------------

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

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

# ---------------------------------------------------------------------
# Ejercicio 5. 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
#    grafo7: 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.
#
# 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 grafo7 == [1,2,4]
# ---------------------------------------------------------------------

grafo7: 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(grafo7) == [1, 2, 4]
    print("Verificado")

# La verificación es
#    >>> test_aislados()
#    Verificado

# ---------------------------------------------------------------------
# Ejercicio 6. 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
#    grafo8 = 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 grafo8 1 3  ==  True
#    conectados grafo8 1 4  ==  False
#    conectados grafo8 6 2  ==  False
#    conectados grafo8 3 1  ==  True
# ---------------------------------------------------------------------

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)))

def conectados(g: Grafo, v1: Vertice, v2: Vertice) -> bool:
```

```
    return v2 in conectadosAux(g, [], [v1])


# Verificación
# ============

def test_conectados() -> None:
    grafo8 = 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)])
    grafo8b = 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(grafo8, 1, 3)
    assert not conectados(grafo8, 1, 4)
    assert not conectados(grafo8, 6, 2)
    assert conectados(grafo8, 3, 1)
    assert conectados(grafo8b, 1, 3)
    assert not conectados(grafo8b, 1, 4)
    assert conectados(grafo8b, 6, 2)
    assert conectados(grafo8b, 3, 1)
    print("Verificado")

# La verificación es
#    >>> test_conectados()
#    Verificado

# Verificación
# ============

# La verificación es
#    src> poetry run pytest -v Ejercicios_sobre_grafos.py
#    test_recorridos PASSED
#    test_anchura PASSED
#    test_conexo PASSED
#    test_correcta PASSED
#    test_aislados PASSED
#    test_conectados PASSED
```

```
#    ====== 6 passed in 0.14s ======
```

Capítulo 13

Procedimiento de divide y vencerás

13.1. Algoritmo divide y vencerás

```
# ----------------------------------------------------------------------
# Introducción                                                         --
# ----------------------------------------------------------------------

# 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.

# ----------------------------------------------------------------------
# Librerías auxiliares                                                 --
# ----------------------------------------------------------------------

from typing import Callable, TypeVar

P = TypeVar('P')
S = TypeVar('S')

# ----------------------------------------------------------------------
# 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.
# + 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]
# ----------------------------------------------------------------------

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
```

13.2. Rompecabeza del triominó mediante divide y vencerás

```
# ----------------------------------------------------------------------
# Introducción
# ----------------------------------------------------------------------

# 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 trominós, el I-tromino (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-trominós de formas que cubran todas las casillas excepto la
# eliminada y los trominós no se solapen.
#
# La casilla eliminada se representará con -1 y los L-trominós 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 )
#
# Con divideVenceras(ind, resuelve, divide, combina pbInicial) se
# 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
#
# En los distintos apartados de esta relación se irán definiendo las
# anteriores funciones.

# ---------------------------------------------------------------------
# Librerías auxiliares
# ---------------------------------------------------------------------

import numpy as np
import numpy.typing as npt

from src.DivideVenceras import divideVenceras

# ---------------------------------------------------------------------
# Tipos
# ---------------------------------------------------------------------

# 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]

# ------------------------------------------------------------------
# Problema inicial
# ------------------------------------------------------------------

# ------------------------------------------------------------------
# Ejercicio 1. Definir la función
#    tablero(int, Posicion) -> Tablero
# tal que 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

# ------------------------------------------------------------------
# Ejercicio 2. Definir la función
#    pbInicial(int, Posicion) -> Problema
# tal que 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)

# ---------------------------------------------------------------------
# Problemas indivisibles
# ---------------------------------------------------------------------

# ---------------------------------------------------------------------
# Ejercicio 3. Definir la función
#    ind :: (Problema) -> bool
# 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

# ---------------------------------------------------------------------
# Resolución de problemas indivisibles
# ---------------------------------------------------------------------

# ---------------------------------------------------------------------
# Ejercicio 4. Definir la función
#    posicionHueco(Tablero) -> Posicion
# 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)

# ---------------------------------------------------------------------
# Ejercicio 5. Definir la función
#    cuadranteHueco(t: Tablero) -> int
# 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

# ---------------------------------------------------------------------
# Ejercicio 6. Definir la función
#    centralHueco :: (Tablero) -> Posicion
# tal que 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)
```

```python
# ---------------------------------------------------------------------
# Ejercicio 7. Definir la función
#    centralesSinHueco : (Tablero) -> [Posicion]
# tal que 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)]

# ---------------------------------------------------------------------
# Ejercicio 8. Definir la función
#    actualiza : (Tablero, list[tuple[Posicion, int]]) -> Tablero
# 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

# ---------------------------------------------------------------------
# Ejercicio 9. Definir la función
#    triominoCentral : (Problema) -> Tablero
# tal que 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 trominoCentral(p: Problema) -> Tablero:
    n, t = p
    return actualiza(t, [((i,j),n) for (i,j) in centralesSinHueco(t)])

# ---------------------------------------------------------------------
# Ejercicio 10. Definir la función
#    resuelve :: (Problema) -> Tablero
# tal que 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 trominoCentral(p)

# ---------------------------------------------------------------------
# División en subproblemas
# ---------------------------------------------------------------------

# ---------------------------------------------------------------------
# Ejercicio 11. Definir la función
#    divide : (Problema) -> list[Problema]
# tal que divide(n,t) es la lista de de los problemas obtenidos
# colocando el trominó 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 trominó. 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 = trominoCentral(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

# ------------------------------------------------------------------
# Combinación de soluciones
# ------------------------------------------------------------------

# ------------------------------------------------------------------
# Ejercicio 12. Definir la función
#    combina : (Problema, list[Tablero]) -> Tablero
# 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

# ----------------------------------------------------------------------
# Solución mediante divide y vencerás
# ----------------------------------------------------------------------

# ----------------------------------------------------------------------
# Ejercicio 13. 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]])
# ------------------------------------------------------------------------

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
```

```
# ---------------------------------------------------------------------
# Referencias
# ---------------------------------------------------------------------

# + Raúl Ibáñez "Embaldosando con L-triominós (Un ejemplo de
#   demostración por inducción)" http://bit.ly/1DKPBbt
# + "Algorithmic puzzles" pp. 10.
```

Capítulo 14

Problemas con búsquedas en espacio de estados

14.1. Búsqueda en espacios de estados por profundidad

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

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

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
```

14.2. El problema de las n reinas (por profundidad)

```
# ----------------------------------------------------------------------
# Introducción
# ----------------------------------------------------------------------

# 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.
#
# El objetivo de esta relación de ejercicios es resolver el problema
# del granjero mediante búsqueda en espacio de estados, utilizando las
# búsqueda en profundidad estudiada en el ejercicio anterior.

# ----------------------------------------------------------------------
# Importaciones
# ----------------------------------------------------------------------

from src.BusquedaEnProfundidad import buscaProfundidad

# ----------------------------------------------------------------------
# Ejercicio 1. Las posiciones de las reinas en el tablero se representan
# por su columna y su fila, que son números enteros.
#
# Una solución del problema de las n reinas es una lista de
# posiciones. Su tipo es SolNR.
#
# Definir los tipos Columna, Fila y SolNR.
# ----------------------------------------------------------------------

Columna = int
Fila = int
SolNR = list[tuple[Columna, Fila]]

# ----------------------------------------------------------------------
# Ejercicio 2. 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.
```

```
#
# Definir el tipo nNodoNR.
# ----------------------------------------------------------------------

NodoNR = tuple[Columna, Columna, SolNR]

# ----------------------------------------------------------------------
# Ejercicio 3. Definir la función
#    valida(SolNR, tuple[Columna, Fila]) -> bool
# tal que 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)

# ----------------------------------------------------------------------
# Ejercicio 4. Definir la función
#    sucesoresNR (NodoNR) -> list[NodoNR]
# tal que 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))]

# ----------------------------------------------------------------------
# Ejercicio 5. Definir la función
```

```
#    esFinalNR(NodoNR) -> bool
# tal que 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

# ---------------------------------------------------------------------
# Ejercicio 6. Definir la función
#    solucionesNR : (int) -> list[SolNR]
# tal 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)]]
# ---------------------------------------------------------------------

def solucionesNR(n: int) -> list[SolNR]:
    nInicial: NodoNR = (1,n,[])
    return [e for (_, _, e) in buscaProfundidad(sucesoresNR,
                                                esFinalNR,
                                                nInicial)]

# ---------------------------------------------------------------------
# Ejercicio 7. Definir la función
#    primeraSolucionNR : (int) -> SolNR
# tal que 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)]
# ---------------------------------------------------------------------

def primeraSolucionNR(n: int) -> SolNR:
    return solucionesNR(n)[0]
```

```
# ---------------------------------------------------------------------
# Ejercicio 8. Definir la función
#    nSolucionesNR : (int) -> int
# tal que nSolucionesNR(n) es el número de soluciones del problema de
# las n reinas, por búsqueda en espacio de estados. Por ejemplo,
#    >>> nSolucionesNR(8)
#    92
# ---------------------------------------------------------------------

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
#    >>> test_nReinas()
#    Verificado
```

14.3. Búsqueda en espacios de estados por anchura

```
# ---------------------------------------------------------------------
# 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())

    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
```

14.4. El problema de las n reinas (por anchura)

```
# -----------------------------------------------------------------
# Introducción
# -----------------------------------------------------------------

# 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.
#
# El objetivo de esta relación de ejercicios es resolver el problema
# del granjero mediante búsqueda en espacio de estados, utilizando las
# búsqueda en profundidad estudiada en el ejercicio anterior.

# 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.

# -----------------------------------------------------------------
# Importaciones
# -----------------------------------------------------------------

from src.BusquedaEnAnchura import buscaAnchura

# -----------------------------------------------------------------
# Ejercicio 1. Las posiciones de las reinas en el tablero se representan
```

```
# por su columna y su fila, que son números enteros.
#
# Una solución del problema de las n reinas es una lista de
# posiciones. Su tipo es SolNR.
#
# Definir los tipos Columna, Fila y SolNR.
# ---------------------------------------------------------------------

Columna = int
Fila = int
SolNR = list[tuple[Columna, Fila]]

# ---------------------------------------------------------------------
# Ejercicio 2. 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.
#
# Definir el tipo nNodoNR.
# ---------------------------------------------------------------------

NodoNR = tuple[Columna, Columna, SolNR]

# ---------------------------------------------------------------------
# Ejercicio 3. Definir la función
#    valida(SolNR, tuple[Columna, Fila]) -> bool
# tal que 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)

# ----------------------------------------------------------------------
# Ejercicio 4. Definir la función
#    sucesoresNR (NodoNR) -> list[NodoNR]
# tal que 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))]

# ----------------------------------------------------------------------
# Ejercicio 5. Definir la función
#    esFinalNR(NodoNR) -> bool
# tal que 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

# ----------------------------------------------------------------------
# Ejercicio 6. Definir la función
#    solucionesNR : (int) -> list[SolNR]
# tal 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)]]
# ----------------------------------------------------------------------

def solucionesNR(n: int) -> list[SolNR]:
    nInicial: NodoNR = (1,n,[])
```

```
    return [e for (_, _, e) in buscaAnchura(sucesoresNR,
                                            esFinalNR,
                                            nInicial)]

# ---------------------------------------------------------------------
# Ejercicio 7. Definir la función
#    primeraSolucionNR : (int) -> SolNR
# tal que primeraSolucionNR(n) es la primera solución del problema de las n
# reinas, por búsqueda en espacio de estados por profundidad. Por
# ejemplo,
# ---------------------------------------------------------------------

def primeraSolucionNR(n: int) -> SolNR:
    return solucionesNR(n)[0]

# ---------------------------------------------------------------------
# Ejercicio 8. Definir la función
#    nSolucionesNR : (int) -> int
# tal que nSolucionesNR(n) es el número de soluciones del problema de
# las n reinas, por búsqueda en espacio de estados. Por ejemplo,
#      >>> nSolucionesNR(8)
#      92
# ---------------------------------------------------------------------

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
```

14.5. El problema de la mochila

```
# ---------------------------------------------------------------------
# Introducción
# ---------------------------------------------------------------------

# 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.
#
# En esta relación de ejercicios se resolverá el problema de la mochila
# mediante búsqueda en espacios de estados.

# ---------------------------------------------------------------------
# Importaciones
# ---------------------------------------------------------------------

from src.BusquedaEnProfundidad import buscaProfundidad

# ---------------------------------------------------------------------
# Ejercicio 1. Para solucionar el problema se definen los siguientes
# tipos:
# + Una solución del problema de la mochila es una lista de objetos.
#      Solucion = 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.
```

```
#      Estado = tuple[Valor, Peso, Peso, list[Objeto], Solucion]
#
# Definir los tipos Peso, Valor, Objeto, Solucion, Estado para los tipis
# de los pesos, valores, objetos, soluciones y estados, respectivamente.
# ----------------------------------------------------------------------

Peso = int
Valor = float
Objeto = tuple[Peso, Valor]
Solucion = list[Objeto]
Estado = tuple[Valor, Peso, Peso, list[Objeto], Solucion]

# ----------------------------------------------------------------------
# Ejercicio 2. Definir la función
#    inicial : (list[Objeto], Peso) -> Estado
# tal que 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) -> Estado:
    return (0,0,l,sorted(os),[])

# ----------------------------------------------------------------------
# Ejercicio 3. Definir la función
#    sucesores : (Estado) -> list[Estado]
# tal que sucesores(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 sucesores(n: Estado) -> list[Estado]:
    (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]

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

```python
# Ejercicio 4. Definir la función
#    esObjetivo : (Estado) -> bool
# tal que esObjetivo(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 esObjetivo(e: Estado) -> bool:
    (_, p, l, os, _) = e
    (p_, _) = os[0]
    return p + p_ > l

# ----------------------------------------------------------------------
# Ejercicio 5. Usando el procedimiento de búsqueda en profundidad,
# 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)
# ----------------------------------------------------------------------

def mochila(os: list[Objeto], l: Peso) -> tuple[Solucion, Valor]:
    (v,_,_,_,sol) = max(buscaProfundidad(sucesores,
                                         esObjetivo,
                                         inicial(os, l)))
    return (sol, v)

# 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
```

14.6. Búsqueda por primero el mejor

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

14.7. El problema del 8 puzzle

```
# -----------------------------------------------------------------------
# Introducción
# -----------------------------------------------------------------------

# 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
#
# En esta relación de ejercicios resolveremos el problema del 8-puzzle
# mediante búsqueda por primero el mejor.

