

Piensa en Haskell y en Python

(Ejercicios de programación con Haskell y con Python)

José A. Alonso Jiménez

Grupo de Lógica Computacional
Dpto. de Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, 12 de marzo 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	11
1.	Definiciones elementales de funciones	13
1.1.	Media aritmética de tres números	14
1.2.	Suma de monedas	15
1.3.	Volumen de la esfera	16
1.4.	Área de la corona circular	17
1.5.	Último dígito	18
1.6.	Máximo de tres números	19
1.7.	El primero al final	19
1.8.	Los primeros al final	22
1.9.	Rango de una lista	23
1.10.	Reconocimiento de palindromos	24
1.11.	Interior de una lista	25
1.12.	Elementos finales	26
1.13.	Segmento de una lista	28
1.14.	Primeros y últimos elementos	30
1.15.	Elemento mediano	31
1.16.	Tres iguales	32
1.17.	Tres diferentes	33
1.18.	División segura	34
1.19.	Disyunción excluyente	36
1.20.	Mayor rectángulo	39
1.21.	Intercambio de componentes de un par	40
1.22.	Distancia entre dos puntos	42
1.23.	Permutación cíclica	45
1.24.	Mayor número con dos dígitos dados	47
1.25.	Número de raíces de la ecuación de segundo grado	48

1.26. Raíces de la ecuación de segundo grado	49
1.27. Fórmula de Herón para el área de un triángulo	52
1.28. Intersección de intervalos cerrados	53
1.29. Números racionales	56
2. Definiciones por comprensión	61
2.1. Reconocimiento de subconjunto	62
2.2. Igualdad de conjuntos	66
2.3. Unión conjuntista de listas	70
2.4. Intersección conjuntista de listas	75
2.5. Diferencia conjuntista de listas	79
2.6. Divisores de un número	84
2.7. Divisores primos	94
2.8. Números libres de cuadrados	102
2.9. Suma de los primeros números naturales	109
2.10. Suma de los cuadrados de los primeros números naturales	114
2.11. Suma de cuadrados menos cuadrado de la suma	119
2.12. Triángulo aritmético	127
2.13. Suma de divisores	134
2.14. Números perfectos	143
2.15. Números abundantes	148
2.16. Números abundantes menores o iguales que n	152
2.17. Todos los abundantes hasta n son pares	158
2.18. Números abundantes impares	164
2.19. Suma de múltiplos de 3 ó 5	169
2.20. Puntos dentro del círculo	176
2.21. Aproximación del número e	180
2.22. Aproximación al límite de $\sin(x)/x$ cuando x tiende a cero	190
2.23. Cálculo del número π mediante la fórmula de Leibniz	199
2.24. Ternas pitagóricas	205
2.25. Ternas pitagóricas con suma dada	209
2.26. Producto escalar	214
2.27. Representación densa de polinomios	219
2.28. Base de datos de actividades.	224
3. Definiciones por recursión	229

3.1. Potencia entera230
3.2. Algoritmo de Euclides del mcd236
3.3. Dígitos de un número238
3.4. Suma de los dígitos de un número246
3.5. Número a partir de sus dígitos251
3.6. Exponente de la mayor potencia de x que divide a y258
3.7. Producto cartesiano de dos conjuntos263
3.8. Subconjuntos de un conjunto267
3.9. El algoritmo de Luhn272
3.10. Números de Lychrel277
3.11. Suma de los dígitos de una cadena288
3.12. Primera en mayúscula y restantes en minúscula292
3.13. Mayúsculas iniciales296
3.14. Posiciones de un carácter en una cadena301
3.15. Reconocimiento de subcadenas305
4. Funciones de orden superior	311
4.1. Segmentos cuyos elementos cumplen una propiedad311
4.2. Elementos consecutivos relacionados315
4.3. Agrupación de elementos por posición316
4.4. Concatenación de una lista de listas323
4.5. Aplica según propiedad327
4.6. Máximo de una lista333
5. Tipos definidos y tipos de datos algebraicos	337
5.1. Movimientos en el plano340
5.2. El tipo de figuras geométricas345
5.3. El tipo de los números naturales347
5.4. El tipo de las listas350
5.5. El tipo de los árboles binarios352
5.6. El tipo de las fórmulas: Variables de una fórmula355
5.7. El tipo de las fórmulas: Valor de una fórmula358
5.8. El tipo de las fórmulas: Interpretaciones de una fórmula363
5.9. El tipo de las fórmulas: Reconocedor de tautologías368
5.10. Altura de un árbol binario374
5.11. Aplicación de una función a un árbol377

5.12. Árboles con la misma forma379
5.13. Árbol con las hojas en la profundidad dada385
5.14. El tipo de las expresiones aritméticas: Valor de una expresión388
5.15. El tipo de las expresiones aritméticas: Valor de la resta391
5.16. Número de hojas de un árbol binario397
5.17. Profundidad de un árbol binario402
5.18. Recorrido de árboles binarios407
5.19. Imagen especular de un árbol binario413
5.20. Subárbol de profundidad dada419
5.21. Árbol de profundidad n con nodos iguales425
5.22. Suma de un árbol431
5.23. Rama izquierda de un árbol binario433
5.24. Árboles balanceados436
5.25. Árboles con bordes iguales439
5.26. Árboles con igual estructura442
5.27. Existencia de elementos del árbol que verifican una propiedad445
5.28. Elementos del nivel k de un árbol448
5.29. Árbol de factorización451
5.30. Valor de un árbol booleano457
5.31. Valor de una expresión aritmética básica461
5.32. Aplicación de una función a una expresión aritmética463
5.33. Valor de una expresión aritmética con una variable466
5.34. Número de variables de una expresión aritmética469
5.35. Valor de una expresión aritmética con variables472
5.36. Número de sumas en una expresión aritmética475
5.37. Sustitución en una expresión aritmética478
5.38. Expresiones aritméticas reducibles481
5.39. Máximos valores de una expresión aritmética484
5.40. Valor de expresiones aritméticas generales488
5.41. Valor de una expresión vectorial491

II Algorítmica 497

6. El tipo abstracto de datos de las pilas 499

6.1. El tipo abstracto de datos de las pilas499
--	------

6.2. El tipo de datos de las pilas mediante listas501
6.3. El tipo de datos de las pilas con librerías509
6.4. Transformación entre pilas y listas516
6.5. Filtrado de pilas según una propiedad523
6.6. Aplicación de una función a los elementos de una pila527
6.7. Pertenencia a una pila531
6.8. Inclusión de pilas534
6.9. Reconocimiento de prefijos de pilas539
6.10. Reconocimiento de subpilas543
6.11. Reconocimiento de ordenación de pilas547
6.12. Ordenación de pilas por inserción552
6.13. Eliminación de repeticiones en una pila557
6.14. Máximo elemento de una pila560
7. El tipo abstracto de datos de las colas	565
7.1. El tipo abstracto de datos de las colas565
7.2. El tipo de datos de las colas mediante listas568
7.3. El tipo de datos de las colas mediante dos listas575
7.4. El tipo de datos de las colas mediante sucesiones585
7.5. Transformaciones entre colas y listas593
7.6. Último elemento de una cola599
7.7. Longitud de una cola603
7.8. Todos los elementos de la cola verifican una propiedad607
7.9. Algún elemento de la verifica una propiedad611
7.10. Extensión de colas615
7.11. Intercalado de dos colas619
7.12. Agrupación de colas625
7.13. Pertenencia a una cola628
7.14. Inclusión de colas631
7.15. Reconocimiento de prefijos de colas636
7.16. Reconocimiento de subcolas640
7.17. Reconocimiento de ordenación de colas644
7.18. Máximo elemento de una cola648
8. El tipo abstracto de datos de los conjuntos	653
8.1. El tipo abstracto de datos de los conjuntos653

8.2. El tipo de datos de los conjuntos mediante listas no ordenadas con duplicados657
8.3. El tipo de datos de los conjuntos mediante listas no ordenadas sin duplicados666
8.4. El tipo de datos de los conjuntos mediante listas ordenadas sin duplicados676
8.5. El tipo de datos de los conjuntos mediante librería686
8.6. Transformaciones entre conjuntos y listas695
8.7. Reconocimiento de subconjunto700
8.8. Reconocimiento de subconjunto propio705
8.9. Conjunto unitario707
8.10. Número de elementos de un conjunto709
8.11. Unión de dos conjuntos712
8.12. Unión de varios conjuntos716
8.13. Intersección de dos conjuntos719
A. Método de Pólya para la resolución de problemas	727
A.1. Método de Pólya para la resolución de problemas matemáticos	.727
A.2. Método de Pólya para resolver problemas de programación . .	.728
Bibliografía	731

Introducción

Este libro es una introducción a la programación con Haskell y Python, a través de la resolución de ejercicios publicados diariamente en el blog [Exercitium](https://www.glc.us.es/jalonso/exercitium) ¹. Estos ejercicios están organizados siguiendo el orden de los [Temas de programación funcional con Haskell](https://jaalonso.github.io/materias/PFconHaskell/temas.html) ². Las soluciones a los ejercicios están disponibles en dos repositorios de GitHub, uno con las [soluciones en Haskell](https://github.com/jaalonso/Exercitium) ³ y otro con las [soluciones en Python](https://github.com/jaalonso/Exercitium-Python) ⁴). Ambos repositorios están estructurados como proyectos utilizando [Stack](https://docs.haskellstack.org/en/stable) ⁵ y [Poetry](https://python-poetry.org) ⁶, respectivamente. Se han escrito soluciones en Python con un estilo funcional similar al de Haskell, y se han comprobado con [mypy](http://mypy-lang.org) ⁷ que los tipos de las definiciones en Python son correctos. Además, se han dado varias soluciones a cada ejercicio, verificando su equivalencia (mediante [QuickCheck](https://hackage.haskell.org/package/QuickCheck) ⁸ en Haskell e [Hypothesis](https://hypothesis.readthedocs.io/en/latest) ⁹ en Python) y comparando su eficiencia.

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

¹<https://www.glc.us.es/jalonso/exercitium>

²<https://jaalonso.github.io/materias/PFconHaskell/temas.html>

³<https://github.com/jaalonso/Exercitium>

⁴<https://github.com/jaalonso/Exercitium-Python>

⁵<https://docs.haskellstack.org/en/stable>

⁶<https://python-poetry.org>

⁷<http://mypy-lang.org>

⁸<https://hackage.haskell.org/package/QuickCheck>

⁹<https://hypothesis.readthedocs.io/en/latest>

¹⁰<https://jaalonso.github.io/materias/PFconHaskell>

Parte I

Introducción a la programación con Python

Capítulo 1

Definiciones elementales de funciones

En este capítulo se plantean ejercicios con definiciones elementales (no recursivas) de funciones. Se corresponden con los 4 primeros temas del [Curso de programación funcional con Haskell](https://jaalonso.github.io/materias/PFconHaskell/temas.html) ¹.

Contenido

1.1. Media aritmética de tres números	14
1.2. Suma de monedas	15
1.3. Volumen de la esfera	16
1.4. Área de la corona circular	17
1.5. Último dígito	18
1.6. Máximo de tres números	19
1.7. El primero al final	19
1.8. Los primeros al final	22
1.9. Rango de una lista	23
1.10. Reconocimiento de palindromos	24
1.11. Interior de una lista	25
1.12. Elementos finales	26
1.13. Segmento de una lista	28
1.14. Primeros y últimos elementos	30

¹<https://jaalonso.github.io/materias/PFconHaskell/temas.html>

1.15. Elemento mediano	31
1.16. Tres iguales	32
1.17. Tres diferentes	33
1.18. División segura	34
1.19. Disyunción excluyente	36
1.20. Mayor rectángulo	39
1.21. Intercambio de componentes de un par	40
1.22. Distancia entre dos puntos	42
1.23. Permutación cíclica	45
1.24. Mayor número con dos dígitos dados	47
1.25. Número de raíces de la ecuación de segundo grado	48
1.26. Raíces de la ecuación de segundo grado	49
1.27. Fórmula de Herón para el área de un triángulo	52
1.28. Intersección de intervalos cerrados	53
1.29. Números racionales	56

1.1. Media aritmética de tres números

En Haskell

```

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

```

```

module Media_aritmetica_de_tres_numeros where

```

```

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

```

En Python

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

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

1.2. Suma de monedas

En Haskell

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

module Suma_de_monedas where

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

En Python

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

```

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

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

```

1.3. Volumen de la esfera

En Haskell

```

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

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

module Volumen_de_la_esfera where

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

```

En Python

```

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

```



```
from math import pi

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

1.4. Área de la corona circular

En Haskell

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

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

```
module Area_corona_circular where
```

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

En Python

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

```
from math import pi

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

1.5. Último dígito

En Haskell

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

```
module Ultimo_digito where
```

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

En Python

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

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

1.6. Máximo de tres números

En Haskell

```

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

```

```

module Maximo_de_tres_numeros where

```

```

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

```

En Python

```

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

```

```

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

```

1.7. El primero al final

En Haskell

```

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

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

```
module El_primer_al_final where
```

```
import Test.QuickCheck
```

```
-- 1ª solución
```

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

```
rotala :: [a] -> [a]
```

```
rotala [] = []
```

```
rotala xs = tail xs ++ [head xs]
```

```
-- 2ª solución
```

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

```
rotalb :: [a] -> [a]
```

```
rotalb [] = []
```

```
rotalb (x:xs) = xs ++ [x]
```

```
-- Comprobación de equivalencia
```

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

```
-- La propiedad es
```

```
prop_rotal :: [Int] -> Bool
```

```
prop_rotal xs =
```

```
    rotala xs == rotalb xs
```

```
-- La comprobación es
```

```
--     λ> quickCheck prop_rotal
```

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

En Python

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

```
from typing import TypeVar
```

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

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

```
# 1ª solución
```

```
def rota1(xs: list[A]) -> list[A]:
    if xs == []:
        return []
    return xs[1:] + [xs[0]]
```

```
# 2ª solución
```

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

```
# 3ª solución
```

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

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

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

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

1.8. Los primeros al final

En Haskell

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

```
module Los_primeros_al_final where
```

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

En Python

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

```
from typing import TypeVar
```

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

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

1.9. Rango de una lista

En Haskell

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

```
module Rango_de_una_lista where
```

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

En Python

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

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

1.10. Reconocimiento de palindromos

En Haskell

```

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

```

```
module Reconocimiento_de_palindromos where
```

```

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

```

En Python

```

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

```

```
from typing import TypeVar
```

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

```

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

```


1.11. Interior de una lista

En Haskell

```

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

```

```

module Interior_de_una_lista where

```

```

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

```

En Python

```

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

```

```

from typing import TypeVar

from hypothesis import given
from hypothesis import strategies as st

```

```

A = TypeVar('A')

```

```

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

```

```

# 2ª solución

```

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

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

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

1.12. Elementos finales

En Haskell

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

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

```
module Elementos_finales where
import Test.QuickCheck
```

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

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

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

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

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

En Python

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

```
from typing import TypeVar
```

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

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

```
# 1ª definición
```

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

```
# 2ª definición
```

```
def finales2(n: int, xs: list[A]) -> list[A]:
    if n == 0:
        return []
    return xs[-n:]
```

```
# 3ª definición
```

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

```

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

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

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

```

1.13. Segmento de una lista

En Haskell

```

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

module Segmento_de_una_lista where

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

```

En Python

```

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

```

```

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

from typing import TypeVar

from hypothesis import given
from hypothesis import strategies as st

A = TypeVar('A')

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

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

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

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

```

1.14. Primeros y últimos elementos

En Haskell

```

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

```

```

module Primeros_y_ultimos_elementos where

```

```

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

```

En Python

```

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

```

```

from typing import TypeVar

```

```

A = TypeVar('A')

```

```

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

```

1.15. Elemento mediano

En Haskell

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

```
module Elemento_mediano where
```

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

En Python

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

1.16. Tres iguales

En Haskell

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

```
module Tres_iguales where
```

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

En Python

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

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

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

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



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

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

1.17. Tres diferentes

En Haskell

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

```
module Tres_diferentes where
```

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

En Python

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

1.18. División segura

En Haskell

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

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

```
module Division_segura where
```

```
import Test.QuickCheck
```

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

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

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

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

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

En Python

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

from hypothesis import given
from hypothesis import strategies as st

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

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

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

# La comprobación es
```

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

1.19. Disyunción excluyente

En Haskell

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

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

module Disyuncion_excluyente **where**

import Test.QuickCheck

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

-- 2ª solución
```

```

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

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

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

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

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

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

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

```

En Python

```

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

```

```

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

```

```

from typing import Any

```

```

from hypothesis import given
from hypothesis import strategies as st

```

```

# 1ª solución

```

```

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

```

```

# 2ª solución

```

```

def xor2(x: bool, y: bool) -> bool:
    if x:
        return not y
    return y

```

```

# 3ª solución

```

```

def xor3(x: bool, y: bool) -> bool:
    return (x or y) and not(x and y)

```

```

# 4ª solución

```

```

def xor4(x: bool, y: bool) -> bool:
    return (x and not y) or (y and not x)

```

```

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

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

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

```

1.20. Mayor rectángulo

En Haskell

```

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

module Mayor_rectangulo where

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

```

En Python

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

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

1.21. Intercambio de componentes de un par

En Haskell

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



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

module Intercambio_de_componentes_de_un_par where

import Test.QuickCheck

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

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

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

En Python

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

from typing import TypeVar

from hypothesis import given
from hypothesis import strategies as st

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

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

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

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

1.22. Distancia entre dos puntos

En Haskell

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

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

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

module Distancia_entre_dos_puntos where

import Test.QuickCheck

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

```

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

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

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

```

En Python

```

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

```

```

from math import sqrt

from hypothesis import given
from hypothesis import strategies as st

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

# La propiedad es
cota = 2 ** 30

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

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

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

```

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

1.23. Permutación cíclica

En Haskell

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

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

module Permutacion_ciclica where

import Test.QuickCheck

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

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

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

En Python

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

from typing import TypeVar

from hypothesis import given
from hypothesis import strategies as st

A = TypeVar('A')

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

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

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

1.24. Mayor número con dos dígitos dados

En Haskell

```

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

module Mayor_numero_con_dos_digitos_dados where

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

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

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

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

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

```

En Python

```

# -----
# Definir la función

```

```

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

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

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

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

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

```

1.25. Número de raíces de la ecuación de segundo grado

En Haskell

```

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

```



```
--      numeroDeRaices 5 23 12  ==  2
--
-- -----
{-# OPTIONS_GHC -fno-warn-type-defaults #-}

module Numero_de_raices_de_la_ecuacion_de_segundo_grado where

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

En Python

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

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

1.26. Raíces de la ecuación de segundo grado

En Haskell

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

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

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

```
module Raices_de_la_ecuacion_de_segundo_grado where
```

```
import Test.QuickCheck
```

```
raices :: Double -> Double -> Double -> [Double]
```

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

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

```
-- La propiedad es
```

```
prop_raices :: Double -> Double -> Double -> Property
```

```
prop_raices a b c =
```

```
  a /= 0 && not (null xs) ==> sum xs ~= (-b/a) && product xs ~= (c/a)
```

```

    where xs = raices a b c

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

```

En Python

```

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

```

```

from math import sqrt

```

```

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

```

```

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

```

```

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

```

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

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

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

1.27. Fórmula de Herón para el área de un triángulo

En Haskell

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

```
module Formula_de_Heron_para_el_area_de_un_triangulo where
```

```
area :: Double -> Double -> Double -> Double
```

```
area a b c = sqrt (s*(s-a)*(s-b)*(s-c))
```

```
  where s = (a+b+c)/2
```

En Python

```
# -----
# 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
#  $area : (float, float, float) \rightarrow float$ 
# tal que  $area(a, b, c)$  es el área del triángulo de lados a, b y c. Por
# ejemplo,
#  $area(3, 4, 5) == 6.0$ 
# -----
```

```
from math import sqrt
```

```
def area(a: float, b: float, c: float) -> float:
```

```
    s = (a+b+c)/2
```

```
    return sqrt(s*(s-a)*(s-b)*(s-c))
```

1.28. Intersección de intervalos cerrados

En Haskell

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

```

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

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

module Interseccion_de_intervalos_cerrados where

import Test.QuickCheck

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

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

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

```

En Python

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

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

```
Rectangulo = list[float]
```

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

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

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

1.29. Números racionales

En Haskell

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



```

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

module Numeros_racionales where

import Test.QuickCheck

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

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

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

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

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

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

```

En Python

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

from math import gcd

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

[illegible]

```
productoRacional(x, z)))
```

```
# La comprobación es
```

```
# src> poetry run pytest -q numeros_racionales.py
```

```
# 1 passed in 0.37s
```

Capítulo 2

Definiciones por comprensión

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

Contenido

2.1. Reconocimiento de subconjunto	62
2.2. Igualdad de conjuntos	66
2.3. Unión conjuntista de listas	70
2.4. Intersección conjuntista de listas	75
2.5. Diferencia conjuntista de listas	79
2.6. Divisores de un número	84
2.7. Divisores primos	94
2.8. Números libres de cuadrados	102
2.9. Suma de los primeros números naturales	109
2.10. Suma de los cuadrados de los primeros números naturales	114
2.11. Suma de cuadrados menos cuadrado de la suma	119
2.12. Triángulo aritmético	127
2.13. Suma de divisores	134
2.14. Números perfectos	143
2.15. Números abundantes	148
2.16. Números abundantes menores o iguales que n	152

¹<https://jaalonso.github.io/materias/PFconHaskell/temas/tema-5.html>

2.17. Todos los abundantes hasta n son pares	158
2.18. Números abundantes impares	164
2.19. Suma de múltiplos de 3 ó 5	169
2.20. Puntos dentro del círculo	176
2.21. Aproximación del número e	180
2.22. Aproximación al límite de $\sin(x)/x$ cuando x tiende a cero	190
2.23. Cálculo del número π mediante la fórmula de Leibniz	199
2.24. Ternas pitagóricas	205
2.25. Ternas pitagóricas con suma dada	209
2.26. Producto escalar	214
2.27. Representación densa de polinomios	219
2.28. Base de datos de actividades.	224

2.1. Reconocimiento de subconjunto

En Haskell

```

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

```

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

```
module Reconocimiento_de_subconjunto where
```

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

```
-- 1ª solución
```

```

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

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

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

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

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

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

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

-- Comparación de eficiencia
-- =====

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

```

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

En Python

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

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

from hypothesis import given
from hypothesis import strategies as st

setrecursionlimit(10**6)

A = TypeVar('A')

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

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



```

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

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

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

# Comprobación de equivalencia
# =====

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

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

# Comparación de eficiencia
# =====

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

# La comparación es

```

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

2.2. Igualdad de conjuntos

En Haskell

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

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

```
module Igualdad_de_conjuntos where
```

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

```
-- 1ª solución
-- =====
```

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

```

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

-- 2ª solución
-- =====

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

-- 3ª solución
-- =====

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

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

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

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

-- Comparación de eficiencia
-- =====

-- La comparación es

```

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

En Python

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

```
from timeit import Timer, default_timer
from typing import Any
```

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

```
# 1ª solución
# =====
```

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

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

```

# 2ª solución
# =====

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

# Equivalencia de las definiciones
# =====

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

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

# Comparación de eficiencia
# =====

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

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

```

2.3. Unión conjuntista de listas

En Haskell

```

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

```

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

```
module Union_conjuntista_de_listas where
```

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

```
-- 1ª solución
-- =====
```

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

```
-- 2ª solución
-- =====
```

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

```
-- 3ª solución
-- =====
```

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

```

union3 = union

-- 4ª solución
-- =====

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

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

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

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

-- Comparación de eficiencia
-- =====

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

```

```

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

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

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

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

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

```

En Python

```

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

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

```



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

setrecursionlimit(10**6)

A = TypeVar('A')

# 1ª solución
# =====

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

# 2ª solución
# =====

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

# 3ª solución
# =====

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

# 4ª solución
# =====

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

# Comprobación de equivalencia
```

```

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

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

# Comparación de eficiencia
# =====

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

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

# Comprobación de la propiedad
# =====

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

```

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

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

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

2.4. Intersección conjuntista de listas

En Haskell

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

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

module Interseccion_conjuntista_de_listas where

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

```

-- 1ª solución
-- =====

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

-- 2ª solución
-- =====

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

-- 3ª solución
-- =====

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

-- 4ª solución
-- =====

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

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

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

```

```

ys' = nub ys

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

-- Comparación de eficiencia
-- =====

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

```

En Python

```

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

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

from hypothesis import given
from hypothesis import strategies as st

```

```
setrecursionlimit(10**6)
```

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

```
# 1ª solución
```

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

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

```
# 2ª solución
```

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

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

```
# 3ª solución
```

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

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

```
# 4ª solución
```

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

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

```
# Comprobación de equivalencia
```

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

```
#
```

```

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

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

# Comparación de eficiencia
# =====

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

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

```

2.5. Diferencia conjuntista de listas

En Haskell

```

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

```

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

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

```
module Diferencia_conjuntista_de_listas where
```

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

```
-- 1ª solución
-- =====
```

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

```
-- 2ª solución
-- =====
```

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

```
-- 3ª solución
-- =====
```

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

```
-- 4ª solución
-- =====
```

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



```

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

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

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

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

-- Comparación de eficiencia
-- =====

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

```

En Python

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

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

from hypothesis import given
from hypothesis import strategies as st

setrecursionlimit(10**6)

A = TypeVar('A')

# 1ª solución
# =====

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

# 2ª solución
# =====

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

# 3ª solución
# =====
```

```

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

# 4ª solución
# =====

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

# Comprobación de equivalencia
# =====

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

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

# Comparación de eficiencia
# =====

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

# La comparación es

```

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

2.6. Divisores de un número

En Haskell

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

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

module Divisores_de_un_numero where

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

-- 1ª solución
-- =====

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

-- 2ª solución
```

```

-- =====

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

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

-- 3ª solución
-- =====

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

-- 4ª solución
-- =====

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

-- 5ª solución
-- =====

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

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

```

```

        x `esDivisorDe` n]

-- 6ª solución
-- =====

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

-- 7ª solución
-- =====

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

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

-- 8ª solución
-- =====

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

-- 9ª solución
-- =====

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

```

```

        . productoCartesiano
        . map inits
        . group
        . primeFactors

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

-- 10ª solución
-- =====

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

-- 11ª solución
-- =====

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

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

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

```

```

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

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

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

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

```



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

En Python

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

```

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

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

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

setrecursionlimit(10**6)

# 1ª solución
# =====

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

# 2ª solución
# =====

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

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

# 3ª solución
# =====

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

```

```
# 4ª solución
```

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

```
# primerosDivisores(n) es la lista de los divisores del número n cuyo  
# cuadrado es menor o igual que n. Por ejemplo,
```

```
#     primerosDivisores(25) == [1,5]
```

```
#     primerosDivisores(30) == [1,2,3,5]
```

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

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

```
# 5ª solución
```

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

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

```
# 6ª solución
```

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

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

7ª solución

=====

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

8ª solución

=====

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

Comprobación de equivalencia

=====

La propiedad es

@given(st.integers(min_value=2, max_value=1000))

```
def test_divisores(n: int) -> None:
```

```
    assert divisores1(n) ==\
           divisores2(n) ==\
           divisores3(n) ==\
           divisores4(n) ==\
           divisores5(n) ==\
           divisores6(n) ==\
           divisores7(n) ==\
           divisores8(n)
```

La comprobación es

src> poetry run pytest -q divisores_de_un_numero.py

1 passed in 0.84s

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

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

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

2.7. Divisores primos

En Haskell

```

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

```

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

```
module Divisores_primos where
```

```

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

```

```

-- 1ª solución
-- =====

```

```

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

```

```

-- (divisores n) es la lista de los divisores del número n. Por ejemplo,
--   divisores 25 == [1,5,25]
--   divisores 30 == [1,2,3,5,6,10,15,30]

```

```

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

```

```

-- (primo n) se verifica si n es primo. Por ejemplo,
--   primo 30 == False
--   primo 31 == True

```

```

primo1 :: Integer -> Bool
primo1 n = divisores1 n == [1, n]

```

```
-- 2ª solución
```

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

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

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

```
-- (primerosDivisores n) es la lista de los divisores del número n cuyo
-- cuadrado es menor o igual que n. Por ejemplo,
```

```
--   primerosDivisores 25 == [1,5]
```

```
--   primerosDivisores 30 == [1,2,3,5]
```

```
primerosDivisores2 :: Integer -> [Integer]
primerosDivisores2 n =
  [x | x <- [1..round (sqrt (fromIntegral n))],
    n `mod` x == 0]
```

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

```
-- 3ª solución
```

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

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

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

```

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

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

-- 4ª solución
-- =====

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

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

-- 5ª solución
-- =====

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

-- 6ª solución
-- =====

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

-- Comprobación de equivalencia

```



```

-- =====

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

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

-- Comparación de eficiencia
-- =====

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

```

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

En Python

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

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

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

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

```
setrecursionlimit(10**6)
```

```
# 1ª solución
# =====
```

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

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

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

```
# 2ª solución
# =====
```

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

```

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

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

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

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

# 3ª solución
# =====

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

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

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

```

```

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

# 4ª solución
# =====

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

# 5ª solución
# =====

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

# Comprobación de equivalencia
# =====

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

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

# Comparación de eficiencia
# =====

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

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

```

```

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

```

2.8. Números libres de cuadrados

En Haskell

```

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

```

```

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

```

```

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

```

```

module Numeros_libres_de_cuadrados where

```

```

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

```

```

-- 1ª solución
-- =====

```

```

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

```

```

-- 2ª solución
-- =====

```

```

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

```

```

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

```

```

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

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

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

-- 3ª solución
-- =====

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

-- 4ª solución
-- =====

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

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

-- 5ª solución
-- =====

```



```

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

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

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

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

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

-- Comparación de eficiencia
-- =====

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

```

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

En Python

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

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

from hypothesis import given
```

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

setrecursionlimit(10**6)

# 1ª solución
# =====

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

# 2ª solución
# =====

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

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

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

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

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

```

    return x == producto(divisoresPrimos1(x))

# 3ª solución
# =====

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

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

# 4ª solución
# =====

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

# Comprobación de equivalencia
# =====

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

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

```

```

# Comparación de eficiencia
# =====

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

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

```

2.9. Suma de los primeros números naturales

En Haskell

```

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

```

```

module Suma_de_los_primeros_numeros_naturales where

```

```

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

```

```

-- 1ª solución
-- =====

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

-- 2ª solución
-- =====

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

-- 3ª solución
-- =====

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

-- 4ª solución
-- =====

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

-- 5ª solución
-- =====

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

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

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

```

        suma3 n,
        suma4 n,
        suma5 n]

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

-- Comparación de eficiencia
-- =====

-- La comparación es
--   λ> suma1 (5*10^6)
--   125000025000000
--   (1.23 secs, 806,692,792 bytes)
--   λ> suma2 (5*10^6)
--   125000025000000
--   (0.02 secs, 559,064 bytes)
--   λ> suma3 (5*10^6)
--   125000025000000
--   (3.06 secs, 1,214,684,352 bytes)
--   λ> suma4 (5*10^6)
--   125000025000000
--   (1.25 secs, 806,692,848 bytes)
--   λ> suma5 (5*10^6)
--   125000025000000
--   (0.26 secs, 440,559,048 bytes)

```

En Python

```

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

from functools import reduce
from operator import add

```

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

from hypothesis import given
from hypothesis import strategies as st

setrecursionlimit(10**8)

# 1ª solución
# =====

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

# 2ª solución
# =====

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

# 3ª solución
# =====

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

# 4ª solución
# =====

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

# 5ª solución
# =====

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



```

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

# 6ª solución
# =====

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

# Comprobación de equivalencia
# =====

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

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

# Comparación de eficiencia
# =====

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

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

```

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

```

2.10. Suma de los cuadrados de los primeros números naturales

En Haskell

```

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

```

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

```
module Suma_de_los_cuadrados_de_los_primeros_numeros_naturales where

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

-- 1ª solución
-- =====

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

-- 2ª solución
-- =====

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

-- 3ª solución
-- =====

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

-- 4ª solución
-- =====

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

-- 5ª solución
-- =====

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

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

```

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

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

-- Comparación de eficiencia
-- =====

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

```

En Python

```

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

```

```

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

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

from hypothesis import given
from hypothesis import strategies as st

setrecursionlimit(10**6)

# 1ª solución
# =====

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

# 2ª solución
# =====

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

# 3ª solución
# =====

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

# 4ª solución
# =====

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

```

5ª solución

=====

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

6ª solución

=====

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

Comprobación de equivalencia

=====

La propiedad es

@given(st.integers(min_value=1, max_value=1000))

```
def test_sumaDeCuadrados(n: int) -> None:
```

```
    r = sumaDeCuadrados1(n)
```

```
    assert sumaDeCuadrados2(n) == r
```

```
    assert sumaDeCuadrados3(n) == r
```

```
    assert sumaDeCuadrados4(n) == r
```

```
    assert sumaDeCuadrados5(n) == r
```

```
    assert sumaDeCuadrados6(n) == r
```

La comprobación es

```
# src> poetry run pytest -q suma_de_los_cuadrados_de_los_primeros_numeros_natu
```

```
# 1 passed in 0.19s
```

Comparación de eficiencia

=====

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

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

2.11. Suma de cuadrados menos cuadrado de la suma

En Haskell

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

```

-- de los n primeros números y la suma de los cuadrados de los n
-- primeros números. Por ejemplo,
--     euler6 10      == 2640
--     euler6 (10^10) == 25000000000166666666641666666665000000000
--
-- Nota: Este ejercicio está basado en el problema 6 del proyecto Euler
-- https://www.projecteuler.net/problem=6
-- -----

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

module Suma_de_cuadrados_menos_cuadrado_de_la_suma where

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

-- 1ª solución
-- =====

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

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

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

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

-- 2ª solución
-- =====

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

```



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

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

-- 3ª solución
-- =====

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

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

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

-- 4ª solución
-- =====

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

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

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

-- 5ª solución
-- =====

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

suma5 :: Integer -> Integer
```

```

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

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

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

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

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

-- Comparación de eficiencia
-- =====

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

```

```
--
--  λ> euler6a (10^7)
--  25000000166666641666665000000
--  (11.14 secs, 8,917,796,648 bytes)
--  λ> euler6b (10^7)
--  25000000166666641666665000000
--  (0.01 secs, 570,752 bytes)
--  λ> euler6c (10^7)
--  *** Exception: stack overflow
--  λ> euler6d (10^7)
--  25000000166666641666665000000
--  (9.47 secs, 8,517,796,760 bytes)
--  λ> euler6e (10^7)
--  25000000166666641666665000000
--  (3.78 secs, 7,049,100,104 bytes)
```

En Python

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

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

from hypothesis import given
from hypothesis import strategies as st

setrecursionlimit(10**6)
```

```
# 1ª solución
# =====

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

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

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

# 2ª solución
# =====

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

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

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

# 3ª solución
# =====

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

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

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

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

# 4ª solución
# =====

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

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

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

# 5ª solución
# =====

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

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

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

# 6ª solución
```

```

# =====

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

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

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

# Comprobación de equivalencia
# =====

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

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

# Comparación de eficiencia
# =====

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

```

```

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

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

```

2.12. Triángulo aritmético

En Haskell

```

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

```

```
-- Definir las funciones
--   linea      :: Integer -> [Integer]
--   triangulo  :: Integer -> [[Integer]]
--   tales que
-- + (linea n) es la línea n-ésima de los triángulos aritméticos. Por
--   ejemplo,
--       linea 4 == [7,8,9,10]
--       linea 5 == [11,12,13,14,15]
--       head (linea (10^20)) == 4999999999999999999500000000000000000001
-- + (triangulo n) es el triángulo aritmético de altura n. Por ejemplo,
--       triangulo 3 == [[1],[2,3],[4,5,6]]
--       triangulo 4 == [[1],[2,3],[4,5,6],[7,8,9,10]]
-- -----
```

```
module Triangulo_aritmetico where
```

```
import Test.QuickCheck
```

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

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

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

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

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

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

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



```

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

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

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

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

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

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

-- La comparación es
--   λ> last (linea1 (10^7))
--   5000000050000000
--   (5.10 secs, 3,945,159,856 bytes)
--   λ> last (linea2 (10^7))
--   5000000050000000
--   (3.11 secs, 2,332,859,512 bytes)
--   λ> last (linea3 (10^7))
--   5000000050000000
--   (0.16 secs, 720,559,384 bytes)

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

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

```

```

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

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

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

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

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

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

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

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

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

```

En Python

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

from timeit import Timer, default_timer

from hypothesis import given
from hypothesis import strategies as st

# 1ª definición de línea
# =====

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

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

```

# 2ª definición de línea
# =====

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

# 3ª definición de línea
# =====

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

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

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

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

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

# Comparación de eficiencia
# =====

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

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

```

```

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

# 1ª definición de triangulo
# =====

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

# 2ª definición de triangulo
# =====

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

# 3ª definición de triangulo
# =====

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

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

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

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

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

```

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

2.13. Suma de divisores

En Haskell

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

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

```
module Suma_de_divisores where
```

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

```
-- 1ª solución
-- =====
```

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

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

```

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

-- 2ª solución
-- =====

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

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

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

-- 3ª solución
-- =====

-- La solución anterior se puede simplificar

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

-- 4ª solución
-- =====

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

-- 5ª solución
-- =====

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

```

```

        | otherwise      = aux xs

-- 6ª solución
-- =====

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

-- 7ª solución
-- =====

-- Si la descomposición de x en factores primos es
--   x = p(1)^e(1) . p(2)^e(2) . .... . p(n)^e(n)
-- entonces la suma de los divisores de x es
--   
$$\frac{p(1)^{(e(1)+1)} - 1}{p(1) - 1} \cdot \frac{p(2)^{(e(2)+1)} - 1}{p(2) - 1} \cdot \dots \cdot \frac{p(n)^{(e(n)+1)} - 1}{p(n) - 1}$$

--   -----
--   Ver la demostración en http://bit.ly/2zUXZPc

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

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

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

```



```

-- 8ª solución
-- =====

sumaDivisores8 :: Integer -> Integer
sumaDivisores8 = sigma 1

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

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

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

-- Comparación de eficiencia
-- =====

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

```

```

--      21386001
--      (0.01 secs, 648,992 bytes)
--      λ> sumaDivisores5 5336100
--      21386001
--      (2.44 secs, 1,323,924,176 bytes)
--      λ> sumaDivisores6 5336100
--      21386001
--      (0.01 secs, 832,104 bytes)
--      λ> sumaDivisores7 5336100
--      21386001
--      (0.01 secs, 571,040 bytes)
--      λ> sumaDivisores8 5336100
--      21386001
--      (0.00 secs, 558,296 bytes)
--
--      λ> sumaDivisores2 2518889234233154695211098800000000
--      1471072204661054993275791673480320
--      (2.30 secs, 1,130,862,080 bytes)
--      λ> sumaDivisores3 2518889234233154695211098800000000
--      1471072204661054993275791673480320
--      (1.83 secs, 896,386,232 bytes)
--      λ> sumaDivisores4 2518889234233154695211098800000000
--      1471072204661054993275791673480320
--      (1.52 secs, 997,992,328 bytes)
--      λ> sumaDivisores6 2518889234233154695211098800000000
--      1471072204661054993275791673480320
--      (2.35 secs, 5,719,848,600 bytes)
--      λ> sumaDivisores7 2518889234233154695211098800000000
--      1471072204661054993275791673480320
--      (0.00 secs, 628,136 bytes)
--      λ> sumaDivisores8 2518889234233154695211098800000000
--      1471072204661054993275791673480320
--      (0.00 secs, 591,352 bytes)
--
--      λ> length (show (sumaDivisores7 (product [1..30000])))
--      121289
--      (2.76 secs, 4,864,576,304 bytes)
--      λ> length (show (sumaDivisores8 (product [1..30000])))
--      121289
--      (1.65 secs, 3,173,319,312 bytes)

```

En Python

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

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

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

setrecursionlimit(10**6)

# 1ª solución
# =====

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

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

# 2ª solución
# =====

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

siguiente definición es la más eficiente.

```
def sumaDivisores2(n: int) -> int:
    return sum(divisors(n))
```

3ª solución

=====

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

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

4ª solución

=====

Si la descomposición de x en factores primos es

$x = p(1)^{e(1)} \cdot p(2)^{e(2)} \cdot \dots \cdot p(n)^{e(n)}$

entonces la suma de los divisores de x es

$\frac{p(1)^{e(1)+1} - 1}{p(1) - 1} \cdot \frac{p(2)^{e(2)+1} - 1}{p(2) - 1} \cdot \dots \cdot \frac{p(n)^{e(n)+1} - 1}{p(n) - 1}$

Ver la demostración en <http://bit.ly/2zUXZPc>

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

5ª solución

=====

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

```

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

# 6ª solución
# =====

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

# Comprobación de equivalencia
# =====

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

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

# Comparación de eficiencia
# =====

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

# La comparación es

```

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

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

2.14. Números perfectos

En Haskell

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

```
module Numeros_perfectos where
```

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

```
-- 1ª solución
-- =====
```

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

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

```

esPerfecto1 x =
    sumaDivisores1 x - x == x

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

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

-- 2ª solución
-- =====

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

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

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

sumaDivisores2 :: Integer -> Integer
sumaDivisores2 = sigma 1

-- 3ª solución
-- =====

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

```



```

-- 4ª solución
-- =====

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

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

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

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

-- Comparación de eficiencia
-- =====

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

```

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

En Python

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

```
from timeit import Timer, default_timer
```

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

```
# 1ª solución
# =====
```

```
# divisores(n) es la lista de los divisores del número n. Por ejemplo,
# divisores(30) == [1,2,3,5,6,10,15,30]
```

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

```
# sumaDivisores(x) es la suma de los divisores de x. Por ejemplo,
# sumaDivisores(12) == 28
# sumaDivisores(25) == 31
```

```
def sumaDivisores1(n: int) -> int:
    return sum(divisores1(n))
```

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

```

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

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

# 2ª solución
# =====

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

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

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

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

# Comprobación de equivalencia
# =====

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

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

# Comparación de eficiencia
# =====

```

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

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

2.15. Números abundantes

En Haskell

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

module Numeros_abundantes where

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

-- 1ª solución
```

```

-- =====

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

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

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

-- 2ª solución
-- =====

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

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

sumaDivisores2 :: Integer -> Integer
sumaDivisores2 = sigma 1

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

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

```

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

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

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

En Python

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

```
from timeit import Timer, default_timer
```

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

```

from sympy import divisor_sigma

# 1ª solución
# =====

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

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

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

# 2ª solución
# =====

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

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

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

# Comprobación de equivalencia
# =====

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

```

```

    assert numeroAbundante1(n) == numeroAbundante2(n)

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

# Comparación de eficiencia
# =====

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

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

```

2.16. Números abundantes menores o iguales que n

En Haskell

```

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

```



```

module Numeros_abundantes_menores_o_iguales_que_n where

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

-- 1ª solución
-- =====

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

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

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

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

-- 2ª solución
-- =====

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

```

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

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

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

```
sumaDivisores2 :: Integer -> Integer
sumaDivisores2 = sigma 1
```

```
-- 3ª solución
-- =====
```

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

```
-- 4ª solución
-- =====
```

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

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

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

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

-- Comparación de eficiencia
-- =====

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

--   λ> length (numerosAbundantesMenores2 (10^6))
--   247545
--   (1.48 secs, 2,543,048,024 bytes)
--   λ> length (numerosAbundantesMenores3 (10^6))
--   247545
--   (1.30 secs, 2,499,087,272 bytes)
--   λ> length (numerosAbundantesMenores4 (10^6))
--   247545
--   (1.30 secs, 2,499,087,248 bytes)
```

En Python

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

```

#     leng(numerosAbundantesMenores(10**6)) == 247545
# -----

from timeit import Timer, default_timer

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

# 1ª solución
# =====

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

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

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

def numerosAbundantesMenores1(n: int) -> list[int]:
    return [x for x in range(1, n + 1) if numeroAbundante1(x)]

# 2ª solución
# =====

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

```

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

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

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

def numerosAbundantesMenores2(n: int) -> list[int]:
    return [x for x in range(1, n + 1) if numeroAbundante2(x)]

# Comprobación de equivalencia
# =====

# La propiedad es
@given(st.integers(min_value=2, max_value=1000))
def test_numerosAbundantesMenores(n: int) -> None:
    assert numerosAbundantesMenores1(n) == numerosAbundantesMenores2(n)

# La comprobación es
# src> poetry run pytest -q numeros_abundantes_menores_o_iguales_que_n.py
# 1 passed in 1.54s

# Comparación de eficiencia
# =====

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

# La comparación es
# >>> tiempo('len(numerosAbundantesMenores1(10**4))')
# 2.21 segundos
# >>> tiempo('len(numerosAbundantesMenores2(10**4))')
# 0.55 segundos
#
# >>> tiempo('len(numerosAbundantesMenores2(10**5))')
```

5.96 segundos

2.17. Todos los abundantes hasta n son pares

En Haskell

```

-- -----
-- Definir la función
--   todosPares :: Integer -> Bool
-- tal que (todosPares n) se verifica si todos los números abundantes
-- menores o iguales que n son pares. Por ejemplo,
--   todosPares 10    == True
--   todosPares 100   == True
--   todosPares 1000  == False
-- -----

module Todos_los_abundantes_hasta_n_son_pares where

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

-- 1ª solución
-- =====

todosPares1 :: Integer -> Bool
todosPares1 n = and [even x | x <- numerosAbundantesMenores1 n]

-- (numerosAbundantesMenores n) es la lista de números abundantes
-- menores o iguales que n. Por ejemplo,
--   numerosAbundantesMenores 50 == [12,18,20,24,30,36,40,42,48]
--   numerosAbundantesMenores 48 == [12,18,20,24,30,36,40,42,48]
numerosAbundantesMenores1 :: Integer -> [Integer]
numerosAbundantesMenores1 n =
    [x | x <- [1..n],
        numeroAbundante1 x]

-- (numeroAbundante n) se verifica si n es un número abundante. Por
-- ejemplo,
--   numeroAbundante 5 == False
--   numeroAbundante 12 == True

```

```

--      numeroAbundante 28 == False
--      numeroAbundante 30 == True
numeroAbundante1 :: Integer -> Bool
numeroAbundante1 x =
  x < sumaDivisores1 x - x

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

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

-- 2ª solución
-- =====

-- Sustituyendo la definición de numerosAbundantesMenores de la solución
-- anterior por cada una de las del ejercicio anterior se obtiene una
-- nueva definición de todosPares. La usada en la definición anterior es
-- la menos eficiente y la que se usa en la siguiente definición es la
-- más eficiente.

todosPares2 :: Integer -> Bool
todosPares2 n = and [even x | x <- numerosAbundantesMenores2 n]

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

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

sumaDivisores2 :: Integer -> Integer
sumaDivisores2 = sigma 1

```

```

-- 3ª solución
-- =====

todosPares3 :: Integer -> Bool
todosPares3 1 = True
todosPares3 n | numeroAbundante1 n = even n && todosPares3 (n-1)
              | otherwise          = todosPares3 (n-1)

-- 4ª solución
-- =====

todosPares4 :: Integer -> Bool
todosPares4 n = all even (numerosAbundantesMenores1 n)

-- 5ª solución
-- =====

todosPares5 :: Integer -> Bool
todosPares5 = all even . numerosAbundantesMenores1

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

-- La propiedad es
prop_todosPares :: Positive Integer -> Bool
prop_todosPares (Positive n) =
  all (== todosPares1 n)
    [todosPares2 n,
     todosPares3 n,
     todosPares4 n,
     todosPares5 n]

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

-- Comparación de eficiencia
-- =====

```



```
-- La comparación es
--   λ> todosPares1 (10^3)
--   False
--   (0.22 secs, 91,257,744 bytes)
--   λ> todosPares2 (10^3)
--   False
--   (0.01 secs, 2,535,656 bytes)
--   λ> todosPares3 (10^3)
--   False
--   (0.03 secs, 11,530,528 bytes)
--   λ> todosPares4 (10^3)
--   False
--   (0.24 secs, 91,231,144 bytes)
--   λ> todosPares5 (10^3)
--   False
--   (0.22 secs, 91,231,208 bytes)
```

En Python

```
# -----
# Definir la función
#   todosPares : (int) -> bool
# tal que todosPares(n) se verifica si todos los números abundantes
# menores o iguales que n son pares. Por ejemplo,
#   todosPares(10)    == True
#   todosPares(100)   == True
#   todosPares(1000)  == False
# -----
```

```
from timeit import Timer, default_timer
```

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

```
# 1ª solución
# =====
```

```
# divisores(n) es la lista de los divisores del número n. Por ejemplo,
#   divisores(30) == [1,2,3,5,6,10,15,30]
```

```

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

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

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

# numerosAbundantesMenores(n) es la lista de números abundantes menores
# o iguales que n. Por ejemplo,
#     numerosAbundantesMenores(50) == [12, 18, 20, 24, 30, 36, 40, 42, 48]
#     numerosAbundantesMenores(48) == [12, 18, 20, 24, 30, 36, 40, 42, 48]
def numerosAbundantesMenores1(n: int) -> list[int]:
    return [x for x in range(1, n + 1) if numeroAbundante1(x)]

def todosPares1(n: int) -> bool:
    return False not in [x % 2 == 0 for x in numerosAbundantesMenores1(n)]

# 2ª solución
# =====

# Sustituyendo la definición de numerosAbundantesMenores de la solución
# anterior por cada una de las del ejercicio anterior se obtiene una
# nueva definición de todosPares. La usada en la definición anterior es
# la menos eficiente y la que se usa en la siguiente definición es la
# más eficiente.

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

```

```

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

def numerosAbundantesMenores2(n: int) -> list[int]:
    return [x for x in range(1, n + 1) if numeroAbundante2(x)]

def todosPares2(n: int) -> bool:
    return False not in [x % 2 == 0 for x in numerosAbundantesMenores2(n)]

# 3ª solución
# =====

def todosPares3(n: int) -> bool:
    return all(x % 2 == 0 for x in numerosAbundantesMenores1(n))

# Comprobación de equivalencia
# =====

# La propiedad es
@given(st.integers(min_value=2, max_value=1000))
def test_todosPares(n: int) -> None:
    assert todosPares1(n) == todosPares2(n) == todosPares3(n)

# La comprobación es
# src> poetry run pytest -q todos_los_abundantes_hasta_n_son_pares.py
# 1 passed in 2.63s

# Comparación de eficiencia
# =====

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

# La comparación es
# >>> tiempo('todosPares1(1000)')
# 0.03 segundos
# >>> tiempo('todosPares2(1000)')
# 0.05 segundos

```

```
# >>> tiempo('todosPares3(1000)')
# 0.02 segundos
#
# >>> tiempo('todosPares1(10000)')
# 2.07 segundos
# >>> tiempo('todosPares2(10000)')
# 0.47 segundos
# >>> tiempo('todosPares3(10000)')
# 2.42 segundos
```

2.18. Números abundantes impares

En Haskell

```
-- -----
-- Definir la lista
-- abundantesImpares :: [Integer]
-- cuyos elementos son los números abundantes impares. Por ejemplo,
-- λ> take 12 abundantesImpares
-- [945,1575,2205,2835,3465,4095,4725,5355,5775,5985,6435,6615]
-- -----
```

```
module Numeros_abundantes_impares where
```

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

```
-- 1ª solución
-- =====
```

```
abundantesImpares1 :: [Integer]
abundantesImpares1 = [x | x <- [1,3..], numeroAbundante1 x]
```

```
-- (numeroAbundante n) se verifica si n es un número abundante. Por
-- ejemplo,
-- numeroAbundante 5 == False
-- numeroAbundante 12 == True
-- numeroAbundante 28 == False
-- numeroAbundante 30 == True
```

```

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

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

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

-- 2ª solución
-- =====

abundantesImpares2 :: [Integer]
abundantesImpares2 = filter numeroAbundante1 [1,3..]

-- 3ª solución
-- =====

-- Sustituyendo la definición de numeroAbundante1 de las soluciones
-- anteriores por cada una de las del ejercicio "Números abundantes"
-- https://bit.ly/3xSlWDU se obtiene una nueva definición de abundantes
-- impares. La usada en las definiciones anteriores es la menos
-- eficiente y la que se usa en la siguiente definición es la más eficiente.

abundantesImpares3 :: [Integer]
abundantesImpares3 = filter numeroAbundante3 [1,3..]

numeroAbundante3 :: Integer -> Bool
numeroAbundante3 x =
    x < sumaDivisores3 x - x

sumaDivisores3 :: Integer -> Integer
sumaDivisores3 = sigma 1

```

```

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

-- La propiedad es
prop_abundantesImpares :: Positive Int -> Bool
prop_abundantesImpares (Positive n) =
    all (== take n abundantesImpares1)
        [take n abundantesImpares2,
         take n abundantesImpares3]

-- La comprobación es
--    λ> quickCheckWith (stdArgs {maxSize=10}) prop_abundantesImpares
--    +++ OK, passed 100 tests.

-- Comparación de eficiencia
-- =====

-- La comparación es
--    λ> abundantesImpares1 !! 5
--    4095
--    (2.07 secs, 841,525,368 bytes)
--    λ> abundantesImpares2 !! 5
--    4095
--    (2.06 secs, 841,443,112 bytes)
--    λ> abundantesImpares3 !! 5
--    4095
--    (0.01 secs, 550,776 bytes)

```

En Python

```

# -----
# Definir la función
# abundantesImpares : (int) -> list[int]
# tal que abundantesImpares(n) son los números abundantes impares
# menores que n. Por ejemplo,
#     >>> abundantesImpares1(10000)[:12]
#     [945, 1575, 2205, 2835, 3465, 4095, 4725, 5355, 5775, 5985, 6435, 6615]
# -----

from timeit import Timer, default_timer

```

```

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

# 1ª solución
# =====

def abundantesImpares1(n: int) -> list[int]:
    return [x for x in range(1, n, 2) if numeroAbundante1(x)]

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

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

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

# 2ª solución
# =====

def abundantesImpares2(n: int) -> list[int]:
    return list(filter(numeroAbundante1, range(1, n, 2)))

# 3ª solución
# =====
#

```

Sustituyendo la definición de numeroAbundante1 de las soluciones anteriores por cada una de las del ejercicio "Números abundantes" <https://bit.ly/3xSlWDU> se obtiene una nueva definición de abundantes impares. La usada en las definiciones anteriores es la menos eficiente y la que se usa en la siguiente definición es la más eficiente.

```
def abundantesImpares3(n: int) -> list[int]:
    return list(filter(numeroAbundante3, range(1, n, 2)))
```

```
def sumaDivisores3(n: int) -> int:
    return divisor_sigma(n, 1)
```

```
def numeroAbundante3(x: int) -> bool:
    return x < sumaDivisores3(x) - x
```

Comprobación de equivalencia
=====

La propiedad es
 @given(st.integers(min_value=1, max_value=1000))
 def test_abundantesImpares(n: int) -> None:
 r = abundantesImpares1(n)
 assert abundantesImpares2(n) == r
 assert abundantesImpares3(n) == r

La comprobación es
src> poetry run pytest -q numeros_abundantes_impares.py
1 passed in 1.42s

Comparación de eficiencia
=====

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

La comparación es
>>> tiempo('abundantesImpares1(10000)[5]')
1.25 segundos


```
# >>> tiempo('abundantesImpares2(10000)[5]')
# 1.22 segundos
# >>> tiempo('abundantesImpares3(10000)[5]')
# 0.33 segundos
```

2.19. Suma de múltiplos de 3 ó 5

En Haskell

```
-----
-- Definir la función
-- euler1 :: Integer -> Integer
-- tal que (euler1 n) es la suma de todos los múltiplos de 3 ó 5 menores
-- que n. Por ejemplo,
-- euler1 10      == 23
-- euler1 (10^2)  == 2318
-- euler1 (10^3)  == 233168
-- euler1 (10^4)  == 23331668
-- euler1 (10^5)  == 2333316668
-- euler1 (10^10) == 23333333331666666668
-- euler1 (10^20) == 233333333333333333316666666666666668
-- euler1 (10^30) == 2333333333333333333333333333333333316666666666666666668
--
-- Nota: Este ejercicio está basado en el problema 1 del Proyecto Euler
-- https://projecteuler.net/problem=1
-----

module Suma_de_multiplos_de_3_o_5 where

import Data.List (nub, union)
import qualified Data.Set as S (fromAscList, union)
import Test.QuickCheck

-- 1ª solución
-- =====

euler1a :: Integer -> Integer
euler1a n =
    sum [x | x <- [1..n-1],
            multiplo x 3 || multiplo x 5]
```

```
-- (multiplo x y) se verifica si x es un múltiplo de y. Por ejemplo.
--     multiplo 12 3 == True
--     multiplo 14 3 == False
multiplo :: Integer -> Integer -> Bool
multiplo x y = mod x y == 0
```

```
-- 2ª solución --
-- =====
```

```
euler1b :: Integer -> Integer
euler1b n =
    sum [x | x <- [1..n-1],
            gcd x 15 > 1]
```

```
-- 3ª solución --
-- =====
```

```
euler1c :: Integer -> Integer
euler1c n =
    sum [3,6..n-1] + sum [5,10..n-1] - sum [15,30..n-1]
```

```
-- 4ª solución --
-- =====
```

```
euler1d :: Integer -> Integer
euler1d n =
    sum (nub ([3,6..n-1] ++ [5,10..n-1]))
```

```
-- 5ª solución --
-- =====
```

```
euler1e :: Integer -> Integer
euler1e n =
    sum ([3,6..n-1] `union` [5,10..n-1])
```

```
-- 6ª solución --
-- =====
```

```
euler1f :: Integer -> Integer
```

```

euler1f n =
    sum (S.fromAscList [3,6..n-1] `S.union` S.fromAscList [5,10..n-1])

-- 7ª solución
-- =====

euler1g :: Integer -> Integer
euler1g n =
    suma 3 n + suma 5 n - suma 15 n

-- (suma d x) es la suma de los múltiplos de d menores que x. Por
-- ejemplo,
--     suma 3 20 == 63
suma :: Integer -> Integer -> Integer
suma d x = (a+b)*n `div` 2
    where a = d
          b = d * ((x-1) `div` d)
          n = 1 + (b-a) `div` d

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

-- La propiedad es
prop_euler1 :: Positive Integer -> Bool
prop_euler1 (Positive n) =
    all (== euler1a n)
        [euler1b n,
         euler1c n,
         euler1d n,
         euler1e n,
         euler1f n,
         euler1g n]

-- La comprobación es
--     λ> quickCheck prop_euler1
--     +++ OK, passed 100 tests.

-- Comparación de eficiencia
-- =====

```

```
-- La comparación es
-- λ> euler1a (5*10^4)
-- 583291668
-- (0.05 secs, 21,895,296 bytes)
-- λ> euler1b (5*10^4)
-- 583291668
-- (0.05 secs, 26,055,096 bytes)
-- λ> euler1c (5*10^4)
-- 583291668
-- (0.01 secs, 5,586,072 bytes)
-- λ> euler1d (5*10^4)
-- 583291668
-- (2.83 secs, 7,922,304 bytes)
-- λ> euler1e (5*10^4)
-- 583291668
-- (4.56 secs, 12,787,705,248 bytes)
-- λ> euler1f (5*10^4)
-- 583291668
-- (0.01 secs, 8,168,584 bytes)
-- λ> euler1g (5*10^4)
-- 583291668
-- (0.02 secs, 557,488 bytes)
--
-- λ> euler1a (3*10^6)
-- 2099998500000
-- (2.72 secs, 1,282,255,816 bytes)
-- λ> euler1b (3*10^6)
-- 2099998500000
-- (2.06 secs, 1,531,855,776 bytes)
-- λ> euler1c (3*10^6)
-- 2099998500000
-- (0.38 secs, 305,127,480 bytes)
-- λ> euler1f (3*10^6)
-- 2099998500000
-- (0.54 secs, 457,358,232 bytes)
-- λ> euler1g (3*10^6)
-- 2099998500000
-- (0.01 secs, 560,472 bytes)
--
-- λ> euler1c (10^7)
```

```
--      23333331666668
--      (1.20 secs, 1,015,920,024 bytes)
--      λ> euler1f (10^7)
--      23333331666668
--      (2.00 secs, 1,523,225,648 bytes)
--      λ> euler1g (10^7)
--      23333331666668
--      (0.01 secs, 561,200 bytes)
```

En Python

```
# -----
# Definir la función
#   euler1 : (int) -> int
# tal que euler1(n) es la suma de todos los múltiplos de 3 ó 5 menores
# que n. Por ejemplo,
#   euler1(10)      == 23
#   euler1(10**2)   == 2318
#   euler1(10**3)   == 233168
#   euler1(10**4)   == 23331668
#   euler1(10**5)   == 2333316668
#   euler1(10**10)  == 23333333331666666668
#   euler1(10**20) == 233333333333333333316666666666666668
#
# Nota: Este ejercicio está basado en el problema 1 del Proyecto Euler
# https://projecteuler.net/problem=1
# -----

from math import gcd
from timeit import Timer, default_timer

from hypothesis import given
from hypothesis import strategies as st

# 1ª solución
# =====

# multiplo(x, y) se verifica si x es un múltiplo de y. Por ejemplo.
#   multiplo(12, 3) == True
#   multiplo(14, 3) == False
```

```

def multiplo(x: int, y: int) -> int:
    return x % y == 0

def euler1a(n: int) -> int:
    return sum(x for x in range(1, n)
               if (multiplo(x, 3) or multiplo(x, 5)))

# 2ª solución
# =====

def euler1b(n: int) -> int:
    return sum(x for x in range(1, n)
               if gcd(x, 15) > 1)

# 3ª solución
# =====

def euler1c(n: int) -> int:
    return sum(range(3, n, 3)) + \
           sum(range(5, n, 5)) - \
           sum(range(15, n, 15))

# 4ª solución
# =====

def euler1d(n: int) -> int:
    return sum(set(list(range(3, n, 3)) + list(range(5, n, 5))))

# 5ª solución
# =====

def euler1e(n: int) -> int:
    return sum(set(list(range(3, n, 3))) | set(list(range(5, n, 5))))

# 6ª solución
# =====

# suma(d, x) es la suma de los múltiplos de d menores que x. Por
# ejemplo,
#     suma(3, 20) == 63

```

```

def suma(d: int, x: int) -> int:
    a = d
    b = d * ((x - 1) // d)
    n = 1 + (b - a) // d
    return (a + b) * n // 2

def euler1f(n: int) -> int:
    return suma(3, n) + suma(5, n) - suma(15, n)

# Comprobación de equivalencia
# =====

# La propiedad es
@given(st.integers(min_value=1, max_value=1000))
def test_euler1(n: int) -> None:
    r = euler1a(n)
    assert euler1b(n) == r
    assert euler1c(n) == r
    assert euler1d(n) == r
    assert euler1e(n) == r

# La comprobación es
# src> poetry run pytest -q suma_de_multiplos_de_3_o_5.py
# 1 passed in 0.16s

# Comparación de eficiencia
# =====

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

# La comparación es
# >>> tiempo('euler1a(10**7)')
# 1.49 segundos
# >>> tiempo('euler1b(10**7)')
# 0.93 segundos
# >>> tiempo('euler1c(10**7)')
# 0.07 segundos

```

```
# >>> tiempo('euler1d(10**7)')
# 0.42 segundos
# >>> tiempo('euler1e(10**7)')
# 0.69 segundos
# >>> tiempo('euler1f(10**7)')
# 0.00 segundos
#
# >>> tiempo('euler1c(10**8)')
# 0.72 segundos
# >>> tiempo('euler1f(10**8)')
# 0.00 segundos
```

2.20. Puntos dentro del círculo

En Haskell

```
-- -----
-- En el círculo de radio 2 hay 6 puntos cuyas coordenadas son puntos
-- naturales:
-- (0,0),(0,1),(0,2),(1,0),(1,1),(2,0)
-- y en de radio 3 hay 11:
-- (0,0),(0,1),(0,2),(0,3),(1,0),(1,1),(1,2),(2,0),(2,1),(2,2),(3,0)
--
-- Definir la función
-- círculo :: Int -> Int
-- tal que (círculo n) es el la cantidad de pares de números naturales
-- (x,y) que se encuentran en el círculo de radio n. Por ejemplo,
-- círculo 1 == 3
-- círculo 2 == 6
-- círculo 3 == 11
-- círculo 4 == 17
-- círculo 100 == 7955
-- -----
```

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

```
module Puntos_dentro_del_circulo where
```

```
import Test.QuickCheck
```



```

-- 1ª solución
-- =====

circulo1 :: Int -> Int
circulo1 n = length (enCirculo1 n)

enCirculo1 :: Int -> [(Int, Int)]
enCirculo1 n = [(x,y) | x <- [0..n],
                        y <- [0..n],
                        x*x+y*y <= n*n]

-- 2ª solución
-- =====

circulo2 :: Int -> Int
circulo2 0 = 1
circulo2 n =
  2 * length (enSemiCirculo n) + ceiling(fromIntegral n / sqrt 2)

enSemiCirculo :: Int -> [(Int, Int)]
enSemiCirculo n =
  [(x,y) | x <- [0..floor (sqrt (fromIntegral (n * n)))],
           y <- [x+1..truncate (sqrt (fromIntegral (n*n - x*x)))] ]

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

-- La propiedad es
prop_circulo :: Positive Int -> Bool
prop_circulo (Positive n) =
  circulo1 n == circulo2 n

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

-- Comparación de eficiencia
-- =====

-- La comparación es

```

```
-- λ> circulo1 (2*10^3)
-- 3143587
-- (3.58 secs, 1,744,162,600 bytes)
-- λ> circulo2 (2*10^3)
-- 3143587
-- (0.41 secs, 266,374,208 bytes)
```

En Python

```
# -----
# En el círculo de radio 2 hay 6 puntos cuyas coordenadas son puntos
# naturales:
# (0,0), (0,1), (0,2), (1,0), (1,1), (2,0)
# y en de radio 3 hay 11:
# (0,0), (0,1), (0,2), (0,3), (1,0), (1,1), (1,2), (2,0), (2,1), (2,2), (3,0)
#
# Definir la función
# circulo : (int) -> int
# tal que circulo(n) es el la cantidad de pares de números naturales
# (x,y) que se encuentran en el círculo de radio n. Por ejemplo,
# circulo(1) == 3
# circulo(2) == 6
# circulo(3) == 11
# circulo(4) == 17
# circulo(100) == 7955
# -----
```

```
from math import ceil, sqrt, trunc
from timeit import Timer, default_timer
```

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

```
# 1ª solución
# =====
```

```
def circulo1(n: int) -> int:
    return len([(x, y)
                 for x in range(0, n + 1)
                 for y in range(0, n + 1)
```

```

        if x * x + y * y <= n * n])

# 2ª solución
# =====

def enSemiCirculo(n: int) -> list[tuple[int, int]]:
    return [(x, y)
            for x in range(0, ceil(sqrt(n**2)) + 1)
            for y in range(x+1, trunc(sqrt(n**2 - x**2)) + 1)]

def circulo2(n: int) -> int:
    if n == 0:
        return 1
    return (2 * len(enSemiCirculo(n)) + ceil(n / sqrt(2)))

# 3ª solución
# =====

def circulo3(n: int) -> int:
    r = 0
    for x in range(0, n + 1):
        for y in range(0, n + 1):
            if x**2 + y**2 <= n**2:
                r = r + 1
    return r

# 4ª solución
# =====

def circulo4(n: int) -> int:
    r = 0
    for x in range(0, ceil(sqrt(n**2)) + 1):
        for y in range(x + 1, trunc(sqrt(n**2 - x**2)) + 1):
            if x**2 + y**2 <= n**2:
                r = r + 1
    return 2 * r + ceil(n / sqrt(2))

# Comprobación de equivalencia
# =====

```

```

# La propiedad es
@given(st.integers(min_value=1, max_value=100))
def test_circulo(n: int) -> None:
    r = circulo1(n)
    assert circulo2(n) == r
    assert circulo3(n) == r
    assert circulo4(n) == r

# La comprobación es
#   src> poetry run pytest -q puntos_dentro_del_circulo.py
#   1 passed in 0.60s

# Comparación de eficiencia
# =====

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

# La comparación es
#   >>> tiempo('circulo1(2000)')
#   0.71 segundos
#   >>> tiempo('circulo2(2000)')
#   0.76 segundos
#   >>> tiempo('circulo3(2000)')
#   2.63 segundos
#   >>> tiempo('circulo4(2000)')
#   1.06 segundos

```

2.21. Aproximación del número e

En Haskell

```

-- -----
-- El [número e](https://bit.ly/3y17R7l) se define como el límite de la
-- sucesión  $(1+1/n)^n$ ; es decir,
--    $e = \lim (1+1/n)^n$ 
--
-- Definir las funciones

```

```

--   aproxE      :: Int -> [Double]
--   errorAproxE :: Double -> Int
--   tales que
--   + (aproxE k) es la lista de los k primeros términos de la sucesión
--      $(1+1/n)^m$ . Por ejemplo,
--     aproxE 4 == [2.0,2.25,2.37037037037037,2.44140625]
--     last (aproxE (7*10^7)) == 2.7182818287372563
--   + (errorE x) es el menor número de términos de la sucesión
--      $(1+1/m)^m$  necesarios para obtener su límite con un error menor que
--     x. Por ejemplo,
--     errorAproxE 0.1    == 13
--     errorAproxE 0.01   == 135
--     errorAproxE 0.001  == 1359
--
--   Indicación: En Haskell, e se calcula como (exp 1).
--   -----

```

```

module Aproximacion_del_numero_e where

```

```

import Test.QuickCheck

```

```

-- 1ª definición de aproxE
-- =====

```

```

aproxE1 :: Int -> [Double]
aproxE1 k = [(1+1/n)**n | n <- [1..k']]
  where k' = fromIntegral k

```

```

-- 2ª definición de aproxE
-- =====

```

```

aproxE2 :: Int -> [Double]
aproxE2 0 = []
aproxE2 n = aproxE2 (n-1) ++ [(1+1/n')**n']
  where n' = fromIntegral n

```

```

-- 3ª definición de aproxE
-- =====

```

```

aproxE3 :: Int -> [Double]

```

```

aproxE3 = reverse . aux . fromIntegral
  where aux 0 = []
        aux n = (1+1/n)**n : aux (n-1)

-- 4ª definición de aproxE
-- =====

aproxE4 :: Int -> [Double]
aproxE4 k = aux [] (fromIntegral k)
  where aux xs 0 = xs
        aux xs n = aux ((1+1/n)**n : xs) (n-1)

-- 5ª definición de aproxE
-- =====

aproxE5 :: Int -> [Double]
aproxE5 k = map (\ n -> (1+1/n)**n) [1..k']
  where k' = fromIntegral k

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

-- La propiedad es
prop_aproxE :: Positive Int -> Bool
prop_aproxE (Positive k) =
  all (== aproxE1 k)
    [aproxE2 k,
     aproxE3 k,
     aproxE4 k,
     aproxE5 k]

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

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

-- La comparación es
--   λ> last (aproxE1 (2*10^4))

```

```

--      2.718213874533619
--      (0.04 secs, 5,368,968 bytes)
--      λ> last (aproxE2 (2*10^4))
--      2.718213874533619
--      (5.93 secs, 17,514,767,104 bytes)
--      λ> last (aproxE3 (2*10^4))
--      2.718213874533619
--      (0.05 secs, 9,529,336 bytes)
--      λ> last (aproxE4 (2*10^4))
--      2.718213874533619
--      (0.05 secs, 9,529,184 bytes)
--      λ> last (aproxE5 (2*10^4))
--      2.718213874533619
--      (0.01 secs, 4,888,960 bytes)
--
--      λ> last (aproxE1 (2*10^6))
--      2.7182811492688552
--      (0.54 secs, 480,570,120 bytes)
--      λ> last (aproxE3 (2*10^6))
--      2.7182811492688552
--      (2.07 secs, 896,570,280 bytes)
--      λ> last (aproxE4 (2*10^6))
--      2.7182811492688552
--      (2.18 secs, 896,570,336 bytes)
--      λ> last (aproxE5 (2*10^6))
--      2.7182811492688552
--      (0.09 secs, 432,570,112 bytes)

-- 1ª definición de errorAproxE
-- =====

errorAproxE1 :: Double -> Int
errorAproxE1 x =
  round (head [n | n <- [1..], abs (exp 1 - (1+1/n)**n) < x])

-- 2ª definición de errorAproxE
-- =====

errorAproxE2 :: Double -> Int
errorAproxE2 x = aux 1

