

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, 28 de enero de 2023

Esta obra está bajo una licencia Reconocimiento-NoComercial-CompartirIgual 2.5 Spain de Creative Commons.

Se permite:

- copiar, distribuir y comunicar públicamente la obra
- hacer obras derivadas

Bajo las condiciones siguientes:

Reconocimiento. Debe reconocer los créditos de la obra de la manera especificada por el autor.



No comercial. No puede utilizar esta obra para fines comerciales.



Compartir bajo la misma licencia. Si altera o transforma esta obra, o genera una obra derivada, sólo puede distribuir la obra generada bajo una licencia idéntica a ésta.

- Al reutilizar o distribuir la obra, tiene que dejar bien claro los términos de la licencia de esta obra.
- alguna de estas condiciones puede no aplicarse si se obtiene el permiso del titular de los derechos de autor.

Esto es un resumen del texto legal (la licencia completa). Para ver una copia de esta licencia, visite <http://creativecommons.org/licenses/by-nc-sa/2.5/es/> o envíe una carta a Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

Índice general

I	Introducción a la programación con Python	7
1	Definiciones elementales de funciones	9
1.1	Definiciones por composición sobre números, listas y booleanos .	9
1.2	Definiciones con condicionales, guardas o patrones	18
2	Definiciones por comprensión	31
2.1	Definiciones por comprensión	31
3	Definiciones por recursión	57
3.1	Definiciones por recursión	57
3.2	Operaciones conjuntistas con listas	66
3.3	El algoritmo de Luhn	78
3.4	Números de Lychrel	81
3.5	Funciones sobre cadenas	86
4	Funciones de orden superior	95
4.1	Funciones de orden superior y definiciones por plegado	95
5	Tipos definidos y de datos algebraicos	109
5.1	Tipos de datos algebraicos: Árboles binarios	109
5.2	Tipos de datos algebraicos: Árboles	118
5.3	Tipos de datos algebraicos: Expresiones	133
II	Algorítmica	151
6	El tipo abstracto de datos de las pilas	153
6.1	El tipo abstracto de datos (TAD) de las pilas	153
6.2	Implementación del TAD de las pilas mediante listas	154
6.3	Implementación del TAD de las pilas mediante deque	159
6.4	Ejercicios con el TAD de las pilas	163

7 El tipo abstracto de datos de las colas	187
7.1 El tipo abstracto de datos (TAD) de las colas	187
7.2 Implementación del TAD de las colas mediante listas	188
7.3 Implementación del TAD de las colas mediante dos listas	193
7.4 Implementación del TAD de las colas mediante deque	198
Bibliografía	203

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

1ª solución

```
def rotala(xs: list[A]) -> list[A]:
    if xs == []:
        return []
    return xs[1:] + [xs[0]]
```

2ª solución

```
def rotalb(xs: list[A]) -> list[A]:
    if xs == []:
        return []
    ys = xs[1:]
    ys.append(xs[0])
    return ys
```

3ª solución

```
def rotalc(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_rotal(xs: list[int]) -> None:
    assert rotala(xs) == rotalb(xs) == rotalc(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]
```

```

# 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 divisionSegura(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 es idempotente; 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)

```

```

# 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/ilm/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
```

```

# 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 sumal(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 sumal(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 cuadrados 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) == 2500000000166666666641666666665000000000
# -----

# 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
# =====
```

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

# 2ª definición
# =====

def linea2(n: int) -> list[int]:
    s = sumal(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
```

```

# -----
# 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 número 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

```

```

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)  == 23333333333333333331666666666666666668
#
# 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)))

```

```
# 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  $(1+1/n)^n$ . Por ejemplo,
# aproxE(4) == [2.0, 2.25, 2.37037037037037, 2.44140625]
# aproxE(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]

```