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

```
# Importaciones
# -----------------------------------------------------------------

from copy import deepcopy
from typing import Optional

from src.BusquedaPrimeroElMejor import buscaPM

# -----------------------------------------------------------------
# Ejercicio 1. Para solucionar el problema se usará el tipo Tablero que
# son listas de listas de números enteros (que representan las piezas en
# cada posición y el 0 representa el hueco).
#
# Definir el tipo Tablero.
# -----------------------------------------------------------------

Tablero = list[list[int]]

# -----------------------------------------------------------------
# Ejercicio 2. Definir la constante tableroFinalpara representar el
# tablero final del 8 puzzle.
# -----------------------------------------------------------------

tableroFinal: Tablero = [[1,2,3],
                         [8,0,4],
                         [7,6,5]]

# -----------------------------------------------------------------
# Ejercicio 3. Una posición es un par de enteros.
#
# Definir el tipo Posicion para representar posiciones.
# -----------------------------------------------------------------

Posicion = tuple[int,int]

# -----------------------------------------------------------------
# Ejercicio 4. Definir la función
#    distancia : (Posicion, Posicion) -> int
# tal que 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)

# ---------------------------------------------------------------------
# Ejercicio 5. Definir la función
#    posicionElemento : (Tablero, int) -> Posicion
# tal que 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)

# ---------------------------------------------------------------------
# Ejercicio 6. Definir la función
#    posicionHueco : (Tablero) -> Posicion
# 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)

# ---------------------------------------------------------------------
# Ejercicio 7. Definir la función
#    heuristica : (Tablero) -> int
```

```
# tal que 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)))

# ----------------------------------------------------------------------
# Ejercicio 8. 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.
#
# Definir el tipo Estado para representar los estados.
# ----------------------------------------------------------------------

Estado = tuple[int, int, list[Tablero]]

# ----------------------------------------------------------------------
# Ejercicio 9. Definir la función
#    inicial (Tablero) -> Estado
# tal que 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])

# ----------------------------------------------------------------------
# Ejercicio 10. Definir la función
#    esFinal : (Estado) -> bool
# tal que esFinal(e) se verifica si e es un estado final.
# ----------------------------------------------------------------------

def esFinal(e: Estado) -> bool:
    (_, _, ts) = e
```

```
    return ts[0] == tableroFinal

# ---------------------------------------------------------------------
# Ejercicio 11. Definir la función
#    posicionesVecinas : (Posicion) -> list[Posicion]
# tal que 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

# ---------------------------------------------------------------------
# Ejercicio 12. Definir la función
#    intercambia : (Tablero, Posicion, Posicion) -> Tablero
# tal que 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

# ---------------------------------------------------------------------
# Ejercicio 13. Definir la función
#    tablerosSucesores : (Tablero) -> list[Tablero]
# tal que 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)]

# ---------------------------------------------------------------------
# Ejercicio 14. Definir la función
#    sucesores : (Estado) -> list[Estado]
# tal que (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]

# ---------------------------------------------------------------------
# Ejercicio 15. 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
# ------------------------------------------------------------------

def solucion_8puzzle(t: Tablero) -> Optional[list[Tablero]]:
    r = buscaPM(sucesores, esFinal, inicial(t))
    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]]]
    print("Verificado")

# La verificación es
#    >>> test_8puzzle()
#    Verificado
```

14.8. Búsqueda en escalada

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

14.9. El algoritmo de Prim del árbol de expansión mínimo

```
# ----------------------------------------------------------------------
# Introducción
# ----------------------------------------------------------------------

# 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)
#
# En esta relación de ejercicios implementaremos el algoritmo de Prim
# mediante búsqueda en escalada usando el tipo abstracto de los grafos.
```

```
# ---------------------------------------------------------------------
# Importaciones
# ---------------------------------------------------------------------

from typing import Optional

from src.BusquedaEnEscalada import buscaEscalada
from src.TAD.Grafo import (Grafo, Orientacion, Peso, Vertice, aristaEn,
                           creaGrafo, nodos, peso)

# ---------------------------------------------------------------------
# Nota. En los ejemplos de los ejercicios usaremos los grafos definidos
# a continuación.
# ---------------------------------------------------------------------

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)])

# ---------------------------------------------------------------------
# Ejercicio 1. Las aristas son pares de la forma ((v1, v2), p) donde v1
# es el vértice origen, v2 es el vértice destino y p es el peso de la
# arista.
#
# Definir el tipo Arista para las aristas.
# ---------------------------------------------------------------------

Arista = tuple[tuple[Vertice, Vertice], Peso]

# ---------------------------------------------------------------------
# Ejercicio 2. Un estado es una tupla de la forma (p,t,r,aem) donde p es
# el peso de la última arista añadida el árbol de expansión mínimo
# (aem), t es la lista de nodos del grafo que están en el aem y r es la
# lista de nodos del grafo que no están en el aem.
#
# Definir el tipo Estado para los estados.
# ---------------------------------------------------------------------

Estado = tuple[Peso, list[Vertice], list[Vertice], list[Arista]]

# ---------------------------------------------------------------------
# Ejercicio 3. Definir la función
#    inicial : (Grafo) -> Estado
# tal que inicial(g) es el estado inicial correspondiente al grafo g.
# ---------------------------------------------------------------------

def inicial(g: Grafo) -> Estado:
    n, *ns = nodos(g)
    return (0, [n], ns, [])

# ---------------------------------------------------------------------
# Ejercicio 4. Definir la función
#    esFinal : (Estado) -> bool
# 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] == []

# ----------------------------------------------------------------------
# Ejercicio 5. Definir la función
#    sucesores : (Grafo, Estado) -> list[Estado]
# 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))]

# ----------------------------------------------------------------------
# Ejercicio 6. Usando la búsqueda en escalada, 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)]
# ----------------------------------------------------------------------

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
```

14.10. El problema del granjero

```
# --------------------------------------------------------------------
# Introducción                                                       --
# --------------------------------------------------------------------

# 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?
#
# El objetivo de esta relación de ejercicios es resolver el problema
# del granjero mediante búsqueda en espacio de estados, utilizando las
# implementaciones estudiadas en los ejercicios anteriores.

# --------------------------------------------------------------------
# Importaciones                                                      --
# --------------------------------------------------------------------

from enum import Enum

from src.BusquedaEnProfundidad import buscaProfundidad

# --------------------------------------------------------------------
# Ejercicio 1. Definir el tipo Orilla con dos constructores I y D que
# representan las orillas izquierda y derecha, respectivamente.
# --------------------------------------------------------------------

class Orilla(Enum):
```

```
    I = 0
    D = 1

    def __repr__(self) -> str:
        return self.name

I = Orilla.I
D = Orilla.D

# ---------------------------------------------------------------------
# Ejercicio 2. Definir el tipo Estado como abreviatura de una tupla que
# representan 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.
# ---------------------------------------------------------------------

Estado = tuple[Orilla, Orilla, Orilla, Orilla]

# ---------------------------------------------------------------------
# Ejercicio 3. Definir la función
#    seguro :  (Estado) -> bool
# tal que 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})

# ---------------------------------------------------------------------
# Ejercicio 4. Definir la función
#    opuesta : (Orilla) -> Orilla
# tal que (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

# ----------------------------------------------------------------------
# Ejercicio 5. Definir la función
#    sucesoresE(Estado) -> list[Estado]
# tal que 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)]
# ----------------------------------------------------------------------

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))]

# ----------------------------------------------------------------------
# Ejercicio 6. 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).
#
# Definir el tipo Nodo.
# ----------------------------------------------------------------------

Nodo = list[Estado]
```

```
# ---------------------------------------------------------------------
# Ejercicio 7. Definir
#    inicial: Nodo
# tal que inicial es el nodo inicial en el que todos están en la orilla
# izquierda.
# ---------------------------------------------------------------------

inicial: Nodo = [(I,I,I,I)]

# ---------------------------------------------------------------------
# Ejercicio 8. Definir la función
#    esFinal : (Nodo) -> bool
# tal que 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)

# ---------------------------------------------------------------------
# Ejercicio 9. Definir la función
#    sucesores : (Nodo) -> list[Nodo]
# tal que 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]

# ---------------------------------------------------------------------
# Ejercicio 10. Usando el procedimiento de búsqueda en profundidad,
# 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
# ----------------------------------------------------------------------

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
```

14.11. El problema de las fichas

```
# ----------------------------------------------------------------------
# Introducción
# ----------------------------------------------------------------------

# 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.
#
# El objetivo de esta relación de ejercicios es resolver el problema
# de las fichas mediante búsqueda en espacio de estados, utilizando las
# implementaciones estudiadas en los ejercicios anteriores.

# ------------------------------------------------------------------------
# Importaciones
# ------------------------------------------------------------------------

from enum import Enum
from functools import partial
from typing import Callable, Optional

from src.BusquedaEnAnchura import buscaAnchura1
from src.BusquedaEnEscalada import buscaEscalada
from src.BusquedaEnProfundidad import buscaProfundidad1
from src.BusquedaPrimeroElMejor import buscaPM

# Representación del problema
# ===========================

# ------------------------------------------------------------------------
# Ejercicio 1. Definir el tipo 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

# ---------------------------------------------------------------------
# Ejercicio 2. Definir el tipo Tablero como una lista de fichas que
# representa las fichas colocadas en el tablero.
# ---------------------------------------------------------------------

Tablero = list[Ficha]

# ---------------------------------------------------------------------
# Ejercicio 3. Definir la función
#    tableroInicial : (int, int) -> Tablero
# tal que 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

# ---------------------------------------------------------------------
# Ejercicio 4. Definir la función
#    tableroFinal : (int, int) -> Tablero
# tal que 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

# ---------------------------------------------------------------------
# Ejercicio 5. Definir la función
#    posicionHueco : (Tablero) -> int
# tal que 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

# ---------------------------------------------------------------------
# Ejercicio 6. Definir la función
#    tablerosSucesores : (Tablero) -> list[Tablero]
# tal que 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
# ==========

# ---------------------------------------------------------------------
# Ejercicio 7. Definir la función
#    heuristicaT : (Tablero) -> int
# tal que 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])

# ---------------------------------------------------------------------
# Ejercicio 8. La clase 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).
#
# Definir la clase Estado.
# ---------------------------------------------------------------------

class Estado(list[Tablero]):
    def __lt__(self, e: list[Tablero]) -> bool:
        return heuristicaT(self[0]) < heuristicaT(e[0])

# ---------------------------------------------------------------------
# Ejercicio 9. Definir la función
#    inicial : (int, int) -> Estado
# tal que 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)])

# ----------------------------------------------------------------------
# Ejercicio 10. Definir la función
#    esFinal(int, int, Estado) -> bool
# tal que 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)

# ----------------------------------------------------------------------
# Ejercicio 11. Definir la función
#    sucesores : (Estado) -> list[Estado]
# tal que 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
# =====================
```

```python
# ---------------------------------------------------------------------
# Ejercicio 12. Definir el tipo Busqueda par representar los
# procedimientos de búsqueda.
# ---------------------------------------------------------------------

Busqueda = Callable[[Callable[[Estado], list[Estado]],
                     Callable[[Estado], bool],
                     Estado],
                    Optional[Estado]]

# ---------------------------------------------------------------------
# Ejercicio 13. 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]]
# ---------------------------------------------------------------------

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
```

14.12. El problema del calendario

```
# ----------------------------------------------------------------------
# Introducción
# ----------------------------------------------------------------------

# 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.
#
# El objetivo de esta relación de ejercicios es resolver el problema
# del calendario mediante búsqueda en espacio de estados, utilizando las
# implementaciones estudiadas en los ejercicios anteriores.

# --------------------------------------------------------------------
# Importaciones
# --------------------------------------------------------------------

from copy import deepcopy
from typing import Optional

import numpy as np
import numpy.typing as npt

from src.BusquedaEnProfundidad import buscaProfundidad

# --------------------------------------------------------------------
# Ejercicio 1. Definir el tipo Calendario como matrices de enteros.
# --------------------------------------------------------------------

Calendario = npt.NDArray[np.int_]

# --------------------------------------------------------------------
# Ejercicio 2. Definir la función
#    inicial : (int) -> Calendario
# tal que 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]])
```

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

def inicial(n: int) -> Calendario:
    return np.zeros((n, n - 1), dtype=int)

# ---------------------------------------------------------------------
# Ejercicio 3. Definir la función
#    primerHueco : (Calendario) -> tuple[int, 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)

# ---------------------------------------------------------------------
# Ejercicio 4. Definir la función
#    libres : (Calendario, int, int) -> list[int]
# tal que 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]))

# ---------------------------------------------------------------------
# Ejercicio 5. Definir la función
#    setElem : (int, int, int, Calendario) -> Calendario
# 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

# ---------------------------------------------------------------------
# Ejercicio 6. Definir la función
#    sucesores : (Calendario) -> list[Calendario]
# tal que 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)]

# ---------------------------------------------------------------------
# Ejercicio 7. Definir la función
#    esFinal : (Calendario) -> bool
# 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)

# ---------------------------------------------------------------------
# Ejercicio 7. Usando el procedimiento de búsqueda en profundidad,
# 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
# ----------------------------------------------------------------------

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")
```

14.13. El problema del dominó

```
# ----------------------------------------------------------------------
# Introducción
# ----------------------------------------------------------------------

# 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.
#
# El objetivo de esta relación de ejercicios es resolver el problema
# del dominó mediante búsqueda en espacio de estados, utilizando las
# implementaciones estudiadas en los ejercicios anteriores.
```

```
# ---------------------------------------------------------------------
# Importaciones
# ---------------------------------------------------------------------

from src.BusquedaEnProfundidad import buscaProfundidad

# ---------------------------------------------------------------------
# Ejercicio 1. Las fichas son pares de números enteros.
#
# Definir el tipo Ficha para las fichas.
# ---------------------------------------------------------------------

Ficha  = tuple[int, int]

# ---------------------------------------------------------------------
# Ejercicio 2. Un problema está definido por la lista de fichas que hay
# que colocar.
#
# Definir el tipo Problema para los problemas.
# ---------------------------------------------------------------------

Problema = list[Ficha]

# ---------------------------------------------------------------------
# Ejercicio 3. Los estados son los pares formados por la listas sin
# colocar y las colocadas.
#
# Definir el tipo Estado para los estados.
# ---------------------------------------------------------------------

Estado = tuple[list[Ficha], list[Ficha]]

# ---------------------------------------------------------------------
# Ejercicio 4. Definir la función
#    inicial : (Problema) -> Estado
# tal que inicial(p) es el estado inicial del problema p. Por ejemplo,
#    >>> inicial([(1,2),(2,3),(1,4)])
#    ([(1, 2), (2, 3), (1, 4)], [])
# ---------------------------------------------------------------------
```

```python
def inicial(p: Problema) -> Estado:
    return (p, [])

# ---------------------------------------------------------------------
# Ejercicio 5. Definir la función
#    esFinal : (Estado) -> bool
# tal que 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]

# ---------------------------------------------------------------------
# Ejercicio 6. Definir la función
#    elimina : (Ficha, list[Ficha]) -> list[Ficha]
# tal que 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]

# ---------------------------------------------------------------------
# Ejercicio 7. Definir la función
#    sucesores : (Estado) -> list[Estado]
# tal que 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

# ----------------------------------------------------------------------
# Ejercicio 8. Definir, mediante búsqueda en espacio de estados, la
# función
#    soluciones : (Problema) -> list[Estado]
# tal que 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))

# ----------------------------------------------------------------------
# Ejercicio 9. Definir 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)])
#    []
```

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

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
```

14.14. El problema de suma cero

```
# ---------------------------------------------------------------------
# Introducción
# ---------------------------------------------------------------------

# El problema de suma cero consiste en, dado el conjunto de números
# enteros, encontrar sus subconjuntos no vacío cuyos elementos sumen
# cero. Por ejemplo, la soluciones para [-7,-3,-2,5,8,-1] son
# [-7,-3,-2,-1,5,8], [-7,-1,8] y [-3,-2,5]
#
# El objetivo de esta relación de ejercicios es resolver el problema
# de suma cero mediante búsqueda en espacio de estados, utilizando las
# implementaciones estudiadas en los ejercicios anteriores.

# ---------------------------------------------------------------------
# Importaciones
```