```

```

    where aux n | abs (exp 1 - (1+1/n)**n) < x = round n
               | otherwise                    = aux (n+1)

-- 3ª definición de errorAproxE
-- =====

errorAproxE3 :: Double -> Int
errorAproxE3 x =
    round (head (dropWhile (\ n -> abs (exp 1 - (1+1/n)**n) >= x) [1..]))

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

-- La propiedad es
prop_errorAproxE :: Positive Double -> Bool
prop_errorAproxE (Positive x) =
    all (== errorAproxE1 x)
        [errorAproxE2 x,
         errorAproxE3 x]

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

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

-- La comparación es
--    λ> errorAproxE1 0.000001
--    1358611
--    (1.70 secs, 674,424,552 bytes)
--    λ> errorAproxE2 0.000001
--    1358611
--    (1.79 secs, 739,637,704 bytes)
--    λ> errorAproxE3 0.000001
--    1358611
--    (1.20 secs, 609,211,144 bytes)

```


En Python

```
# -----
# El [número e](https://bit.ly/3y17R7l) se define como el límite de la
# sucesión  $(1+1/n)^n$ ; es decir,
#    $e = \lim (1+1/n)^n$ 
#
# Definir las funciones
#   aproxE      : (int) -> list[float]
#   errorAproxE : (float) -> int
# tales que
# + aproxE(k) es la lista de los k primeros términos de la sucesión
#    $(1+1/n)^n$ . Por ejemplo,
#   aproxE(4) == [2.0, 2.25, 2.37037037037037, 2.44140625]
#   aproxE6(7*10**7)[-1] == 2.7182818287372563
# + errorE(x) es el menor número de términos de la sucesión
#    $(1+1/m)^m$  necesarios para obtener su límite con un error menor que
#   x. Por ejemplo,
#   errorAproxE(0.1)    == 13
#   errorAproxE(0.01)   == 135
#   errorAproxE(0.001)  == 1359
# -----

from itertools import dropwhile, islice
from math import e
from sys import setrecursionlimit
from timeit import Timer, default_timer
from typing import Iterator

from hypothesis import given
from hypothesis import strategies as st

setrecursionlimit(10**6)

# 1ª definición de aproxE
# =====

def aproxE1(k: int) -> list[float]:
    return [(1 + 1/n)**n for n in range(1, k + 1)]

# 2ª definición de aproxE
```

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

```
def aproxE2(n: int) -> list[float]:
    if n == 0:
        return []
    return aproxE2(n - 1) + [(1 + 1/n)**n]
```

```
# 3ª definición de aproxE
```

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

```
def aproxE3(n: int) -> list[float]:
    def aux(n: int) -> list[float]:
        if n == 0:
            return []
        return [(1 + 1/n)**n] + aux(n - 1)

    return list(reversed(aux(n)))
```

```
# 4ª definición de aproxE
```

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

```
def aproxE4(n: int) -> list[float]:
    def aux(xs: list[float], n: int) -> list[float]:
        if n == 0:
            return xs
        return aux([(1 + 1/n)**n] + xs, n - 1)

    return aux([], n)
```

```
# 5ª definición de aproxE
```

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

```
def aproxE5(n: int) -> list[float]:
    return list(map((lambda k: (1+1/k)**k), range(1, n+1)))
```

```
# 6ª definición de aproxE
```

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

```
def aproxE6(n: int) -> list[float]:
    r = []
```

```

    for k in range(1, n+1):
        r.append((1+1/k)**k)
    return r

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

# La propiedad es
@given(st.integers(min_value=1, max_value=100))
def test_aproxE(n: int) -> None:
    r = aproxE1(n)
    assert aproxE2(n) == r
    assert aproxE3(n) == r
    assert aproxE4(n) == r
    assert aproxE5(n) == r
    assert aproxE6(n) == r

# La comprobación es
#   src> poetry run pytest -q aproximacion_del_numero_e.py
#   1 passed in 0.60s

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

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

# La comparación es
#   >>> tiempo('aproxE1(20000)')
#   0.00 segundos
#   >>> tiempo('aproxE2(20000)')
#   0.43 segundos
#   >>> tiempo('aproxE3(20000)')
#   0.60 segundos
#   >>> tiempo('aproxE4(20000)')
#   1.23 segundos
#   >>> tiempo('aproxE5(20000)')
#   0.00 segundos

```

```

# >>> tiempo('aproxE6(20000)')
# 0.00 segundos

# >>> tiempo('aproxE1(10**7)')
# 1.18 segundos
# >>> tiempo('aproxE5(10**7)')
# 1.48 segundos
# >>> tiempo('aproxE6(10**7)')
# 1.43 segundos

# 1ª definición de errorAproxE
# =====

# naturales es el generador de los números naturales positivos, Por
# ejemplo,
# >>> list(islice(naturales(), 5))
# [1, 2, 3, 4, 5]
def naturales() -> Iterator[int]:
    i = 1
    while True:
        yield i
        i += 1

def errorAproxE1(x: float) -> int:
    return list(islice((n for n in naturales()
                        if abs(e - (1 + 1/n)**n) < x), 1))[0]

# # 2ª definición de errorAproxE
# # =====

def errorAproxE2(x: float) -> int:
    def aux(n: int) -> int:
        if abs(e - (1 + 1/n)**n) < x:
            return n
        return aux(n + 1)

    return aux(1)

# 3ª definición de errorAproxE
# =====

```

```

def errorAproxE3(x: float) -> int:
    return list(islice(dropwhile(lambda n: abs(e - (1 + 1/n)**n) >= x,
                                   naturales()),
                1))[0]

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

@given(st.integers(min_value=1, max_value=100))
def test_errorAproxE(n: int) -> None:
    r = errorAproxE1(n)
    assert errorAproxE2(n) == r
    assert errorAproxE3(n) == r

# La comprobación es
#   src> poetry run pytest -q aproximacion_del_numero_e.py
#   2 passed in 0.60s

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

# La comparación es
#   >>> tiempo('errorAproxE1(0.0001)')
#   0.00 segundos
#   >>> tiempo('errorAproxE2(0.0001)')
#   0.00 segundos
#   >>> tiempo('errorAproxE3(0.0001)')
#   0.00 segundos
#
#   >>> tiempo('errorAproxE1(0.0000001)')
#   2.48 segundos
#   >>> tiempo('errorAproxE3(0.0000001)')
#   2.61 segundos

```

2.22. Aproximación al límite de $\sin(x)/x$ cuando x tiende a cero

En Haskell

```

-- -----
-- El límite 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 las funciones
--   aproxLimSeno :: Int -> [Double]
--   errorLimSeno :: Double -> Int
-- tales que
-- + (aproxLimSeno n) es la lista cuyos elementos son los n primeros
--   términos de la sucesión  $\sin(1/m)/(1/m)$ . Por ejemplo,
--   aproxLimSeno 1 == [0.8414709848078965]
--   aproxLimSeno 2 == [0.8414709848078965, 0.958851077208406]
-- + (errorLimSeno x) es el menor número de términos de la sucesión
--    $\sin(1/m)/(1/m)$  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
-- -----

module Limite_del_seno where

import Test.QuickCheck

-- 1ª definición de aproxLimSeno
-- =====

aproxLimSeno1 :: Int -> [Double]
aproxLimSeno1 k = [sin(1/n)/(1/n) | n <- [1..k']]
  where k' = fromIntegral k

-- 2ª definición de aproxLimSeno
-- =====

```

```

aproxLimSeno2 :: Int -> [Double]
aproxLimSeno2 0 = []
aproxLimSeno2 n = aproxLimSeno2 (n-1) ++ [sin(1/n')/(1/n')]
  where n' = fromIntegral n

```

```

-- 3ª definición de aproxLimSeno
-- =====

```

```

aproxLimSeno3 :: Int -> [Double]
aproxLimSeno3 = reverse . aux . fromIntegral
  where aux 0 = []
        aux n = sin(1/n)/(1/n) : aux (n-1)

```

```

-- 4ª definición de aproxLimSeno
-- =====

```

```

aproxLimSeno4 :: Int -> [Double]
aproxLimSeno4 k = aux [] (fromIntegral k)
  where aux xs 0 = xs
        aux xs n = aux (sin(1/n)/(1/n) : xs) (n-1)

```

```

-- 5ª definición de aproxLimSeno
-- =====

```

```

aproxLimSeno5 :: Int -> [Double]
aproxLimSeno5 k = map (\ n -> sin(1/n)/(1/n)) [1..k']
  where k' = fromIntegral k

```

```

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

```

```

-- La propiedad es
prop_aproxLimSeno :: Positive Int -> Bool
prop_aproxLimSeno (Positive k) =
  all (== aproxLimSeno1 k)
    [aproxLimSeno2 k,
     aproxLimSeno3 k,
     aproxLimSeno4 k,
     aproxLimSeno5 k]

```

```

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

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

-- La comparación es
--   λ> last (aproxLimSeno1 (2*10^4))
--   0.9999999995833334
--   (0.01 secs, 5,415,816 bytes)
--   λ> last (aproxLimSeno2 (2*10^4))
--   0.9999999995833334
--   (4.48 secs, 17,514,768,064 bytes)
--   λ> last (aproxLimSeno3 (2*10^4))
--   0.9999999995833334
--   (0.02 secs, 9,530,120 bytes)
--   λ> last (aproxLimSeno4 (2*10^4))
--   0.9999999995833334
--   (0.02 secs, 9,529,968 bytes)
--   λ> last (aproxLimSeno5 (2*10^4))
--   0.9999999995833334
--   (0.01 secs, 4,889,720 bytes)
--
--   λ> last (aproxLimSeno1 (2*10^6))
--   0.9999999999999583
--   (0.46 secs, 480,569,808 bytes)
--   λ> last (aproxLimSeno3 (2*10^6))
--   0.9999999999999583
--   (1.96 secs, 896,569,992 bytes)
--   λ> last (aproxLimSeno4 (2*10^6))
--   0.9999999999999583
--   (1.93 secs, 896,570,048 bytes)
--   λ> last (aproxLimSeno5 (2*10^6))
--   0.9999999999999583
--   (0.05 secs, 432,569,800 bytes)
--
--   λ> last (aproxLimSeno1 (10^7))
--   0.9999999999999983
--   (2.26 secs, 2,400,569,760 bytes)

```



```

--      λ> last (aproxLimSeno5 (10^7))
--      0.99999999999999983
--      (0.24 secs, 2,160,569,752 bytes)

-- 1ª definición de errorLimSeno
-- =====

errorLimSeno1 :: Double -> Int
errorLimSeno1 x =
    round (head [m | m <- [1..], abs (1 - sin(1/m)/(1/m)) < x])

-- 2ª definición de errorLimSeno
-- =====

errorLimSeno2 :: Double -> Int
errorLimSeno2 x = aux 1
    where aux n | abs (1 - sin(1/n)/(1/n)) < x = round n
                | otherwise                    = aux (n+1)

-- 3ª definición de errorLimSeno
-- =====

errorLimSeno3 :: Double -> Int
errorLimSeno3 x =
    round (head (dropWhile (\ n -> abs (1 - sin(1/n)/(1/n)) >= x) [1..]))

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

-- La propiedad es
prop_errorLimSeno :: Positive Double -> Bool
prop_errorLimSeno (Positive x) =
    all (== errorLimSeno1 x)
        [errorLimSeno2 x,
         errorLimSeno3 x]

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

```

```
-- Comparación de eficiencia de errorLimSeno
-- =====

-- La comparación es
--     λ> errorLimSeno1 (10**(-12))
--     408230
--     (0.41 secs, 206,300,808 bytes)
--     λ> errorLimSeno2 (10**(-12))
--     408230
--     (0.46 secs, 225,895,672 bytes)
--     λ> errorLimSeno3 (10**(-12))
--     408230
--     (0.37 secs, 186,705,688 bytes)
```

En Python

```
# -----
# El límite 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 las funciones
#     aproxLimSeno : (int) -> list[float]
#     errorLimSeno : (float) -> int
# tales que
# + aproxLimSeno( $n$ ) es la lista cuyos elementos son los  $n$  primeros
#   términos de la sucesión  $\sin(1/m)/(1/m)$ . Por ejemplo,
#     aproxLimSeno(1) == [0.8414709848078965]
#     aproxLimSeno(2) == [0.8414709848078965, 0.958851077208406]
# + errorLimSeno( $x$ ) es el menor número de términos de la sucesión
#    $\sin(1/m)/(1/m)$  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
# -----
```

```
from itertools import dropwhile, islice
from math import sin
from sys import setrecursionlimit
```

```
from timeit import Timer, default_timer
from typing import Iterator

from hypothesis import given
from hypothesis import strategies as st

setrecursionlimit(10**6)

# 1ª definición de aproxLimSeno
# =====

def aproxLimSeno1(k: int) -> list[float]:
    return [sin(1/n)/(1/n) for n in range(1, k + 1)]

# 2ª definición de aproxLimSeno
# =====

def aproxLimSeno2(n: int) -> list[float]:
    if n == 0:
        return []
    return aproxLimSeno2(n - 1) + [sin(1/n)/(1/n)]

# 3ª definición de aproxLimSeno
# =====

def aproxLimSeno3(n: int) -> list[float]:
    def aux(n: int) -> list[float]:
        if n == 0:
            return []
        return [sin(1/n)/(1/n)] + aux(n - 1)

    return list(reversed(aux(n)))

# 4ª definición de aproxLimSeno
# =====

def aproxLimSeno4(n: int) -> list[float]:
    def aux(xs: list[float], n: int) -> list[float]:
        if n == 0:
            return xs
```

```

        return aux([sin(1/n)/(1/n)] + xs, n - 1)

    return aux([], n)

# 5ª definición de aproxLimSeno
# =====

def aproxLimSeno5(n: int) -> list[float]:
    return list(map((lambda k: sin(1/k)/(1/k)), range(1, n+1)))

# 6ª definición de aproxLimSeno
# =====

def aproxLimSeno6(n: int) -> list[float]:
    r = []
    for k in range(1, n+1):
        r.append(sin(1/k)/(1/k))
    return r

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

# La propiedad es
@given(st.integers(min_value=1, max_value=100))
def test_aproxLimSeno(n: int) -> None:
    r = aproxLimSeno1(n)
    assert aproxLimSeno2(n) == r
    assert aproxLimSeno3(n) == r
    assert aproxLimSeno4(n) == r
    assert aproxLimSeno5(n) == r
    assert aproxLimSeno6(n) == r

# La comprobación es
# src> poetry run pytest -q limite_del_seno.py
# 1 passed in 0.60s

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

def tiempo(e: str) -> None:

```

```

"""Tiempo (en segundos) de evaluar la expresión e."""
t = Timer(e, "", default_timer, globals()).timeit(1)
print(f"{t:0.2f} segundos")

# La comparación es
# >>> tiempo('aproxLimSeno1(3*10**5)')
# 0.03 segundos
# >>> tiempo('aproxLimSeno2(3*10**5)')
# Process Python violación de segmento (core dumped)
# >>> tiempo('aproxLimSeno3(3*10**5)')
# Process Python violación de segmento (core dumped)
# >>> tiempo('aproxLimSeno4(3*10**5)')
# Process Python violación de segmento (core dumped)
# >>> tiempo('aproxLimSeno5(3*10**5)')
# 0.04 segundos
# >>> tiempo('aproxLimSeno6(3*10**5)')
# 0.07 segundos
#
# >>> tiempo('aproxLimSeno1(10**7)')
# 1.29 segundos
# >>> tiempo('aproxLimSeno5(10**7)')
# 1.40 segundos
# >>> tiempo('aproxLimSeno6(10**7)')
# 1.45 segundos

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

# naturales es el generador de los números naturales positivos, Por
# ejemplo,
# >>> list(islice(naturales(), 5))
# [1, 2, 3, 4, 5]
def naturales() -> Iterator[int]:
    i = 1
    while True:
        yield i
        i += 1

def errorLimSeno1(x: float) -> int:
    return list(islice((n for n in naturales())

```

```

        if abs(1 - sin(1/n)/(1/n)) < x), 1))[0]

# 2ª definición de errorLimSeno
# =====

def errorLimSeno2(x: float) -> int:
    def aux(n: int) -> int:
        if abs(1 - sin(1/n)/(1/n)) < x:
            return n
        return aux(n + 1)

    return aux(1)

# 3ª definición de errorLimSeno
# =====

def errorLimSeno3(x: float) -> int:
    return list(islice(dropwhile(lambda n: abs(1 - sin(1/n)/(1/n)) >= x,
                                     naturales()),
                  1))[0]

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

@given(st.integers(min_value=1, max_value=100))
def test_errorLimSeno(n: int) -> None:
    r = errorLimSeno1(n)
    assert errorLimSeno2(n) == r
    assert errorLimSeno3(n) == r

# La comprobación es
#   src> poetry run pytest -q limite_del_seno.py
#   2 passed in 0.60s

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

# La comparación es
#   >>> tiempo('errorLimSeno1(10**(-12))')
```

```
# 0.07 segundos
# >>> tiempo('errorLimSeno2(10**(-12))')
# Process Python violación de segmento (core dumped)
# >>> tiempo('errorLimSeno3(10**(-12))')
# 0.10 segundos
```

2.23. Cálculo del número π mediante la fórmula de Leibniz

En Haskell

```
-- -----
-- El número  $\pi$  puede calcularse con la [fórmula de
-- Leibniz](https://bit.ly/3ERCwZd)
--  $\pi/4 = 1 - 1/3 + 1/5 - 1/7 + \dots + (-1)^n/(2n+1) + \dots$ 
--
-- Definir las funciones
--   calculaPi :: Int -> Double
--   errorPi   :: Double -> Int
-- tales que
-- + (calculaPi n) es la aproximación del número  $\pi$  calculada
--   mediante la expresión
--    $4*(1 - 1/3 + 1/5 - 1/7 + \dots + (-1)^n/(2n+1))$ 
-- Por ejemplo,
--   calculaPi 3    == 2.8952380952380956
--   calculaPi 300  == 3.1449149035588526
-- + (errorPi x) es el menor número de términos de la serie
--   necesarios para obtener pi con un error menor que x. Por ejemplo,
--   errorPi 0.1    == 9
--   errorPi 0.01   == 99
--   errorPi 0.001  == 999
-- -----
```

```
module Calculo_de_pi_mediante_la_formula_de_Leibniz where
```

```
import Test.QuickCheck
```

```
-- 1ª definición de calculaPi
-- =====
```

```

calculaPi1 :: Int -> Double
calculaPi1 k = 4 * sum [(-1)**n/(2*n+1) | n <- [0..k']]
  where k' = fromIntegral k

-- 2ª definición de calculaPi
-- =====

calculaPi2 :: Int -> Double
calculaPi2 0 = 4
calculaPi2 n = calculaPi2 (n-1) + 4*(-1)**n'/(2*n'+1)
  where n' = fromIntegral n

-- 3ª definición de calculaPi
-- =====

calculaPi3 :: Int -> Double
calculaPi3 = aux . fromIntegral
  where aux 0 = 4
        aux n = 4*(-1)**n/(2*n+1) + aux (n-1)

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

-- La propiedad es
prop_calculaPi :: Positive Int -> Bool
prop_calculaPi (Positive k) =
  all (== calculaPi1 k)
    [calculaPi2 k,
     calculaPi3 k]

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

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

-- La comparación es
-- λ> calculaPi1 (10^6)

```



```

--      3.1415936535887745
--      (1.31 secs, 609,797,408 bytes)
--      λ> calculaPi2 (10^6)
--      3.1415936535887745
--      (1.68 secs, 723,032,272 bytes)
--      λ> calculaPi3 (10^6)
--      3.1415936535887745
--      (2.22 secs, 1,099,032,608 bytes)

-- 1ª definición de errorPi
-- =====

errorPi1 :: Double -> Int
errorPi1 x =
  head [n | n <- [1..]
        , abs (pi - calculaPi1 n) < x]

-- 2ª definición de errorPi
-- =====

errorPi2 :: Double -> Int
errorPi2 x = aux 1
  where aux n | abs (pi - calculaPi1 n) < x = n
              | otherwise                  = aux (n+1)

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

-- La propiedad es
prop_errorPi :: Positive Double -> Bool
prop_errorPi (Positive x) =
  errorPi1 x == errorPi2 x

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

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

```

```
-- La comparación es
--   λ> errorPi1 0.0005
--   1999
--   (1.88 secs, 1,189,226,384 bytes)
--   λ> errorPi2 0.0005
--   1999
--   (1.87 secs, 1,213,341,096 bytes)
```

En Python

```
# -----
# El número  $\pi$  puede calcularse con la [fórmula de
# Leibniz](https://bit.ly/3ERCwZd)
#    $\pi/4 = 1 - 1/3 + 1/5 - 1/7 + \dots + (-1)**n/(2*n+1) + \dots$ 
#
# Definir las funciones
#   calculaPi : (int) -> float
#   errorPi   : (float) -> int
# tales que
# + calculaPi(n) es la aproximación del número  $\pi$  calculada
#   mediante la expresión
#    $4*(1 - 1/3 + 1/5 - 1/7 + \dots + (-1)**n/(2*n+1))$ 
# Por ejemplo,
#   calculaPi(3)      == 2.8952380952380956
#   calculaPi(300)   == 3.1449149035588526
# + errorPi(x) es el menor número de términos de la serie
#   necesarios para obtener pi con un error menor que x. Por ejemplo,
#   errorPi(0.1)      == 9
#   errorPi(0.01)     == 99
#   errorPi(0.001)    == 999
# -----

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

from hypothesis import given
```

```

from hypothesis import strategies as st

setrecursionlimit(10**6)

# 1ª definición de calculaPi
# =====

def calculaPi1(k: int) -> float:
    return 4 * sum((-1)**n/(2*n+1) for n in range(0, k+1))

# 2ª definición de calculaPi
# =====

def calculaPi2(n: int) -> float:
    if n == 0:
        return 4
    return calculaPi2(n-1) + 4*(-1)**n/(2*n+1)

# 3ª definición de calculaPi
# =====

def calculaPi3(n: int) -> float:
    r = 1
    for k in range(1, n+1):
        r = r + (-1)**k/(2*k+1)
    return 4 * r

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

# La propiedad es
@given(st.integers(min_value=1, max_value=100))
def test_calculaPi(n: int) -> None:
    r = calculaPi1(n)
    assert calculaPi2(n) == r
    assert calculaPi3(n) == r

# La comprobación es
# src> poetry run pytest -q calculo_de_pi_mediante_la_formula_de_Leibniz.py
# 1 passed in 0.14s

```

```

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

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

# La comparación es
# >>> tiempo('calculaPi1(10**6)')
# 0.37 segundos
# >>> tiempo('calculaPi2(10**6)')
# Process Python violación de segmento (core dumped)
# >>> tiempo('calculaPi3(10**6)')
# 0.39 segundos

# 1ª definición de errorPi
# =====

# naturales es el generador de los números naturales positivos, Por
# ejemplo,
# >>> list(islice(naturales(), 5))
# [1, 2, 3, 4, 5]
def naturales() -> Iterator[int]:
    i = 1
    while True:
        yield i
        i += 1

def errorPi1(x: float) -> int:
    return list(islice((n for n in naturales()
                        if abs(pi - calculaPi1(n)) < x), 1))[0]

# 2ª definición de errorPi
# =====

def errorPi2(x: float) -> int:
    def aux(n: int) -> int:
        if abs(pi - calculaPi1(n)) < x:

```

```

        return n
    return aux(n + 1)

return aux(1)

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

@given(st.integers(min_value=1, max_value=100))
def test_errorPi(n: int) -> None:
    assert errorPi1(n) == errorPi2(n)

# La comprobación es
#   src> poetry run pytest -q calculo_de_pi_mediante_la_formula_de_Leibniz.py
#   2 passed in 0.60s

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

# La comparación es
#   >>> tiempo('errorPi1(0.0005)')
#   0.63 segundos
#   >>> tiempo('errorPi2(0.0005)')
#   0.58 segundos

```

2.24. Ternas pitagóricas

En Haskell

```

-- -----
-- Una terna (x,y,z) de enteros positivos es pitagórica si  $x^2 + y^2 = z^2$  y  $x < y < z$ .
--
-- Definir, por comprensión, la función
--   pitagoricas :: Int -> [(Int,Int,Int)]
-- tal que (pitagoricas n) es la lista de todas las ternas pitagóricas
-- cuyas componentes están entre 1 y n. Por ejemplo,
--   pitagoricas 10 == [(3,4,5),(6,8,10)]
--   pitagoricas 15 == [(3,4,5),(5,12,13),(6,8,10),(9,12,15)]
-- -----

```

```
module Ternas_pitagoricas where
```

```
import Test.QuickCheck
```

```
-- 1ª solución
```

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

```
pitagoricas1 :: Int -> [(Int,Int,Int)]
pitagoricas1 n = [(x,y,z) | x <- [1..n]
                        , y <- [1..n]
                        , z <- [1..n]
                        , x^2 + y^2 == z^2
                        , x < y && y < z]
```

```
-- 2ª solución
```

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

```
pitagoricas2 :: Int -> [(Int,Int,Int)]
pitagoricas2 n = [(x,y,z) | x <- [1..n]
                        , y <- [x+1..n]
                        , z <- [ceiling (sqrt (fromIntegral (x^2+y^2)))..n]
                        , x^2 + y^2 == z^2]
```

```
-- 3ª solución
```

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

```
pitagoricas3 :: Int -> [(Int,Int,Int)]
pitagoricas3 n = [(x,y,z) | x <- [1..n]
                        , y <- [x+1..n]
                        , let z = round (sqrt (fromIntegral (x^2+y^2)))
                        , y < z
                        , z <= n
                        , x^2 + y^2 == z^2]
```

```
-- Comprobación de equivalencia
```

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

```
-- La propiedad es
```

```
prop_pitagoricas :: Positive Int -> Bool
```

```
prop_pitagoricas (Positive n) =
  all (== pitagoricas1 n)
    [pitagoricas2 n,
     pitagoricas3 n]

-- La comprobación es
--   λ> quickCheck prop_pitagoricas
--   +++ OK, passed 100 tests.

-- Comparación de eficiencia
-- =====

-- La comparación es
--   λ> length (pitagoricas1 200)
--   127
--   (12.25 secs, 12,680,320,400 bytes)
--   λ> length (pitagoricas2 200)
--   127
--   (1.61 secs, 1,679,376,824 bytes)
--   λ> length (pitagoricas3 200)
--   127
--   (0.06 secs, 55,837,072 bytes)
```

En Python

```
# -----
# Una terna (x,y,z) de enteros positivos es pitagórica si  $x^2 + y^2 = z^2$  y  $x < y < z$ .
#
# Definir, por comprensión, la función
#   pitagoricas : (int) -> list[tuple[int,int,int]]
# tal que pitagoricas(n) es la lista de todas las ternas pitagóricas
# cuyas componentes están entre 1 y n. Por ejemplo,
#   pitagoricas(10) == [(3, 4, 5), (6, 8, 10)]
#   pitagoricas(15) == [(3, 4, 5), (5, 12, 13), (6, 8, 10), (9, 12, 15)]
# -----

from math import ceil, sqrt
from timeit import Timer, default_timer
```

```

from hypothesis import given
from hypothesis import strategies as st

# 1ª solución
# =====

def pitagoricas1(n: int) -> list[tuple[int, int, int]]:
    return [(x, y, z)
            for x in range(1, n+1)
            for y in range(1, n+1)
            for z in range(1, n+1)
            if x**2 + y**2 == z**2 and x < y < z]

# 2ª solución
# =====

def pitagoricas2(n: int) -> list[tuple[int, int, int]]:
    return [(x, y, z)
            for x in range(1, n+1)
            for y in range(x+1, n+1)
            for z in range(ceil(sqrt(x**2+y**2)), n+1)
            if x**2 + y**2 == z**2]

# 3ª solución
# =====

def pitagoricas3(n: int) -> list[tuple[int, int, int]]:
    return [(x, y, z)
            for x in range(1, n+1)
            for y in range(x+1, n+1)
            for z in [ceil(sqrt(x**2+y**2))]
            if y < z <= n and x**2 + y**2 == z**2]

# Comprobación de equivalencia
# =====

# La propiedad es
@given(st.integers(min_value=1, max_value=50))
def test_pitagoricas(n: int) -> None:
    r = pitagoricas1(n)

```



```

    assert pitagoricas2(n) == r
    assert pitagoricas3(n) == r

# La comprobación es
#   src> poetry run pytest -q ternas_pitagoricas.py
#   1 passed in 1.83s

# Comparación de eficiencia
# =====

def tiempo(e: str) -> None:
    """Tiempo (en segundos) de evaluar la expresión e."""
    t = Timer(e, "", default_timer, globals()).timeit(1)
    print(f"{t:0.2f} segundos")

# La comparación es
#   >>> tiempo('pitagoricas1(200)')
#   4.76 segundos
#   >>> tiempo('pitagoricas2(200)')
#   0.69 segundos
#   >>> tiempo('pitagoricas3(200)')
#   0.02 segundos

```

2.25. Ternas pitagóricas con suma dada

En Haskell

```

-- -----
-- Una terna pitagórica es una terna de números naturales (a,b,c) tal
-- que  $a < b < c$  y  $a^2 + b^2 = c^2$ . Por ejemplo (3,4,5) es una terna pitagórica.
--
-- Definir la función
--   ternasPitagoricas :: Integer -> [(Integer,Integer,Integer)]
-- tal que (ternasPitagoricas x) es la lista de las ternas pitagóricas
-- cuya suma es x. Por ejemplo,
--   ternasPitagoricas 12      == [(3,4,5)]
--   ternasPitagoricas 60     == [(10,24,26),(15,20,25)]
--   ternasPitagoricas (10^6) == [(218750,360000,421250),(200000,375000,425000)]
-- -----

```

```
module Ternas_pitagoricas_con_suma_dada where
```

```
import Data.List (nub, sort)
```

```
import Test.QuickCheck
```

```
-- 1ª solución
```

```
--
```

```
-- =====
```

```
ternasPitagoricas1 :: Integer -> [(Integer,Integer,Integer)]
```

```
ternasPitagoricas1 x =
```

```
  [(a,b,c) | a <- [0..x],
              b <- [a+1..x],
              c <- [b+1..x],
              a^2 + b^2 == c^2,
              a+b+c == x]
```

```
-- 2ª solución
```

```
--
```

```
-- =====
```

```
ternasPitagoricas2 :: Integer -> [(Integer,Integer,Integer)]
```

```
ternasPitagoricas2 x =
```

```
  [(a,b,c) | a <- [1..x],
              b <- [a+1..x-a],
              let c = x-a-b,
              a^2+b^2 == c^2]
```

```
-- 3ª solución
```

```
--
```

```
-- =====
```

```
-- Todas las ternas pitagóricas primitivas (a,b,c) pueden representarse
-- por
```

```
--   a = m^2 - n^2, b = 2*m*n, c = m^2 + n^2,
--   con 1 <= n < m. (Ver en https://bit.ly/35UNY6L ).
```

```
ternasPitagoricas3 :: Integer -> [(Integer,Integer,Integer)]
```

```
ternasPitagoricas3 x =
```

```
  nub [(d*a,d*b,d*c) | d <- [1..x],
                       x `mod` d == 0,
                       (a,b,c) <- aux (x `div` d)]
```

```
where
```

```

    aux y = [(a,b,c) | m <- [2..limite],
                        n <- [1..m-1],
                        let [a,b] = sort [m^2 - n^2, 2*m*n],
                        let c = m^2 + n^2,
                        a+b+c == y]
    where limite = ceiling (sqrt (fromIntegral y))

-- Equivalencia de las definiciones
-- =====

-- La propiedad es
prop_ternasPitagoricas :: Positive Integer -> Bool
prop_ternasPitagoricas (Positive x) =
    all (== (ternasPitagoricas1 x))
        [ternasPitagoricas2 x,
         ternasPitagoricas3 x]

-- La comprobación es
--    λ> quickCheck prop_ternasPitagoricas
--    +++ OK, passed 100 tests.

-- Comparación de eficiencia
-- =====

-- La comparación es
--    λ> ternasPitagoricas1 200
--    [(40,75,85)]
--    (1.90 secs, 2,404,800,856 bytes)
--    λ> ternasPitagoricas2 200
--    [(40,75,85)]
--    (0.06 secs, 19,334,232 bytes)
--    λ> ternasPitagoricas3 200
--    [(40,75,85)]
--    (0.01 secs, 994,224 bytes)
--
--    λ> ternasPitagoricas2 3000
--    [(500,1200,1300),(600,1125,1275),(750,1000,1250)]
--    (4.41 secs, 4,354,148,136 bytes)
--    λ> ternasPitagoricas3 3000
--    [(500,1200,1300),(600,1125,1275),(750,1000,1250)]

```

```
-- (0.05 secs, 17,110,360 bytes)
```

En Python

```
# -----
# Una terna pitagórica es una terna de números naturales (a,b,c) tal
# que  $a < b < c$  y  $a^2 + b^2 = c^2$ . Por ejemplo (3,4,5) es una terna pitagórica.
#
# Definir la función
#   ternasPitagoricas : (int) -> list[tuple[int, int, int]]
# tal que ternasPitagoricas(x) es la lista de las ternas pitagóricas
# cuya suma es x. Por ejemplo,
#   ternasPitagoricas(12)    == [(3, 4, 5)]
#   ternasPitagoricas(60)    == [(10, 24, 26), (15, 20, 25)]
#   ternasPitagoricas(10**6) == [(218750, 360000, 421250),
#                                (200000, 375000, 425000)]
# -----
```

```
from math import ceil, sqrt
from timeit import Timer, default_timer
```

```
from hypothesis import given
from hypothesis import strategies as st
```

```
# 1ª solución                                     --
# =====
```

```
def ternasPitagoricas1(x: int) -> list[tuple[int, int, int]]:
    return [(a, b, c)
            for a in range(0, x+1)
            for b in range(a+1, x+1)
            for c in range(b+1, x+1)
            if a**2 + b**2 == c**2 and a + b + c == x]
```

```
# 2ª solución                                     --
# =====
```

```
def ternasPitagoricas2(x: int) -> list[tuple[int, int, int]]:
    return [(a, b, c)
            for a in range(1, x+1)
```

```

        for b in range(a+1, x-a+1)
        for c in [x - a - b]
        if a**2 + b**2 == c**2]

# 3ª solución
# =====

# Todas las ternas pitagóricas primitivas (a,b,c) pueden representarse
# por
#  $a = m^2 - n^2$ ,  $b = 2*m*n$ ,  $c = m^2 + n^2$ ,
# con  $1 \leq n < m$ . (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 list(set(((d*a, d*b, d*c)
                     for d in range(1, x+1)
                     for (a, b, c) in aux(x // d)
                     if x % d == 0))))

# Comprobación de equivalencia
# =====

# La propiedad es
@given(st.integers(min_value=1, max_value=50))
def test_ternasPitagoricas(n: int) -> None:
    r = set(ternasPitagoricas1(n))
    assert set(ternasPitagoricas2(n)) == r
    assert set(ternasPitagoricas3(n)) == r

# La comprobación es
# src> poetry run pytest -q ternas_pitagoricas_con_suma_dada.py
# 1 passed in 0.35s

```

```

# Comparación de eficiencia
# =====

def tiempo(e: str) -> None:
    """Tiempo (en segundos) de evaluar la expresión e."""
    t = Timer(e, "", default_timer, globals()).timeit(1)
    print(f"{t:0.2f} segundos")

# La comparación es
# >>> tiempo('ternasPitagoricas1(300)')
# 2.83 segundos
# >>> tiempo('ternasPitagoricas2(300)')
# 0.01 segundos
# >>> tiempo('ternasPitagoricas3(300)')
# 0.00 segundos
#
# >>> tiempo('ternasPitagoricas2(3000)')
# 1.48 segundos
# >>> tiempo('ternasPitagoricas3(3000)')
# 0.02 segundos

```

2.26. Producto escalar

En Haskell

```

-- -----
-- El producto escalar de dos listas de enteros xs y ys de longitud n
-- viene dado por la suma de los productos de los elementos
-- correspondientes.
--
-- Definir la función
--   productoEscalar :: [Integer] -> [Integer] -> Integer
-- tal que (productoEscalar xs ys) es el producto escalar de las listas
-- xs e ys. Por ejemplo,
--   productoEscalar [1,2,3] [4,5,6] == 32
-- -----

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

```

```

module Producto_escalar where

import Test.QuickCheck (quickCheck)

-- 1ª solución
-- =====

productoEscalar1 :: [Integer] -> [Integer] -> Integer
productoEscalar1 xs ys = sum [x*y | (x,y) <- zip xs ys]

-- 2ª solución
-- =====

productoEscalar2 :: [Integer] -> [Integer] -> Integer
productoEscalar2 xs ys = sum (zipWith (*) xs ys)

-- 3ª solución
-- =====

productoEscalar3 :: [Integer] -> [Integer] -> Integer
productoEscalar3 = (sum .) . zipWith (*)

-- 4ª solución
-- =====

productoEscalar4 :: [Integer] -> [Integer] -> Integer
productoEscalar4 [] _ = 0
productoEscalar4 _ [] = 0
productoEscalar4 (x:xs) (y:ys) = x*y + productoEscalar4 xs ys

-- 5ª solución
-- =====

productoEscalar5 :: [Integer] -> [Integer] -> Integer
productoEscalar5 (x:xs) (y:ys) = x*y + productoEscalar5 xs ys
productoEscalar5 _ _ = 0

-- Comprobación de equivalencia
-- =====

```

```

-- La propiedad es
prop_productoEscalar :: [Integer] -> [Integer] -> Bool
prop_productoEscalar xs ys =
  all (== productoEscalar1 xs ys)
    [productoEscalar2 xs ys,
     productoEscalar3 xs ys,
     productoEscalar4 xs ys,
     productoEscalar5 xs ys]

-- La comprobación es
-- λ> quickCheck prop_productoEscalar
-- +++ OK, passed 100 tests.

-- Comparación de eficiencia
-- =====

-- La comparación es
-- λ> productoEscalar1 (replicate (2*10^6) 1) (replicate (2*10^6) 1)
-- 20000000
-- (1.37 secs, 803,827,520 bytes)
-- λ> productoEscalar2 (replicate (2*10^6) 1) (replicate (2*10^6) 1)
-- 20000000
-- (0.69 secs, 611,008,272 bytes)
-- λ> productoEscalar3 (replicate (2*10^6) 1) (replicate (2*10^6) 1)
-- 20000000
-- (0.69 secs, 611,008,536 bytes)
-- λ> productoEscalar4 (replicate (2*10^6) 1) (replicate (2*10^6) 1)
-- 20000000
-- (1.64 secs, 742,290,272 bytes)
-- λ> productoEscalar5 (replicate (2*10^6) 1) (replicate (2*10^6) 1)
-- 20000000
-- (1.63 secs, 742,290,064 bytes)
-- λ> productoEscalar6 (replicate (2*10^6) 1) (replicate (2*10^6) 1)
-- 20000000
-- (0.32 secs, 835,679,200 bytes)
--
-- λ> productoEscalar2 (replicate (6*10^6) 1) (replicate (6*10^6) 1)
-- 60000000
-- (1.90 secs, 1,831,960,336 bytes)
-- λ> productoEscalar3 (replicate (6*10^6) 1) (replicate (6*10^6) 1)

```



```
--      6000000
--      (1.87 secs, 1,831,960,600 bytes)
--      λ> productoEscalar6 (replicate (6*10^6) 1) (replicate (6*10^6) 1)
--      6000000
--      (0.78 secs, 2,573,005,952 bytes)
```

En Python

```
# -----
# El producto escalar de dos listas de enteros xs y ys de longitud n
# viene dado por la suma de los productos de los elementos
# correspondientes.
#
# Definir la función
#   productoEscalar : (list[int], list[int]) -> int
# tal que productoEscalar(xs, ys) es el producto escalar de las listas
# xs e ys. Por ejemplo,
#   productoEscalar([1, 2, 3], [4, 5, 6]) == 32
# -----
```

```
from operator import mul
from sys import setrecursionlimit
from timeit import Timer, default_timer
```

```
from hypothesis import given
from hypothesis import strategies as st
from numpy import dot
```

```
setrecursionlimit(10**6)
```

```
# 1ª solución
# =====
```

```
def productoEscalar1(xs: list[int], ys: list[int]) -> int:
    return sum(x * y for (x, y) in zip(xs, ys))
```

```
# 2ª solución
# =====
```

```
def productoEscalar2(xs: list[int], ys: list[int]) -> int:
```

```

    return sum(map(mul, xs, ys))

# 3ª solución
# =====

def productoEscalar3(xs: list[int], ys: list[int]) -> int:
    if xs and ys:
        return xs[0] * ys[0] + productoEscalar3(xs[1:], ys[1:])
    return 0

# 4ª solución
# =====

def productoEscalar4(xs: list[int], ys: list[int]) -> int:
    return dot(xs, ys)

# Comprobación de equivalencia
# =====

# La propiedad es
@given(st.lists(st.integers(min_value=1, max_value=100)),
       st.lists(st.integers(min_value=1, max_value=100)))
def test_productoEscalar(xs: list[int], ys: list[int]) -> None:
    r = productoEscalar1(xs, ys)
    assert productoEscalar2(xs, ys) == r
    assert productoEscalar3(xs, ys) == r
    n = min(len(xs), len(ys))
    xs1 = xs[:n]
    ys1 = ys[:n]
    assert productoEscalar4(xs1, ys1) == r

# La comprobación es
#   src> poetry run pytest -q producto_escalar.py
#   1 passed in 0.37s

# Comparación de eficiencia
# =====

def tiempo(e: str) -> None:
    """Tiempo (en segundos) de evaluar la expresión e."""

```

```

t = Timer(e, "", default_timer, globals()).timeit(1)
print(f"{t:0.2f} segundos")

# La comparación es
# >>> tiempo('productoEscalar1([1]*(10**4), [1]*(10**4))')
# 0.00 segundos
# >>> tiempo('productoEscalar3([1]*(10**4), [1]*(10**4))')
# 0.55 segundos
#
# >>> tiempo('productoEscalar1([1]*(10**7), [1]*(10**7))')
# 0.60 segundos
# >>> tiempo('productoEscalar2([1]*(10**7), [1]*(10**7))')
# 0.26 segundos
# >>> tiempo('productoEscalar4([1]*(10**7), [1]*(10**7))')
# 1.73 segundos

```

2.27. Representación densa de polinomios

En Haskell

```

-----
-- Los polinomios pueden representarse de forma dispersa o densa. Por
-- ejemplo, el polinomio  $6x^4 - 5x^2 + 4x - 7$  se puede representar de forma
-- dispersa por  $[6, 0, -5, 4, -7]$  y de forma densa por  $[(4, 6), (2, -5), (1, 4), (0, -7)]$ .
--
-- Definir la función
--   densa :: [Int] -> [(Int, Int)]
-- tal que (densa xs) es la representación densa del polinomio cuya
-- representación dispersa es xs. Por ejemplo,
--   densa [6, 0, -5, 4, -7] == [(4, 6), (2, -5), (1, 4), (0, -7)]
--   densa [6, 0, 0, 3, 0, 4] == [(5, 6), (2, 3), (0, 4)]
--   densa [0]                == [(0, 0)]
-----

{-# OPTIONS_GHC -fno-warn-incomplete-patterns #-}

module Representacion_densa_de_polinomios where

import Test.QuickCheck

```

```

-- 1ª solución
-- =====

densa1 :: [Int] -> [(Int,Int)]
densa1 xs =
  [(x,y) | (x,y) <- zip [n-1,n-2..1] xs, y /= 0]
  ++ [(0, last xs)]
  where n = length xs

-- 2ª solución
-- =====

densa2 :: [Int] -> [(Int,Int)]
densa2 xs =
  filter (\ (_,y) -> y /= 0) (zip [n-1,n-2..1] xs)
  ++ [(0, last xs)]
  where n = length xs

-- 3ª solución
-- =====

densa3 :: [Int] -> [(Int,Int)]
densa3 xs = filter ((/= 0) . snd) (zip [n-1,n-2..1] xs)
  ++ [(0, last xs)]
  where n = length xs

-- 4ª solución
-- =====

densa4 :: [Int] -> [(Int,Int)]
densa4 xs = aux xs (length xs - 1)
  where aux [y] 0 = [(0, y)]
        aux (y:ys) n | y == 0    = aux ys (n-1)
                      | otherwise = (n,y) : aux ys (n-1)

-- Comprobación de equivalencia
-- =====

-- La propiedad es
prop_densa :: NonEmptyList Int -> Bool

```

```
prop_densa (NonEmpty xs) =
  all (== densa1 xs)
    [densa2 xs,
     densa3 xs,
     densa4 xs]

-- La comprobación es
--   λ> quickCheck prop_densa
--   +++ OK, passed 100 tests.

-- Comparación de eficiencia
-- =====

-- La comparación es
--   λ> last (densa1 [1..2*10^6])
--   (0,2000000)
--   (0.95 secs, 880,569,400 bytes)
--   λ> last (densa2 [1..2*10^6])
--   (0,2000000)
--   (0.52 secs, 800,569,432 bytes)
--   λ> last (densa3 [1..2*10^6])
--   (0,2000000)
--   (0.53 secs, 752,569,552 bytes)
--   λ> last (densa4 [1..2*10^6])
--   (0,2000000)
--   (3.05 secs, 1,267,842,032 bytes)
--
--   λ> last (densa1 [1..10^7])
--   (0,10000000)
--   (5.43 secs, 4,400,570,128 bytes)
--   λ> last (densa2 [1..10^7])
--   (0,10000000)
--   (3.03 secs, 4,000,570,160 bytes)
--   λ> last (densa3 [1..10^7])
--   (0,10000000)
--   (2.34 secs, 3,760,570,280 bytes)
```

En Python

```
# -----
# Los polinomios pueden representarse de forma dispersa o densa. Por
# ejemplo, el polinomio  $6x^4-5x^2+4x-7$  se puede representar de forma
# dispersa por  $[6,0,-5,4,-7]$  y de forma densa por
#  $[(4,6),(2,-5),(1,4),(0,-7)]$ .
#
# Definir la función
#   densa : (list[int]) -> list[tuple[int, int]]
# tal que densa(xs) es la representación densa del polinomio cuya
# representación dispersa es xs. Por ejemplo,
#   densa([6, 0, -5, 4, -7]) == [(4, 6), (2, -5), (1, 4), (0, -7)]
#   densa([6, 0, 0, 3, 0, 4]) == [(5, 6), (2, 3), (0, 4)]
#   densa([0]) == [(0, 0)]
# -----
```

```
from sys import setrecursionlimit
from timeit import Timer, default_timer
```

```
from hypothesis import given
from hypothesis import strategies as st
```

```
setrecursionlimit(10**6)
```

```
# 1ª solución
# =====
```

```
def densa1(xs: list[int]) -> list[tuple[int, int]]:
    n = len(xs)
    return [(x, y)
            for (x, y) in zip(range(n-1, 0, -1), xs)
            if y != 0] + [(0, xs[-1])]
```

```
# 2ª solución
# =====
```

```
def densa2(xs: list[int]) -> list[tuple[int, int]]:
    n = len(xs)
    return list(filter(lambda p: p[1] != 0,
                      zip(range(n-1, 0, -1), xs))) + [(0, xs[-1])]
```

```

# 3ª solución
# =====

def densa3(xs: list[int]) -> list[tuple[int, int]]:
    def aux(ys: list[int], n: int) -> list[tuple[int, int]]:
        if n == 0:
            return [(0, ys[0])]
        if ys[0] == 0:
            return aux(ys[1:], n-1)
        return [(n, ys[0])] + aux(ys[1:], n-1)

    return aux(xs, len(xs) - 1)

# Comprobación de equivalencia
# =====

# La propiedad es
@given(st.lists(st.integers(), min_size=1))
def test_densa(xs: list[int]) -> None:
    r = densa1(xs)
    assert densa2(xs) == r
    assert densa3(xs) == r

# La comprobación es
# src> poetry run pytest -q representacion_densa_de_polinomios.py
# 1 passed in 0.27s

# Comparación de eficiencia
# =====

def tiempo(e: str) -> None:
    """Tiempo (en segundos) de evaluar la expresión e."""
    t = Timer(e, "", default_timer, globals()).timeit(1)
    print(f"{t:0.2f} segundos")

# La comparación es
# >>> tiempo('densa1(range(1, 10**4))')
# 0.00 segundos
# >>> tiempo('densa2(range(1, 10**4))')
```

```
# 0.00 segundos
# >>> tiempo('densa3(range(1, 10**4))')
# 0.25 segundos
#
# >>> tiempo('densa1(range(1, 10**7))')
# 1.87 segundos
# >>> tiempo('densa2(range(1, 10**7))')
# 2.15 segundos
```

2.28. Base de datos de actividades.

En Haskell

```
-- -----
-- Las bases de datos sobre actividades de personas pueden representarse
-- mediante listas de elementos de la forma (a,b,c,d), donde a es el
-- nombre de la persona, b su actividad, c su fecha de nacimiento y d la
-- de su fallecimiento. Un ejemplo es la siguiente que usaremos a lo
-- largo de este ejercicio,
--
-- personas :: [(String,String,Int,Int)]
-- personas = [("Cervantes","Literatura",1547,1616),
--             ("Velazquez","Pintura",1599,1660),
--             ("Picasso","Pintura",1881,1973),
--             ("Beethoven","Musica",1770,1823),
--             ("Poincare","Ciencia",1854,1912),
--             ("Quevedo","Literatura",1580,1654),
--             ("Goya","Pintura",1746,1828),
--             ("Einstein","Ciencia",1879,1955),
--             ("Mozart","Musica",1756,1791),
--             ("Botticelli","Pintura",1445,1510),
--             ("Borromini","Arquitectura",1599,1667),
--             ("Bach","Musica",1685,1750)]
--
-- Definir las funciones
--
-- nombres  :: [(String,String,Int,Int)] -> [String]
-- musicos  :: [(String,String,Int,Int)] -> [String]
-- seleccion :: [(String,String,Int,Int)] -> String -> [String]
-- musicos'  :: [(String,String,Int,Int)] -> [String]
-- vivas     :: [(String,String,Int,Int)] -> Int -> [String]
-- tales que
```



```

-- + (nombres bd) es la lista de los nombres de las personas de la base
--   de datos bd. Por ejemplo,
--   λ> nombres personas
--   ["Cervantes","Velazquez","Picasso","Beethoven","Poincare",
--    "Quevedo","Goya","Einstein","Mozart","Botticelli","Borromini",
--    "Bach"]
-- + (musicos bd) es la lista de los nombres de los músicos de la base
--   de datos bd. Por ejemplo,
--   musicos personas == ["Beethoven","Mozart","Bach"]
-- + (seleccion bd m) es la lista de los nombres de las personas de la
--   base de datos bd cuya actividad es m. Por ejemplo,
--   λ> seleccion personas "Pintura"
--   ["Velazquez","Picasso","Goya","Botticelli"]
--   λ> seleccion personas "Musica"
--   ["Beethoven","Mozart","Bach"]
-- + (musicos' bd) es la lista de los nombres de los músicos de la base
--   de datos bd. Por ejemplo,
--   musicos' personas == ["Beethoven","Mozart","Bach"]
-- + (vivas bd a) es la lista de los nombres de las personas de la base
--   de datos bd que estaban vivas en el año a. Por ejemplo,
--   λ> vivas personas 1600
--   ["Cervantes","Velazquez","Quevedo","Borromini"]
-- -----

```

module Base_de_dato_de_actividades **where**

```

personas :: [(String,String,Int,Int)]
personas = [ ("Cervantes","Literatura",1547,1616),
              ("Velazquez","Pintura",1599,1660),
              ("Picasso","Pintura",1881,1973),
              ("Beethoven","Musica",1770,1823),
              ("Poincare","Ciencia",1854,1912),
              ("Quevedo","Literatura",1580,1654),
              ("Goya","Pintura",1746,1828),
              ("Einstein","Ciencia",1879,1955),
              ("Mozart","Musica",1756,1791),
              ("Botticelli","Pintura",1445,1510),
              ("Borromini","Arquitectura",1599,1667),
              ("Bach","Musica",1685,1750)]

```

```

nombres :: [(String,String,Int,Int)] -> [String]
nombres bd = [x | (x,_,_,_) <- bd]

musicos :: [(String,String,Int,Int)] -> [String]
musicos bd = [x | (x,"Musica",_,_) <- bd]

seleccion :: [(String,String,Int,Int)] -> String -> [String]
seleccion bd m = [ x | (x,m',_,_) <- bd, m == m' ]

musicos' :: [(String,String,Int,Int)] -> [String]
musicos' bd = seleccion bd "Musica"

vivas :: [(String,String,Int,Int)] -> Int -> [String]
vivas bd a = [x | (x,_,a1,a2) <- bd, a1 <= a, a <= a2]

```

En Python

```

# -----
# Las bases de datos sobre actividades de personas pueden representarse
# mediante listas de elementos de la forma (a,b,c,d), donde a es el
# nombre de la persona, b su actividad, c su fecha de nacimiento y d la
# de su fallecimiento. Un ejemplo es la siguiente que usaremos a lo
# largo de este ejercicio,
# BD = list[tuple[str, str, int, int]]
#
# personas: BD = [
#     ("Cervantes", "Literatura", 1547, 1616),
#     ("Velazquez", "Pintura", 1599, 1660),
#     ("Picasso", "Pintura", 1881, 1973),
#     ("Beethoven", "Musica", 1770, 1823),
#     ("Poincare", "Ciencia", 1854, 1912),
#     ("Quevedo", "Literatura", 1580, 1654),
#     ("Goya", "Pintura", 1746, 1828),
#     ("Einstein", "Ciencia", 1879, 1955),
#     ("Mozart", "Musica", 1756, 1791),
#     ("Botticelli", "Pintura", 1445, 1510),
#     ("Borromini", "Arquitectura", 1599, 1667),
#     ("Bach", "Musica", 1685, 1750)]
#
# Definir las funciones

```

```

#   nombres   : (BD) -> list[str]
#   musicos   : (BD) -> list[str]
#   seleccion : (BD, str) -> list[str]
#   musicos2  : (BD) -> list[str]
#   vivas     : (BD, int) -> list[str]
# tales que
# + nombres(bd) es la lista de los nombres de las personas de la- base
#   de datos bd. Por ejemplo,
#   >>> nombres(personas)
#   ['Cervantes', 'Velazquez', 'Picasso', 'Beethoven', 'Poincare',
#    'Quevedo', 'Goya', 'Einstein', 'Mozart', 'Botticelli', 'Borromini',
#    'Bach']
# + musicos(bd) es la lista de los nombres de los músicos de la base
#   de datos bd. Por ejemplo,
#   musicos(personas) == ['Beethoven', 'Mozart', 'Bach']
# + seleccion(bd, m) es la lista de los nombres de las personas de la
#   base de datos bd cuya actividad es m. Por ejemplo,
#   >>> seleccion(personas, 'Pintura')
#   ['Velazquez', 'Picasso', 'Goya', 'Botticelli']
#   >>> seleccion(personas, 'Musica')
#   ['Beethoven', 'Mozart', 'Bach']
# + musicos2(bd) es la lista de los nombres de los músicos de la base
#   de datos bd. Por ejemplo,
#   musicos2(personas) == ['Beethoven', 'Mozart', 'Bach']
# + vivas(bd, a) es la lista de los nombres de las personas de la base
#   de datos bd que estaban vivas en el año a. Por ejemplo,
#   >>> vivas(personas, 1600)
#   ['Cervantes', 'Velazquez', 'Quevedo', 'Borromini']
# -----

```

```
BD = list[tuple[str, str, int, int]]
```

```

personas: BD = [
    ("Cervantes", "Literatura", 1547, 1616),
    ("Velazquez", "Pintura", 1599, 1660),
    ("Picasso", "Pintura", 1881, 1973),
    ("Beethoven", "Musica", 1770, 1823),
    ("Poincare", "Ciencia", 1854, 1912),
    ("Quevedo", "Literatura", 1580, 1654),
    ("Goya", "Pintura", 1746, 1828),

```

```
("Einstein", "Ciencia", 1879, 1955),  
("Mozart", "Musica", 1756, 1791),  
("Botticelli", "Pintura", 1445, 1510),  
("Borromini", "Arquitectura", 1599, 1667),  
("Bach", "Musica", 1685, 1750)]  
  
def nombres(bd: BD) -> list[str]:  
    return [p[0] for p in bd]  
  
def musicos(bd: BD) -> list[str]:  
    return [p[0] for p in bd if p[1] == "Musica"]  
  
def seleccion(bd: BD, m: str) -> list[str]:  
    return [p[0] for p in bd if p[1] == m]  
  
def musicos2(bd: BD) -> list[str]:  
    return seleccion(bd, "Musica")  
  
def vivas(bd: BD, a: int) -> list[str]:  
    return [p[0] for p in bd if p[2] <= a <= p[3]]
```

Capítulo 3

Definiciones por recursión

En este capítulo se presentan ejercicios con definiciones por comprensión. Se corresponden con el [tema 6 del curso de programación funcional con Haskell](https://jaalonso.github.io/materias/PFconHaskell/temas/tema-6.html) ¹.

Contenido

3.1. Potencia entera	230
3.2. Algoritmo de Euclides del mcd	236
3.3. Dígitos de un número	238
3.4. Suma de los dígitos de un número	246
3.5. Número a partir de sus dígitos	251
3.6. Exponente de la mayor potencia de x que divide a y	258
3.7. Producto cartesiano de dos conjuntos	263
3.8. Subconjuntos de un conjunto	267
3.9. El algoritmo de Luhn	272
3.10. Números de Lychrel	277
3.11. Suma de los dígitos de una cadena	288
3.12. Primera en mayúscula y restantes en minúscula	292
3.13. Mayúsculas iniciales	296
3.14. Posiciones de un carácter en una cadena	301
3.15. Reconocimiento de subcadenas	305

¹<https://jaalonso.github.io/materias/PFconHaskell/temas/tema-6.html>

3.1. Potencia entera

En Haskell

```

-----
-- Definir la función
-- potencia :: Integer -> Integer -> Integer
-- tal que (potencia x n) es x elevado al número natural n. Por ejemplo,
-- potencia 2 3 == 8
-----

module Potencia_entera where

import Data.List (foldl')
import Control.Arrow ((***))
import Test.QuickCheck

-- 1ª solución
-- =====

potencial :: Integer -> Integer -> Integer
potencial _ 0 = 1
potencial m n = m * potencial m (n-1)

-- 2ª solución
-- =====

potencia2 :: Integer -> Integer -> Integer
potencia2 m = aux
  where aux 0 = 1
        aux n = m * aux (n-1)

-- 3ª solución
-- =====

potencia3 :: Integer -> Integer -> Integer
potencia3 m = aux 1
  where aux r 0 = r
        aux r n = aux (r*m) (n-1)

-- 4ª solución

```

```

-- =====

potencia4 :: Integer -> Integer -> Integer
potencia4 m = aux 1
  where aux r 0 = r
        aux r n = (aux $! (r*m)) $! (n-1)

-- 5ª solución
-- =====

potencia5 :: Integer -> Integer -> Integer
potencia5 m n = product [m | _ <- [1..n]]

-- 6ª solución
-- =====

potencia6 :: Integer -> Integer -> Integer
potencia6 m n = foldl' (*) 1 [m | _ <- [1..n]]

-- 7ª solución
-- =====

potencia7 :: Integer -> Integer -> Integer
potencia7 m n =
  fst (until (\ (_,k) -> k == n)
    (\ (r,k) -> (r*m, k+1))
    (1,0))

-- 8ª solución
-- =====

potencia8 :: Integer -> Integer -> Integer
potencia8 m n =
  fst (until ((== n) . snd)
    ((m *) *** (1 +))
    (1,0))

-- 9ª solución
-- =====

```

```

potencia9 :: Integer -> Integer -> Integer
potencia9 m n = m^n

-- Comprobación de equivalencia
-- =====

-- La propiedad es
prop_potencia :: Integer -> NonNegative Integer -> Bool
prop_potencia m (NonNegative n) =
  all (== potencia1 m n)
    [potencia2 m n,
     potencia3 m n,
     potencia4 m n,
     potencia5 m n,
     potencia6 m n,
     potencia7 m n,
     potencia8 m n,
     potencia9 m n]

-- La comprobación es
--   λ> quickCheck prop_potencia
--   +++ OK, passed 100 tests.

-- Comparación de eficiencia
-- =====

-- La comparación es
--   λ> length (show (potencia1 2 (2*10^5)))
--   60206
--   (2.97 secs, 2,602,252,408 bytes)
--   λ> length (show (potencia2 2 (2*10^5)))
--   60206
--   (2.63 secs, 2,624,652,624 bytes)
--   λ> length (show (potencia3 2 (2*10^5)))
--   60206
--   (3.41 secs, 2,619,606,368 bytes)
--   λ> length (show (potencia4 2 (2*10^5)))
--   60206
--   (0.64 secs, 2,636,888,928 bytes)
--   λ> length (show (potencia5 2 (2*10^5)))

```



```
--      60206
--      (2.47 secs, 2,597,108,000 bytes)
--      λ> length (show (potencia6 2 (2*10^5)))
--      60206
--      (0.35 secs, 2,582,488,824 bytes)
--      λ> length (show (potencia7 2 (2*10^5)))
--      60206
--      (2.48 secs, 2,616,406,272 bytes)
--      λ> length (show (potencia8 2 (2*10^5)))
--      60206
--      (2.40 secs, 2,608,652,736 bytes)
--      λ> length (show (potencia9 2 (2*10^5)))
--      60206
--      (0.01 secs, 4,212,968 bytes)
--
--      λ> length (show (potencia4 2 (10^6)))
--      301030
--      (10.39 secs, 63,963,999,656 bytes)
--      λ> length (show (potencia6 2 (10^6)))
--      301030
--      (8.90 secs, 63,691,999,552 bytes)
--      λ> length (show (potencia9 2 (10^6)))
--      301030
--      (0.04 secs, 19,362,032 bytes)
```

En Python

```
# -----
# Definir la función
# potencia : (int, int) -> int
# tal que potencia(x, n) es x elevado al número natural n. Por ejemplo,
# potencia(2, 3) == 8
# -----

from functools import reduce
from operator import mul
from sys import setrecursionlimit
from timeit import Timer, default_timer

from hypothesis import given
```

```
from hypothesis import strategies as st

setrecursionlimit(10**6)

# 1ª solución
# =====

def potencial(m: int, n: int) -> int:
    if n == 0:
        return 1
    return m * potencial(m, n-1)

# 2ª solución
# =====

def potencia2(m: int, n: int) -> int:
    def aux(k: int) -> int:
        if k == 0:
            return 1
        return m * aux(k-1)
    return aux(n)

# 3ª solución
# =====

def potencia3(m: int, n: int) -> int:
    def aux(r: int, k: int) -> int:
        if k == 0:
            return r
        return aux(r*m, k-1)
    return aux(1, n)

# 4ª solución
# =====

# producto(xs) es el producto de los elementos de xs. Por ejemplo,
# producto([2, 3, 5]) == 30
def producto(xs: list[int]) -> int:
    return reduce(mul, xs, 1)
```

```

def potencia4(m: int, n: int) -> int:
    return producto([m]*n)

# 5ª solución
# =====

def potencia5(m: int, n: int) -> int:
    r = 1
    for _ in range(0, n):
        r = r * m
    return r

# 6ª solución
# =====

def potencia6(m: int, n: int) -> int:
    return m**n

# Comprobación de equivalencia
# =====

# La propiedad es
@given(st.integers(),
      st.integers(min_value=0, max_value=100))
def test_potencia(m: int, n: int) -> None:
    r = potencia1(m, n)
    assert potencia2(m, n) == r
    assert potencia3(m, n) == r
    assert potencia4(m, n) == r
    assert potencia5(m, n) == r
    assert potencia6(m, n) == r

# La comprobación es
#   src> poetry run pytest -q potencia_entera.py
#   1 passed in 0.17s

# Comparación de eficiencia
# =====

def tiempo(e: str) -> None:

```

```

"""Tiempo (en segundos) de evaluar la expresión e."""
t = Timer(e, "", default_timer, globals()).timeit(1)
print(f"{t:0.2f} segundos")

# La comparación es
# >>> tiempo('potencia1(2, 2*10**4)')
# 0.01 segundos
# >>> tiempo('potencia2(2, 2*10**4)')
# 0.01 segundos
# >>> tiempo('potencia3(2, 2*10**4)')
# 0.02 segundos
# >>> tiempo('potencia4(2, 2*10**4)')
# 0.01 segundos
# >>> tiempo('potencia5(2, 2*10**4)')
# 0.01 segundos
# >>> tiempo('potencia6(2, 2*10**4)')
# 0.00 segundos
#
# >>> tiempo('potencia4(2, 5*10**5)')
# 2.87 segundos
# >>> tiempo('potencia5(2, 5*10**5)')
# 3.17 segundos
# >>> tiempo('potencia6(2, 5*10**5)')
# 0.00 segundos

```

3.2. Algoritmo de Euclides del mcd

En Haskell

```

-----
-- Dados dos números naturales, a y b, es posible calcular su máximo
-- común divisor mediante el Algoritmo de Euclides. Este algoritmo se
-- puede resumir en la siguiente fórmula:
--      
$$\text{mcd}(a,b) = a, \quad \text{si } b = 0$$

--      
$$= \text{mcd}(b, a \bmod b), \text{ si } b > 0$$

--
-- Definir la función
--      
$$\text{mcd} :: \text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer}$$

-- 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
--
--  Comprobar con QuickCheck que el máximo común divisor de dos números a
--  y b (ambos mayores que 0) es siempre mayor o igual que 1 y además es
--  menor o igual que el menor de los números a  y b.
--  -----

module Algoritmo_de_Euclides_del_mcd where

import Test.QuickCheck

mcd :: Integer -> Integer -> Integer
mcd a 0 = a
mcd a b = mcd b (a `mod` b)

-- La propiedad es
prop_mcd :: Positive Integer -> Positive Integer -> Bool
prop_mcd (Positive a) (Positive b) =
  m >= 1 && m <= min a b
  where m = mcd a b

-- La comprobación es
--      λ> quickCheck prop_mcd
--      OK, passed 100 tests.