```

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  $\sin(x)/x$ , cuando  $x$  tiende a cero, se puede
# calcular como el límite de la sucesión  $\sin(1/n)/(1/n)$ , 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  $\sin(1/m)/(1/m)$ . 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
#  $\sin(1/m)/(1/m)$  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  $\pi$  puede calcularse con la fórmula de
# Leibniz
#  $\pi/4 = 1 - 1/3 + 1/5 - 1/7 + \dots + (-1)^n/(2n+1) + \dots$ 
#
# Definir la función
# calculaPi : (int) -> float
# tal que calculaPi(n) es la aproximación del número  $\pi$  calculada
# mediante la expresión
#  $4 \cdot (1 - 1/3 + 1/5 - 1/7 + \dots + (-1)^n/(2n+1))$ 
# 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  $x < y < z$ .

```

```

#
# 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  $1 \leq n < m$ . (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 aux(x)

```

```

    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:])]

# -----

```

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

```

```

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/ilm/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')

# -----
# Ejercicio 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
#

```

```
# 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
```

```

@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.
# -----

```

```

# 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))

```

```

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

```

```

# 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 ==)
# -----

```

```

# -----
# 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
#  $A \cup (B \cap C) = (A \cup B) \cap C$ 
# 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  $\cup$  (ys  $\cap$  zs) = [0]  $\cup$  ([]  $\cap$  []) = [0]  $\cup$  [] = [0]
# (xs  $\cup$  ys)  $\cap$  zs = ([0]  $\cup$  [])  $\cap$  [] = [0]  $\cap$  [] = []

# -----
# 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]

```

```

# 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.
# -----

```

```
# 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

```
# -----
# § 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
# -----

```

```

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(siguiete(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(siguiete(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
#     menoresDeOrdenMayor : (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 --
# -----
```

```

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/ilm/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

# -----

```

```

# 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 agrupar1(xss: list[list[A]]) -> list[list[A]]:
    if not xss:
        return []
    if [] in xss:
        return []
    return [primeros(xss)] + agrupar1(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.
# -----

```



```

# 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/ilm/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

# -----

```

```

# 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

```

```

@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

# -----

```

```

# 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)))
# -----

```

```

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/ilm/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 árbol 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()),
#                     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

```



```

# 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
#        /    \
#       /      \
#      /        \
#     Disy       Conj
#    /  \       /  \
#   Conj Neg  Conj True
#  /  \  |    |
# True False False False
#
#           Conj
#        /    \
#       /      \
#      /        \
#     Disy       Conj
#    /  \       /  \
#   Conj Neg  Conj True
#  /  \  |    |
# 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
#
#
#
#
#
#
# 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/ilm/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

# -----

```

```

# 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
```



```

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'])
# Las 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 e. 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

```

```

@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',
    'vacía',
    'apila',
    'esVacia',
    'cima',
    'desapila',
    'pilaAleatoria'
]
from src.TAD.pilaConListas import (Pila, apila, cima, desapila, esVacia,
                                    pilaAleatoria, vacía)

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

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

```

```

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(esVacía(vacía()))
#     True
#
# Finalmente, se define un generador aleatorio de pilas y se comprueba
# que las pilas cumplen las propiedades de su especificación.

__all__ = [
    'Pila',
    'vacía',
    'apila',
    'esVacía',
    '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 pila 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 las
# 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)

```

```

# 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)
```

```
# 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(ordenInserPila(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))
```

```

# 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_listaPila PASSED
# test_pilaAlista PASSED
# test_1_listaPila PASSED
# test_2_listaPila 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 (resto (vacía, c))
# + 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',
    'vacía',
    'inserta',
    'primero',
    'resto',
    'esVacia',
    'colaAleatoria'
]
from src.TAD.colaConListas import (Cola, colaAleatoria, esVacia, inserta,
                                   primero, resto, vacía)

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

```

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.primer()
#     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.primer()

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_colal(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 testCola2(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.primer()
# 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.primer()

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_colal(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_colal2(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.primer()
#     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.primer()

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_colal(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 testCola2(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
```


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.