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

from src.BusquedaEnProfundidad import buscaProfundidad

# ---------------------------------------------------------------------
# Ejercicio 1. Los estados son ternas formadas por los números
# seleccionados, su suma y los restantes números.
#
# Definir el tipo Estado para los estados.
# ---------------------------------------------------------------------

Estado = tuple[list[int], int, list[int]]

# ---------------------------------------------------------------------
# Ejercicio 2. Definir la función
#    inicial : (list[int]) -> Estado
# tal que inicial(ns) es el estado inicial dell problema de suma cero a
# partir de los números de ns. Por ejemplo,
#    >>> inicial([-7,-3,-2,5,8,-1])
#    ([], 0, [-7, -3, -2, 5, 8, -1])
# ---------------------------------------------------------------------

def inicial(ns: list[int]) -> Estado:
    return ([], 0, ns)

# ---------------------------------------------------------------------
# Ejercicio 3. Definir la función
#    esFinal : (Estado) -> bool
# tal es Final(e) se verifica si e es un estado final. Por ejemplo,
#    >>> esFinal(([-7, -1, 8], 0, [-3, -2, 5]))
#    True
#    >>> esFinal(([-3, 8, -1], 4, [-7, -2, 5]))
#    False
# ---------------------------------------------------------------------

def esFinal(e: Estado) -> bool:
    (xs,s,_) = e
    return xs != [] and s == 0

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

```python
# Ejercicio 4. Definir la función
#    sucesores : (Estado) -> list[Estado]
# tal que sucesores(e) es la lista de los sucesores del estado e. Por
# ejemplo,
#    >>> sucesores(([-2, -3, -7], -12, [5, 8, -1]))
#    [([5, -2, -3, -7], -7,   [8, -1]),
#     ([8, -2, -3, -7], -4,   [5, -1]),
#     ([-1, -2, -3, -7], -13, [5, 8])]
# ----------------------------------------------------------------------

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]

# ----------------------------------------------------------------------
# Ejercicio 5. Usando el procedimiento de búsqueda en profundidad,
# 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])
#    []
# ----------------------------------------------------------------------

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
```

14.15. El problema de las jarras

```
# ----------------------------------------------------------------------
# Introducción
# ----------------------------------------------------------------------

# 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. Por ejemplo, una solución del problema de las
# jarras (4,3,2) consiste en los siguientes estados y acciones:
#    (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.
#
# El objetivo de esta relación de ejercicios es resolver el problema
# de las jarras mediante búsqueda en espacio de estados, utilizando las
# implementaciones estudiadas en los ejercicios anteriores.
```

```python
# -----------------------------------------------------------------
# Importaciones
# -----------------------------------------------------------------

from src.BusquedaEnAnchura import buscaAnchura

# -----------------------------------------------------------------
# Ejercicio 1. 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.
#
# Definir el tipo Problema para los problemas.
# -----------------------------------------------------------------

Problema = tuple[int, int, int]

# -----------------------------------------------------------------
# Ejercicio 2. Una configuracion es una lista de dos números. El primero
# es el contenido de la primera jarra y el segundo el de la segunda.
#
# Definir el tipo Configuracion para las configuraciones.
# -----------------------------------------------------------------

Configuracion = tuple[int, int]

# -----------------------------------------------------------------
# Ejercicio 3. Inicialmente, las dos jarras están vacías.
#
# Definir
#    configuracionInicial: Configuracion
# tal que configuracionInicial es la configuración inicial.
# -----------------------------------------------------------------

configuracionInicial: Configuracion = (0,0)

# -----------------------------------------------------------------
# Ejercicio 4. Definir la función
#    esConfiguracionFinal : (Problema, Configuracion) -> bool
```

```python
# tal 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]

# -----------------------------------------------------------------
# Ejercicio 5. Definir la función
#    sucesorasConfiguracion : (Problema, Configuracion) -> list[Configuracion]
# 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

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

```
# Ejercicio 6. 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).
#
# Definir el tipo Estado para los estados.
# ---------------------------------------------------------------------

Estado = list[Configuracion]

# ---------------------------------------------------------------------
# Ejercicio 7. Definir
#    inicial: Estado
# tal que inicial es el estado cuyo único elemento es la configuración
# inicial.
# ---------------------------------------------------------------------

inicial: Estado = [configuracionInicial]

# ---------------------------------------------------------------------
# Ejercicio 8. Definir la función
#    esFinal(Problema, Estado) -> bool
# tal que 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])

# ---------------------------------------------------------------------
# Ejercicio 9. Definir la función
#    sucesores : (Problema, Estado) -> list[Estado]
# 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)]]
# ---------------------------------------------------------------------
```

```python
def sucesores(p: Problema, e: Estado) -> list[Estado]:
    return [[c] + e
            for c in sucesorasConfiguracion(p, e[0])
            if c not in e]

# ---------------------------------------------------------------------
# Ejercicio 10. Usando el procedimiento de búsqueda en anchura,
# 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))
#    []
# ---------------------------------------------------------------------

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]
```

```python
# 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 15

Programación dinámica

15.1. La función de Fibonacci por programación dinámica

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

15.2. Coeficientes binomiales

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

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
```

```python
# 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
# 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

# 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
```

15.3. Longitud de la subsecuencia común máxima

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

```python
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,
```

```python
# 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
```

15.4. Subsecuencia común máxima

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

15.5. La distancia Levenshtein

```
# ----------------------------------------------------------------------
# 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
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")
```

15.6. Caminos en una retícula

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

15.7. Caminos en una matriz

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

15.8. Máxima suma de los caminos en una matriz

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

15.9. Camino de máxima suma en una matriz

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

```python
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 16

Cálculo numérico

16.1. Cálculo numérico: Diferenciación y métodos de Herón y de Newton

```
# ----------------------------------------------------------------------
# Introducción                                                       --
# ----------------------------------------------------------------------

# En esta relación se definen funciones para resolver los siguientes
# problemas de cálculo numérico:
# + diferenciación numérica,
# + cálculo de la raíz cuadrada mediante el método de Herón,
# + cálculo de los ceros de una función por el método de Newton y
# + cálculo de funciones inversas.

# ----------------------------------------------------------------------
# Librerías auxiliares                                               --
# ----------------------------------------------------------------------

from functools import partial
from math import cos, pi, sin
from typing import Callable

from hypothesis import given
from hypothesis import strategies as st

# ----------------------------------------------------------------------
# Diferenciación numérica                                            --
```

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

# ---------------------------------------------------------------------
# Ejercicio 1.1. Definir la función
#    derivada : (float, Callable[[float], float], float) -> float
# tal que derivada(a, f, x) es el valor de la derivada de la función f
# en el punto x con aproximación a. Por ejemplo,
#    derivada(0.001, sin, pi)  ==  -0.9999998333332315
#    derivada(0.001, cos, pi)  ==  4.999999583255033e-4
# ---------------------------------------------------------------------

def derivada(a: float, f: Callable[[float], float], x: float) -> float:
    return (f(x+a) - f(x)) / a

# Verificación
# ============

def test_derivada() -> None:
    assert derivada(0.001, sin, pi) == -0.9999998333332315
    assert derivada(0.001, cos, pi) == 4.999999583255033e-4
    print("Verificado")

# La comprobación es
#    >>> test_derivada()
#    Verificado

# ---------------------------------------------------------------------
# Ejercicio 1.2. Definir las funciones
#    derivadaBurda : Callable[[Callable[[float], float], float], float]
#    derivadaFina  : Callable[[Callable[[float], float], float], float]
#    derivadaSuper : Callable[[Callable[[float], float], float], float]
# tales que
#    * derivadaBurda(f, x) es el valor de la derivada de la función f
#      en el punto x con aproximación 0.01,
#    * (derivadaFina(f, x) es el valor de la derivada de la función f
#      en el punto x con aproximación 0.0001.
#    * (derivadaSuper(f, x) es el valor de la derivada de la función f
#      en el punto x con aproximación 0.000001.
# Por ejemplo,
#    derivadaBurda(cos, pi)  ==  0.004999958333473664
```

```
#    derivadaFina(cos, pi)   ==  4.999999969612645e-05
#    derivadaSuper(cos, pi)  ==  5.000444502911705e-07
# ----------------------------------------------------------------------

derivadaBurda: Callable[[Callable[[float], float], float], float] =\
    partial(derivada, 0.01)

derivadaFina: Callable[[Callable[[float], float], float], float] =\
    partial(derivada, 0.0001)

derivadaSuper: Callable[[Callable[[float], float], float], float] =\
    partial(derivada, 0.000001)

# Verificación
# ============

def test_derivadas() -> None:
    assert derivadaBurda(cos, pi) == 0.004999958333473664
    assert derivadaFina(cos, pi) == 4.999999969612645e-05
    assert derivadaSuper(cos, pi) == 5.000444502911705e-07
    print("Verificado")

# La comprobación es
#    >>> test_derivadas()
#    Verificado

# ----------------------------------------------------------------------
# Ejercicio 1.3. Definir la función
#    derivadaFinaDelSeno : Callable[[float], float]
# tal que derivadaFinaDelSeno(x) es el valor de la derivada fina del
# seno en x. Por ejemplo,
#    derivadaFinaDelSeno(pi) == -0.9999999983354435
# ----------------------------------------------------------------------

derivadaFinaDelSeno: Callable[[float], float] =\
    partial(derivadaFina, sin)

# Verificación
# ============
```

```
def test_derivadaFinaSeno() -> None:
    assert derivadaFinaDelSeno(pi) == -0.9999999983354435
    print("Verificado")

# La comprobación es
#    >>> test_derivadaFinaSeno()
#    Verificado

# ---------------------------------------------------------------------
# Cálculo de la raíz cuadrada                                         --
# ---------------------------------------------------------------------

# ---------------------------------------------------------------------
# Ejercicio 2.1. En los siguientes apartados de este ejercicio se va a
# calcular la raíz cuadrada de un número basándose 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, por recursión, 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
# ---------------------------------------------------------------------

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)

# Verificación
# ============

def test_raiz() -> None:
    assert raiz(9) == 3.000000001396984
    print("Verificado")

# La comprobación es
#    >>> test_raiz()
#    Verificado

# ----------------------------------------------------------------------
# Ejercicio 3.2. Definir el operador
#    casiIgual : (float, float) -> bool
# tal que casiIgual(x, y) si |x-y| < 0.001. Por ejemplo,
#    casiIgual(3.05, 3.07)        == False
#    casiIgual(3.00005, 3.00007)  == True
# ----------------------------------------------------------------------

def casiIgual(x: float, y: float) -> bool:
    return abs(x - y) < 0.001

# Verificación
# ============

def test_casiIgual() -> None:
    assert not casiIgual(3.05, 3.07)
    assert casiIgual(3.00005, 3.00007)
    print("Verificado")

# La comprobación es
#    >>> test_casiIgual()
#    Verificado

# ----------------------------------------------------------------------
# Ejercicio 3.3. Comprobar con Hypothesis que si x es positivo,
# entonces
#    casiIgual(raiz(x)**2, x)
```

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

# La propiedad es
@given(st.floats(min_value=0, max_value=100))
def test_cuadradro_de_raiz(x):
    assert casiIgual(raiz(x)**2, x)

# La comprobación es
#    >>> test_cuadradro_de_raiz()
#    >>>

# ----------------------------------------------------------------------
# Ejercicio 3.4. Definir, por iteración, la función
#    raizI : (float) -> float
# tal que raizI(x) es la raíz cuadrada de x calculada usando la
# propiedad anterior. Por ejemplo,
#    raizI(9)  ==  3.000000001396984
# ----------------------------------------------------------------------

def raizI(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_raizI() -> None:
    assert raizI(9) == 3.000000001396984
    print("Verificado")

# La comprobación es
#    >>> test_raizI()
#    Verificado
```

```
# ----------------------------------------------------------------------
# Ejercicio 3.5. Comprobar con Hypothesis que si x es positivo,
# entonces
#    casiIgual(raizI(x)**2, x)
# ----------------------------------------------------------------------

# La propiedad es
@given(st.floats(min_value=0, max_value=100))
def test_cuadrado_de_raizI(x):
    assert casiIgual(raizI(x)**2, x)

# La comprobación es
#    >>> test_cuadrado_de_raizI()
#    >>>

# ----------------------------------------------------------------------
# Ceros de una función                                                 --
# ----------------------------------------------------------------------

# ----------------------------------------------------------------------
# Ejercicio 4. 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.
# ----------------------------------------------------------------------

# ----------------------------------------------------------------------
# Ejercicio 4.1. Definir, por recursión, 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
# ----------------------------------------------------------------------

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) / derivadaFina(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)

# Verificación
# ============

def test_puntoCero () -> None:
    assert puntoCero(cos) == 1.5707963267949576
    assert puntoCero(cos) - pi/2 == 6.106226635438361e-14
    assert puntoCero(sin) == -5.8094940533562345e-15
    print("Verificado")

# La comprobación es
#    >>> test_puntoCero()
#    Verificado

# ---------------------------------------------------------------------
# Ejercicio 4.2. Definir, por iteración, la función
#    puntoCeroI : (Callable[[float], float]) -> float
# tal que puntoCeroI(f) es un cero de la función f calculado usando la
# propiedad anterior. Por ejemplo,
#    puntoCeroI(cos)  ==  1.5707963267949576
# ---------------------------------------------------------------------

def puntoCeroI(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) / derivadaFina(f, b)
    y = 1.0
    while not aceptable(y):
        y = mejora(y)
    return y
```

```
# Verificación
# ============

def test_puntoCeroI() -> None:
    assert puntoCeroI(cos) == 1.5707963267949576
    assert puntoCeroI(cos) - pi/2 == 6.106226635438361e-14
    assert puntoCeroI(sin) == -5.8094940533562345e-15
    print("Verificado")

# La comprobación es
#    >>> test_puntoCeroI()
#    Verificado

# ---------------------------------------------------------------------
# Funciones inversas                                                  --
# ---------------------------------------------------------------------

# ---------------------------------------------------------------------
# Ejercicio 5. En este ejercicio se usará la función puntoCero para
# definir la inversa de distintas funciones.
# ---------------------------------------------------------------------

# ---------------------------------------------------------------------
# Ejercicio 5.1. Definir, usando puntoCero, la función
#    raizCuadrada : (float) -> float
# tal que raizCuadrada(x) es la raíz cuadrada de x. Por ejemplo,
#    raizCuadrada(9)  ==  3.000000002941184
# ---------------------------------------------------------------------

def raizCuadrada(a: float) -> float:
    return puntoCero(lambda x : x*x-a)

# Verificación
# ============

def test_raizCuadrada() -> None:
    assert raizCuadrada(9) == 3.000000002941184
    print("Verificado")

# La comprobación es
```

```
#    >>> test_raizCuadrada()
#    Verificado

# ----------------------------------------------------------------------
# Ejercicio 5.2. Comprobar con Hypothesis que si x es positivo,
# entonces
#    casiIgual(raizCuadrada(x)**2, x)
# ----------------------------------------------------------------------

# La propiedad es
@given(st.floats(min_value=0, max_value=100))
def test_cuadrado_de_raizCuadrada(x):
    assert casiIgual(raizCuadrada(x)**2, x)

# La comprobación es
#    >>> test_cuadrado_de_raizCuadrada()
#    >>>

# ----------------------------------------------------------------------
# Ejercicio 5.3. Definir, usando puntoCero, la función
#    raizCubica : (float) -> float
# tal que raizCubica(x) es la raíz cúbica de x. Por ejemplo,
#    raizCubica(27) == 3.0000000000196048
# ----------------------------------------------------------------------

def raizCubica(a: float) -> float:
    return puntoCero(lambda x : x*x*x-a)

# Verificación
# ============

def test_raizCubica() -> None:
    assert raizCubica(27) == 3.0000000000196048
    print("Verificado")

# La comprobación es
#    >>> test_raizCubica()
#    Verificado

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