```

En Python

```

# -----
#  Dados dos números naturales, a y b, es posible calcular su máximo
#  común divisor mediante el Algoritmo de Euclides. Este algoritmo se
#  puede resumir en la siguiente fórmula:
#      mcd(a,b) = a,                si b = 0
#               = mcd (b, a módulo b), si b > 0
#
#  Definir la función
#      mcd : (int, nt) -> int
#  tal que mcd(a, b) es el máximo común divisor de a y b calculado
#  mediante el algoritmo de Euclides. Por ejemplo,
#      mcd(30, 45) == 15
#      mcd(45, 30) == 15

```

```

#
# Comprobar con Hypothesis que el máximo común divisor de dos números a
# y b (ambos mayores que 0) es siempre mayor o igual que 1 y además es
# menor o igual que el menor de los números a y b.
# -----

from hypothesis import given
from hypothesis import strategies as st

def mcd(a: int, b: int) -> int:
    if b == 0:
        return a
    return mcd(b, a % b)

# -- La propiedad es
@given(st.integers(min_value=1, max_value=1000),
       st.integers(min_value=1, max_value=1000))
def test_mcd(a: int, b: int) -> None:
    assert 1 <= mcd(a, b) <= min(a, b)

# La comprobación es
# src> poetry run pytest -q algoritmo_de_Euclides_del_mcd.py
# 1 passed in 0.22s

```

3.3. Dígitos de un número

En Haskell

```

-- -----
-- Definir la función
--   digitos :: Integer -> [Int]
-- tal que (digitos n) es la lista de los dígitos del número n. Por
-- ejemplo,
--   digitos 320274 == [3,2,0,2,7,4]
-- -----

module Digitos_de_un_numero where

import Data.Char (digitToInt)

```

```
import qualified Data.Digits as D (digits)
import qualified Data.FastDigits as FD (digits)
import Test.QuickCheck

-- 1ª solución
-- =====

digitos1 :: Integer -> [Int]
digitos1 n = map fromInteger (aux n)
  where aux :: Integer -> [Integer]
        aux m
          | m < 10    = [m]
          | otherwise = aux (m `div` 10) ++ [m `rem` 10]

-- 2ª solución
-- =====

digitos2 :: Integer -> [Int]
digitos2 n = map fromInteger (reverse (aux n))
  where aux :: Integer -> [Integer]
        aux m
          | m < 10    = [m]
          | otherwise = (m `rem` 10) : aux (m `div` 10)

-- 3ª solución
-- =====

digitos3 :: Integer -> [Int]
digitos3 n = map fromInteger (aux [] n)
  where aux :: [Integer] -> Integer -> [Integer]
        aux ds m
          | m < 10    = m : ds
          | otherwise = aux (m `rem` 10 : ds) (m `div` 10)

-- 4ª solución
-- =====

digitos4 :: Integer -> [Int]
digitos4 n = [read [x] | x <- show n]
```

```
-- 5ª solución
```

```
-- =====
```

```
digitos5 :: Integer -> [Int]
```

```
digitos5 n = map (\ x -> read [x]) (show n)
```

```
-- 6ª solución
```

```
-- =====
```

```
digitos6 :: Integer -> [Int]
```

```
digitos6 = map (read . return) . show
```

```
-- 7ª solución
```

```
-- =====
```

```
digitos7 :: Integer -> [Int]
```

```
digitos7 n = map digitToInt (show n)
```

```
-- 8ª solución
```

```
-- =====
```

```
digitos8 :: Integer -> [Int]
```

```
digitos8 = map digitToInt . show
```

```
-- 9ª solución
```

```
-- =====
```

```
digitos9 :: Integer -> [Int]
```

```
digitos9 0 = [0]
```

```
digitos9 n = map fromInteger (D.digits 10 n)
```

```
-- 10ª solución
```

```
-- =====
```

```
digitos10 :: Integer -> [Int]
```

```
digitos10 0 = [0]
```

```
digitos10 n = reverse (FD.digits 10 n)
```

```
-- Comprobación de equivalencia
```

```
-- =====
```



```

-- La propiedad es
prop_digitos :: NonNegative Integer -> Bool
prop_digitos (NonNegative n) =
  all (== digitos1 n)
    [digitos2 n,
     digitos3 n,
     digitos4 n,
     digitos5 n,
     digitos6 n,
     digitos7 n,
     digitos8 n,
     digitos9 n,
     digitos10 n]

-- La comprobación es
--   λ> quickCheck prop_digitos
--   +++ OK, passed 100 tests.

-- Comparación de eficiencia
-- =====

-- La comparación es
--   λ> n = product [1..5000]
--   λ> length (digitos1 n)
--   16326
--   (3.00 secs, 11,701,450,912 bytes)
--   λ> length (digitos2 n)
--   16326
--   (0.13 secs, 83,393,816 bytes)
--   λ> length (digitos3 n)
--   16326
--   (0.11 secs, 83,132,552 bytes)
--   λ> length (digitos4 n)
--   16326
--   (0.01 secs, 23,054,920 bytes)
--   λ> length (digitos5 n)
--   16326
--   (0.01 secs, 22,663,088 bytes)
--   λ> length (digitos6 n)

```

```
-- 16326
-- (0.06 secs, 22,663,224 bytes)
-- λ> length (digitos7 n)
-- 16326
-- (0.01 secs, 22,663,064 bytes)
-- λ> length (digitos8 n)
-- 16326
-- (0.03 secs, 22,663,192 bytes)
-- λ> length (digitos9 n)
-- 16326
-- (0.05 secs, 82,609,944 bytes)
-- λ> length (digitos10 n)
-- 16326
-- (0.01 secs, 26,295,416 bytes)
--
-- λ> n = product [1..5*10^4]
-- λ> length (digitos2 n)
-- 213237
-- (10.17 secs, 12,143,633,056 bytes)
-- λ> length (digitos3 n)
-- 213237
-- (10.54 secs, 12,140,221,216 bytes)
-- λ> length (digitos4 n)
-- 213237
-- (1.29 secs, 2,638,199,328 bytes)
-- λ> length (digitos5 n)
-- 213237
-- (2.48 secs, 2,633,081,632 bytes)
-- λ> length (digitos6 n)
-- 213237
-- (2.59 secs, 2,633,081,600 bytes)
-- λ> length (digitos7 n)
-- 213237
-- (2.55 secs, 2,633,081,608 bytes)
-- λ> length (digitos8 n)
-- 213237
-- (2.49 secs, 2,633,081,600 bytes)
-- λ> length (digitos9 n)
-- 213237
-- (7.07 secs, 12,133,397,456 bytes)
```

```
-- λ> length (digitos10 n)
-- 213237
-- (2.47 secs, 2,725,182,064 bytes)
```

En Python

```
# -----
# Definir la función
#   digitos : (int) -> list[int]
# tal que digitos(n) es la lista de los dígitos del número n. Por
# ejemplo,
#   digitos(320274) == [3, 2, 0, 2, 7, 4]
# -----

from math import factorial
from sys import setrecursionlimit
from timeit import Timer, default_timer

from hypothesis import given
from hypothesis import strategies as st
from sympy.ntheory.digits import digits

setrecursionlimit(10**6)

# 1ª solución
# =====

def digitos1(n: int) -> list[int]:
    if n < 10:
        return [n]
    return digitos1(n // 10) + [n % 10]

# 2ª solución
# =====

def digitos2(n: int) -> list[int]:
    return [int(x) for x in str(n)]

# 3ª solución
# =====
```

```
def digitos3(n: int) -> list[int]:
    r: list[int] = []
    for x in str(n):
        r.append(int(x))
    return r
```

4ª solución

=====

```
def digitos4(n: int) -> list[int]:
    return list(map(int, list(str(n))))
```

5ª solución

=====

```
def digitos5(n: int) -> list[int]:
    r: list[int] = []
    while n > 0:
        r = [n % 10] + r
        n = n // 10
    return r
```

6ª solución

=====

```
def digitos6(n: int) -> list[int]:
    r: list[int] = []
    while n > 0:
        r.append(n % 10)
        n = n // 10
    return list(reversed(r))
```

7ª solución

=====

```
def digitos7(n: int) -> list[int]:
    return digits(n)[1:]
```

Comprobación de equivalencia

```
# =====

# La propiedad es
@given(st.integers(min_value=1))
def test_digitos(n: int) -> None:
    r = digitos1(n)
    assert digitos2(n) == r
    assert digitos3(n) == r
    assert digitos4(n) == r
    assert digitos5(n) == r
    assert digitos6(n) == r
    assert digitos7(n) == r

# La comprobación es
# src> poetry run pytest -q digitos_de_un_numero.py
# 1 passed in 0.49s

# Comparación de eficiencia
# =====

def tiempo(ex: str) -> None:
    """Tiempo (en segundos) de evaluar la expresión e."""
    t = Timer(ex, "", default_timer, globals()).timeit(1)
    print(f"{t:0.2f} segundos")

# La comparación es
# >>> tiempo('digitos1(factorial(6000))')
# 0.58 segundos
# >>> tiempo('digitos2(factorial(6000))')
# 0.01 segundos
# >>> tiempo('digitos3(factorial(6000))')
# 0.01 segundos
# >>> tiempo('digitos4(factorial(6000))')
# 0.01 segundos
# >>> tiempo('digitos5(factorial(6000))')
# 0.60 segundos
# >>> tiempo('digitos6(factorial(6000))')
# 0.17 segundos
# >>> tiempo('digitos7(factorial(6000))')
# 0.10 segundos
```

```
#
# >>> tiempo('digitos2(factorial(2*10**4))')
# 0.10 segundos
# >>> tiempo('digitos3(factorial(2*10**4))')
# 0.10 segundos
# >>> tiempo('digitos4(factorial(2*10**4))')
# 0.09 segundos
# >>> tiempo('digitos6(factorial(2*10**4))')
# 2.33 segundos
# >>> tiempo('digitos7(factorial(2*10**4))')
# 1.18 segundos
#
# >>> tiempo('digitos2(factorial(10**5))')
# 3.53 segundos
# >>> tiempo('digitos3(factorial(10**5))')
# 3.22 segundos
# >>> tiempo('digitos4(factorial(10**5))')
# 3.02 segundos
```

3.4. Suma de los dígitos de un número

En Haskell

```
-- -----
-- Definir la función
-- sumaDigitos :: Integer -> Integer
-- tal que (sumaDigitos n) es la suma de los dígitos de n. Por ejemplo,
-- sumaDigitos 3 == 3
-- sumaDigitos 2454 == 15
-- sumaDigitos 20045 == 11
-- -----

module Suma_de_los_digitos_de_un_numero where

import Data.List (foldl')
import Test.QuickCheck

-- 1ª solución
-- =====
```

```

sumaDigitos1 :: Integer -> Integer
sumaDigitos1 n = sum (digitos n)

-- (digitos n) es la lista de los dígitos del número n. Por ejemplo,
--   digitos 320274 == [3,2,0,2,7,4]
digitos :: Integer -> [Integer]
digitos n = [read [x] | x <- show n]

-- Nota. En lugar de la definición anterior de digitos se puede usar
-- cualquiera del ejercicio "Dígitos de un número" https://bit.ly/3Tkhc2T

-- 2ª solución
-- =====

sumaDigitos2 :: Integer -> Integer
sumaDigitos2 n = foldl' (+) 0 (digitos n)

-- 3ª solución
-- =====

sumaDigitos3 :: Integer -> Integer
sumaDigitos3 n
  | n < 10    = n
  | otherwise = n `rem` 10 + sumaDigitos3 (n `div` 10)

-- 4ª solución
-- =====

sumaDigitos4 :: Integer -> Integer
sumaDigitos4 = aux 0
  where aux r n
        | n < 10    = r + n
        | otherwise = aux (r + n `rem` 10) (n `div` 10)

-- Comprobación de equivalencia
-- =====

-- La propiedad es
prop_sumaDigitos :: NonNegative Integer -> Bool
prop_sumaDigitos (NonNegative n) =

```

```

all (== sumaDigitos1 n)
    [sumaDigitos2 n,
     sumaDigitos3 n,
     sumaDigitos4 n]

-- La comprobación es
--   λ> quickCheck prop_sumaDigitos
--   +++ OK, passed 100 tests.

-- Comparación de eficiencia
-- =====

-- La comparación es
--   λ> sumaDigitos1 (product [1..2*10^4])
--   325494
--   (0.64 secs, 665,965,832 bytes)
--   λ> sumaDigitos2 (product [1..2*10^4])
--   325494
--   (0.41 secs, 660,579,064 bytes)
--   λ> sumaDigitos3 (product [1..2*10^4])
--   325494
--   (1.58 secs, 1,647,082,224 bytes)
--   λ> sumaDigitos4 (product [1..2*10^4])
--   325494
--   (1.72 secs, 1,662,177,792 bytes)
--
--   λ> sumaDigitos1 (product [1..5*10^4])
--   903555
--   (2.51 secs, 3,411,722,136 bytes)
--   λ> sumaDigitos2 (product [1..5*10^4])
--   903555
--   (2.30 secs, 3,396,802,856 bytes)

```

En Python

```

# -----
# Definir la función
#   sumaDigitos : (int) -> int
# tal que sumaDigitos(n) es la suma de los dígitos de n. Por ejemplo,
#   sumaDigitos(3)      == 3

```



```

# sumaDigitos(2454) == 15
# sumaDigitos(20045) == 11
# -----

from functools import reduce
from math import factorial
from operator import add
from sys import setrecursionlimit
from timeit import Timer, default_timer

from hypothesis import given
from hypothesis import strategies as st

setrecursionlimit(10**6)

# 1ª solución
# =====

# digitos(n) es la lista de los dígitos del número n. Por ejemplo,
# digitos(320274) == [3, 2, 0, 2, 7, 4]
def digitos(n: int) -> list[int]:
    return list(map(int, list(str(n))))

def sumaDigitos1(n: int) -> int:
    return sum(digitos(n))

# Nota. En lugar de la definición anterior de digitos se puede usar
# cualquiera del ejercicio "Dígitos de un número" https://bit.ly/3Tkhc2T

# 2ª solución
# =====

def sumaDigitos2(n: int) -> int:
    return reduce(add, digitos(n))

# 3ª solución
# =====

def sumaDigitos3(n: int) -> int:
    if n < 10:

```

```

        return n
    return n % 10 + sumaDigitos3(n // 10)

# 4ª solución
# =====

def sumaDigitos4(n: int) -> int:
    def aux(r: int, m: int) -> int:
        if m < 10:
            return r + m
        return aux(r + m % 10, m // 10)
    return aux(0, n)

# 5ª solución
# =====

def sumaDigitos5(n: int) -> int:
    r = 0
    for x in digitos(n):
        r = r + x
    return r

# Comprobación de equivalencia
# =====

# La propiedad es
@given(st.integers(min_value=0, max_value=1000))
def test_sumaDigitos(n: int) -> None:
    r = sumaDigitos1(n)
    assert sumaDigitos2(n) == r
    assert sumaDigitos3(n) == r
    assert sumaDigitos4(n) == r
    assert sumaDigitos5(n) == r

# La comprobación es
#   src> poetry run pytest -q suma_de_los_digitos_de_un_numero.py
#   1 passed in 0.35s

# Comparación de eficiencia
# =====

```

```
def tiempo(e: str) -> None:
    """Tiempo (en segundos) de evaluar la expresión e."""
    t = Timer(e, "", default_timer, globals()).timeit(1)
    print(f"{t:0.2f} segundos")

# La comparación es
# >>> tiempo('sumaDigitos1(factorial(6*10**3))')
# 0.01 segundos
# >>> tiempo('sumaDigitos2(factorial(6*10**3))')
# 0.01 segundos
# >>> tiempo('sumaDigitos3(factorial(6*10**3))')
# 0.13 segundos
# >>> tiempo('sumaDigitos4(factorial(6*10**3))')
# 0.13 segundos
# >>> tiempo('sumaDigitos5(factorial(6*10**3))')
# 0.01 segundos
#
# >>> tiempo('sumaDigitos1(factorial(10**5))')
# 2.20 segundos
# >>> tiempo('sumaDigitos2(factorial(10**5))')
# 2.22 segundos
# >>> tiempo('sumaDigitos5(factorial(10**5))')
# 2.19 segundos
```

3.5. Número a partir de sus dígitos

En Haskell

```
-- -----
-- Definir la función
-- listaNumero :: [Integer] -> Integer
-- tal que (listaNumero xs) es el número formado por los dígitos xs. Por
-- ejemplo,
-- listaNumero [5] == 5
-- listaNumero [1,3,4,7] == 1347
-- listaNumero [0,0,1] == 1
-- -----

{-# OPTIONS_GHC -fno-warn-unused-imports #-}
```

```
{-# OPTIONS_GHC -fno-warn-type-defaults #-}
```

```
module Numero_a_partir_de_sus_digitos where
```

```
import Data.List (foldl')
import Data.Digits (unDigits)
import Test.QuickCheck
```

```
-- 1ª solución
-- =====
```

```
listaNumero1 :: [Integer] -> Integer
listaNumero1 = aux . reverse
  where
    aux :: [Integer] -> Integer
    aux [] = 0
    aux (x:xs) = x + 10 * aux xs
```

```
-- 2ª solución
-- =====
```

```
listaNumero2 :: [Integer] -> Integer
listaNumero2 = aux 0
  where
    aux :: Integer -> [Integer] -> Integer
    aux r [] = r
    aux r (x:xs) = aux (x+10*r) xs
```

```
-- 3ª solución
-- =====
```

```
listaNumero3 :: [Integer] -> Integer
listaNumero3 = aux 0
  where
    aux :: Integer -> [Integer] -> Integer
    aux = foldl (\ r x -> x + 10 * r)
```

```
-- 4ª solución
-- =====
```

```

listaNumero4 :: [Integer] -> Integer
listaNumero4 = foldl' (\ r x -> x + 10 * r) 0

-- 5ª solución
-- =====

listaNumero5 :: [Integer] -> Integer
listaNumero5 xs = sum [y*10^n | (y,n) <- zip (reverse xs) [0..]]

-- 6ª solución
-- =====

listaNumero6 :: [Integer] -> Integer
listaNumero6 xs = sum (zipWith (\ y n -> y*10^n) (reverse xs) [0..])

-- 7ª solución
-- =====

listaNumero7 :: [Integer] -> Integer
listaNumero7 = unDigits 10

-- 7ª solución
-- =====

listaNumero8 :: [Integer] -> Integer
listaNumero8 = read . concatMap show

-- Comprobación de equivalencia
-- =====

-- La propiedad es
prop_listaNumero :: NonEmptyList Integer -> Bool
prop_listaNumero (NonEmpty xs) =
  all (== listaNumero1 ys)
    [listaNumero2 ys,
     listaNumero3 ys,
     listaNumero4 ys,
     listaNumero5 ys,
     listaNumero6 ys,
     listaNumero7 ys,

```

```

        listaNumero8 ys]
where ys = map (`mod` 10) xs

-- La comprobación es
--   λ> quickCheck prop_listaNumero
--   +++ OK, passed 100 tests.

-- Comparación de eficiencia
-- =====

-- La comparación es
--   λ> length (show (listaNumero1 (replicate (10^5) 9)))
--   100000
--   (4.01 secs, 4,309,740,064 bytes)
--   λ> length (show (listaNumero2 (replicate (10^5) 9)))
--   100000
--   (4.04 secs, 4,307,268,856 bytes)
--   λ> length (show (listaNumero3 (replicate (10^5) 9)))
--   100000
--   (4.08 secs, 4,300,868,816 bytes)
--   λ> length (show (listaNumero4 (replicate (10^5) 9)))
--   100000
--   (0.42 secs, 4,288,480,208 bytes)
--   λ> length (show (listaNumero4 (replicate (10^5) 9)))
--   100000
--   (0.41 secs, 4,288,480,208 bytes)
--   λ> length (show (listaNumero5 (replicate (10^5) 9)))
--   100000
--   (43.35 secs, 10,702,827,328 bytes)
--   λ> length (show (listaNumero6 (replicate (10^5) 9)))
--   100000
--   (46.89 secs, 10,693,227,280 bytes)
--   λ> length (show (listaNumero7 (replicate (10^5) 9)))
--   100000
--   (4.33 secs, 4,297,499,344 bytes)
--   λ> length (show (listaNumero8 (replicate (10^5) 9)))
--   100000
--   (0.03 secs, 60,760,360 bytes)

```

En Python

```
# -----
# Definir la función
# listaNumero : (list[int]) -> int
# tal que listaNumero(xs) es el número formado por los dígitos xs. Por
# ejemplo,
# listaNumero([5]) == 5
# listaNumero([1, 3, 4, 7]) == 1347
# listaNumero([0, 0, 1]) == 1
# -----
```

```
from functools import reduce
from sys import setrecursionlimit
from timeit import Timer, default_timer
```

```
from hypothesis import given
from hypothesis import strategies as st
```

```
setrecursionlimit(10**6)
```

```
# 1ª solución
# =====
```

```
def listaNumero1(xs: list[int]) -> int:
    def aux(ys: list[int]) -> int:
        if ys:
            return ys[0] + 10 * aux(ys[1:])
        return 0
    return aux(list(reversed(xs)))
```

```
# 2ª solución
# =====
```

```
def listaNumero2(xs: list[int]) -> int:
    def aux(r: int, ys: list[int]) -> int:
        if ys:
            return aux(ys[0] + 10 * r, ys[1:])
        return r
    return aux(0, xs)
```

3ª solución

=====

```
def listaNumero3(xs: list[int]) -> int:
    return reduce((lambda r, x: x + 10 * r), xs)
```

4ª solución

=====

```
def listaNumero4(xs: list[int]) -> int:
    r = 0
    for x in xs:
        r = x + 10 * r
    return r
```

5ª solución

=====

```
def listaNumero5(xs: list[int]) -> int:
    return sum((y * 10**n
                for (y, n) in zip(list(reversed(xs)), range(0, len(xs)))))
```

6ª solución

=====

```
def listaNumero6(xs: list[int]) -> int:
    return int("".join(list(map(str, xs))))
```

Comprobación de equivalencia

=====

La propiedad es

```
@given(st.lists(st.integers(min_value=0, max_value=9), min_size=1))
```

```
def test_listaNumero(xs: list[int]) -> None:
    r = listaNumero1(xs)
    assert listaNumero2(xs) == r
    assert listaNumero3(xs) == r
    assert listaNumero4(xs) == r
    assert listaNumero5(xs) == r
    assert listaNumero6(xs) == r
```



```
# La comprobación es
# src> poetry run pytest -q numero_a_partir_de_sus_digitos.py
# 1 passed in 0.27s

# Comparación de eficiencia
# =====

def tiempo(e: str) -> None:
    """Tiempo (en segundos) de evaluar la expresión e."""
    t = Timer(e, "", default_timer, globals()).timeit(1)
    print(f"{t:0.2f} segundos")

# La comparación es
# >>> tiempo('listaNumero1([9]*(10**4))')
# 0.28 segundos
# >>> tiempo('listaNumero2([9]*(10**4))')
# 0.16 segundos
# >>> tiempo('listaNumero3([9]*(10**4))')
# 0.01 segundos
# >>> tiempo('listaNumero4([9]*(10**4))')
# 0.01 segundos
# >>> tiempo('listaNumero5([9]*(10**4))')
# 0.41 segundos
# >>> tiempo('listaNumero6([9]*(10**4))')
# 0.00 segundos
#
# >>> tiempo('listaNumero3([9]*(2*10**5))')
# 3.45 segundos
# >>> tiempo('listaNumero4([9]*(2*10**5))')
# 3.29 segundos
# >>> tiempo('listaNumero6([9]*(2*10**5))')
# 0.19 segundos
```

3.6. Exponente de la mayor potencia de x que divide a y

En Haskell

```

-- -----
-- Definir la función
--   mayorExponente :: Integer -> Integer -> Integer
-- tal que (mayorExponente a b) es el exponente de la mayor potencia de
-- a que divide b. Por ejemplo,
--   mayorExponente 2 8    == 3
--   mayorExponente 2 9    == 0
--   mayorExponente 5 100  == 2
--   mayorExponente 2 60   == 2
--
-- Nota: Se supone que a > 1 y b > 0.
-- -----

{-# OPTIONS_GHC -fno-warn-type-defaults #-}

module Exponente_mayor where

import Test.QuickCheck

-- 1ª solución
-- =====

mayorExponente1 :: Integer -> Integer -> Integer
mayorExponente1 a b
  | rem b a /= 0 = 0
  | otherwise   = 1 + mayorExponente1 a (b `div` a)

-- 2ª solución
-- =====

mayorExponente2 :: Integer -> Integer -> Integer
mayorExponente2 a b = aux b 0
  where
    aux c r | rem c a /= 0 = r
             | otherwise   = aux (c `div` a) (r + 1)

```

```

-- 3ª solución
-- =====

mayorExponente3 :: Integer -> Integer -> Integer
mayorExponente3 a b = head [x-1 | x <- [0..], mod b (a^x) /= 0]

-- 4ª solución
-- =====

mayorExponente4 :: Integer -> Integer -> Integer
mayorExponente4 a b =
    fst (until (\ (_,c) -> rem c a /= 0)
              (\ (r,c) -> (r+1, c `div` a))
        (0,b))

-- Comprobación de equivalencia
-- =====

-- La propiedad es
prop_mayorExponente :: Integer -> Integer -> Property
prop_mayorExponente a b =
    a > 1 && b > 0 ==>
    all (== mayorExponente1 a b)
        [mayorExponente2 a b,
         mayorExponente3 a b,
         mayorExponente4 a b]

-- La comprobación es
--    λ> quickCheck prop_mayorExponente
--    +++ OK, passed 100 tests; 457 discarded.

-- Comparación de eficiencia
-- =====

-- La comparación es
--    λ> mayorExponente1 2 (2^(5*10^4))
--    50000
--    (0.12 secs, 179,578,424 bytes)
--    λ> mayorExponente2 2 (2^(5*10^4))

```

```
-- 50000
-- (0.13 secs, 181,533,376 bytes)
-- λ> mayorExponente3 2 (2^(5*10^4))
-- 50000
-- (3.88 secs, 818,319,096 bytes)
-- λ> mayorExponente4 2 (2^(5*10^4))
-- 50000
-- (0.13 secs, 181,133,344 bytes)
--
-- λ> mayorExponente1 2 (2^(3*10^5))
-- 300000
-- (2.94 secs, 5,762,199,064 bytes)
-- λ> mayorExponente2 2 (2^(3*10^5))
-- 300000
-- (2.91 secs, 5,773,829,624 bytes)
-- λ> mayorExponente4 2 (2^(3*10^5))
-- 300000
-- (3.70 secs, 5,771,396,824 bytes)
```

En Python

```
# -----
# Definir la función
#   mayorExponente : (int, int) -> int
# tal que mayorExponente(a, b) es el exponente de la mayor potencia de
# a que divide b. Por ejemplo,
#   mayorExponente(2, 8)    == 3
#   mayorExponente(2, 9)    == 0
#   mayorExponente(5, 100)  == 2
#   mayorExponente(2, 60)   == 2
#
# Nota: Se supone que a > 1 y b > 0.
# -----

from itertools import islice
from sys import setrecursionlimit
from timeit import Timer, default_timer
from typing import Iterator

from hypothesis import given
```

```
from hypothesis import strategies as st

setrecursionlimit(10**6)

# 1ª solución
# =====

def mayorExponente1(a: int, b: int) -> int:
    if b % a != 0:
        return 0
    return 1 + mayorExponente1(a, b // a)

# 2ª solución
# =====

def mayorExponente2(a: int, b: int) -> int:
    def aux(c: int, r: int) -> int:
        if c % a != 0:
            return r
        return aux(c // a, r + 1)
    return aux(b, 0)

# 3ª solución
# =====

# naturales es el generador de los números naturales, Por ejemplo,
# >>> list(islice(naturales(), 5))
# [0, 1, 2, 3, 4]
def naturales() -> Iterator[int]:
    i = 0
    while True:
        yield i
        i += 1

def mayorExponente3(a: int, b: int) -> int:
    return list(islice((x - 1 for x in naturales() if b % (a**x) != 0), 1))[0]

# 4ª solución
# =====
```

```

def mayorExponente4(a: int, b: int) -> int:
    r = 0
    while b % a == 0:
        b = b // a
        r = r + 1
    return r

# Comprobación de equivalencia
# =====

def pruebal() -> None:
    for x in range(2, 11):
        for y in range(1, 11):
            print(x, y, mayorExponente4(x, y))

# La propiedad es
@given(st.integers(min_value=2, max_value=10),
       st.integers(min_value=1, max_value=10))
def test_mayorExponente(a: int, b: int) -> None:
    r = mayorExponente1(a, b)
    assert mayorExponente2(a, b) == r
    assert mayorExponente3(a, b) == r
    assert mayorExponente4(a, b) == r

# La comprobación es
# src> poetry run pytest -q exponente_mayor.py
# 1 passed in 0.16s

# Comparación de eficiencia
# =====

def tiempo(e: str) -> None:
    """Tiempo (en segundos) de evaluar la expresión e."""
    t = Timer(e, "", default_timer, globals()).timeit(1)
    print(f"{t:0.2f} segundos")

# La comparación es
# >>> tiempo('mayorExponente1(2, 2**(2*10**4))')
# 0.13 segundos

```

```
# >>> tiempo('mayorExponente2(2, 2**(2*10**4))')
# 0.13 segundos
# >>> tiempo('mayorExponente3(2, 2**(2*10**4))')
# 1.81 segundos
# >>> tiempo('mayorExponente4(2, 2**(2*10**4))')
# 0.12 segundos
#
# >>> tiempo('mayorExponente4(2, 2**(2*10**5))')
# 12.19 segundos
```

3.7. Producto cartesiano de dos conjuntos

En Haskell

```
-- -----
-- Definir la función
--   producto :: [a] -> [b] -> [(a,b)]
-- tal que (producto xs ys) es el producto cartesiano de xs e ys. Por
-- ejemplo,
--   producto [1,3] [2,4] == [(1,2),(1,4),(3,2),(3,4)]
--
-- Comprobar con QuickCheck que el número de elementos de (producto xs
-- ys) es el producto del número de elementos de xs y de ys.
-- -----
```

```
{-# OPTIONS_GHC -fno-warn-unused-imports #-}
```

```
module Producto_cartesiano_de_dos_conjuntos where
```

```
import Test.QuickCheck
```

```
-- 1ª solución
-- =====
```

```
producto1 :: [a] -> [a] -> [(a,a)]
producto1 xs ys = [(x,y) | x <- xs, y <- ys]
```

```
-- 2ª solución
-- =====
```

```

producto2 :: [a] -> [a] -> [(a,a)]
producto2 [] _ = []
producto2 (x:xs) ys = [(x,y) | y <- ys] ++ producto2 xs ys

-- Comprobación de equivalencia
-- =====

-- La propiedad es
prop_producto :: [Int] -> [Int] -> Bool
prop_producto xs ys =
  producto1 xs ys `iguales` producto2 xs ys

-- (iguales xs ys) se verifica si xs e ys son iguales. Por ejemplo,
--   iguales [3,2,3] [2,3]      == True
--   iguales [3,2,3] [2,3,2]    == True
--   iguales [3,2,3] [2,3,4]    == False
--   iguales [2,3] [4,5]        == False
iguales :: Ord a => [a] -> [a] -> Bool
iguales xs ys =
  subconjunto xs ys && subconjunto ys xs

-- (subconjunto xs ys) se verifica si xs es un subconjunto de ys. por
-- ejemplo,
--   subconjunto [3,2,3] [2,5,3,5] == True
--   subconjunto [3,2,3] [2,5,6,5] == False
subconjunto :: Ord a => [a] -> [a] -> Bool
subconjunto xs ys =
  [x | x <- xs, x `elem` ys] == xs

-- La comprobación es
--   λ> quickCheck prop_producto
--   +++ OK, passed 100 tests.

-- Comparación de eficiencia
-- =====

-- La comparación es
--   λ> length (producto1 [1..4000] [1..4000])
--   16000000
--   (2.33 secs, 1,537,551,208 bytes)

```



```
-- λ> length (producto2 [1..4000] [1..4000])
-- 16000000
-- (2.87 secs, 2,434,095,160 bytes)

-- Comprobación de la propiedad
-- =====

-- La propiedad es
prop_elementos_producto :: [Int] -> [Int] -> Bool
prop_elementos_producto xs ys =
    length (producto1 xs ys) == length xs * length ys

-- La comprobación es
-- λ> quickCheck prop_elementos_producto
-- +++ OK, passed 100 tests.
```

En Python

```
# -----
# Definir la función
# producto : (list[A], list[B]) -> list[tuple[(A, B)]]
# tal que producto(xs, ys) es el producto cartesiano de xs e ys. Por
# ejemplo,
# producto([1, 3], [2, 4]) == [(1, 2), (1, 4), (3, 2), (3, 4)]
#
# Comprobar con Hypothesis que el número de elementos de (producto xs
# ys) es el producto del número de elementos de xs y de ys.
# -----

from sys import setrecursionlimit
from timeit import Timer, default_timer
from typing import TypeVar

from hypothesis import given
from hypothesis import strategies as st

setrecursionlimit(10**6)

A = TypeVar('A')
B = TypeVar('B')
```

1ª solución

=====

```
def producto1(xs: list[A], ys: list[B]) -> list[tuple[A, B]]:
    return [(x, y) for x in xs for y in ys]
```

2ª solución

=====

```
def producto2(xs: list[A], ys: list[B]) -> list[tuple[A, B]]:
    if xs:
        return [(xs[0], y) for y in ys] + producto2(xs[1:], ys)
    return []
```

Comprobación de equivalencia

=====

La propiedad es

```
@given(st.lists(st.integers()),
       st.lists(st.integers()))
```

```
def test_producto(xs: list[int], ys: list[int]) -> None:
    assert sorted(producto1(xs, ys)) == sorted(producto2(xs, ys))
```

La comprobación es

```
# src> poetry run pytest -q producto_cartesiano_de_dos_conjuntos.py
# 1 passed in 0.31s
```

Comparación de eficiencia

=====

```
def tiempo(e: str) -> None:
    """Tiempo (en segundos) de evaluar la expresión e."""
    t = Timer(e, "", default_timer, globals()).timeit(1)
    print(f"{t:0.2f} segundos")
```

La comparación es

```
# >>> tiempo('len(producto1(range(0, 1000), range(0, 500)))')
# 0.03 segundos
# >>> tiempo('len(producto2(range(0, 1000), range(0, 500)))')
```

```
# 2.58 segundos

# Comprobación de la propiedad
# =====

# La propiedad es
@given(st.lists(st.integers()),
      st.lists(st.integers()))
def test_elementos_producto(xs: list[int], ys: list[int]) -> None:
    assert len(producto1(xs, ys)) == len(xs) * len(ys)

# La comprobación es
# src> poetry run pytest -q producto_cartesiano_de_dos_conjuntos.py
# 2 passed in 0.48s
```

3.8. Subconjuntos de un conjunto

En Haskell

```
-- -----
-- Definir la función
--   subconjuntos :: [a] -> [[a]]
-- tal que (subconjuntos xs) es la lista de las subconjuntos de la lista
-- xs. Por ejemplo,
--   λ> subconjuntos [2,3,4]
--   [[2,3,4],[2,3],[2,4],[2],[3,4],[3],[4],[]]
--   λ> subconjuntos [1,2,3,4]
--   [[1,2,3,4],[1,2,3],[1,2,4],[1,2],[1,3,4],[1,3],[1,4],[1],
--     [2,3,4], [2,3], [2,4], [2], [3,4], [3], [4], []]
--
-- Comprobar con QuickChek que el número de elementos de
-- (subconjuntos xs) es 2 elevado al número de elementos de xs.
--
-- Nota. Al hacer la comprobación limitar el tamaño de las pruebas como
-- se indica a continuación
--   quickCheckWith (stdArgs {maxSize=7}) prop_length_subconjuntos
-- -----

{-# OPTIONS_GHC -fno-warn-unused-imports #-}
```

```

module Subconjuntos_de_un_conjunto where

import Data.List (sort, subsequences)
import Test.QuickCheck

-- 1ª solución
-- =====

subconjuntos1 :: [a] -> [[a]]
subconjuntos1 []      = [[]]
subconjuntos1 (x:xs) = [x:ys | ys <- sub] ++ sub
    where sub = subconjuntos1 xs

-- 2ª solución
-- =====

subconjuntos2 :: [a] -> [[a]]
subconjuntos2 []      = [[]]
subconjuntos2 (x:xs) = map (x:) sub ++ sub
    where sub = subconjuntos2 xs

-- 3ª solución
-- =====

subconjuntos3 :: [a] -> [[a]]
subconjuntos3 = subsequences

-- Comprobación de equivalencia
-- =====

-- La propiedad es
prop_subconjuntos :: [Int] -> Bool
prop_subconjuntos xs =
    all (== sort (subconjuntos1 xs))
        [sort (subconjuntos2 xs),
         sort (subconjuntos3 xs)]

-- La comprobación es
--    λ> quickCheckWith (stdArgs {maxSize=7}) prop_subconjuntos
--    +++ OK, passed 100 tests.

```

```
-- Comparación de eficiencia
-- =====

-- La comparación es
--   λ> length (subconjuntos1 [1..23])
--   8388608
--   (2.05 secs, 1,476,991,840 bytes)
--   λ> length (subconjuntos2 [1..23])
--   8388608
--   (0.87 secs, 1,208,555,312 bytes)
--   λ> length (subconjuntos3 [1..23])
--   8388608
--   (0.09 secs, 873,006,608 bytes)

-- Comprobación de la propiedad
-- =====

-- La propiedad es
prop_length_subconjuntos :: [Int] -> Bool
prop_length_subconjuntos xs =
  length (subconjuntos1 xs) == 2 ^ length xs

-- La comprobación es
--   λ> quickCheckWith (stdArgs {maxSize=7}) prop_length_subconjuntos
--   +++ OK, passed 100 tests.
```

En Python

```
# -----
# Definir la función
#   subconjuntos : (list[A]) -> list[list[A]]
# tal que subconjuntos(xs) es la lista de las subconjuntos de la lista
# xs. Por ejemplo,
#   >>> subconjuntos([2, 3, 4])
#   [[2,3,4], [2,3], [2,4], [2], [3,4], [3], [4], []]
#   >>> subconjuntos([1, 2, 3, 4])
#   [[1,2,3,4], [1,2,3], [1,2,4], [1,2], [1,3,4], [1,3], [1,4], [1],
#     [2,3,4], [2,3], [2,4], [2], [3,4], [3], [4], []]
#
```

```

# Comprobar con Hypothesis que el número de elementos de
# (subconjuntos xs) es 2 elevado al número de elementos de xs.
# -----

from itertools import combinations
from sys import setrecursionlimit
from timeit import Timer, default_timer
from typing import TypeVar

from hypothesis import given
from hypothesis import strategies as st
from sympy import FiniteSet

setrecursionlimit(10**6)

A = TypeVar('A')

# 1ª solución
# =====

def subconjuntos1(xs: list[A]) -> list[list[A]]:
    if xs:
        sub = subconjuntos1(xs[1:])
        return [[xs[0]] + ys for ys in sub] + sub
    return [[]]

# 2ª solución
# =====

def subconjuntos2(xs: list[A]) -> list[list[A]]:
    if xs:
        sub = subconjuntos1(xs[1:])
        return list(map((lambda ys: [xs[0]] + ys), sub)) + sub
    return [[]]

# 3ª solución
# =====

def subconjuntos3(xs: list[A]) -> list[list[A]]:
    c = FiniteSet(*xs)

```

```

    return list(map(list, c.powerset()))

# 4ª solución
# =====

def subconjuntos4(xs: list[A]) -> list[list[A]]:
    return [list(ys)
            for r in range(len(xs)+1)
            for ys in combinations(xs, r)]

# Comprobación de equivalencia
# =====

# La propiedad es
@given(st.lists(st.integers(), max_size=5))
def test_subconjuntos(xs: list[int]) -> None:
    ys = list(set(xs))
    r = sorted([sorted(zs) for zs in subconjuntos1(ys)])
    assert sorted([sorted(zs) for zs in subconjuntos2(ys)]) == r
    assert sorted([sorted(zs) for zs in subconjuntos3(ys)]) == r
    assert sorted([sorted(zs) for zs in subconjuntos4(ys)]) == r

# La comprobación es
#   src> poetry run pytest -q subconjuntos_de_un_conjunto.py
#   1 passed in 0.89s

# Comparación de eficiencia
# =====

def tiempo(e: str) -> None:
    """Tiempo (en segundos) de evaluar la expresión e."""
    t = Timer(e, "", default_timer, globals()).timeit(1)
    print(f"{t:0.2f} segundos")

# La comparación es
#   >>> tiempo('subconjuntos1(range(14))')
#   0.00 segundos
#   >>> tiempo('subconjuntos2(range(14))')
#   0.00 segundos
#   >>> tiempo('subconjuntos3(range(14))')
```

```

#      6.01 segundos
#      >>> tiempo('subconjuntos4(range(14))')
#      0.00 segundos
#
#      >>> tiempo('subconjuntos1(range(23))')
#      1.95 segundos
#      >>> tiempo('subconjuntos2(range(23))')
#      2.27 segundos
#      >>> tiempo('subconjuntos4(range(23))')
#      1.62 segundos

# Comprobación de la propiedad
# =====

# La propiedad es
@given(st.lists(st.integers(), max_size=7))
def test_length_subconjuntos(xs: list[int]) -> None:
    assert len(subconjuntos1(xs)) == 2 ** len(xs)

# La comprobación es
#      src> poetry run pytest -q subconjuntos_de_un_conjunto.py
#      2 passed in 0.95s

```

3.9. El algoritmo de Luhn

En Haskell

```
module El_algoritmo_de_Luhn where
```

```

-- -----
-- El objetivo de este ejercicio es estudiar un algoritmo para validar
-- algunos identificadores numéricos como los números de algunas tarjetas
-- de crédito; por ejemplo, las de tipo Visa o Master Card.
--
-- El algoritmo que vamos a estudiar es el [algoritmo de
-- Luhn](https://bit.ly/3DX1llv) consistente en aplicar los siguientes
-- pasos a los dígitos del número de la tarjeta.
--      1. Se invierten los dígitos del número; por ejemplo, [9,4,5,5] se
--          transforma en [5,5,4,9].
--      2. Se duplican los dígitos que se encuentra en posiciones impares

```



```

--      (empezando a contar en 0); por ejemplo, [5,5,4,9] se transforma
--      en [5,10,4,18].
--      3. Se suman los dígitos de cada número; por ejemplo, [5,10,4,18]
--      se transforma en  $5 + (1 + 0) + 4 + (1 + 8) = 19$ .
--      4. Si el último dígito de la suma es 0, el número es válido; y no
--      lo es, en caso contrario.
--
-- A los números válidos, se les llama números de Luhn.
--
-- Definir las siguientes funciones:
--      digitosInv      :: Integer -> [Integer]
--      doblePosImpar  :: [Integer] -> [Integer]
--      sumaDigitos    :: [Integer] -> Integer
--      ultimoDigito   :: Integer -> Integer
--      luhn           :: Integer -> Bool
--
-- tales que
-- + (digitosInv n) es la lista de los dígitos del número n. en orden
--   inverso. Por ejemplo,
--      digitosInv 320274 == [4,7,2,0,2,3]
-- + (doblePosImpar ns) es la lista obtenida doblando los elementos en
--   las posiciones impares (empezando a contar en cero y dejando igual
--   a los que están en posiciones pares. Por ejemplo,
--      doblePosImpar [4,9,5,5]    == [4,18,5,10]
--      doblePosImpar [4,9,5,5,7] == [4,18,5,10,7]
-- + (sumaDigitos ns) es la suma de los dígitos de ns. Por ejemplo,
--      sumaDigitos [10,5,18,4] = 1 + 0 + 5 + 1 + 8 + 4 =
--                               = 19
-- + (ultimoDigito n) es el último dígito de n. Por ejemplo,
--      ultimoDigito 123 == 3
--      ultimoDigito 0  == 0
-- + (luhn n) se verifica si n es un número de Luhn. Por ejemplo,
--      luhn 5594589764218858 == True
--      luhn 1234567898765432 == False
--
-- -----
--
-- Definición de digitosInv
-- =====
digitosInv :: Integer -> [Integer]
digitosInv n = [read [x] | x <- reverse (show n)]

```

-- Nota: En el ejercicio "Dígitos de un número" <https://bit.ly/3Tkhc2T>
 -- se presentan otras definiciones.

-- Definiciones de doblePosImpar
 -- =====

-- 1ª definición

```
doblePosImpar :: [Integer] -> [Integer]
doblePosImpar []      = []
doblePosImpar [x]     = [x]
doblePosImpar (x:y:zs) = x : 2*y : doblePosImpar zs
```

-- 2ª definición

```
doblePosImpar2 :: [Integer] -> [Integer]
doblePosImpar2 (x:y:zs) = x : 2*y : doblePosImpar2 zs
doblePosImpar2 xs      = xs
```

-- 3ª definición

```
doblePosImpar3 :: [Integer] -> [Integer]
doblePosImpar3 xs = [f n x | (n,x) <- zip [0..] xs]
  where f n x | odd n      = 2*x
              | otherwise = x
```

-- Definiciones de sumaDigitos
 -- =====

```
sumaDigitos :: [Integer] -> Integer
sumaDigitos ns = sum [sum (digitosInv n) | n <- ns]
```

-- Nota: En el ejercicio "Suma de los dígitos de un número"
 -- <https://bit.ly/3U4u7WR> se presentan otras definiciones.

-- Definición de ultimoDigito
 -- =====

```
ultimoDigito :: Integer -> Integer
ultimoDigito n = n `rem` 10
```

-- Definiciones de luhn

```
-- =====

-- 1ª definición
luhn1 :: Integer -> Bool
luhn1 n =
    ultimoDigito (sumaDigitos (doblesPosImpar (digitosInv n))) == 0

-- 2ª definición
luhn2 :: Integer -> Bool
luhn2 =
    (==0) . ultimoDigito . sumaDigitos . doblePosImpar . digitosInv
```

En Python

```
# -----
# El objetivo de este ejercicio es estudiar un algoritmo para validar
# algunos identificadores numéricos como los números de algunas tarjetas
# de crédito; por ejemplo, las de tipo Visa o Master Card.
#
# El algoritmo que vamos a estudiar es el [algoritmo de
# Luhn](https://bit.ly/3DX1llv) consistente en aplicar los siguientes
# pasos a los dígitos del número de la tarjeta.
# 1. Se invierten los dígitos del número; por ejemplo, [9,4,5,5] se
#     transforma en [5,5,4,9].
# 2. Se duplican los dígitos que se encuentra en posiciones impares
#     (empezando a contar en 0); por ejemplo, [5,5,4,9] se transforma
#     en [5,10,4,18].
# 3. Se suman los dígitos de cada número; por ejemplo, [5,10,4,18]
#     se transforma en  $5 + (1 + 0) + 4 + (1 + 8) = 19$ .
# 4. Si el último dígito de la suma es 0, el número es válido; y no
#     lo es, en caso contrario.
#
# A los números válidos, se les llama números de Luhn.
#
# Definir las siguientes funciones:
# digitosInv      : (int) -> list[int]
# doblePosImpar  : (list[int]) -> list[int]
# sumaDigitos    : (list[int]) -> int
# ultimoDigito   : (int) -> int
# luhn           : (int) -> bool
```

```

# tales que
# + digitosInv(n) es la lista de los dígitos del número n. en orden
#   inverso. Por ejemplo,
#       digitosInv(320274) == [4,7,2,0,2,3]
# + doblePosImpar(ns) es la lista obtenida doblando los elementos en
#   las posiciones impares (empezando a contar en cero y dejando igual
#   a los que están en posiciones pares. Por ejemplo,
#       doblePosImpar([4,9,5,5]) == [4,18,5,10]
#       doblePosImpar([4,9,5,5,7]) == [4,18,5,10,7]
# + sumaDigitos(ns) es la suma de los dígitos de ns. Por ejemplo,
#       sumaDigitos([10,5,18,4]) = 1 + 0 + 5 + 1 + 8 + 4 =
#                                   = 19
# + ultimoDigito(n) es el último dígito de n. Por ejemplo,
#       ultimoDigito(123) == 3
#       ultimoDigito(0) == 0
# + luhn(n) se verifica si n es un número de Luhn. Por ejemplo,
#       luhn(5594589764218858) == True
#       luhn(1234567898765432) == False
# -----

# Definición de digitosInv
# =====

def digitosInv(n: int) -> list[int]:
    return [int(x) for x in reversed(str(n))]

# Nota: En el ejercicio "Dígitos de un número" https://bit.ly/3Tkhc2T
# se presentan otras definiciones.

# Definiciones de doblePosImpar
# =====

# 1ª definición
def doblePosImpar(xs: list[int]) -> list[int]:
    if len(xs) <= 1:
        return xs
    return [xs[0]] + [2*xs[1]] + doblePosImpar(xs[2:])

# 2ª definición
def doblePosImpar2(xs: list[int]) -> list[int]:

```

```

def f(n: int, x: int) -> int:
    if n % 2 == 1:
        return 2 * x
    return x
return [f(n, x) for (n, x) in enumerate(xs)]

# Definiciones de sumaDigitos
# =====

def sumaDigitos(ns: list[int]) -> int:
    return sum((sum(digitosInv(n)) for n in ns))

# Nota: En el ejercicio "Suma de los dígitos de un número"
# https://bit.ly/3U4u7WR se presentan otras definiciones.

# Definición de ultimoDigito
# =====

def ultimoDigito(n: int) -> int:
    return n % 10

# Definiciones de luhn
# =====

def luhn(n: int) -> bool:
    return ultimoDigito(sumaDigitos(doblePosImpar(digitosInv(n)))) == 0

```

3.10. Números de Lychrel

En Haskell

```

-- -----
-- Un [número de Lychrel](http://bit.ly/2X4DzMf) es un número natural
-- para el que nunca se obtiene un capicúa mediante el proceso de
-- invertir las cifras y sumar los dos números. Por ejemplo, los
-- siguientes números no son números de Lychrel:
-- + 56, ya que en un paso se obtiene un capicúa: 56+65=121.
-- + 57, ya que en dos pasos se obtiene un capicúa: 57+75=132,
--   132+231=363
-- + 59, ya que en dos pasos se obtiene un capicúa: 59+95=154,

```

```
-- 154+451=605, 605+506=1111
-- + 89, ya que en 24 pasos se obtiene un capicúa.
-- En esta serie de ejercicios vamos a buscar el primer número de
-- Lychrel.
```

```
module Numeros_de_Lychrel where
import Test.QuickCheck
```

```
-- -----
-- Ejercicio 1. Definir la función
--   esCapicua :: Integer -> Bool
-- tal que (esCapicua x) se verifica si x es capicúa. Por ejemplo,
--   esCapicua 252 == True
--   esCapicua 253 == False
-- -----
```

```
esCapicua :: Integer -> Bool
esCapicua x = x' == reverse x'
  where x' = show x
```

```
-- -----
-- Ejercicio 2. Definir la función
--   inverso :: Integer -> Integer
-- tal que (inverso x) es el número obtenido escribiendo las cifras de x
-- en orden inverso. Por ejemplo,
--   inverso 253 == 352
-- -----
```

```
inverso :: Integer -> Integer
inverso = read . reverse . show
```

```
-- -----
-- Ejercicio 3. Definir la función
--   siguiente :: Integer -> Integer
-- tal que (siguiente x) es el número obtenido sumándole a x su
-- inverso. Por ejemplo,
--   siguiente 253 == 605
-- -----
```

```
siguiente :: Integer -> Integer
```

```
siguiente x = x + inverso x
```

```
-- -----  
-- Ejercicio 4. Definir la función
```

```
--   busquedaDeCapicua :: Integer -> [Integer]
```

```
-- tal que (busquedaDeCapicua x) es la lista de los números tal que el  
-- primero es x, el segundo es (siguiente de x) y así sucesivamente
```

```
-- hasta que se alcanza un capicúa. Por ejemplo,
```

```
--   busquedaDeCapicua 253 == [253,605,1111]
```

```
-- -----  
busquedaDeCapicua :: Integer -> [Integer]
```

```
busquedaDeCapicua x | esCapicua x = [x]
```

```
                  | otherwise   = x : busquedaDeCapicua (siguiente x)
```

```
-- -----  
-- Ejercicio 5. Definir la función
```

```
--   capicuaFinal :: Integer -> Integer
```

```
-- tal que (capicuaFinal x) es la capicúa con la que termina la búsqueda  
-- de capicúa a partir de x. Por ejemplo,
```

```
--   capicuaFinal 253 == 1111
```

```
-- -----  
capicuaFinal :: Integer -> Integer
```

```
capicuaFinal x = last (busquedaDeCapicua x)
```

```
-- -----  
-- Ejercicio 6. Definir la función
```

```
--   orden :: Integer -> Integer
```

```
-- tal que (orden x) es el número de veces que se repite el proceso de  
-- calcular el inverso a partir de x hasta alcanzar un número
```

```
-- capicúa. Por ejemplo,
```

```
--   orden 253 == 2
```

```
-- -----  
orden :: Integer -> Integer
```

```
orden x | esCapicua x = 0
```

```
      | otherwise   = 1 + orden (siguiente x)
```

```

-- -----
-- Ejercicio 7. Definir la función
--   ordenMayor :: Integer -> Integer -> Bool
-- tal que (ordenMayor x n) se verifica si el orden de x es mayor o
-- igual que n. Dar la definición sin necesidad de evaluar el orden de
-- x. Por ejemplo,
--   λ> ordenMayor 1186060307891929990 2
--   True
--   λ> orden 1186060307891929990
--   261
-- -----

```

```

ordenMayor :: Integer -> Integer -> Bool
ordenMayor x n | esCapicua x == 0
               | n <= 0          = True
               | otherwise      = ordenMayor (siguiente x) (n-1)

```

```

-- -----
-- Ejercicio 8. Definir la función
--   ordenEntre :: Integer -> Integer -> [Integer]
-- tal que (ordenEntre m n) es la lista de los elementos cuyo orden es
-- mayor o igual que m y menor que n. Por ejemplo,
--   take 5 (ordenEntre 10 11) == [829,928,9059,9149,9239]
-- -----

```

```

ordenEntre :: Integer -> Integer -> [Integer]
ordenEntre m n = [x | x <- [1..], ordenMayor x m, not (ordenMayor x n)]

```

```

-- -----
-- Ejercicio 9. Definir la función
--   menorDeOrdenMayor :: Integer -> Integer
-- tal que (menorDeOrdenMayor n) es el menor elemento cuyo orden es
-- mayor que n. Por ejemplo,
--   menorDeOrdenMayor 2    == 19
--   menorDeOrdenMayor 20   == 89
-- -----

```

```

menorDeOrdenMayor :: Integer -> Integer
menorDeOrdenMayor n = head [x | x <- [1..], ordenMayor x n]

```



```

-- -----
-- Ejercicio 10. Definir la función
--   menoresdDeOrdenMayor :: Integer -> [(Integer,Integer)]
-- tal que (menoresdDeOrdenMayor m) es la lista de los pares (n,x) tales
-- que n es un número entre 1 y m y x es el menor elemento de orden
-- mayor que n. Por ejemplo,
--   menoresdDeOrdenMayor 5 == [(1,10),(2,19),(3,59),(4,69),(5,79)]
-- -----

menoresdDeOrdenMayor :: Integer -> [(Integer,Integer)]
menoresdDeOrdenMayor m = [(n,menorDeOrdenMayor n) | n <- [1..m]]

-- -----
-- Ejercicio 11. A la vista de los resultados de (menoresdDeOrdenMayor 5)
-- conjeturar sobre la última cifra de menorDeOrdenMayor.
-- -----

-- Solución: La conjetura es que para n mayor que 1, la última cifra de
-- (menorDeOrdenMayor n) es 9.

-- -----
-- Ejercicio 12. Decidir con QuickCheck la conjetura.
-- -----

-- La conjetura es
prop_menorDeOrdenMayor :: Integer -> Property
prop_menorDeOrdenMayor n =
  n > 1 ==> menorDeOrdenMayor n `mod` 10 == 9

-- La comprobación es
--   λ> quickCheck prop_menorDeOrdenMayor
--   *** Failed! Falsifiable (after 22 tests and 2 shrinks):
--   25

-- Se puede comprobar que 25 es un contraejemplo,
--   λ> menorDeOrdenMayor 25
--   196

-- -----
-- Ejercicio 13. Calcular (menoresdDeOrdenMayor 50)

```

```

-- -----
-- Solución: El cálculo es
--   λ> menoresDeOrdenMayor 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 (menoresDeOrdenMayor 50), conjeturar el
-- orden de 196.
-- -----
-- Solución: El orden de 196 es infinito y, por tanto, 196 es un número
-- del Lychrel.
-- -----
-- Ejercicio 15. Comprobar con QuickCheck la conjetura sobre el orden de
-- 196.
-- -----
-- La propiedad es
prop_ordenDe196 :: Integer -> Bool
prop_ordenDe196 n =
  ordenMayor 196 n

-- La comprobación es
--   λ> quickCheck prop_ordenDe196
--   +++ OK, passed 100 tests.

```

En Python

```

# -----
# Un [número de Lychrel](http://bit.ly/2X4DzMf) es un número natural
# para el que nunca se obtiene un capicúa mediante el proceso de
# invertir las cifras y sumar los dos números. Por ejemplo, los

```

```
# siguientes números no son números de Lychrel:
# + 56, ya que en un paso se obtiene un capicúa: 56+65=121.
# + 57, ya que en dos pasos se obtiene un capicúa: 57+75=132,
#   132+231=363
# + 59, ya que en dos pasos se obtiene un capicúa: 59+95=154,
#   154+451=605, 605+506=1111
# + 89, ya que en 24 pasos se obtiene un capicúa.
# En esta serie de ejercicios vamos a buscar el primer número de
# Lychrel.
# -----

from itertools import islice
from sys import setrecursionlimit
from typing import Generator, Iterator

from hypothesis import given, settings
from hypothesis import strategies as st

setrecursionlimit(10**6)

# -----
# Ejercicio 1. Definir la función
#   esCapicua : (int) -> bool
# tal que esCapicua(x) se verifica si x es capicúa. Por ejemplo,
#   esCapicua(252) == True
#   esCapicua(253) == False
# -----

def esCapicua(x: int) -> bool:
    return x == int(str(x)[::-1])

# -----
# Ejercicio 2. Definir la función
#   inverso : (int) -> int
# tal que inverso(x) es el número obtenido escribiendo las cifras de x
# en orden inverso. Por ejemplo,
#   inverso(253) == 352
# -----

def inverso(x: int) -> int:
```

```

    return int(str(x)[::-1])

# -----
# Ejercicio 3. Definir la función
# siguiente : (int) -> int
# tal que siguiente(x) es el número obtenido sumándole a x su
# inverso. Por ejemplo,
# siguiente(253) == 605
# -----

def siguiente(x: int) -> int:
    return x + inverso(x)

# -----
# Ejercicio 4. Definir la función
# busquedaDeCapicua : (int) -> list[int]
# tal que busquedaDeCapicua(x) es la lista de los números tal que el
# primero es x, el segundo es (siguiente de x) y así sucesivamente
# hasta que se alcanza un capicúa. Por ejemplo,
# busquedaDeCapicua(253) == [253,605,1111]
# -----

def busquedaDeCapicua(x: int) -> list[int]:
    if esCapicua(x):
        return [x]
    return [x] + busquedaDeCapicua(siguiente(x))

# -----
# Ejercicio 5. Definir la función
# capicuaFinal : (int) -> int
# tal que (capicuaFinal x) es la capicúa con la que termina la búsqueda
# de capicúa a partir de x. Por ejemplo,
# capicuaFinal(253) == 1111
# -----

def capicuaFinal(x: int) -> int:
    return busquedaDeCapicua(x)[-1]

# -----
# Ejercicio 6. Definir la función

```

```
#    orden : (int) -> int
# tal que orden(x) es el número de veces que se repite el proceso de
# calcular el inverso a partir de x hasta alcanzar un número capicúa.
# Por ejemplo,
#    orden(253) == 2
# -----
```

```
def orden(x: int) -> int:
    if esCapicua(x):
        return 0
    return 1 + orden(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

menoresdDeOrdenMayor : (int) -> list[tuple[int, int]]

tal que (menoresdDeOrdenMayor m) es la lista de los pares (n,x) tales que n es un número entre 1 y m y x es el menor elemento de orden mayor que n. Por ejemplo,

```
# menoresdDeOrdenMayor(5) == [(1,10),(2,19),(3,59),(4,69),(5,79)]
```

```
# -----
```

```
def menoresdDeOrdenMayor(m: int) -> list[tuple[int, int]]:
```

```
    return [(n, menorDeOrdenMayor(n)) for n in range(1, m + 1)]
```

```
# -----
```

Ejercicio 11. A la vista de los resultados de (menoresdDeOrdenMayor 5)

```

# conjeturar sobre la última cifra de menorDeOrdenMayor.
# -----

# Solución: La conjetura es que para n mayor que 1, la última cifra de
# (menorDeOrdenMayor n) es 9.

# -----
# Ejercicio 12. Decidir con Hypothesis la conjetura.
# -----

# La conjetura es
# @given(st.integers(min_value=2, max_value=200))
# def test_menorDeOrdenMayor(n: int) -> None:
#     assert menorDeOrdenMayor(n) % 10 == 9

# La comprobación es
# src> poetry run pytest -q numeros_de_Lychrel.py
# E      assert (196 % 10) == 9
# E      + where 196 = menorDeOrdenMayor(25)
# E      Falsifying example: test_menorDeOrdenMayor(
# E          n=25,
# E      )

# Se puede comprobar que 25 es un contraejemplo,
# >>> menorDeOrdenMayor(25)
# 196

# -----
# Ejercicio 13. Calcular menoresdDeOrdenMayor(50)
# -----

# Solución: El cálculo es
# λ> menoresdDeOrdenMayor 50
# [(1,10),(2,19),(3,59),(4,69),(5,79),(6,79),(7,89),(8,89),(9,89),
#  (10,89),(11,89),(12,89),(13,89),(14,89),(15,89),(16,89),(17,89),
#  (18,89),(19,89),(20,89),(21,89),(22,89),(23,89),(24,89),(25,196),
#  (26,196),(27,196),(28,196),(29,196),(30,196),(31,196),(32,196),
#  (33,196),(34,196),(35,196),(36,196),(37,196),(38,196),(39,196),
#  (40,196),(41,196),(42,196),(43,196),(44,196),(45,196),(46,196),
#  (47,196),(48,196),(49,196),(50,196)]

```

```

# -----
# Ejercicio 14. A la vista de menoresDeOrdenMayor(50), conjeturar el
# orden de 196.
# -----

# Solución: El orden de 196 es infinito y, por tanto, 196 es un número
# del Lychrel.

# -----
# Ejercicio 15. Comprobar con Hypothesis la conjetura sobre el orden de
# 196.
# -----

# La propiedad es
@settings(deadline=None)
@given(st.integers(min_value=2, max_value=5000))
def test_ordenDe196(n: int) -> None:
    assert ordenMayor(196, n)

# La comprobación es
#   src> poetry run pytest -q numeros_de_Lychrel.py
#   1 passed in 7.74s

```

3.11. Suma de los dígitos de una cadena

En Haskell

```

-- -----
-- Definir la función
--   sumaDigitos :: String -> Int
-- tal que (sumaDigitos xs) es la suma de los dígitos de la cadena
-- xs. Por ejemplo,
--   sumaDigitos "SE 2431 X" == 10
-- -----

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module Suma_de_digitos_de_cadena where

```



```
import Data.Char (digitToInt, isDigit)
import Test.QuickCheck

-- 1ª solución
-- =====

sumaDigitos1 :: String -> Int
sumaDigitos1 xs = sum [digitToInt x | x <- xs, isDigit x]

-- 2ª solución
-- =====

sumaDigitos2 :: String -> Int
sumaDigitos2 [] = 0
sumaDigitos2 (x:xs)
  | isDigit x = digitToInt x + sumaDigitos2 xs
  | otherwise = sumaDigitos2 xs

-- 3ª solución
-- =====

sumaDigitos3 :: String -> Int
sumaDigitos3 xs = sum (map digitToInt (filter isDigit xs))

-- 4ª solución
-- =====

sumaDigitos4 :: String -> Int
sumaDigitos4 = sum . map digitToInt . filter isDigit

-- Comprobación de equivalencia
-- =====

-- La propiedad es
prop_sumaDigitos :: String -> Bool
prop_sumaDigitos xs =
  all (== sumaDigitos1 xs)
    [sumaDigitos2 xs,
     sumaDigitos3 xs,
     sumaDigitos4 xs]
```

```
-- La comprobación es
--   λ> quickCheck prop_sumaDigitos
--   +++ OK, passed 100 tests.

-- Comparación de eficiencia
--   =====

-- La comparación es
--   λ> sumaDigitos1 (take (4*10^6) (cycle "ab12"))
--   30000000
--   (1.92 secs, 819,045,328 bytes)
--   λ> sumaDigitos2 (take (4*10^6) (cycle "ab12"))
--   30000000
--   (1.79 secs, 856,419,112 bytes)
--   λ> sumaDigitos3 (take (4*10^6) (cycle "ab12"))
--   30000000
--   (0.62 secs, 723,045,296 bytes)
--   λ> sumaDigitos4 (take (4*10^6) (cycle "ab12"))
--   30000000
--   (0.63 secs, 723,045,552 bytes)
```

En Python

```
# -----
# Definir la función
#   sumaDigitos : (str) -> int
# tal que sumaDigitos(xs) es la suma de los dígitos de la cadena
# xs. Por ejemplo,
#   sumaDigitos("SE 2431 X") == 10
# -----

from sys import setrecursionlimit
from timeit import Timer, default_timer

from hypothesis import given
from hypothesis import strategies as st

setrecursionlimit(10**6)
```

```
# 1ª solución
# =====

def sumaDigitos1(xs: str) -> int:
    return sum((int(x) for x in xs if x.isdigit()))

# 2ª solución
# =====

def sumaDigitos2(xs: str) -> int:
    if xs:
        if xs[0].isdigit():
            return int(xs[0]) + sumaDigitos2(xs[1:])
        return sumaDigitos2(xs[1:])
    return 0

# 3ª solución
# =====

def sumaDigitos3(xs: str) -> int:
    r = 0
    for x in xs:
        if x.isdigit():
            r = r + int(x)
    return r

# Comprobación de equivalencia
# =====

# La propiedad es
@given(st.text())
def test_sumaDigitos(xs: str) -> None:
    r = sumaDigitos1(xs)
    assert sumaDigitos2(xs) == r
    assert sumaDigitos3(xs) == r

# La comprobación es
# src> poetry run pytest -q suma_de_digitos_de_cadena.py
# 1 passed in 0.41s
```

```
# Comparación de eficiencia
# =====

def tiempo(e: str) -> None:
    """Tiempo (en segundos) de evaluar la expresión e."""
    t = Timer(e, "", default_timer, globals()).timeit(1)
    print(f"{t:0.2f} segundos")

# La comparación es
# >>> tiempo('sumaDigitos1("ab12"*5000)')
# 0.00 segundos
# >>> tiempo('sumaDigitos2("ab12"*5000)')
# 0.02 segundos
# >>> tiempo('sumaDigitos3("ab12"*5000)')
# 0.00 segundos
#
# >>> tiempo('sumaDigitos1("ab12"*(5*10**6))')
# 1.60 segundos
# >>> tiempo('sumaDigitos3("ab12"*(5*10**6))')
# 1.83 segundos
```

3.12. Primera en mayúscula y restantes en minúscula

En Haskell

```
-- -----
-- Definir la función
--   mayusculaInicial :: String -> String
-- tal que (mayusculaInicial xs) es la palabra xs con la letra inicial
-- en mayúscula y las restantes en minúsculas. Por ejemplo,
--   mayusculaInicial "sEviLLa" == "Sevilla"
--   mayusculaInicial ""        == ""
-- -----

{-# OPTIONS_GHC -fno-warn-unused-imports #-}
```

module Mayuscula_inicial where

```

import Data.Char (toUpper, toLower)
import Test.QuickCheck

-- 1ª solución
-- =====

mayusculaInicial1 :: String -> String
mayusculaInicial1 [] = []
mayusculaInicial1 (x:xs) = toUpper x : [toLower y | y <- xs]

-- 2ª solución
-- =====

mayusculaInicial2 :: String -> String
mayusculaInicial2 [] = []
mayusculaInicial2 (x:xs) = toUpper x : aux xs
  where aux (y:ys) = toLower y : aux ys
        aux [] = []

-- 3ª solución
-- =====

mayusculaInicial3 :: String -> String
mayusculaInicial3 [] = []
mayusculaInicial3 (x:xs) = toUpper x : map toLower xs

-- Comprobación de equivalencia
-- =====

-- La propiedad es
prop_mayusculaInicial :: String -> Bool
prop_mayusculaInicial xs =
  all (== mayusculaInicial1 xs)
    [mayusculaInicial2 xs,
     mayusculaInicial3 xs]

-- La comprobación es
-- λ> quickCheck prop_mayusculaInicial
-- +++ OK, passed 100 tests.

```

```
-- Comparación de eficiencia
-- =====

-- La comparación es
-- λ> length (mayusculaInicial1 (take (10^7) (cycle "aA")))
-- 100000000
-- (2.22 secs, 1,680,592,240 bytes)
-- λ> length (mayusculaInicial2 (take (10^7) (cycle "aA")))
-- 100000000
-- (2.57 secs, 2,240,592,192 bytes)
-- λ> length (mayusculaInicial3 (take (10^7) (cycle "aA")))
-- 100000000
-- (0.16 secs, 1,440,592,192 bytes)
```

En Python

```
# -----
# Definir la función
# mayusculaInicial : (str) -> str
# tal que mayusculaInicial(xs) es la palabra xs con la letra inicial
# en mayúscula y las restantes en minúsculas. Por ejemplo,
# mayusculaInicial("sEviLLa") == "Sevilla"
# mayusculaInicial("") == ""
# -----

from sys import setrecursionlimit
from timeit import Timer, default_timer

from hypothesis import given
from hypothesis import strategies as st

setrecursionlimit(10**6)

# 1ª solución
# =====

def mayusculaInicial1(xs: str) -> str:
    if xs:
        return "".join([xs[0].upper()] + [y.lower() for y in xs[1:]])
    return ""
```

```

# 2ª solución
# =====

def mayusculaInicial2(xs: str) -> str:
    def aux(ys: str) -> str:
        if ys:
            return ys[0].lower() + aux(ys[1:])
        return ""
    if xs:
        return "".join(xs[0].upper() + aux(xs[1:]))
    return ""

# 3ª solución
# =====

def mayusculaInicial3(xs: str) -> str:
    if xs:
        return "".join([xs[0].upper()] + list(map(str.lower, xs[1:])))
    return ""

# 4ª solución
# =====

def mayusculaInicial4(xs: str) -> str:
    return xs.capitalize()

# Comprobación de equivalencia
# =====

# La propiedad es
@given(st.text())
def test_mayusculaInicial(xs: str) -> None:
    r = mayusculaInicial1(xs)
    assert mayusculaInicial2(xs) == r
    assert mayusculaInicial3(xs) == r

# La comprobación es
# src> poetry run pytest -q mayuscula_inicial.py
# 1 passed in 0.26s

```

```
# Comparación de eficiencia
# =====

def tiempo(e: str) -> None:
    """Tiempo (en segundos) de evaluar la expresión e."""
    t = Timer(e, "", default_timer, globals()).timeit(1)
    print(f"{t:0.2f} segundos")

# La comparación es
# >>> tiempo('len(mayusculaInicial1("aB"*(10**7)))')
# 1.92 segundos
# >>> tiempo('len(mayusculaInicial2("aB"*(10**7)))')
# Process Python terminado (killed)
# >>> tiempo('len(mayusculaInicial3("aB"*(10**7)))')
# 1.59 segundos
# >>> tiempo('len(mayusculaInicial4("aB"*(10**7)))')
# 0.13 segundos
```

3.13. Mayúsculas iniciales

En Haskell

```
-- -----
-- Se consideran las siguientes reglas de mayúsculas iniciales para los
-- títulos:
-- + la primera palabra comienza en mayúscula y
-- + todas las palabras que tienen 4 letras como mínimo empiezan con
-- mayúsculas
--
-- Definir la función
-- titulo :: [String] -> [String]
-- tal que (titulo ps) es la lista de las palabras de ps con
-- las reglas de mayúsculas iniciales de los títulos. Por ejemplo,
-- λ> titulo ["eL", "arTE", "DE", "La", "proGraMacion"]
-- ["El", "Arte", "de", "la", "Programacion"]
-- -----

{-# OPTIONS_GHC -fno-warn-unused-imports #-}
```



```

module Mayusculas_iniciales where

import Data.Char (toUpper, toLower)
import Test.QuickCheck

-- 1ª solución
-- =====

titulo1 :: [String] -> [String]
titulo1 []      = []
titulo1 (p:ps) = mayusculaInicial p : [transforma q | q <- ps]

-- (mayusculaInicial xs) es la palabra xs con la letra inicial
-- en mayúscula y las restantes en minúsculas. Por ejemplo,
--   mayusculaInicial "sEviLLa" == "Sevilla"
mayusculaInicial :: String -> String
mayusculaInicial []      = []
mayusculaInicial (x:xs) = toUpper x : [toLower y | y <- xs]

-- (transforma p) es la palabra p con mayúscula inicial si su longitud
-- es mayor o igual que 4 y es p en minúscula en caso contrario
transforma :: String -> String
transforma p | length p >= 4 = mayusculaInicial p
              | otherwise    = minuscula p

-- (minuscula xs) es la palabra xs en minúscula.
minuscula :: String -> String
minuscula xs = [toLower x | x <- xs]

-- 2ª solución
-- =====

titulo2 :: [String] -> [String]
titulo2 []      = []
titulo2 (p:ps) = mayusculaInicial p : aux ps
  where aux []      = []
        aux (q:qs) = transforma q : aux qs

-- 3ª solución
-- =====

```

```

titulo3 :: [String] -> [String]
titulo3 []      = []
titulo3 (p:ps) = mayusculaInicial p : map transforma ps

-- Comprobación de equivalencia
-- =====

-- La propiedad es
prop_titulo :: [String] -> Bool
prop_titulo xs =
  all (== titulo1 xs)
    [titulo2 xs,
     titulo3 xs]

-- La comprobación es
--   λ> quickCheck prop_titulo
--   +++ OK, passed 100 tests.

-- Comparación de eficiencia
-- =====

-- La comparación es
--   λ> length (titulo1 (take (10^7) (cycle ["h0y","Es","juEves","dE","Noviembre
--   100000000
--   (2.17 secs, 1,680,592,512 bytes)
--   λ> length (titulo2 (take (10^7) (cycle ["h0y","Es","juEves","dE","Noviembre
--   100000000
--   (2.45 secs, 2,240,592,464 bytes)
--   λ> length (titulo3 (take (10^7) (cycle ["h0y","Es","juEves","dE","Noviembre
--   100000000
--   (0.16 secs, 1,440,592,464 bytes)

```

En Python

```

# -----
# Se consideran las siguientes reglas de mayúsculas iniciales para los
# títulos:
# + la primera palabra comienza en mayúscula y
# + todas las palabras que tienen 4 letras como mínimo empiezan con

```

```

# mayúsculas
#
# Definir la función
# titulo : (list[str]) -> list[str]
# tal que titulo(ps) es la lista de las palabras de ps con
# las reglas de mayúsculas iniciales de los títulos. Por ejemplo,
# >>> titulo(["eL", "arTE", "DE", "La", "proGraMacion"])
# ["El", "Arte", "de", "la", "Programacion"]
# -----

from sys import setrecursionlimit
from timeit import Timer, default_timer

from hypothesis import given
from hypothesis import strategies as st

setrecursionlimit(10**6)

# 1ª solución
# =====

# (mayusculaInicial xs) es la palabra xs con la letra inicial
# en mayúscula y las restantes en minúsculas. Por ejemplo,
# mayusculaInicial("sEviLLa") == "Sevilla"
def mayusculaInicial(xs: str) -> str:
    return xs.capitalize()

# (minuscuala xs) es la palabra xs en minúscula.
def minuscuala(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 minuscuala(p)

def titulo1(ps: list[str]) -> list[str]:
    if ps:

```

```

        return [mayusculaInicial(ps[0])] + [transforma(q) for q in ps[1:]]
    return []

# 2ª solución
# =====

def titulo2(ps: list[str]) -> list[str]:
    def aux(qs: list[str]) -> list[str]:
        if qs:
            return [transforma(qs[0])] + aux(qs[1:])
        return []
    if ps:
        return [mayusculaInicial(ps[0])] + aux(ps[1:])
    return []

# 3ª solución
# =====

def titulo3(ps: list[str]) -> list[str]:
    if ps:
        return [mayusculaInicial(ps[0])] + list(map(transforma, ps[1:]))
    return []

# Comprobación de equivalencia
# =====

# La propiedad es
@given(st.lists(st.text()))
def test_titulo(ps: list[str]) -> None:
    r = titulo1(ps)
    assert titulo2(ps) == r
    assert titulo3(ps) == r

# La comprobación es
#   src> poetry run pytest -q mayusculas_iniciales.py
#   1 passed in 0.55s

# Comparación de eficiencia
# =====

```

```
def tiempo(e: str) -> None:
    """Tiempo (en segundos) de evaluar la expresión e."""
    t = Timer(e, "", default_timer, globals()).timeit(1)
    print(f"{t:0.2f} segundos")

# La comparación es
# >>> tiempo('titulo1(["eL","arTE","DE","La","proGraMacion "]*1900)')
# 0.00 segundos
# >>> tiempo('titulo2(["eL","arTE","DE","La","proGraMacion "]*1900)')
# 0.30 segundos
# >>> tiempo('titulo3(["eL","arTE","DE","La","proGraMacion "]*1900)')
# 0.00 segundos
#
# >>> tiempo('titulo1(["eL","arTE","DE","La","proGraMacion "]*(2*10**6))')
# 2.93 segundos
# >>> tiempo('titulo3(["eL","arTE","DE","La","proGraMacion "]*(2*10**6))')
# 2.35 segundos
```

3.14. Posiciones de un carácter en una cadena

En Haskell

```
-- -----
-- Definir la función
-- posiciones :: Char -> String -> [Int]
-- tal que (posiciones x ys) es la lista de la posiciones del carácter x
-- en la cadena ys. Por ejemplo,
-- posiciones 'a' "Salamamca" == [1,3,5,8]
-- -----

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module Posiciones_de_un_caracter_en_una_cadena where

import Data.List (elemIndices)
import Test.QuickCheck

-- 1ª solución
-- =====
```

```

posiciones1 :: Char -> String -> [Int]
posiciones1 x ys = [n | (y,n) <- zip ys [0..], y == x]

-- 2ª solución
-- =====

posiciones2 :: Char -> String -> [Int]
posiciones2 x ys = aux x ys 0
  where
    aux _ [] _ = []
    aux b (a:as) n | a == b    = n : aux b as (n+1)
                   | otherwise = aux b as (n+1)

-- 3ª solución
-- =====

posiciones3 :: Char -> String -> [Int]
posiciones3 = elemIndices

-- Comprobación de equivalencia
-- =====

-- La propiedad es
prop_posiciones :: Char -> String -> Bool
prop_posiciones x ys =
  all (== posiciones1 x ys)
    [posiciones2 x ys,
     posiciones3 x ys]

-- La comprobación es
--   λ> quickCheck prop_posiciones
--   +++ OK, passed 100 tests.