```
# Ejercicio 5.4. Comprobar con Hypothesis que si x es positivo,
# entonces
#    casiIgual(raizCubica(x)**3, x)
# ----------------------------------------------------------------------

# La propiedad es
@given(st.floats(min_value=0, max_value=100))
def test_cubo_de_raizCubica(x):
    assert casiIgual(raizCubica(x)**3, x)

# La comprobación es
#    >>> test_cubo_de_raizCubica()
#    >>>

# ----------------------------------------------------------------------
# Ejercicio 5.5. Definir, usando puntoCero, la función
#    arcoseno : (float) -> float
# tal que arcoseno(x) es el arcoseno de x. Por ejemplo,
#    arcoseno(1) == 1.5665489428306574
# ----------------------------------------------------------------------

def arcoseno(a: float) -> float:
    return puntoCero(lambda x : sin(x) - a)

# Verificación
# ============

def test_arcoseno() -> None:
    assert arcoseno(1) == 1.5665489428306574
    print("Verificado")

# La comprobación es
#    >>> test_arcoseno()
#    Verificado

# ----------------------------------------------------------------------
# Ejercicio 5.6. Comprobar con Hypothesis que si x está entre 0 y 1,
# entonces
#    casiIgual(sin(arcoseno(x)), x)
# ----------------------------------------------------------------------
```

```
# La propiedad es
@given(st.floats(min_value=0, max_value=1))
def test_seno_de_arcoseno(x):
    assert casiIgual(sin(arcoseno(x)), x)

# La comprobación es
#    >>> test_seno_de_arcoseno()
#    >>>

# ---------------------------------------------------------------------
# Ejercicio 5.7. Definir, usando puntoCero, la función
#    arcocoseno : (float) -> float
# tal que arcoseno(x) es el arcoseno de x. Por ejemplo,
#    arcocoseno(0) == 1.5707963267949576
# ---------------------------------------------------------------------

def arcocoseno(a: float) -> float:
    return puntoCero(lambda x : cos(x) - a)

# Verificación
# ============

def test_arcocoseno() -> None:
    assert arcocoseno(0) == 1.5707963267949576
    print("Verificado")

# La comprobación es
#    >>> test_arcocoseno()
#    Verificado

# ---------------------------------------------------------------------
# Ejercicio 5.8. Comprobar con Hypothesis que si x está entre 0 y 1,
# entonces
#    casiIgual(cos(arcocoseno(x)), x)
# ---------------------------------------------------------------------

# La propiedad es
@given(st.floats(min_value=0, max_value=1))
def test_coseno_de_arcocoseno(x):
```

```
    assert casiIgual(cos(arcocoseno(x)), x)

# La comprobación es
#    >>> test_coseno_de_arcocoseno()
#    >>>

# ---------------------------------------------------------------------
# Ejercicio 5.7. 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
# ---------------------------------------------------------------------

def inversa(g: Callable[[float],float], a: float) -> float:
    return puntoCero(lambda x: g(x) - a)

# Verificación
# ============

def test_inversa() -> None:
    assert inversa(lambda x: x**2, 9) == 3.000000002941184
    print("Verificado")

# La comprobación es
#    >>> test_inversa()
#    Verificado

# ---------------------------------------------------------------------
# Ejercicio 5.8. Redefinir, usando inversa, las funciones raizCuadrada,
# raizCubica, arcoseno y arcocoseno.
# ---------------------------------------------------------------------

raizCuadrada2 = partial(inversa, lambda x : x**2)
raizCubica2   = partial(inversa, lambda x : x**3)
arcoseno2     = partial(inversa, sin)
arcocoseno2   = partial(inversa, cos)

# Verificación
# ============
```

```
def test_inversas() -> None:
    assert raizCuadrada2(9) == 3.000000002941184
    assert raizCubica2(27) == 3.0000000000196048
    assert arcoseno2(1) == 1.5665489428306574
    assert arcocoseno2(0) == 1.5707963267949576
    print("Verificado")

# La comprobación es
#    >>> test_inversas()
#    Verificado

# Verificación de todo
# ====================

# La comprobación es
#    src> poetry run pytest Calculo_numerico_Diferenciacion_y_metodos_de_Heron_y_
#    ===== 20 passed in 0.88s =====
```

16.2. Cálculo numérico (2): Límites, bisección e integrales

```
# ------------------------------------------------------------------
# Introducción                                                     --
# ------------------------------------------------------------------

# En esta relación se definen funciones para resolver los siguientes
# problemas de cálculo numérico:
# + Cálculo de límites.
# + Cálculo de los ceros de una función por el método de la bisección.
# + Cálculo de raíces enteras.
# + Cálculo de integrales por el método de los rectángulos.
# + Algoritmo de bajada para resolver un sistema triangular inferior.

# ------------------------------------------------------------------
# Librerías auxiliares                                             --
# ------------------------------------------------------------------

from itertools import count, takewhile
```

```python
from math import cos, floor, log, pi
from sys import setrecursionlimit
from timeit import Timer, default_timer
from typing import Callable

from hypothesis import given
from hypothesis import strategies as st

setrecursionlimit(10**6)

# ---------------------------------------------------------------------
# Cálculo de límites                                                  --
# ---------------------------------------------------------------------

# ---------------------------------------------------------------------
# Ejercicio 1. 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
# ---------------------------------------------------------------------

# 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

# --------------------------------------------------------------------
# Ceros de una función por el método de la bisección                --
# --------------------------------------------------------------------

# --------------------------------------------------------------------
# Ejercicio 2. 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
# ----------------------------------------------------------------------

# 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

# --------------------------------------------------------------------
# Cálculo de raíces enteras                                        --
# --------------------------------------------------------------------

# --------------------------------------------------------------------
# Ejercicio 3.1. 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) ==  100000000000000000000000
# ----------------------------------------------------------------------

# 1ª solución
# ===========

def raizEnt(x: int, n: int) -> int:
    return list(takewhile(lambda y : y ** n <= x, count(0)))[-1]

# 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)
```

```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('raizEnt(10**14, 2)')
#    2.71 segundos
#    >>> tiempo('raizEnt2(10**14, 2)')
#    0.00 segundos
#    >>> tiempo('raizEnt3(10**14, 2)')
#    0.00 segundos
#
#    >>> raizEnt2(10**50, 2)
#    100000000000000000905969664
#    >>> raizEnt3(10**50, 2)
#    100000000000000000000000000

# 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

# ---------------------------------------------------------------------
# Ejercicio 3.2. Comprobar con Hypothesis que para todo número natural
# n, raizEnt(10**(2*n), 2) == 10**n
# ---------------------------------------------------------------------

# 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()
#    >>>

# ---------------------------------------------------------------------
# Integración por el método de los rectángulos                      --
# ---------------------------------------------------------------------

# ---------------------------------------------------------------------
# Ejercicio 4. 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-
```

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

# 1ª solución
# ===========

# 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, 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) == -8.333333331386
    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

# ---------------------------------------------------------------------
# Algoritmo de bajada para resolver un sistema triangular inferior   --
# ---------------------------------------------------------------------

# ---------------------------------------------------------------------
# Ejercicio 5. 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

# Verificación de todo
# ====================

# La comprobación es
#    > poetry run pytest Calculo_numerico_2_Limites_biseccion_e_integrales.py
#    ===== passed in 2.36s =====
```

Capítulo 17

Miscelánea

17.1. Números de Pentanacci

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

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).
```

17.2. El teorema de Navidad de Fermat

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

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()
#    >>>
```

17.3. Números primos de Hilbert

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

17.4. Factorizaciones de números de Hilbert

```
# ----------------------------------------------------------------------
# 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)
```

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

```python
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))
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.
```

17.5. Representaciones de un número como suma de dos cuadrados

```
# ---------------------------------------------------------------------
# Definir la función
#    representaciones : (int) -> list[tuple[int, int]]
# tal que representaciones(n) es la lista de pares de números
# naturales (x,y) tales que n = x^2 + y^2. Por ejemplo.
#    representaciones(20)          ==  [(2,4)]
#    representaciones(25)          ==  [(0,5),(3,4)]
#    representaciones(325)         ==  [(1,18),(6,17),(10,15)]
#    len(representaciones(10**14)) == 8
#
# Comprobar con Hypothesis que un número natural n se puede representar
# como suma de dos cuadrados si, y sólo si, en la factorización prima
# de n todos los exponentes de sus factores primos congruentes con 3
# módulo 4 son pares.
# ---------------------------------------------------------------------

from math import floor, sqrt
from timeit import Timer, default_timer

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

# 1ª solución
# ===========

def representaciones1(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 solución
# ===========

# raiz(x) es la raíz cuadrada entera de x. Por ejemplo,
#    raiz(25)     ==  5
#    raiz(24)     ==  4
#    raiz(26)     ==  5
#    raiz(10**46) == 10000000000000000000000
```

```python
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)
#    9999999999999991611392
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 solución
# ===========

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 representaciones1(20) == [(2,4)]
    assert representaciones1(25) == [(0,5),(3,4)]
    assert representaciones1(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 = representaciones1(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('representaciones1(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

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

# La propiedad es
@given(st.integers(min_value=1, max_value=1000))
def test_representaciones_prop(n: int) -> None:
    factores = factorint(n)
    assert (len(representaciones2(n)) > 0) == \
        all(p % 4 != 3 or e % 2 == 0 for p, e in factores.items())

# La comprobación es
#    >>> test_representaciones_prop()
```

```
#    >>>
```

17.6. La serie de Thue-Morse

```
# ---------------------------------------------------------------------
# La [serie de Thue-Morse](http://bit.ly/1KvZONW) comienza con el
# término [0] y sus siguientes términos se construyen añadiéndole al
# anterior su complementario (es decir, la lista obtenida cambiando el
# 0 por 1 y el 1 por 0). Los primeros términos de la serie son
#    [0]
#    [0,1]
#    [0,1,1,0]
#    [0,1,1,0,1,0,0,1]
#    [0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0]
#
# Definir la función
#    serieThueMorse : () -> Iterator[list[list[int]]]
# tal que sus elementos son los términos de la serie de Thue-Morse. Por
# ejemplo,
#    >>> list(islice(serieThueMorse(), 4))
#    [[0], [0, 1], [0, 1, 1, 0], [0, 1, 1, 0, 1, 0, 0, 1]]
# ---------------------------------------------------------------------

from itertools import count, islice
from typing import Iterator

# Solución
# ========

# complementaria(xs) es la complementaria de la lista xs (formada por
# ceros y unos); es decir, la lista obtenida cambiando el 0 por 1 y el
# 1 por 0. Por ejemplo,
#    complementaria([1,0,0,1,1,0,1])  ==  [0,1,1,0,0,1,0]
def complementaria(xs: list[int]) -> list[int]:
    return [1 - x for x in xs]

# termSerieThueMorse(n) es el término n-ésimo de la serie de
# Thue-Morse. Por ejemplo,
#    termSerieThueMorse(1)  ==  [0,1]
#    termSerieThueMorse(2)  ==  [0,1,1,0]
```

```
#    termSerieThueMorse(3)  ==  [0,1,1,0,1,0,0,1]
#    termSerieThueMorse(4)  ==  [0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0]
def termSerieThueMorse(n: int) -> list[int]:
    if n == 0:
        return [0]
    xs = termSerieThueMorse(n-1)
    return xs + complementaria(xs)

def serieThueMorse() -> Iterator[list[int]]:
    return (termSerieThueMorse(n) for n in count())

# Verificación
# ============

def test_serieThueMorse() -> None:
    assert list(islice(serieThueMorse(), 4)) == \
        [[0], [0, 1], [0, 1, 1, 0], [0, 1, 1, 0, 1, 0, 0, 1]]
    print("Verificado")

# La verificación es
#    >>> test_serieThueMorse()
#    Verificado

# ----------------------------------------------------------------------
# § Referencias                                                        --
# ----------------------------------------------------------------------

# + N.J.A. Sloane "Sucesión A010060" en OEIS http://oeis.org/A010060
# + Programming Praxis "Thue-Morse sequence" http://bit.ly/1n2PdFk
# + Wikipedia "Thue–Morse sequence" http://bit.ly/1KvZONW
```

17.7. La sucesión de Thue-Morse

```
# ----------------------------------------------------------------------
# La serie de Thue-Morse comienza con el término [0] y sus siguientes
# términos se construyen añadiéndole al anterior su complementario. Los
# primeros términos de la serie son
#    [0]
#    [0,1]
#    [0,1,1,0]
```

```
#    [0,1,1,0,1,0,0,1]
#    [0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0]
# De esta forma se va formando una sucesión
#    0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0,...
# que se conoce como la [sucesión de Thue-Morse](https://bit.ly/3PE9LRJ).
#
# Definir la sucesión
#    sucThueMorse : () -> Iterator[int]
# cuyos elementos son los de la sucesión de Thue-Morse. Por ejemplo,
#    >>> list(islice(sucThueMorse(), 30))
#    [0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0,1,0,0,1,0,1,1,0,0,1,1,0,1,0]
#
# Comprobar con Hypothesis que si s(n) representa el término n-ésimo de
# la sucesión de Thue-Morse, entonces
#    s(2n)   = s(n)
#    s(2n+1) = 1 - s(n)
# ----------------------------------------------------------------------

from itertools import count, islice
from math import floor, log2
from timeit import Timer, default_timer
from typing import Iterator, TypeVar

from hypothesis import given
from hypothesis import strategies as st

from src.La_serie_de_Thue_Morse import serieThueMorse

A = TypeVar('A')

# 1ª solución
# ===========

# nth(i, n) es el n-ésimo elemento del iterador i.
def nth(i: Iterator[A], n: int) -> A:
    return list(islice(i, n, n+1))[0]

# termSucThueMorse(n) es el n-ésimo término de la sucesión de
# Thue-Morse. Por ejemplo,
#    termSucThueMorse(0)  ==  0
```

```
#    termSucThueMorse(1)  ==  1
#    termSucThueMorse(2)  ==  1
#    termSucThueMorse(3)  ==  0
#    termSucThueMorse(4)  ==  1
def termSucThueMorse(n: int) -> int:
    if n == 0:
        return 0
    k = 1 + floor(log2(n))
    return nth(serieThueMorse(), k)[n]

def sucThueMorse() -> Iterator[int]:
    return (termSucThueMorse(n) for n in count())

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

# La propiedad es
@given(st.integers(min_value=0, max_value=100))
def test_prop_termSucThueMorse(n: int) -> None:
    sn = nth(sucThueMorse(), n)
    assert nth(sucThueMorse(), 2*n) == sn
    assert nth(sucThueMorse(), 2*n+1) == 1 - sn

# La comprobación es
#    >>> test_prop_termSucThueMorse()
#    >>>

# 2ª solución
# ===========

# termSucThueMorse2(n) es el n-ésimo término de la sucesión de
# Thue-Morse. Por ejemplo,
#    termSucThueMorse2(0)  ==  0
#    termSucThueMorse2(1)  ==  1
#    termSucThueMorse2(2)  ==  1
#    termSucThueMorse2(3)  ==  0
#    termSucThueMorse2(4)  ==  1
def termSucThueMorse2(n: int) -> int:
    if n == 0:
        return 0
```

```
    if n % 2 == 0:
        return termSucThueMorse2(n // 2)
    return 1 - termSucThueMorse2(n // 2)

def sucThueMorse2() -> Iterator[int]:
    return (termSucThueMorse2(n) for n in count())

# Verificación
# ============

def test_sucThueMorse() -> None:
    r = [0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0,1,0,0,1,0,1,1,0,0,1,1,0,1,0]
    assert list(islice(sucThueMorse(), 30)) == r
    assert list(islice(sucThueMorse2(), 30)) == r
    print("Verificado")

# La verificación es
#    >>> test_sucThueMorse()
#    Verificado

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

# La propiedad es
@given(st.integers(min_value=0, max_value=100))
def test_sucThueMorse_equiv(n: int) -> None:
    assert nth(sucThueMorse(), n) == nth(sucThueMorse2(), n)

# La comprobación es
#    >>> test_sucThueMorse_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(sucThueMorse(), 6000)')
#    2.05 segundos
#    >>> tiempo('nth(sucThueMorse2(), 6000)')
#    0.01 segundos

# ----------------------------------------------------------------------
# § Referencias                                                        --
# ----------------------------------------------------------------------

# + N.J.A. Sloane "Sucesión A010060" en OEIS http://oeis.org/A010060
# + Programming Praxis "Thue-Morse sequence" http://bit.ly/1n2PdFk
# + Wikipedia "Thue–Morse sequence" http://bit.ly/1KvZONW
```

17.8. Huecos maximales entre primos

```
# ----------------------------------------------------------------------
# El **hueco de un número primo** p es la distancia entre p y primo
# siguiente de p. Por ejemplo, el hueco de 7 es 4 porque el primo
# siguiente de 7 es 11 y 4 = 11-7. Los huecos de los primeros números son
#    Primo Hueco
#     2    1
#     3    2
#     7    4
#    11    2
#
# El hueco de un número primo p es **maximal** si es mayor que los
# huecos de todos los números menores que p. Por ejemplo, 4 es un hueco
# maximal de 7 ya que los huecos de los primos menores que 7 son 1 y 2
# y ambos son menores que 4. La tabla de los primeros huecos maximales es
#    Primo Hueco
#      2    1
#      3    2
#      7    4
#     23    6
#     89    8
#    113   14
#    523   18
#    887   20
#
```

```python
# Definir la sucesión
#    primosYhuecosMaximales : () -> Iterator[tuple[int, int]]
# cuyos elementos son los números primos con huecos maximales junto son
# sus huecos. Por ejemplo,
#    >>> list(islice(primosYhuecosMaximales1(), 8))
#    [(2,1),(3,2),(7,4),(23,6),(89,8),(113,14),(523,18),(887,20)]
# ----------------------------------------------------------------------

from itertools import count, islice, pairwise, takewhile
from timeit import Timer, default_timer
from typing import Iterator

from sympy import isprime, nextprime

# 1ª solución
# ===========

# 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))

# huecoPrimo(p) es la distancia del primo p hasta el siguiente
# primo. Por ejemplo,
#    huecoPrimo(7)  ==  4
def huecoPrimo(p: int) -> int:
    return nextprime(p) - p

# esMaximalHuecoPrimo(p) se verifica si el hueco primo de p es
# maximal. Por ejemplo,
#    esMaximalHuecoPrimo(7)   ==  True
#    esMaximalHuecoPrimo(11)  ==  False
def esMaximalHuecoPrimo(p: int) -> bool:
    h = huecoPrimo(p)
    return all(huecoPrimo(n) < h for n in takewhile(lambda x: x < p, primos()))

def primosYhuecosMaximales1() -> Iterator[tuple[int, int]] :
    return ((p,huecoPrimo(p)) for p in primos() if esMaximalHuecoPrimo(p))
```

```python
# 2ª solución
# ===========

# primosYhuecos es la lista de los números primos junto son sus
# huecos. Por ejemplo,
#    >>> list(islice(primosYhuecos(), 10))
#    [(2,1),(3,2),(5,2),(7,4),(11,2),(13,4),(17,2),(19,4),(23,6),(29,2)]
def primosYhuecos() -> Iterator[tuple[int, int]]:
    return ((x,y-x) for (x,y) in pairwise(primos()))

def primosYhuecosMaximales2() -> Iterator[tuple[int, int]]:
    n = 0
    for (x,y) in primosYhuecos():
        if y > n:
            yield (x,y)
            n = y

# Verificación
# ============

def test_primosYhuecosMaximales() -> None:
    r = [(2,1),(3,2),(7,4),(23,6),(89,8),(113,14),(523,18),(887,20)]
    assert list(islice(primosYhuecosMaximales1(), 8)) == r
    assert list(islice(primosYhuecosMaximales2(), 8)) == r
    print("Verificado")

# La verificación es
#    >>> test_primosYhuecosMaximales()
#    Verificado


# 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('list(islice(primosYhuecosMaximales1(), 15))')
#    8.08 segundos
#    >>> tiempo('list(islice(primosYhuecosMaximales2(), 15))')
#    0.17 segundos

# ------------------------------------------------------------------------
# § Referencias                                                         --
# ------------------------------------------------------------------------

# Basado en el ejercicio "Maximal prime gaps" http://bit.ly/22UfDJN de
# Programming Praxis http://programmingpraxis.com

# Otras referencias
# + C. Caldwell, "The gaps between primes" http://bit.ly/1Znusp5
# + J.K. Andersen, "Maximal prime gaps" http://bit.ly/1ZntwRi
# + N.J.A. Sloane "Sequence A002386" en OEIS http://oeis.org/A002386
# + N.J.A. Sloane "Sequence A005250" en OEIS http://oeis.org/A005250
# + E.W. Weisstein, "Prime gaps" en MathWorld http://bit.ly/1ZnubCq
```

17.9. La función indicatriz de Euler

```
# ------------------------------------------------------------------------
# La [indicatriz de Euler](https://bit.ly/3yQbzA6) (también llamada
# función φ de Euler) es una función importante en teoría de
# números. Si n es un entero positivo, entonces φ(n) se define como el
# número de enteros positivos menores o iguales a n y coprimos con
# n. Por ejemplo, φ(36) = 12 ya que los números menores o iguales a 36
# y coprimos con 36 son doce: 1, 5, 7, 11, 13, 17, 19, 23, 25, 29, 31,
# y 35.
#
# Definir la función
#    phi : (int) -> int
# tal que phi(n) es igual a φ(n). Por ejemplo,
#    phi(36)                          ==  12
#    list(map(phi, range(10, 21)))    ==  [4,10,4,12,6,8,8,16,6,18,8]
#    phi(3**10**5) % (10**9)          ==  681333334
#    len(str(phi2(10**(10**5))))      == 100000
#
# Comprobar con Hypothesis que, para todo n > 0, φ(10^n) tiene n
# dígitos.
```

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

from functools import reduce
from math import gcd
from operator import mul
from sys import set_int_max_str_digits
from timeit import Timer, default_timer

from hypothesis import given
from hypothesis import strategies as st
from sympy import factorint, totient

set_int_max_str_digits(10**6)

# 1ª solución
# ===========

def phi1(n: int) -> int:
    return len([x for x in range(1, n+1) if gcd(x, n) == 1])

# 2ª solución
# ===========

def producto(xs: list[int]) -> int:
    return reduce(mul, xs, 1)

def phi2(n: int) -> int:
    factores = factorint(n)
    return producto([(p-1)*p**(e-1) for p, e in factores.items()])

# 3ª solución
# ============

def phi3(n: int) -> int:
    return totient(n)

# Verificación
# ============

def test_phi() -> None:
```

```
    for phi in [phi1, phi2, phi3]:
        assert phi(36) == 12
        assert list(map(phi, range(10, 21))) == [4,10,4,12,6,8,8,16,6,18,8]
    print("Verificado")

# La verificación es
#    >>> test_phi()
#    Verificado

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

# La propiedad es
@given(st.integers(min_value=1, max_value=1000))
def test_phi_equiv(n: int) -> None:
    r = phi1(n)
    assert phi2(n) == r
    assert phi3(n) == r

# La comprobación es
#    >>> test_phi_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('phi1(9*10**6)')
#    2.09 segundos
#    >>> tiempo('phi2(9*10**6)')
#    0.00 segundos
#    >>> tiempo('phi3(9*10**6)')
#    0.00 segundos
#
#    >>> tiempo('phi2(10**1000000)')
```

```
#    3.55 segundos
#    >>> tiempo('phi3(10**1000000)')
#    3.37 segundos

# Verificación de la propiedad
# ============================

# La propiedad es
@given(st.integers(min_value=1, max_value=1000))
def test_phi_prop(n: int) -> None:
    assert len(str(phi2(10**n))) == n

# La comprobación es
#    >>> test_phi_prop()
#    >>>

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

# La comprobación es
#    src> poetry run pytest -v La_funcion_indicatriz_de_Euler.py
#    ===== test session starts =====
#       test_phi PASSED
#       test_phi_equiv PASSED
#       test_phi_prop PASSED
#    ===== passed in 0.55s =====
```

17.10. Ceros finales del factorial

```
# ----------------------------------------------------------------------
# Definir la función
#    cerosDelFactorial : (int) -> int
# tal que cerosDelFactorial(n) es el número de ceros en que termina el
# factorial de n. Por ejemplo,
#    cerosDelFactorial(24)                  == 4
#    cerosDelFactorial(25)                  == 6
#    len(str(cerosDelFactorial(10**70000))) == 70000
# ----------------------------------------------------------------------

from itertools import takewhile
```

```
from math import factorial
from sys import set_int_max_str_digits
from timeit import Timer, default_timer

from hypothesis import given
from hypothesis import strategies as st

set_int_max_str_digits(10**6)

# 1ª solución
# ===========

# ceros(n) es el número de ceros en los que termina el número n. Por
# ejemplo,
#    ceros(320000)  ==  4
def ceros(n: int) -> int:
    r = 0
    while n % 10 == 0 and n != 0:
        r += 1
        n //= 10
    return r

def cerosDelFactorial1(n: int) -> int:
    return ceros(factorial(n))

# 2ª solución
# ===========

def ceros2(n: int) -> int:
    return len(list(takewhile(lambda x: x == '0', reversed(str(n)))))

def cerosDelFactorial2(n: int) -> int:
    return ceros2(factorial(n))

# 3ª solución
# ============

def cerosDelFactorial3(n: int) -> int:
    r = 0
    while n >= 5:
```

```
        n = n // 5
        r += n
    return r

# Verificación
# ============

def test_cerosDelFactorial() -> None:
    for cerosDelFactorial in [cerosDelFactorial1,
                              cerosDelFactorial2,
                              cerosDelFactorial3]:
        assert cerosDelFactorial(24) == 4
        assert cerosDelFactorial(25) == 6
    print("Verificado")

# La verificación es
#    >>> test_cerosDelFactorial()
#    Verificado

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

# La propiedad es
@given(st.integers(min_value=0, max_value=1000))
def test_cerosDelFactorial_equiv(n: int) -> None:
    r = cerosDelFactorial1(n)
    assert cerosDelFactorial2(n) == r
    assert cerosDelFactorial3(n) == r

# La comprobación es
#    >>> test_cerosDelFactorial_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('cerosDelFactorial1(6*10**4)')
#    2.47 segundos
#    >>> tiempo('cerosDelFactorial2(6*10**4)')
#    0.77 segundos
#    >>> tiempo('cerosDelFactorial3(6*10**4)')
#    0.00 segundos

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

# La comprobación es
#    src> poetry run pytest -v Ceros_finales_del_factorial.py
#       test_cerosDelFactorial PASSED
#       test_cerosDelFactorial_equiv PASSED
```

17.11. Primos cubanos

```
# ----------------------------------------------------------------------
# Un [primo cubano](http://bit.ly/1jPy5QZ) es un número primo que se
# puede escribir como diferencia de dos cubos consecutivos. Por
# ejemplo, el 61 es un primo cubano porque es primo y 61 = 5³-4³.
#
# Definir la sucesión
#    cubanos : () -> Iterator[int]
# tal que sus elementos son los números cubanos. Por ejemplo,
#    >>> list(islice(cubanos(), 15))
#    [7,19,37,61,127,271,331,397,547,631,919,1657,1801,1951,2269]
# ----------------------------------------------------------------------

from itertools import count, islice, tee
from timeit import Timer, default_timer
from typing import Iterator

from sympy import isprime

# 1ª solución
# ===========
```

```python
# cubos() es la lista de los cubos. Por ejemplo,
#    >>> list(islice(cubos(), 10))
#    [1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
def cubos() -> Iterator[int]:
    return (x**3 for x in count(1))

# parejasDeCubos() es la lista de las parejas de cubos consecutivos. Por
# ejemplo,
#    >>> list(islice(parejasDeCubos(), 5))
#    [(1, 8), (8, 27), (27, 64), (64, 125), (125, 216)]
def parejasDeCubos() -> Iterator[tuple[int, int]]:
    a, b = tee(cubos())
    next(b, None)
    return zip(a, b)

# diferenciasDeCubos() es la lista de las diferencias de cubos
# consecutivos. Por ejemplo,
#    >>> list(islice(diferenciasDeCubos(), 5))
#    [7, 19, 37, 61, 91]
def diferenciasDeCubos() -> Iterator[int]:
    return (j - i for i, j in parejasDeCubos())

def cubanos1() -> Iterator[int]:
    return (x for x in diferenciasDeCubos() if isprime(x))

# 2ª solución
# ===========

def cubanos2() -> Iterator[int]:
    return ((x+1)**3 - x**3 for x in count(1) if isprime((x+1)**3 - x**3))

# 3ª solución
# ===========

def cubanos3() -> Iterator[int]:
    return (y for x in count(1) if isprime((y := (x+1)**3 - x**3)))

# 4ª solución
# ===========
```

```
def cubanos4() -> Iterator[int]:
    return (y for x in count(1) if isprime((y := 3*x**2 + 3*x + 1)))

# Verificación
# ============

def test_cubanos() -> None:
    for cubanos in [cubanos1, cubanos2, cubanos3, cubanos4]:
        assert list(islice(cubanos(), 15)) == \
            [7,19,37,61,127,271,331,397,547,631,919,1657,1801,1951,2269]
    print ("Verificado")

# La verificación es
#    >>> test_cubanos()
#    Verificado

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

# La propiedad es
def test_cubanos_equiv(n: int) -> None:
    r = list(islice(cubanos1(), n))
    assert list(islice(cubanos2(), n)) == r
    assert list(islice(cubanos3(), n)) == r
    assert list(islice(cubanos4(), n)) == r
    print("Verificado")