-- Comparación de eficiencia
-- =====

-- La comparación es
--   λ> length (posiciones1 'a' (take (6*10^6) (cycle "abc")))
--   2000000
--   (2.48 secs, 1,680,591,672 bytes)

```

```
-- λ> length (posiciones2 'a' (take (6*10^6) (cycle "abc")))
-- 20000000
-- (2.98 secs, 1,584,591,720 bytes)
-- λ> length (posiciones3 'a' (take (6*10^6) (cycle "abc")))
-- 20000000
-- (0.11 secs, 496,591,600 bytes)
```

En Python

```
# -----
# Definir la función
# posiciones : (str, str) -> list[int]
# tal que (posiciones x ys) es la lista de la posiciones del carácter x
# en la cadena ys. Por ejemplo,
# posiciones('a', "Salamamca") == [1,3,5,8]
# -----

from sys import setrecursionlimit
from timeit import Timer, default_timer

from hypothesis import given
from hypothesis import strategies as st

setrecursionlimit(10**6)

# -- 1ª solución
# -- =====

def posiciones1(x: str, ys: str) -> list[int]:
    return [n for (n, y) in enumerate(ys) if y == x]

# -- 2ª solución
# -- =====

def posiciones2(x: str, ys: str) -> list[int]:
    def aux(a: str, bs: str, n: int) -> list[int]:
        if bs:
            if a == bs[0]:
                return [n] + aux(a, bs[1:], n + 1)
            return aux(a, bs[1:], n + 1)
```

```

        return []
    return aux(x, ys, 0)

# -- 3ª solución
# -- =====

def posiciones3(x: str, ys: str) -> list[int]:
    r = []
    for n, y in enumerate(ys):
        if x == y:
            r.append(n)
    return r

# Comprobación de equivalencia
# =====

# La propiedad es
@given(st.text(), st.text())
def test_posiciones(x: str, ys: str) -> None:
    r = posiciones1(x, ys)
    assert posiciones2(x, ys) == r
    assert posiciones3(x, ys) == r

# La comprobación es
# src> poetry run pytest -q posiciones_de_un_caracter_en_una_cadena.py
# 1 passed in 0.29s

# Comparación de eficiencia
# =====

def tiempo(e: str) -> None:
    """Tiempo (en segundos) de evaluar la expresión e."""
    t = Timer(e, "", default_timer, globals()).timeit(1)
    print(f"{t:0.2f} segundos")

# La comparación es
# >>> tiempo('posiciones1("a", "abc"*6000)')
# 0.00 segundos
# >>> tiempo('posiciones2("a", "abc"*6000)')
# 0.06 segundos

```



```
# >>> tiempo('posiciones3("a", "abc"*6000)')
# 0.00 segundos
#
# >>> tiempo('posiciones1("a", "abc"*(2*10**7))')
# 3.02 segundos
# >>> tiempo('posiciones3("a", "abc"*(2*10**7))')
# 3.47 segundos
```

3.15. Reconocimiento de subcadenas

En Haskell

```
-- -----
-- Definir, por recursión, la función
--   esSubcadena :: String -> String -> Bool
-- tal que (esSubcadena xs ys) se verifica si xs es una subcadena de
-- ys. Por ejemplo,
--   esSubcadena "casa" "escasamente" == True
--   esSubcadena "cante" "escasamente" == False
--   esSubcadena "" "" == True
-- -----
```

```
{-# OPTIONS_GHC -fno-warn-unused-imports #-}
```

```
module Reconocimiento_de_subcadenas where
```

```
import Data.List (isPrefixOf, isInfixOf, tails)
import Test.QuickCheck
```

```
-- 1ª solución
-- =====
```

```
esSubcadena1 :: String -> String -> Bool
esSubcadena1 [] _ = True
esSubcadena1 _ [] = False
esSubcadena1 xs (y:ys) = xs `isPrefixOf` (y:ys) || xs `esSubcadena1` ys
```

```
-- 2ª solución
-- =====
```

```

esSubcadena2 :: String -> String -> Bool
esSubcadena2 xs ys =
    or [xs `isPrefixOf` zs | zs <- sufijos ys]

-- (sufijos xs) es la lista de sufijos de xs. Por ejemplo,
--     sufijos "abc" == ["abc","bc","c",""]
sufijos :: String -> [String]
sufijos xs = [drop i xs | i <- [0..length xs]]

-- 3ª solución
-- =====

esSubcadena3 :: String -> String -> Bool
esSubcadena3 xs ys =
    or [xs `isPrefixOf` zs | zs <- tails ys]

-- 4ª solución
-- =====

esSubcadena4 :: String -> String -> Bool
esSubcadena4 xs ys =
    any (xs `isPrefixOf`) (tails ys)

-- 5ª solución
-- =====

esSubcadena5 :: String -> String -> Bool
esSubcadena5 = (. tails) . any . isPrefixOf

-- 6ª solución
-- =====

esSubcadena6 :: String -> String -> Bool
esSubcadena6 = isInfixOf

-- Comprobación de equivalencia
-- =====

-- La propiedad es
prop_esSubcadena :: String -> String -> Bool

```

```

prop_esSubcadena xs ys =
  all (== esSubcadena1 xs ys)
    [esSubcadena2 xs ys,
     esSubcadena3 xs ys,
     esSubcadena4 xs ys,
     esSubcadena5 xs ys,
     esSubcadena6 xs ys]

-- La comprobación es
--   λ> quickCheck prop_esSubcadena
--   +++ OK, passed 100 tests.

-- Comparación de eficiencia
-- =====

-- La comparación es
--   λ> esSubcadena1 "abc" (replicate (5*10^4) 'd' ++ "abc")
--   True
--   (0.03 secs, 17,789,392 bytes)
--   λ> esSubcadena2 "abc" (replicate (5*10^4) 'd' ++ "abc")
--   True
--   (6.32 secs, 24,989,912 bytes)
--
--   λ> esSubcadena1 "abc" (replicate (5*10^6) 'd' ++ "abc")
--   True
--   (3.24 secs, 1,720,589,432 bytes)
--   λ> esSubcadena3 "abc" (replicate (5*10^6) 'd' ++ "abc")
--   True
--   (1.81 secs, 1,720,589,656 bytes)
--   λ> esSubcadena4 "abc" (replicate (5*10^6) 'd' ++ "abc")
--   True
--   (0.71 secs, 1,120,589,480 bytes)
--   λ> esSubcadena5 "abc" (replicate (5*10^6) 'd' ++ "abc")
--   True
--   (0.41 secs, 1,120,589,584 bytes)
--   λ> esSubcadena6 "abc" (replicate (5*10^6) 'd' ++ "abc")
--   True
--   (0.11 secs, 560,589,200 bytes)

```

En Python

```
# -----
# Definir la función
#   esSubcadena : (str, str) -> bool
# tal que esSubcadena(xs ys) se verifica si xs es una subcadena de ys.
# Por ejemplo,
#   esSubcadena("casa", "escasamente") == True
#   esSubcadena("cante", "escasamente") == False
#   esSubcadena("", "") == True
# -----

from sys import setrecursionlimit
from timeit import Timer, default_timer

from hypothesis import given
from hypothesis import strategies as st

setrecursionlimit(10**6)

# 1ª solución
# =====

def esSubcadena1(xs: str, ys: str) -> bool:
    if not xs:
        return True
    if not ys:
        return False
    return ys.startswith(xs) or esSubcadena1(xs, ys[1:])

# 2ª solución
# =====

# sufijos(xs) es la lista de sufijos de xs. Por ejemplo,
#   sufijos("abc") == ['abc', 'bc', 'c', '']
def sufijos(xs: str) -> list[str]:
    return [xs[i:] for i in range(len(xs) + 1)]

def esSubcadena2(xs: str, ys: str) -> bool:
    return any(zs.startswith(xs) for zs in sufijos(ys))
```

```

# 3ª solución
# =====

def esSubcadena3(xs: str, ys: str) -> bool:
    return xs in ys

# Comprobación de equivalencia
# =====

# La propiedad es
@given(st.text(), st.text())
def test_esSubcadena(xs: str, ys: str) -> None:
    r = esSubcadena1(xs, ys)
    assert esSubcadena2(xs, ys) == r
    assert esSubcadena3(xs, ys) == r

# La comprobación es
# src> poetry run pytest -q reconocimiento_de_subcadenas.py
# 1 passed in 0.35s

# Comparación de eficiencia
# =====

def tiempo(e: str) -> None:
    """Tiempo (en segundos) de evaluar la expresión e."""
    t = Timer(e, "", default_timer, globals()).timeit(1)
    print(f"{t:0.2f} segundos")

# La comparación es
# >>> tiempo('esSubcadena1("abc", "d"*(10**4) + "abc")')
# 0.02 segundos
# >>> tiempo('esSubcadena2("abc", "d"*(10**4) + "abc")')
# 0.01 segundos
# >>> tiempo('esSubcadena3("abc", "d"*(10**4) + "abc")')
# 0.00 segundos
#
# >>> tiempo('esSubcadena2("abc", "d"*(10**5) + "abc")')
# 1.74 segundos
# >>> tiempo('esSubcadena3("abc", "d"*(10**5) + "abc")')
# 0.00 segundos

```


Capítulo 4

Funciones de orden superior

En este capítulo se presentan ejercicios con definiciones por comprensión. Se corresponden con el [tema 7 del curso de programación funcional con Haskell](#) ¹.

Contenido

4.1. Segmentos cuyos elementos cumplen una propiedad	311
4.2. Elementos consecutivos relacionados	315
4.3. Agrupación de elementos por posición	316
4.4. Concatenación de una lista de listas	323
4.5. Aplica según propiedad	327
4.6. Máximo de una lista	333

4.1. Segmentos cuyos elementos cumplen una propiedad

En Haskell

```
-- -----  
-- Definir la función  
-- segmentos :: (a -> Bool) -> [a] -> [[a]]  
-- tal que (segmentos p xs) es la lista de los segmentos de xs cuyos  
-- elementos verifican la propiedad p. Por ejemplo,
```

¹<https://jaalonso.github.io/materias/PFconHaskell/temas/tema-7.html>

```
-- segmentos even [1,2,0,4,9,6,4,5,7,2] == [[2,0,4],[6,4],[2]]
-- segmentos odd  [1,2,0,4,9,6,4,5,7,2] == [[1],[9],[5,7]]
-- -----
```

```
{-# OPTIONS_GHC -fno-warn-unused-imports #-}
```

```
module Segmentos_cuyos_elementos_cumple_una_propiedad where
```

```
import Data.List.Split (splitWhen)
import Test.QuickCheck.HigherOrder (quickCheck')
```

```
-- 1ª solución
```

```
-- =====
```

```
segmentos1 :: (a -> Bool) -> [a] -> [[a]]
segmentos1 _ [] = []
segmentos1 p (x:xs)
  | p x      = takeWhile p (x:xs) : segmentos1 p (dropWhile p xs)
  | otherwise = segmentos1 p xs
```

```
-- 2ª solución
```

```
-- =====
```

```
segmentos2 :: (a -> Bool) -> [a] -> [[a]]
segmentos2 p xs = filter (not . null) (splitWhen (not . p) xs)
```

```
-- 3ª solución
```

```
-- =====
```

```
segmentos3 :: (a -> Bool) -> [a] -> [[a]]
segmentos3 = (filter (not . null) .) . splitWhen . (not .)
```

```
-- Comprobación de equivalencia
```

```
-- =====
```

```
-- La propiedad es
```

```
prop_segmentos :: (Int -> Bool) -> [Int] -> Bool
prop_segmentos p xs =
  all (== segmentos1 p xs)
    [segmentos2 p xs,
```



```

segmentos3 p xs]

-- La comprobación es
--   λ> quickCheck' prop_segmentos
--   +++ OK, passed 100 tests.

-- Comparación de eficiencia
--   =====

-- La comparación es
--   λ> length (segmentos1 even [1..5*10^6])
--   2500000
--   (2.52 secs, 2,080,591,088 bytes)
--   λ> length (segmentos2 even [1..5*10^6])
--   2500000
--   (0.78 secs, 2,860,591,688 bytes)
--   λ> length (segmentos3 even [1..5*10^6])
--   2500000
--   (0.82 secs, 2,860,592,000 bytes)

```

En Python

```

# -----
# Definir la función
#   segmentos : (Callable[[A], bool], list[A]) -> list[list[A]]
# tal que segmentos(p, xs) es la lista de los segmentos de xs cuyos
# elementos verifican la propiedad p. Por ejemplo,
#   >>> segmentos1((lambda x: x % 2 == 0), [1,2,0,4,9,6,4,5,7,2])
#   [[2, 0, 4], [6, 4], [2]]
#   >>> segmentos1((lambda x: x % 2 == 1), [1,2,0,4,9,6,4,5,7,2])
#   [[1], [9], [5, 7]]
# -----

from itertools import dropwhile, takewhile
from sys import setrecursionlimit
from timeit import Timer, default_timer
from typing import Callable, TypeVar

from more_itertools import split_at

```

```
setrecursionlimit(10**6)
```

```
A = TypeVar('A')
```

```
# 1ª solución
```

```
# =====
```

```
def segmentos1(p: Callable[[A], bool], xs: list[A]) -> list[list[A]]:
    if not xs:
        return []
    if p(xs[0]):
        return [list(takewhile(p, xs))] + \
            segmentos1(p, list(dropwhile(p, xs[1:])))
    return segmentos1(p, xs[1:])
```

```
# 2ª solución
```

```
# =====
```

```
def segmentos2(p: Callable[[A], bool], xs: list[A]) -> list[list[A]]:
    return list(filter((lambda x: x), split_at(xs, lambda x: not p(x))))
```

```
# Comparación de eficiencia
```

```
# =====
```

```
def tiempo(e: str) -> None:
    """Tiempo (en segundos) de evaluar la expresión e."""
    t = Timer(e, "", default_timer, globals()).timeit(1)
    print(f"{t:0.2f} segundos")
```

```
# La comparación es
```

```
# >>> tiempo('segmentos1(lambda x: x % 2 == 0, range(10**4))')
# 0.55 segundos
# >>> tiempo('segmentos2(lambda x: x % 2 == 0, range(10**4))')
# 0.00 segundos
```

4.2. Elementos consecutivos relacionados

En Haskell

```

-- -----
-- Definir la función
--   relacionados :: (a -> a -> Bool) -> [a] -> Bool
-- tal que (relacionados r xs) se verifica si para todo par (x,y) de
-- elementos consecutivos de xs se cumple la relación r. Por ejemplo,
--   relacionados (<) [2,3,7,9]           == True
--   relacionados (<) [2,3,1,9]           == False
-- -----

```

module Elementos_consecutivos_relacionados where

```

-- 1ª solución
-- =====

```

```

relacionados1 :: (a -> a -> Bool) -> [a] -> Bool
relacionados1 r xs = and [r x y | (x,y) <- zip xs (tail xs)]

```

```

-- 2ª solución
-- =====

```

```

relacionados2 :: (a -> a -> Bool) -> [a] -> Bool
relacionados2 r (x:y:zs) = r x y && relacionados2 r (y:zs)
relacionados2 _ _       = True

```

```

-- 3ª solución
-- =====

```

```

relacionados3 :: (a -> a -> Bool) -> [a] -> Bool
relacionados3 r xs = and (zipWith r xs (tail xs))

```

```

-- 4ª solución
-- =====

```

```

relacionados4 :: (a -> a -> Bool) -> [a] -> Bool
relacionados4 r xs = all (uncurry r) (zip xs (tail xs))

```

En Python

```
# -----
# Definir la función
# relacionados : (Callable[[A, A], bool], list[A]) -> bool
# tal que relacionados(r, xs) se verifica si para todo par (x,y) de
# elementos consecutivos de xs se cumple la relación r. Por ejemplo,
# >>> relacionados(lambda x, y: x < y, [2, 3, 7, 9])
# True
# >>> relacionados(lambda x, y: x < y, [2, 3, 1, 9])
# False
# -----
```

```
from typing import Callable, TypeVar
```

```
A = TypeVar('A')
```

```
# 1ª solución
# =====
```

```
def relacionados1(r: Callable[[A, A], bool], xs: list[A]) -> bool:
    return all((r(x, y) for (x, y) in zip(xs, xs[1:])))
```

```
# 2ª solución
# =====
```

```
def relacionados2(r: Callable[[A, A], bool], xs: list[A]) -> bool:
    if len(xs) >= 2:
        return r(xs[0], xs[1]) and relacionados2(r, xs[1:])
    return True
```

4.3. Agrupación de elementos por posición

En Haskell

```
-- -----
-- Definir la función
-- agrupa :: Eq a => [[a]] -> [[a]]
-- tal que (agrupa xss) es la lista de las listas obtenidas agrupando
-- los primeros elementos, los segundos, ... Por ejemplo,
```

```

--      agrupa [[1..6],[7..9],[10..20]] == [[1,7,10],[2,8,11],[3,9,12]]
--
-- Comprobar con QuickChek que la longitud de todos los elementos de
-- (agrupa xs) es igual a la longitud de xs.
-- -----

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module Agrupacion_de_elementos_por_posicion where

import Data.List (transpose)
import qualified Data.Matrix as M (fromLists, toLists, transpose)
import Test.QuickCheck

-- 1ª solución
-- =====

-- (primeros xss) es la lista de los primeros elementos de xss. Por
-- ejemplo,
--      primeros [[1..6],[7..9],[10..20]] == [1,7,10]
primeros :: [[a]] -> [a]
primeros = map head

-- (restos xss) es la lista de los restos de elementos de xss. Por
-- ejemplo,
--      restos [[1..3],[7,8],[4..7]] == [[2,3],[8],[5,6,7]]
restos :: [[a]] -> [[a]]
restos = map tail

agrupal :: Eq a => [[a]] -> [[a]]
agrupal [] = []
agrupal xss
  | [] `elem` xss = []
  | otherwise    = primeros xss : agrupal (restos xss)

-- 2ª solución
-- =====

-- (conIgualLongitud xss) es la lista obtenida recortando los elementos
-- de xss para que todos tengan la misma longitud. Por ejemplo,

```

```

--      > conIgualLongitud [[1..6],[7..9],[10..20]]
--      [[1,2,3],[7,8,9],[10,11,12]]
conIgualLongitud :: [[a]] -> [[a]]
conIgualLongitud xss = map (take n) xss
    where n = minimum (map length xss)

agrupa2 :: Eq a => [[a]] -> [[a]]
agrupa2 = transpose . conIgualLongitud

-- 3ª solución
-- =====

agrupa3 :: Eq a => [[a]] -> [[a]]
agrupa3 = M.toLists . M.transpose . M.fromLists . conIgualLongitud

-- Comprobación de equivalencia
-- =====

-- La propiedad es
prop_agrupa :: NonEmptyList [Int] -> Bool
prop_agrupa (NonEmpty xss) =
    all (== agrupa1 xss)
        [agrupa2 xss,
         agrupa3 xss]

-- Comparación de eficiencia
-- =====

-- La comparación es
--      λ> length (agrupa1 [[1..10^4] | _ <- [1..10^4]])
--      10000
--      (3.96 secs, 16,012,109,904 bytes)
--      λ> length (agrupa2 [[1..10^4] | _ <- [1..10^4]])
--      10000
--      (25.80 secs, 19,906,197,528 bytes)
--      λ> length (agrupa3 [[1..10^4] | _ <- [1..10^4]])
--      10000
--      (9.56 secs, 7,213,797,984 bytes)

-- La comprobación es

```

```
-- λ> quickCheck prop_agrupa
-- +++ OK, passed 100 tests.

-- La propiedad es
prop_agrupa_length :: [[Int]] -> Bool
prop_agrupa_length xss =
  and [length xs == n | xs <- agrupa1 xss]
  where n = length xss

-- La comprobación es
-- λ> quickCheck prop_agrupa_length
-- +++ OK, passed 100 tests.
```

En Python

```
# -----
# Definir la función
#   agrupa : (list[list[A]]) -> list[list[A]]
# tal que agrupa(xss) es la lista de las listas obtenidas agrupando
# los primeros elementos, los segundos, ... Por ejemplo,
#   >>> agrupa([[1,6],[7,8,9],[3,4,5]])
#   [[1, 7, 3], [6, 8, 4]]
#
# Comprobar con QuickChek que la longitud de todos los elementos de
# (agrupa xs) es igual a la longitud de xs.
# -----

from sys import setrecursionlimit
from timeit import Timer, default_timer
from typing import TypeVar

from hypothesis import given
from hypothesis import strategies as st
from numpy import array, transpose

setrecursionlimit(10**6)

A = TypeVar('A')

# 1ª solución
```

```

# =====

# primeros(xss) es la lista de los primeros elementos de xss. Por
# ejemplo,
#     primeros([[1,6],[7,8,9],[3,4,5]]) == [1, 7, 3]
def primeros(xss: list[list[A]]) -> list[A]:
    return [xs[0] for xs in xss]

# restos(xss) es la lista de los restos de elementos de xss. Por
# ejemplo,
#     >>> restos([[1,6],[7,8,9],[3,4,5]])
#     [[6], [8, 9], [4, 5]]
def restos(xss: list[list[A]]) -> list[list[A]]:
    return [xs[1:] for xs in xss]

def 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
```

```
# La comprobación es
```

```
# src> poetry run pytest -q agrupacion_de_elementos_por_posicion.py
# 1 passed in 0.74s
```

```

# Comparación de eficiencia
# =====

def tiempo(e: str) -> None:
    """Tiempo (en segundos) de evaluar la expresión e."""
    t = Timer(e, "", default_timer, globals()).timeit(1)
    print(f"{t:0.2f} segundos")

# La comparación es
# >>> tiempo('agrupa1([list(range(10**3)) for _ in range(10**3)])')
# 4.44 segundos
# >>> tiempo('agrupa2([list(range(10**3)) for _ in range(10**3)])')
# 0.10 segundos
# >>> tiempo('agrupa3([list(range(10**3)) for _ in range(10**3)])')
# 0.10 segundos
# >>> tiempo('agrupa4([list(range(10**3)) for _ in range(10**3)])')
# 0.12 segundos
# >>> tiempo('agrupa5([list(range(10**3)) for _ in range(10**3)])')
# 0.15 segundos
#
# >>> tiempo('agrupa2([list(range(10**4)) for _ in range(10**4)])')
# 21.25 segundos
# >>> tiempo('agrupa3([list(range(10**4)) for _ in range(10**4)])')
# 20.82 segundos
# >>> tiempo('agrupa4([list(range(10**4)) for _ in range(10**4)])')
# 13.46 segundos
# >>> tiempo('agrupa5([list(range(10**4)) for _ in range(10**4)])')
# 21.70 segundos

# La propiedad es
@given(st.lists(st.lists(st.integers()), min_size=1))
def test_agrupa_length(xss: list[list[int]]) -> None:
    n = len(xss)
    assert all((len(xs) == n for xs in agrupa2(xss)))

# La comprobación es
# src> poetry run pytest -q agrupacion_de_elementos_por_posicion.py
# 2 passed in 1.25s

```

4.4. Concatenación de una lista de listas

En Haskell

```

-- -----
-- Definir, por recursión, la función
--   conc :: [[a]] -> [a]
-- tal que (conc xss) es la concenación de las listas de xss. Por
-- ejemplo,
--   conc [[1,3],[2,4,6],[1,9]] == [1,3,2,4,6,1,9]
--
-- Comprobar con QuickCheck que la longitud de (conc xss) es la suma de
-- las longitudes de los elementos de xss.
-- -----

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module Contenacion_de_una_lista_de_listas where

import Test.QuickCheck

-- 1ª solución
-- =====

conc1 :: [[a]] -> [a]
conc1 xss = [x | xs <- xss, x <- xs]

-- 2ª solución
-- =====

conc2 :: [[a]] -> [a]
conc2 []      = []
conc2 (xs:xss) = xs ++ conc2 xss

-- 3ª solución
-- =====

conc3 :: [[a]] -> [a]
conc3 = foldr (++) []

-- 4ª solución

```

```

-- =====

conc4 :: [[a]] -> [a]
conc4 = concat

-- Comprobación de equivalencia
-- =====

-- La propiedad es
prop_conc :: [[Int]] -> Bool
prop_conc xss =
  all (== conc1 xss)
    [conc2 xss,
     conc3 xss,
     conc4 xss]

-- La comprobación es
--   λ> quickCheck prop_conc
--   +++ OK, passed 100 tests.

-- Comparación de eficiencia
-- =====

-- La comparación es
--   λ> length (conc1 [[1..n] | n <- [1..5000]])
--   12502500
--   (2.72 secs, 1,802,391,200 bytes)
--   λ> length (conc2 [[1..n] | n <- [1..5000]])
--   12502500
--   (0.27 secs, 1,602,351,160 bytes)
--   λ> length (conc3 [[1..n] | n <- [1..5000]])
--   12502500
--   (0.28 secs, 1,602,071,192 bytes)
--   λ> length (conc4 [[1..n] | n <- [1..5000]])
--   12502500
--   (0.26 secs, 1,602,071,184 bytes)

-- Comprobación de la propiedad
-- =====

```

```
-- La propiedad es
prop_long_conc :: [[Int]] -> Bool
prop_long_conc xss =
    length (concl xss) == sum (map length xss)

-- La comprobación es
--    λ> quickCheck prop_long_conc
--    +++ OK, passed 100 tests.
```

En Python

```
# -----
# Definir, por recursión, la función
#   conc : (list[list[A]]) -> list[A]
# tal que conc(xss) es la concenación de las listas de xss. Por
# ejemplo,
#   conc([[1,3],[2,4,6],[1,9]]) == [1,3,2,4,6,1,9]
#
# Comprobar con hypothesis que la longitud de conc(xss) es la suma de
# las longitudes de los elementos de xss.
# -----

from functools import reduce
from operator import concat
from sys import setrecursionlimit
from timeit import Timer, default_timer
from typing import Any, TypeVar

from hypothesis import given
from hypothesis import strategies as st

setrecursionlimit(10**6)

A = TypeVar('A')

# 1ª solución
# =====

def concl(xss: list[list[A]]) -> list[A]:
    return [x for xs in xss for x in xs]
```

2ª solución

=====

```
def conc2(xss: list[list[A]]) -> list[A]:
    if not xss:
        return []
    return xss[0] + conc2(xss[1:])
```

3ª solución

=====

```
def conc3(xss: Any) -> Any:
    return reduce(concat, xss)
```

4ª solución

=====

```
def conc4(xss: list[list[A]]) -> list[A]:
    r = []
    for xs in xss:
        for x in xs:
            r.append(x)
    return r
```

La propiedad es

@given(st.lists(st.lists(st.integers()), min_size=1))

```
def test_conc(xss: list[list[int]]) -> None:
```

```
    r = conc1(xss)
    assert conc2(xss) == r
    assert conc3(xss) == r
    assert conc4(xss) == r
```

La comprobación es

```
# src> poetry run pytest -q concatenacion_de_una_lista_de_listas.py
# 1 passed in 0.63s
```

Comparación de eficiencia

=====

```

def tiempo(e: str) -> None:
    """Tiempo (en segundos) de evaluar la expresión e."""
    t = Timer(e, "", default_timer, globals()).timeit(1)
    print(f"{t:0.2f} segundos")

# La comparación es
# >>> tiempo('conc1([list(range(n)) for n in range(1500)])')
# 0.04 segundos
# >>> tiempo('conc2([list(range(n)) for n in range(1500)])')
# 6.28 segundos
# >>> tiempo('conc3([list(range(n)) for n in range(1500)])')
# 2.55 segundos
# >>> tiempo('conc4([list(range(n)) for n in range(1500)])')
# 0.09 segundos
#
# >>> tiempo('conc1([list(range(n)) for n in range(10000)])')
# 2.01 segundos
# >>> tiempo('conc4([list(range(n)) for n in range(10000)])')
# 2.90 segundos
#
# Comprobación de la propiedad
# =====

# La propiedad es
@given(st.lists(st.lists(st.integers()), min_size=1))
def test_long_conc(xss: list[list[int]]) -> None:
    assert len(conc1(xss)) == sum(map(len, xss))

# La comprobación es
# src> poetry run pytest -q concatenacion_de_una_lista_de_listas.py
# 2 passed in 0.81s

```

4.5. Aplica según propiedad

En Haskell

```

-- -----
-- Definir la función
--   filtraAplica :: (a -> b) -> (a -> Bool) -> [a] -> [b]
-- tal que (filtraAplica f p xs) es la lista obtenida aplicándole a los

```

```
-- elementos de xs que cumplen el predicado p la función f. Por ejemplo,
--   filtraAplica (4+) (<3) [1..7] == [5,6]
-- -----
```

```
{-# OPTIONS_GHC -fno-warn-unused-imports #-}
```

```
module Aplica_segun_propiedad where
```

```
import Test.QuickCheck.HigherOrder (quickCheck')
```

```
-- 1ª solución
```

```
-- =====
```

```
filtraAplica1 :: (a -> b) -> (a -> Bool) -> [a] -> [b]
```

```
filtraAplica1 f p xs = [f x | x <- xs, p x]
```

```
-- 2ª solución
```

```
-- =====
```

```
filtraAplica2 :: (a -> b) -> (a -> Bool) -> [a] -> [b]
```

```
filtraAplica2 f p xs = map f (filter p xs)
```

```
-- 3ª solución
```

```
-- =====
```

```
filtraAplica3 :: (a -> b) -> (a -> Bool) -> [a] -> [b]
```

```
filtraAplica3 _ _ [] = []
```

```
filtraAplica3 f p (x:xs) | p x      = f x : filtraAplica3 f p xs
                          | otherwise = filtraAplica3 f p xs
```

```
-- 4ª solución
```

```
-- =====
```

```
filtraAplica4 :: (a -> b) -> (a -> Bool) -> [a] -> [b]
```

```
filtraAplica4 f p = foldr g []
```

```
  where g x y | p x      = f x : y
              | otherwise = y
```

```
-- 5ª solución
```

```
-- =====
```



```

filtraAplica5 :: (a -> b) -> (a -> Bool) -> [a] -> [b]
filtraAplica5 f p =
    foldr (\x y -> if p x then f x : y else y) []

-- Comprobación de equivalencia
-- =====

-- La propiedad es
prop_filtraAplica :: (Int -> Int) -> (Int -> Bool) -> [Int] -> Bool
prop_filtraAplica f p xs =
    all (== filtraAplica1 f p xs)
        [filtraAplica2 f p xs,
         filtraAplica3 f p xs,
         filtraAplica4 f p xs,
         filtraAplica5 f p xs]

-- La comprobación es
--    λ> quickCheck' prop_filtraAplica
--    +++ OK, passed 100 tests.

-- Comparación de eficiencia
-- =====

-- La comparación es
--    λ> sum (filtraAplica1 id even [1..5*10^6])
--    6250002500000
--    (2.92 secs, 1,644,678,696 bytes)
--    λ> sum (filtraAplica2 id even [1..5*10^6])
--    6250002500000
--    (1.17 secs, 1,463,662,848 bytes)
--    λ> sum (filtraAplica3 id even [1..5*10^6])
--    6250002500000
--    (3.18 secs, 1,964,678,640 bytes)
--    λ> sum (filtraAplica4 id even [1..5*10^6])
--    6250002500000
--    (2.64 secs, 1,924,678,752 bytes)
--    λ> sum (filtraAplica5 id even [1..5*10^6])
--    6250002500000
--    (2.61 secs, 1,824,678,712 bytes)

```

En Python

```
# -----
# Definir la función
#   filtraAplica : (Callable[[A], B], Callable[[A], bool], list[A])
#                   -> list[B]
# tal que filtraAplica(f, p, xs) es la lista obtenida aplicándole a los
# elementos de xs que cumplen el predicado p la función f. Por ejemplo,
#   >>> filtraAplica(lambda x: x + 4, lambda x: x < 3, range(1, 7))
#   [5, 6]
# -----
```

```
from functools import reduce
from sys import setrecursionlimit
from timeit import Timer, default_timer
from typing import Callable, TypeVar
```

```
from hypothesis import given
from hypothesis import strategies as st
```

```
setrecursionlimit(10**6)
```

```
A = TypeVar('A')
```

```
B = TypeVar('B')
```

```
# 1ª solución
```

```
# =====
```

```
def filtraAplica1(f: Callable[[A], B],
                  p: Callable[[A], bool],
                  xs: list[A]) -> list[B]:
    return [f(x) for x in xs if p(x)]
```

```
# 2ª solución
```

```
# =====
```

```
def filtraAplica2(f: Callable[[A], B],
                  p: Callable[[A], bool],
                  xs: list[A]) -> list[B]:
    return list(map(f, filter(p, xs)))
```

```
# 3ª solución
```

```
# =====
```

```
def filtraAplica3(f: Callable[[A], B],
                  p: Callable[[A], bool],
                  xs: list[A]) -> list[B]:
    if not xs:
        return []
    if p(xs[0]):
        return [f(xs[0])] + filtraAplica3(f, p, xs[1:])
    return filtraAplica3(f, p, xs[1:])
```

```
# 4ª solución
```

```
# =====
```

```
def filtraAplica4(f: Callable[[A], B],
                  p: Callable[[A], bool],
                  xs: list[A]) -> list[B]:
    def g(ys: list[B], x: A) -> list[B]:
        if p(x):
            return ys + [f(x)]
        return ys

    return reduce(g, xs, [])
```

```
# 5ª solución
```

```
# =====
```

```
def filtraAplica5(f: Callable[[A], B],
                  p: Callable[[A], bool],
                  xs: list[A]) -> list[B]:
    r = []
    for x in xs:
        if p(x):
            r.append(f(x))
    return r
```

```
# Comprobación de equivalencia
```

```
# =====
```

```

# La propiedad es
@given(st.lists(st.integers()))
def test_filtraAplica(xs: list[int]) -> None:
    f = lambda x: x + 4
    p = lambda x: x < 3
    r = filtraAplica1(f, p, xs)
    assert filtraAplica2(f, p, xs) == r
    assert filtraAplica3(f, p, xs) == r
    assert filtraAplica4(f, p, xs) == r
    assert filtraAplica5(f, p, xs) == r

# La comprobación es
# src> poetry run pytest -q aplica_segun_propiedad.py
# 1 passed in 0.25s

# Comparación de eficiencia
# =====

def tiempo(e: str) -> None:
    """Tiempo (en segundos) de evaluar la expresión e."""
    t = Timer(e, "", default_timer, globals()).timeit(1)
    print(f"{t:0.2f} segundos")

# La comparación es
# >>> tiempo('filtraAplica1(lambda x: x, lambda x: x % 2 == 0, range(10**5))')
# 0.02 segundos
# >>> tiempo('filtraAplica2(lambda x: x, lambda x: x % 2 == 0, range(10**5))')
# 0.01 segundos
# >>> tiempo('filtraAplica3(lambda x: x, lambda x: x % 2 == 0, range(10**5))')
# Process Python violación de segmento (core dumped)
# >>> tiempo('filtraAplica4(lambda x: x, lambda x: x % 2 == 0, range(10**5))')
# 4.07 segundos
# >>> tiempo('filtraAplica5(lambda x: x, lambda x: x % 2 == 0, range(10**5))')
# 0.01 segundos
#
# >>> tiempo('filtraAplica1(lambda x: x, lambda x: x % 2 == 0, range(10**7))')
# 1.66 segundos
# >>> tiempo('filtraAplica2(lambda x: x, lambda x: x % 2 == 0, range(10**7))')
# 1.00 segundos
# >>> tiempo('filtraAplica5(lambda x: x, lambda x: x % 2 == 0, range(10**7))')

```

1.21 segundos

4.6. Máximo de una lista

En Haskell

```

-----
-- Definir la función
--   maximo :: Ord a => [a] -> a
-- tal que (maximo xs) es el máximo de la lista xs. Por ejemplo,
--   maximo [3,7,2,5]           == 7
--   maximo ["todo","es","falso"] == "todo"
--   maximo ["menos","alguna","cosa"] == "menos"
-----

```

```
{-# OPTIONS_GHC -fno-warn-incomplete-patterns #-}
```

```
module Maximo_de_una_lista where
```

```
import Data.List (foldl1')
import Test.QuickCheck
```

```

-- 1ª solución
-- =====

```

```

maximo1 :: Ord a => [a] -> a
maximo1 [x]      = x
maximo1 (x:y:ys) = max x (maximo1 (y:ys))

```

```

-- 2ª solución
-- =====

```

```

maximo2 :: Ord a => [a] -> a
maximo2 = foldr1 max

```

```

-- 3ª solución
-- =====

```

```

maximo3 :: Ord a => [a] -> a
maximo3 = foldl1' max

```

```

-- 4ª solución
-- =====

maximo4 :: Ord a => [a] -> a
maximo4 = maximum

-- Comprobación de equivalencia
-- =====

-- La propiedad es
prop_maximo :: NonEmptyList Int -> Bool
prop_maximo (NonEmpty xs) =
  all (== maximo1 xs)
    [maximo2 xs,
     maximo3 xs]

-- La comprobación es
--   λ> quickCheck prop_maximo
--   +++ OK, passed 100 tests.

-- Comparación de eficiencia
-- =====

-- La comparación es
--   λ> maximo1 [0..5*10^6]
--   5000000
--   (3.42 secs, 1,783,406,728 bytes)
--   λ> maximo2 [0..5*10^6]
--   5000000
--   (0.80 secs, 934,638,080 bytes)
--   λ> maximo3 [0..5*10^6]
--   5000000
--   (0.12 secs, 360,591,360 bytes)
--   λ> maximo4 [0..5*10^6]
--   5000000
--   (1.40 secs, 892,891,608 bytes)

```

En Python

```
# -----
# Definir la función
#     maximo : (list[A]) -> A:
# tal que maximo(xs) es el máximo de la lista xs. Por ejemplo,
#     maximo([3,7,2,5])           == 7
#     maximo(["todo","es","falso"]) == "todo"
#     maximo(["menos","alguna","cosa"]) == "menos"
# -----

from functools import reduce
from sys import setrecursionlimit
from timeit import Timer, default_timer
from typing import TypeVar, Union

from hypothesis import given
from hypothesis import strategies as st

setrecursionlimit(10**6)

A = TypeVar('A', bound=Union[int, float, str])

# 1ª solución
# =====

def maximo1(xs: list[A]) -> A:
    if len(xs) == 1:
        return xs[0]
    return max(xs[0], maximo1(xs[1:]))

# 2ª solución
# =====

def maximo2(xs: list[A]) -> A:
    return reduce(max, xs)

# 3ª solución
# =====

def maximo3(xs: list[A]) -> A:
```

```

    return max(xs)

# Comprobación de equivalencia
# =====

# La propiedad es
@given(st.lists(st.integers(), min_size=2))
def test_maximo(xs: list[int]) -> None:
    r = maximo1(xs)
    assert maximo2(xs) == r
    assert maximo3(xs) == r

# La comprobación es
#   src> poetry run pytest -q maximo_de_una_lista.py
#   1 passed in 0.33s

# Comparación de eficiencia
# =====

def tiempo(e: str) -> None:
    """Tiempo (en segundos) de evaluar la expresión e."""
    t = Timer(e, "", default_timer, globals()).timeit(1)
    print(f"{t:0.2f} segundos")

# La comparación es
#   >>> tiempo('maximo1(range(2*10**4))')
#   0.03 segundos
#   >>> tiempo('maximo2(range(2*10**4))')
#   0.00 segundos
#   >>> tiempo('maximo3(range(2*10**4))')
#   0.00 segundos
#
#   >>> tiempo('maximo2(range(5*10**6))')
#   0.38 segundos
#   >>> tiempo('maximo3(range(5*10**6))')
#   0.21 segundos

```


Capítulo 5

Tipos definidos y tipos de datos algebraicos

En este capítulo se presentan ejercicios con definiciones por comprensión. Se corresponden con el [tema 9 del curso de programación funcional con Haskell](https://jaalonso.github.io/materias/PFconHaskell/temas/tema-9.html) ¹.

Contenido

5.1. Movimientos en el plano	340
5.2. El tipo de figuras geométricas	345
5.3. El tipo de los números naturales	347
5.4. El tipo de las listas	350
5.5. El tipo de los árboles binarios	352
5.6. El tipo de las fórmulas: Variables de una fórmula	355
5.7. El tipo de las fórmulas: Valor de una fórmula	358
5.8. El tipo de las fórmulas: Interpretaciones de una fórmula . . .	363
5.9. El tipo de las fórmulas: Reconocedor de tautologías	368
5.10. Altura de un árbol binario	374
5.11. Aplicación de una función a un árbol	377
5.12. Árboles con la misma forma	379
5.13. Árbol con las hojas en la profundidad dada	385
5.14. El tipo de las expresiones aritméticas: Valor de una expresión	388

¹<https://jaalonso.github.io/materias/PFconHaskell/temas/tema-9.html>

5.15. El tipo de las expresiones aritméticas: Valor de la resta . . .	391
5.16. Número de hojas de un árbol binario	397
5.17. Profundidad de un árbol binario	402
5.17.1.En Haskell	402
5.18. Recorrido de árboles binarios	407
5.18.1.En Haskell	407
5.18.2.En Python	409
5.19. Imagen especular de un árbol binario	413
5.19.1.En Haskell	413
5.19.2.En Python	416
5.20. Subárbol de profundidad dada	419
5.20.1.En Haskell	419
5.20.2.En Python	422
5.21. Árbol de profundidad n con nodos iguales	425
5.21.1.En Haskell	425
5.21.2.En Python	428
5.22. Suma de un árbol	431
5.22.1.En Haskell	431
5.22.2.En Python	431
5.23. Rama izquierda de un árbol binario	433
5.23.1.En Haskell	433
5.23.2.En Python	434
5.24. Árboles balanceados	436
5.24.1.En Haskell	436
5.24.2.En Python	437
5.25. Árboles con bordes iguales	439
5.25.1.En Haskell	439
5.25.2.En Python	440
5.26. Árboles con igual estructura	442
5.26.1.En Haskell	442
5.26.2.En Python	443
5.27. Existencia de elementos del árbol que verifican una propiedad	445

5.27.1.En Haskell	.445
5.27.2.En Python	.446
5.28. Elementos del nivel k de un árbol	.448
5.28.1.En Haskell	.448
5.28.2.En Python	.449
5.29. Árbol de factorización	.451
5.29.1.En Haskell	.451
5.29.2.En Python	.454
5.30. Valor de un árbol booleano	.457
5.30.1.En Haskell	.457
5.30.2.En Python	.458
5.31. Valor de una expresión aritmética básica	.461
5.31.1.En Haskell	.461
5.31.2.En Python	.462
5.32. Aplicación de una función a una expresión aritmética	.463
5.32.1.En Haskell	.463
5.32.2.En Python	.464
5.33. Valor de una expresión aritmética con una variable	.466
5.33.1.En Haskell	.466
5.33.2.En Python	.467
5.34. Número de variables de una expresión aritmética	.469
5.34.1.En Haskell	.469
5.34.2.En Python	.470
5.35. Valor de una expresión aritmética con variables	.472
5.35.1.En Haskell	.472
5.35.2.En Python	.473
5.36. Número de sumas en una expresión aritmética	.475
5.36.1.En Haskell	.475
5.36.2.En Python	.476
5.37. Sustitución en una expresión aritmética	.478
5.37.1.En Haskell	.478
5.37.2.En Python	.479

5.38. Expresiones aritméticas reducibles	481
5.38.1.En Haskell481
5.38.2.En Python482
5.39. Máximos valores de una expresión aritmética	484
5.39.1.En Haskell484
5.39.2.En Python485
5.40. Valor de expresiones aritméticas generales	488
5.40.1.En Haskell488
5.40.2.En Python489
5.41. Valor de una expresión vectorial	491
5.41.1.En Haskell491
5.41.2.En Python492

5.1. Movimientos en el plano

En Haskell

```

-- -----
-- Se consideran el tipo de las posiciones del plano definido por
--   type Posicion = (Int,Int)
-- y el tipo de las direcciones definido por
--   data Direccion = Izquierda | Derecha | Arriba | Abajo
--   deriving Show
--
-- Definir las siguientes funciones
--   opuesta      :: Direccion -> Direccion
--   movimiento   :: Posicion -> Direccion -> Posicion
--   movimientos  :: Posicion -> [Direccion] -> Posicion
-- tales que
-- + (opuesta d) es la dirección opuesta de d. Por ejemplo,
--   opuesta Izquierda == Derecha
-- + (movimiento p d) es la posición resultante de moverse, desde la
--   posición p, un paso en la dirección d . Por ejemplo,
--   movimiento (2,5) Arriba      == (2,6)
--   movimiento (2,5) (opuesta Abajo) == (2,6)
-- + (movimientos p ds) es la posición obtenida aplicando la lista de

```

```

--    movimientos según las direcciones de ds a la posición p. Por ejemplo,
--    movimientos (2,5) [Arriba, Izquierda] == (1,6)
--    -----

module Movimientos_en_el_plano where

type Posicion = (Int,Int)

data Direccion = Izquierda | Derecha | Arriba | Abajo
  deriving Show

-- Definición de opuesta
-- =====

opuesta :: Direccion -> Direccion
opuesta Izquierda = Derecha
opuesta Derecha   = Izquierda
opuesta Arriba    = Abajo
opuesta Abajo     = Arriba

-- 1ª definición de movimiento
-- =====

movimiento1 :: Posicion -> Direccion -> Posicion
movimiento1 (x,y) Izquierda = (x-1,y)
movimiento1 (x,y) Derecha   = (x+1,y)
movimiento1 (x,y) Arriba    = (x,y+1)
movimiento1 (x,y) Abajo     = (x,y-1)

-- 2ª definición de movimiento
-- =====

movimiento2 :: Posicion -> Direccion -> Posicion
movimiento2 (x,y) d =
  case d of
    Izquierda -> (x-1,y)
    Derecha   -> (x+1,y)
    Arriba    -> (x,y+1)
    Abajo     -> (x,y-1)

```

```
-- 1ª definición de movimientos
-- =====

movimientos1 :: Posicion -> [Direccion] -> Posicion
movimientos1 p [] = p
movimientos1 p (d:ds) = movimientos1 (movimiento1 p d) ds

-- 2ª definición de movimientos
-- =====

movimientos2 :: Posicion -> [Direccion] -> Posicion
movimientos2 = foldl movimiento1
```

En Python

```
# -----
# Se consideran el tipo de las posiciones del plano definido por
#   Posicion = tuple[int, int]
# y el tipo de las direcciones definido por
#   Direccion = Enum('Direccion', ['Izquierda', 'Derecha', 'Arriba', 'Abajo'])
# Definir las siguientes funciones
#   opuesta      : (Direccion) -> Direccion
#   movimiento   : (Posicion, Direccion) -> Posicion
#   movimientos  : (Posicion, list[Direccion]) -> Posicion
# tales que
# + opuesta(d) es la dirección opuesta de d. Por ejemplo,
#   opuesta1(Direccion.Izquierda) == Direccion.Derecha
# + movimiento(p, d) es la posición resultante de moverse, desde la
#   posición p, un paso en la dirección d. Por ejemplo,
#   movimiento1((2, 5), Direccion.Arriba) == (2, 6)
#   movimiento1((2, 5), opuesta1(Direccion.Abajo)) == (2, 6)
# + movimientos(p, ds) es la posición obtenida aplicando la lista de
#   movimientos según las direcciones de ds a la posición p. Por
#   ejemplo,
#   >>> movimientos1((2, 5), [Direccion.Arriba, Direccion.Izquierda])
#   (1, 6)
# -----

from enum import Enum
from functools import reduce
```

```
Posicion = tuple[int, int]
```

```
Direccion = Enum('Direccion', ['Izquierda', 'Derecha', 'Arriba', 'Abajo'])
```

```
# 1ª definición de opuesta
```

```
# =====
```

```
def opuesta1(d: Direccion) -> Direccion:
```

```
    if d == Direccion.Izquierda:
```

```
        return Direccion.Derecha
```

```
    if d == Direccion.Derecha:
```

```
        return Direccion.Izquierda
```

```
    if d == Direccion.Arriba:
```

```
        return Direccion.Abajo
```

```
    if d == Direccion.Abajo:
```

```
        return Direccion.Arriba
```

```
    assert False
```

```
# 2ª definición de opuesta
```

```
# =====
```

```
def opuesta2(d: Direccion) -> Direccion:
```

```
    match d:
```

```
        case Direccion.Izquierda:
```

```
            return Direccion.Derecha
```

```
        case Direccion.Derecha:
```

```
            return Direccion.Izquierda
```

```
        case Direccion.Arriba:
```

```
            return Direccion.Abajo
```

```
        case Direccion.Abajo:
```

```
            return Direccion.Arriba
```

```
    assert False
```

```
# 1ª definición de movimiento
```

```
# =====
```

```
def movimientol(p: Posicion, d: Direccion) -> Posicion:
```

```
    (x, y) = p
```

```
    if d == Direccion.Izquierda:
```

```

        return (x - 1, y)
    if d == Direccion.Derecha:
        return (x + 1, y)
    if d == Direccion.Arriba:
        return (x, y + 1)
    if d == Direccion.Abajo:
        return (x, y - 1)
    assert False

# 2ª definición de movimiento
# =====

def movimiento2(p: Posicion, d: Direccion) -> Posicion:
    (x, y) = p
    match d:
        case Direccion.Izquierda:
            return (x - 1, y)
        case Direccion.Derecha:
            return (x + 1, y)
        case Direccion.Arriba:
            return (x, y + 1)
        case Direccion.Abajo:
            return (x, y - 1)
    assert False

# 1ª definición de movimientos
# =====

def movimientos1(p: Posicion, ds: list[Direccion]) -> Posicion:
    if not ds:
        return p
    return movimiento1(movimiento1(p, ds[0]), ds[1:])

# 2ª definición de movimientos
# =====

def movimientos2(p: Posicion, ds: list[Direccion]) -> Posicion:
    return reduce(movimiento1, ds, p)

```


5.2. El tipo de figuras geométricas

En Haskell

```

-----
-- Se consideran las figuras geométricas formadas por círculos
-- (definidos por su radio) y rectángulos (definidos por su base y su
-- altura). El tipo de las figura geométricas se define por
--   data Figura = Circulo Float | Rect Float Float
--
-- Definir las funciones
--   area      :: Figura -> Float
--   cuadrado  :: Float -> Figura
-- tales que
-- + (area f) es el área de la figura f. Por ejemplo,
--   area (Circulo 1)  == 3.1415927
--   area (Circulo 2)  == 12.566371
--   area (Rect 2 5)   == 10.0
-- + (cuadrado n) es el cuadrado de lado n. Por ejemplo,
--   area (cuadrado 3) == 9.0
-----

```

```
module El_tipo_de_figuras_geometricas where
```

```
data Figura = Circulo Float | Rect Float Float
```

```
area :: Figura -> Float
area (Circulo r) = pi*r^2
area (Rect x y)  = x*y
```

```
cuadrado :: Float -> Figura
cuadrado n = Rect n n
```

En Python

```

# -----
# Se consideran las figuras geométricas formadas por círculos
# (definidos por su radio) y rectángulos (definidos por su base y su
# altura). El tipo de las figura geométricas se define por
#   @dataclass

```

```

#     class Figura:
#         """Figuras geométricas"""
#
#     @dataclass
#     class Circulo(Figura):
#         r: float
#
#     @dataclass
#     class Rect(Figura):
#         x: float
#         y: float
#
# Definir las funciones
#     area      : (Figura) -> float
#     cuadrado  : (float) -> Figura
# tales que
# + area(f) es el área de la figura f. Por ejemplo,
#     area(Circulo(1)) == 3.141592653589793
#     area(Circulo(2)) == 12.566370614359172
#     area(Rect(2, 5)) == 10
# + cuadrado(n) es el cuadrado de lado n. Por ejemplo,
#     area(cuadrado(3)) == 9.0
# -----

from dataclasses import dataclass
from math import pi

@dataclass
class Figura:
    """Figuras geométricas"""

@dataclass
class Circulo(Figura):
    r: float

@dataclass
class Rect(Figura):
    x: float
    y: float

```

```
def area(f: Figura) -> float:
    match f:
        case Circulo(r):
            return pi * r**2
        case Rect(x, y):
            return x * y
    assert False

def cuadrado(n: float) -> Figura:
    return Rect(n, n)
```

5.3. El tipo de los números naturales

En Haskell

```
-----
-- El tipo de los números naturales se puede definir por
--   data Nat = Cero | Suc Nat
--   deriving (Show, Eq)
-- de forma que (Suc (Suc (Suc Cero))) representa el número 3.
--
-- Definir las siguientes funciones
--   nat2int :: Nat -> Int
--   int2nat :: Int -> Nat
--   suma    :: Nat -> Nat -> Nat
-- tales que
-- + (nat2int n) es el número entero correspondiente al número natural
--   n. Por ejemplo,
--   nat2int (Suc (Suc (Suc Cero))) == 3
-- + (int2nat n) es el número natural correspondiente al número entero n. Por eje
--   int2nat 3 == Suc (Suc (Suc Cero))
-- + (suma m n) es la suma de los número naturales m y n. Por ejemplo,
--   λ> suma (Suc (Suc Cero)) (Suc Cero)
--       Suc (Suc (Suc Cero))
--   λ> nat2int (suma (Suc (Suc Cero)) (Suc Cero))
--       3
--   λ> nat2int (suma (int2nat 2) (int2nat 1))
--       3
-----
```

```
module El_tipo_de_los_numeros_naturales where
```

```
data Nat = Cero | Suc Nat
  deriving (Show, Eq)
```

```
nat2int :: Nat -> Int
nat2int Cero    = 0
nat2int (Suc n) = 1 + nat2int n
```

```
int2nat :: Int -> Nat
int2nat 0 = Cero
int2nat n = Suc (int2nat (n-1))
```

```
suma :: Nat -> Nat -> Nat
suma Cero    n = n
suma (Suc m) n = Suc (suma m n)
```

En Python

```
# -----
# El tipo de los números naturales se puede definir por
# @dataclass
# class Nat:
#     pass
#
# @dataclass
# class Cero(Nat):
#     pass
#
# @dataclass
# class Suc(Nat):
#     n: Nat
# de forma que Suc(Suc(Suc(Cero()))) representa el número 3.
#
# Definir las siguientes funciones
# nat2int : (Nat) -> int
# int2nat : (int) -> Nat
# suma    : (Nat, Nat) -> Nat
# tales que
```

```

# + nat2int(n) es el número entero correspondiente al número natural
#   n. Por ejemplo,
#       nat2int(Suc(Suc(Suc(Cero())))) == 3
# + int2nat(n) es el número natural correspondiente al número entero
#   n. Por ejemplo,
#       int2nat(3) == Suc(Suc(Suc(Cero())))
# + suma(m, n) es la suma de los números naturales m y n. Por ejemplo,
#       >>> suma(Suc(Suc(Cero())), Suc(Cero()))
#           Suc(Suc(Suc(Cero())))
#       >>> nat2int(suma(Suc(Suc(Cero())), Suc(Cero())))
#           3
#       >>> nat2int(suma(int2nat(2), int2nat(1)))
#           3
# -----

```

```

from dataclasses import dataclass

```

```

@dataclass
class Nat:
    pass

```

```

@dataclass
class Cero(Nat):
    pass

```

```

@dataclass
class Suc(Nat):
    n: Nat

```

```

def nat2int(n: Nat) -> int:
    match n:
        case Cero():
            return 0
        case Suc(n):
            return 1 + nat2int(n)
    assert False

```

```

def int2nat(n: int) -> Nat:
    if n == 0:

```

```

        return Cero()
    return Suc(int2nat(n - 1))

def suma(m: Nat, n: Nat) -> Nat:
    match m:
        case Cero():
            return n
        case Suc(m):
            return Suc(suma(m, n))
    assert False

```

5.4. El tipo de las listas

En Haskell

```

-- -----
-- El tipo de las listas, con elementos de tipo a, se puede definir por
--   data Lista a = Nil | Cons a (Lista a)
-- Por ejemplo, la lista [4,2,5] se representa por
--   Cons 4 (Cons 2 (Cons 5 Nil)).
--
-- Definir la función
--   longitud :: Lista a -> Int
-- tal que (longitud xs) es la longitud de la lista xs. Por ejemplo,
--   longitud (Cons 4 (Cons 2 (Cons 5 Nil))) == 3
-- -----

```

```

module El_tipo_de_las_listas where

data Lista a = Nil | Cons a (Lista a)

longitud :: Lista a -> Int
longitud Nil      = 0
longitud (Cons _ xs) = 1 + longitud xs

```

En Python

```

# -----
# El tipo de las listas, con elementos de tipo a, se puede definir por

```

```

# @dataclass
# class Lista(Generic[A]):
#     pass
#
# @dataclass
# class Nil(Lista[A]):
#     pass
#
# @dataclass
# class Cons(Lista[A]):
#     x: A
#     xs: Lista[A]
# Por ejemplo, la lista [4,2,5] se representa por
# Cons(4, Cons(2, Cons(5, Nil()))).
#
# Definir la función
# longitud :: Lista a -> Int
# tal que (longitud xs) es la longitud de la lista xs. Por ejemplo,
# >>> longitud(Cons(4, Cons(2, Cons(5, Nil()))))
# 3
# -----

```

```

from dataclasses import dataclass
from typing import Generic, TypeVar

```

```
A = TypeVar("A")
```

```

@dataclass
class Lista(Generic[A]):
    pass

```

```

@dataclass
class Nil(Lista[A]):
    pass

```

```

@dataclass
class Cons(Lista[A]):
    x: A
    xs: Lista[A]

```

```
def longitud(xs: Lista[A]) -> int:
    match xs:
        case Nil():
            return 0
        case Cons(_, xs):
            return 1 + longitud(xs)
    assert False
```

5.5. El tipo de los árboles binarios

En Haskell

```
-- -----
-- El árbol binario
--
--      5
--     / \
--    /   \
--   3     7
--  / \   / \
-- 1  4 6  9
-- se puede representar por
--   ejArbol = Nodo (Nodo (Hoja 1) 3 (Hoja 4))
--                5
--                (Nodo (Hoja 6) 7 (Hoja 9))
--
-- El tipo de los árboles binarios se puede definir por
--   data Arbol = Hoja Int
--              | Nodo Arbol Int Arbol
--
-- Definir las funciones
--   ocurre :: Int -> Arbol -> Bool
--   aplana :: Arbol -> [Int]
-- tales que
-- + (ocurre m a) se verifica si m ocurre en el árbol a. Por ejemplo,
--   ocurre 4 ejArbol == True
--   ocurre 10 ejArbol == False
-- + (aplana a) es la lista obtenida aplanando el árbol a. Por ejemplo,
--   aplana ejArbol == [1,3,4,5,6,7,9]
-- -----
```



```

module El_tipo_de_los_arboles_binarios where

data Arbol = Hoja Int
           | Nodo Arbol Int Arbol

ejArbol :: Arbol
ejArbol = Nodo (Nodo (Hoja 1) 3 (Hoja 4))
           5
           (Nodo (Hoja 6) 7 (Hoja 9))

ocurre :: Int -> Arbol -> Bool
ocurre m (Hoja n)      = m == n
ocurre m (Nodo i n d) = m == n || ocurre m i || ocurre m d

aplana :: Arbol -> [Int]
aplana (Hoja n)      = [n]
aplana (Nodo i n d) = aplana i ++ [n] ++ aplana d

```

En Python

```

# -----
# El árbol binario
#
#      5
#     / \
#    /   \
#   3     7
#  / \   / \
# 1  4 6  9
# se puede representar por
# ejArbol = Nodo (Nodo (Hoja 1) 3 (Hoja 4))
#
#           5
#           (Nodo (Hoja 6) 7 (Hoja 9))
#
# El tipo de los árboles binarios se puede definir por
# @dataclass
# class Arbol:
#     pass
#
# @dataclass
# class Hoja(Arbol):

```

```

#         x: int
#
#     @dataclass
#     class Nodo(Arbol):
#         i: Arbol
#         x: int
#         d: Arbol
#
# Definir las funciones
#     ocurre : (int, Arbol) -> bool
#     aplana : (Arbol) -> list[int]
# tales que
# + ocurre(m, a) se verifica si m ocurre en el árbol a. Por ejemplo,
#     ocurre(4, ejArbol) == True
#     ocurre(0, ejArbol) == False
# + aplana(a) es la lista obtenida aplanando el árbol a. Por ejemplo,
#     aplana(ejArbol) == [1,3,4,5,6,7,9]
# -----

```

```

from dataclasses import dataclass

```

```

@dataclass
class Arbol:
    pass

```

```

@dataclass
class Hoja(Arbol):
    x: int

```

```

@dataclass
class Nodo(Arbol):
    i: Arbol
    x: int
    d: Arbol

```

```

ejArbol = Nodo(Nodo(Hoja(1), 3, Hoja(4)),
               5,
               Nodo(Hoja(6), 7, Hoja(9)))

```

```

def ocurre(m: Int, a: Arbol) -> Bool:
  match a:
    case Hoja(n):
      return m == n
    case Nodo(i, n, d):
      return m == n or ocurre(m, i) or ocurre(m, d)
  assert False

def aplana(a: Arbol) -> List[Int]:
  match a:
    case Hoja(n):
      return [n]
    case Nodo(i, n, d):
      return aplana(i) + [n] + aplana(d)
  assert False

```

5.6. El tipo de las fórmulas: Variables de una fórmula

En Haskell

```

-- -----
-- El tipo de las fórmulas proposicionales se puede definir por
--   data FProp = Const Bool
--               | Var Char
--               | Neg FProp
--               | Conj FProp FProp
--               | Impl FProp FProp
--   deriving Show
-- de modo que la fórmula  $A \rightarrow \perp \wedge \neg B$  se representa por
--   Impl (Var 'A') (Conj (Const False) (Neg (Var 'B'))))
--
-- Definir la función
--   variables :: FProp -> [Char]
-- tal que (variables p) es la lista de las variables de la fórmula
-- p. Por ejemplo,
--   λ> variables (Impl (Var 'A') (Conj (Const False) (Neg (Var 'B'))))
--   "AB"
--   λ> variables (Impl (Var 'A') (Conj (Var 'A') (Neg (Var 'B'))))

```

```
--      "AAB"
```

```
module Variables_de_una_formula where
```

```
data FProp = Const Bool
           | Var Char
           | Neg FProp
           | Conj FProp FProp
           | Impl FProp FProp
deriving Show
```

```
variables :: FProp -> [Char]
variables (Const _) = []
variables (Var x)   = [x]
variables (Neg p)   = variables p
variables (Conj p q) = variables p ++ variables q
variables (Impl p q) = variables p ++ variables q
```

En Python

```
# -----
# El tipo de las fórmulas proposicionales se puede definir por
# @dataclass
# class FProp:
#     pass
#
# @dataclass
# class Const(FProp):
#     x: bool
#
# @dataclass
# class Var(FProp):
#     x: str
#
# @dataclass
# class Neg(FProp):
#     x: FProp
#
# @dataclass
```

```

#     class Conj(FProp):
#         x: FProp
#         y: FProp
#
#     @dataclass
#     class Impl(FProp):
#         x: FProp
#         y: FProp
# de modo que la fórmula  $A \rightarrow \perp \wedge \neg B$  se representa por
#     Impl(Var('A'), Conj(Const(False), Neg (Var('B'))))
#
# Definir la función
#     variables : (FProp) -> list[str]:
# tal que variables(p) es la lista de las variables de la fórmula
# p. Por ejemplo,
#     >>> variables (Impl(Var('A'), Conj(Const(False), Neg (Var('B')))))
#     ['A', 'B']
#     >>> variables (Impl(Var('A'), Conj(Var('A'), Neg (Var('B')))))
#     ['A', 'A', 'B']
# -----

```

```

from dataclasses import dataclass

```

```

@dataclass
class FProp:
    pass

```

```

@dataclass
class Const(FProp):
    x: bool

```

```

@dataclass
class Var(FProp):
    x: str

```

```

@dataclass
class Neg(FProp):
    x: FProp

```

```

@dataclass
class Conj(FProp):
    x: FProp
    y: FProp

@dataclass
class Impl(FProp):
    x: FProp
    y: FProp

def variables(f: FProp) -> list[str]:
    match f:
        case Const(_):
            return []
        case Var(x):
            return [x]
        case Neg(p):
            return variables(p)
        case Conj(p, q):
            return variables(p) + variables(q)
        case Impl(p, q):
            return variables(p) + variables(q)
    assert False

```

5.7. El tipo de las fórmulas: Valor de una fórmula

En Haskell

```

-----
-- El tipo de las fórmulas proposicionales se puede definir por
--   data FProp = Const Bool
--               | Var Char
--               | Neg FProp
--               | Conj FProp FProp
--               | Impl FProp FProp
--   deriving Show
-- de modo que la fórmula  $A \rightarrow \perp \wedge \neg B$  se representa por
--   Impl (Var 'A') (Conj (Const False) (Neg (Var 'B')))

```

```
--
-- Una interpretación de una fórmula es una función de sus variables en
-- los booleanos. Por ejemplo, la interpretación que a la variable A le
-- asigna verdadero y a la B falso se puede representar por
--    [('A', True), ('B', False)]
--
-- El tipo de las interpretaciones se puede definir por
--    type Interpretacion = [(Char, Bool)]
--
-- El valor de una fórmula en una interpretación se calcula usando las
-- funciones de verdad de las conectivas que se muestran a continuación
--
--    |---+---|    |---+---+-----+-----|
--    | p | ¬p |    | p | q | p ∧ q | p → q |
--    |---+---|    |---+---+-----+-----|
--    | T | F  |    | T | T | T      | T      |
--    | F | T  |    | T | F | F      | F      |
--    |---+---|    | F | T | F      | T      |
--                  | F | F | F      | T      |
--                  |---+---+-----+-----|
--
-- Definir la función
--    valor :: Interpretacion -> FProp -> Bool
-- tal que (valor i p) es el valor de la fórmula p en la interpretación
-- i. Por ejemplo,
--    λ> p = Impl (Var 'A') (Conj (Var 'A') (Var 'B'))
--    λ> valor [('A',False),('B',False)] p
--    True
--    λ> valor [('A',True),('B',False)] p
--    False
--
-----
```

```
module Valor_de_una_formula where
```

```
data FProp = Const Bool
           | Var Char
           | Neg FProp
           | Conj FProp FProp
           | Impl FProp FProp
deriving Show
```

```

type Interpretacion = [(Char, Bool)]

valor :: Interpretacion -> FProp -> Bool
valor _ (Const b) = b
valor i (Var x)    = busca x i
valor i (Neg p)    = not (valor i p)
valor i (Conj p q) = valor i p && valor i q
valor i (Impl p q) = valor i p <= valor i q

-- (busca c t) es el valor del primer elemento de la lista de asociación
-- t cuya clave es c. Por ejemplo,
--     busca 2 [(1,'a'),(3,'d'),(2,'c')] == 'c'
busca :: Eq c => c -> [(c,v)] -> v
busca c t = head [v | (c',v) <- t, c == c']

```

En Python

```

# -----
# El tipo de las fórmulas proposicionales se puede definir por
# @dataclass
# class FProp:
#     pass
#
# @dataclass
# class Const(FProp):
#     x: bool
#
# @dataclass
# class Var(FProp):
#     x: str
#
# @dataclass
# class Neg(FProp):
#     x: FProp
#
# @dataclass
# class Conj(FProp):
#     x: FProp
#     y: FProp
#

```



```

# @dataclass
# class Impl(FProp):
#     x: FProp
#     y: FProp
# de modo que la fórmula  $A \rightarrow \perp \wedge \neg B$  se representa por
#     Impl(Var('A'), Conj(Const(False), Neg (Var('B'))))
#
# Una interpretación de una fórmula es una función de sus variables en
# los booleanos. Por ejemplo, la interpretación que a la variable A le
# asigna verdadero y a la B falso se puede representar por
#     [('A', True), ('B', False)]
#
# El tipo de las interpretaciones se puede definir por
#     Interpretacion = list[tuple[str, bool]]
#
# El valor de una fórmula en una interpretación se calcula usando las
# funciones de verdad de las conectivas que se muestran a continuación
#
# |---+---| |---+---+-----+-----|
# | p | ¬p | | p | q | p ∧ q | p → q |
# |---+---| |---+---+-----+-----|
# | T | F | | T | T | T      | T      |
# | F | T | | T | F | F      | F      |
# |---+---| | F | T | F      | T      |
#           | F | F | F      | T      |
#           |---+---+-----+-----|
#
# Definir la función
#     (i: Interpretacion, f: FProp) -> bool:
# tal que valor(i, p) es el valor de la fórmula p en la interpretación
# i. Por ejemplo,
#     >>> p = Impl(Var('A'), Conj(Var('A'), Var('B')))
#     >>> valor([('A',False),('B',False)], p)
#     True
#     >>> valor([('A',True),('B',False)], p)
#     False
# -----

```

```

from dataclasses import dataclass

```

```
@dataclass
class FProp:
    pass
```

```
@dataclass
class Const(FProp):
    x: bool
```

```
@dataclass
class Var(FProp):
    x: str
```

```
@dataclass
class Neg(FProp):
    x: FProp
```

```
@dataclass
class Conj(FProp):
    x: FProp
    y: FProp
```

```
@dataclass
class Impl(FProp):
    x: FProp
    y: FProp
```

```
Interpretacion = list[tuple[str, bool]]
```

```
# busca(c, t) es el valor del primer elemento de la lista de asociación
# t cuya clave es c. Por ejemplo,
# >>> busca('B', [('A', True), ('B', False), ('C', True)])
# False
```

```
def busca(c: str, i: Interpretacion) -> bool:
    return [v for (d, v) in i if d == c][0]
```

```
def valor(i: Interpretacion, f: FProp) -> bool:
    match f:
        case Const(b):
            return b
        case Var(x):
```

```

        return busca(x, i)
    case Neg(p):
        return not valor(i, p)
    case Conj(p, q):
        return valor(i, p) and valor(i, q)
    case Impl(p, q):
        return valor(i, p) <= valor(i, q)
assert False

```

5.8. El tipo de las fórmulas: Interpretaciones de una fórmula

En Haskell