# La comprobación es
#    >>> test_cubanos_equiv(10000)
#    Verificado

# 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('list(islice(cubanos1(), 30000))')
#    1.54 segundos
#    >>> tiempo('list(islice(cubanos1(), 40000))')
#    2.20 segundos
#    >>> tiempo('list(islice(cubanos2(), 40000))')
#    2.22 segundos
#    >>> tiempo('list(islice(cubanos3(), 40000))')
#    2.19 segundos
#    >>> tiempo('list(islice(cubanos4(), 40000))')
#    2.15 segundos
```

17.12. Cuadrado más cercano

```
# ---------------------------------------------------------------------
# Definir la función
#    cuadradoCercano : (int) -> int
# tal que cuadradoCercano(n) es el número cuadrado más cercano a n,
# donde n es un entero positivo. Por ejemplo,
#    cuadradoCercano(2)     == 1
#    cuadradoCercano(6)     == 4
#    cuadradoCercano(8)     == 9
#    cuadradoCercano(10^46) == 10000000000000000000000000000000000000000000
# ---------------------------------------------------------------------

from itertools import count, takewhile
from math import sqrt
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
# ===========

# raizEntera(x) es el mayor entero cuyo cuadrado no es mayor que
# x. Por ejemplo,
#    raizEntera(8)   ==  2
```

```
#    raizEntera(9)   ==  3
#    raizEntera(10)  ==  3
def raizEntera1(x: int) -> int:
    return list(takewhile(lambda n: n**2 <= x, count(1)))[-1]

def cuadradoCercano1(n: int) -> int:
    a = raizEntera1(n)
    b = a**2
    c = (a+1)**2
    if n - b < c - n:
        return b
    return c

# 2ª solución
# ===========

def raizEntera2(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 c
        if x <= d:
            return aux(a,c)
        return aux(c,b)
    return aux(1,x)

def cuadradoCercano2(n: int) -> int:
    a = raizEntera2(n)
    b = a**2
    c = (a+1)**2
    if n - b < c - n:
        return b
    return c

# 3ª solución
# ===========
```

```
def raizEntera3(n: int) -> int:
    if n == 0:
        return 0
    if n == 1:
        return 1
    x = n
    while x * x > n:
        x = (x + n // x) // 2
    return x

def cuadradoCercano3(n: int) -> int:
    a = raizEntera3(n)
    b = a**2
    c = (a+1)**2
    if n - b < c - n:
        return b
    return c

# 4ª solución
# ===========

def cuadradoCercano4(n: int) -> int:
    return round(sqrt(n)) ** 2

# La 4ª solución es incorrecta. Por ejemplo,
#    >>> cuadradoCercano4(10**46)
#    99999999999999983222784000000000703687441776 64
#    >>> cuadradoCercano3(10**46)
#    10000000000000000000000000000000000000000000

# Verificación
# ============

def test_cuadradoCercano() -> None:
    for cuadradoCercano in [cuadradoCercano1, cuadradoCercano2,
                            cuadradoCercano3, cuadradoCercano4]:
        assert cuadradoCercano(2) == 1
        assert cuadradoCercano(6) == 4
        assert cuadradoCercano(8) == 9
    print("Verificado")
```

```
# La verificación es
#    >>> test_cuadradoCercano()
#    Verificado

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

# La propiedad es
@given(st.integers(min_value=1, max_value=1000))
def test_cuadradoCercano_equiv(x: int) -> None:
    r = cuadradoCercano1(x)
    assert cuadradoCercano2(x) == r
    assert cuadradoCercano3(x) == r
    assert cuadradoCercano4(x) == r

# La comprobación es
#    >>> test_cuadradoCercano_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('cuadradoCercano1(10**14)')
#    2.88 segundos
#    >>> tiempo('cuadradoCercano2(10**14)')
#    0.00 segundos
#    >>> tiempo('cuadradoCercano3(10**14)')
#    0.00 segundos
#
#    >>> tiempo('cuadradoCercano2(10**6000)')
#    1.21 segundos
#    >>> tiempo('cuadradoCercano3(10**6000)')
#    2.08 segundos
```

17.13. Suma de cadenas

```
# ---------------------------------------------------------------------
# Definir la función
#    sumaCadenas : (str, str) -> str
# tal que sumaCadenas(xs, ys) es la cadena formada por el número entero
# que es la suma de los números enteros cuyas cadenas que lo
# representan son xs e ys; además, se supone que la cadena vacía
# representa al cero. Por ejemplo,
#    sumaCadenas("2",   "6")  == "8"
#    sumaCadenas("14",  "2")  == "16"
#    sumaCadenas("14",  "-5") == "9"
#    sumaCadenas("-14", "-5") == "-19"
#    sumaCadenas("5",   "-5") == "0"
#    sumaCadenas("",    "5")  == "5"
#    sumaCadenas("6",   "")   == "6"
#    sumaCadenas("",    "")   == "0"
# ---------------------------------------------------------------------

# 1ª solución
# ===========

def sumaCadenas1(xs: str, ys: str) -> str:
    return str(sum(map(int, filter(lambda x: x != '', [xs, ys]))))

# 2ª solución
# ===========

def sumaCadenas2(xs: str, ys: str) -> str:
    if xs == "" and ys == "":
        return "0"
    if xs == "":
        return ys
    if ys == "":
        return xs
    return str(int(xs) + int(ys))

# 3ª solución
# ===========
```

```
# numero(xs) es el número entero representado por la cadena xs
# suponiendo que la cadena vacía representa al cero.. Por ejemplo,
#    numero "12"   ==  12
#    numero "-12"  ==  -12
#    numero "0"    ==  0
#    numero ""     ==  0
def numero(s: str) -> int:
    if not s:
        return 0
    return int(s)

def sumaCadenas3(xs: str, ys: str) -> str:
    return str(numero(xs) + numero(ys))

# 4ª solución
# ===========

def sumaCadenas4(xs: str, ys: str) -> str:
    x = int(xs or "0")
    y = int(ys or "0")
    return str(x + y)

# Verificación
# ============

def test_sumaCadenas() -> None:
    for sumaCadenas in [sumaCadenas1, sumaCadenas2, sumaCadenas3,
                        sumaCadenas4]:
        assert sumaCadenas("2",   "6")  == "8"
        assert sumaCadenas("14",  "2")  == "16"
        assert sumaCadenas("14",  "-5") == "9"
        assert sumaCadenas("-14", "-5") == "-19"
        assert sumaCadenas("5",   "-5") == "0"
        assert sumaCadenas("",    "5")  == "5"
        assert sumaCadenas("6",   "")   == "6"
        assert sumaCadenas("",    "")   == "0"
    print("Verificado")

# La verificación es
#    >>> test_sumaCadenas()
```

```
#    Verificado
```

17.14. Sistema factorádico de numeración

```
# ----------------------------------------------------------------------
# El [sistema factorádico](https://bit.ly/3KQZRue) es un sistema
# numérico basado en factoriales en el que el n-ésimo dígito, empezando
# desde la derecha, debe ser multiplicado por n! Por ejemplo, el número
# "341010" en el sistema factorádico es 463 en el sistema decimal ya
# que
#    3×5! + 4×4! + 1×3! + 0×2! + 1×1! + 0×0! = 463
#
# En este sistema numérico, el dígito de más a la derecha es siempre 0,
# el segundo 0 o 1, el tercero 0,1 o 2 y así sucesivamente.
#
# Con los dígitos del 0 al 9 el mayor número que podemos codificar es el
# 10!-1 = 3628799. En cambio, si lo ampliamos con las letras A a Z podemos
# codificar hasta 36!-1 = 371993326789901217467999448150835199999999910.
#
# Definir las funciones
#    factoradicoAdecimal : (str) -> int
#    decimalAfactoradico : int -> str
# tales que
# + factoradicoAdecimal(cs) es el número decimal correspondiente al
#   número factorádico cs. Por ejemplo,
#      >>> factoradicoAdecimal("341010")
#      463
#      >>> factoradicoAdecimal("2441000")
#      2022
#      >>> factoradicoAdecimal("A0000000000")
#      36288000
#      >>> list(map(factoradicoAdecimal, ["10","100","110","200","210","1000","10
#      [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
#      >>> factoradicoAdecimal("3KXWVUTSRQPONMLKJIHGFEDCBA9876543210")
#      371993326789901217467999448150835199999999
# + decimalAfactoradico(n) es el número factorádico correpondiente al
#   número decimal n. Por ejemplo,
#      >>> decimalAfactoradico(463)
#      '341010'
#      >>> decimalAfactoradico(2022)
```

```
#      '2441000'
#      >>> decimalAfactoradico(36288000)
#      'A0000000000'
#      >>> list(map(decimalAfactoradico, range(1, 11)))
#      ['10', '100', '110', '200', '210', '1000', '1010', '1100', '1110', '1200']
#      >>> decimalAfactoradico(3719933267899012174679994481508351999999)
#      '3KXWVUTSRQPONMLKJIHGFEDCBA9876543210'
#
# Comprobar con Hypothesis que, para cualquier entero positivo n,
#    factoradicoAdecimal(decimalAfactoradico(n)) == n
# ----------------------------------------------------------------------

from itertools import count, takewhile
from math import factorial
from typing import Iterator

from hypothesis import given
from hypothesis import strategies as st

# 1ª definición de factoradicoAdecimal
# ====================================

# caracterAentero(c) es la posición del carácter c en la lista de
# caracteres ['0', '1',..., '9', 'A', 'B',..., 'Z']. Por ejemplo,
#    caracterAentero('0')  ==  0
#    caracterAentero('1')  ==  1
#    caracterAentero('9')  ==  9
#    caracterAentero('A')  ==  10
#    caracterAentero('B')  ==  11
#    caracterAentero('Z')  ==  35
def caracterAentero(c: str) -> int:
    if c.isdigit():
        return int(c)
    return ord(c) - ord('A') +  10

def factoradicoAdecimal1(cs: str) -> int:
    xs = map(caracterAentero, cs)
    n  = len(cs)
    ys = reversed([factorial(k) for k in range(n)])
    return sum((x * y for (x, y) in zip(xs, ys)))
```

```
# 2ª definición de factoradicoAdecimal
# ====================================

# caracteres es la cadena de los caracteres.
caracteres: str = '0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ'

# caracteresEnteros es el diccionario cuyas claves son los caracteres y
# los valores son los números de 0 a 35.
caracteresEnteros: dict[str, int] = {c: i for i, c in enumerate(caracteres)}

# caracterAentero2(c) es la posición del carácter c en la lista de
# caracteres ['0', '1',..., '9', 'A', 'B',..., 'Z']. Por ejemplo,
#    caracterAentero2('0')  ==  0
#    caracterAentero2('1')  ==  1
#    caracterAentero2('9')  ==  9
#    caracterAentero2('A')  ==  10
#    caracterAentero2('B')  ==  11
#    caracterAentero2('Z')  ==  35
def caracterAentero2(c: str) -> int:
    return caracteresEnteros[c]

def factoradicoAdecimal2(cs: str) -> int:
    xs = map(caracterAentero2, cs)
    n  = len(cs)
    ys = reversed([factorial(k) for k in range(n)])
    return sum((x * y for (x, y) in zip(xs, ys)))

# 3ª definición de factoradicoAdecimal
# ====================================

# caracterAentero3(c) es la posición del carácter c en la lista de
# caracteres ['0', '1',..., '9', 'A', 'B',..., 'Z']. Por ejemplo,
#    caracterAentero3('0')  ==  0
#    caracterAentero3('1')  ==  1
#    caracterAentero3('9')  ==  9
#    caracterAentero3('A')  ==  10
#    caracterAentero3('B')  ==  11
#    caracterAentero3('Z')  ==  35
def caracterAentero3(c: str) -> int:
```

```
    return len(list(takewhile(lambda x: x != c, caracteres)))

def factoradicoAdecimal3(cs: str) -> int:
    return sum(x * y for x, y in zip([factorial(k) for k in range(len(cs))],
                                     reversed(list(map(caracterAentero3, cs)))))

# 1ª definición de decimalAfactoradico
# ====================================

# enteroAcaracter(k) es el k-ésimo elemento de la lista
# ['0', '1',..., '9', 'A', 'B',..., 'Z']. . Por ejemplo,
#    enteroAcaracter(0)   ==  '0'
#    enteroAcaracter(1)   ==  '1'
#    enteroAcaracter(9)   ==  '9'
#    enteroAcaracter(10)  ==  'A'
#    enteroAcaracter(11)  ==  'B'
#    enteroAcaracter(35)  ==  'Z'
def enteroAcaracter(k: int) -> str:
    return caracteres[k]

# facts() es la lista de los factoriales. Por ejemplo,
#    >>> list(takewhile(lambda x : x < 900, facts()))
#    [1, 1, 2, 6, 24, 120, 720]
def facts() -> Iterator[int]:
    return (factorial(n) for n in count())

def decimalAfactoradico1(n: int) -> str:
    def aux(m: int, xs: list[int]) -> str:
        if m == 0:
            return "0" * len(xs)
        y, *ys = xs
        print(m, y, m // y)
        return enteroAcaracter(m // y) + aux(m % y, ys)
    return aux(n, list(reversed(list(takewhile(lambda x : x <= n, facts())))))

# 2ª definición de decimalAfactoradico
# ====================================

# enterosCaracteres es el diccionario cuyas claves son los número de 0
# a 35 y las claves son los caracteres.
```

```python
enterosCaracteres: dict[int, str] = dict(enumerate(caracteres))

# enteroAcaracter2(k) es el k-ésimo elemento de la lista
# ['0', '1',..., '9', 'A', 'B',..., 'Z']. . Por ejemplo,
#    enteroAcaracter2(0)   ==  '0'
#    enteroAcaracter2(1)   ==  '1'
#    enteroAcaracter2(9)   ==  '9'
#    enteroAcaracter2(10)  ==  'A'
#    enteroAcaracter2(11)  ==  'B'
#    enteroAcaracter2(35)  ==  'Z'
def enteroAcaracter2(k: int) -> str:
    return enterosCaracteres[k]

def decimalAfactoradico2(n: int) -> str:
    def aux(m: int, xs: list[int]) -> str:
        if m == 0:
            return "0" * len(xs)
        y, *ys = xs
        return enteroAcaracter2(m // y) + aux(m % y, ys)
    return aux(n, list(reversed(list(takewhile(lambda x : x <= n, facts())))))

# 3ª definición de decimalAfactoradico
# ====================================

# enteroAcaracter3(k) es el k-ésimo elemento de la lista
# ['0', '1',..., '9', 'A', 'B',..., 'Z']. . Por ejemplo,
#    enteroAcaracter3(0)   ==  '0'
#    enteroAcaracter3(1)   ==  '1'
#    enteroAcaracter3(9)   ==  '9'
#    enteroAcaracter3(10)  ==  'A'
#    enteroAcaracter3(11)  ==  'B'
#    enteroAcaracter3(35)  ==  'Z'
def enteroAcaracter3(n: int) -> str:
    return caracteres[n]

def decimalAfactoradico3(n: int) -> str:
    def aux(m: int, xs: list[int]) -> str:
        if m == 0:
            return "0" * len(xs)
        y, *ys = xs
```

```
        return enteroAcaracter3(m // y) + aux(m % y, ys)
    return aux(n, list(reversed(list(takewhile(lambda x : x <= n, facts())))))