```

-- -----
-- El tipo de las fórmulas proposicionales se puede definir por
--   data FProp = Const Bool
--               | Var Char
--               | Neg FProp
--               | Conj FProp FProp
--               | Impl FProp FProp
--   deriving Show
-- de modo que la fórmula  $A \rightarrow \perp \wedge \neg B$  se representa por
--   Impl (Var 'A') (Conj (Const False) (Neg (Var 'B')))
--
-- Una interpretación de una fórmula es una función de sus variables en
-- los booleanos. Por ejemplo, la interpretación que a la variable A le
-- asigna verdadero y a la B falso se puede representar por
--   [('A', True), ('B', False)]
--
-- El tipo de las interpretaciones se puede definir por
--   type Interpretacion = [(Char, Bool)]
--
-- Definir la función
--   interpretaciones :: FProp -> [Interpretacion]
-- tal que (interpretaciones p) es la lista de las interpretaciones de
-- la fórmula p. Por ejemplo,
--   λ> interpretaciones (Impl (Var 'A') (Conj (Var 'A') (Var 'B')))
--   [[('A',False),('B',False)],

```

```

--      [('A',False),('B',True)],
--      [('A',True),('B',False)],
--      [('A',True),('B',True)]]
--      -----

module Interpretaciones_de_una_formula where

import Data.List (nub)

data FProp = Const Bool
           | Var Char
           | Neg FProp
           | Conj FProp FProp
           | Impl FProp FProp
deriving Show

type Interpretacion = [(Char, Bool)]

interpretaciones :: FProp -> [Interpretacion]
interpretaciones p =
  [zip vs i | i <- interpretacionesVar (length vs)]
  where vs = nub (variables p)

-- (interpretacionesVar n) es la lista de las interpretaciones de n
-- variables. Por ejemplo,
--      λ> interpretacionesVar 2
--      [[False,False],
--       [False,True],
--       [True,False],
--       [True,True]]
interpretacionesVar :: Int -> [[Bool]]
interpretacionesVar 0 = [[]]
interpretacionesVar n = map (False::) bss ++ map (True::) bss
  where bss = interpretacionesVar (n-1)

-- (variables p) es la lista de las variables de la fórmula p. Por
-- ejemplo,
--      λ> variables (Impl (Var 'A') (Conj (Const False) (Neg (Var 'B'))))
--      "AB"
--      λ> variables (Impl (Var 'A') (Conj (Var 'A') (Neg (Var 'B'))))

```

```
--      "AAB"
variables :: FProp -> [Char]
variables (Const _) = []
variables (Var x)    = [x]
variables (Neg p)    = variables p
variables (Conj p q) = variables p ++ variables q
variables (Impl p q) = variables p ++ variables q
```

En Python

```
# -----
# El tipo de las fórmulas proposicionales se puede definir por
# @dataclass
# class FProp:
#     pass
#
# @dataclass
# class Const(FProp):
#     x: bool
#
# @dataclass
# class Var(FProp):
#     x: str
#
# @dataclass
# class Neg(FProp):
#     x: FProp
#
# @dataclass
# class Conj(FProp):
#     x: FProp
#     y: FProp
#
# @dataclass
# class Impl(FProp):
#     x: FProp
#     y: FProp
# de modo que la fórmula  $A \rightarrow \perp \wedge \neg B$  se representa por
# Impl(Var('A'), Conj(Const(False), Neg (Var('B'))))
#
```

```

# Una interpretación de una fórmula es una función de sus variables en
# los booleanos. Por ejemplo, la interpretación que a la variable A le
# asigna verdadero y a la B falso se puede representar por
#    [('A', True), ('B', False)]
#
# El tipo de las interpretaciones se puede definir por
#    Interpretacion = list[tuple[str, bool]]
#
# Definir la función
#    interpretaciones : (FProp) -> list[Interpretacion]
# tal que interpretaciones(p) es la lista de las interpretaciones de
# la fórmula p. Por ejemplo,
#    >>> interpretaciones(Impl(Var('A'), Conj(Var('A'), Var('B'))))
#    [('B', False), ('A', False)],
#    [('B', False), ('A', True)],
#    [('B', True), ('A', False)],
#    [('B', True), ('A', True)]
# -----

```

```

from dataclasses import dataclass

```

```

@dataclass
class FProp:
    pass

```

```

@dataclass
class Const(FProp):
    x: bool

```

```

@dataclass
class Var(FProp):
    x: str

```

```

@dataclass
class Neg(FProp):
    x: FProp

```

```

@dataclass
class Conj(FProp):

```

```
x: FProp
y: FProp
```

```
@dataclass
class Impl(FProp):
    x: FProp
    y: FProp
```

```
Interpretacion = list[tuple[str, bool]]
```

```
# variables(p) es la lista de las variables de la fórmula p. Por
# ejemplo,
# >>> variables (Impl(Var('A'), Conj(Const(False), Neg (Var('B')))))
# ['A', 'B']
# >>> variables (Impl(Var('A'), Conj(Var('A'), Neg (Var('B')))))
# ['A', 'A', 'B']
def variables(f: FProp) -> list[str]:
    match f:
        case Const(_):
            return []
        case Var(x):
            return [x]
        case Neg(p):
            return variables(p)
        case Conj(p, q):
            return variables(p) + variables(q)
        case Impl(p, q):
            return variables(p) + variables(q)
    assert False

# interpretacionesVar(n) es la lista de las interpretaciones de n
# variables. Por ejemplo,
# >>> interpretacionesVar 2
# [[False, False],
#  [False, True],
#  [True, False],
#  [True, True]]
def interpretacionesVar(n: int) -> list[list[bool]]:
    if n == 0:
        return [[]]
```

```

bss = interpretacionesVar(n-1)
return [[False] + x for x in bss] + [[True] + x for x in bss]

def interpretaciones(f: FProp) -> list[Interpretacion]:
    vs = list(set(variables(f)))
    return [list(zip(vs, i)) for i in interpretacionesVar(len(vs))]

```

5.9. El tipo de las fórmulas: Reconocedor de tautologías

En Haskell

```

-- -----
-- El tipo de las fórmulas proposicionales se puede definir por
--   data FProp = Const Bool
--               | Var Char
--               | Neg FProp
--               | Conj FProp FProp
--               | Impl FProp FProp
--   deriving Show
-- de modo que la fórmula  $A \rightarrow \perp \wedge \neg B$  se representa por
--   Impl (Var 'A') (Conj (Const False) (Neg (Var 'B'))))
--
-- Una fórmula es una tautología si es verdadera en todas sus
-- interpretaciones. Por ejemplo,
-- +  $(A \wedge B) \rightarrow A$  es una tautología
-- +  $A \rightarrow (A \wedge B)$  no es una tautología
--
-- Definir la función
--   esTautologia :: FProp -> Bool
-- tal que (esTautologia p) se verifica si la fórmula p es una
-- tautología. Por ejemplo,
--   λ> esTautologia (Impl (Conj (Var 'A') (Var 'B')) (Var 'A'))
--   True
--   λ> esTautologia (Impl (Var 'A') (Conj (Var 'A') (Var 'B'))))
--   False
-- -----

```

```

module Validez_de_una_formula where

```



```

import Data.List (nub)

data FProp = Const Bool
           | Var Char
           | Neg FProp
           | Conj FProp FProp
           | Impl FProp FProp
deriving Show

type Interpretacion = [(Char, Bool)]

esTautologia :: FProp -> Bool
esTautologia p =
  and [valor i p | i <- interpretaciones p]

-- (valor i p) es el valor de la fórmula p en la interpretación i. Por
-- ejemplo,
--   λ> p = Impl (Var 'A') (Conj (Var 'A') (Var 'B'))
--   λ> valor [('A',False),('B',False)] p
--   True
--   λ> valor [('A',True),('B',False)] p
--   False
valor :: Interpretacion -> FProp -> Bool
valor _ (Const b) = b
valor i (Var x)    = busca x i
valor i (Neg p)    = not (valor i p)
valor i (Conj p q) = valor i p && valor i q
valor i (Impl p q) = valor i p <= valor i q

-- (busca c t) es el valor del primer elemento de la lista de asociación
-- t cuya clave es c. Por ejemplo,
--   busca 2 [(1,'a'),(3,'d'),(2,'c')] == 'c'
busca :: Eq c => c -> [(c,v)] -> v
busca c t = head [v | (c',v) <- t, c == c']

interpretaciones :: FProp -> [Interpretacion]
interpretaciones p =
  [zip vs i | i <- interpretacionesVar (length vs)]
  where vs = nub (variables p)

```

```

-- (interpretacionesVar n) es la lista de las interpretaciones de n
-- variables. Por ejemplo,
--   λ> interpretacionesVar 2
--   [[False,False],
--    [False,True],
--    [True,False],
--    [True,True]]
interpretacionesVar :: Int -> [[Bool]]
interpretacionesVar 0 = [[]]
interpretacionesVar n = map (False:) bss ++ map (True:) bss
  where bss = interpretacionesVar (n-1)

-- (variables p) es la lista de las variables de la fórmula p. Por
-- ejemplo,
--   λ> variables (Impl (Var 'A') (Conj (Const False) (Neg (Var 'B'))))
--   "AB"
--   λ> variables (Impl (Var 'A') (Conj (Var 'A') (Neg (Var 'B'))))
--   "AAB"
variables :: FProp -> [Char]
variables (Const _) = []
variables (Var x)   = [x]
variables (Neg p)   = variables p
variables (Conj p q) = variables p ++ variables q
variables (Impl p q) = variables p ++ variables q

```

En Python

```

# -----
# El tipo de las fórmulas proposicionales se puede definir por
#   @dataclass
#   class FProp:
#       pass
#
#   @dataclass
#   class Const(FProp):
#       x: bool
#
#   @dataclass
#   class Var(FProp):

```

```

#         x: str
#
#     @dataclass
#     class Neg(FProp):
#         x: FProp
#
#     @dataclass
#     class Conj(FProp):
#         x: FProp
#         y: FProp
#
#     @dataclass
#     class Impl(FProp):
#         x: FProp
#         y: FProp
# de modo que la fórmula  $A \rightarrow \perp \wedge \neg B$  se representa por
#     Impl(Var('A'), Conj(Const(False), Neg (Var('B'))))
#
# Una fórmula es una tautología si es verdadera en todas sus
# interpretaciones. Por ejemplo,
# +  $(A \wedge B) \rightarrow A$  es una tautología
# +  $A \rightarrow (A \wedge B)$  no es una tautología
#
# Definir la función
#     esTautologia :: FProp -> Bool
# tal que (esTautologia p) se verifica si la fórmula p es una
# tautología. Por ejemplo,
#     >>> esTautologia(Impl(Conj(Var('A'), Var ('B')), Var('A')))
#     True
#     >>> esTautologia(Impl(Var('A'), Conj(Var('A'), Var('B'))))
#     False
# -----

```

```

from dataclasses import dataclass

```

```

@dataclass
class FProp:
    pass

```

```
@dataclass
class Const(FProp):
    x: bool
```

```
@dataclass
class Var(FProp):
    x: str
```

```
@dataclass
class Neg(FProp):
    x: FProp
```

```
@dataclass
class Conj(FProp):
    x: FProp
    y: FProp
```

```
@dataclass
class Impl(FProp):
    x: FProp
    y: FProp
```

```
Interpretacion = list[tuple[str, bool]]
```

```
# busca(c, t) es el valor del primer elemento de la lista de asociación
# t cuya clave es c. Por ejemplo,
# >>> busca('B', [('A', True), ('B', False), ('C', True)])
# False
```

```
def busca(c: str, i: Interpretacion) -> bool:
    return [v for (d, v) in i if d == c][0]
```

```
def valor(i: Interpretacion, f: FProp) -> bool:
    match f:
        case Const(b):
            return b
        case Var(x):
            return busca(x, i)
        case Neg(p):
            return not valor(i, p)
        case Conj(p, q):
```

```

        return valor(i, p) and valor(i, q)
    case Impl(p, q):
        return valor(i, p) <= valor(i, q)
assert False

# variables(p) es la lista de las variables de la fórmula p. Por
# ejemplo,
# >>> variables (Impl(Var('A'), Conj(Const(False), Neg (Var('B')))))
# ['A', 'B']
# >>> variables (Impl(Var('A'), Conj(Var('A'), Neg (Var('B')))))
# ['A', 'A', 'B']
def variables(f: FProp) -> list[str]:
    match f:
        case Const(_):
            return []
        case Var(x):
            return [x]
        case Neg(p):
            return variables(p)
        case Conj(p, q):
            return variables(p) + variables(q)
        case Impl(p, q):
            return variables(p) + variables(q)
    assert False

# interpretacionesVar(n) es la lista de las interpretaciones de n
# variables. Por ejemplo,
# >>> interpretacionesVar 2
# [[False, False],
#  [False, True],
#  [True, False],
#  [True, True]]
def interpretacionesVar(n: int) -> list[list[bool]]:
    if n == 0:
        return [[]]
    bss = interpretacionesVar(n-1)
    return [[False] + x for x in bss] + [[True] + x for x in bss]

# interpretaciones(p) es la lista de las interpretaciones de la fórmula
# p. Por ejemplo,

```

```
# >>> interpretaciones(Impl(Var('A'), Conj(Var('A'), Var('B'))))
# [[('B', False), ('A', False)],
#  [('B', False), ('A', True)],
#  [('B', True), ('A', False)],
#  [('B', True), ('A', True)]]
def interpretaciones(f: FProp) -> list[Interpretacion]:
    vs = list(set(variables(f)))
    return [list(zip(vs, i)) for i in interpretacionesVar(len(vs))]

def esTautologia(p: FProp) -> bool:
    return all((valor(i, p) for i in interpretaciones(p)))
```

5.10. Altura de un árbol binario

En Haskell

```
-- -----
-- El árbol binario
--
--      .
--     / \
--    /   \
--   .     .
--  / \   / \
-- 1  4 6 9
--
-- se puede representar por
--   ejArbol = Nodo (Nodo (Hoja 1) (Hoja 4))
--               (Nodo (Hoja 6) (Hoja 9))
--
-- El tipo de los árboles binarios se puede definir por
--   data Arbol a = Hoja a
--               | Nodo (Arbol a) (Arbol a)
--
-- Definir la función
--   altura :: Arbol a -> Int
-- tal que (altura t) es la altura del árbol t. Por ejemplo,
--   λ> altura (Hoja 1)
--   0
--   λ> altura (Nodo (Hoja 1) (Hoja 6))
--   1
--   λ> altura (Nodo (Nodo (Hoja 1) (Hoja 6)) (Hoja 2))
```

```
--      2
--      λ> altura (Nodo (Nodo (Hoja 1) (Hoja 6)) (Nodo (Hoja 2) (Hoja 7)))
--      2
--      -----
```

```
module Altura_de_un_arbol_binario where
```

```
data Arbol a = Hoja a
              | Nodo (Arbol a) (Arbol a)
```

```
altura :: Arbol a -> Int
altura (Hoja _) = 0
altura (Nodo i d) = 1 + max (altura i) (altura d)
```

En Python

```
# -----
# El árbol binario
#
#      .
#     / \
#    /   \
#   .     .
#  / \   / \
# 1  4 6  9
# se puede representar por
#   ejArbol = Nodo(Nodo(Hoja(1), Hoja(4)),
#                  Nodo(Hoja(6), Hoja(9)))
#
# El tipo de los árboles binarios se puede definir por
# @dataclass
# class Arbol(Generic[A]):
#     pass
#
# @dataclass
# class Hoja(Arbol[A]):
#     x: A
#
# @dataclass
# class Nodo(Arbol[A]):
#     i: Arbol
```

```

#         d: Arbol
#
# Definir la función
#     altura : (Arbol) -> int
# tal que altura(t) es la altura del árbol t. Por ejemplo,
#     >>> altura(Hoja(1))
#     0
#     >>> altura(Nodo(Hoja(1), Hoja(6)))
#     1
#     >>> altura(Nodo(Nodo(Hoja(1), Hoja(6)), Hoja(2)))
#     2
#     >>> altura(Nodo(Nodo(Hoja(1), Hoja(6)), Nodo(Hoja(2), Hoja(7))))
#     2
# -----

```

```

from dataclasses import dataclass
from typing import Generic, TypeVar

```

```

A = TypeVar("A")

```

```

@dataclass
class Arbol(Generic[A]):
    pass

```

```

@dataclass
class Hoja(Arbol[A]):
    x: A

```

```

@dataclass
class Nodo(Arbol[A]):
    i: Arbol[A]
    d: Arbol[A]

```

```

def altura(a: Arbol[A]) -> int:
    match a:
        case Hoja(_):
            return 0
        case Nodo(i, d):
            return 1 + max(altura(i), altura(d))
    assert False

```


5.11. Aplicación de una función a un árbol

En Haskell

```

-- -----
-- El árbol binario
--
--      .
--     / \
--    /   \
--   .     .
--  / \   / \
-- 1  4 6  9
--
-- se puede representar por
--   ejArbol = Nodo (Nodo (Hoja 1) (Hoja 4))
--               (Nodo (Hoja 6) (Hoja 9))
--
-- El tipo de los árboles binarios se puede definir por
--   data Arbol a = Hoja a
--               | Nodo (Arbol a) (Arbol a)
--   deriving (Show, Eq)
--
-- Definir la función
--   mapArbol :: (a -> b) -> Arbol a -> Arbol b
-- tal que (mapArbol f t) es el árbol obtenido aplicando la función f a
-- los elementos del árbol t. Por ejemplo,
--   λ> mapArbol (+ 1) (Nodo (Hoja 2) (Hoja 4))
--   Nodo (Hoja 3) (Hoja 5)
-- -----

```

```
module Aplicacion_de_una_funcion_a_un_arbol where
```

```
data Arbol a = Hoja a
              | Nodo (Arbol a) (Arbol a)
  deriving (Show, Eq)
```

```
mapArbol :: (a -> b) -> Arbol a -> Arbol b
mapArbol f (Hoja a)   = Hoja (f a)
mapArbol f (Nodo l r) = Nodo (mapArbol f l) (mapArbol f r)
```

En Python

```

# -----
# El árbol binario
#
#      .
#     / \
#    /   \
#   .     .
#  / \   / \
# 1  4 6  9
# se puede representar por
# ejArbol = Nodo(Nodo(Hoja(1), Hoja(4)),
#                Nodo(Hoja(6), Hoja(9)))
#
#
# El tipo de los árboles binarios se puede definir por
# @dataclass
# class Arbol(Generic[A]):
#     pass
#
# @dataclass
# class Hoja(Arbol[A]):
#     x: A
#
# @dataclass
# class Nodo(Arbol[A]):
#     i: Arbol
#     d: Arbol
#
# Definir la función
# mapArbol : (Callable[[A], B], Arbol[A]) -> Arbol[B]
# tal que mapArbol(f, t) es el árbol obtenido aplicando la función f a
# los elementos del árbol t. Por ejemplo,
# >>> mapArbol(lambda x: 1 + x, Nodo(Hoja(2), Hoja(4)))
#      Nodo(i=Hoja(x=3), d=Hoja(x=5))
# -----

from dataclasses import dataclass
from typing import Callable, Generic, TypeVar

A = TypeVar("A")

```

```

B = TypeVar("B")

@dataclass
class Arbol(Generic[A]):
    pass

@dataclass
class Hoja(Arbol[A]):
    x: A

@dataclass
class Nodo(Arbol[A]):
    i: Arbol[A]
    d: Arbol[A]

def mapArbol(f: Callable[[A], B], a: Arbol[A]) -> Arbol[B]:
    match a:
        case Hoja(x):
            return Hoja(f(x))
        case Nodo(i, d):
            return Nodo(mapArbol(f, i), mapArbol(f, d))
    assert False

```

5.12. Árboles con la misma forma

En Haskell

```

-- -----
-- El árbol binario
--
--      .
--     / \
--    /   \
--   .     .
--  / \   / \
-- 1  4 6 9
--
-- se puede representar por
--   ejArbol = Nodo (Nodo (Hoja 1) (Hoja 4))
--                (Nodo (Hoja 6) (Hoja 9))
--
-- El tipo de los árboles binarios se puede definir por

```

```
--      data Arbol a = Hoja a
--                      | Nodo (Arbol a) (Arbol a)
--      deriving (Show, Eq)
--
-- Definir la función
--      mismaForma :: Arbol a -> Arbol b -> Bool
-- tal que (mismaForma t1 t2) se verifica si t1 y t2 tienen la misma
-- estructura. Por ejemplo,
--      λ> arbol1 = Hoja 5
--      λ> arbol2 = Hoja 3
--      λ> mismaForma arbol1 arbol2
--      True
--      λ> arbol3 = Nodo (Hoja 6) (Hoja 7)
--      λ> mismaForma arbol1 arbol3
--      False
--      λ> arbol4 = Nodo (Hoja 9) (Hoja 5)
--      λ> mismaForma arbol3 arbol4
--      True
```

```
-----
module Arboles_con_la_misma_forma where
```

```
import Test.QuickCheck
```

```
data Arbol a = Hoja a
              | Nodo (Arbol a) (Arbol a)
              deriving (Show, Eq)
```

```
-- 1ª solución
-- =====
```

```
mismaForma1 :: Arbol a -> Arbol b -> Bool
mismaForma1 (Hoja _) (Hoja _)      = True
mismaForma1 (Nodo l r) (Nodo l' r') = mismaForma1 l l' && mismaForma1 r r'
mismaForma1 _          _           = False
```

```
-- 2ª solución
-- =====
```

```
mismaForma2 :: Arbol a -> Arbol b -> Bool
```

```

mismaForma2 x y = f x == f y
  where
    f = mapArbol (const ())

-- (mapArbol f t) es el árbol obtenido aplicando la función f a los
-- elementos del árbol t. Por ejemplo,
--   λ> mapArbol (+ 1) (Nodo (Hoja 2) (Hoja 4))
--   Nodo (Hoja 3) (Hoja 5)
mapArbol :: (a -> b) -> Arbol a -> Arbol b
mapArbol f (Hoja a)   = Hoja (f a)
mapArbol f (Nodo i d) = Nodo (mapArbol f i) (mapArbol f d)

-- Comprobación de equivalencia
-- =====

-- (arbolArbitrario n) es un árbol aleatorio de altura n. Por ejemplo,
--   λ> sample (arbolArbitrario 3 :: Gen (Arbol Int))
--   Nodo (Nodo (Nodo (Hoja 0) (Hoja 0)) (Hoja 0)) (Hoja 0)
--   Nodo (Nodo (Hoja 4) (Hoja 8)) (Hoja (-4))
--   Nodo (Nodo (Nodo (Hoja 4) (Hoja 10)) (Hoja (-6))) (Hoja (-1))
--   Nodo (Nodo (Hoja 3) (Hoja 6)) (Hoja (-5))
--   Nodo (Nodo (Hoja (-11)) (Hoja (-13))) (Hoja 14)
--   Nodo (Nodo (Hoja (-7)) (Hoja 15)) (Hoja (-2))
--   Nodo (Nodo (Hoja (-9)) (Hoja (-2))) (Hoja (-6))
--   Nodo (Nodo (Hoja (-15)) (Hoja (-16))) (Hoja (-20))
arbolArbitrario :: Arbitrary a => Int -> Gen (Arbol a)
arbolArbitrario n
  | n <= 1    = Hoja <$> arbitrary
  | otherwise = do
    k <- choose (2, n - 1)
    Nodo <$> arbolArbitrario k <*> arbolArbitrario (n - k)

-- Arbol es subclase de Arbitraria
instance Arbitrary a => Arbitrary (Arbol a) where
  arbitrary = sized arbolArbitrario
  shrink (Hoja x)   = Hoja <$> shrink x
  shrink (Nodo l r) = l :
    r :
    [Nodo l' r | l' <- shrink l] ++
    [Nodo l r' | r' <- shrink r]

```

```
-- La propiedad es
prop_mismaForma :: Arbol Int -> Arbol Int -> Property
prop_mismaForma a1 a2 =
    mismaForma1 a1 a2 ==> mismaForma2 a1 a2

-- La comprobación es
--    λ> quickCheck prop_mismaForma
--    +++ OK, passed 100 tests.
```

En Python

```
# -----
# El árbol binario
#
#      .
#     / \
#    /   \
#   .     .
#  / \   / \
# 1  4 6  9
# se puede representar por
#   ejArbol = Nodo(Nodo(Hoja(1), Hoja(4)),
#                  Nodo(Hoja(6), Hoja(9)))
#
#
# El tipo de los árboles binarios se puede definir por
# @dataclass
# class Arbol(Generic[A]):
#     pass
#
# @dataclass
# class Hoja(Arbol[A]):
#     x: A
#
# @dataclass
# class Nodo(Arbol[A]):
#     i: Arbol
#     d: Arbol
#
# Definir la función
```

```

#     mismaForma : (Arbol[A], Arbol[B]) -> bool
# tal que mismaForma(t1, t2) se verifica si t1 y t2 tienen la misma
# estructura. Por ejemplo,
#     >>> arbol1 = Hoja(5)
#     >>> arbol2 = Hoja(3)
#     >>> mismaForma(arbol1, arbol2)
#     True
#     >>> arbol3 = Nodo(Hoja(6), Hoja(7))
#     >>> mismaForma(arbol1, arbol3)
#     False
#     >>> arbol4 = Nodo(Hoja(9), Hoja(5))
#     >>> mismaForma(arbol3, arbol4)
#     True
# -----

```

```

from dataclasses import dataclass
from random import randint
from typing import Callable, Generic, TypeVar

```

```

from hypothesis import given
from hypothesis import strategies as st

```

```

A = TypeVar("A")
B = TypeVar("B")

```

```

@dataclass
class Arbol(Generic[A]):
    pass

```

```

@dataclass
class Hoja(Arbol[A]):
    x: A

```

```

@dataclass
class Nodo(Arbol[A]):
    i: Arbol[A]
    d: Arbol[A]

```

```

# -- 1ª solución
# -- =====

```

```

def mismaForma1(a: Arbol[A], b: Arbol[B]) -> bool:
    match (a, b):
        case (Hoja(_), Hoja(_)):
            return True
        case (Nodo(i1, d1), Nodo(i2, d2)):
            return mismaForma1(i1, i2) and mismaForma1(d1, d2)
        case (_, _):
            return False
    assert False

# -- 2ª solución
# -- =====

# mapArbol(f, t) es el árbol obtenido aplicando la función f a
# los elementos del árbol t. Por ejemplo,
#     >>> mapArbol(lambda x: 1 + x, Nodo(Hoja(2), Hoja(4)))
#     Nodo(i=Hoja(x=3), d=Hoja(x=5))
def mapArbol(f: Callable[[A], B], a: Arbol[A]) -> Arbol[B]:
    match a:
        case Hoja(x):
            return Hoja(f(x))
        case Nodo(i, d):
            return Nodo(mapArbol(f, i), mapArbol(f, d))
    assert False

def mismaForma2(a: Arbol[A], b: Arbol[B]) -> bool:
    return mapArbol(lambda x: 0, a) == mapArbol(lambda x: 0, b)

# Comprobación de equivalencia
# =====

# arbolArbitrario(n) es un árbol aleatorio de orden n. Por ejemplo,
#     >>> arbolArbitrario(3)
#     Nodo(i=Hoja(x=2), d=Nodo(i=Hoja(x=5), d=Hoja(x=2)))
#     >>> arbolArbitrario(3)
#     Nodo(i=Nodo(i=Hoja(x=6), d=Hoja(x=9)), d=Hoja(x=1))
def arbolArbitrario(n: int) -> Arbol[int]:
    if n == 0:
        return Hoja(randint(1, 10 * n))

```



```

    k = min(randint(1, n), n - 1)
    return Nodo(arbolArbitrario(k), arbolArbitrario(n - k))

# La propiedad es
@given(st.integers(min_value=1, max_value=10),
       st.integers(min_value=1, max_value=10))
def test_mismaForma(n1: int, n2: int) -> None:
    a1 = arbolArbitrario(n1)
    a2 = arbolArbitrario(n2)
    assert mismaForma1(a1, a2) == mismaForma2(a1, a2)

# La comprobación es
#   src> poetry run pytest -q arboles_con_la_misma_forma.py
#   1 passed in 0.22s

```

5.13. Árbol con las hojas en la profundidad dada

En Haskell

```

-- -----
-- El árbol binario
--
--      .
--     / \
--    /   \
--   .     .
--  / \   / \
-- 1  4 6  9
--
-- se puede representar por
--   ejArbol = Nodo (Nodo (Hoja 1) (Hoja 4))
--               (Nodo (Hoja 6) (Hoja 9))
--
-- El tipo de los árboles binarios se puede definir por
--   data Arbol a = Hoja a
--               | Nodo (Arbol a) (Arbol a)
--   deriving (Show, Eq)
--
-- Definir la función
--   creaArbol :: Int -> Arbol ()

```

```
-- tal que (creaArbol n) es el árbol cuyas hojas están en la profundidad
-- n. Por ejemplo,
--     λ> creaArbol 2
--     Nodo (Nodo (Hoja ()) (Hoja ())) (Nodo (Hoja ()) (Hoja ()))
-- -----
```

```
module Arbol_con_las_hojas_en_la_profundidad_dada where
```

```
data Arbol a = Hoja a
              | Nodo (Arbol a) (Arbol a)
  deriving (Show, Eq)
```

```
creaArbol :: Int -> Arbol ()
creaArbol h
  | h <= 0    = Hoja ()
  | otherwise = Nodo x x
  where x = creaArbol (h - 1)
```

En Python

```
# -----
# El árbol binario
#
#      .
#     / \
#    /   \
#   .     .
#  / \   / \
# 1  4 6  9
# se puede representar por
#   ejArbol = Nodo(Nodo(Hoja(1), Hoja(4)),
#                  Nodo(Hoja(6), Hoja(9)))
#
#
# El tipo de los árboles binarios se puede definir por
# @dataclass
# class Arbol(Generic[A]):
#     pass
#
# @dataclass
# class Hoja(Arbol[A]):
```

```

#         x: A
#
#     @dataclass
#     class Nodo(Arbol[A]):
#         i: Arbol
#         d: Arbol
#
# Definir la función
#     creaArbol : (int) -> Arbol[Any]:
# tal que creaArbol(n) es el árbol cuyas hojas están en la profundidad
# n. Por ejemplo,
#     >>> creaArbol(2)
#     Nodo(Nodo(Hoja(None), Hoja(None)), Nodo(Hoja(None), Hoja(None)))
# -----

from dataclasses import dataclass
from typing import Any, Generic, TypeVar

A = TypeVar("A")

@dataclass
class Arbol(Generic[A]):
    pass

@dataclass
class Hoja(Arbol[A]):
    x: A

@dataclass
class Nodo(Arbol[A]):
    i: Arbol[A]
    d: Arbol[A]

def creaArbol(h: int) -> Arbol[Any]:
    if h <= 0:
        return Hoja(None)
    x = creaArbol(h - 1)
    return Nodo(x, x)

```

5.14. El tipo de las expresiones aritméticas: Valor de una expresión

En Haskell

```

-----
-- Se considera el tipo de las expresiones aritméticas definido por
--   data Expr =
--       Lit Int
--       | Suma Expr Expr
--       | Op Expr
--       | SiCero Expr Expr Expr
--       deriving (Eq, Show)
-- formado por
-- + literales (p.e. Lit 7),
-- + sumas (p.e. Suma (Lit 7) (Suma (Lit 3) (Lit 5)))
-- + opuestos (p.e. Op (Suma (Op (Lit 7)) (Suma (Lit 3) (Lit 5))))
-- + expresiones condicionales (p.e. (SiCero (Lit 3) (Lit 4) (Lit 5)))
--
-- Definir la función
--   valor :: Expr -> Int
-- tal que (valor e) es el valor de la expresión e (donde el valor de
-- (SiCero e e1 e2) es el valor de e1 si el valor de e es cero y el es
-- el valor de e2, en caso contrario). Por ejemplo,
--   valor (Op (Suma (Lit 3) (Lit 5)))      == -8
--   valor (SiCero (Lit 0) (Lit 4) (Lit 5)) == 4
--   valor (SiCero (Lit 1) (Lit 4) (Lit 5)) == 5
-----

```

```
module Valor_de_una_expresion_aritmetica where
```

```

data Expr =
    Lit Int
  | Suma Expr Expr
  | Op Expr
  | SiCero Expr Expr Expr
  deriving (Eq, Show)

```

```

valor :: Expr -> Int
valor (Lit n)      = n

```

```

valor (Suma x y)      = valor x + valor y
valor (Op x)          = - valor x
valor (SiCero x y z) | valor x == 0 = valor y
                    | otherwise    = valor z

```

En Python

```

# -----
# Se considera el tipo de las expresiones aritméticas definido por
# @dataclass
# class Expr:
#     pass
#
# @dataclass
# class Lit(Expr):
#     x: int
#
# @dataclass
# class Suma(Expr):
#     x: Expr
#     y: Expr
#
# @dataclass
# class Op(Expr):
#     x: Expr
#
# @dataclass
# class SiCero(Expr):
#     x: Expr
#     y: Expr
#     z: Expr
#
# formado por
# + literales (p.e. Lit 7),
# + sumas (p.e. Suma (Lit 7) (Suma (Lit 3) (Lit 5)))
# + opuestos (p.e. Op (Suma (Op (Lit 7)) (Suma (Lit 3) (Lit 5))))
# + expresiones condicionales (p.e. (SiCero (Lit 3) (Lit 4) (Lit 5))
#
# Definir la función
# valor : (Expr) -> int

```

```
# tal que valor(e) es el valor de la expresión e (donde el valor de
# (SiCero e el e2) es el valor de e1 si el valor de e es cero y el es
# el valor de e2, en caso contrario). Por ejemplo,
# valor(Op(Suma(Lit(3), Lit(5)))) == -8
# valor(SiCero(Lit(0), Lit(4), Lit(5))) == 4
# valor(SiCero(Lit(1), Lit(4), Lit(5))) == 5
# -----
```

```
from dataclasses import dataclass
```

```
@dataclass
class Expr:
    pass
```

```
@dataclass
class Lit(Expr):
    x: int
```

```
@dataclass
class Suma(Expr):
    x: Expr
    y: Expr
```

```
@dataclass
class Op(Expr):
    x: Expr
```

```
@dataclass
class SiCero(Expr):
    x: Expr
    y: Expr
    z: Expr
```

```
def valor(e: Expr) -> int:
    match e:
        case Lit(n):
            return n
        case Suma(x, y):
            return valor(x) + valor(y)
```

```

        case Op(x):
            return -valor(x)
        case SiCero(x, y, z):
            return valor(y) if valor(x) == 0 else valor(z)
    assert False

# 2ª solución
# =====

def valor2(e: Expr) -> int:
    if isinstance(e, Lit):
        return e.x
    if isinstance(e, Suma):
        return valor2(e.x) + valor2(e.y)
    if isinstance(e, Op):
        return -valor2(e.x)
    if isinstance(e, SiCero):
        if valor2(e.x) == 0:
            return valor2(e.y)
        return valor2(e.z)
    assert False

```

5.15. El tipo de las expresiones aritméticas: Valor de la resta

En Haskell

```

-----
-- Se considera el tipo de las expresiones aritméticas definido por
--   data Expr = Lit Int
--             | Suma Expr Expr
--             | Op Expr
--             | SiCero Expr Expr Expr
--   deriving (Eq, Show)
-- formado por
-- + literales (p.e. Lit 7),
-- + sumas (p.e. Suma (Lit 7) (Suma (Lit 3) (Lit 5)))
-- + opuestos (p.e. Op (Suma (Op (Lit 7)) (Suma (Lit 3) (Lit 5))))
-- + expresiones condicionales (p.e. (SiCero (Lit 3) (Lit 4) (Lit 5))

```

```
--
-- La función para calcular el valor de una expresión es
--   valor :: Expr -> Int
--   valor (Lit n)           = n
--   valor (Suma x y)        = valor x + valor y
--   valor (Op x)            = - valor x
--   valor (SiCero x y z) | valor x == 0 = valor y
--                       | otherwise    = valor z
--
-- Definir la función
--   resta :: Expr -> Expr -> Expr
-- tal que (resta e1 e2) es la expresión correspondiente a la diferencia
-- de e1 y e2. Por ejemplo,
--   resta (Lit 42) (Lit 2) == Suma (Lit 42) (Op (Lit 2))
--
-- Comprobar con QuickCheck que
--   valor (resta x y) == valor x - valor y
-- -----
```

```
module Valor_de_la_resta where
```

```
import Test.QuickCheck
```

```
data Expr = Lit Int
          | Suma Expr Expr
          | Op Expr
          | SiCero Expr Expr Expr
  deriving (Eq, Show)
```

```
valor :: Expr -> Int
valor (Lit n)           = n
valor (Suma x y)        = valor x + valor y
valor (Op x)            = - valor x
valor (SiCero x y z) | valor x == 0 = valor y
                    | otherwise    = valor z
```

```
resta :: Expr -> Expr -> Expr
resta x y = Suma x (Op y)
```

```
-- Comprobación de la propiedad
```



```

-- =====

-- (exprArbitraria n) es una expresión aleatoria de tamaño n. Por
-- ejemplo,
--   λ> sample (exprArbitraria 3)
--   Op (Op (Lit 0))
--   SiCero (Lit 0) (Lit (-2)) (Lit (-1))
--   Op (Suma (Lit 3) (Lit 0))
--   Op (Lit 5)
--   Op (Lit (-1))
--   Op (Op (Lit 9))
--   Suma (Lit (-12)) (Lit (-12))
--   Suma (Lit (-9)) (Lit 10)
--   Op (Suma (Lit 8) (Lit 15))
--   SiCero (Lit 16) (Lit 9) (Lit (-5))
--   Suma (Lit (-3)) (Lit 1)
exprArbitraria :: Int -> Gen Expr
exprArbitraria n
  | n <= 1 = Lit <$> arbitrary
  | otherwise = oneof
    [ Lit <$> arbitrary
    , let m = div n 2
      in Suma <$> exprArbitraria m <*> exprArbitraria m
    , Op <$> exprArbitraria (n - 1)
    , let m = div n 3
      in SiCero <$> exprArbitraria m
        <*> exprArbitraria m
        <*> exprArbitraria m ]

-- Expr es subclase de Arbitrary
instance Arbitrary Expr where
  arbitrary = sized exprArbitraria

-- La propiedad es
prop_resta :: Expr -> Expr -> Property
prop_resta x y =
  valor (resta x y) == valor x - valor y

-- La comprobación es

```

```
--    λ> quickCheck prop_resta
--    +++ OK, passed 100 tests.
```

En Python

```
# -----
# Se considera el tipo de las expresiones aritméticas definido por
# @dataclass
# class Expr:
#     pass
#
# @dataclass
# class Lit(Expr):
#     x: int
#
# @dataclass
# class Suma(Expr):
#     x: Expr
#     y: Expr
#
# @dataclass
# class Op(Expr):
#     x: Expr
#
# @dataclass
# class SiCero(Expr):
#     x: Expr
#     y: Expr
#     z: Expr
#
# formado por
# + literales (p.e. Lit 7),
# + sumas (p.e. Suma (Lit 7) (Suma (Lit 3) (Lit 5)))
# + opuestos (p.e. Op (Suma (Op (Lit 7)) (Suma (Lit 3) (Lit 5))))
# + expresiones condicionales (p.e. (SiCero (Lit 3) (Lit 4) (Lit 5))
#
# La función para calcular el valor de una expresión es
# def valor(e: Expr) -> int:
#     match e:
#         case Lit(n):
```

```

#             return n
#         case Suma(x, y):
#             return valor(x) + valor(y)
#         case Op(x):
#             return -valor(x)
#         case SiCero(x, y, z):
#             return valor(y) if valor(x) == 0 else valor(z)
#     assert False
#
# Definir la función
#     resta : (Expr, Expr) -> Expr
# tal que resta(e1, e2) es la expresión correspondiente a la diferencia
# de e1 y e2. Por ejemplo,
#     resta(Lit(42), Lit(2)) == Suma(Lit(42), Op(Lit(2)))
#
# Comprobar con Hypothesis que
#     valor(resta(x, y)) == valor(x) - valor(y)
# -----

```

```

from dataclasses import dataclass
from random import choice, randint

```

```

from hypothesis import given
from hypothesis import strategies as st

```

```

@dataclass
class Expr:
    pass

```

```

@dataclass
class Lit(Expr):
    x: int

```

```

@dataclass
class Suma(Expr):
    x: Expr
    y: Expr

```

```

@dataclass

```

```

class Op(Expr):
    x: Expr

@dataclass
class SiCero(Expr):
    x: Expr
    y: Expr
    z: Expr

def valor(e: Expr) -> int:
    match e:
        case Lit(n):
            return n
        case Suma(x, y):
            return valor(x) + valor(y)
        case Op(x):
            return -valor(x)
        case SiCero(x, y, z):
            return valor(y) if valor(x) == 0 else valor(z)
    assert False

def resta(x: Expr, y: Expr) -> Expr:
    return Suma(x, Op(y))

# -- Comprobación de la propiedad
# -- =====

# exprArbitraria(n) es una expresión aleatoria de tamaño n. Por
# ejemplo,
# >>> exprArbitraria(3)
# Op(x=Op(x=Lit(x=9)))
# >>> exprArbitraria(3)
# Op(x=SiCero(x=Lit(x=6), y=Lit(x=2), z=Lit(x=6)))
# >>> exprArbitraria(3)
# Suma(x=Lit(x=8), y=Lit(x=2))
def exprArbitraria(n: int) -> Expr:
    if n <= 1:
        return Lit(randint(0, 10))
    m = n // 2
    return choice([Lit(randint(0, 10)),

```

```

        Suma(exprArbitraria(m), exprArbitraria(m)),
        Op(exprArbitraria(n - 1)),
        SiCero(exprArbitraria(m),
                exprArbitraria(m),
                exprArbitraria(m)))

# La propiedad es
@given(st.integers(min_value=1, max_value=10),
       st.integers(min_value=1, max_value=10))
def test_mismaForma(n1: int, n2: int) -> None:
    x = exprArbitraria(n1)
    y = exprArbitraria(n2)
    assert valor(resta(x, y)) == valor(x) - valor(y)

# La comprobación es
#   src> poetry run pytest -q valor_de_la Resta.py
#   1 passed in 0.21s

```

5.16. Número de hojas de un árbol binario

En Haskell

```

-- -----
-- El árbol binario
--           9
--         /  \
--       /    \
--      3      7
--     /  \
--    2    4
-- se puede representar por
--   N 9 (N 3 (H 2) (H 4)) (H 7)
--
-- El tipo de los árboles binarios se puede definir por
--   data Arbol a = H a
--               | N a (Arbol a) (Arbol a)
--   deriving (Show, Eq)
--
-- Definir las funciones
--   nHojas :: Arbol a -> Int

```

```
--      nNodos :: Arbol a -> Int
--      tales que
--      + (nHojas x) es el número de hojas del árbol x. Por ejemplo,
--          nHojas (N 9 (N 3 (H 2) (H 4)) (H 7)) == 3
--      + (nNodos x) es el número de nodos del árbol x. Por ejemplo,
--          nNodos (N 9 (N 3 (H 2) (H 4)) (H 7)) == 2
--
--      Comprobar con QuickCheck que en todo árbol binario el número de sus
--      hojas es igual al número de sus nodos más uno.
--      -----
```

```
{-# OPTIONS_GHC -fno-warn-unused-imports #-}
```

```
module Numero_de_hojas_de_un_arbol_binario where
```

```
import Test.QuickCheck
```

```
data Arbol a = H a
              | N a (Arbol a) (Arbol a)
  deriving (Show, Eq)
```

```
nHojas :: Arbol a -> Int
nHojas (H _)      = 1
nHojas (N _ i d) = nHojas i + nHojas d
```

```
nNodos :: Arbol a -> Int
nNodos (H _)      = 0
nNodos (N _ i d) = 1 + nNodos i + nNodos d
```

```
-- Comprobación de equivalencia
-- =====
```

```
-- (arbolArbitrario n) es un árbol aleatorio de altura n. Por ejemplo,
--      λ> sample (arbolArbitrario 3 :: Gen (Arbol Int))
--      N 0 (H 0) (H 0)
--      N 1 (N (-2) (H (-1)) (H 1)) (H 2)
--      N 3 (H 1) (H 2)
--      N 6 (N 0 (H 5) (H (-5))) (N (-5) (H (-5)) (H 4))
--      H 7
--      N (-8) (H (-8)) (H 9)
```

```

--      H 2
--      N (-1) (H 7) (N 9 (H (-2)) (H (-8)))
--      H (-3)
--      N 0 (N 16 (H (-14)) (H (-18))) (H 7)
--      N (-16) (H 18) (N (-19) (H (-15)) (H (-18)))
arbolArbitrario :: Arbitrary a => Int -> Gen (Arbol a)
arbolArbitrario 0 = H <$> arbitrary
arbolArbitrario n =
  oneof [H <$> arbitrary,
         N <$> arbitrary <*> arbolArbitrario (div n 2) <*> arbolArbitrario (div n 2)]

-- Arbol es subclase de Arbitrary
instance Arbitrary a => Arbitrary (Arbol a) where
  arbitrary = sized arbolArbitrario

-- La propiedad es
prop_nHojas :: Arbol Int -> Bool
prop_nHojas x =
  nHojas x == nNodos x + 1

-- La comprobación es
--      λ> quickCheck prop_nHojas
--      OK, passed 100 tests.

```

En Python

```

# -----
# El árbol binario
#
#      9
#     / \
#    /   \
#   3     7
#  / \
# 2   4
# se puede representar por
#      N(9, N(3, H(2), H(4)), H(7))
#
# El tipo de los árboles binarios se puede definir por
#      @dataclass
#      class Arbol(Generic[A]):

```

```

#         pass
#
#     @dataclass
#     class H(Arbol[A]):
#         x: A
#
#     @dataclass
#     class N(Arbol[A]):
#         x: A
#         i: Arbol[A]
#         d: Arbol[A]
#
# Definir las funciones
#     nHojas : (Arbol[A]) -> int
#     nNodos : (Arbol[A]) -> int
# tales que
# + nHojas(x) es el número de hojas del árbol x. Por ejemplo,
#     nHojas(N(9, N(3, H(2), H(4)), H(7))) == 3
# + nNodos(x) es el número de nodos del árbol x. Por ejemplo,
#     nNodos(N(9, N(3, H(2), H(4)), H(7))) == 2
#
# Comprobar con Hypothesis que en todo árbol binario el número de sus
# hojas es igual al número de sus nodos más uno.
# -----

from dataclasses import dataclass
from random import choice, randint
from typing import Generic, TypeVar

from hypothesis import given
from hypothesis import strategies as st

A = TypeVar("A")

@dataclass
class Arbol(Generic[A]):
    pass

@dataclass
class H(Arbol[A]):

```



```

    x: A

@dataclass
class N(Arbol[A]):
    x: A
    i: Arbol[A]
    d: Arbol[A]

def nHojas(a: Arbol[A]) -> int:
    match a:
        case H(_):
            return 1
        case N(_, i, d):
            return nHojas(i) + nHojas(d)
    assert False

def nNodos(a: Arbol[A]) -> int:
    match a:
        case H(_):
            return 0
        case N(_, i, d):
            return 1 + nNodos(i) + nNodos(d)
    assert False

# Comprobación de equivalencia
# =====

# (arbolArbitrario n) es un árbol aleatorio de orden n. Por ejemplo,
# >>> arbolArbitrario(4)
# N(x=2, i=H(x=1), d=H(x=9))
# >>> arbolArbitrario(4)
# H(x=10)
# >>> arbolArbitrario(4)
# N(x=4, i=N(x=7, i=H(x=4), d=H(x=0)), d=H(x=6))
def arbolArbitrario(n: int) -> Arbol[int]:
    if n <= 1:
        return H(randint(0, 10))
    m = n // 2
    return choice([H(randint(0, 10)),
                  N(randint(0, 10),

```

```

        arbolArbitrario(m),
        arbolArbitrario(m))])

# La propiedad es
@given(st.integers(min_value=1, max_value=10))
def test_nHojas(n: int) -> None:
    a = arbolArbitrario(n)
    assert nHojas(a) == nNodos(a) + 1

# La comprobación es
# src> poetry run pytest -q numero_de_hojas_de_un_arbol_binario.py
# 1 passed in 0.10s

```

5.17. Profundidad de un árbol binario

5.17.1. En Haskell

```

-- -----
-- El árbol binario
--
--      9
--     / \
--    /   \
--   3     7
--  / \
-- 2   4
-- se puede representar por
-- N 9 (N 3 (H 2) (H 4)) (H 7)
--
-- El tipo de los árboles binarios se puede definir por
-- data Arbol a = H a
--              | N a (Arbol a) (Arbol a)
--              deriving (Show, Eq)
--
-- Definir la función
-- profundidad :: Arbol a -> Int
-- tal que (profundidad x) es la profundidad del árbol x. Por ejemplo,
-- profundidad (N 9 (N 3 (H 2) (H 4)) (H 7)) == 2
-- profundidad (N 9 (N 3 (H 2) (N 1 (H 4) (H 5))) (H 7)) == 3
-- profundidad (N 4 (N 5 (H 4) (H 2)) (N 3 (H 7) (H 4))) == 2
--

```

```
-- Comprobar con QuickCheck que para todo árbol biario x, se tiene que
--   nNodos x <= 2^(profundidad x) - 1
-- -----
```

```
{-# OPTIONS_GHC -fno-warn-unused-imports #-}
```

```
module Profundidad_de_un_arbol_binario where
```

```
import Test.QuickCheck
```

```
data Arbol a = H a
              | N a (Arbol a) (Arbol a)
  deriving (Show, Eq)
```

```
profundidad :: Arbol a -> Int
profundidad (H _)      = 0
profundidad (N _ i d) = 1 + max (profundidad i) (profundidad d)
```

```
-- Comprobación de equivalencia
-- =====
```

```
-- (arbolArbitrario n) es un árbol aleatorio de altura n. Por ejemplo,
--   λ> sample (arbolArbitrario 3 :: Gen (Arbol Int))
--   N 0 (H 0) (H 0)
--   N 1 (N (-2) (H (-1)) (H 1)) (H 2)
--   N 3 (H 1) (H 2)
--   N 6 (N 0 (H 5) (H (-5))) (N (-5) (H (-5)) (H 4))
--   H 7
--   N (-8) (H (-8)) (H 9)
--   H 2
--   N (-1) (H 7) (N 9 (H (-2)) (H (-8)))
--   H (-3)
--   N 0 (N 16 (H (-14)) (H (-18))) (H 7)
--   N (-16) (H 18) (N (-19) (H (-15)) (H (-18)))
```

```
arbolArbitrario :: Arbitrary a => Int -> Gen (Arbol a)
```

```
arbolArbitrario 0 = H <$> arbitrary
```

```
arbolArbitrario n =
```

```
  oneof [H <$> arbitrary,
```

```
         N <$> arbitrary <*> arbolArbitrario (div n 2) <*> arbolArbitrario (div n 2)]
```

```

-- Arbol es subclase de Arbitrary
instance Arbitrary a => Arbitrary (Arbol a) where
    arbitrary = sized arbolArbitrario

-- La propiedad es
prop_nNodosProfundidad :: Arbol Int -> Bool
prop_nNodosProfundidad x =
    nNodos x <= 2 ^ profundidad x - 1

-- (nNodos x) es el número de nodos del árbol x. Por ejemplo,
--      nNodos (N 9 (N 3 (H 2) (H 4)) (H 7)) == 2
nNodos :: Arbol a -> Int
nNodos (H _)      = 0
nNodos (N _ i d) = 1 + nNodos i + nNodos d

-- La comprobación es
--      λ> quickCheck prop_nNodosProfundidad
--      OK, passed 100 tests.

```

En Python

```

# -----
# El árbol binario
#
#      9
#     / \
#    /   \
#   3     7
#  / \
# 2   4
# se puede representar por
#   N(9, N(3, H(2), H(4)), H(7))
#
# El tipo de los árboles binarios se puede definir por
#   @dataclass
#   class Arbol(Generic[A]):
#       pass
#
#   @dataclass
#   class H(Arbol[A]):
#       x: A

```

```

#
# @dataclass
# class N(Arbol[A]):
#     x: A
#     i: Arbol[A]
#     d: Arbol[A]
#
# Definir la función
# profundidad : (Arbol[A]) -> int
# tal que profundidad(x) es la profundidad del árbol x. Por ejemplo,
# profundidad(N(9, N(3, H(2), H(4)), H(7))) == 2
# profundidad(N(9, N(3, H(2), N(1, H(4), H(5))), H(7))) == 3
# profundidad(N(4, N(5, H(4), H(2)), N(3, H(7), H(4)))) == 2
#
# Comprobar con Hypothesis que para todo árbol biario x, se tiene que
# nNodos(x) <= 2^profundidad(x) - 1
# -----

from dataclasses import dataclass
from random import choice, randint
from typing import Generic, TypeVar

from hypothesis import given
from hypothesis import strategies as st

A = TypeVar("A")

@dataclass
class Arbol(Generic[A]):
    pass

@dataclass
class H(Arbol[A]):
    x: A

@dataclass
class N(Arbol[A]):
    x: A
    i: Arbol[A]
    d: Arbol[A]

```

```

def profundidad(a: Arbol[A]) -> int:
    match a:
        case H(_):
            return 0
        case N(_, i, d):
            return 1 + max(profundidad(i), profundidad(d))
    assert False

# Comprobación de equivalencia
# =====

# (arbolArbitrario n) es un árbol aleatorio de orden n. Por ejemplo,
# >>> arbolArbitrario(4)
# N(x=2, i=H(x=1), d=H(x=9))
# >>> arbolArbitrario(4)
# H(x=10)
# >>> arbolArbitrario(4)
# N(x=4, i=N(x=7, i=H(x=4), d=H(x=0)), d=H(x=6))
def arbolArbitrario(n: int) -> Arbol[int]:
    if n <= 1:
        return H(randint(0, 10))
    m = n // 2
    return choice([H(randint(0, 10)),
                  N(randint(0, 10),
                    arbolArbitrario(m),
                    arbolArbitrario(m))])

# nNodos(x) es el número de nodos del árbol x. Por ejemplo,
# nNodos(N(9, N(3, H(2), H(4)), H(7))) == 2
def nNodos(a: Arbol[A]) -> int:
    match a:
        case H(_):
            return 0
        case N(_, i, d):
            return 1 + nNodos(i) + nNodos(d)
    assert False

# La propiedad es
@given(st.integers(min_value=1, max_value=10))

```

```
def test_nHojas(n: int) -> None:
    a = arbolArbitrario(n)
    assert nNodos(a) <= 2 ** profundidad(a) - 1

# La comprobación es
#   src> poetry run pytest -q profundidad_de_un_arbol_binario.py
#   1 passed in 0.11s
```

5.18. Recorrido de árboles binarios

5.18.1. En Haskell

```
-- -----
-- El árbol binario
--
--      9
--     /\
--    /\
--   3  7
--  /\
-- 2  4
-- se puede representar por
--   N 9 (N 3 (H 2) (H 4)) (H 7)
--
-- El tipo de los árboles binarios se puede definir por
--   data Arbol a = H a
--                | N a (Arbol a) (Arbol a)
--   deriving (Show, Eq)
--
-- Definir las funciones
--   preorden :: Arbol a -> [a]
--   postorden :: Arbol a -> [a]
-- tales que
-- + (preorden x) es la lista correspondiente al recorrido preorden del
--   árbol x; es decir, primero visita la raíz del árbol, a continuación
--   recorre el subárbol izquierdo y, finalmente, recorre el subárbol
--   derecho. Por ejemplo,
--   preorden (N 9 (N 3 (H 2) (H 4)) (H 7)) == [9,3,2,4,7]
-- + (postorden x) es la lista correspondiente al recorrido postorden
--   del árbol x; es decir, primero recorre el subárbol izquierdo, a
--   continuación el subárbol derecho y, finalmente, la raíz del
```

```

-- árbol. Por ejemplo,
--     postorden (N 9 (N 3 (H 2) (H 4)) (H 7)) == [2,4,3,7,9]
--
-- Comprobar con QuickCheck que la longitud de la lista
-- obtenida recorriendo un árbol en cualquiera de los sentidos es igual
-- al número de nodos del árbol más el número de hojas.
-- -----

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module Recorrido_de_arboles_binarios where

import Test.QuickCheck

data Arbol a = H a
             | N a (Arbol a) (Arbol a)
             deriving (Show, Eq)

preorden :: Arbol a -> [a]
preorden (H x)      = [x]
preorden (N x i d) = x : preorden i ++ preorden d

postorden :: Arbol a -> [a]
postorden (H x)      = [x]
postorden (N x i d) = postorden i ++ postorden d ++ [x]

-- Comprobación de la propiedad
-- =====

-- (arbolArbitrario n) es un árbol aleatorio de altura n. Por ejemplo,
--     λ> sample (arbolArbitrario 3 :: Gen (Arbol Int))
--     N 0 (H 0) (H 0)
--     N 1 (N (-2) (H (-1)) (H 1)) (H 2)
--     N 3 (H 1) (H 2)
--     N 6 (N 0 (H 5) (H (-5))) (N (-5) (H (-5)) (H 4))
--     H 7
--     N (-8) (H (-8)) (H 9)
--     H 2
--     N (-1) (H 7) (N 9 (H (-2)) (H (-8)))
--     H (-3)

```



```

--      N 0 (N 16 (H (-14)) (H (-18))) (H 7)
--      N (-16) (H 18) (N (-19) (H (-15)) (H (-18)))
arbolArbitrario :: Arbitrary a => Int -> Gen (Arbol a)
arbolArbitrario 0 = H <$> arbitrary
arbolArbitrario n =
  oneof [H <$> arbitrary,
        N <$> arbitrary <*> arbolArbitrario (div n 2) <*> arbolArbitrario (div n 2)]

-- Arbol es subclase de Arbitrary
instance Arbitrary a => Arbitrary (Arbol a) where
  arbitrary = sized arbolArbitrario

-- La propiedad es
prop_longitud_recorrido :: Arbol Int -> Bool
prop_longitud_recorrido x =
  length (preorden x) == n &&
  length (postorden x) == n
  where n = nNodos x + nHojas x

-- (nNodos x) es el número de nodos del árbol x. Por ejemplo,
--      nNodos (N 9 (N 3 (H 2) (H 4)) (H 7)) == 2
nNodos :: Arbol a -> Int
nNodos (H _) = 0
nNodos (N _ i d) = 1 + nNodos i + nNodos d

-- (nHojas x) es el número de hojas del árbol x. Por ejemplo,
--      nHojas (N 9 (N 3 (H 2) (H 4)) (H 7)) == 3
nHojas :: Arbol a -> Int
nHojas (H _) = 1
nHojas (N _ i d) = nHojas i + nHojas d

-- La comprobación es
--      λ> quickCheck prop_longitud_recorrido
--      OK, passed 100 tests.

```

5.18.2. En Python

```

# -----
# El árbol binario
#      9

```

```

#      / \
#     /   \
#    3     7
#   / \
#  2   4
# se puede representar por
#   N(9, N(3, H(2), H(4)), H(7))
#
# El tipo de los árboles binarios se puede definir por
# @dataclass
# class Arbol(Generic[A]):
#     pass
#
# @dataclass
# class H(Arbol[A]):
#     x: A
#
# @dataclass
# class N(Arbol[A]):
#     x: A
#     i: Arbol[A]
#     d: Arbol[A]
#
# Definir las funciones
#   preorden : (Arbol[A]) -> list[A]
#   postorden : (Arbol[A]) -> list[A]
# tales que
# + preorden(x) es la lista correspondiente al recorrido preorden del
#   árbol x; es decir, primero visita la raíz del árbol, a continuación
#   recorre el subárbol izquierdo y, finalmente, recorre el subárbol
#   derecho. Por ejemplo,
#       >>> preorden(N(9, N(3, H(2), H(4)), H(7)))
#       [9, 3, 2, 4, 7]
# + (postorden x) es la lista correspondiente al recorrido postorden
#   del árbol x; es decir, primero recorre el subárbol izquierdo, a
#   continuación el subárbol derecho y, finalmente, la raíz del
#   árbol. Por ejemplo,
#       >>> postorden(N(9, N(3, H(2), H(4)), H(7)))
#       [2, 4, 3, 7, 9]
#

```

```
# Comprobar con Hypothesis que la longitud de la lista obtenida
# recorriendo un árbol en cualquiera de los sentidos es igual al número
# de nodos del árbol más el número de hojas.
# -----
```

```
from dataclasses import dataclass
from random import choice, randint
from typing import Generic, TypeVar

from hypothesis import given
from hypothesis import strategies as st
```

```
A = TypeVar("A")
```

```
@dataclass
class Arbol(Generic[A]):
    pass
```

```
@dataclass
class H(Arbol[A]):
    x: A
```

```
@dataclass
class N(Arbol[A]):
    x: A
    i: Arbol[A]
    d: Arbol[A]
```

```
def preorden(a: Arbol[A]) -> list[A]:
    match a:
        case H(x):
            return [x]
        case N(x, i, d):
            return [x] + preorden(i) + preorden(d)
    assert False
```

```
def postorden(a: Arbol[A]) -> list[A]:
    match a:
        case H(x):
            return [x]
```

```

        case N(x, i, d):
            return postorden(i) + postorden(d) + [x]
    assert False

# Comprobación de la propiedad
# =====

# (arbolArbitrario n) es un árbol aleatorio de orden n. Por ejemplo,
# >>> arbolArbitrario(4)
# N(x=2, i=H(x=1), d=H(x=9))
# >>> arbolArbitrario(4)
# H(x=10)
# >>> arbolArbitrario(4)
# N(x=4, i=N(x=7, i=H(x=4), d=H(x=0)), d=H(x=6))
def arbolArbitrario(n: int) -> Arbol[int]:
    if n <= 1:
        return H(randint(0, 10))
    m = n // 2
    return choice([H(randint(0, 10)),
                  N(randint(0, 10),
                    arbolArbitrario(m),
                    arbolArbitrario(m))])

# nNodos(x) es el número de nodos del árbol x. Por ejemplo,
# nNodos(N(9, N(3, H(2), H(4)), H(7))) == 2
def nNodos(a: Arbol[A]) -> int:
    match a:
        case H(_):
            return 0
        case N(_, i, d):
            return 1 + nNodos(i) + nNodos(d)
    assert False

# (nHojas x) es el número de hojas del árbol x. Por ejemplo,
# nHojas (N 9 (N 3 (H 2) (H 4)) (H 7)) == 3
def nHojas(a: Arbol[A]) -> int:
    match a:
        case H(_):
            return 1
        case N(_, i, d):

```

```

        return nHojas(i) + nHojas(d)
    assert False

# La propiedad es
@given(st.integers(min_value=1, max_value=10))
def test_recorrido(n: int) -> None:
    a = arbolArbitrario(n)
    m = nNodos(a) + nHojas(a)
    assert len(preorden(a)) == m
    assert len(postorden(a)) == m

# La comprobación es
#   src> poetry run pytest -q recorrido_de_arboles_binarios.py
#   1 passed in 0.16s

```

5.19. Imagen especular de un árbol binario

5.19.1. En Haskell

```

-- -----
-- El árbol binario
--
--      9
--     / \
--    /   \
--   3     7
--  / \
-- 2   4
-- se puede representar por
--   N 9 (N 3 (H 2) (H 4)) (H 7)
--
-- El tipo de los árboles binarios se puede definir por
--   data Arbol a = H a
--               | N a (Arbol a) (Arbol a)
--   deriving (Show, Eq)
--
-- Definir la función
--   espejo :: Arbol a -> Arbol a
-- tal que (espejo x) es la imagen especular del árbol x. Por ejemplo,
--   espejo (N 9 (N 3 (H 2) (H 4)) (H 7)) == N 9 (H 7) (N 3 (H 4) (H 2))
--

```

```

-- Comprobar con QuickCheck las siguientes propiedades, para todo árbol
-- x,
--     espejo (espejo x) = x
--     reverse (preorden (espejo x)) = postorden x
--     postorden (espejo x) = reverse (preorden x)
-----

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module Imagen_especular_de_un_arbol_binario where

import Test.QuickCheck

data Arbol a = H a
             | N a (Arbol a) (Arbol a)
             deriving (Show, Eq)

espejo :: Arbol a -> Arbol a
espejo (H x)      = H x
espejo (N x i d) = N x (espejo d) (espejo i)

-- Generador para las comprobaciones
-- =====

-- (arbolArbitrario n) es un árbol aleatorio de altura n. Por ejemplo,
--     λ> sample (arbolArbitrario 3 :: Gen (Arbol Int))
--     N 0 (H 0) (H 0)
--     N 1 (N (-2) (H (-1)) (H 1)) (H 2)
--     N 3 (H 1) (H 2)
--     N 6 (N 0 (H 5) (H (-5))) (N (-5) (H (-5)) (H 4))
--     H 7
--     N (-8) (H (-8)) (H 9)
--     H 2
--     N (-1) (H 7) (N 9 (H (-2)) (H (-8)))
--     H (-3)
--     N 0 (N 16 (H (-14)) (H (-18))) (H 7)
--     N (-16) (H 18) (N (-19) (H (-15)) (H (-18)))
arbolArbitrario :: Arbitrary a => Int -> Gen (Arbol a)
arbolArbitrario 0 = H <$> arbitrary
arbolArbitrario n =

```

```

oneof [H <$> arbitrary,
      N <$> arbitrary <*> arbolArbitrario (div n 2) <*> arbolArbitrario (div n 2)]

-- Arbol es subclase de Arbitrary
instance Arbitrary a => Arbitrary (Arbol a) where
  arbitrary = sized arbolArbitrario

-- Funciones auxiliares para la comprobación
-- =====

-- (preorden x) es la lista correspondiente al recorrido preorden del
-- árbol x; es decir, primero visita la raíz del árbol, a continuación
-- recorre el subárbol izquierdo y, finalmente, recorre el subárbol
-- derecho. Por ejemplo,
--   preorden (N 9 (N 3 (H 2) (H 4)) (H 7)) == [9,3,2,4,7]
preorden :: Arbol a -> [a]
preorden (H x)      = [x]
preorden (N x i d) = x : preorden i ++ preorden d

-- (postorden x) es la lista correspondiente al recorrido postorden
-- del árbol x; es decir, primero recorre el subárbol izquierdo, a
-- continuación el subárbol derecho y, finalmente, la raíz del
-- árbol. Por ejemplo,
--   postorden (N 9 (N 3 (H 2) (H 4)) (H 7)) == [2,4,3,7,9]
postorden :: Arbol a -> [a]
postorden (H x)      = [x]
postorden (N x i d) = postorden i ++ postorden d ++ [x]

-- Comprobación de las propiedades
-- =====

-- Las propiedades son
prop_espejo :: Arbol Int -> Bool
prop_espejo x =
  espejo (espejo x) == x &&
  reverse (preorden (espejo x)) == postorden x &&
  postorden (espejo x) == reverse (preorden x)

-- La comprobación es
--   λ> quickCheck prop_espejo

```

```
--      OK, passed 100 tests.
```

5.19.2. En Python

```
# -----
# El árbol binario
#
#      9
#     / \
#    /   \
#   3     7
#  / \
# 2   4
# se puede representar por
#   N(9, N(3, H(2), H(4)), H(7))
#
# El tipo de los árboles binarios se puede definir por
# @dataclass
# class Arbol(Generic[A]):
#     pass
#
# @dataclass
# class H(Arbol[A]):
#     x: A
#
# @dataclass
# class N(Arbol[A]):
#     x: A
#     i: Arbol[A]
#     d: Arbol[A]
#
# Definir la función
#   espejo : (Arbol[A]) -> Arbol[A]
# tal que espejo(x) es la imagen especular del árbol x. Por ejemplo,
#   espejo(N(9, N(3, H(2), H(4)), H(7))) == N(9, H(7), N(3, H(4), H(2)))
#
# Comprobar con Hypothesis las siguientes propiedades, para todo árbol
# x,
#   espejo(espejo(x)) = x
#   list(reversed(preorden(espejo(x)))) == postorden(x)
#   postorden(espejo(x)) == list(reversed(preorden(x)))
```



```

# -----

from dataclasses import dataclass
from random import choice, randint
from typing import Generic, TypeVar

from hypothesis import given
from hypothesis import strategies as st

A = TypeVar("A")

@dataclass
class Arbol(Generic[A]):
    pass

@dataclass
class H(Arbol[A]):
    x: A

@dataclass
class N(Arbol[A]):
    x: A
    i: Arbol[A]
    d: Arbol[A]

def espejo(a: Arbol[A]) -> Arbol[A]:
    match a:
        case H(x):
            return H(x)
        case N(x, i, d):
            return N(x, espejo(d), espejo(i))
    assert False

# Generador para las comprobaciones
# =====

# (arbolArbitrario n) es un árbol aleatorio de orden n. Por ejemplo,
# >>> arbolArbitrario(4)
# N(x=2, i=H(x=1), d=H(x=9))
# >>> arbolArbitrario(4)

```

```

#     H(x=10)
#     >>> arbolArbitrario(4)
#     N(x=4, i=N(x=7, i=H(x=4), d=H(x=0)), d=H(x=6))
def arbolArbitrario(n: int) -> Arbol[int]:
    if n <= 1:
        return H(randint(0, 10))
    m = n // 2
    return choice([H(randint(0, 10)),
                  N(randint(0, 10),
                    arbolArbitrario(m),
                    arbolArbitrario(m))])

# Funciones auxiliares para la comprobación
# =====

# preorden(x) es la lista correspondiente al recorrido preorden del
# árbol x; es decir, primero visita la raíz del árbol, a continuación
# recorre el subárbol izquierdo y, finalmente, recorre el subárbol
# derecho. Por ejemplo,
#     >>> preorden(N(9, N(3, H(2), H(4)), H(7)))
#     [9, 3, 2, 4, 7]
def preorden(a: Arbol[A]) -> list[A]:
    match a:
        case H(x):
            return [x]
        case N(x, i, d):
            return [x] + preorden(i) + preorden(d)
    assert False

# (postorden x) es la lista correspondiente al recorrido postorden
# del árbol x; es decir, primero recorre el subárbol izquierdo, a
# continuación el subárbol derecho y, finalmente, la raíz del
# árbol. Por ejemplo,
#     >>> postorden(N(9, N(3, H(2), H(4)), H(7)))
#     [2, 4, 3, 7, 9]
def postorden(a: Arbol[A]) -> list[A]:
    match a:
        case H(x):
            return [x]
        case N(x, i, d):

```

```

        return postorden(i) + postorden(d) + [x]
    assert False

# Comprobación de las propiedades
# =====

# Las propiedades son
@given(st.integers(min_value=1, max_value=10))
def test_espejo(n: int) -> None:
    x = arbolArbitrario(n)
    assert espejo(espejo(x)) == x
    assert list(reversed(preorden(espejo(x)))) == postorden(x)
    assert postorden(espejo(x)) == list(reversed(preorden(x)))

# La comprobación es
#   src> poetry run pytest -q imagen_especular_de_un_arbol_binario.py
#   1 passed in 0.16s

```

5.20. Subárbol de profundidad dada

5.20.1. En Haskell

```

-- -----
-- El árbol binario
--           9
--         /  \
--       /      \
--      3        7
--     /  \
--    2    4
-- se puede representar por
--   N 9 (N 3 (H 2) (H 4)) (H 7)
--
-- El tipo de los árboles binarios se puede definir por
--   data Arbol a = H a
--               | N a (Arbol a) (Arbol a)
--   deriving (Show, Eq)
--
-- La función take está definida por
--   take :: Int -> [a] -> [a]

```

```

--      take 0                = []
--      take (n+1) []        = []
--      take (n+1) (x:xs) = x : take n xs
--
-- Definir la función
--      takeArbol :: Int -> Arbol a -> Arbol a
-- tal que (takeArbol n t) es el subárbol de t de profundidad n. Por
-- ejemplo,
--      takeArbol 0 (N 9 (N 3 (H 2) (H 4)) (H 7)) == H 9
--      takeArbol 1 (N 9 (N 3 (H 2) (H 4)) (H 7)) == N 9 (H 3) (H 7)
--      takeArbol 2 (N 9 (N 3 (H 2) (H 4)) (H 7)) == N 9 (N 3 (H 2) (H 4)) (H 7)
--      takeArbol 3 (N 9 (N 3 (H 2) (H 4)) (H 7)) == N 9 (N 3 (H 2) (H 4)) (H 7)
--
-- Comprobar con QuickCheck que la profundidad de (takeArbol n x) es
-- menor o igual que n, para todo número natural n y todo árbol x.
-- -----

```