# Verificación
# ============

def test_factoradico() -> None:
    for factoradicoAdecimal in [factoradicoAdecimal1,
                                factoradicoAdecimal2,
                                factoradicoAdecimal3]:
        assert factoradicoAdecimal("341010") == 463
        assert factoradicoAdecimal("2441000") == 2022
        assert factoradicoAdecimal("A0000000000") == 36288000
    for decimalAfactoradico in [decimalAfactoradico1,
                                decimalAfactoradico2,
                                decimalAfactoradico3]:
        assert decimalAfactoradico(463) == "341010"
        assert decimalAfactoradico(2022) == "2441000"
        assert decimalAfactoradico(36288000) == "A0000000000"
    print("Verificado")

# La verificación es
#    >>> test_factoradico()
#    Verificado

# Propiedad de inverso
# ====================

@given(st.integers(min_value=0, max_value=1000))
def test_factoradico_equiv(n: int) -> None:
    assert factoradicoAdecimal1(decimalAfactoradico1(n)) == n
    assert factoradicoAdecimal2(decimalAfactoradico3(n)) == n
    assert factoradicoAdecimal3(decimalAfactoradico3(n)) == n

# La comprobación es
#    >>> test_factoradico_equiv()
#    >>>
```

17.15. Duplicación de cada elemento

```
# ----------------------------------------------------------------------
# Definir la función
#    duplicaElementos : (list[A]) -> list[A]
# tal que duplicaElementos(xs) es la lista obtenida duplicando cada
# elemento de xs. Por ejemplo,
#    >>> duplicaElementos([3,2,5])
#    [3, 3, 2, 2, 5, 5]
#    >>> "".join(duplicaElementos("Haskell"))
#    'HHaasskkeellll'
# ----------------------------------------------------------------------

from functools import reduce
from typing import TypeVar

from hypothesis import given
from hypothesis import strategies as st

A = TypeVar('A')

# 1ª solución
# ===========

def duplicaElementos1(ys: list[A]) -> list[A]:
    if not ys:
        return []
    x, *xs = ys
    return [x, x] + duplicaElementos1(xs)

# 2 solución
# ===========

def duplicaElementos2(xs: list[A]) -> list[A]:
    return reduce(lambda ys, x: ys + [x, x], xs, [])

# 3ª solución
# ===========

def duplicaElementos3(xs: list[A]) -> list[A]:
```

```
    return [x for x in xs for _ in range(2)]

# 4ª solución
# ===========

def duplicaElementos4(xs: list[A]) -> list[A]:
    ys = []
    for x in xs:
        ys.append(x)
        ys.append(x)
    return ys

# Verificación
# ============

def test_duplicaElementos() -> None:
    for duplicaElementos in [duplicaElementos1, duplicaElementos2,
                             duplicaElementos3, duplicaElementos4]:
        assert duplicaElementos([3,2,5]) == [3,3,2,2,5,5]
    print("Verificado")

# La verificación es
#    >>> test_duplicaElementos()
#    Verificado

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

# La propiedad es
@given(st.lists(st.integers()))
def test_duplicaElementos_equiv(xs: list[int]) -> None:
    r = duplicaElementos1(xs)
    assert duplicaElementos2(xs) == r
    assert duplicaElementos3(xs) == r
    assert duplicaElementos4(xs) == r

# La comprobación es
#    >>> test_duplicaElementos_equiv()
#    >>>
```

17.16. Suma de fila del triángulo de los impares

```
# ----------------------------------------------------------------------
# Se condidera el siguiente triángulo de números impares
#
#              1
#           3     5
#        7     9    11
#    13    15    17    19
# 21    23    25    27    29
# ...
#
# Definir la función
#    sumaFilaTrianguloImpares : (int) -> int
# tal que sumaFilaTrianguloImpares(n) es la suma de la n-ésima fila
# del triángulo de los números impares. Por ejemplo,
#    sumaFilaTrianguloImpares(1)  ==  1
#    sumaFilaTrianguloImpares(2)  ==  8
#    len(str(sumaFilaTrianguloImpares(10**500)))    ==  1501
#    len(str(sumaFilaTrianguloImpares(10**5000)))   ==  15001
#    len(str(sumaFilaTrianguloImpares(10**50000)))  ==  150001
# ----------------------------------------------------------------------

from sys import set_int_max_str_digits
from timeit import Timer, default_timer

from hypothesis import given
from hypothesis import strategies as st

set_int_max_str_digits(10**6)

# 1ª solución
# ===========

def sumaFilaTrianguloImpares1(n: int) -> int:
    return sum(range(n**2-n+1, n**2+n, 2))

# 2ª solución
# ===========
```

```python
def sumaFilaTrianguloImpares2(n: int) -> int:
    return n**3

# Verificación
# ============

def test_sumaFilaTrianguloImpares() -> None:
    for sumaFilaTrianguloImpares in [sumaFilaTrianguloImpares1,
                                     sumaFilaTrianguloImpares2]:
        assert sumaFilaTrianguloImpares(1) == 1
        assert sumaFilaTrianguloImpares(2) == 8
    print("Verificado")

# La verificación es
#    >>> test_sumaFilaTrianguloImpares()
#    Verificado

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

# La propiedad es
@given(st.integers(min_value=1, max_value=1000))
def test_sumaFilaTrianguloImpares_equiv(n: int) -> None:
    assert sumaFilaTrianguloImpares1(n) == sumaFilaTrianguloImpares2(n)

# La comprobación es
#    >>> test_sumaFilaTrianguloImpares_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('len(str(sumaFilaTrianguloImpares1(6*10**7)))')
```

```
#    2.04 segundos
#    >>> tiempo('len(str(sumaFilaTrianguloImpares2(6*10**7)))')
#    0.00 segundos
```

17.17. Reiteración de suma de consecutivos

```
# ----------------------------------------------------------------------
# La reiteración de la suma de los elementos consecutivos de la lista
# [1,5,3] es 14 como se explica en el siguiente diagrama
#    1 + 5 = 6
#              \
#               ==> 14
#              /
#    5 + 3 = 8
# y la de la lista [1,5,3,4] es 29 como se explica en el siguiente
# diagrama
#    1 + 5 = 6
#              \
#               ==> 14
#              /       \
#    5 + 3 = 8          ==> 29
#              \       /
#               ==> 15
#              /
#    3 + 4 = 7
#
# Definir la función
#    sumaReiterada : (list[int]) -> int
# tal que sumaReiterada(xs) es la suma reiterada de los elementos
# consecutivos de la lista no vacía xs. Por ejemplo,
#    sumaReiterada([1,5,3])    ==  14
#    sumaReiterada([1,5,3,4])  ==  29
# ----------------------------------------------------------------------

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

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

```
A = TypeVar('A')
setrecursionlimit(10**6)

# 1ª solución
# ===========

# consecutivos(xs) es la lista de pares de elementos consecutivos de
# xs. Por ejemplo,
#    consecutivos([1,5,3,4])  ==  [(1,5),(5,3),(3,4)]
def consecutivos(xs: list[A]) -> list[tuple[A, A]]:
    return list(zip(xs, xs[1:]))

def sumaReiterada1(xs: list[int]) -> int:
    if len(xs) == 1:
        return xs[0]
    return sumaReiterada1([x + y for (x, y) in consecutivos(xs)])

# 2ª solución
# ===========

# sumaConsecutivos(xs) es la suma de los de pares de elementos
# consecutivos de xs. Por ejemplo,
#    sumaConsecutivos([1,5,3,4])   ==  [6,8,7]
def sumaConsecutivos(xs : list[int]) -> list[int]:
    return [x + y for (x, y) in list(zip(xs, xs[1:]))]

def sumaReiterada2(xs: list[int]) -> int:
    if len(xs) == 1:
        return xs[0]
    return sumaReiterada2(sumaConsecutivos(xs))

# 3ª solución
# ===========

def sumaReiterada3(xs: list[int]) -> int:
    if len(xs) == 1:
        return xs[0]
    return sumaReiterada3([x + y for (x, y) in list(zip(xs, xs[1:]))])
```

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

def test_sumaReiterada() -> None:
    for sumaReiterada in [sumaReiterada1, sumaReiterada2,
                          sumaReiterada3]:
        assert sumaReiterada([1,5,3]) == 14
        assert sumaReiterada([1,5,3,4]) == 29
    print("Verificado")

# La verificación es
#    >>> test_sumaReiterada()
#    Verificado

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

# La propiedad es
@given(st.lists(st.integers(), min_size=1))
def test_sumaReiterada_equiv(xs: list[int]) -> None:
    r = sumaReiterada1(xs)
    assert sumaReiterada2(xs) == r
    assert sumaReiterada3(xs) == r

# La comprobación es
#    >>> test_sumaReiterada_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('sumaReiterada1(range(4000))')
#    2.18 segundos
#    >>> tiempo('sumaReiterada2(range(4000))')
```

```
#    1.90 segundos
#    >>> tiempo('sumaReiterada3(range(4000))')
#    1.97 segundos
```

17.18. Producto de los elementos de la diagonal principal

```
# ---------------------------------------------------------------------
# Las matrices se pueden representar como lista de listas de la misma
# longitud, donde cada uno de sus elementos representa una fila de la
# matriz.
#
# Definir la función
#    productoDiagonalPrincipal :: Num a => [[a]] -> a
# tal que (productoDiagonalPrincipal xss) es el producto de los
# elementos de la diagonal principal de la matriz cuadrada xss. Por
# ejemplo,
#    productoDiagonal([[3,5,2],[4,7,1],[6,9,8]])  ==  168
#    productoDiagonal([[1, 2, 3, 4, 5]]*5)        ==  120
#    len(str(productoDiagonal([range(1, 30001)]*30000)))  ==  121288
# ---------------------------------------------------------------------

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

set_int_max_str_digits(10**6)
setrecursionlimit(10**6)

# 1ª solución
# ===========

# diagonal1(xss) es la diagonal de la matriz xss. Por ejemplo,
#    diagonal1([[3,5,2],[4,7,1],[6,9,0]])  ==  [3,7,0]
#    diagonal1([[3,5],[4,7],[6,9]])        ==  [3,7]
#    diagonal1([[3,5,2],[4,7,1]])          ==  [3,7]
def diagonal1(xss: list[list[int]]) -> list[int]:
    if not xss:
```

```
        return []
    if not xss[0]:
        return []
    return [xss[0][0]] + diagonal1(list(map((lambda ys : ys[1:]), xss[1:])))

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

def productoDiagonal1(xss: list[list[int]]) -> int:
    return producto(diagonal1(xss))

# 2ª solución
# ===========

def diagonal2(xss: list[list[int]]) -> list[int]:
    n = min(len(xss), len(xss[0]))
    return [xss[k][k] for k in range(n)]

def productoDiagonal2(xss: list[list[int]]) -> int:
    return producto(diagonal2(xss))

# 3ª solución
# ===========

def productoDiagonal3(xss: list[list[int]]) -> int:
    if not xss:
        return 1
    if not xss[0]:
        return 1
    return xss[0][0] * productoDiagonal3(list(map((lambda ys : ys[1:]), xss[1:])))

# Verificación
# ============

def test_productoDiagonal() -> None:
    for productoDiagonal in [productoDiagonal1, productoDiagonal2,
                             productoDiagonal3]:
        assert productoDiagonal([[3,5,2],[4,7,1],[6,9,8]]) == 168
        assert productoDiagonal([[1, 2, 3, 4, 5]]*5) == 120
    print("Verificado")
```

```
# La verificación es
#    >>> test_productoDiagonal()
#    Verificado

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

# ejemplo(n) es la matriz con n filas formadas por los números de 1 a
# n. Por ejemplo,
#    >>> ejemplo(3)
#    [[1, 2, 3], [1, 2, 3], [1, 2, 3]]
def ejemplo(n: int) -> list[list[int]]:
    return [list(range(1, n+1))]*n

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('productoDiagonal1(ejemplo(1200))')
#    1.97 segundos
#    >>> tiempo('productoDiagonal2(ejemplo(1200))')
#    0.00 segundos
#    >>> tiempo('productoDiagonal3(ejemplo(1200))')
#    1.56 segundos
```

17.19. Reconocimiento de potencias de 4

```
# ---------------------------------------------------------------------
# Definir la función
#    esPotenciaDe4 : (int) -> bool
# tal que esPotenciaDe4(n) se verifica si n es una potencia de 4. Por
# ejemplo,
#    esPotenciaDe4(16)               ==  True
#    esPotenciaDe4(17)               ==  False
#    esPotenciaDe4(4**(4*10**5))     ==  True
#    esPotenciaDe4(1 + 4**(4*10**5))  ==  False
# ---------------------------------------------------------------------
```

```python
from itertools import count, dropwhile, islice
from sys import setrecursionlimit
from timeit import Timer, default_timer
from typing import Callable, Iterator

setrecursionlimit(10**6)

# 1ª solución
# ===========

def esPotenciaDe4_1(n: int) -> bool:
    if n == 0:
        return False
    if n == 1:
        return True
    return n % 4 == 0 and esPotenciaDe4_1(n // 4)

# 2ª solución
# ===========

# potenciassDe4() es la lista de las potencias de 4. Por ejemplo,
#    >>> list(islice(potenciasDe4(), 5))
#    [1, 4, 16, 64, 256]
def potenciasDe4() -> Iterator[int]:
    return (4 ** n for n in count())

# pertenece(x, ys) se verifica si x pertenece a la lista ordenada
# (posiblemente infinita xs). Por ejemplo,
#    >>> pertenece(8, count(2, 2))
#    True
#    >>> pertenece(9, count(2, 2))
#    False
def pertenece(x: int, ys: Iterator[int]) -> bool:
    return next(dropwhile(lambda y: y < x, ys), None) == x

def esPotenciaDe4_2(n: int) -> bool:
    return pertenece(n, potenciasDe4())

# 3ª solución
```

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

# iterate(f, x) es el iterador obtenido aplicando f a x y continuando
# aplicando f al resultado anterior. Por ejemplo,
#    >>> list(islice(iterate(lambda x : 4 * x, 1), 5))
#    [1, 4, 16, 64, 256]
def iterate(f: Callable[[int], int], x: int) -> Iterator[int]:
    r = x
    while True:
        yield r
        r = f(r)

def potenciasDe4_2() -> Iterator[int]:
    return iterate(lambda x : 4 * x, 1)

def esPotenciaDe4_3(n: int) -> bool:
    return pertenece(n, potenciasDe4_2())

# 4ª solución
# ===========

def esPotenciaDe4_4(n: int) -> bool:
    return next(dropwhile(lambda y: y < n,
                          iterate(lambda x : 4 * x, 1)),
                          None) == n

# 5ª solución
# ===========

def esPotenciaDe4_5(n: int) -> bool:
    r = 1
    while r < n:
        r = 4 * r
    return r == n

# Verificación
# ============

def test_esPotenciaDe4() -> None:
    for esPotenciaDe4 in [esPotenciaDe4_1, esPotenciaDe4_2,
```

```
                            esPotenciaDe4_3, esPotenciaDe4_4,
                            esPotenciaDe4_5]:
            assert esPotenciaDe4(16)
            assert not esPotenciaDe4(17)
        assert list(islice(potenciasDe4(), 5)) == [1, 4, 16, 64, 256]
        print("Verificado")

# La verificación es
#    >>> test_esPotenciaDe4()
#    Verificado

# 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('esPotenciaDe4_1(4**(2*10**4))')
#    0.33 segundos
#    >>> tiempo('esPotenciaDe4_2(4**(2*10**4))')
#    0.63 segundos
#    >>> tiempo('esPotenciaDe4_3(4**(2*10**4))')
#    0.04 segundos
#    >>> tiempo('esPotenciaDe4_4(4**(2*10**4))')
#    0.05 segundos
#    >>> tiempo('esPotenciaDe4_5(4**(2*10**4))')
#    0.04 segundos
#
#    >>> tiempo('esPotenciaDe4_3(4**(3*10**5))')
#    2.29 segundos
#    >>> tiempo('esPotenciaDe4_4(4**(3*10**5))')
#    2.28 segundos
#    >>> tiempo('esPotenciaDe4_5(4**(3*10**5))')
#    2.31 segundos
```