```

module Subarbol_de_profundidad_dada where

```

```

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

```

```

import Test.QuickCheck

```

```

data Arbol a = H a
              | N a (Arbol a) (Arbol a)
  deriving (Show, Eq)

```

```

takeArbol :: Int -> Arbol a -> Arbol a
takeArbol _ (H x)      = H x
takeArbol 0 (N x _ _) = H x
takeArbol n (N x i d) = N x (takeArbol (n-1) i) (takeArbol (n-1) d)

```

```

-- Generador para las comprobaciones
-- =====

```

```

-- (arbolArbitrario n) es un árbol aleatorio de altura n. Por ejemplo,
--      λ> sample (arbolArbitrario 3 :: Gen (Arbol Int))
--      N 0 (H 0) (H 0)
--      N 1 (N (-2) (H (-1)) (H 1)) (H 2)
--      N 3 (H 1) (H 2)

```

```

--      N 6 (N 0 (H 5) (H (-5))) (N (-5) (H (-5)) (H 4))
--      H 7
--      N (-8) (H (-8)) (H 9)
--      H 2
--      N (-1) (H 7) (N 9 (H (-2)) (H (-8)))
--      H (-3)
--      N 0 (N 16 (H (-14)) (H (-18))) (H 7)
--      N (-16) (H 18) (N (-19) (H (-15)) (H (-18)))
arbolArbitrario :: Arbitrary a => Int -> Gen (Arbol a)
arbolArbitrario 0 = H <$> arbitrary
arbolArbitrario n =
  oneof [H <$> arbitrary,
        N <$> arbitrary <*> arbolArbitrario (div n 2) <*> arbolArbitrario (div n 2)]

-- Arbol es subclase de Arbitrary
instance Arbitrary a => Arbitrary (Arbol a) where
  arbitrary = sized arbolArbitrario

-- Función auxiliar para la comprobación
-- =====

-- (profundidad x) es la profundidad del árbol x. Por ejemplo,
--      profundidad (N 9 (N 3 (H 2) (H 4)) (H 7))           == 2
--      profundidad (N 9 (N 3 (H 2) (N 1 (H 4) (H 5))) (H 7)) == 3
--      profundidad (N 4 (N 5 (H 4) (H 2)) (N 3 (H 7) (H 4))) == 2
profundidad :: Arbol a -> Int
profundidad (H _) = 0
profundidad (N _ i d) = 1 + max (profundidad i) (profundidad d)

-- Comprobación de la propiedad
-- =====

-- La propiedad es
prop_takeArbol :: Int -> Arbol Int -> Property
prop_takeArbol n x =
  n >= 0 ==> profundidad (takeArbol n x) <= n

-- La comprobación es
--      λ> quickCheck prop_takeArbol
--      +++ OK, passed 100 tests.

```

5.20.2. En Python

```

# -----
# El árbol binario
#
#      9
#     / \
#    /   \
#   3     7
#  / \
# 2   4
# se puede representar por
#   N(9, N(3, H(2), H(4)), H(7))
#
# El tipo de los árboles binarios se puede definir por
#   @dataclass
#   class Arbol(Generic[A]):
#       pass
#
#   @dataclass
#   class H(Arbol[A]):
#       x: A
#
#   @dataclass
#   class N(Arbol[A]):
#       x: A
#       i: Arbol[A]
#       d: Arbol[A]
#
# Definir la función
#   takeArbol : (int, Arbol[A]) -> Arbol[A]
# tal que takeArbol(n, t) es el subárbol de t de profundidad n. Por
# ejemplo,
#   >>> takeArbol(0, N(9, N(3, H(2), H(4)), H(7)))
#   H(9)
#   >>> takeArbol(1, N(9, N(3, H(2), H(4)), H(7)))
#   N(9, H(3), H(7))
#   >>> takeArbol(2, N(9, N(3, H(2), H(4)), H(7)))
#   N(9, N(3, H(2), H(4)), H(7))
#   >>> takeArbol(3, N(9, N(3, H(2), H(4)), H(7)))
#   N(9, N(3, H(2), H(4)), H(7))
#

```

```
# Comprobar con Hypothesis que la profundidad de takeArbol(n, x) es
# menor o igual que n, para todo número natural n y todo árbol x.
```

```
# -----
```

```
from dataclasses import dataclass
from random import choice, randint
from typing import Generic, TypeVar

from hypothesis import given
from hypothesis import strategies as st
```

```
A = TypeVar("A")
```

```
@dataclass
class Arbol(Generic[A]):
    pass
```

```
@dataclass
class H(Arbol[A]):
    x: A
```

```
@dataclass
class N(Arbol[A]):
    x: A
    i: Arbol[A]
    d: Arbol[A]
```

```
def takeArbol(n: int, a: Arbol[A]) -> Arbol[A]:
    match (n, a):
        case (_, H(x)):
            return H(x)
        case (0, N(x, _, _)):
            return H(x)
        case (n, N(x, i, d)):
            return N(x, takeArbol(n - 1, i), takeArbol(n - 1, d))
    assert False
```

```
# Generador para las comprobaciones
# =====
```

```

# (arbolArbitrario n) es un árbol aleatorio de orden n. Por ejemplo,
#   >>> arbolArbitrario(4)
#   N(x=2, i=H(x=1), d=H(x=9))
#   >>> arbolArbitrario(4)
#   H(x=10)
#   >>> arbolArbitrario(4)
#   N(x=4, i=N(x=7, i=H(x=4), d=H(x=0)), d=H(x=6))
def arbolArbitrario(n: int) -> Arbol[int]:
    if n <= 1:
        return H(randint(0, 10))
    m = n // 2
    return choice([H(randint(0, 10)),
                  N(randint(0, 10),
                    arbolArbitrario(m),
                    arbolArbitrario(m))])

# Función auxiliar para la comprobación
# =====

# profundidad(x) es la profundidad del árbol x. Por ejemplo,
#   profundidad(N(9, N(3, H(2), H(4)), H(7))) == 2
#   profundidad(N(9, N(3, H(2), N(1, H(4), H(5))), H(7))) == 3
#   profundidad(N(4, N(5, H(4), H(2)), N(3, H(7), H(4)))) == 2
def profundidad(a: Arbol[A]) -> int:
    match a:
        case H(_):
            return 0
        case N(_, i, d):
            return 1 + max(profundidad(i), profundidad(d))
    assert False

# Comprobación de la propiedad
# =====

# La propiedad es
@given(st.integers(min_value=0, max_value=12),
      st.integers(min_value=1, max_value=10))
def test_takeArbol(n: int, m: int) -> None:
    x = arbolArbitrario(m)
    assert profundidad(takeArbol(n, x)) <= n

```



```
# La comprobación es
# src> poetry run pytest -q subarbol_de_profundidad_dada.py
# 1 passed in 0.23s
```

5.21. Árbol de profundidad n con nodos iguales

5.21.1. En Haskell

```
-- -----
-- El árbol binario
--      9
--    /  \
--   /    \
--  3      7
-- /  \
-- 2   4
-- se puede representar por
-- N 9 (N 3 (H 2) (H 4)) (H 7)
--
-- El tipo de los árboles binarios se puede definir por
-- data Arbol a = H a
--              | N a (Arbol a) (Arbol a)
--              deriving (Show, Eq)
--
-- Definir las funciones
-- repeatArbol :: a -> Arbol a
-- replicateArbol :: Int -> a -> Arbol a
-- tales que
-- + (repeatArbol x) es es árbol con infinitos nodos x. Por ejemplo,
--   takeArbol 0 (repeatArbol 3) == H 3
--   takeArbol 1 (repeatArbol 3) == N 3 (H 3) (H 3)
--   takeArbol 2 (repeatArbol 3) == N 3 (N 3 (H 3) (H 3)) (N 3 (H 3) (H 3))
-- + (replicate n x) es el árbol de profundidad n cuyos nodos son x. Por
--   ejemplo,
--   replicateArbol 0 5 == H 5
--   replicateArbol 1 5 == N 5 (H 5) (H 5)
--   replicateArbol 2 5 == N 5 (N 5 (H 5) (H 5)) (N 5 (H 5) (H 5))
```

```

--
-- Comprobar con QuickCheck que el número de hojas de
-- (replicateArbol n x) es 2^n, para todo número natural n.
-- -----

module Arbol_de_profundidad_n_con_nodos_iguales where

import Test.QuickCheck

data Arbol a = H a
              | N a (Arbol a) (Arbol a)
  deriving (Show, Eq)

repeatArbol :: a -> Arbol a
repeatArbol x = N x t t
  where t = repeatArbol x

replicateArbol :: Int -> a -> Arbol a
replicateArbol n = takeArbol n . repeatArbol

-- (takeArbol n t) es el subárbol de t de profundidad n. Por ejemplo,
--   takeArbol 0 (N 9 (N 3 (H 2) (H 4)) (H 7)) == H 9
--   takeArbol 1 (N 9 (N 3 (H 2) (H 4)) (H 7)) == N 9 (H 3) (H 7)
--   takeArbol 2 (N 9 (N 3 (H 2) (H 4)) (H 7)) == N 9 (N 3 (H 2) (H 4)) (H 7)
--   takeArbol 3 (N 9 (N 3 (H 2) (H 4)) (H 7)) == N 9 (N 3 (H 2) (H 4)) (H 7)
takeArbol :: Int -> Arbol a -> Arbol a
takeArbol _ (H x)      = H x
takeArbol 0 (N x _ _) = H x
takeArbol n (N x i d) = N x (takeArbol (n-1) i) (takeArbol (n-1) d)

-- Generador para las comprobaciones
-- =====

-- (arbolArbitrario n) es un árbol aleatorio de altura n. Por ejemplo,
--   λ> sample (arbolArbitrario 3 :: Gen (Arbol Int))
--   N 0 (H 0) (H 0)
--   N 1 (N (-2) (H (-1)) (H 1)) (H 2)
--   N 3 (H 1) (H 2)
--   N 6 (N 0 (H 5) (H (-5))) (N (-5) (H (-5)) (H 4))
--   H 7

```

```

--      N (-8) (H (-8)) (H 9)
--      H 2
--      N (-1) (H 7) (N 9 (H (-2)) (H (-8)))
--      H (-3)
--      N 0 (N 16 (H (-14)) (H (-18))) (H 7)
--      N (-16) (H 18) (N (-19) (H (-15)) (H (-18)))
arbolArbitrario :: Arbitrary a => Int -> Gen (Arbol a)
arbolArbitrario 0 = H <$> arbitrary
arbolArbitrario n =
  oneof [H <$> arbitrary,
         N <$> arbitrary <*> arbolArbitrario (div n 2) <*> arbolArbitrario (div n 2)]

-- Arbol es subclase de Arbitrary
instance Arbitrary a => Arbitrary (Arbol a) where
  arbitrary = sized arbolArbitrario

-- Función auxiliar para la comprobación
-- =====

-- (nHojas x) es el número de hojas del árbol x. Por ejemplo,
--      nHojas (N 9 (N 3 (H 2) (H 4)) (H 7)) == 3
nHojas :: Arbol a -> Int
nHojas (H _)      = 1
nHojas (N _ i d) = nHojas i + nHojas d

-- Comprobación de la propiedad
-- =====

-- La propiedad es
prop_replicateArbol :: Int -> Int -> Property
prop_replicateArbol n x =
  n >= 0 ==> nHojas (replicateArbol n x) == 2^n

-- La comprobación es
--      λ> quickCheckWith (stdArgs {maxSize=7}) prop_replicateArbol
--      +++ OK, passed 100 tests.

```

5.21.2. En Python

```

# -----
# El árbol binario
#
#      9
#     / \
#    /   \
#   3     7
#  / \
# 2   4
# se puede representar por
#   N(9, N(3, H(2), H(4)), H(7))
#
# El tipo de los árboles binarios se puede definir por
#   @dataclass
#   class Arbol(Generic[A]):
#       pass
#
#   @dataclass
#   class H(Arbol[A]):
#       x: A
#
#   @dataclass
#   class N(Arbol[A]):
#       x: A
#       i: Arbol[A]
#       d: Arbol[A]
#
# Definir la función
#   replicateArbol : (int, A) -> Arbol[A]
# tal que (replicate n x) es el árbol de profundidad n cuyos nodos son
# x. Por ejemplo,
#   >>> replicateArbol(0, 5)
#   H(5)
#   >>> replicateArbol(1, 5)
#   N(5, H(5), H(5))
#   >>> replicateArbol(2, 5)
#   N(5, N(5, H(5), H(5)), N(5, H(5), H(5)))
#
# Comprobar con Hypothesis que el número de hojas de
# replicateArbol(n, x) es  $2^n$ , para todo número natural n.

```

```

# -----

from dataclasses import dataclass
from random import choice, randint
from typing import Generic, TypeVar

from hypothesis import given
from hypothesis import strategies as st

A = TypeVar("A")

@dataclass
class Arbol(Generic[A]):
    pass

@dataclass
class H(Arbol[A]):
    x: A

@dataclass
class N(Arbol[A]):
    x: A
    i: Arbol[A]
    d: Arbol[A]

def replicateArbol(n: int, x: A) -> Arbol[A]:
    match n:
        case 0:
            return H(x)
        case n:
            t = replicateArbol(n - 1, x)
            return N(x, t, t)
    assert False

# Generador para las comprobaciones
# =====

# (arbolArbitrario n) es un árbol aleatorio de orden n. Por ejemplo,
#     >>> arbolArbitrario(4)
#     N(x=2, i=H(x=1), d=H(x=9))

```

```

#     >>> arbolArbitrario(4)
#     H(x=10)
#     >>> arbolArbitrario(4)
#     N(x=4, i=N(x=7, i=H(x=4), d=H(x=0)), d=H(x=6))
def arbolArbitrario(n: int) -> Arbol[int]:
    if n <= 1:
        return H(randint(0, 10))
    m = n // 2
    return choice([H(randint(0, 10)),
                  N(randint(0, 10),
                    arbolArbitrario(m),
                    arbolArbitrario(m))])

# Función auxiliar para la comprobación
# =====

# nHojas(x) es el número de hojas del árbol x. Por ejemplo,
#     nHojas(N(9, N(3, H(2), H(4)), H(7))) == 3
def nHojas(a: Arbol[A]) -> int:
    match a:
        case H(_):
            return 1
        case N(_, i, d):
            return nHojas(i) + nHojas(d)
    assert False

# Comprobación de la propiedad
# =====

# La propiedad es
@given(st.integers(min_value=1, max_value=10),
       st.integers(min_value=1, max_value=10))
def test_nHojas(n: int, x: int) -> None:
    assert nHojas(replicateArbol(n, x)) == 2**n

# La comprobación es
#     src> poetry run pytest -q arbol_de_profundidad_n_con_nodos_iguales.py
#     1 passed in 0.20s

```

5.22. Suma de un árbol

5.22.1. En Haskell

```

-----
-- Los árboles binarios con valores en los nodos se pueden definir por
--   data Arbol a = H
--                 | N a (Arbol a) (Arbol a)
--   deriving (Show, Eq)
-- Por ejemplo, el árbol
--       9
--      / \
--     /   \
--    8     6
--   / \   / \
--  3  2 4  5
-- se puede representar por
--   N 9 (N 8 (N 3 H H) (N 2 H H)) (N 6 (N 4 H H) (N 5 H H))
--
-- Definir por recursión la función
--   sumaArbol :: Num a => Arbol1 a -> a
-- tal (sumaArbol x) es la suma de los valores que hay en el árbol
-- x. Por ejemplo,
--   sumaArbol (N 2 (N 5 (N 3 H H) (N 7 H H)) (N 4 H H)) == 21
-----

```

```
module Suma_de_un_arbol where
```

```

data Arbol1 a = H
              | N a (Arbol1 a) (Arbol1 a)
  deriving (Show, Eq)

sumaArbol :: Num a => Arbol1 a -> a
sumaArbol H           = 0
sumaArbol (N x i d) = x + sumaArbol i + sumaArbol d

```

5.22.2. En Python

```

# -----
# Los árboles binarios con valores en los nodos se pueden definir por

```

```

# @dataclass
# class Arbol:
#     pass
#
# @dataclass
# class H(Arbol):
#     pass
#
# @dataclass
# class N(Arbol):
#     x: int
#     i: Arbol
#     d: Arbol
# Por ejemplo, el árbol
#
#      9
#     / \
#    /   \
#   8     6
#  / \   / \
# 3  2 4  5
# se puede representar por
# N(9, N(8, N(3, H(), H()), N(2, H(), H()))), N(6, N(4, H(), H()), N(5, H(), H(
#
# Definir la función
# sumaArbol : (Arbol) -> int
# tal sumaArbol(x) es la suma de los valores que hay en el árbol x.
# Por ejemplo,
# >>> sumaArbol(N(2, N(5, N(3, H(), H()), N(7, H(), H()))), N(4, H(), H()))
# 21
# -----

```

```

from dataclasses import dataclass

```

```

@dataclass
class Arbol:
    pass

```

```

@dataclass
class H(Arbol):

```



```

pass

@dataclass
class N(Arbol):
    x: int
    i: Arbol
    d: Arbol

def sumaArbol(a: Arbol) -> int:
    match a:
        case H():
            return 0
        case N(x, i, d):
            return x + sumaArbol(i) + sumaArbol(d)
    assert False

```

5.23. Rama izquierda de un árbol binario

5.23.1. En Haskell

```

-- -----
-- Los árboles binarios con valores en los nodos se pueden definir por
--   data Arbol a = H
--               | N a (Arbol a) (Arbol a)
--   deriving (Show, Eq)
-- Por ejemplo, el árbol
--       9
--      / \
--     /   \
--    8     6
--   / \   / \
--  3  2 4 5
-- se puede representar por
--   N 9 (N 8 (N 3 H H) (N 2 H H)) (N 6 (N 4 H H) (N 5 H H))
--
-- Definir la función
--   ramaIzquierda :: Arbol a -> [a]
-- tal que (ramaIzquierda a) es la lista de los valores de los nodos de
-- la rama izquierda del árbol a. Por ejemplo,
--   λ> ramaIzquierda (N 2 (N 5 (N 3 H H) (N 7 H H)) (N 4 H H))

```

```
--      [2,5,3]
```

```
-----
```

```
module Rama_izquierda_de_un_arbol_binario where
```

```
data Arbol a = H
              | N a (Arbol a) (Arbol a)
  deriving (Show, Eq)
```

```
ramaIzquierda :: Arbol a -> [a]
ramaIzquierda H      = []
ramaIzquierda (N x i _) = x : ramaIzquierda i
```

5.23.2. En Python

```
# -----
# Los árboles binarios con valores en los nodos se pueden definir por
# @dataclass
# class Arbol(Generic[A]):
#     pass
#
# @dataclass
# class H(Arbol[A]):
#     pass
#
# @dataclass
# class N(Arbol[A]):
#     x: A
#     i: Arbol[A]
#     d: Arbol[A]
# Por ejemplo, el árbol
#      9
#     / \
#    /   \
#   8     6
#  / \   / \
# 3  2 4  5
# se puede representar por
# N(9, N(8, N(3, H(), H()), N(2, H(), H()))), N(6, N(4, H(), H()), N(5, H(), H(
#
```

```
# Definir la función
#   ramaIzquierda : (Arbol[A]) -> list[A]
# tal que ramaIzquierda(a) es la lista de los valores de los nodos de
# la rama izquierda del árbol a. Por ejemplo,
#   >>> ramaIzquierda(N(2, N(5, N(3, H(), H())), N(7, H(), H())), N(4, H(), H()))
#   [2, 5, 3]
# -----
```

```
from dataclasses import dataclass
from typing import Generic, TypeVar
```

```
A = TypeVar("A")
```

```
@dataclass
class Arbol(Generic[A]):
    pass
```

```
@dataclass
class H(Arbol[A]):
    pass
```

```
@dataclass
class N(Arbol[A]):
    x: A
    i: Arbol[A]
    d: Arbol[A]
```

```
def ramaIzquierda(a: Arbol[A]) -> list[A]:
    match a:
        case H():
            return []
        case N(x, i, _):
            return [x] + ramaIzquierda(i)
    assert False
```

5.24. Árboles balanceados

5.24.1. En Haskell

```

-- -----
-- Los árboles binarios con valores en los nodos se pueden definir por
--   data Arbol a = H
--               | N a (Arbol a) (Arbol a)
--   deriving (Show, Eq)
-- Por ejemplo, el árbol
--       9
--      / \
--     /   \
--    8     6
--   / \   / \
--  3  2 4  5
-- se puede representar por
--   N 9 (N 8 (N 3 H H) (N 2 H H)) (N 6 (N 4 H H) (N 5 H H))
--
-- Diremos que un árbol está balanceado si para cada nodo la diferencia
-- entre el número de nodos de sus subárboles izquierdo y derecho es
-- menor o igual que uno.
--
-- Definir la función
--   balanceado :: Arbol a -> Bool
-- tal que (balanceado a) se verifica si el árbol a está balanceado. Por
-- ejemplo,
--   λ> balanceado (N 5 H (N 3 H H))
--   True
--   λ> balanceado (N 4 (N 3 (N 2 H H) H) (N 5 H (N 6 H (N 7 H H))))
--   False
-- -----

```

```

module Arboles_balanceados where

```

```

data Arbol a = H
              | N a (Arbol a) (Arbol a)
  deriving (Show, Eq)

```

```

balanceado :: Arbol a -> Bool

```

```

balanceado H = True
balanceado (N _ i d) = abs (numeroNodos i - numeroNodos d) <= 1
                        && balanceado i
                        && balanceado d

-- (numeroNodos a) es el número de nodos del árbol a. Por ejemplo,
--     numeroNodos (N 5 H (N 3 H H)) == 2
numeroNodos :: Arbol a -> Int
numeroNodos H = 0
numeroNodos (N _ i d) = 1 + numeroNodos i + numeroNodos d

```

5.24.2. En Python

```

# -----
# Los árboles binarios con valores en los nodos se pueden definir por
# @dataclass
# class Arbol(Generic[A]):
#     pass
#
# @dataclass
# class H(Arbol[A]):
#     pass
#
# @dataclass
# class N(Arbol[A]):
#     x: A
#     i: Arbol[A]
#     d: Arbol[A]
# Por ejemplo, el árbol
#
#       9
#      / \
#     /   \
#    8     6
#   / \   / \
#  3  2 4 5
# se puede representar por
#     N(9, N(8, N(3, H()), H()), N(2, H(), H())), N(6, N(4, H(), H()), N(5, H(), H(
#
# Diremos que un árbol está balanceado si para cada nodo la diferencia
# entre el número de nodos de sus subárboles izquierdo y derecho es

```

```

# menor o igual que uno.
#
# Definir la función
#     balanceado : (Arbol[A]) -> bool
# tal que balanceado(a) se verifica si el árbol a está balanceado. Por
# ejemplo,
#     >>> balanceado(N(5, H(), N(3, H(), H())))
#     True
#     >>> balanceado(N(4, N(3, N(2, H(), H()), H()), H()), N(5, H(), N(6, H(), N(7, H(),
#     False
# -----

from dataclasses import dataclass
from typing import Generic, TypeVar

A = TypeVar("A")

@dataclass
class Arbol(Generic[A]):
    pass

@dataclass
class H(Arbol[A]):
    pass

@dataclass
class N(Arbol[A]):
    x: A
    i: Arbol[A]
    d: Arbol[A]

def numeroNodos(a: Arbol[A]) -> int:
    match a:
        case H():
            return 0
        case N(_, i, d):
            return 1 + numeroNodos(i) + numeroNodos(d)
    assert False

def balanceado(a: Arbol[A]) -> bool:

```

```

match a:
  case H():
    return True
  case N(_, i, d):
    return abs(numeroNodos(i) - numeroNodos(d)) <= 1 \
      and balanceado(i) and balanceado(d)
assert False

```

5.25. Árboles con bordes iguales

5.25.1. En Haskell

```

-----
-- Los árboles binarios con valores en las hojas se pueden definir por
--   data Arbol a = H a
--                   | N (Arbol a) (Arbol a)
--                   deriving Show
-- Por ejemplo, los árboles
--   árbol1      árbol2      árbol3      árbol4
--       0          0          0          0
--      / \       / \       / \       / \
--     1  0      0  3      0  3      0  1
--    / \     / \     / \     / \
--   2  3    1  2    1  4    2  3
-- se representan por
--   arbol1, arbol2, arbol3, arbol4 :: Arbol Int
--   arbol1 = N (H 1) (N (H 2) (H 3))
--   arbol2 = N (N (H 1) (H 2)) (H 3)
--   arbol3 = N (N (H 1) (H 4)) (H 3)
--   arbol4 = N (N (H 2) (H 3)) (H 1)
--
-- Definir la función
--   igualBorde :: Eq a => Arbol a -> Arbol a -> Bool
-- tal que (igualBorde t1 t2) se verifica si los bordes de los árboles
-- t1 y t2 son iguales. Por ejemplo,
--   igualBorde arbol1 arbol2 == True
--   igualBorde arbol1 arbol3 == False
--   igualBorde arbol1 arbol4 == False
-----

```

```
module Arboles_con_bordes_iguales where
```

```
data Arbol a = N (Arbol a) (Arbol a)
             | H a
             deriving Show
```

```
arbol1, arbol2, arbol3, arbol4 :: Arbol Int
arbol1 = N (H 1) (N (H 2) (H 3))
arbol2 = N (N (H 1) (H 2)) (H 3)
arbol3 = N (N (H 1) (H 4)) (H 3)
arbol4 = N (N (H 2) (H 3)) (H 1)
```

```
igualBorde :: Eq a => Arbol a -> Arbol a -> Bool
igualBorde t1 t2 = borde t1 == borde t2
```

```
-- (borde t) es el borde del árbol t; es decir, la lista de las hojas
-- del árbol t leídas de izquierda a derecha. Por ejemplo,
--   borde arbol4 == [2,3,1]
```

```
borde :: Arbol a -> [a]
borde (N i d) = borde i ++ borde d
borde (H x)   = [x]
```

5.25.2. En Python

```
# -----
# Los árboles binarios con valores en las hojas se pueden definir por
#   @dataclass
#   class Arbol(Generic[A]):
#       pass
#
#   @dataclass
#   class H(Arbol[A]):
#       x: A
#
#   @dataclass
#   class N(Arbol[A]):
#       i: Arbol[A]
#       d: Arbol[A]
# Por ejemplo, los árboles
#   árbol1          árbol2          árbol3          árbol4
```



```

#           o           o           o           o
#         / \         / \         / \         / \
#        1  o       o  3       o  3       o  1
#         / \         / \         / \         / \
#        2  3       1  2       1  4       2  3
# se representan por
#   arbol1: Arbol[int] = N(H(1), N(H(2), H(3)))
#   arbol2: Arbol[int] = N(N(H(1), H(2)), H(3))
#   arbol3: Arbol[int] = N(N(H(1), H(4)), H(3))
#   arbol4: Arbol[int] = N(N(H(2), H(3)), H(1))
#
# Definir la función
#   igualBorde : (Arbol[A], Arbol[A]) -> bool
# tal que igualBorde(t1, t2) se verifica si los bordes de los árboles
# t1 y t2 son iguales. Por ejemplo,
#   igualBorde(arbol1, arbol2) == True
#   igualBorde(arbol1, arbol3) == False
#   igualBorde(arbol1, arbol4) == False
# -----

```

```

from dataclasses import dataclass
from typing import Generic, TypeVar

```

```
A = TypeVar("A")
```

```

@dataclass
class Arbol(Generic[A]):
    pass

```

```

@dataclass
class H(Arbol[A]):
    x: A

```

```

@dataclass
class N(Arbol[A]):
    i: Arbol[A]
    d: Arbol[A]

```

```

arbol1: Arbol[int] = N(H(1), N(H(2), H(3)))
arbol2: Arbol[int] = N(N(H(1), H(2)), H(3))

```

```

arbol3: Arbol[int] = N(N(H(1), H(4)), H(3))
arbol4: Arbol[int] = N(N(H(2), H(3)), H(1))

# borde(t) es el borde del árbol t; es decir, la lista de las hojas
# del árbol t leídas de izquierda a derecha. Por ejemplo,
#   borde(arbol4) == [2, 3, 1]
def borde(a: Arbol[A]) -> List[A]:
    match a:
        case H(x):
            return [x]
        case N(i, d):
            return borde(i) + borde(d)
    assert False

def igualBorde(t1: Arbol[A], t2: Arbol[A]) -> bool:
    return borde(t1) == borde(t2)

```

5.26. Árboles con igual estructura

5.26.1. En Haskell

```

-- -----
-- Los árboles binarios con valores en las hojas y en los nodos se
-- definen por
--   data Arbol a = H a
--                 | N a (Arbol a) (Arbol a)
--   deriving Show
-- Por ejemplo, los árboles
--
--           5           8           5           5
--        /  \        /  \        /  \        /  \
--       /    \       /    \       /    \       /    \
--      9      7      9     3      9     2      4     7
--     / \   / \   / \   / \   / \   / \   / \
--    1  4 6 8 1  4 6 2 1  4 1  4 6 2 6  2
--
-- se pueden representar por
--   ej3arbol1, ej3arbol2, ej3arbol3, ej3arbol4 :: Arbol Int
--   ej3arbol1 = N 5 (N 9 (H 1) (H 4)) (N 7 (H 6) (H 8))
--   ej3arbol2 = N 8 (N 9 (H 1) (H 4)) (N 3 3 (H 6) (H 2))
--   ej3arbol3 = N 5 (N 9 (H 1) (H 4)) (H 2)
--   ej3arbol4 = N 5 (H 4) (N 7 (H 6) (H 2))

```

```
--
-- Definir la función
--   igualEstructura :: Arbol -> Arbol -> Bool
-- tal que (igualEstructura a1 a2) se verifica si los árboles a1 y a2
-- tienen la misma estructura. Por ejemplo,
--   igualEstructura ej3arbol1 ej3arbol2 == True
--   igualEstructura ej3arbol1 ej3arbol3 == False
--   igualEstructura ej3arbol1 ej3arbol4 == False
-- -----
```

```
module Arboles_con_igual_estructura where
```

```
data Arbol a = H a
              | N a (Arbol a) (Arbol a)
  deriving (Show, Eq)
```

```
ej3arbol1, ej3arbol2, ej3arbol3, ej3arbol4 :: Arbol Int
ej3arbol1 = N 5 (N 9 (H 1) (H 4)) (N 7 (H 6) (H 8))
ej3arbol2 = N 8 (N 9 (H 1) (H 4)) (N 3 (H 6) (H 2))
ej3arbol3 = N 5 (N 9 (H 1) (H 4)) (H 2)
ej3arbol4 = N 5 (H 4) (N 7 (H 6) (H 2))
```

```
igualEstructura :: Arbol a -> Arbol a -> Bool
igualEstructura (H _) (H _)                = True
igualEstructura (N _ i1 d1) (N _ i2 d2) =
  igualEstructura i1 i2 &&
  igualEstructura d1 d2
igualEstructura _ _                        = False
```

5.26.2. En Python

```
# -----
# Los árboles binarios con valores en las hojas y en los nodos se
# definen por
#   @dataclass
#   class Arbol(Generic[A]):
#       pass
#
#   @dataclass
#   class H(Arbol[A]):
```

```

#         x: A
#
#     @dataclass
#     class N(Arbol[A]):
#         x: A
#         i: Arbol[A]
#         d: Arbol[A]
# Por ejemplo, los árboles
#
#         5           8           5           5
#        / \        / \        / \        / \
#       /   \      /   \      /   \      /   \
#      9     7    9     3    9     2    4     7
#     / \  / \  / \  / \      / \      / \
#    1  4 6  8  1  4 6  2  1  4      6  2
# se pueden representar por
#     ej3arbol1: Arbol[int] = N(5, N(9, H(1), H(4)), N(7, H(6), H(8)))
#     ej3arbol2: Arbol[int] = N(8, N(9, H(1), H(4)), N(3, H(6), H(2)))
#     ej3arbol3: Arbol[int] = N(5, N(9, H(1), H(4)), H(2))
#     ej3arbol4: Arbol[int] = N(5, H(4), N(7, H(6), H(2)))
#
# Definir la función
#     igualEstructura : (Arbol[A], Arbol[A]) -> bool
# tal que igualEstructura(a1, a2) se verifica si los árboles a1 y a2
# tienen la misma estructura. Por ejemplo,
#     igualEstructura(ej3arbol1, ej3arbol2) == True
#     igualEstructura(ej3arbol1, ej3arbol3) == False
#     igualEstructura(ej3arbol1, ej3arbol4) == False
# -----

```

```

from dataclasses import dataclass
from typing import Generic, TypeVar

```

```

A = TypeVar("A")

```

```

@dataclass
class Arbol(Generic[A]):
    pass

```

```

@dataclass
class H(Arbol[A]):

```

```

x: A

@dataclass
class N(Arbol[A]):
    x: A
    i: Arbol[A]
    d: Arbol[A]

ej3arbol1: Arbol[int] = N(5, N(9, H(1), H(4)), N(7, H(6), H(8)))
ej3arbol2: Arbol[int] = N(8, N(9, H(1), H(4)), N(3, H(6), H(2)))
ej3arbol3: Arbol[int] = N(5, N(9, H(1), H(4)), H(2))
ej3arbol4: Arbol[int] = N(5, H(4), N(7, H(6), H(2)))

def igualEstructura(a: Arbol[A], b: Arbol[A]) -> bool:
    match (a, b):
        case (H(_), H(_)):
            return True
        case (N(_, i1, d1), N(_, i2, d2)):
            return igualEstructura(i1, i2) and igualEstructura(d1, d2)
        case (_, _):
            return False
    assert False

```

5.27. Existencia de elementos del árbol que verifican una propiedad

5.27.1. En Haskell

```

-- -----
-- Los árboles binarios con valores en las hojas y en los nodos se
-- definen por
--     data Arbol a = H a
--                   | N a (Arbol a) (Arbol a)
--     deriving Show
-- Por ejemplo, el árbol
--           5
--        /  \
--       /    \
--      /      \
--     3        2

```

```

--      / \
--     1   4
-- se representa por
--     N 5 (N 3 (H 1) (H 4)) (H 2)
--
-- Definir la función
--     algunoArbol :: Arbol t -> (t -> Bool) -> Bool
-- tal que (algunoArbol a p) se verifica si algún elemento del árbol a
-- cumple la propiedad p. Por ejemplo,
--     algunoArbol (N 5 (N 3 (H 1) (H 4)) (H 2)) (>4) == True
--     algunoArbol (N 5 (N 3 (H 1) (H 4)) (H 2)) (>7) == False
-- -----

```

```

module Existencia_de_elemento_del_arbol_con_propiedad where

```

```

data Arbol a = H a
              | N a (Arbol a) (Arbol a)
deriving Show

```

```

algunoArbol :: Arbol a -> (a -> Bool) -> Bool
algunoArbol (H x) p      = p x
algunoArbol (N x i d) p = p x || algunoArbol i p || algunoArbol d p

```

5.27.2. En Python

```

# -----
# Los árboles binarios con valores en las hojas y en los nodos se
# definen por
#
# @dataclass
# class Arbol(Generic[A]):
#     pass
#
# @dataclass
# class H(Arbol[A]):
#     x: A
#
# @dataclass
# class N(Arbol[A]):
#     x: A
#     i: Arbol[A]

```

```

#           d: Arbol[A]
# Por ejemplo, el árbol
#           5
#          / \
#         /   \
#        3     2
#       / \
#      1   4
# se representa por
#   N(5, N(3, H(1), H(4)), H(2))
#
# Definir la función
#   algunoArbol : (Arbol[A], Callable[[A], bool]) -> bool
# tal que algunoArbol(a, p) se verifica si algún elemento del árbol a
# cumple la propiedad p. Por ejemplo,
#   >>> algunoArbol(N(5, N(3, H(1), H(4)), H(2)), lambda x: x > 4)
#   True
#   >>> algunoArbol(N(5, N(3, H(1), H(4)), H(2)), lambda x: x > 7)
#   False
# -----

```

```

from dataclasses import dataclass
from typing import Callable, Generic, TypeVar

```

```

A = TypeVar("A")

```

```

@dataclass
class Arbol(Generic[A]):
    pass

```

```

@dataclass
class H(Arbol[A]):
    x: A

```

```

@dataclass
class N(Arbol[A]):
    x: A
    i: Arbol[A]
    d: Arbol[A]

```

```
def algunoArbol(a: Arbol[A], p: Callable[[A], bool]) -> bool:
    match a:
        case H(x):
            return p(x)
        case N(x, i, d):
            return p(x) or algunoArbol(i, p) or algunoArbol(d, p)
    assert False
```

5.28. Elementos del nivel k de un árbol

5.28.1. En Haskell

```
-- -----
-- Los árboles binarios con valores en las hojas y en los nodos se
-- definen por
--     data Arbol a = H a
--                   | N a (Arbol a) (Arbol a)
-- Por ejemplo, el árbol
--       7
--      / \
--     /   \
--    2     9
--   / \
--  5   4
-- se representa por
--     N 7 (N 2 (H 5) (H 4)) (H 9)
--
-- Un elemento de un árbol se dirá de nivel k si aparece en el árbol a
-- distancia k de la raíz.
--
-- Definir la función
--     nivel :: Int -> Arbol a -> [a]
-- tal que (nivel k a) es la lista de los elementos de nivel k del árbol
-- a. Por ejemplo,
--     nivel 0 (H 5) == [5]
--     nivel 1 (H 5) == []
--     nivel 0 (N 7 (N 2 (H 5) (H 4)) (H 9)) == [7]
--     nivel 1 (N 7 (N 2 (H 5) (H 4)) (H 9)) == [2,9]
--     nivel 2 (N 7 (N 2 (H 5) (H 4)) (H 9)) == [5,4]
--     nivel 3 (N 7 (N 2 (H 5) (H 4)) (H 9)) == []
```

```
module Elementos_del_nivel_k_de_un_arbol where
```

```
data Arbol a = H a
              | N a (Arbol a) (Arbol a)
```

```
nivel :: Int -> Arbol a -> [a]
nivel 0 (H x)      = [x]
nivel 0 (N x _ _) = [x]
nivel _ (H _ )     = []
nivel k (N _ i d) = nivel (k-1) i ++ nivel (k-1) d
```

5.28.2. En Python

```
# -----
# Los árboles binarios con valores en las hojas y en los nodos se
# definen por
# @dataclass
# class Arbol(Generic[A]):
#     pass
#
# @dataclass
# class H(Arbol[A]):
#     x: A
#
# @dataclass
# class N(Arbol[A]):
#     x: A
#     i: Arbol[A]
#     d: Arbol[A]
# Por ejemplo, el árbol
#       7
#      / \
#     /   \
#    2     9
#   / \
#  5   4
# se representa por
# N(7, N(2, H(5), H(4)), H(9))
```

```

#
# Un elemento de un árbol se dirá de nivel k si aparece en el árbol a
# distancia k de la raíz.
#
# Definir la función
# nivel : (int, Arbol[A]) -> list[A]
# tal que nivel(k, a) es la lista de los elementos de nivel k del árbol
# a. Por ejemplo,
# >>> nivel(0, N(7, N(2, H(5), H(4)), H(9)))
# [7]
# >>> nivel(1, N(7, N(2, H(5), H(4)), H(9)))
# [2, 9]
# >>> nivel(2, N(7, N(2, H(5), H(4)), H(9)))
# [5, 4]
# >>> nivel(3, N(7, N(2, H(5), H(4)), H(9)))
# []
# -----

```

```

from dataclasses import dataclass
from typing import Generic, TypeVar

```

```

A = TypeVar("A")

```

```

@dataclass
class Arbol(Generic[A]):
    pass

```

```

@dataclass
class H(Arbol[A]):
    x: A

```

```

@dataclass
class N(Arbol[A]):
    x: A
    i: Arbol[A]
    d: Arbol[A]

```

```

def nivel(k: int, a: Arbol[A]) -> list[A]:
    match (k, a):
        case (0, H(x)):

```

```

        return [x]
    case (0, N(x, _, _)):
        return [x]
    case (_, H(_)):
        return []
    case (_, N(_, i, d)):
        return nivel(k - 1, i) + nivel(k - 1, d)
assert False

```

5.29. Árbol de factorización

5.29.1. En Haskell

```

-- -----
-- Los divisores medios de un número son los que ocupan la posición
-- media entre los divisores de n, ordenados de menor a mayor. Por
-- ejemplo, los divisores de 60 son [1,2,3,4,5,6,10,12,15,20,30,60] y
-- sus divisores medios son 6 y 10. Para los números que son cuadrados
-- perfectos, sus divisores medios de son sus raíces cuadradas; por
-- ejemplos, los divisores medios de 9 son 3 y 3.
--
-- El árbol de factorización de un número compuesto n se construye de la
-- siguiente manera:
--   * la raíz es el número n,
--   * la rama izquierda es el árbol de factorización de su divisor
--     medio menor y
--   * la rama derecha es el árbol de factorización de su divisor
--     medio mayor
-- Si el número es primo, su árbol de factorización sólo tiene una hoja
-- con dicho número. Por ejemplo, el árbol de factorización de 60 es
--
--      60
--     /  \
--    6    10
--   / \  / \
--  2  3 2  5
--
-- Definir la función
--   arbolFactorizacion :: Int -> Arbol
-- tal que (arbolFactorizacion n) es el árbol de factorización de n. Por
-- ejemplo,

```

```
-- arbolFactorizacion 60 == N 60 (N 6 (H 2) (H 3)) (N 10 (H 2) (H 5))
-- arbolFactorizacion 45 == N 45 (H 5) (N 9 (H 3) (H 3))
-- arbolFactorizacion 7  == H 7
-- arbolFactorizacion 9  == N 9 (H 3) (H 3)
-- arbolFactorizacion 14 == N 14 (H 2) (H 7)
-- arbolFactorizacion 28 == N 28 (N 4 (H 2) (H 2)) (H 7)
-- arbolFactorizacion 84 == N 84 (H 7) (N 12 (H 3) (N 4 (H 2) (H 2)))
-- -----
```

```
{-# OPTIONS_GHC -fno-warn-type-defaults #-}
```

```
module Arbol_de_factorizacion where
```

```
import Test.QuickCheck
```

```
-- 1ª solución
```

```
-- =====
```

```
data Arbol = H Int
           | N Int Arbol Arbol
  deriving (Eq, Show)
```

```
arbolFactorizacion1 :: Int -> Arbol
arbolFactorizacion1 n
  | esPrimo n = H n
  | otherwise = N n (arbolFactorizacion1 x) (arbolFactorizacion1 y)
  where (x,y) = divisoresMedio n
```

```
-- (esPrimo n) se verifica si n es primo. Por ejemplo,
--   esPrimo 7 == True
--   esPrimo 9 == False
```

```
esPrimo :: Int -> Bool
esPrimo n = divisores n == [1,n]
```

```
-- (divisoresMedio n) es el par formado por los divisores medios de
-- n. Por ejemplo,
--   divisoresMedio 30 == (5,6)
--   divisoresMedio 7  == (1,7)
--   divisoresMedio 16 == (4,4)
divisoresMedio :: Int -> (Int,Int)
```

```

divisoresMedio n = (n `div` x,x)
  where xs = divisores n
        x  = xs !! (length xs `div` 2)

-- (divisores n) es la lista de los divisores de n. Por ejemplo,
--   divisores 30 == [1,2,3,5,6,10,15,30]
divisores :: Int -> [Int]
divisores n = [x | x <- [1..n], n `rem` x == 0]

-- 2ª solución
-- =====

arbolFactorizacion2 :: Int -> Arbol
arbolFactorizacion2 n
  | x == 1      = H n
  | otherwise = N n (arbolFactorizacion2 x) (arbolFactorizacion2 y)
  where (x,y) = divisoresMedio n

-- (divisoresMedio2 n) es el par formado por los divisores medios de
-- n. Por ejemplo,
--   divisoresMedio2 30 == (5,6)
--   divisoresMedio2 7  == (1,7)
divisoresMedio2 :: Int -> (Int,Int)
divisoresMedio2 n = (n `div` x,x)
  where m = ceiling (sqrt (fromIntegral n))
        x = head [y | y <- [m..n], n `rem` y == 0]

-- Comprobación de equivalencia
-- =====

-- La propiedad es
prop_arbolFactorizacion :: Int -> Property
prop_arbolFactorizacion n =
  n > 1 ==> arbolFactorizacion1 n == arbolFactorizacion2 n

-- La comprobación es
--   λ> quickCheck prop_arbolFactorizacion
--   +++ OK, passed 100 tests; 162 discarded.

```

5.29.2. En Python

```
# -----
# Los divisores medios de un número son los que ocupan la posición
# media entre los divisores de n, ordenados de menor a mayor. Por
# ejemplo, los divisores de 60 son [1,2,3,4,5,6,10,12,15,20,30,60] y
# sus divisores medios son 6 y 10. Para los números que son cuadrados
# perfectos, sus divisores medios de son sus raíces cuadradas; por
# ejemplos, los divisores medios de 9 son 3 y 3.
#
# El árbol de factorización de un número compuesto n se construye de la
# siguiente manera:
#   * la raíz es el número n,
#   * la rama izquierda es el árbol de factorización de su divisor
#     medio menor y
#   * la rama derecha es el árbol de factorización de su divisor
#     medio mayor
# Si el número es primo, su árbol de factorización sólo tiene una hoja
# con dicho número. Por ejemplo, el árbol de factorización de 60 es
#
#      60
#     /  \
#    6    10
#   / \  / \
#  2  3 2  5
#
# Definir la función
#   arbolFactorizacion :: Int -> Arbol
# tal que (arbolFactorizacion n) es el árbol de factorización de n. Por
# ejemplo,
#   arbolFactorizacion(60) == N(60, N(6, H(2), H(3)), N(10, H(2), H(5)))
#   arbolFactorizacion(45) == N(45, H(5), N(9, H(3), H(3)))
#   arbolFactorizacion(7)  == H(7)
#   arbolFactorizacion(9)  == N(9, H(3), H(3))
#   arbolFactorizacion(14) == N(14, H(2), H(7))
#   arbolFactorizacion(28) == N(28, N(4, H(2), H(2)), H(7))
#   arbolFactorizacion(84) == N(84, H(7), N(12, H(3), N(4, H(2), H(2))))
# -----

from dataclasses import dataclass
from math import ceil, sqrt
```

```

from hypothesis import given
from hypothesis import strategies as st

# 1ª solución
# =====

@dataclass
class Arbol:
    pass

@dataclass
class H(Arbol):
    x: int

@dataclass
class N(Arbol):
    x: int
    i: Arbol
    d: Arbol

# divisores(n) es la lista de los divisores de n. Por ejemplo,
#   divisores(30) == [1,2,3,5,6,10,15,30]
def divisores(n: int) -> list[int]:
    return [x for x in range(1, n + 1) if n % x == 0]

# divisoresMedio(n) es el par formado por los divisores medios de
# n. Por ejemplo,
#   divisoresMedio(30) == (5,6)
#   divisoresMedio(7)  == (1,7)
#   divisoresMedio(16) == (4,4)
def divisoresMedio(n: int) -> tuple[int, int]:
    xs = divisores(n)
    x = xs[len(xs) // 2]
    return (n // x, x)

# esPrimo(n) se verifica si n es primo. Por ejemplo,
#   esPrimo(7) == True
#   esPrimo(9) == False
def esPrimo(n: int) -> bool:
    return divisores(n) == [1, n]

```

```

def arbolFactorizacion1(n: int) -> Arbol:
    if esPrimo(n):
        return H(n)
    (x, y) = divisoresMedio(n)
    return N(n, arbolFactorizacion1(x), arbolFactorizacion1(y))

# 2ª solución
# =====

# divisoresMedio2(n) es el par formado por los divisores medios de
# n. Por ejemplo,
#     divisoresMedio2(30) == (5,6)
#     divisoresMedio2(7)  == (1,7)
#     divisoresMedio2(16) == (4,4)
def divisoresMedio2(n: int) -> tuple[int, int]:
    m = ceil(sqrt(n))
    x = [y for y in range(m, n + 1) if n % y == 0][0]
    return (n // x, x)

def arbolFactorizacion2(n: int) -> Arbol:
    if esPrimo(n):
        return H(n)
    (x, y) = divisoresMedio2(n)
    return N(n, arbolFactorizacion2(x), arbolFactorizacion2(y))

# Comprobación de equivalencia
# =====

# La propiedad es
@given(st.integers(min_value=2, max_value=200))
def test_arbolFactorizacion(n: int) -> None:
    assert arbolFactorizacion1(n) == arbolFactorizacion2(n)

# La comprobación es
#     src> poetry run pytest -q arbol_de_factorizacion.py
#     1 passed in 0.14s

```


5.30. Valor de un árbol booleano

5.30.1. En Haskell

```

-- -----
-- Se consideran los árboles con operaciones booleanas definidos por
--   data Arbol = H Bool
--               | Conj Arbol Arbol
--               | Disy Arbol Arbol
--               | Neg Arbol
--
-- Por ejemplo, los árboles
--
--           Conj                               Conj
--          /  \                               /  \
--         /    \                             /    \
--        /      \                           /      \
--       Disy      Conj                     Disy      Conj
--      /  \      /  \                     /  \      /  \
--     Conj  Neg Conj  True                Conj  Neg Neg  True
--    /  \  |  /  \                       /  \  |  /  \
--   True False False False              True False True False
--
-- se definen por
--   ej1, ej2 :: Arbol
--   ej1 = Conj (Disy (Conj (H True) (H False))
--                  (Neg (H False)))
--          (Conj (Neg (H False))
--                (H True))
--
--   ej2 = Conj (Disy (Conj (H True) (H False))
--                  (Neg (H True)))
--          (Conj (Neg (H False))
--                (H True))
--
-- Definir la función
--   valor :: Arbol -> Bool
-- tal que (valor ar) es el resultado de procesar el árbol realizando
-- las operaciones booleanas especificadas en los nodos. Por ejemplo,
--   valor ej1 == True
--   valor ej2 == False
-- -----

```

```

module Valor_de_un_arbol_booleano where

data Arbol = H Bool
           | Conj Arbol Arbol
           | Disy Arbol Arbol
           | Neg Arbol

ej1, ej2 :: Arbol
ej1 = Conj (Disy (Conj (H True) (H False))
            (Neg (H False)))
      (Conj (Neg (H False))
            (H True))

ej2 = Conj (Disy (Conj (H True) (H False))
            (Neg (H True)))
      (Conj (Neg (H False))
            (H True))

valor :: Arbol -> Bool
valor (H x)      = x
valor (Neg a)    = not (valor a)
valor (Conj i d) = valor i && valor d
valor (Disy i d) = valor i || valor d

```

5.30.2. En Python

```

# -----
# Se consideran los árboles con operaciones booleanas definidos por
# @dataclass
# class Arbol:
#     pass
#
# @dataclass
# class H(Arbol):
#     x: bool
#
# @dataclass
# class Conj(Arbol):
#     i: Arbol
#     d: Arbol

```

```

#
# @dataclass
# class Disy(Arbol):
#     i: Arbol
#     d: Arbol
#
# @dataclass
# class Neg(Arbol):
#     a: Arbol
#
# Por ejemplo, los árboles
#
#           Conj
#         /    \
#       /        \
#     Disy        Conj
#   /  \        /  \
# Conj Neg Neg True
# /  \  |  |
# True False False False
#
#           Conj
#         /    \
#       /        \
#     Disy        Conj
#   /  \        /  \
# Conj Neg Neg True
# /  \  |  |
# True False True False
#
# se definen por
# ej1: Arbol = Conj(Disy(Conj(H(True), H(False)),
#                        (Neg(H(False)))),
#                  (Conj(Neg(H(False)),
#                        (H(True)))))
#
# ej2: Arbol = Conj(Disy(Conj(H(True), H(False)),
#                        (Neg(H(True)))),
#                  (Conj(Neg(H(False)),
#                        (H(True)))))
#
# Definir la función
# valor : (Arbol) -> bool
# tal que valor(a) es el resultado de procesar el árbol a realizando
# las operaciones booleanas especificadas en los nodos. Por ejemplo,
# valor(ej1) == True
# valor(ej2) == False
# -----

```

```

from dataclasses import dataclass

```

```

@dataclass
class Arbol:
    pass

@dataclass
class H(Arbol):
    x: bool

@dataclass
class Conj(Arbol):
    i: Arbol
    d: Arbol

@dataclass
class Disy(Arbol):
    i: Arbol
    d: Arbol

@dataclass
class Neg(Arbol):
    a: Arbol

ej1: Arbol = Conj(Disy(Conj(H(True), H(False)),
                       (Neg(H(False)))),
                  (Conj(Neg(H(False)),
                        (H(True)))))

ej2: Arbol = Conj(Disy(Conj(H(True), H(False)),
                       (Neg(H(True)))),
                  (Conj(Neg(H(False)),
                        (H(True)))))

def valor(a: Arbol) -> bool:
    match a:
        case H(x):
            return x
        case Neg(b):
            return not valor(b)

```

```

    case Conj(i, d):
        return valor(i) and valor(d)
    case Disy(i, d):
        return valor(i) or valor(d)
assert False

```

5.31. Valor de una expresión aritmética básica

5.31.1. En Haskell

```

-----
-- Las expresiones aritméticas básicas pueden representarse usando el
-- siguiente tipo de datos
--   data Expr = C Int
--             | S Expr Expr
--             | P Expr Expr
-- Por ejemplo, la expresión 2*(3+7) se representa por
--   P (C 2) (S (C 3) (C 7))
--
-- Definir la función
--   valor :: Expr -> Int
-- tal que (valor e) es el valor de la expresión aritmética e. Por
-- ejemplo,
--   valor (P (C 2) (S (C 3) (C 7))) == 20
-----

```

```

module Valor_de_una_expresion_aritmetica_basica where

```

```

data Expr = C Int
          | S Expr Expr
          | P Expr Expr

```

```

valor :: Expr -> Int
valor (C x)    = x
valor (S x y)  = valor x + valor y
valor (P x y)  = valor x * valor y

```

5.31.2. En Python

```
# -----
# Las expresiones aritméticas básicas pueden representarse usando el
# siguiente tipo de datos
#     @dataclass
#     class Expr:
#         pass
#
#     @dataclass
#     class C(Expr):
#         x: int
#
#     @dataclass
#     class S(Expr):
#         x: Expr
#         y: Expr
#
#     @dataclass
#     class P(Expr):
#         x: Expr
#         y: Expr
# Por ejemplo, la expresión 2*(3+7) se representa por
#     P(C(2), S(C(3), C(7)))
#
# Definir la función
#     valor : (Expr) -> int:
# tal que valor(e) es el valor de la expresión aritmética e. Por
# ejemplo,
#     valor(P(C(2), S(C(3), C(7)))) == 20
# -----
```

```
from dataclasses import dataclass
```

```
@dataclass
class Expr:
    pass
```

```
@dataclass
class C(Expr):
```

```

    x: int

@dataclass
class S(Expr):
    x: Expr
    y: Expr

@dataclass
class P(Expr):
    x: Expr
    y: Expr

def valor(e: Expr) -> int:
    match e:
        case C(x):
            return x
        case S(x, y):
            return valor(x) + valor(y)
        case P(x, y):
            return valor(x) * valor(y)
    assert False

```

5.32. Aplicación de una función a una expresión aritmética

5.32.1. En Haskell

```

-- -----
-- Las expresiones aritméticas básicas pueden representarse usando el
-- siguiente tipo de datos
--     data Expr = C Int
--               | S Expr Expr
--               | P Expr Expr
--     deriving (Show, Eq)
-- Por ejemplo, la expresión 2*(3+7) se representa por
--     P (C 2) (S (C 3) (C 7))
--
-- Definir la función
--     aplica :: (Int -> Int) -> Expr -> Expr

```

```
-- tal que (aplica f e) es la expresión obtenida aplicando la función f
-- a cada uno de los números de la expresión e. Por ejemplo,
--   λ> aplica (+2) (S (P (C 3) (C 5)) (P (C 6) (C 7)))
--   S (P (C 5) (C 7)) (P (C 8) (C 9))
--   λ> aplica (*2) (S (P (C 3) (C 5)) (P (C 6) (C 7)))
--   S (P (C 6) (C 10)) (P (C 12) (C 14))
-- -----
```

```
module Aplicacion_de_una_funcion_a_una_expresion_aritmetica where
```

```
data Expr = C Int
          | S Expr Expr
          | P Expr Expr
  deriving (Show, Eq)
```

```
aplica :: (Int -> Int) -> Expr -> Expr
aplica f (C x)      = C (f x)
aplica f (S e1 e2) = S (aplica f e1) (aplica f e2)
aplica f (P e1 e2) = P (aplica f e1) (aplica f e2)
```

5.32.2. En Python

```
# -----
# Las expresiones aritméticas básicas pueden representarse usando el
# siguiente tipo de datos
#   @dataclass
#   class Expr:
#       pass
#
#   @dataclass
#   class C(Expr):
#       x: int
#
#   @dataclass
#   class S(Expr):
#       x: Expr
#       y: Expr
#
#   @dataclass
#   class P(Expr):
```



```

#         x: Expr
#         y: Expr
# Por ejemplo, la expresión 2*(3+7) se representa por
#     P(C(2), S(C(3), C(7)))
#
# Definir la función
#     aplica : (Callable[[int], int], Expr) -> Expr
# tal que aplica(f, e) es la expresión obtenida aplicando la función f
# a cada uno de los números de la expresión e. Por ejemplo,
#     >>> aplica(lambda x: 2 + x, S(P(C(3), C(5)), P(C(6), C(7))))
#         S(P(C(5), C(7)), P(C(8), C(9)))
#     >>> aplica(lambda x: 2 * x, S(P(C(3), C(5)), P(C(6), C(7))))
#         S(P(C(6), C(10)), P(C(12), C(14)))
# -----

```

```

from dataclasses import dataclass
from typing import Callable

```

```

@dataclass
class Expr:
    pass

```

```

@dataclass
class C(Expr):
    x: int

```

```

@dataclass
class S(Expr):
    x: Expr
    y: Expr

```

```

@dataclass
class P(Expr):
    x: Expr
    y: Expr

```

```

def aplica(f: Callable[[int], int], e: Expr) -> Expr:
    match e:
        case C(x):

```

```

        return C(f(x))
    case S(x, y):
        return S(aplica(f, x), aplica(f, y))
    case P(x, y):
        return P(aplica(f, x), aplica(f, y))
assert False

```

5.33. Valor de una expresión aritmética con una variable

5.33.1. En Haskell

```

-- -----
-- 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 Expr definido por
--   data Expr = X
--             | C Int
--             | S Expr Expr
--             | P Expr Expr
-- Por ejemplo, la expresión  $X*(13+X)$  se representa por
--   P X (S (C 13) X)
--
-- Definir la función
--   valor :: Expr -> Int -> Int
-- tal que (valor e n) es el valor de la expresión e cuando se
-- sustituye su variable por n. Por ejemplo,
--   valor (P X (S (C 13) X)) 2 == 30
-- -----

```

```

module Valor_de_una_expresion_aritmetica_con_una_variable where

```

```

data Expr = X
          | C Int
          | S Expr Expr
          | P Expr Expr

```

```

valor :: Expr -> Int -> Int
valor X      n = n

```

```

valor (C a)      _ = a
valor (S e1 e2) n = valor e1 n + valor e2 n
valor (P e1 e2) n = valor e1 n * valor e2 n

```

5.33.2. En Python

```

# -----
# 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 Expr definido por
# @dataclass
# class Expr:
#     pass
#
# @dataclass
# class X(Expr):
#     pass
#
# @dataclass
# class C(Expr):
#     x: int
#
# @dataclass
# class S(Expr):
#     x: Expr
#     y: Expr
#
# @dataclass
# class P(Expr):
#     x: Expr
#     y: Expr
# Por ejemplo, la expresión X*(13+X) se representa por
# P(X(), S(C(13), X()))
#
# Definir la función
# valor : (Expr, int) -> int
# tal que valor(e, n) es el valor de la expresión e cuando se
# sustituye su variable por n. Por ejemplo,
# valor(P(X(), S(C(13), X())), 2) == 30
# -----

```

```
from dataclasses import dataclass

@dataclass
class Expr:
    pass

@dataclass
class X(Expr):
    pass

@dataclass
class C(Expr):
    x: int

@dataclass
class S(Expr):
    x: Expr
    y: Expr

@dataclass
class P(Expr):
    x: Expr
    y: Expr

def valor(e: Expr, n: int) -> int:
    match e:
        case X():
            return n
        case C(a):
            return a
        case S(e1, e2):
            return valor(e1, n) + valor(e2, n)
        case P(e1, e2):
            return valor(e1, n) * valor(e2, n)
    assert False
```

5.34. Número de variables de una expresión aritmética

5.34.1. En Haskell

```

-----
-- 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 Expr definido por
--   data Expr = X
--             | C Int
--             | S Expr Expr
--             | P Expr Expr
-- Por ejemplo, la expresión  $X*(13+X)$  se representa por
--   P X (S (C 13) X)
--
-- Definir la función
--   numVars :: Expr -> Int
-- tal que (numVars e) es el número de variables en la expresión e. Por
-- ejemplo,
--   numVars (C 3)           == 0
--   numVars X               == 1
--   numVars (P X (S (C 13) X)) == 2
-----

```

```
module Numero_de_variables_de_una_expresion_aritmetica where
```

```

data Expr = X
          | C Int
          | S Expr Expr
          | P Expr Expr

numVars :: Expr -> Int
numVars X      = 1
numVars (C _)  = 0
numVars (S a b) = numVars a + numVars b
numVars (P a b) = numVars a + numVars b

```

5.34.2. En Python

```
# -----
# 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 Expr definido por
#   @dataclass
#   class Expr:
#       pass
#
#   @dataclass
#   class X(Expr):
#       pass
#
#   @dataclass
#   class C(Expr):
#       x: int
#
#   @dataclass
#   class S(Expr):
#       x: Expr
#       y: Expr
#
#   @dataclass
#   class P(Expr):
#       x: Expr
#       y: Expr
# Por ejemplo, la expresión  $X*(13+X)$  se representa por
#   P(X(), S(C(13), X()))
#
# Definir la función
#   numVars : (Expr) -> int
# tal que numVars(e) es el número de variables en la expresión e. Por
# ejemplo,
#   numVars(C(3))                == 0
#   numVars(X())                 == 1
#   numVars(P(X(), S(C(13), X()))) == 2
# -----
```

```
from dataclasses import dataclass
```

```
@dataclass
class Expr:
    pass

@dataclass
class X(Expr):
    pass

@dataclass
class C(Expr):
    x: int

@dataclass
class S(Expr):
    x: Expr
    y: Expr

@dataclass
class P(Expr):
    x: Expr
    y: Expr

def numVars(e: Expr) -> int:
    match e:
        case X():
            return 1
        case C(_):
            return 0
        case S(e1, e2):
            return numVars(e1) + numVars(e2)
        case P(e1, e2):
            return numVars(e1) + numVars(e2)
    assert False
```

5.35. Valor de una expresión aritmética con variables

5.35.1. En Haskell

```

-- -----
-- Las expresiones aritméticas con variables pueden representarse usando
-- el siguiente tipo de datos
--   data Expr = C Int
--             | V Char
--             | S Expr Expr
--             | P Expr Expr
-- Por ejemplo, la expresión 2*(a+5) se representa por
--   P (C 2) (S (V 'a') (C 5))
--
-- Definir la función
--   valor :: Expr -> [(Char,Int)] -> Int
-- tal que (valor x e) es el valor de la expresión x en el entorno e (es
-- decir, el valor de la expresión donde las variables de x se sustituyen
-- por los valores según se indican en el entorno e). Por ejemplo,
--   λ> valor (P (C 2) (S (V 'a') (V 'b')))) [('a',2),('b',5)]
--   14
-- -----

```

```

module Valor_de_una_expresion_aritmetica_con_variables where

```

```

data Expr = C Int
          | V Char
          | S Expr Expr
          | P Expr Expr

valor :: Expr -> [(Char,Int)] -> Int
valor (C x)      _ = x
valor (V x)      e = head [y | (z,y) <- e, z == x]
valor (S x y)    e = valor x e + valor y e
valor (P x y)    e = valor x e * valor y e

```


5.35.2. En Python

```
# -----
# Las expresiones aritméticas con variables pueden representarse usando
# el siguiente tipo de datos
# @dataclass
# class Expr:
#     pass
#
# @dataclass
# class C(Expr):
#     x: int
#
# @dataclass
# class V(Expr):
#     x: str
#
# @dataclass
# class S(Expr):
#     x: Expr
#     y: Expr
#
# @dataclass
# class P(Expr):
#     x: Expr
#     y: Expr
#
# Por ejemplo, la expresión  $2*(a+5)$  se representa por
# P(C(2), S(V('a'), C(5)))
#
# Definir la función
# valor : (Expr, list[tuple[str, int]]) -> int
# tal que valor(x, e) es el valor de la expresión x en el entorno e (es
# decir, el valor de la expresión donde las variables de x se sustituyen
# por los valores según se indican en el entorno e). Por ejemplo,
# λ> valor(P(C(2), S(V('a'), V('b'))), [('a', 2), ('b', 5)])
# 14
# -----
```

```
from dataclasses import dataclass
```

```
@dataclass
class Expr:
    pass

@dataclass
class C(Expr):
    x: int

@dataclass
class V(Expr):
    x: str

@dataclass
class S(Expr):
    x: Expr
    y: Expr

@dataclass
class P(Expr):
    x: Expr
    y: Expr

def valor(e: Expr, xs: list[tuple[str, int]]) -> int:
    match e:
        case C(a):
            return a
        case V(x):
            return [y for (z, y) in xs if z == x][0]
        case S(e1, e2):
            return valor(e1, xs) + valor(e2, xs)
        case P(e1, e2):
            return valor(e1, xs) * valor(e2, xs)
    assert False
```

5.36. Número de sumas en una expresión aritmética

5.36.1. En Haskell

```

-----
-- Las expresiones aritméticas con variables pueden representarse usando
-- el siguiente tipo de datos
--   data Expr = C Int
--             | V Char
--             | S Expr Expr
--             | P Expr Expr
-- Por ejemplo, la expresión 2*(a+5) se representa por
--   P (C 2) (S (V 'a') (C 5))
--
-- Definir la función
--   sumas :: Expr -> Int
-- tal que (sumas e) es el número de sumas en la expresión e. Por
-- ejemplo,
--   sumas (P (V 'z') (S (C 3) (V 'x')))) == 1
--   sumas (S (V 'z') (S (C 3) (V 'x')))) == 2
--   sumas (P (V 'z') (P (C 3) (V 'x')))) == 0
-----

```

```

module Numero_de_sumas_en_una_expresion_aritmetica where

```

```

data Expr = C Int
          | V Char
          | S Expr Expr
          | P Expr Expr

```

```

sumas :: Expr -> Int
sumas (V _) = 0
sumas (C _) = 0
sumas (S x y) = 1 + sumas x + sumas y
sumas (P x y) = sumas x + sumas y

```

5.36.2. En Python

```
# -----
# Las expresiones aritméticas con variables pueden representarse usando
# el siguiente tipo de datos
# @dataclass
# class Expr:
#     pass
#
# @dataclass
# class C(Expr):
#     x: int
#
# @dataclass
# class V(Expr):
#     x: str
#
# @dataclass
# class S(Expr):
#     x: Expr
#     y: Expr
#
# @dataclass
# class P(Expr):
#     x: Expr
#     y: Expr
#
# Por ejemplo, la expresión 2*(a+5) se representa por
# P(C(2), S(V('a'), C(5)))
#
# Definir la función
# sumas : (Expr) -> int
# tal que sumas(e) es el número de sumas en la expresión e. Por
# ejemplo,
# sumas(P(V('z'), S(C(3), V('x')))) == 1
# sumas(S(V('z'), S(C(3), V('x')))) == 2
# sumas(P(V('z'), P(C(3), V('x')))) == 0
# -----
```

```
from dataclasses import dataclass
```

```
@dataclass
class Expr:
    pass

@dataclass
class C(Expr):
    x: int

@dataclass
class V(Expr):
    x: str

@dataclass
class S(Expr):
    x: Expr
    y: Expr

@dataclass
class P(Expr):
    x: Expr
    y: Expr

def sumas(e: Expr) -> int:
    match e:
        case C(_):
            return 0
        case V(_):
            return 0
        case S(e1, e2):
            return 1 + sumas(e1) + sumas(e2)
        case P(e1, e2):
            return sumas(e1) + sumas(e2)
    assert False
```

5.37. Sustitución en una expresión aritmética

5.37.1. En Haskell

```

-----
-- Las expresiones aritméticas con variables pueden representarse usando
-- el siguiente tipo de datos
--   data Expr = C Int
--             | V Char
--             | S Expr Expr
--             | P Expr Expr
--   deriving (Eq, Show)
-- Por ejemplo, la expresión 2*(a+5) se representa por
--   P (C 2) (S (V 'a') (C 5))
--
-- Definir la función
--   sustitucion :: Expr -> [(Char, Int)] -> Expr
-- tal que (sustitucion e s) es la expresión obtenida sustituyendo las
-- variables de la expresión e según se indica en la sustitución s. Por
-- ejemplo,
--   λ> sustitucion (P (V 'z') (S (C 3) (V 'x')))) [('x',7),('z',9)]
--   P (C 9) (S (C 3) (C 7))
--   λ> sustitucion (P (V 'z') (S (C 3) (V 'y')))) [('x',7),('z',9)]
--   P (C 9) (S (C 3) (V 'y'))
-----

```

```

module Sustitucion_en_una_expresion_aritmetica where

```

```

data Expr = C Int
          | V Char
          | S Expr Expr
          | P Expr Expr
  deriving (Eq, Show)

sustitucion :: Expr -> [(Char, Int)] -> Expr
sustitucion e [] = e
sustitucion (V c) ((d,n):ps)
  | c == d = C n
  | otherwise = sustitucion (V c) ps
sustitucion (C n) _ = C n
sustitucion (S e1 e2) ps = S (sustitucion e1 ps) (sustitucion e2 ps)

```

`sustitucion (P e1 e2) ps = P (sustitucion e1 ps) (sustitucion e2 ps)`

5.37.2. En Python

```
# -----
# Las expresiones aritméticas con variables pueden representarse usando
# el siguiente tipo de datos
# @dataclass
# class Expr:
#     pass
#
# @dataclass
# class C(Expr):
#     x: int
#
# @dataclass
# class V(Expr):
#     x: str
#
# @dataclass
# class S(Expr):
#     x: Expr
#     y: Expr
#
# @dataclass
# class P(Expr):
#     x: Expr
#     y: Expr
#
# Por ejemplo, la expresión 2*(a+5) se representa por
# P(C(2), S(V('a'), C(5)))
#
# Definir la función
# sustitucion : (Expr, list[tuple[str, int]]) -> Expr
# tal que sustitucion(e s) es la expresión obtenida sustituyendo las
# variables de la expresión e según se indica en la sustitución s. Por
# ejemplo,
# >>> sustitucion(P(V('z'), S(C(3), V('x'))), [('x', 7), ('z', 9)])
# P(C(9), S(C(3), C(7)))
# >>> sustitucion(P(V('z'), S(C(3), V('y'))), [('x', 7), ('z', 9)])
```

```

#      P(C(9), S(C(3), V('y'))))
# -----

from dataclasses import dataclass

@dataclass
class Expr:
    pass

@dataclass
class C(Expr):
    x: int

@dataclass
class V(Expr):
    x: str

@dataclass
class S(Expr):
    x: Expr
    y: Expr

@dataclass
class P(Expr):
    x: Expr
    y: Expr

def sustitucion(e: Expr, ps: list[tuple[str, int]]) -> Expr:
    match (e, ps):
        case (e, []):
            return e
        case (V(c), ps):
            if c == ps[0][0]:
                return C(ps[0][1])
            return sustitucion(V(c), ps[1:])
        case (C(n), _):
            return C(n)
        case (S(e1, e2), ps):
            return S(sustitucion(e1, ps), sustitucion(e2, ps))

```



```

    case (P(e1, e2), ps):
        return P(sustitucion(e1, ps), sustitucion(e2, ps))
assert False

```

5.38. Expresiones aritméticas reducibles

5.38.1. En Haskell

```

-- -----
-- Las expresiones aritméticas con variables pueden representarse usando
-- el siguiente tipo de datos
--   data Expr = C Int
--             | V Char
--             | S Expr Expr
--             | P Expr Expr
-- Por ejemplo, la expresión 2*(a+5) se representa por
--   P (C 2) (S (V 'a') (C 5))
--
-- Definir la función
--   reducible :: Expr -> Bool
-- tal que (reducible a) se verifica si a es una expresión reducible; es
-- decir, contiene una operación en la que los dos operandos son números.
-- Por ejemplo,
--   reducible (S (C 3) (C 4))           == True
--   reducible (S (C 3) (V 'x'))         == False
--   reducible (S (C 3) (P (C 4) (C 5))) == True
--   reducible (S (V 'x') (P (C 4) (C 5))) == True
--   reducible (S (C 3) (P (V 'x') (C 5))) == False
--   reducible (C 3)                     == False
--   reducible (V 'x')                   == False
-- -----

```

```

module Expresiones_aritmeticas_reducibles where

```

```

data Expr = C Int
          | V Char
          | S Expr Expr
          | P Expr Expr

```

```

reducible :: Expr -> Bool

```

```

reducible (C _)           = False
reducible (V _)           = False
reducible (S (C _) (C _)) = True
reducible (S a b)         = reducible a || reducible b
reducible (P (C _) (C _)) = True
reducible (P a b)         = reducible a || reducible b

```

5.38.2. En Python

```

# -----
# Las expresiones aritméticas con variables pueden representarse usando
# el siguiente tipo de datos
# @dataclass
# class Expr:
#     pass
#
# @dataclass
# class C(Expr):
#     x: int
#
# @dataclass
# class V(Expr):
#     x: str
#
# @dataclass
# class S(Expr):
#     x: Expr
#     y: Expr
#
# @dataclass
# class P(Expr):
#     x: Expr
#     y: Expr
#
# Por ejemplo, la expresión 2*(a+5) se representa por
# P(C(2), S(V('a'), C(5)))
#
# Definir la función
# reducible : (Expr) -> bool
# tal que reducible(a) se verifica si a es una expresión reducible; es

```

```

# decir, contiene una operación en la que los dos operandos son números.
# Por ejemplo,
#     reducible(S(C(3), C(4)))           == True
#     reducible(S(C(3), V('x')))        == False
#     reducible(S(C(3), P(C(4), C(5))))  == True
#     reducible(S(V('x'), P(C(4), C(5)))) == True
#     reducible(S(C(3), P(V('x'), C(5)))) == False
#     reducible(C(3))                    == False
#     reducible(V('x'))                    == False
# -----

```

```

from dataclasses import dataclass

```

```

@dataclass
class Expr:
    pass

@dataclass
class C(Expr):
    x: int

@dataclass
class V(Expr):
    x: str

@dataclass
class S(Expr):
    x: Expr
    y: Expr

@dataclass
class P(Expr):
    x: Expr
    y: Expr

def reducible(e: Expr) -> bool:
    match e:
        case C(_):
            return False

```

```

    case V(_):
        return False
    case S(C(_), C(_)):
        return True
    case S(a, b):
        return reducible(a) or reducible(b)
    case P(C(_), C(_)):
        return True
    case P(a, b):
        return reducible(a) or reducible(b)
assert False

```

5.39. Máximos valores de una expresión aritmética

5.39.1. En Haskell

```

-- -----
-- Las expresiones aritméticas generales se pueden definir usando el
-- siguiente tipo de datos
--   data Expr = C Int
--             | X
--             | S Expr Expr
--             | R Expr Expr
--             | P Expr Expr
--             | E Expr Int
--   deriving (Eq, Show)
-- Por ejemplo, la expresión
--   3*x - (x+2)^7
-- se puede definir por
--   R (P (C 3) X) (E (S X (C 2)) 7)
--
-- Definir la función
--   maximo :: Expr -> [Int] -> (Int,[Int])
-- tal que (maximo e xs) es el par formado por el máximo valor de la
-- expresión e para los puntos de xs y en qué puntos alcanza el
-- máximo. Por ejemplo,
--   λ> maximo (E (S (C 10) (P (R (C 1) X) X)) 2) [-3..3]
--   (100,[0,1])

```

```
module Maximos_valores_de_una_expresion_aritmetica where
```

```
data Expr = C Int
          | X
          | S Expr Expr
          | R Expr Expr
          | P Expr Expr
          | E Expr Int
deriving (Eq, Show)
```

```
maximo :: Expr -> [Int] -> (Int,[Int])
maximo e ns = (m,[n | n <- ns, valor e n == m])
  where m = maximum [valor e n | n <- ns]
```

```
valor :: Expr -> Int -> Int
valor (C x) _ = x
valor X n = n
valor (S e1 e2) n = valor e1 n + valor e2 n
valor (R e1 e2) n = valor e1 n - valor e2 n
valor (P e1 e2) n = valor e1 n * valor e2 n
valor (E e1 m1) n = valor e1 n ^ m1
```

5.39.2. En Python

```
# -----
# Las expresiones aritméticas generales se pueden definir usando el
# siguiente tipo de datos
# @dataclass
# class Expr:
#     pass
#
# @dataclass
# class C(Expr):
#     x: int
#
# @dataclass
# class X(Expr):
#     pass
```

```

#
# @dataclass
# class S(Expr):
#     x: Expr
#     y: Expr
#
# @dataclass
# class R(Expr):
#     x: Expr
#     y: Expr
#
# @dataclass
# class P(Expr):
#     x: Expr
#     y: Expr
#
# @dataclass
# class E(Expr):
#     x: Expr
#     y: int
# Por ejemplo, la expresión
# 3*x - (x+2)^7
# se puede definir por
# R(P(C(3), X()), E(S(X(), C(2)), 7))
#
# Definir la función
# maximo : (Expr, list[int]) -> tuple[int, list[int]]
# tal que maximo(e, xs) es el par formado por el máximo valor de la
# expresión e para los puntos de xs y en qué puntos alcanza el
# máximo. Por ejemplo,
# >>> maximo(E(S(C(10), P(R(C(1), X()), X()))), 2), range(-3, 4))
# (100, [0, 1])
# -----

```

```

from dataclasses import dataclass

```

```

@dataclass
class Expr:
    pass

```

```
@dataclass
class C(Expr):
    x: int

@dataclass
class X(Expr):
    pass

@dataclass
class S(Expr):
    x: Expr
    y: Expr

@dataclass
class R(Expr):
    x: Expr
    y: Expr

@dataclass
class P(Expr):
    x: Expr
    y: Expr

@dataclass
class E(Expr):
    x: Expr
    y: int

def valor(e: Expr, n: int) -> int:
    match e:
        case C(a):
            return a
        case X():
            return n
        case S(e1, e2):
            return valor(e1, n) + valor(e2, n)
        case R(e1, e2):
            return valor(e1, n) - valor(e2, n)
        case P(e1, e2):
```

```

        return valor(e1, n) * valor(e2, n)
    case E(e1, m):
        return valor(e1, n) ** m
assert False

def maximo(e: Expr, ns: list[int]) -> tuple[int, list[int]]:
    m = max((valor(e, n) for n in ns))
    return (m, [n for n in ns if valor(e, n) == m])

```

5.40. Valor de expresiones aritméticas generales

5.40.1. En Haskell

```

-- -----
-- Las operaciones de suma, resta y multiplicación se pueden
-- representar mediante el siguiente tipo de datos
--   data Op = S | R | M
-- La expresiones aritméticas con dichas operaciones se pueden
-- representar mediante el siguiente tipo de dato algebraico
--   data Expr = C Int
--               | A Op Expr Expr
-- Por ejemplo, la expresión
--   (7-3)+(2*5)
-- se representa por
--   A S (A R (C 7) (C 3)) (A M (C 2) (C 5))
--
-- Definir la función
--   valor :: Expr -> Int
-- tal que (valor e) es el valor de la expresión e. Por ejemplo,
--   valor (A S (A R (C 7) (C 3)) (A M (C 2) (C 5))) == 14
--   valor (A M (A R (C 7) (C 3)) (A S (C 2) (C 5))) == 28
-- -----

```

```
module Valor_de_expresiones_aritmeticas_generales where
```

```
data Op = S | R | M
```

```
data Expr = C Int
```


| **A Op Expr Expr**

```
-- 1ª solución
-- =====

valor :: Expr -> Int
valor (C x)      = x
valor (A o e1 e2) = aplica o (valor e1) (valor e2)
  where aplica :: Op -> Int -> Int -> Int
        aplica S x y = x+y
        aplica R x y = x-y
        aplica M x y = x*y

-- 2ª solución
-- =====

valor2 :: Expr -> Int
valor2 (C n)      = n
valor2 (A o x y)  = sig o (valor2 x) (valor2 y)
  where sig :: Op -> Int -> Int -> Int
        sig S = (+)
        sig M = (*)
        sig R = (-)
```

5.40.2. En Python

```
# -----
# 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 Expr:
#     pass
#
# @dataclass
# class C(Expr):
#     x: int
#
```

```

# @dataclass
# class A(Expr):
#     o: Op
#     x: Expr
#     y: Expr
# Por ejemplo, la expresión
# (7-3)+(2*5)
# se representa por
# A(Op.S, A(Op.R, C(7), C(3)), A(Op.M, C(2), C(5)))
#
# Definir la función
# valor : (Expr) -> int
# tal que valor(e) es el valor de la expresión e. Por ejemplo,
# >>> valor(A(Op.S, A(Op.R, C(7), C(3)), A(Op.M, C(2), C(5))))
# 14
# >>> valor(A(Op.M, A(Op.R, C(7), C(3)), A(Op.S, C(2), C(5))))
# 28
# -----

```

```

from dataclasses import dataclass
from enum import Enum

```

```
Op = Enum('Op', ['S', 'R', 'M'])
```

```

@dataclass
class Expr:
    pass

```

```

@dataclass
class C(Expr):
    x: int

```

```

@dataclass
class A(Expr):
    o: Op
    x: Expr
    y: Expr

```

```

def aplica(o: Op, x: int, y: int) -> int:
    match o:

```

```

    case Op.S:
        return x + y
    case Op.R:
        return x - y
    case Op.M:
        return x * y
assert False

def valor(e: Expr) -> Int:
    match e:
        case C(x):
            return x
        case A(o, e1, e2):
            return aplica(o, valor(e1), valor(e2))
assert False

```

5.41. Valor de una expresión vectorial

5.41.1. En Haskell

```

-----
-- Se consideran las expresiones vectoriales formadas por un vector, la
-- suma de dos expresiones vectoriales o el producto de un entero por
-- una expresión vectorial. El siguiente tipo de dato define las
-- expresiones vectoriales
--   data ExpV = Vec Int Int
--             | Sum ExpV ExpV
--             | Mul Int ExpV
--   deriving Show
--
-- Definir la función
--   valorEV :: ExpV -> (Int,Int)
-- tal que (valorEV e) es el valorEV de la expresión vectorial c. Por
-- ejemplo,
--   valorEV (Vec 1 2) == (1,2)
--   valorEV (Sum (Vec 1 2) (Vec 3 4)) == (4,6)
--   valorEV (Mul 2 (Vec 3 4)) == (6,8)
--   valorEV (Mul 2 (Sum (Vec 1 2) (Vec 3 4))) == (8,12)
--   valorEV (Sum (Mul 2 (Vec 1 2)) (Mul 2 (Vec 3 4))) == (8,12)
-----

```

```

module Valor_de_una_expresion_vectorial where

data ExpV = Vec Int Int
           | Sum ExpV ExpV
           | Mul Int ExpV
deriving Show

-- 1ª solución
-- =====

valorEV1 :: ExpV -> (Int,Int)
valorEV1 (Vec x y)    = (x,y)
valorEV1 (Sum e1 e2) = (x1+x2,y1+y2)
    where (x1,y1) = valorEV1 e1
          (x2,y2) = valorEV1 e2
valorEV1 (Mul n e)    = (n*x,n*y)
    where (x,y) = valorEV1 e

-- 2ª solución
-- =====

valorEV2 :: ExpV -> (Int,Int)
valorEV2 (Vec a b)    = (a, b)
valorEV2 (Sum e1 e2) = suma (valorEV2 e1) (valorEV2 e2)
valorEV2 (Mul n e1)  = multiplica n (valorEV2 e1)

suma :: (Int,Int) -> (Int,Int) -> (Int,Int)
suma (a,b) (c,d) = (a+c,b+d)

multiplica :: Int -> (Int, Int) -> (Int, Int)
multiplica n (a,b) = (n*a,n*b)

```

5.41.2. En Python

```

# -----
# Se consideran las expresiones vectoriales formadas por un vector, la
# suma de dos expresiones vectoriales o el producto de un entero por
# una expresión vectorial. El siguiente tipo de dato define las
# expresiones vectoriales

```

```

# @dataclass
# class ExpV:
#     pass
#
# @dataclass
# class Vec(ExpV):
#     x: int
#     y: int
#
# @dataclass
# class Sum(ExpV):
#     x: ExpV
#     y: ExpV
#
# @dataclass
# class Mul(ExpV):
#     x: int
#     y: ExpV
#
# Definir la función
# valorEV : (ExpV) -> tuple[int, int]
# tal que valorEV(e) es el valorEV de la expresión vectorial 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)
# -----

```

```

from dataclasses import dataclass

```

```

@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 de las pilas

6.1.1. En Haskell

```
-- Una pila es una estructura de datos, caracterizada por ser una
-- secuencia de elementos en la que las operaciones de inserción y
-- extracción se realizan por el mismo extremo.
--
-- Las operaciones que definen a tipo abstracto de datos (TAD) de las
-- pilas (cuyos elementos son del tipo a) son las siguientes:
--   vacia      :: Pila a
--   apila      :: a -> Pila a -> Pila a
--   cima      :: Pila a -> a
--   desapila  :: Pila a -> Pila a
--   esVacia   :: Pila a -> Bool
-- tales que
--   + vacia es la pila vacía.
--   + (apila x p) es la pila obtenida añadiendo x al principio de p.
--   + (cima p) es la cima de la pila p.
--   + (desapila p) es la pila obtenida suprimiendo la cima de p.
--   + (esVacia p) se verifica si p es la pila vacía.
--
-- Las operaciones tienen que verificar las siguientes propiedades:
--   + cima(apila(x, p)) == x
--   + desapila(apila(x, p)) == p
--   + esVacia(vacia)
```

```
-- + not esVacia(apila(x, p))
--
-- Para usar el TAD hay que usar una implementación concreta. En
-- principio, consideraremos dos una usando listas y otra usando
-- sucesiones. Hay que elegir la que se desee utilizar, descomentándola
-- y comentando las otras.
```

```
{-# OPTIONS_GHC -fno-warn-unused-imports #-}
```

```
module TAD.Pila
```

```
  (Pila,
   vacia,      -- Pila a
   apila,      -- a -> Pila a -> Pila a
   cima,       -- Pila a -> a
   desapila,   -- Pila a -> Pila a
   esVacia,    -- Pila a -> Bool
  ) where
```

```
import TAD.PilaConListas
```

```
-- import TAD.PilaConSucesiones
```

6.1.2. En Python

```
# Una pila es una estructura de datos, caracterizada por ser una
# secuencia de elementos en la que las operaciones de inserción y
# extracción se realizan por el mismo extremo.
#
# Las operaciones que definen a tipo abstracto de datos (TAD) de las
# pilas (cuyos elementos son del tipo a) son las siguientes:
#   vacia      :: Pila a
#   apila      :: a -> Pila a -> Pila a
#   cima       :: Pila a -> a
#   desapila   :: Pila a -> Pila a
#   esVacia    :: Pila a -> Bool
# tales que
# + vacia es la pila vacía.
# + (apila x p) es la pila obtenida añadiendo x al principio de p.
# + (cima p) es la cima de la pila p.
# + (desapila p) es la pila obtenida suprimiendo la cima de p.
# + (esVacia p) se verifica si p es la pila vacía.
```

```

#
# Las operaciones tienen que verificar las siguientes propiedades:
# + cima(apila(x, p) == x
# + desapila(apila(x, p)) == p
# + esVacia(vacia)
# + not esVacia(apila(x, p))
#
# Para usar el TAD hay que usar una implementación concreta. En
# principio, consideraremos dos una usando listas y otra usando
# sucesiones. Hay que elegir la que se desee utilizar, descomentándola
# y comentando las otras.

__all__ = [
    'Pila',
    'vacia',
    'apila',
    'esVacia',
    'cima',
    'desapila',
    'pilaAleatoria'
]
from src.TAD.pilaConListas import (Pila, apila, cima, desapila, esVacia,
                                    pilaAleatoria, vacia)

# from src.TAD.pilaConDeque import (Pila, apila, cima, desapila, esVacia,
#                                    pilaAleatoria, vacia)

```

6.2. El tipo de datos de las pilas mediante listas

6.2.1. En Haskell

```
{-# OPTIONS_GHC -fno-warn-unused-top-binds #-}
```

```

module TAD.PilaConListas
  (Pila,
   vacia,      -- Pila a
   apila,      -- a -> Pila a -> Pila a
   cima,       -- Pila a -> a

```

```

    desapila,    -- Pila a -> Pila a
    esVacía,     -- Pila a -> Bool
    escribePila  -- Show a => Pila a -> String
  ) where

import Test.QuickCheck

-- Representación de las pilas mediante listas.
newtype Pila a = P [a]
  deriving Eq

-- (escribePila p) es la cadena correspondiente a la pila p. Por
-- ejemplo,
--   escribePila (apila 5 (apila 2 (apila 3 vacía))) == "5 | 2 | 3"
escribePila :: Show a => Pila a -> String
escribePila (P [])      = "-"
escribePila (P [x])     = show x
escribePila (P (x:xs)) = show x ++ " | " ++ escribePila (P xs)

-- Procedimiento de escritura de pilas.
instance Show a => Show (Pila a) where
  show = escribePila

-- Ejemplo de pila:
--   λ> apila 1 (apila 2 (apila 3 vacía))
--   1 | 2 | 3

-- vacía es la pila vacía. Por ejemplo,
--   λ> vacía
--   -
vacía :: Pila a
vacía = P []

-- (apila x p) es la pila obtenida añadiendo x encima de la pila p. Por
-- ejemplo,
--   λ> apila 4 (apila 3 (apila 2 (apila 5 vacía)))
--   4 | 3 | 2 | 5
apila :: a -> Pila a -> Pila a
apila x (P xs) = P (x:xs)

```

```

-- (cima p) es la cima de la pila p. Por ejemplo,
--   λ> cima (apila 4 (apila 3 (apila 2 (apila 5 vacia))))
--   4
cima :: Pila a -> a
cima (P []) = error "cima de la pila vacia"
cima (P (x:_)) = x

-- (desapila p) es la pila obtenida suprimiendo la cima de la pila
-- p. Por ejemplo,
--   λ> desapila (apila 4 (apila 3 (apila 2 (apila 5 vacia))))
--   3 | 2 | 5
desapila :: Pila a -> Pila a
desapila (P []) = error "desapila la pila vacia"
desapila (P (_,xs)) = P xs

-- (esVacia p) se verifica si p es la pila vacía. Por ejemplo,
--   esVacia (apila 1 (apila 2 (apila 3 vacia))) == False
--   esVacia vacia                               == True
esVacia :: Pila a -> Bool
esVacia (P xs) = null xs

-- Generador de pilas
-- =====

-- genPila es un generador de pilas. Por ejemplo,
--   λ> sample genPila
--   -
--   0|0|-
--   -
--   -6|4|-3|3|0|-
--   -
--   9|5|-1|-3|0|-8|-5|-7|2|-
--   -3|-10|-3|-12|11|6|1|-2|0|-12|-6|-
--   2|-14|-5|2|-
--   5|9|-
--   -1|-14|5|-
--   6|13|0|17|-12|-7|-8|-19|-14|-5|10|14|3|-18|2|-14|-11|-6|-
genPila :: (Arbitrary a, Num a) => Gen (Pila a)
genPila = do
  xs <- listOf arbitrary

```

```

return (foldr apila vacia xs)

-- El tipo pila es una instancia del arbitrario.
instance (Arbitrary a, Num a) => Arbitrary (Pila a) where
  arbitrary = genPila

-- Propiedades
-- =====

-- Las propiedades son
prop_pilas :: Int -> Pila Int -> Bool
prop_pilas x p =
  cima (apila x p) == x &&
  desapila (apila x p) == p &&
  esVacia vacia &&
  not (esVacia (apila x p))

-- La comprobación es:
--   λ> quickCheck prop_pilas
--   +++ OK, passed 100 tests.

```

6.2.2. En Python

```

# Se define la clase Pila con los siguientes métodos:
#   + apila(x) añade x al principio de la pila.
#   + cima() devuelve la cima de la pila.
#   + desapila() elimina la cima de la pila.
#   + esVacia() se verifica si la pila es vacía.
# Por ejemplo,
#   >>> p = Pila()
#   >>> p
#   -
#   >>> p.apila(5)
#   >>> p.apila(2)
#   >>> p.apila(3)
#   >>> p.apila(4)
#   >>> p
#   4 | 3 | 2 | 5
#   >>> p.cima()
#   4

```



```

#     >>> p.desapila()
#     >>> p
#     3 | 2 | 5
#     >>> p.esVacia()
#     False
#     >>> p = Pila()
#     >>> p.esVacia()
#     True
#
# Además se definen las correspondientes funciones. Por ejemplo,
#     >>> vacia()
#     -
#     >>> apila(4, apila(3, apila(2, apila(5, vacia()))))
#     4 | 3 | 2 | 5
#     >>> cima(apila(4, apila(3, apila(2, apila(5, vacia()))))
#     4
#     >>> desapila(apila(4, apila(3, apila(2, apila(5, vacia()))))
#     3 | 2 | 5
#     >>> esVacia(apila(4, apila(3, apila(2, apila(5, vacia()))))
#     False
#     >>> esVacia(vacia())
#     True
#
# Finalmente, se define un generador aleatorio de pilas y se comprueba
# que las pilas cumplen las propiedades de su especificación.

__all__ = [
    'Pila',
    'vacia',
    'apila',
    'esVacia',
    'cima',
    'desapila',
    'pilaAleatoria'
]

from copy import deepcopy
from dataclasses import dataclass, field
from typing import Generic, TypeVar

```

```

from hypothesis import given
from hypothesis import strategies as st

A = TypeVar('A')

# Clase de las pilas mediante Listas
# =====

@dataclass
class Pila(Generic[A]):
    _elementos: list[A] = field(default_factory=list)

    def __repr__(self) -> str:
        """
        Devuelve una cadena con los elementos de la pila separados por " | ".
        Si la pila está vacía, devuelve "-".
        """
        if len(self._elementos) == 0:
            return '-'
        return " | ".join(str(x) for x in self._elementos)

    def apila(self, x: A) -> None:
        """
        Agrega el elemento x al inicio de la pila.
        """
        self._elementos.insert(0, x)

    def esVacía(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. El tipo de datos de las pilas con librerías

6.3.1. En Haskell

```
{-# OPTIONS_GHC -fno-warn-unused-top-binds #-}

module TAD.PilaConSucesiones
  (Pila,
   vacia,      -- Pila a
   apila,      -- a -> Pila a -> Pila a
   cima,       -- Pila a -> a
   desapila,   -- Pila a -> Pila a
   esVacia,    -- Pila a -> Bool
   escribePila -- Show a => Pila a -> String
  ) where

import Data.Sequence as S
import Test.QuickCheck

-- Representación de las pilas mediante sucesiones.
newtype Pila a = P (Seq a)
  deriving Eq

-- (escribePila p) es la cadena correspondiente a la pila p. Por
-- ejemplo,
--     escribePila (apila 5 (apila 2 (apila 3 vacia))) == "5 | 2 | 3"
escribePila :: Show a => Pila a -> String
escribePila (P xs) = case viewl xs of
  EmptyL    -> "-"
  x :< xs' -> case viewl xs' of
    EmptyL -> show x
    _      -> show x ++ " | " ++ escribePila (P xs')

-- Procedimiento de escritura de pilas.
instance Show a => Show (Pila a) where
  show = escribePila

-- Ejemplo de pila:
--     λ> apila 1 (apila 2 (apila 3 vacia))
--     1 | 2 | 3
```

```

-- vacia es la pila vacía. Por ejemplo,
--   λ> vacia
--   -
vacia  :: Pila a
vacia = P empty

-- (apila x p) es la pila obtenida añadiendo x encima de la pila p. Por
-- ejemplo,
--   λ> apila 4 (apila 3 (apila 2 (apila 5 vacia)))
--   5 | 2 | 3 | 4
apila :: a -> Pila a -> Pila a
apila x (P xs) = P (x <| xs)

-- (cima p) es la cima de la pila p. Por ejemplo,
--   λ> cima (apila 4 (apila 3 (apila 2 (apila 5 vacia))))
--   4
cima :: Pila a -> a
cima (P xs) = case viewl xs of
  EmptyL -> error "cima de la pila vacia"
  x   :< _ -> x

-- (desapila p) es la pila obtenida suprimiendo la cima de la pila
-- p. Por ejemplo,
--   λ> desapila (apila 4 (apila 3 (apila 2 (apila 5 vacia))))
--   3 | 2 | 5
desapila :: Pila a -> Pila a
desapila (P xs) = case viewl xs of
  EmptyL   -> error "desapila la pila vacia"
  _   :< xs' -> P xs'

-- (esVacia p) se verifica si p es la pila vacía. Por ejemplo,
--   esVacia (apila 1 (apila 2 (apila 3 vacia))) == False
--   esVacia vacia                               == True
esVacia :: Pila a -> Bool
esVacia (P xs) = S.null xs

-- Generador de pilas
-- =====

-- genPila es un generador de pilas. Por ejemplo,

```

```

--      λ> sample genPila
--      -
--      -2
--      -
--      4 | -1 | 5 | 4 | -4 | 3
--      -8 | 2
--      4
--      5 | 7 | 10 | 6 | -4 | 11 | -1 | 0 | 7 | -3
--      -1 | -10
--      2 | -3 | -4 | 15 | -15 | 1 | -10 | -2 | -4 | 6 | -13 | 16 | -8 | 3 | 7
--      6
--      1 | -6 | -19 | 15 | -5 | -4 | -6 | -12 | -13 | 11 | 19 | -18 | -14 | -13 |
genPila :: (Arbitrary a, Num a) => Gen (Pila a)
genPila = do
  xs <- listOf arbitrary
  return (foldr apila vacia xs)

-- El tipo pila es una instancia del arbitrario.
instance (Arbitrary a, Num a) => Arbitrary (Pila a) where
  arbitrary = genPila

-- Propiedades
-- =====

-- Las propiedades son
prop_pilas :: Int -> Pila Int -> Bool
prop_pilas x p =
  cima (apila x p) == x &&
  desapila (apila x p) == p &&
  esVacia vacia &&
  not (esVacia (apila x p))

-- La comprobación e:
--      λ> quickCheck prop_pilas
--      +++ OK, passed 100 tests.

```

6.3.2. En Python

```

# Se define la clase Pila con los siguientes métodos:
#   + apila(x) añade x al principio de la pila.

```

```

# + cima() devuelve la cima de la pila.
# + desapila() elimina la cima de la pila.
# + esVacia() se verifica si la pila es vacía.
# Por ejemplo,
# >>> p = Pila()
# >>> p
# -
# >>> p.apila(5)
# >>> p.apila(2)
# >>> p.apila(3)
# >>> p.apila(4)
# >>> p
# 4 | 3 | 2 | 5
# >>> p.cima()
# 4
# >>> p.desapila()
# >>> p
# 3 | 2 | 5
# >>> p.esVacia()
# False
# >>> p = Pila()
# >>> p.esVacia()
# True
#
# Además se definen las correspondientes funciones. Por ejemplo,
# >>> vacia()
# -
# >>> apila(4, apila(3, apila(2, apila(5, vacia()))))
# 4 | 3 | 2 | 5
# >>> cima(apila(4, apila(3, apila(2, apila(5, vacia()))))
# 4
# >>> desapila(apila(4, apila(3, apila(2, apila(5, vacia()))))
# 3 | 2 | 5
# >>> esVacia(apila(4, apila(3, apila(2, apila(5, vacia()))))
# False
# >>> esVacia(vacia())
# True
#
# Finalmente, se define un generador aleatorio de pilas y se comprueba
# que las pilas cumplen las propiedades de su especificación.

```



```

__all__ = [
    'Pila',
    '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 __repr__(self) -> str:
        """
        Devuelve una cadena con los elementos de la pila separados por " | ".
        Si la pila está vacía, devuelve "-".
        """
        if len(self._elementos) == 0:
            return '-'
        return ' | '.join(str(x) for x in self._elementos)

    def apila(self, x: A) -> None:
        """
        Agrega el elemento x al inicio de la pila.

```

```

        """
        self._elementos.appendleft(x)

def esVacia(self) -> bool:
    """
    Verifica si la pila está vacía.

    Devuelve True si la pila está vacía, False en caso contrario.
    """
    return len(self._elementos) == 0

def cima(self) -> A:
    """
    Devuelve el elemento en la cima de la pila.
    """
    return self._elementos[0]

def desapila(self) -> None:
    """
    Elimina el elemento en la cima de la pila.
    """
    self._elementos.popleft()

# Funciones del tipo de las listas
# =====

def vacia() -> Pila[A]:
    """
    Crea y devuelve una pila vacía de tipo A.
    """
    p: Pila[A] = Pila()
    return p

def apila(x: A, p: Pila[A]) -> Pila[A]:
    """
    Añade un elemento x al tope de la pila p y devuelve una copia de la
    pila modificada.
    """
    _aux = deepcopy(p)
    _aux.apila(x)

```

```
    return _aux

def esVacia(p: Pila[A]) -> bool:
    """
    Devuelve True si la pila está vacía, False si no lo está.
    """
    return p.esVacia()

def cima(p: Pila[A]) -> A:
    """
    Devuelve el elemento en la cima de la pila p.
    """
    return p.cima()

def desapila(p: Pila[A]) -> Pila[A]:
    """
    Elimina el elemento en la cima de la 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. Transformación entre pilas y listas

6.4.1. En Haskell

```
-----
-- Utilizando el [tipo abstracto de datos de las pilas](https://bit.ly/3GTTToyK),
-- definir la función
--   listaApila :: [a] -> Pila a
--   pilaAlista :: Pila a -> [a]
-- tales que
-- + (listaApila xs) es la pila formada por los elementos de xs.
--   Por ejemplo,
--   λ> listaApila [3, 2, 5]
--   5 | 2 | 3
-- + (pilaAlista p) es la lista formada por los elementos de la
--   pila p. Por ejemplo,
--   λ> pilaAlista (apila 5 (apila 2 (apila 3 vacia)))
--   [3, 2, 5]
--
-- Comprobar con QuickCheck que ambas funciones son inversa; es decir,
--   pilaAlista (listaApila xs) = xs
--   listaApila (pilaAlista p) = p
-----

{-# OPTIONS_GHC -fno-warn-unused-imports #-}
```

```

module Transformaciones_pilas_listas where

import TAD.Pila (Pila, vacia, apila, esVacia, cima, desapila)
import Test.QuickCheck

-- 1ª definición de listaApila
-- =====

listaApila :: [a] -> Pila a
listaApila ys = aux (reverse ys)
  where aux [] = vacia
        aux (x:xs) = apila x (aux xs)

-- 2ª definición de listaApila
-- =====

listaApila2 :: [a] -> Pila a
listaApila2 = aux . reverse
  where aux [] = vacia
        aux (x:xs) = apila x (aux xs)

-- 3ª definición de listaApila
-- =====

listaApila3 :: [a] -> Pila a
listaApila3 = aux . reverse
  where aux = foldr apila vacia

-- 4ª definición de listaApila
-- =====

listaApila4 :: [a] -> Pila a
listaApila4 xs = foldr apila vacia (reverse xs)

-- 5ª definición de listaApila
-- =====

listaApila5 :: [a] -> Pila a
listaApila5 = foldr apila vacia . reverse

```

```

-- Comprobación de equivalencia de las definiciones de listaApila
-- =====

-- La propiedad es
prop_listaApila :: [Int] -> Bool
prop_listaApila xs =
  all (== listaApila xs)
    [listaApila2 xs,
     listaApila3 xs,
     listaApila4 xs,
     listaApila5 xs]

-- La comprobación es
--    λ> quickCheck prop_listaApila
--    +++ OK, passed 100 tests.

-- 1ª definición de pilaAlista
-- =====

pilaAlista :: Pila a -> [a]
pilaAlista p
  | esVacía p = []
  | otherwise = pilaAlista dp ++ [cp]
  where cp = cima p
        dp = desapila p

-- 2ª definición de pilaAlista
-- =====

pilaAlista2 :: Pila a -> [a]
pilaAlista2 = reverse . aux
  where aux p | esVacía p = []
              | otherwise = cp : aux dp
            where cp = cima p
                  dp = desapila p

-- Comprobación de equivalencia de las definiciones de pilaAlista
-- =====

```

```

-- La propiedad es
prop_pilaAlista :: Pila Int -> Bool
prop_pilaAlista p =
  pilaAlista p == pilaAlista2 p

-- La comprobación es
--   λ> quickCheck prop_pilaAlista
--   +++ OK, passed 100 tests.

-- Comprobación de las propiedades
-- =====

-- La primera propiedad es
prop_1_listaApila :: [Int] -> Bool
prop_1_listaApila xs =
  pilaAlista (listaApila xs) == xs

-- La comprobación es
--   λ> quickCheck prop_1_listaApila
--   +++ OK, passed 100 tests.

-- La segunda propiedad es
prop_2_listaApila :: Pila Int -> Bool
prop_2_listaApila p =
  listaApila (pilaAlista p) == p

-- La comprobación es
--   λ> quickCheck prop_2_listaApila
--   +++ OK, passed 100 tests.

```

6.4.2. En Python

```

# -----
# Utilizando el [tipo abstracto de datos de las pilas](https://bit.ly/3GTTToyK)
# definir las funciones
#   listaApila : (list[A]) -> Pila[A]
#   pilaAlista : (Pila[A]) -> list[A]
# tales que
# + listaApila(xs) es la pila formada por los elementos de xs.
# Por ejemplo,

```

```

#     >>> listaApila([3, 2, 5])
#     5 | 2 | 3
# + pilaAlista(p) es la lista formada por los elementos de la
#   pila p. Por ejemplo,
#     >>> ej = apila(5, apila(2, apila(3, vacia())))
#     >>> pilaAlista(ej)
#     [3, 2, 5]
#     >>> print(ej)
#     5 | 2 | 3
#
# Comprobar con Hypothesis que ambas funciones son inversas; es decir,
#   pilaAlista(listaApila(xs)) == xs
#   listaApila(pilaAlista(p)) == p
# -----

from copy import deepcopy
from typing import TypeVar

from hypothesis import given
from hypothesis import strategies as st

from src.TAD.pila import (Pila, apila, cima, desapila, esVacia, pilaAleatoria,
                           vacia)

A = TypeVar('A')

# 1ª definición de listaApila
# =====

def listaApila(ys: list[A]) -> Pila[A]:
    def aux(xs: list[A]) -> Pila[A]:
        if not xs:
            return vacia()
        return apila(xs[0], aux(xs[1:]))

    return aux(list(reversed(ys)))

# 2ª solución de listaApila
# =====

```



```

def listaApila2(xs: list[A]) -> Pila[A]:
    p: Pila[A] = Pila()
    for x in xs:
        p.apila(x)
    return p

# Comprobación de equivalencia de las definiciones de listaApila
# =====

# La propiedad es
@given(st.lists(st.integers()))
def test_listaApila(xs: list[int]) -> None:
    assert listaApila(xs) == listaApila2(xs)

# 1ª definición de pilaAlista
# =====

def pilaAlista(p: Pila[A]) -> list[A]:
    if esVacia(p):
        return []
    cp = cima(p)
    dp = desapila(p)
    return pilaAlista(dp) + [cp]

# 2ª definición de pilaAlista
# =====

def pilaAlista2Aux(p: Pila[A]) -> list[A]:
    if p.esVacia():
        return []
    cp = p.cima()
    p.desapila()
    return pilaAlista2Aux(p) + [cp]

def pilaAlista2(p: Pila[A]) -> list[A]:
    p1 = deepcopy(p)
    return pilaAlista2Aux(p1)

# 3ª definición de pilaAlista
# =====

```

```

def pilaAlista3Aux(p: Pila[A]) -> list[A]:
    r = []
    while not p.esVacia():
        r.append(p.cima())
        p.desapila()
    return r[::-1]

def pilaAlista3(p: Pila[A]) -> list[A]:
    p1 = deepcopy(p)
    return pilaAlista3Aux(p1)

# Comprobación de equivalencia de las definiciones de pilaAlista
# =====

@given(p=pilaAleatoria())
def test_pilaAlista(p: Pila[int]) -> None:
    assert pilaAlista(p) == pilaAlista2(p)
    assert pilaAlista(p) == pilaAlista3(p)

# Comprobación de las propiedades
# =====

# La primera propiedad es
@given(st.lists(st.integers()))
def test_1_listaApila(xs: list[int]) -> None:
    assert pilaAlista(listaApila(xs)) == xs

# La segunda propiedad es
@given(p=pilaAleatoria())
def test_2_listaApila(p: Pila[int]) -> None:
    assert listaApila(pilaAlista(p)) == p

# La comprobación es
# src> poetry run pytest -v transformaciones_pilas_listas.py
# test_listaApila PASSED
# test_pilaAlista PASSED
# test_1_listaApila PASSED
# test_2_listaApila PASSED

```

6.5. Filtrado de pilas según una propiedad

6.5.1. En Haskell

```

-----
-- Utilizando el [tipo abstracto de datos de las pilas](https://bit.ly/3GTTToyK),
-- definir la función
--   filtraPila :: (a -> Bool) -> Pila a -> Pila a
-- tal que (filtraPila p q) es la pila obtenida con los elementos de
-- pila q que verifican el predicado p, en el mismo orden. Por ejemplo,
--   λ> ejPila = apila 6 (apila 3 (apila 1 (apila 4 vacia)))
--   λ> ejPila
--   6 | 3 | 1 | 4
--   λ> filtraPila even ejPila
--   6 | 4
--   λ> filtraPila odd ejPila
--   3 | 1
-----

```

```
{-# OPTIONS_GHC -fno-warn-unused-imports #-}
```

```
module FiltraPila where
```

```
import TAD.Pila (Pila, vacia, apila, esVacia, cima, desapila)
import Transformaciones_pilas_listas (listaApila, pilaAlista)
import Test.QuickCheck.HigherOrder
```

```
-- 1ª solución
-- =====
```

```
filtraPila1 :: (a -> Bool) -> Pila a -> Pila a
filtraPila1 p q
  | esVacia q = vacia
  | p cq      = apila cq (filtraPila1 p dq)
  | otherwise = filtraPila1 p dq
  where cq = cima q
        dq = desapila q
```

```
-- 2ª solución
-- =====
```

```
-- Se usarán las funciones listaApila y pilaAlista del ejercicio
-- "Transformaciones entre pilas y listas" que se encuentra en
-- https://bit.ly/3ZHewQ8
```

```
filtraPila2 :: (a -> Bool) -> Pila a -> Pila a
filtraPila2 p q =
  listaApila (filter p (pilaAlista q))
```

```
-- Comprobación de equivalencia
-- =====
```

```
-- La propiedad es
prop_filtraPila :: (Int -> Bool) -> [Int] -> Bool
prop_filtraPila p xs =
  filtraPila1 p q == filtraPila2 p q
  where q = listaApila xs
```

```
-- La comprobación es
-- λ> quickCheck' prop_filtraPila
-- +++ OK, passed 100 tests.
```

6.5.2. En Python

```
# -----
# Utilizando el [tipo abstracto de datos de las pilas](https://bit.ly/3GTTToyK)
# definir la función
#   filtraPila : (Callable[[A], bool], Pila[A]) -> Pila[A]
# tal que filtraPila(p, q) es la pila obtenida con los elementos de
# pila q que verifican el predicado p, en el mismo orden. Por ejemplo,
#   >>> ej = apila(3, apila(4, apila(6, apila(5, vacia()))))
#   >>> filtraPila(lambda x: x % 2 == 0, ej)
#   4 | 6
#   >>> filtraPila(lambda x: x % 2 == 1, ej)
#   3 | 5
#   >>> ej
#   3 | 4 | 6 | 5
# -----
```

```
from copy import deepcopy
from typing import Callable, TypeVar
```

```
from hypothesis import given

from src.TAD.pila import (Pila, apila, cima, desapila, esVacia, pilaAleatoria,
                          vacia)
from src.transformaciones_pilas_listas import listaApila, pilaAlista

A = TypeVar('A')

# 1ª solución
# =====

def filtraPila1(p: Callable[[A], bool], q: Pila[A]) -> Pila[A]:
    if esVacia(q):
        return q
    cq = cima(q)
    dq = desapila(q)
    r = filtraPila1(p, dq)
    if p(cq):
        return apila(cq, r)
    return r

# 2ª solución
# =====

# Se usarán las funciones listaApila y pilaAlista del ejercicio
# "Transformaciones entre pilas y listas" que se encuentra en
# https://bit.ly/3ZHewQ8

def filtraPila2(p: Callable[[A], bool], q: Pila[A]) -> Pila[A]:
    return listaApila(list(filter(p, pilaAlista(q))))

# 3ª solución
# =====

def filtraPila3Aux(p: Callable[[A], bool], q: Pila[A]) -> Pila[A]:
    if q.esVacia():
        return q
    cq = q.cima()
    q.desapila()
```

```

    r = filtraPila3Aux(p, q)
    if p(cq):
        r.apila(cq)
    return r

def filtraPila3(p: Callable[[A], bool], q: Pila[A]) -> Pila[A]:
    q1 = deepcopy(q)
    return filtraPila3Aux(p, q1)

# 4ª solución
# =====

def filtraPila4Aux(p: Callable[[A], bool], q: Pila[A]) -> Pila[A]:
    r: Pila[A] = Pila()
    while not q.esVacia():
        cq = q.cima()
        q.desapila()
        if p(cq):
            r.apila(cq)
    r1: Pila[A] = Pila()
    while not r.esVacia():
        r1.apila(r.cima())
        r.desapila()
    return r1

def filtraPila4(p: Callable[[A], bool], q: Pila[A]) -> Pila[A]:
    q1 = deepcopy(q)
    return filtraPila4Aux(p, q1)

# Comprobación de equivalencia de las definiciones
# =====

# La propiedad es
@given(p=pilaAleatoria())
def test_filtraPila(p: Pila[int]) -> None:
    r = filtraPila1(lambda x: x % 2 == 0, p)
    assert filtraPila2(lambda x: x % 2 == 0, p) == r
    assert filtraPila3(lambda x: x % 2 == 0, p) == r
    assert filtraPila4(lambda x: x % 2 == 0, p) == r

```

```
# La comprobación es
#   src> poetry run pytest -q filtraPila.py
#   1 passed in 0.25s
```

6.6. Aplicación de una función a los elementos de una pila

6.6.1. En Haskell

```
-- -----
-- Utilizando el [tipo abstracto de datos de las pilas](https://bit.ly/3GTTToyK),
-- definir la función
--   mapPila :: (a -> a) -> Pila a -> Pila a
-- tal que (mapPila f p) es la pila formada con las imágenes por f de
-- los elementos de pila p, en el mismo orden. Por ejemplo,
--   λ> mapPila (+1) (apila 5 (apila 2 (apila 7 vacia)))
--   6 | 3 | 8
-- -----
```

```
{-# OPTIONS_GHC -fno-warn-unused-imports #-}
```

```
module MapPila where
```

```
import TAD.Pila (Pila, vacia, apila, esVacia, cima, desapila)
import Transformaciones_pilas_listas (listaApila, pilaAlista)
import Test.QuickCheck.HigherOrder
```

```
-- 1ª solución
-- =====
```

```
mapPila1 :: (a -> a) -> Pila a -> Pila a
mapPila1 f p
  | esVacia p = p
  | otherwise = apila (f cp) (mapPila1 f dp)
  where cp = cima p
        dp = desapila p
```

```
-- 2ª solución
-- =====
```

```
-- Se usarán las funciones listaApila y pilaAlista del ejercicio
-- "Transformaciones entre pilas y listas" que se encuentra en
-- https://bit.ly/3ZHewQ8
```

```
mapPila2 :: (a -> a) -> Pila a -> Pila a
mapPila2 f p =
    listaApila (map f (pilaAlista p))
```

```
-- Comprobación de equivalencia
-- =====
```

```
-- La propiedad es
prop_mapPila :: (Int -> Int) -> [Int] -> Bool
prop_mapPila f p =
    mapPila1 f q == mapPila2 f q
    where q = listaApila p
```

```
-- La comprobación es
--    λ> quickCheck' prop_mapPila
--    +++ OK, passed 100 tests.
```

6.6.2. En Python

```
# -----
# Utilizando el [tipo abstracto de datos de las pilas](https://bit.ly/3GTTToyK)
# definir la función
#     mapPila : (Callable[[A], A], Pila[A]) -> Pila[A]
# tal que mapPila(f, p) es la pila formada con las imágenes por f de
# los elementos de pila p, en el mismo orden. Por ejemplo,
#     >>> ej = apila(5, apila(2, apila(7, vacia())))
#     >>> mapPila1(lambda x: x + 1, ej)
#     6 | 3 | 8
#     >>> ej
#     5 | 2 | 7
# -----
```

```
from copy import deepcopy
from typing import Callable, TypeVar
```



```
from hypothesis import given

from src.TAD.pila import (Pila, apila, cima, desapila, esVacia, pilaAleatoria,
                          vacia)
from src.transformaciones_pilas_listas import listaApila, pilaAlista

A = TypeVar('A')

# 1ª solución
# =====

def mapPila1(f: Callable[[A], A], p: Pila[A]) -> Pila[A]:
    if esVacia(p):
        return p
    cp = cima(p)
    dp = desapila(p)
    return apila(f(cp), mapPila1(f, dp))

# 2ª solución
# =====

# Se usarán las funciones listaApila y pilaAlista del ejercicio
# "Transformaciones entre pilas y listas" que se encuentra en
# https://bit.ly/3ZHewQ8

def mapPila2(f: Callable[[A], A], p: Pila[A]) -> Pila[A]:
    return listaApila(list(map(f, pilaAlista(p))))

# 3ª solución
# =====

def mapPila3Aux(f: Callable[[A], A], p: Pila[A]) -> Pila[A]:
    if p.esVacia():
        return p
    cp = p.cima()
    p.desapila()
    r = mapPila3Aux(f, p)
    r.apila(f(cp))
    return r
```

```

def mapPila3(f: Callable[[A], A], p: Pila[A]) -> Pila[A]:
    p1 = deepcopy(p)
    return mapPila3Aux(f, p1)

# 4ª solución
# =====

def mapPila4Aux(f: Callable[[A], A], p: Pila[A]) -> Pila[A]:
    r: Pila[A] = Pila()
    while not p.esVacia():
        cp = p.cima()
        p.desapila()
        r.apila(f(cp))
    r1: Pila[A] = Pila()
    while not r.esVacia():
        r1.apila(r.cima())
        r.desapila()
    return r1

def mapPila4(f: Callable[[A], A], p: Pila[A]) -> Pila[A]:
    p1 = deepcopy(p)
    return mapPila4Aux(f, p1)

# Comprobación de equivalencia de las definiciones
# =====

# La propiedad es
@given(p=pilaAleatoria())
def test_mapPila(p: Pila[int]) -> None:
    r = mapPila1(lambda x: x + 1 == 0, p)
    assert mapPila2(lambda x: x + 1 == 0, p) == r
    assert mapPila3(lambda x: x + 1 == 0, p) == r
    assert mapPila4(lambda x: x + 1 == 0, p) == r

# La comprobación es
#     src> poetry run pytest -q mapPila.py
#     1 passed in 0.29s

```

6.7. Pertenencia a una pila

6.7.1. En Haskell

```

-----
-- Utilizando el [tipo abstracto de datos de las pilas](https://bit.ly/3GTTToyK),
-- definir la función
--   pertenecePila :: Eq a => a -> Pila a -> Bool
-- tal que (pertenecePila y p) se verifica si y es un elemento de la
-- pila p. Por ejemplo,
--   pertenecePila 2 (apila 5 (apila 2 (apila 3 vacia))) == True
--   pertenecePila 4 (apila 5 (apila 2 (apila 3 vacia))) == False
-----

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module PertenecePila where

import TAD.Pila (Pila, vacia, apila, esVacia, cima, desapila)
import Transformaciones_pilas_listas (pilaAlista)
import Test.QuickCheck

-- 1ª solución
-- =====

pertenecePila :: Eq a => a -> Pila a -> Bool
pertenecePila x p
  | esVacia p   = False
  | otherwise   = x == cp || pertenecePila x dp
  where cp = cima p
        dp = desapila p

-- 2ª solución
-- =====

-- Se usará la función pilaAlista del ejercicio
-- "Transformaciones entre pilas y listas" que se encuentra en
-- https://bit.ly/3ZHewQ8

pertenecePila2 :: Eq a => a -> Pila a -> Bool
pertenecePila2 x p =

```

```

x `elem` pilaAlista p

-- Comprobación de equivalencia
-- =====

-- La propiedad es
prop_pertenecePila :: Int -> Pila Int -> Bool
prop_pertenecePila x p =
    pertenecePila x p == pertenecePila2 x p

-- La comprobación es
--    λ> quickCheck prop_pertenecePila
--    +++ OK, passed 100 tests.

```

6.7.2. En Python

```

# -----
# Utilizando el [tipo abstracto de las pilas de las pilas](https://bit.ly/3GTTToyK)
# 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
# -----

from copy import deepcopy
from typing import TypeVar

from hypothesis import given
from hypothesis import strategies as st

from src.TAD.pila import (Pila, apila, cima, desapila, esVacia, pilaAleatoria,
                          vacia)
from src.transformaciones_pilas_listas import pilaAlista

A = TypeVar('A')

```

```
# 1ª solución
```

```
# =====
```

```
def pertenecePila(x: A, p: Pila[A]) -> bool:
    if esVacia(p):
        return False
    cp = cima(p)
    dp = desapila(p)
    return x == cp or pertenecePila(x, dp)
```

```
# 2ª solución
```

```
# =====
```

```
# Se usará la función pilaAlista del ejercicio
# "Transformaciones entre pilas y listas" que se encuentra en
# https://bit.ly/3ZHewQ8
```

```
def pertenecePila2(x: A, p: Pila[A]) -> bool:
    return x in pilaAlista(p)
```

```
# 3ª solución
```

```
# =====
```

```
def pertenecePila3Aux(x: A, p: Pila[A]) -> bool:
    if p.esVacia():
        return False
    cp = p.cima()
    p.desapila()
    return x == cp or pertenecePila3Aux(x, p)
```

```
def pertenecePila3(x: A, p: Pila[A]) -> bool:
    p1 = deepcopy(p)
    return pertenecePila3Aux(x, p1)
```

```
# 4ª solución
```

```
# =====
```

```
def pertenecePila4Aux(x: A, p: Pila[A]) -> bool:
    while not p.esVacia():
        cp = p.cima()
```

```

        p.desapila()
        if x == cp:
            return True
    return False

def pertenecePila4(x: A, p: Pila[A]) -> bool:
    p1 = deepcopy(p)
    return pertenecePila4Aux(x, p1)

# Comprobación de equivalencia de las definiciones
# =====

# La propiedad es
@given(x=st.integers(), p=pilaAleatoria())
def test_pertenecePila(x: int, p: Pila[int]) -> None:
    r = pertenecePila(x, p)
    assert pertenecePila2(x, p) == r
    assert pertenecePila3(x, p) == r
    assert pertenecePila4(x, p) == r

# La comprobación es
#   src> poetry run pytest -q pertenecePila.py
#   1 passed in 0.37s

```

6.8. Inclusión de pilas

6.8.1. En Haskell

```

-- -----
-- Utilizando el [tipo abstracto de datos de las pilas](https://bit.ly/3GTTToyK),
-- definir la función
--   contenidaPila :: Eq a => Pila a -> Pila a -> Bool
-- tal que (contenidaPila p1 p2) se verifica si todos los elementos de
-- de la pila p1 son elementos de la pila p2. Por ejemplo,
--   λ> ej1 = apila 3 (apila 2 vacia)
--   λ> ej2 = apila 3 (apila 4 vacia)
--   λ> ej3 = apila 5 (apila 2 (apila 3 vacia))
--   λ> contenidaPila ej1 ej3
--   True
--   λ> contenidaPila ej2 ej3

```

```

--      False
--      -----

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module ContenidaPila where

import TAD.Pila (Pila, vacia, apila, esVacia, cima, desapila)
import PertenecePila (pertenecePila)
import Test.QuickCheck

-- 1ª solución
-- =====

-- Se usará la función pertenecePila del ejercicio
-- "Perteneencia a una pila" que se encuentra en
-- https://bit.ly/3WdM9GC

contenidaPila1 :: Eq a => Pila a -> Pila a -> Bool
contenidaPila1 p1 p2
  | esVacia p1 = True
  | otherwise  = pertenecePila cp1 p2 && contenidaPila1 dp1 p2
  where cp1 = cima p1
        dp1 = desapila p1

-- 2ª solución
-- =====

contenidaPila2 :: Eq a => Pila a -> Pila a -> Bool
contenidaPila2 p1 p2 =
  contenidaLista (pilaAlista p1) (pilaAlista p2)

contenidaLista :: Eq a => [a] -> [a] -> Bool
contenidaLista xs ys =
  all (`elem` ys) xs

-- (pilaAlista p) es la lista formada por los elementos de la
-- lista p. Por ejemplo,
--   λ> pilaAlista (apila 5 (apila 2 (apila 3 vacia)))
--   [3, 2, 5]

```

```

pilaAlista :: Pila a -> [a]
pilaAlista = reverse . aux
  where aux p | esVacia p = []
              | otherwise = cp : aux dp
            where cp = cima p
                  dp = desapila p

-- Comprobación de equivalencia
-- =====

-- La propiedad es
prop_contenidaPila :: Pila Int -> Pila Int -> Bool
prop_contenidaPila p1 p2 =
  contenidaPila1 p1 p2 == contenidaPila2 p1 p2

-- La comprobación es
--   λ> quickCheck prop_contenidaPila
--   +++ OK, passed 100 tests.

```

6.8.2. En Python

```

# -----
# Utilizando el [tipo abstracto de datos de las pilas](https://bit.ly/3GTTToyK),
# 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
# -----

from copy import deepcopy
from typing import TypeVar

from hypothesis import given

```



```
from src.pertenecePila import pertenecePila
from src.TAD.pila import (Pila, apila, cima, desapila, esVacia, pilaAleatoria,
                          vacia)
from src.transformaciones_pilas_listas import pilaAlista

A = TypeVar('A')

# 1ª solución
# =====

# Se usará la función pertenecePila del ejercicio
# "Pertenencia a una pila" que se encuentra en
# https://bit.ly/3WdM9GC

def contenidaPila1(p1: Pila[A], p2: Pila[A]) -> bool:
    if esVacia(p1):
        return True
    cp1 = cima(p1)
    dp1 = desapila(p1)
    return pertenecePila(cp1, p2) and contenidaPila1(dp1, p2)

# 2ª solución
# =====

# Se usará la función pilaAlista del ejercicio
# "Transformaciones entre pilas y listas" que se encuentra en
# https://bit.ly/3ZHewQ8

def contenidaPila2(p1: Pila[A], p2: Pila[A]) -> bool:
    return set(pilaAlista(p1)) <= set(pilaAlista(p2))

# 3ª solución
# =====

def contenidaPila3Aux(p1: Pila[A], p2: Pila[A]) -> bool:
    if p1.esVacia():
        return True
    cp1 = p1.cima()
    p1.desapila()
```

```

    return pertenecePila(cp1, p2) and contenidaPila1(p1, p2)

def contenidaPila3(p1: Pila[A], p2: Pila[A]) -> bool:
    q = deepcopy(p1)
    return contenidaPila3Aux(q, p2)

# 4ª solución
# =====

def contenidaPila4Aux(p1: Pila[A], p2: Pila[A]) -> bool:
    while not p1.esVacia():
        cp1 = p1.cima()
        p1.desapila()
        if not pertenecePila(cp1, p2):
            return False
    return True

def contenidaPila4(p1: Pila[A], p2: Pila[A]) -> bool:
    q = deepcopy(p1)
    return contenidaPila4Aux(q, p2)

# Comprobación de equivalencia de las definiciones
# =====

# La propiedad es
@given(p1=pilaAleatoria(), p2=pilaAleatoria())
def test_contenidaPila(p1: Pila[int], p2: Pila[int]) -> None:
    r = contenidaPila1(p1, p2)
    assert contenidaPila2(p1, p2) == r
    assert contenidaPila3(p1, p2) == r
    assert contenidaPila4(p1, p2) == r

# La comprobación es
#   src> poetry run pytest -q contenidaPila.py
#   1 passed in 0.40s

```

6.9. Reconocimiento de prefijos de pilas

6.9.1. En Haskell

```

-----
-- Utilizando el [tipo abstracto de datos de las pilas](https://bit.ly/3GTTToyK),
-- definir la función
--   prefijoPila :: Eq a => Pila a -> Pila a -> Bool
-- tal que (prefijoPila p1 p2) se verifica si la pila p1 es justamente
-- un prefijo de la pila p2. Por ejemplo,
--   λ> ej1 = apila 4 (apila 2 vacia)
--   λ> ej2 = apila 4 (apila 2 (apila 5 vacia))
--   λ> ej3 = apila 5 (apila 4 (apila 2 vacia))
--   λ> prefijoPila ej1 ej2
--   True
--   λ> prefijoPila ej1 ej3
--   False
-----

```

```
{-# OPTIONS_GHC -fno-warn-unused-imports #-}
```

```
module PrefijoPila where
```

```
import TAD.Pila (Pila, vacia, apila, esVacia, cima, desapila)
import Transformaciones_pilas_listas (pilaAlista)
import Data.List (isSuffixOf)
import Test.QuickCheck
```

```
-- 1ª solución
-- =====
```

```

prefijoPila :: Eq a => Pila a -> Pila a -> Bool
prefijoPila p1 p2
  | esVacia p1 = True
  | esVacia p2 = False
  | otherwise  = cp1 == cp2 && prefijoPila dp1 dp2
where cp1 = cima p1
      dp1 = desapila p1
      cp2 = cima p2
      dp2 = desapila p2

```

```
-- 2ª solución
-- =====

-- Se usará la función pilaAlista del ejercicio
-- "Transformaciones entre pilas y listas" que se encuentra en
-- https://bit.ly/3ZHewQ8

prefijoPila2 :: Eq a => Pila a -> Pila a -> Bool
prefijoPila2 p1 p2 =
  pilaAlista p1 `isSuffixOf` pilaAlista p2

-- Comprobación de equivalencia
-- =====

-- La propiedad es
prop_prefijoPila :: Pila Int -> Pila Int -> Bool
prop_prefijoPila p1 p2 =
  prefijoPila p1 p2 == prefijoPila2 p1 p2

-- La comprobación es
-- λ> quickCheck prop_prefijoPila
-- +++ OK, passed 100 tests.
```

6.9.2. En Python

```
# -----
# Utilizando el [tipo abstracto de datos de las pilas](https://bit.ly/3GTTToyK),
# 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
# -----
```

```
from copy import deepcopy
from typing import TypeVar

from hypothesis import given

from src.TAD.pila import (Pila, apila, cima, desapila, esVacia, pilaAleatoria,
                          vacia)
from src.transformaciones_pilas_listas import pilaAlista

A = TypeVar('A')

# 1ª solución
# =====

def prefijoPila(p1: Pila[A], p2: Pila[A]) -> bool:
    if esVacia(p1):
        return True
    if esVacia(p2):
        return False
    cp1 = cima(p1)
    dp1 = desapila(p1)
    cp2 = cima(p2)
    dp2 = desapila(p2)
    return cp1 == cp2 and prefijoPila(dp1, dp2)

# 2ª solución
# =====

# Se usará la función pilaAlista del ejercicio
# "Transformaciones entre pilas y listas" que se encuentra en
# https://bit.ly/3ZHewQ8

def esSufijoLista(xs: list[A], ys: list[A]) -> bool:
    if not xs:
        return True
    return xs == ys[-len(xs):]

def prefijoPila2(p1: Pila[A], p2: Pila[A]) -> bool:
    return esSufijoLista(pilaAlista(p1), pilaAlista(p2))
```

```
# 3ª solución
```

```
# =====
```

```
def prefijoPila3Aux(p1: Pila[A], p2: Pila[A]) -> bool:
    if p1.esVacia():
        return True
    if p2.esVacia():
        return False
    cp1 = p1.cima()
    p1.desapila()
    cp2 = p2.cima()
    p2.desapila()
    return cp1 == cp2 and prefijoPila3(p1, p2)

def prefijoPila3(p1: Pila[A], p2: Pila[A]) -> bool:
    q1 = deepcopy(p1)
    q2 = deepcopy(p2)
    return prefijoPila3Aux(q1, q2)
```

```
# 4ª solución
```

```
# =====
```

```
def prefijoPila4Aux(p1: Pila[A], p2: Pila[A]) -> bool:
    while not p2.esVacia() and not p1.esVacia():
        if p1.cima() != p2.cima():
            return False
        p1.desapila()
        p2.desapila()
    return p1.esVacia()

def prefijoPila4(p1: Pila[A], p2: Pila[A]) -> bool:
    q1 = deepcopy(p1)
    q2 = deepcopy(p2)
    return prefijoPila4Aux(q1, q2)
```

```
# Comprobación de equivalencia de las definiciones
```

```
# =====
```

```
# La propiedad es
```

```
@given(p1=pilaAleatoria(), p2=pilaAleatoria())
```

```
def test_prefijoPila(p1: Pila[int], p2: Pila[int]) -> None:
    r = prefijoPila(p1, p2)
    assert prefijoPila2(p1, p2) == r
    assert prefijoPila3(p1, p2) == r
    assert prefijoPila4(p1, p2) == r
```

```
# La comprobación es
#     src> poetry run pytest -q prefijoPila.py
#     1 passed in 0.32s
```

6.10. Reconocimiento de subpilas

6.10.1. En Haskell

```
-- -----
-- Utilizando el [tipo abstracto de datos de las pilas](https://bit.ly/3GTTToyK),
-- definir la función
--     subPila :: Eq a => Pila a -> Pila a -> Bool
-- tal que (subPila p1 p2) se verifica si p1 es una subpila de p2. Por
-- ejemplo,
--     λ> ej1 = apila 2 (apila 3 vacia)
--     λ> ej2 = apila 7 (apila 2 (apila 3 (apila 5 vacia)))
--     λ> ej3 = apila 2 (apila 7 (apila 3 (apila 5 vacia)))
--     λ> subPila ej1 ej2
--     True
--     λ> subPila ej1 ej3
--     False
-- -----
```

```
{-# OPTIONS_GHC -fno-warn-unused-imports #-}
```

```
module SubPila where
```

```
import TAD.Pila (Pila, vacia, apila, esVacia, cima, desapila)
import Transformaciones_pilas_listas (pilaAlista)
import PrefijoPila (prefijoPila)
import Data.List (isPrefixOf, tails)
import Test.QuickCheck
```

```
-- 1ª solución
```

```

-- =====

-- Se usará la función PrefijoPila del ejercicio
-- "Reconocimiento de prefijos de pilas" que se encuentra en
-- https://bit.ly/3Xqu7lo

subPila1 :: Eq a => Pila a -> Pila a -> Bool
subPila1 p1 p2
  | esVacía p1 = True
  | esVacía p2 = False
  | cp1 == cp2 = prefijoPila dp1 dp2 || subPila1 p1 dp2
  | otherwise  = subPila1 p1 dp2
  where cp1 = cima p1
        dp1 = desapila p1
        cp2 = cima p2
        dp2 = desapila p2

-- 2ª solución
-- =====

-- Se usará la función pilaAlista del ejercicio
-- "Transformaciones entre pilas y listas" que se encuentra en
-- https://bit.ly/3ZHewQ8

subPila2 :: Eq a => Pila a -> Pila a -> Bool
subPila2 p1 p2 =
  sublista (pilaAlista p1) (pilaAlista p2)

-- (sublista xs ys) se verifica si xs es una sublista de ys. Por
-- ejemplo,
--   sublista [3,2] [5,3,2,7] == True
--   sublista [3,2] [5,3,7,2] == False
sublista :: Eq a => [a] -> [a] -> Bool
sublista xs ys =
  any (xs `isPrefixOf`) (tails ys)

-- Comprobación de equivalencia
-- =====

```



```
-- La propiedad es
prop_subPila :: Pila Int -> Pila Int -> Bool
prop_subPila p1 p2 =
    subPila1 p1 p2 == subPila2 p1 p2

-- La comprobación es
--    λ> quickCheck prop_subPila
--    +++ OK, passed 100 tests.
```

6.10.2. En Python

```
# -----
# Utilizando el [tipo abstracto de datos de las pilas](https://bit.ly/3GTTToyK),
# 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
# -----

from copy import deepcopy
from typing import TypeVar

from hypothesis import given

from src.prefijoPila import prefijoPila
from src.TAD.pila import (Pila, apila, cima, desapila, esVacia, pilaAleatoria,
                          vacia)
from src.transformaciones_pilas_listas import pilaAlista

A = TypeVar('A')

# 1ª solución
# =====
```

```
# Se usará la función PrefijoPila del ejercicio
# "Reconocimiento de prefijos de pilas" que se encuentra en
# https://bit.ly/3Xqu7lo
```

```
def subPila1(p1: Pila[A], p2: Pila[A]) -> bool:
    if esVacia(p1):
        return True
    if esVacia(p2):
        return False
    cp1 = cima(p1)
    dp1 = desapila(p1)
    cp2 = cima(p2)
    dp2 = desapila(p2)
    if cp1 == cp2:
        return prefijoPila(dp1, dp2) or subPila1(p1, dp2)
    return subPila1(p1, dp2)
```

```
# 2ª solución
# =====
```

```
# Se usará la función pilaAlista del ejercicio
# "Transformaciones entre pilas y listas" que se encuentra en
# https://bit.ly/3ZHewQ8
```

```
# sublista(xs, ys) se verifica si xs es una sublista de ys. Por
# ejemplo,
```

```
# >>> sublista([3,2], [5,3,2,7])
# True
# >>> sublista([3,2], [5,3,7,2])
# False
```

```
def sublista(xs: list[A], ys: list[A]) -> bool:
    return any(xs == ys[i:i+len(xs)] for i in range(len(ys) - len(xs) + 1))
```

```
def subPila2(p1: Pila[A], p2: Pila[A]) -> bool:
    return sublista(pilaAlista(p1), pilaAlista(p2))
```

```
# 3ª solución
# =====
```

```

def subPila3Aux(p1: Pila[A], p2: Pila[A]) -> bool:
    if p1.esVacia():
        return True
    if p2.esVacia():
        return False
    if p1.cima() != p2.cima():
        p2.desapila()
        return subPila3Aux(p1, p2)
    q1 = deepcopy(p1)
    p1.desapila()
    p2.desapila()
    return prefijoPila(p1, p2) or subPila3Aux(q1, p2)

def subPila3(p1: Pila[A], p2: Pila[A]) -> bool:
    q1 = deepcopy(p1)
    q2 = deepcopy(p2)
    return subPila3Aux(q1, q2)

# Comprobación de equivalencia de las definiciones
# =====

# La propiedad es
@given(p1=pilaAleatoria(), p2=pilaAleatoria())
def test_subPila(p1: Pila[int], p2: Pila[int]) -> None:
    r = subPila1(p1, p2)
    assert subPila2(p1, p2) == r
    assert subPila3(p1, p2) == r

# La comprobación es
#   src> poetry run pytest -q subPila.py
#   1 passed in 0.32s

```

6.11. Reconocimiento de ordenación de pilas

6.11.1. En Haskell

```

-- -----
-- Utilizando el [tipo abstracto de datos de las pilas](https://bit.ly/3GTTToyK),
-- definir la función
--   ordenadaPila :: Ord a => Pila a -> Bool

```

```

-- tal que (ordenadaPila p) se verifica si los elementos de la pila p
-- están ordenados en orden creciente. Por ejemplo,
--   ordenadaPila (apila 1 (apila 5 (apila 6 vacia))) == True
--   ordenadaPila (apila 1 (apila 0 (apila 6 vacia))) == False
-- -----

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module OrdenadaPila where

import TAD.Pila (Pila, vacia, apila, esVacia, cima, desapila)
import Transformaciones_pilas_listas (pilaAlista)
import Test.QuickCheck

-- 1ª solución
-- =====

ordenadaPila :: Ord a => Pila a -> Bool
ordenadaPila p
  | esVacia p   = True
  | esVacia dp  = True
  | otherwise   = cp <= cdp && ordenadaPila dp
  where cp      = cima p
        dp      = desapila p
        cdp     = cima dp

-- 2ª solución
-- =====

ordenadaPila2 :: Ord a => Pila a -> Bool
ordenadaPila2 =
  ordenadaLista . reverse . pilaAlista

-- (ordenadaLista xs) se verifica si la lista xs está ordenada de menor
-- a mayor. Por ejemplo,
ordenadaLista :: Ord a => [a] -> Bool
ordenadaLista xs =
  and [x <= y | (x,y) <- zip xs (tail xs)]

-- Se usará la función pilaAlista del ejercicio

```

```
-- "Transformaciones entre pilas y listas" que se encuentra en
-- https://bit.ly/3ZHewQ8

-- Comprobación de equivalencia
-- =====

-- La propiedad es
prop_ordenadaPila :: Pila Int -> Bool
prop_ordenadaPila p =
  ordenadaPila p == ordenadaPila2 p

-- La comprobación es
--   λ> quickCheck prop_ordenadaPila
--   +++ OK, passed 100 tests.
```

6.11.2. En Python

```
# -----
# Utilizando el [tipo abstracto de datos de las pilas](https://bit.ly/3GTTToyK),
# 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
# -----

from copy import deepcopy
from typing import TypeVar

from hypothesis import given

from src.TAD.pila import (Pila, apila, cima, desapila, esVacia, pilaAleatoria,
                          vacia)
from src.transformaciones_pilas_listas import pilaAlista

A = TypeVar('A', int, float, str)
```

1ª solución

=====

```
def ordenadaPila(p: Pila[A]) -> bool:
    if esVacia(p):
        return True
    cp = cima(p)
    dp = desapila(p)
    if esVacia(dp):
        return True
    cdp = cima(dp)
    return cp <= cdp and ordenadaPila(dp)
```

2ª solución

=====

Se usará la función pilaAlista del ejercicio

"Transformaciones entre pilas y listas" que se encuentra en

<https://bit.ly/3ZHewQ8>

ordenadaLista(xs, ys) se verifica si xs es una lista ordenada. Por
ejemplo,

>>> ordenadaLista([2, 5, 8])

True

>>> ordenadaLista([2, 8, 5])

False

```
def ordenadaLista(xs: list[A]) -> bool:
    return all((x <= y for (x, y) in zip(xs, xs[1:])))
```

```
def ordenadaPila2(p: Pila[A]) -> bool:
    return ordenadaLista(list(reversed(pilaAlista(p))))
```

3ª solución

=====

```
def ordenadaPila3Aux(p: Pila[A]) -> bool:
    if p.esVacia():
        return True
    cp = p.cima()
    p.desapila()
```

```

    if p.esVacia():
        return True
    return cp <= p.cima() and ordenadaPila3Aux(p)

def ordenadaPila3(p: Pila[A]) -> bool:
    q = deepcopy(p)
    return ordenadaPila3Aux(q)

# 4ª solución
# =====

def ordenadaPila4Aux(p: Pila[A]) -> bool:
    while not p.esVacia():
        cp = p.cima()
        p.desapila()
        if not p.esVacia() and cp > p.cima():
            return False
    return True

def ordenadaPila4(p: Pila[A]) -> bool:
    q = deepcopy(p)
    return ordenadaPila4Aux(q)

# Comprobación de equivalencia de las definiciones
# =====

# La propiedad es
@given(p=pilaAleatoria())
def test_ordenadaPila(p: Pila[int]) -> None:
    r = ordenadaPila(p)
    assert ordenadaPila2(p) == r
    assert ordenadaPila3(p) == r
    assert ordenadaPila4(p) == r

# La comprobación es
#   src> poetry run pytest -q ordenadaPila.py
#   1 passed in 0.31s

```

6.12. Ordenación de pilas por inserción

6.12.1. En Haskell

```

-----
-- Utilizando el [tipo abstracto de datos de las pilas](https://bit.ly/3GTTToyK),
-- definir la función
--   ordenaInserPila :: Ord a => Pila a -> Pila a
-- tal que (ordenaInserPila p) es la pila obtenida ordenando por
-- inserción los los elementos de la pila p. Por ejemplo,
--   λ> ordenaInserPila (apila 4 (apila 1 (apila 3 vacia)))
--   1 | 3 | 4
--
-- Comprobar con QuickCheck que la pila (ordenaInserPila p) está
-- ordenada.
-----

```

```
{-# OPTIONS_GHC -fno-warn-unused-imports #-}
```

```
module OrdenaInserPila where
```

```

import TAD.Pila (Pila, vacia, apila, esVacia, cima, desapila)
import Transformaciones_pilas_listas (listaApila, pilaAlista)
import OrdenadaPila (ordenadaPila)
import Test.QuickCheck

```

```

-- 1ª solución
-- =====

```

```

ordenaInserPila1 :: Ord a => Pila a -> Pila a
ordenaInserPila1 p
  | esVacia p = p
  | otherwise = insertaPila cp (ordenaInserPila1 dp)
  where cp = cima p
        dp = desapila p

```

```

insertaPila :: Ord a => a -> Pila a -> Pila a
insertaPila x p
  | esVacia p = apila x p
  | x < cp    = apila x p
  | otherwise = apila cp (insertaPila x dp)

```



```

    where cp = cima p
          dp = desapila p

-- 2ª solución
-- =====

ordenaInserPila2 :: Ord a => Pila a -> Pila a
ordenaInserPila2 =
    listaApila . reverse . ordenaInserLista . pilaALista

ordenaInserLista :: Ord a => [a] -> [a]
ordenaInserLista [] = []
ordenaInserLista (x:xs) = insertaLista x (ordenaInserLista xs)

insertaLista :: Ord a => a -> [a] -> [a]
insertaLista x [] = [x]
insertaLista x (y:ys) | x < y = x : y : ys
                      | otherwise = y : insertaLista x ys

-- Se usarán las funciones listaApila y pilaALista del ejercicio
-- "Transformaciones entre pilas y listas" que se encuentra en
-- https://bit.ly/3ZHewQ8

-- Comprobación de equivalencia
-- =====

-- La propiedad es
prop_ordenaInserPila :: Pila Int -> Bool
prop_ordenaInserPila p =
    ordenaInserPila1 p == ordenaInserPila2 p

-- La comprobación es
--    λ> quickCheck prop_ordenaInserPila
--    +++ OK, passed 100 tests.

-- Comprobación de la propiedad
-- =====

-- Se usará la función ordenadaPila del ejercicio
-- "Reconocimiento de ordenación de pilas" que se encuentra en

```

```
-- https://bit.ly/3C0qRbK

-- La propiedad es
prop_ordenadaOrdenaInserPila :: Pila Int -> Bool
prop_ordenadaOrdenaInserPila p =
    ordenadaPila (ordenaInserPila1 p)

-- La comprobación es
--    λ> quickCheck prop_ordenadaOrdenaInserPila
--    +++ OK, passed 100 tests.
```

6.12.2. En Python

```
# -----
# Utilizando el [tipo abstracto de datos de las pilas](https://bit.ly/3GTTToyK),
# definir la función
#     ordenaInserPila : (A, Pila[A]) -> Pila[A]
# tal que ordenaInserPila(p) es la pila obtenida ordenando por
# inserción los los elementos de la pila p. Por ejemplo,
#     >>> ordenaInserPila(apila(4, apila(1, apila(3, vacia()))))
#     1 | 3 | 4
#
# Comprobar con Hypothesis que la pila ordenaInserPila(p) está
# ordenada.
# -----

from copy import deepcopy
from typing import TypeVar

from hypothesis import given

from src.ordenadaPila import ordenadaPila
from src.TAD.pila import (Pila, apila, cima, desapila, esVacia, pilaAleatoria,
                          vacia)
from src.transformaciones_pilas_listas import listaApila, pilaAlista

A = TypeVar('A', int, float, str)

# 1ª solución
# =====
```

```
def insertaPila(x: A, p: Pila[A]) -> Pila[A]:
    if esVacia(p):
        return apila(x, p)
    cp = cima(p)
    if x < cp:
        return apila(x, p)
    dp = desapila(p)
    return apila(cp, insertaPila(x, dp))

def ordenaInserPila1(p: Pila[A]) -> Pila[A]:
    if esVacia(p):
        return p
    cp = cima(p)
    dp = desapila(p)
    return insertaPila(cp, ordenaInserPila1(dp))

# 2ª solución
# =====

# Se usarán las funciones listaApila y pilaAlista del ejercicio
# "Transformaciones entre pilas y listas" que se encuentra en
# https://bit.ly/3ZHewQ8

def insertaLista(x: A, ys: list[A]) -> list[A]:
    if not ys:
        return [x]
    if x < ys[0]:
        return [x] + ys
    return [ys[0]] + insertaLista(x, ys[1:])

def ordenaInserLista(xs: list[A]) -> list[A]:
    if not xs:
        return []
    return insertaLista(xs[0], ordenaInserLista(xs[1:]))

def ordenaInserPila2(p: Pila[A]) -> Pila[A]:
    return listaApila(list(reversed(ordenaInserLista(pilaAlista(p)))))

# 3ª solución
```

```

# =====

def ordenaInserPila3Aux(p: Pila[A]) -> Pila[A]:
    if p.esVacia():
        return p
    cp = p.cima()
    p.desapila()
    return insertaPila(cp, ordenaInserPila3Aux(p))

def ordenaInserPila3(p: Pila[A]) -> Pila[A]:
    q = deepcopy(p)
    return ordenaInserPila3Aux(q)

# Comprobación de equivalencia de las definiciones
# =====

# La propiedad es
@given(p=pilaAleatoria())
def test_ordenaInserPila(p: Pila[int]) -> None:
    r = ordenaInserPila1(p)
    assert ordenaInserPila2(p) == r
    assert ordenaInserPila3(p) == r

# La comprobación es
# src> poetry run pytest -q ordenaInserPila.py
# 1 passed in 0.31s

# Comprobación de la propiedad
# =====

# Se usará la función ordenadaPila del ejercicio
# "Reconocimiento de ordenación de pilas" que se encuentra en
# https://bit.ly/3C0qRbK

# La propiedad es
@given(p=pilaAleatoria())
def test_ordenadaOrdenaInserPila(p: Pila[int]) -> None:
    ordenadaPila(ordenaInserPila1(p))

# La comprobación es

```

```
# src> poetry run pytest -q ordenaInserPila.py
# 2 passed in 0.47s
```

6.13. Eliminación de repeticiones en una pila

6.13.1. En Haskell

```
-- -----
-- Utilizando el [tipo abstracto de datos de las pilas](https://bit.ly/3GTTToyK),
-- definir la función
--   nubPila :: Eq a => Pila a -> Pila a
-- tal que (nubPila p) es la pila con los elementos de p sin repeticiones.
-- Por ejemplo,
--   λ> nubPila (apila 3 (apila 1 (apila 3 (apila 5 vacia))))
--   1 | 3 | 5
-- -----

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module NubPila where

import TAD.Pila (Pila, vacia, apila, esVacia, cima, desapila)
import Transformaciones_pilas_listas (listaApila, pilaAlista)
import PertenecePila (pertenecePila)
import Data.List (nub)
import Test.QuickCheck

-- 1ª solución
-- =====

-- Se usará la función pertenecePila del ejercicio
-- "Perteneencia a una pila" que se encuentra en
-- https://bit.ly/3WdM9GC

nubPila1 :: Eq a => Pila a -> Pila a
nubPila1 p
  | esVacia p           = vacia
  | pertenecePila cp dp = nubPila1 dp
  | otherwise           = apila cp (nubPila1 dp)
  where cp = cima p
```

```

        dp = desapila p

-- 2ª solución
-- =====

-- Se usarán las funciones listaApila y pilaAlista del ejercicio
-- "Transformaciones entre pilas y listas" que se encuentra en
-- https://bit.ly/3ZHewQ8

nubPila2 :: Eq a => Pila a -> Pila a
nubPila2 =
    listaApila . nub . pilaAlista

-- Comprobación de equivalencia
-- =====

-- La propiedad es
prop_nubPila :: Pila Int -> Bool
prop_nubPila p =
    nubPila1 p == nubPila2 p

-- La comprobación es
--    λ> quickCheck prop_nubPila
--    +++ OK, passed 100 tests.