17.20. Exponente en la factorización

```
# ---------------------------------------------------------------------
# Definir la función
#    exponente : (int, int) -> int
# tal que exponente(x, n) es el exponente de x en la factorización
# prima de n (se supone que x > 1 y n > 0). Por ejemplo,
#    exponente(2, 24)  ==  3
#    exponente(3, 24)  ==  1
#    exponente(6, 24)  ==  0
#    exponente(7, 24)  ==  0
# ---------------------------------------------------------------------

from itertools import takewhile
from typing import Callable, Iterator

from hypothesis import given
from hypothesis import strategies as st
from sympy.ntheory import factorint

# 1ª solución
# ===========

# esPrimo(x) se verifica si x es un número primo. Por ejemplo,
#    esPrimo(7)  ==  True
#    esPrimo(8)  ==  False
def esPrimo(x: int) -> bool:
    return [y for y in range(1, x+1) if x % y == 0] == [1,x]

def exponente1(x: int, n: int) -> int:
    def aux (m: int) -> int:
        if m % x == 0:
            return 1 + aux(m // x)
        return 0
    if esPrimo(x):
        return aux(n)
    return 0

# 2ª solución
# ===========
```

```
# iterate(f, x) es el iterador obtenido aplicando f a x y continuando
# aplicando f al resultado anterior. Por ejemplo,
#    >>> list(islice(iterate(lambda x : 4 * x, 1), 5))
#    [1, 4, 16, 64, 256]
def iterate(f: Callable[[int], int], x: int) -> Iterator[int]:
    r = x
    while True:
        yield r
        r = f(r)

# divisible(n, x) se verifica si n es divisible por x. Por ejemplo,
#    divisible(6, 2)  ==  True
#    divisible(7, 2)  ==  False
def divisible(n: int, x: int) -> bool:
    return n % x == 0

def exponente2(x: int, n: int) -> int:
    if esPrimo(x):
        return len(list(takewhile(lambda m : divisible(m, x),
                                  iterate(lambda m : m // x, n))))
    return 0

# 3ª solución
# ===========

def exponente3(x: int, n: int) -> int:
    return factorint(n, multiple = True).count(x)

# Verificación
# ============

def test_exponente() -> None:
    for exponente in [exponente1, exponente2, exponente3]:
        assert exponente(2, 24) == 3
        assert exponente(3, 24) == 1
        assert exponente(6, 24) == 0
        assert exponente(7, 24) == 0
    print("Verificado")
```

```
# La verificación es
#    >>> test_exponente()
#    Verificado

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

# La propiedad es
@given(st.integers(min_value=1, max_value=1000),
       st.integers(min_value=0, max_value=1000))
def test_exponente_equiv(x: int, n: int) -> None:
    r = exponente1(x, n)
    assert r == exponente2(x, n)
    assert r == exponente3(x, n)

# La comprobación es
#    >>> test_exponente_equiv()
#    >>>
```

17.21. Mayor órbita de la sucesión de Collatz

```
# ----------------------------------------------------------------------
# Se considera la siguiente operación, aplicable a cualquier número
# entero positivo:
#    * Si el número es par, se divide entre 2.
#    * Si el número es impar, se multiplica por 3 y se suma 1.
#
# Dado un número cualquiera, podemos calcular su órbita; es decir,
# las imágenes sucesivas al iterar la función. Por ejemplo, la órbita
# de 13 es
#    13, 40, 20, 10, 5, 16, 8, 4, 2, 1, 4, 2, 1,...
#
# Si observamos este ejemplo, la órbita de 13 es periódica, es decir,
# se repite indefinidamente a partir de un momento dado). La conjetura
# de Collatz dice que siempre alcanzaremos el 1 para cualquier número
# con el que comencemos. Ejemplos:
#    * Empezando en n = 6 se obtiene 6, 3, 10, 5, 16, 8, 4, 2, 1.
#    * Empezando en n = 11 se obtiene: 11, 34, 17, 52, 26, 13, 40, 20,
#      10, 5, 16, 8, 4, 2, 1.
#    * Empezando en n = 27, la sucesión tiene 112 pasos, llegando hasta
```

```
#      9232 antes de descender a 1:  27, 82, 41, 124, 62, 31, 94, 47,
#      142, 71, 214, 107, 322, 161, 484, 242, 121, 364, 182, 91, 274,
#      137, 412, 206, 103, 310, 155, 466, 233, 700, 350, 175, 526, 263,
#      790, 395, 1186, 593, 1780, 890, 445, 1336, 668, 334, 167, 502,
#      251, 754, 377, 1132, 566, 283, 850, 425, 1276, 638, 319, 958,
#      479, 1438, 719, 2158, 1079, 3238, 1619, 4858, 2429, 7288, 3644,
#      1822, 911, 2734, 1367, 4102, 2051, 6154, 3077, 9232, 4616, 2308,
#      1154, 577, 1732, 866, 433, 1300, 650, 325, 976, 488, 244, 122,
#      61, 184, 92, 46, 23, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5,
#      16, 8, 4, 2, 1.
#
# Definir la función
#    mayoresGeneradores :: Integer -> [Integer]
# tal que (mayoresGeneradores n) es la lista de los números menores o
# iguales que n cuyas órbitas de Collatz son las de mayor longitud. Por
# ejemplo,
#    mayoresGeneradores(20)    ==  [18,19]
#    mayoresGeneradores(10^6)  ==  [837799]
# ----------------------------------------------------------------------

from functools import lru_cache
from itertools import count, islice
from timeit import Timer, default_timer
from typing import Iterator

from hypothesis import given
from hypothesis import strategies as st

# 1ª solución
# ===========

# siguiente(n) es el siguiente de n en la sucesión de Collatz. Por
# ejemplo,
#    siguiente(13)  ==  40
#    siguiente(40)  ==  20
def siguiente (n: int) -> int:
    if n % 2 == 0:
        return n // 2
    return 3*n+1
```

```python
# collatz1(n) es la órbita de Collatz de n hasta alcanzar el
# 1. Por ejemplo,
#    collatz1(13)  ==  [13,40,20,10,5,16,8,4,2,1]
def collatz1(n: int) -> list[int]:
    if n == 1:
        return [1]
    return [n]+ collatz1(siguiente(n))

# longitudesOrbita() es la lista de los números junto a las longitudes de
# las órbitas de Collatz que generan. Por ejemplo,
#    >>> list(islice(longitudesOrbitas(), 10))
#    [(1,1),(2,2),(3,8),(4,3),(5,6),(6,9),(7,17),(8,4),(9,20),(10,7)]
def longitudesOrbitas() -> Iterator[tuple[int, int]]:
    return ((n, len(collatz1(n))) for n in count(1))

def mayoresGeneradores1(n: int) -> list[int]:
    ps = list(islice(longitudesOrbitas(), n))
    m = max((y for (_, y) in ps))
    return [x for (x,y) in ps if y == m]

# 2ª solución
# ===========

def collatz2(n: int) -> list[int]:
    r = [n]
    while n != 1:
        n = siguiente(n)
        r.append(n)
    return r

def longitudesOrbitas2() -> Iterator[tuple[int, int]]:
    return ((n, len(collatz2(n))) for n in count(1))

def mayoresGeneradores2(n: int) -> list[int]:
    ps = list(islice(longitudesOrbitas2(), n))
    m = max((y for (_, y) in ps))
    return [x for (x,y) in ps if y == m]

# 3ª solución
# ===========
```

```python
# longitudOrbita(x) es la longitud de la órbita de x. Por ejemplo,
#    longitudOrbita(13)  ==  10
def longitudOrbita(x: int) -> int:
    if x == 1:
        return 1
    return 1 + longitudOrbita(siguiente(x))

def mayoresGeneradores3(n: int) -> list[int]:
    ps = [(x, longitudOrbita(x)) for x in range(1, n+1)]
    m = max((y for (_, y) in ps))
    return [x for (x,y) in ps if y == m]

# 4ª solución
# ===========

# longitudOrbita2(x) es la longitud de la órbita de x. Por ejemplo,
#    longitudOrbita2(13)  ==  10
def longitudOrbita2(x: int) -> int:
    r = 0
    while x != 1:
        x = siguiente(x)
        r += 1
    return r + 1

def mayoresGeneradores4(n: int) -> list[int]:
    ps = [(x, longitudOrbita2(x)) for x in range(1, n+1)]
    m = max((y for (_, y) in ps))
    return [x for (x,y) in ps if y == m]

# 5ª solución
# ===========

@lru_cache(maxsize=None)
def longitudOrbita3(x: int) -> int:
    if x == 1:
        return 1
    return 1 + longitudOrbita3(siguiente(x))

def mayoresGeneradores5(n: int) -> list[int]:
```

```
    ps = [(x, longitudOrbita3(x)) for x in range(1, n+1)]
    m = max((y for (_, y) in ps))
    return [x for (x,y) in ps if y == m]

# Verificación
# ============

def test_mayoresGeneradores() -> None:
    for mayoresGeneradores in [mayoresGeneradores1,
                               mayoresGeneradores2,
                               mayoresGeneradores3,
                               mayoresGeneradores4,
                               mayoresGeneradores5]:
        assert mayoresGeneradores(20) == [18,19]
    print("Verificado")

# La verificación es
#    >>> test_mayoresGeneradores()
#    Verificado

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

# La propiedad es
@given(st.integers(min_value=1, max_value=1000))
def test_mayoresGeneradores_equiv(n: int) -> None:
    r = mayoresGeneradores1(n)
    assert mayoresGeneradores2(n) == r
    assert mayoresGeneradores3(n) == r

# La comprobación es
#    >>> test_mayoresGeneradores_equiv()
#    >>>

# Comprobación 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 comprobación es
#    >>> tiempo('mayoresGeneradores1(10**5)')
#    4.08 segundos
#    >>> tiempo('mayoresGeneradores2(10**5)')
#    1.95 segundos
#    >>> tiempo('mayoresGeneradores3(10**5)')
#    2.16 segundos
#    >>> tiempo('mayoresGeneradores4(10**5)')
#    1.71 segundos
#    >>> tiempo('mayoresGeneradores5(10**5)')
#    0.14 segundos
```

17.22. Máximos locales

```
# ---------------------------------------------------------------------
# Un máximo local de una lista es un elemento de la lista que es mayor
# que su predecesor y que su sucesor en la lista. Por ejemplo, 5 es un
# máximo local de [3,2,5,3,7,7,1,6,2] ya que es mayor que 2 (su
# predecesor) y que 3 (su sucesor).
#
# Definir la función
#    maximosLocales : (xs: list[int]) -> list[int]
# tal que maximosLocales(xs) es la lista de los máximos locales de la
# lista xs. Por ejemplo,
#    maximosLocales([3,2,5,3,7,7,1,6,2]) == [5,6]
#    maximosLocales(list(range(0, 100))) == []
# ---------------------------------------------------------------------

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 maximosLocales1(xs: list[int]) -> list[int]:
    if len(xs) < 3:
        return []
    x, y, z, *ys = xs
    if y > x and y > z:
        return [y] + maximosLocales1([z] + ys)
    return maximosLocales1([y, z] + ys)

# 2ª solución
# ===========

def maximosLocales2(xs: list[int]) -> list[int]:
    ys: list[int] = []
    if len(xs) < 3:
        return ys
    for i in range(1, len(xs) - 1):
        if xs[i] > xs[i - 1] and xs[i] > xs[i + 1]:
            ys.append(xs[i])
    return ys

# 3ª solución
# ===========

def maximosLocales3(xs: list[int]) -> list[int]:
    return [y for x, y, z in zip(xs, xs[1:], xs[2:]) if y > x and y > z]

# Verificación
# ============

def test_maximosLocales() -> None:
    for maximosLocales in [maximosLocales1,
                           maximosLocales2,
                           maximosLocales3]:
        assert maximosLocales([3,2,5,3,7,7,1,6,2]) == [5,6]
        assert maximosLocales(list(range(0, 100))) == []
    print("Verificado")

# La verificación es
#    >>> test_maximosLocales()
```

```
#    Verificado

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

# La propiedad es
@given(st.lists(st.integers()))
def test_maximosLocales_equiv(xs: list[int]) -> None:
    r = maximosLocales1(xs)
    assert maximosLocales2(xs) == r
    assert maximosLocales3(xs) == r

# La comprobación es
#    >>> test_maximosLocales_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('maximosLocales1([1,2,3]*(10**4))')
#    3.19 segundos
#    >>> tiempo('maximosLocales2([1,2,3]*(10**4))')
#    0.01 segundos
#    >>> tiempo('maximosLocales3([1,2,3]*(10**4))')
#    0.01 segundos
#
#    >>> tiempo('maximosLocales2([1,2,3]*(10**7))')
#    3.95 segundos
#    >>> tiempo('maximosLocales3([1,2,3]*(10**7))')
#    1.85 segundos
```

Bibliografía

[1] A. Casamayou-Boucau, P. Chauvin, and G. Connan. *Programmation en Python pour les mathématiques*. Dunod, 2012.

[2] A. Downey, J. Elkner, and C. Meyers. *Aprenda a pensar como un programador con Python*. Green Tea Press, 2002.

[3] M. Goodrich, R. Tamassia, and M. Goldwasser. *Data structures and algorithms in Python*. Wiley, 2013.

[4] J. Guttag. *Introduction to computation and programming using python, second edition*. MIT Press, 2016.

[5] T. Hall and J. Stacey. *Python 3 for absolute beginners*. Apress, 2010.

[6] M. Hetland. *Python algorithms: Mastering basic algorithms in the Python language*. Apress, 2011.

[7] J. Hunt. *A beginners guide to Python 3 programming*. Springer International Publishing, 2019.

[8] J. Hunt. *Advanced guide to Python 3 programming*. Springer International Publishing, 2019.

[9] S. Lott. *Functional Python programming, 2nd Edition*. Packt Publishing, 2018.

[10] T. Padmanabhan. *Programming with Python*. Springer Singapore, 2017.

[11] M. Rubio-Sanchez. *Introduction to recursive programming*. CRC Press, 2017.

[12] A. Saha. *Doing Math with Python: Use Programming to explore algebra, statistics, calculus, and more!* No Starch Press, 2015.

[13] B. Stephenson. *The Python workbook: A brief introduction with exercises and solutions*. Springer International Publishing, 2015.

[14] R. van Hattem. *Mastering Python: Write powerful and efficient code using the full range of Python's capabilities*. Packt Publishing, 2022.