```

6.13.2. En Python

```

# -----
# Utilizando el [tipo abstracto de datos de las pilas](https://bit.ly/3GTTToyK),
# definir la función
#     nubPila : (Pila[A]) -> Pila[A]
# tal que nubPila(p) es la pila con los elementos de p sin repeticiones.
# Por ejemplo,
#     >>> ej = apila(3, apila(1, apila(3, apila(5, vacia()))))
#     >>> ej
#     3 | 1 | 3 | 5
#     >>> nubPila1(ej)
#     1 | 3 | 5
# -----

```

```
from copy import deepcopy
from typing import TypeVar

from hypothesis import given

from src.pertenecePila import pertenecePila
from src.TAD.pila import (Pila, apila, cima, desapila, esVacia, pilaAleatoria,
                          vacia)
from src.transformaciones_pilas_listas import listaApila, pilaAlista

A = TypeVar('A')

# 1ª solución
# =====

# Se usará la función pertenecePila del ejercicio
# "Pertenencia a una pila" que se encuentra en
# https://bit.ly/3WdM9GC

def nubPila1(p: Pila[A]) -> Pila[A]:
    if esVacia(p):
        return p
    cp = cima(p)
    dp = desapila(p)
    if pertenecePila(cp, dp):
        return nubPila1(dp)
    return apila(cp, nubPila1(dp))

# 2ª solución
# =====

# Se usarán las funciones listaApila y pilaAlista del ejercicio
# "Transformaciones entre pilas y listas" que se encuentra en
# https://bit.ly/3ZHewQ8

def nub(xs: list[A]) -> list[A]:
    return [x for i, x in enumerate(xs) if x not in xs[:i]]

def nubPila2(p: Pila[A]) -> Pila[A]:
    return listaApila(nub(pilaAlista(p)))
```

```

# 3ª solución
# =====

def nubPila3Aux(p: Pila[A]) -> Pila[A]:
    if p.esVacia():
        return p
    cp = p.cima()
    p.desapila()
    if pertenecePila(cp, p):
        return nubPila3Aux(p)
    return apila(cp, nubPila3Aux(p))

def nubPila3(p: Pila[A]) -> Pila[A]:
    q = deepcopy(p)
    return nubPila3Aux(q)

# Comprobación de equivalencia de las definiciones
# =====

# La propiedad es
@given(p=pilaAleatoria())
def test_nubPila(p: Pila[int]) -> None:
    r = nubPila1(p)
    assert nubPila2(p) == r
    assert nubPila3(p) == r

# La comprobación es
#   src> poetry run pytest -q nubPila.py
#   1 passed in 0.27s

```

6.14. Máximo elemento de una pila

6.14.1. En Haskell

```

-- -----
-- Utilizando el [tipo abstracto de datos de las pilas](https://bit.ly/3GTTToyK),
-- definir la función
--   maxPila :: Ord a => Pila a -> a
-- tal que (maxPila p) sea el mayor de los elementos de la pila p. Por

```



```

-- ejemplo,
--   λ> maxPila (apila 3 (apila 5 (apila 1 vacia)))
--   5
-- -----

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module MaxPila where

import TAD.Pila (Pila, vacia, apila, esVacia, cima, desapila)
import Transformaciones_pilas_listas (pilaAlista)
import Test.QuickCheck

-- 1ª solución
-- =====

maxPila1 :: Ord a => Pila a -> a
maxPila1 p
  | esVacia dp = cp
  | otherwise  = max cp (maxPila1 dp)
  where cp = cima p
        dp = desapila p

-- 2ª solución
-- =====

-- Se usará la función pilaAlista del ejercicio
-- "Transformaciones entre pilas y listas" que se encuentra en
-- https://bit.ly/3ZHewQ8

maxPila2 :: Ord a => Pila a -> a
maxPila2 =
  maximum . pilaAlista

-- Comprobación de equivalencia
-- =====

-- La propiedad es
prop_maxPila :: Pila Int -> Property
prop_maxPila p =

```

```
not (esVacia p) ==> maxPila1 p == maxPila2 p
```

```
-- La comprobación es
--   λ> quickCheck prop_maxPila
--   +++ OK, passed 100 tests; 17 discarded.
```

6.14.2. En Python

```
# -----
# Utilizando el [tipo abstracto de datos de las pilas](https://bit.ly/3GTTToyK),
# 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
# -----
```

```
from copy import deepcopy
from typing import TypeVar
```

```
from hypothesis import assume, given
```

```
from src.TAD.pila import (Pila, apila, cima, desapila, esVacia, pilaAleatoria,
                          vacia)
from src.transformaciones_pilas_listas import pilaAlista
```

```
A = TypeVar('A', int, float, str)
```

```
# 1ª solución
```

```
# =====
```

```
def maxPila1(p: Pila[A]) -> A:
    cp = cima(p)
    dp = desapila(p)
    if esVacia(dp):
        return cp
    return max(cp, maxPila1(dp))
```

```
# 2ª solución
```

```
# =====

# Se usará la función pilaAlista del ejercicio
# "Transformaciones entre pilas y listas" que se encuentra en
# https://bit.ly/3ZHewQ8

def maxPila2(p: Pila[A]) -> A:
    return max(pilaAlista(p))

# 3ª solución
# =====

def maxPila3Aux(p: Pila[A]) -> A:
    cp = p.cima()
    p.desapila()
    if esVacia(p):
        return cp
    return max(cp, maxPila3Aux(p))

def maxPila3(p: Pila[A]) -> A:
    q = deepcopy(p)
    return maxPila3Aux(q)

# 4ª solución
# =====

def maxPila4Aux(p: Pila[A]) -> A:
    r = p.cima()
    while not esVacia(p):
        cp = p.cima()
        if cp > r:
            r = cp
        p.desapila()
    return r

def maxPila4(p: Pila[A]) -> A:
    q = deepcopy(p)
    return maxPila4Aux(q)

# Comprobación de equivalencia de las definiciones
```

```
# =====  
  
# La propiedad es  
@given(p=pilaAleatoria())  
def test_maxPila(p: Pila[int]) -> None:  
    assume(not esVacia(p))  
    r = maxPila1(p)  
    assert maxPila2(p) == r  
    assert maxPila3(p) == r  
    assert maxPila4(p) == r  
  
# La comprobación es  
#     src> poetry run pytest -q maxPila.py  
#     1 passed in 0.25s
```

Capítulo 7

El tipo abstracto de datos de las colas

7.1. El tipo abstracto de datos de las colas

7.1.1. En Haskell

```
-- Una cola es una estructura de datos, caracterizada por ser una
-- secuencia de elementos en la que la operación de inserción se realiza
-- por un extremo (el posterior o final) y la operación de extracción
-- por el otro (el anterior o frente).
--
-- Las operaciones que definen a tipo abstracto de datos (TAD) de las
-- colas (cuyos elementos son del tipo a) son las siguientes:
--   vacia    :: Cola a
--   inserta  :: a -> Cola a -> Cola a
--   primero  :: Cola a -> a
--   resto    :: Cola a -> Cola a
--   esVacia  :: Cola a -> Bool
--   tales que
--     + vacia es la cola vacía.
--     + (inserta x c) es la cola obtenida añadiendo x al final de c.
--     + (primero c) es el primero de la cola c.
--     + (resto c) es la cola obtenida eliminando el primero de c.
--     + (esVacia c) se verifica si c es la cola vacía.
--
-- Las operaciones tienen que verificar las siguientes propiedades:
--   + primero (inserta x vacia) == x
--   + Si c es una cola no vacía, entonces primero (inserta x c) == primero c,
```

```
-- + resto (inserta x vacia) == vacia
-- + Si c es una cola no vacía, entonces resto (inserta x c) == inserta x (resto 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.

{-# OPTIONS_GHC -fno-warn-unused-imports #-}
```

module TAD.Cola

```
(Cola,
 vacia,      -- Cola a
 inserta,    -- a -> Cola a -> Cola a
 primero,    -- Cola a -> a
 resto,      -- Cola a -> Cola a
 esVacia,    -- Cola a -> Bool
) where
```

import TAD.ColaConListas

```
-- import TAD.ColaConDosListas
-- import TAD.ColaConSucesiones
```

7.1.2. En Python

```
# Una cola es una estructura de datos, caracterizada por ser una
# secuencia de elementos en la que la operación de inserción se realiza
# por un extremo (el posterior o final) y la operación de extracción
# por el otro (el anterior o frente).
#
# Las operaciones que definen a tipo abstracto de datos (TAD) de las
# colas (cuyos elementos son del tipo a) son las siguientes:
#   vacia    :: Cola a
#   inserta  :: a -> Cola a -> Cola a
#   primero  :: Cola a -> a
#   resto    :: Cola a -> Cola a
#   esVacia  :: Cola a -> Bool
# tales que
```


7.2. El tipo de datos de las colas mediante listas

7.2.1. En Haskell

```
{-# LANGUAGE FlexibleInstances #-}
{-# OPTIONS_GHC -fno-warn-unused-top-binds #-}

module TAD.ColaConListas
  ( Cola,
    vacia,    -- Cola a
    inserta,  -- a -> Cola a -> Cola a
    primero,  -- Cola a -> a
    resto,    -- Cola a -> Cola a
    esVacia,  -- Cola a -> Bool
  ) where

import Test.QuickCheck

-- Colas como listas:
newtype Cola a = C [a]
  deriving Eq

-- (escribeCola c) es la cadena correspondiente a la cola c. Por
-- ejemplo,
--     escribeCola (inserta 5 (inserta 2 (inserta 3 vacia))) == "3 | 2 | 5"
escribeCola :: Show a => Cola a -> String
escribeCola (C [])      = "-"
escribeCola (C [x])     = show x
escribeCola (C (x:xs)) = show x ++ " | " ++ escribeCola (C xs)

-- Procedimiento de escritura de colas.
instance Show a => Show (Cola a) where
  show = escribeCola

-- Ejemplo de cola:
--     λ> inserta 5 (inserta 2 (inserta 3 vacia))
--     3 | 2 | 5

-- vacia es la cola vacía. Por ejemplo,
```



```

--      λ> vacia
--      -
vacia :: Cola a
vacia = C []

-- (inserta x c) es la cola obtenida añadiendo x al final de la cola
-- c. Por ejemplo,
--      λ> ej = inserta 2 (inserta 3 vacia)
--      λ> ej
--      3 | 2
--      λ> inserta 5 ej
--      3 | 2 | 5
inserta :: a -> Cola a -> Cola a
inserta x (C c) = C (c ++ [x])

-- (primero c) es el primer elemento de la cola c. Por ejemplo,
--      λ> primero (inserta 5 (inserta 2 (inserta 3 vacia)))
--      3
primero :: Cola a -> a
primero (C []) = error "primero: cola vacia"
primero (C (x:_)) = x

-- (resto c) es la cola obtenida eliminando el primer elemento de la
-- cola c. Por ejemplo,
--      λ> resto (inserta 5 (inserta 2 (inserta 3 vacia)))
--      2 | 5
resto :: Cola a -> Cola a
resto (C (_:xs)) = C xs
resto (C []) = error "resto: cola vacia"

-- (esVacia c) se verifica si c es la cola vacía. Por ejemplo,
--      esVacia (inserta 5 (inserta 2 (inserta 3 vacia))) == False
--      esVacia vacia == True
esVacia :: Cola a -> Bool
esVacia (C xs) = null xs

-- Generador de colas
-- =====

-- genCola es un generador de colas de enteros. Por ejemplo,

```

```

--      λ> sample genCola
--      -
--      -
--      -3 | 2
--      6 | 0 | 1
--      -5 | 0 | -5 | 0 | -4
--      2 | 9 | -6 | 9 | 0 | -1
--      -
--      11 | -5 | 5
--      -
--      16 | 6 | 15 | -3 | -9
--      11 | 6 | 15 | 13 | 20 | -7 | 11 | -5 | 13
genCola :: (Arbitrary a, Num a) => Gen (Cola a)
genCola = do
  xs <- listOf arbitrary
  return (foldr inserta vacia xs)

-- El tipo pila es una instancia del arbitrario.
instance (Arbitrary a, Num a) => Arbitrary (Cola a) where
  arbitrary = genCola

-- Propiedades de las colas
-- =====

-- Las propiedades son
prop_colas1 :: Int -> Cola Int -> Bool
prop_colas1 x c =
  primero (inserta x vacia) == x &&
  resto (inserta x vacia) == vacia &&
  esVacia vacia &&
  not (esVacia (inserta x c))

prop_colas2 :: Int -> Cola Int -> Property
prop_colas2 x c =
  not (esVacia c) ==>
  primero (inserta x c) == primero c &&
  resto (inserta x c) == inserta x (resto c)

-- La comprobación es:
--      λ> quickCheck prop_colas1

```

```
--      +++ OK, passed 100 tests.
--      λ> quickCheck prop_colas2
--      +++ OK, passed 100 tests; 3 discarded.
```

7.2.2. En Python

```
# Se define la clase Cola con los siguientes métodos:
#   + inserta(x) añade x al final de la cola.
#   + primero() es el primero de la cola.
#   + resto() elimina el primero de la cola.
#   + esVacia() se verifica si la cola es vacía.
# Por ejemplo,
#   >>> c = Cola()
#   >>> c
#   -
#   >>> c.inserta(5)
#   >>> c.inserta(2)
#   >>> c.inserta(3)
#   >>> c.inserta(4)
#   >>> c
#   5 | 2 | 3 | 4
#   >>> c.primer()
#   5
#   >>> c.resto()
#   >>> c
#   2 | 3 | 4
#   >>> c.esVacia()
#   False
#   >>> c = Cola()
#   >>> c.esVacia()
#   True
#
# Además se definen las correspondientes funciones. Por ejemplo,
#   >>> vacia()
#   -
#   >>> inserta(4, inserta(3, inserta(2, inserta(5, vacia()))))
#   5 | 2 | 3 | 4
#   >>> inserta(4, inserta(3, inserta(2, inserta(5, vacia()))))
#   5
#   >>> resto(inserta(4, inserta(3, inserta(2, inserta(5, vacia())))))
```

```

#     2 | 3 | 4
#     >>> esVacia(inserta(4, inserta(3, inserta(2, inserta(5, vacia())))))
#     False
#     >>> esVacia(vacia())
#     True
#
# Finalmente, se define un generador aleatorio de colas y se comprueba
# que las colas cumplen las propiedades de su especificación.

__all__ = [
    'Cola',
    'vacía',
    'inserta',
    'primero',
    'resto',
    'esVacia',
    'colaAleatoria'
]

from copy import deepcopy
from dataclasses import dataclass, field
from typing import Generic, TypeVar

from hypothesis import assume, given
from hypothesis import strategies as st

A = TypeVar('A')

# Clase de las colas mediante listas
# =====

@dataclass
class Cola(Generic[A]):
    _elementos: list[A] = field(default_factory=list)

    def __repr__(self) -> str:
        """
        Devuelve una cadena con los elementos de la cola separados por " | ".
        Si la cola está vacía, devuelve "-".
        """

```

```

        if not self._elementos:
            return '-'
        return ' | '.join(str(x) for x in self._elementos)

def inserta(self, x: A) -> None:
    """
    Inserta el elemento x al final de la cola.
    """
    self._elementos.append(x)

def esVacia(self) -> bool:
    """
    Comprueba si la cola está vacía.

    Devuelve True si la cola está vacía, False en caso contrario.
    """
    return not self._elementos

def primero(self) -> A:
    """
    Devuelve el primer elemento de la cola.
    """
    return self._elementos[0]

def resto(self) -> None:
    """
    Elimina el primer elemento de la cola
    """
    self._elementos.pop(0)

# Funciones del tipo de las listas
# =====

def vacia() -> Cola[A]:
    """
    Crea y devuelve una cola vacía de tipo A.
    """
    c: Cola[A] = Cola()
    return c

```

```
def inserta(x: A, c: Cola[A]) -> Cola[A]:
    """
    Inserta un elemento x en la cola c y devuelve una nueva cola con
    el elemento insertado.
    """
    _aux = deepcopy(c)
    _aux.inserta(x)
    return _aux

def esVacia(c: Cola[A]) -> bool:
    """
    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
# 1 passed in 0.24s

```

7.3. El tipo de datos de las colas mediante dos listas

7.3.1. En Haskell

```

-- listas (xs,ys) de modo que los elementos de c son, en ese orden, los
-- elementos de la lista xs++(reverse ys).
--
-- Al dividir la lista en dos parte e invertir la segunda de ellas,
-- esperamos hacer más eficiente las operaciones sobre las colas.
--
-- Impondremos también una restricción adicional sobre la
-- representación: las colas serán representadas mediante pares (xs,ys)
-- tales que si xs es vacía, entonces ys será también vacía. Esta
-- restricción ha de ser conservada por los programas que crean colas.

```

```

{-# OPTIONS_GHC -fno-warn-unused-top-binds #-}

module TAD.ColaConDosListas
  ( Cola,
    vacia,      -- Cola a
    inserta,    -- a -> Cola a -> Cola a
    primero,    -- Cola a -> a
    resto,      -- Cola a -> Cola a
    esVacia,    -- Cola a -> Bool
  ) where

import Test.QuickCheck

-- Las colas como pares listas.
newtype Cola a = C ([a],[a])

-- (escribeCola p) es la cadena correspondiente a la cola p. Por
-- ejemplo,
--   λ> escribeCola (inserta 5 (inserta 2 (inserta 3 vacia)))
--   "3 | 2 | 5"
escribeCola :: Show a => Cola a -> String
escribeCola (C (xs,ys)) = aux (xs ++ reverse ys)
  where aux []      = "-"
        aux [z]     = show z
        aux (z:zs) = show z ++ " | " ++ aux zs

-- Procedimiento de escritura de colas.
instance Show a => Show (Cola a) where
  show = escribeCola

-- Ejemplo de cola:
--   λ> inserta 5 (inserta 2 (inserta 3 vacia))
--   3 | 2 | 5

-- vacia es la cola vacía. Por ejemplo,
--   λ> vacia
--   -
vacia :: Cola a
vacia = C ([],[ ])

```



```

-- (inserta x c) es la cola obtenida añadiendo x al final de la cola
-- c. Por ejemplo,
--   λ> inserta 5 (inserta 2 (inserta 3 vacia))
--   3 | 2 | 5
inserta :: a -> Cola a -> Cola a
inserta y (C (xs,ys)) = C (normaliza (xs,y:ys))

-- (normaliza p) es la cola obtenida al normalizar el par de listas
-- p. Por ejemplo,
--   normaliza ([],[2,5,3]) == ([3,5,2],[ ])
--   normaliza ([4],[2,5,3]) == ([4],[2,5,3])
normaliza :: ([a],[a]) -> ([a],[a])
normaliza ([ ], ys) = (reverse ys, [ ])
normaliza p         = p

-- (primero c) es el primer elemento de la cola c. Por ejemplo,
--   λ> primero (inserta 5 (inserta 2 (inserta 3 vacia)))
--   3
primero  :: Cola a -> a
primero (C (x:_,_)) = x
primero _           = error "primero: cola vacia"

-- (resto c) es la cola obtenida eliminando el primer elemento de la
-- cola c. Por ejemplo,
--   λ> resto (inserta 5 (inserta 2 (inserta 3 vacia)))
--   2 | 5
resto    :: Cola a -> Cola a
resto (C ([],[ ])) = error "resto: cola vacia"
resto (C (_,xs,ys)) = C (normaliza (xs,ys))
resto (C ([ ],_: _)) = error "Imposible"

-- (esVacia c) se verifica si c es la cola vacía. Por ejemplo,
--   esVacia (inserta 5 (inserta 2 (inserta 3 vacia))) == False
--   esVacia vacia == True
esVacia :: Cola a -> Bool
esVacia (C (xs, _)) = null xs

-- (valida c) se verifica si la cola c es válida; es decir, si
-- su primer elemento es vacío entonces también lo es el segundo. Por

```

```

-- ejemplo,
-- valida (C ([2],[5])) == True
-- valida (C ([2],[])) == True
-- valida (C ([],[5])) == False
valida :: Cola a -> Bool
valida (C (xs,ys)) = not (null xs) || null ys

-----
-- Igualdad de colas
-----

-- (elementos c) es la lista de los elementos de la cola c en el orden de
-- la cola. Por ejemplo,
-- λ> elementos (inserta 5 (inserta 2 (inserta 3 vacia)))
-- [3,2,5]
elementos :: Cola a -> [a]
elementos (C (xs,ys)) = xs ++ reverse ys

-- (igualColas c1 c2) se verifica si las colas c1 y c2 son iguales. Por
-- ejemplo,
-- igualColas (C ([3,2],[5,4,7])) (C ([3],[5,4,7,2])) == True
-- igualColas (C ([3,2],[5,4,7])) (C ([],[5,4,7,2,3])) == False
igualColas :: Eq a => Cola a -> Cola a -> Bool
igualColas c1 c2 =
  valida c1 &&
  valida c2 &&
  elementos c1 == elementos c2

instance Eq a => Eq (Cola a) where
  (==) = igualColas

-- Generador de colas
-- =====

-- genCola es un generador de colas de enteros. Por ejemplo,
-- λ> sample genCola
-- -
-- -
-- -3 | 2
-- 6 | 0 | 1

```

```

--      -5 | 0 | -5 | 0 | -4
--      2 | 9 | -6 | 9 | 0 | -1
--      -
--      11 | -5 | 5
--      -
--      16 | 6 | 15 | -3 | -9
--      11 | 6 | 15 | 13 | 20 | -7 | 11 | -5 | 13
genCola :: (Arbitrary a, Num a) => Gen (Cola a)
genCola = do
  xs <- listOf arbitrary
  return (foldr inserta vacia xs)

-- El tipo pila es una instancia del arbitrario.
instance (Arbitrary a, Num a) => Arbitrary (Cola a) where
  arbitrary = genCola

-- Propiedades de las colas
-- =====

-- Las propiedades son
prop_colas1 :: Int -> Cola Int -> Bool
prop_colas1 x c =
  primero (inserta x vacia) == x &&
  resto (inserta x vacia) == vacia &&
  esVacia vacia &&
  not (esVacia (inserta x c))

prop_colas2 :: Int -> Cola Int -> Property
prop_colas2 x c =
  not (esVacia c) ==>
  primero (inserta x c) == primero c &&
  resto (inserta x c) == inserta x (resto c)

-- La comprobación es:
--      λ> quickCheck prop_colas1
--      +++ OK, passed 100 tests.
--      λ> quickCheck prop_colas2
--      +++ OK, passed 100 tests; 3 discarded.

```

7.3.2. En Python

```
# Se define la clase Cola con los siguientes métodos:
# + inserta(x) añade x al final de la cola.
# + primero() es el primero de la cola.
# + resto() elimina el primero de la cola.
# + esVacia() se verifica si la cola es vacía.
# Por ejemplo,
# >>> c = Cola()
# >>> c
# -
# >>> c.inserta(5)
# >>> c.inserta(2)
# >>> c.inserta(3)
# >>> c.inserta(4)
# >>> c
# 5 | 2 | 3 | 4
# >>> c.primer()
# 5
# >>> c.resto()
# >>> c
# 2 | 3 | 4
# >>> c.esVacia()
# False
# >>> c = Cola()
# >>> c.esVacia()
# True
#
# Además se definen las correspondientes funciones. Por ejemplo,
# >>> vacia()
# -
# >>> inserta(4, inserta(3, inserta(2, inserta(5, vacia()))))
# 5 | 2 | 3 | 4
# >>> primero(inserta(4, inserta(3, inserta(2, inserta(5, vacia()))))
# 5
# >>> resto(inserta(4, inserta(3, inserta(2, inserta(5, vacia()))))
# 2 | 3 | 4
# >>> esVacia(inserta(4, inserta(3, inserta(2, inserta(5, vacia()))))
# False
# >>> esVacia(vacia())
# True
```

```

#
# Finalmente, se define un generador aleatorio de colas y se comprueba
# que las colas cumplen las propiedades de su especificación.

__all__ = [
    'Cola',
    'vacía',
    'inserta',
    'primero',
    'resto',
    'esVacía',
    'colaAleatoria'
]

from copy import deepcopy
from dataclasses import dataclass, field
from typing import Any, Generic, TypeVar

from hypothesis import assume, given
from hypothesis import strategies as st

A = TypeVar('A')

# Clase de las colas mediante listas
# =====

@dataclass
class Cola(Generic[A]):
    _primera: list[A] = field(default_factory=list)
    _segunda: list[A] = field(default_factory=list)

    def _elementos(self) -> list[A]:
        """
        Devuelve una lista con los elementos de la cola en orden.
        """
        return self._primera + self._segunda[::-1]

    def __repr__(self) -> str:
        """
        Devuelve una cadena con los elementos de la cola separados por " | ".

```

```

Si la cola está vacía, devuelve "-".
"""

elementos = self._elementos()
if not elementos:
    return "-"
return " | ".join(map(str, elementos))

def __eq__(self, c: Any) -> bool:
    """
    Comprueba si la cola actual es igual a otra cola.
    Se considera que dos colas son iguales si tienen los mismos
    elementos en el mismo orden.

    Parámetro:
    - c (Cola): La cola con la que se va a comparar.

    Devuelve True si las dos colas son iguales, False en caso
    contrario.
    """

    return self._elementos() == c._elementos()

def inserta(self, y: A) -> None:
    """
    Inserta el elemento y en la cola.
    """

    xs = self._primera
    ys = self._segunda
    # Si no hay elementos en la primera lista, se inserta en la segunda
    if not xs:
        ys.insert(0, y)
        # Se invierte la segunda lista y se asigna a la primera
        self._primera = ys[::-1]
        self._segunda = []
    else:
        # Si hay elementos en la primera lista, se inserta en la segunda
        ys.insert(0, y)

def esVacia(self) -> bool:
    """
    Devuelve si la cola está vacía.

```

```

        """
        return not self._primera

def primero(self) -> A:
    """
    Devuelve el primer elemento de la cola.
    """
    return self._primera[0]

def resto(self) -> None:
    """
    Elimina el primer elemento de la cola.
    """
    xs = self._primera
    ys = self._segunda
    del xs[0]
    if not xs:
        self._primera = ys[::-1]
        self._segunda = []

# Funciones del tipo de las listas
# =====

def vacia() -> Cola[A]:
    """
    Crea y devuelve una cola vacía de tipo A.
    """
    c: Cola[A] = Cola()
    return c

def inserta(x: A, c: Cola[A]) -> Cola[A]:
    """
    Inserta un elemento x en la cola c y devuelve una nueva cola con
    el elemento insertado.
    """
    _aux = deepcopy(c)
    _aux.inserta(x)
    return _aux

def esVacia(c: Cola[A]) -> bool:

```

```

    """
    Devuelve True si la cola está vacía, False si no lo está.
    """
    return c.esVacia()

def primero(c: Cola[A]) -> A:
    """
    Devuelve el primer elemento de la cola c.
    """
    return c.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
# 2 passed in 0.40s

```

7.4. El tipo de datos de las colas mediante sucesiones

7.4.1. En Haskell

```

{-# LANGUAGE FlexibleInstances #-}
{-# OPTIONS_GHC -fno-warn-unused-top-binds #-}

module TAD.ColaConSucesiones
  ( Cola,
    vacia,    -- Cola a
    inserta,  -- a -> Cola a -> Cola a
    primero,  -- Cola a -> a
    resto,    -- Cola a -> Cola a
    esVacia,  -- Cola a -> Bool
  ) where

import Data.Sequence as S
import Test.QuickCheck

-- Colas como sucesiones:
newtype Cola a = C (Seq a)
  deriving Eq

-- (escribeCola c) es la cadena correspondiente a la cola c. Por

```

```

-- ejemplo,
--     escribeCola (inserta 5 (inserta 2 (inserta 3 vacia))) == "3 | 2 | 5"
escribeCola :: Show a => Cola a -> String
escribeCola (C xs) = case viewl xs of
    EmptyL    -> "-"
    x :< xs' -> case viewl xs' of
        EmptyL -> show x
        _      -> show x ++ " | " ++ escribeCola (C xs')

-- Procedimiento de escritura de colas.
instance Show a => Show (Cola a) where
    show = escribeCola

-- Ejemplo de cola:
--     λ> inserta 5 (inserta 2 (inserta 3 vacia))
--     3 | 2 | 5

-- vacia es la cola vacía. Por ejemplo,
--     λ> vacia
--     C []
vacia :: Cola a
vacia = C empty

-- (inserta x c) es la cola obtenida añadiendo x al final de la cola
-- c. Por ejemplo,
--     λ> ej = inserta 2 (inserta 3 vacia)
--     λ> ej
--     3 | 2
--     λ> inserta 5 ej
--     3 | 2 | 5
inserta :: a -> Cola a -> Cola a
inserta x (C xs) = C (xs |> x )

-- (primero c) es el primer elemento de la cola c. Por ejemplo,
--     λ> primero (inserta 5 (inserta 2 (inserta 3 vacia)))
--     3
primero :: Cola a -> a
primero (C xs) = case viewl xs of
    EmptyL -> error "primero de la pila vacia"
    x :< _ -> x

```

```

-- (resto c) es la cola obtenida eliminando el primer elemento de la
-- cola c. Por ejemplo,
--   λ> resto (inserta 5 (inserta 2 (inserta 3 vacia)))
--   2 | 5
resto :: Cola a -> Cola a
resto (C xs) = case viewl xs of
  EmptyL   -> error "resto la pila vacia"
  _ :< xs' -> C xs'

-- (esVacia c) se verifica si c es la cola vacía. Por ejemplo,
--   esVacia (inserta 5 (inserta 2 (inserta 3 vacia))) == False
--   esVacia vacia == True
esVacia :: Cola a -> Bool
esVacia (C xs) = S.null xs

-- Generador de colas
-- =====

-- genCola es un generador de colas de enteros. Por ejemplo,
--   λ> sample genCola
--   -
--   2 | -2
--   0 | 0 | 0 | 4
--   -
--   2
--   -1 | -6 | 9
--   12 | -12 | -12 | 7 | -2 | -3 | 5 | -8 | -3 | -9 | -6
--   -11 | -5 | -7 | -8 | -10 | 8 | -9 | -7 | 6 | -12 | 8 | -9 | -1
--   -16 | -12
--   -17 | -17 | 1 | 2 | -15 | -15 | -13 | 8 | 13 | -12 | 15
--   -16 | -18
genCola :: (Arbitrary a, Num a) => Gen (Cola a)
genCola = do
  xs <- listOf arbitrary
  return (foldr inserta vacia xs)

-- El tipo pila es una instancia del arbitrario.
instance (Arbitrary a, Num a) => Arbitrary (Cola a) where
  arbitrary = genCola

```

```

-- Propiedades de las colas
-- =====

-- Las propiedades son
prop_colas1 :: Int -> Cola Int -> Bool
prop_colas1 x c =
  primero (inserta x vacia) == x &&
  resto (inserta x vacia) == vacia &&
  esVacia vacia &&
  not (esVacia (inserta x c))

prop_colas2 :: Int -> Cola Int -> Property
prop_colas2 x c =
  not (esVacia c) ==>
  primero (inserta x c) == primero c &&
  resto (inserta x c) == inserta x (resto c)

-- La comprobación es:
--   λ> quickCheck prop_colas1
--   +++ OK, passed 100 tests.
--   λ> quickCheck prop_colas2
--   +++ OK, passed 100 tests; 9 discarded.

```

7.4.2. En Python

```

# Se define la clase Cola con los siguientes métodos:
#   + inserta(x) añade x al final de la cola.
#   + primero() es el primero de la cola.
#   + resto() elimina el primero de la cola.
#   + esVacia() se verifica si la cola es vacía.
# Por ejemplo,
#   >>> c = Cola()
#   >>> c
#   -
#   >>> c.inserta(5)
#   >>> c.inserta(2)
#   >>> c.inserta(3)
#   >>> c.inserta(4)
#   >>> c

```

```

# 5 | 2 | 3 | 4
# >>> c.primer()
# 5
# >>> c.resto()
# >>> c
# 2 | 3 | 4
# >>> c.esVacia()
# False
# >>> c = Cola()
# >>> c.esVacia()
# True
#
# Además se definen las correspondientes funciones. Por ejemplo,
# >>> vacia()
# -
# >>> inserta(4, inserta(3, inserta(2, inserta(5, vacia()))))
# 5 | 2 | 3 | 4
# >>> primero(inserta(4, inserta(3, inserta(2, inserta(5, vacia()))))
# 5
# >>> resto(inserta(4, inserta(3, inserta(2, inserta(5, vacia()))))
# 2 | 3 | 4
# >>> esVacia(inserta(4, inserta(3, inserta(2, inserta(5, vacia()))))
# False
# >>> esVacia(vacia())
# True
#
# Finalmente, se define un generador aleatorio de colas y se comprueba
# que las colas cumplen las propiedades de su especificación.

__all__ = [
    'Cola',
    'vacia',
    'inserta',
    'primero',
    'resto',
    'esVacia',
    'colaAleatoria'
]

```

```
from collections import deque
```

```
from copy import deepcopy
from dataclasses import dataclass, field
from typing import Generic, TypeVar

from hypothesis import assume, given
from hypothesis import strategies as st

A = TypeVar('A')

# Clase de las colas mediante deque
# =====

@dataclass
class Cola(Generic[A]):
    _elementos: deque[A] = field(default_factory=deque)

    def __repr__(self) -> str:
        """
        Devuelve una cadena con los elementos de la cola separados por " | ".
        Si la cola está vacía, devuelve "-".
        """
        if self.esVacia():
            return '-'
        return ' | '.join(map(str, self._elementos))

    def inserta(self, x: A) -> None:
        """
        Inserta el elemento x en la cola.
        """
        self._elementos.append(x)

    def esVacia(self) -> bool:
        """
        Devuelve si la cola está vacía.
        """
        return not self._elementos

    def primero(self) -> A:
        """
        Devuelve el primer elemento de la cola.
```

```

        """
        return self._elementos[0]

    def resto(self) -> None:
        """
        Elimina el primer elemento de la cola.
        """
        self._elementos.popleft()

# Funciones del tipo de las deque
# =====

def vacia() -> Cola[A]:
    """
    Crea y devuelve una cola vacía de tipo A.
    """
    c: Cola[A] = Cola()
    return c

def inserta(x: A, c: Cola[A]) -> Cola[A]:
    """
    Inserta un elemento x en la cola c y devuelve una nueva cola con
    el elemento insertado.
    """
    _aux = deepcopy(c)
    _aux.inserta(x)
    return _aux

def esVacia(c: Cola[A]) -> bool:
    """
    Devuelve True si la cola está vacía, False si no lo está.
    """
    return c.esVacia()

def primero(c: Cola[A]) -> A:
    """
    Devuelve el primer elemento de la cola c.
    """
    return c.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

```

7.5. Transformaciones entre colas y listas

7.5.1. En Haskell

```

-----
-- Utilizando el [tipo abstracto de datos de las colas](https://bit.ly/3QWTsRL),
-- definir las funciones
--   listaAcola :: [a] -> Cola a
--   colaAlista :: Cola a -> [a]
-- tales que
-- + (listaAcola xs) es la cola formada por los elementos de xs.
--   Por ejemplo,
--   λ> listaAcola [3, 2, 5]
--   3 | 2 | 5
-- + (colaAlista c) es la lista formada por los elementos de la
--   cola c. Por ejemplo,
--   λ> colaAlista (inserta 5 (inserta 2 (inserta 3 vacia)))
--   [3, 2, 5]
--
-- Comprobar con QuickCheck que ambas funciones son inversa; es decir,
--   colaAlista (listaAcola xs) = xs
--   listaAcola (colaAlista c)  = c
-----

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module Transformaciones_colas_listas where

import TAD.Cola (Cola, vacia, inserta, esVacia, primero, resto)
import Test.QuickCheck

```

```

-- 1ª definición de listaAcola
-- =====

listaAcola :: [a] -> Cola a
listaAcola ys = aux (reverse ys)
  where aux []      = vacia
        aux (x:xs) = inserta x (aux xs)

-- 2ª definición de listaAcola
-- =====

listaAcola2 :: [a] -> Cola a
listaAcola2 = aux . reverse
  where aux [] = vacia
        aux (x:xs) = inserta x (aux xs)

-- 3ª definición de listaAcola
-- =====

listaAcola3 :: [a] -> Cola a
listaAcola3 = aux . reverse
  where aux = foldr inserta vacia

-- 4ª definición de listaAcola
-- =====

listaAcola4 :: [a] -> Cola a
listaAcola4 xs = foldr inserta vacia (reverse xs)

-- 5ª definición de listaAcola
-- =====

listaAcola5 :: [a] -> Cola a
listaAcola5 = foldr inserta vacia . reverse

-- Comprobación de equivalencia de las definiciones de listaAcola
-- =====

-- La propiedad es

```

```

prop_listaAcola :: [Int] -> Bool
prop_listaAcola xs =
  all (== listaAcola xs)
    [listaAcola2 xs,
     listaAcola3 xs,
     listaAcola4 xs,
     listaAcola5 xs]

-- La comprobación es
--   λ> quickCheck prop_listaAcola
--   +++ OK, passed 100 tests.

-- Definición de colaAlista
-- =====

colaAlista :: Cola a -> [a]
colaAlista c
  | esVacía c = []
  | otherwise = pc : colaAlista rc
  where pc = primero c
        rc = resto c

-- Comprobación de las propiedades
-- =====

-- La primera propiedad es
prop_1_listaAcola :: [Int] -> Bool
prop_1_listaAcola xs =
  colaAlista (listaAcola xs) == xs

-- La comprobación es
--   λ> quickCheck prop_1_listaAcola
--   +++ OK, passed 100 tests.

-- La segunda propiedad es
prop_2_listaAcola :: Cola Int -> Bool
prop_2_listaAcola c =
  listaAcola (colaAlista c) == c

-- La comprobación es

```

```
--    λ> quickCheck prop_2_listaAcola
--    +++ OK, passed 100 tests.
```

7.5.2. En Python

```
# -----
# Utilizando el [tipo abstracto de datos de las colas](https://bit.ly/3GTTToyK)
# definir las funciones
#   listaAcola : (list[A]) -> Cola[A]
#   colaAlista : (Cola[A]) -> list[A]
# tales que
# + listaAcola(xs) es la cola formada por los elementos de xs.
#   Por ejemplo,
#       >>> listaAcola([3, 2, 5])
#       3 | 2 | 5
# + colaAlista(c) es la lista formada por los elementos de la
#   cola c. Por ejemplo,
#       >>> ej = inserta(5, inserta(2, inserta(3, vacia())))
#       >>> colaAlista(ej)
#       [3, 2, 5]
#       >>> ej
#       3 | 2 | 5
#
# Comprobar con Hypothesis que ambas funciones son inversas; es decir,
#   colaAlista(listaAcola(xs)) == xs
#   listaAcola(colaAlista(c)) == c
# -----
```

```
from copy import deepcopy
from typing import TypeVar
```

```
from hypothesis import given
from hypothesis import strategies as st
```

```
from src.TAD.cola import (Cola, colaAleatoria, esVacia, inserta, primero,
                          resto, vacia)
```

```
A = TypeVar('A')
```

```
# 1ª definición de listaAcola
```

```

# =====

def listaAcola(ys: list[A]) -> Cola[A]:
    def aux(xs: list[A]) -> Cola[A]:
        if not xs:
            return vacia()
        return inserta(xs[0], aux(xs[1:]))

    return aux(list(reversed(ys)))

# 2ª solución de listaAcola
# =====

def listaAcola2(xs: list[A]) -> Cola[A]:
    p: Cola[A] = Cola()
    for x in xs:
        p.inserta(x)
    return p

# Comprobación de equivalencia de las definiciones de listaAcola
# =====

# La propiedad es
@given(st.lists(st.integers()))
def test_listaAcola(xs: list[int]) -> None:
    assert listaAcola(xs) == listaAcola2(xs)

# 1ª definición de colaAlista
# =====

def colaAlista(c: Cola[A]) -> list[A]:
    if esVacia(c):
        return []
    pc = primero(c)
    rc = resto(c)
    return [pc] + colaAlista(rc)

# 2ª definición de colaAlista
# =====

```

```

def colaAlista2Aux(c: Cola[A]) -> list[A]:
    if c.esVacia():
        return []
    pc = c.primer()
    c.resto()
    return [pc] + colaAlista2Aux(c)

def colaAlista2(c: Cola[A]) -> list[A]:
    c1 = deepcopy(c)
    return colaAlista2Aux(c1)

# 3ª definición de colaAlista
# =====

def colaAlista3Aux(c: Cola[A]) -> list[A]:
    r = []
    while not c.esVacia():
        r.append(c.primer())
        c.resto()
    return r

def colaAlista3(c: Cola[A]) -> list[A]:
    c1 = deepcopy(c)
    return colaAlista3Aux(c1)

# Comprobación de equivalencia de las definiciones de colaAlista
# =====

@given(p=colaAleatoria())
def test_colaAlista(p: Cola[int]) -> None:
    assert colaAlista(p) == colaAlista2(p)
    assert colaAlista(p) == colaAlista3(p)

# Comprobación de las propiedades
# =====

# La primera propiedad es
@given(st.lists(st.integers()))
def test_l1_listaAcola(xs: list[int]) -> None:
    assert colaAlista(listaAcola(xs)) == xs

```

```

# La segunda propiedad es
@given(c=colaAleatoria())
def test_2_listaCola(c: Cola[int]) -> None:
    assert listaCola(colaAlista(c)) == c

# La comprobación es
#      src> poetry run pytest -v transformaciones_colas_listas.py
#      test_listaCola PASSED
#      test_colaAlista PASSED
#      test_1_listaCola PASSED
#      test_2_listaCola PASSED

```

7.6. Último elemento de una cola

7.6.1. En Haskell

```

-- -----
-- Utilizando el [tipo abstracto de datos de las colas](https://bit.ly/3QWTsRL),
-- definir la función
--      ultimoCola :: Cola a -> a
-- tal que (ultimoCola c) es el último elemento de la cola c. Por
-- ejemplo:
--      ultimoCola (inserta 3 (inserta 5 (inserta 2 vacia))) == 3
--      ultimoCola (inserta 2 vacia)                        == 2
-- -----

```

```
{-# OPTIONS_GHC -fno-warn-unused-imports #-}
```

```
module UltimoCola where
```

```
import TAD.Cola (Cola, vacia, inserta, primero, resto, esVacia)
import Transformaciones_colas_listas (colaAlista)
import Test.QuickCheck
```

```
-- 1ª solución
-- =====
```

```
ultimoCola :: Cola a -> a
ultimoCola c
```

```

| esVacia c = error "cola vacia"
| esVacia rc = pc
| otherwise = ultimoCola rc
where pc = primero c
      rc = resto c

-- 2ª solución
-- =====

-- Se usarán la función colaAlista del ejercicio
-- "Transformaciones entre colas y listas" que se encuentra en
-- https://bit.ly/3Xv0oIt

ultimoCola2 :: Cola a -> a
ultimoCola2 c
| esVacia c = error "cola vacia"
| otherwise = last (colaAlista c)

-- Comprobación de equivalencia
-- =====

-- La propiedad es
prop_ultimoCola :: Cola Int -> Property
prop_ultimoCola c =
  not (esVacia c) ==> ultimoCola c == ultimoCola2 c

-- La comprobación es
-- λ> quickCheck prop_ultimoCola
-- +++ OK, passed 100 tests; 16 discarded.

```

7.6.2. En Python

```

# -----
# Utilizando el [tipo abstracto de datos de las colas](https://bit.ly/3QWTsRL),
# definir la función
#   ultimoCola : (Cola[A]) -> A
# tal que ultimoCola(c) es el último elemento de la cola c. Por
# ejemplo:
#   >>> ultimoCola(inserta(3, inserta(5, inserta(2, vacia()))))
#   3

```



```

#     >>> ultimoCola(inserta(2, vacia()))
#     2
# -----

from copy import deepcopy
from typing import TypeVar

from hypothesis import assume, given

from src.TAD.cola import (Cola, colaAleatoria, esVacia, inserta, primero,
                          resto, vacia)
from src.transformaciones_colas_listas import colaAlista

A = TypeVar('A')

# 1ª solución
# =====

def ultimoCola(c: Cola[A]) -> A:
    if esVacia(c):
        raise ValueError("cola vacia")
    pc = primero(c)
    rc = resto(c)
    if esVacia(rc):
        return pc
    return ultimoCola(rc)

# 2ª solución
# =====

def ultimoCola2Aux(c: Cola[A]) -> A:
    if c.esVacia():
        raise ValueError("cola vacia")
    pc = primero(c)
    c.resto()
    if c.esVacia():
        return pc
    return ultimoCola2(c)

def ultimoCola2(c: Cola[A]) -> A:

```

```
_c = deepcopy(c)
return ultimoCola2Aux(_c)

# 3ª solución
# =====

def ultimoCola3(c: Cola[A]) -> A:
    if esVacia(c):
        raise ValueError("cola vacia")
    while not esVacia(resto(c)):
        c = resto(c)
    return primero(c)

# 4ª solución
# =====

def ultimoCola4Aux(c: Cola[A]) -> A:
    if c.esVacia():
        raise ValueError("cola vacia")
    r = primero(c)
    while not c.esVacia():
        c.resto()
        if not c.esVacia():
            r = primero(c)
    return r

def ultimoCola4(c: Cola[A]) -> A:
    _c = deepcopy(c)
    return ultimoCola4Aux(_c)

# 5ª solución
# =====

# Se usarán la función colaAlista del ejercicio
# "Transformaciones entre colas y listas" que se encuentra en
# https://bit.ly/3Xv0oIt

def ultimoCola5(c: Cola[A]) -> A:
    if esVacia(c):
        raise ValueError("cola vacia")
```

```

    return colaAlista(c)[-1]

# Comprobación de equivalencia
# =====

# La propiedad es
@given(c=colaAleatoria())
def test_ultimoCola(c: Cola[int]) -> None:
    assume(not esVacia(c))
    r = ultimoCola(c)
    assert ultimoCola2(c) == r
    assert ultimoCola3(c) == r
    assert ultimoCola4(c) == r
    assert ultimoCola5(c) == r

# La comprobación es
#      src> poetry run pytest -q ultimoCola.py
#      1 passed in 0.25s

```

7.7. Longitud de una cola

7.7.1. En Haskell

```

-- -----
-- Utilizando el [tipo abstracto de datos de las colas](https://bit.ly/3QWTsRL),
-- definir la función
--   longitudCola :: Cola a -> Int
-- tal que (longitudCola c) es el número de elementos de la cola c. Por
-- ejemplo,
--   longitudCola (inserta 4 (inserta 2 (inserta 5 vacia))) == 3
-- -----

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module LongitudCola where

import TAD.Cola (Cola, vacia, inserta, resto, esVacia)
import Transformaciones_colas_listas (colaAlista)
import Test.QuickCheck

```

```

-- 1ª solución
-- =====

longitudCola1 :: Cola a -> Int
longitudCola1 c
  | esVacia c = 0
  | otherwise = 1 + longitudCola1 rc
  where rc = resto c

-- 2ª solución
-- =====

longitudCola2 :: Cola a -> Int
longitudCola2 = length . colaAlista

-- La función colaAlista está definida en el ejercicio
-- "Transformaciones entre colas y listas" que se encuentra en
-- https://bit.ly/3Xv0oIt

-- Comprobación de equivalencia
-- =====

-- La propiedad es
prop_longitudCola :: Cola Int -> Bool
prop_longitudCola c =
  longitudCola1 c == longitudCola2 c

-- La comprobación es
--   λ> quickCheck prop_longitudCola
--   +++ OK, passed 100 tests.

```

7.7.2. En Python

```

# -----
# Utilizando el [tipo abstracto de datos de las colas](https://bit.ly/3QWTsRL),
# definir la función
#   longitudCola : (Cola[A]) -> int
# tal que longitudCola(c) es el número de elementos de la cola c. Por
# ejemplo,
#   >>> longitudCola(inserta(4, inserta(2, inserta(5, vacia()))))

```

```
# 3
# -----

from copy import deepcopy
from typing import TypeVar

from hypothesis import given

from src.TAD.cola import Cola, colaAleatoria, esVacia, inserta, resto, vacia
from src.transformaciones_colas_listas import colaAlista

A = TypeVar('A')

# 1ª solución
# =====

def longitudCola1(c: Cola[A]) -> int:
    if esVacia(c):
        return 0
    return 1 + longitudCola1(resto(c))

# 2ª solución
# =====

def longitudCola2(c: Cola[A]) -> int:
    return len(colaAlista(c))

# 3ª solución
# =====

def longitudCola3Aux(c: Cola[A]) -> int:
    if c.esVacia():
        return 0
    c.resto()
    return 1 + longitudCola3Aux(c)

def longitudCola3(c: Cola[A]) -> int:
    _c = deepcopy(c)
    return longitudCola3Aux(_c)
```

```

# 4ª solución
# =====

def longitudCola4Aux(c: Cola[A]) -> int:
    r = 0
    while not esVacia(c):
        r = r + 1
        c = resto(c)
    return r

def longitudCola4(c: Cola[A]) -> int:
    _c = deepcopy(c)
    return longitudCola4Aux(_c)

# 5ª solución
# =====

def longitudCola5Aux(c: Cola[A]) -> int:
    r = 0
    while not c.esVacia():
        r = r + 1
        c.resto()
    return r

def longitudCola5(c: Cola[A]) -> int:
    _c = deepcopy(c)
    return longitudCola5Aux(_c)

# Comprobación de equivalencia
# =====

# La propiedad es
@given(c=colaAleatoria())
def test_longitudCola_(c: Cola[int]) -> None:
    r = longitudCola1(c)
    assert longitudCola2(c) == r
    assert longitudCola3(c) == r
    assert longitudCola4(c) == r
    assert longitudCola5(c) == r

```

```
# La comprobación es
#   src> poetry run pytest -q longitudCola.py
#   1 passed in 0.28s
```

7.8. Todos los elementos de la cola verifican una propiedad

7.8.1. En Haskell

```
-- -----
-- Utilizando el [tipo abstracto de datos de las colas](https://bit.ly/3QWTsRL),
-- definir la función
--   todosVerifican :: (a -> Bool) -> Cola a -> Bool
-- tal que (todosVerifican p c) se verifica si todos los elementos de la
-- cola c cumplen la propiedad p. Por ejemplo,
--   todosVerifican (>0) (inserta 3 (inserta 2 vacia))    == True
--   todosVerifican (>0) (inserta 3 (inserta (-2) vacia)) == False
-- -----

{-# OPTIONS_GHC -fno-warn-unused-imports #-}
```

module TodosVerifican where

```
import TAD.Cola (Cola, vacia, inserta, primero, resto, esVacia)
import Transformaciones_colas_listas (colaAlista, listaAcola)
import Test.QuickCheck.HigherOrder

-- 1ª solución
-- =====

todosVerifican1 :: (a -> Bool) -> Cola a -> Bool
todosVerifican1 p c
  | esVacia c = True
  | otherwise = p pc && todosVerifican1 p rc
  where pc = primero c
        rc = resto c

-- 2ª solución
-- =====
```

```

todosVerifican2 :: (a -> Bool) -> Cola a -> Bool
todosVerifican2 p c =
    all p (colaAlista c)

-- La función colaAlista está definida en el ejercicio
-- "Transformaciones entre colas y listas" que se encuentra en
-- https://bit.ly/3Xv0oIt

-- Comprobación de equivalencia
-- =====

-- La propiedad es
prop_todosVerifican :: (Int -> Bool) -> [Int] -> Bool
prop_todosVerifican p xs =
    todosVerifican1 p c == todosVerifican2 p c
    where c = listaAcola xs

-- La comprobación es
--    λ> quickCheck' prop_todosVerifican
--    +++ OK, passed 100 tests.

```

7.8.2. En Python

```

# -----
# Utilizando el [tipo abstracto de datos de las colas](https://bit.ly/3QWTsRL),
# definir la función
#     todosVerifican : (Callable[[A], bool], Cola[A]) -> bool
# tal que todosVerifican(p, c) se verifica si todos los elementos de la
# cola c cumplen la propiedad p. Por ejemplo,
#     >>> todosVerifican(lambda x: x > 0, inserta(3, inserta(2, vacia())))
#     True
#     >>> todosVerifican(lambda x: x > 0, inserta(3, inserta(-2, vacia())))
#     False
# -----

from copy import deepcopy
from typing import Callable, TypeVar

from hypothesis import given

```



```
from src.TAD.cola import (Cola, colaAleatoria, esVacia, inserta, primero,
                          resto, vacia)
from src.transformaciones_colas_listas import colaAlista

A = TypeVar('A')

# 1ª solución
# =====

def todosVerifican1(p: Callable[[A], bool], c: Cola[A]) -> bool:
    if esVacia(c):
        return True
    pc = primero(c)
    rc = resto(c)
    return p(pc) and todosVerifican1(p, rc)

# 2ª solución
# =====

def todosVerifican2(p: Callable[[A], bool], c: Cola[A]) -> bool:
    return all(p(x) for x in colaAlista(c))

# La función colaAlista está definida en el ejercicio
# "Transformaciones entre colas y listas" que se encuentra en
# https://bit.ly/3Xv0oIt

# 3ª solución
# =====

def todosVerifican3Aux(p: Callable[[A], bool], c: Cola[A]) -> bool:
    if c.esVacia():
        return True
    pc = c.primer()
    c.resto()
    return p(pc) and todosVerifican3Aux(p, c)

def todosVerifican3(p: Callable[[A], bool], c: Cola[A]) -> bool:
    _c = deepcopy(c)
    return todosVerifican3Aux(p, _c)
```

4ª solución

=====

```
def todosVerifican4Aux(p: Callable[[A], bool], c: Cola[A]) -> bool:
    if c.esVacia():
        return True
    pc = c.primer()
    c.resto()
    return p(pc) and todosVerifican4Aux(p, c)

def todosVerifican4(p: Callable[[A], bool], c: Cola[A]) -> bool:
    _c = deepcopy(c)
    return todosVerifican4Aux(p, _c)
```

5ª solución

=====

```
def todosVerifican5Aux(p: Callable[[A], bool], c: Cola[A]) -> bool:
    while not c.esVacia():
        if not p(c.primer()):
            return False
        c.resto()
    return True

def todosVerifican5(p: Callable[[A], bool], c: Cola[A]) -> bool:
    _c = deepcopy(c)
    return todosVerifican5Aux(p, _c)
```

Comprobación de equivalencia

=====

La propiedad es

@given(c=colaAleatoria())

```
def test_filtraPila(c: Cola[int]) -> None:
    r = todosVerifican1(lambda x: x > 0, c)
    assert todosVerifican2(lambda x: x > 0, c) == r
    assert todosVerifican3(lambda x: x > 0, c) == r
    assert todosVerifican4(lambda x: x > 0, c) == r
    assert todosVerifican5(lambda x: x > 0, c) == r
```

```
# La comprobación es
#   src> poetry run pytest -q todosVerifican.py
#   1 passed in 0.25s
```

7.9. Algún elemento de la verifica una propiedad

7.9.1. En Haskell

```
-- -----
-- Utilizando el [tipo abstracto de datos de las colas](https://bit.ly/3QWTsRL),
-- definir la función
--   algunoVerifica :: (a -> Bool) -> Cola a -> Bool
-- tal que (algunoVerifica p c) se verifica si alguno de los elementos de la
-- cola c cumplen la propiedad p. Por ejemplo,
--   algunoVerifica (< 0) (inserta 3 (inserta (-2) vacia)) == True
--   algunoVerifica (< 0) (inserta 3 (inserta 2 vacia))    == False
-- -----

{-# OPTIONS_GHC -fno-warn-unused-imports #-}
```

module AlgunoVerifica **where**

```
import TAD.Cola (Cola, vacia, inserta, primero, resto, esVacia)
import Transformaciones_colas_listas (colaAlista, listaAcola)
import Test.QuickCheck.HigherOrder

-- 1ª solución
-- =====

algunoVerifica1 :: (a -> Bool) -> Cola a -> Bool
algunoVerifica1 p c
  | esVacia c = False
  | otherwise = p pc || algunoVerifica1 p rc
  where pc = primero c
        rc = resto c

-- 2ª solución
```

```

-- =====

algunoVerifica2 :: (a -> Bool) -> Cola a -> Bool
algunoVerifica2 p c =
    any p (colaAlista c)

-- La función colaAlista está definida en el ejercicio
-- "Transformaciones entre colas y listas" que se encuentra en
-- https://bit.ly/3Xv0oIt

-- Comprobación de equivalencia
-- =====

-- La propiedad es
prop_algunoVerifica :: (Int -> Bool) -> [Int] -> Bool
prop_algunoVerifica p xs =
    algunoVerifica1 p c == algunoVerifica2 p c
    where c = listaAcola xs

-- La comprobación es
--    λ> quickCheck' prop_algunoVerifica
--    +++ OK, passed 100 tests.

```

7.9.2. En Python

```

# -----
# Utilizando el [tipo abstracto de datos de las colas](https://bit.ly/3QWTsRL),
# definir la función
#     algunoVerifica : (Callable[[A], bool], Cola[A]) -> bool
# tal que algunoVerifica(p, c) se verifica si alguno de los elementos de la
# cola c cumplen la propiedad p. Por ejemplo,
#     >>> algunoVerifica(lambda x: x > 0, inserta(-3, inserta(2, vacia())))
#     True
#     >>> algunoVerifica(lambda x: x > 0, inserta(-3, inserta(-2, vacia())))
#     False
# -----

from copy import deepcopy
from typing import Callable, TypeVar

```

```
from hypothesis import given

from src.TAD.cola import (Cola, colaAleatoria, esVacia, inserta, primero,
                          resto, vacia)
from src.transformaciones_colas_listas import colaAlista

A = TypeVar('A')

# 1ª solución
# =====

def algunoVerifica1(p: Callable[[A], bool], c: Cola[A]) -> bool:
    if esVacia(c):
        return False
    pc = primero(c)
    rc = resto(c)
    return p(pc) or algunoVerifica1(p, rc)

# 2ª solución
# =====

def algunoVerifica2(p: Callable[[A], bool], c: Cola[A]) -> bool:
    return any(p(x) for x in colaAlista(c))

# La función colaAlista está definida en el ejercicio
# "Transformaciones entre colas y listas" que se encuentra en
# https://bit.ly/3Xv0oIt

# 3ª solución
# =====

def algunoVerifica3Aux(p: Callable[[A], bool], c: Cola[A]) -> bool:
    if c.esVacia():
        return False
    pc = c.primer()
    c.resto()
    return p(pc) or algunoVerifica3Aux(p, c)

def algunoVerifica3(p: Callable[[A], bool], c: Cola[A]) -> bool:
    _c = deepcopy(c)
```

```

    return algunoVerifica3Aux(p, _c)

# 4ª solución
# =====

def algunoVerifica4Aux(p: Callable[[A], bool], c: Cola[A]) -> bool:
    if c.esVacia():
        return False
    pc = c.primer()
    c.resto()
    return p(pc) or algunoVerifica4Aux(p, c)

def algunoVerifica4(p: Callable[[A], bool], c: Cola[A]) -> bool:
    _c = deepcopy(c)
    return algunoVerifica4Aux(p, _c)

# 5ª solución
# =====

def algunoVerifica5Aux(p: Callable[[A], bool], c: Cola[A]) -> bool:
    while not c.esVacia():
        if p(c.primer()):
            return True
        c.resto()
    return False

def algunoVerifica5(p: Callable[[A], bool], c: Cola[A]) -> bool:
    _c = deepcopy(c)
    return algunoVerifica5Aux(p, _c)

# Comprobación de equivalencia
# =====

# La propiedad es
@given(c=colaAleatoria())
def test_algunoVerifica(c: Cola[int]) -> None:
    r = algunoVerifica1(lambda x: x > 0, c)
    assert algunoVerifica2(lambda x: x > 0, c) == r
    assert algunoVerifica3(lambda x: x > 0, c) == r
    assert algunoVerifica4(lambda x: x > 0, c) == r

```

```

    assert algunoVerifica5(lambda x: x > 0, c) == r

# La comprobación es
#     src> poetry run pytest -q algunoVerifica.py
#     1 passed in 0.31s

```

7.10. Extensión de colas

7.10.1. En Haskell

```

-----
-- Utilizando el [tipo abstracto de datos de las colas](https://bit.ly/3QWTsRL),
-- definir la función
--     extiendeCola :: Cola a -> Cola a -> Cola a
-- tal que (extiendeCola c1 c2) es la cola que resulta de poner los
-- elementos de la cola c2 a continuación de los de la cola de c1. Por
-- ejemplo,
--     λ> ej1 = inserta 3 (inserta 2 vacia)
--     λ> ej2 = inserta 5 (inserta 3 (inserta 4 vacia))
--     λ> ej1
--     2 | 3
--     λ> ej2
--     4 | 3 | 5
--     λ> extiendeCola ej1 ej2
--     2 | 3 | 4 | 3 | 5
--     λ> extiendeCola ej2 ej1
--     4 | 3 | 5 | 2 | 3
-----

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module ExtiendeCola where

import TAD.Cola (Cola, vacia, inserta, primero, resto, esVacia)
import Transformaciones_colas_listas (colaAlista, listaAcola)
import Test.QuickCheck

-- 1ª solución
-- =====

```

```

extiendeCola :: Cola a -> Cola a -> Cola a
extiendeCola c1 c2
  | esVacia c2 = c1
  | otherwise = extiendeCola (inserta pc2 c1) rq2
  where pc2 = primero c2
        rq2 = resto c2

-- 2ª solución
-- =====

extiendeCola2 :: Cola a -> Cola a -> Cola a
extiendeCola2 c1 c2 =
  listaAcola (colaAlista c1 ++ colaAlista c2)

-- Las funciones colaAlista y listaAcola están definidas en el ejercicio
-- "Transformaciones entre colas y listas" que se encuentra en
-- https://bit.ly/3Xv0oIt

-- Comprobación de equivalencia
-- =====

-- La propiedad es
prop_extiendeCola :: Cola Int -> Cola Int -> Bool
prop_extiendeCola p c =
  extiendeCola p c == extiendeCola2 p c

-- La comprobación es
--   λ> quickCheck prop_extiendeCola
--   +++ OK, passed 100 tests.

```

7.10.2. En Python

```

# -----
# Utilizando el [tipo abstracto de datos de las colas](https://bit.ly/3QWTsRL),
# definir la función
#   extiendeCola : (Cola[A], Cola[A]) -> Cola[A]
# tal que extiendeCola(c1, c2) es la cola que resulta de poner los
# elementos de la cola c2 a continuación de los de la cola de c1. Por
# ejemplo,
#   >>> ej1 = inserta(3, inserta(2, vacia()))

```



```

#     >>> ej2 = inserta(5, inserta(3, inserta(4, vacia())))
#     >>> ej1
#     2 | 3
#     >>> ej2
#     4 | 3 | 5
#     >>> extiendeCola(ej1, ej2)
#     2 | 3 | 4 | 3 | 5
#     >>> extiendeCola(ej2, ej1)
#     4 | 3 | 5 | 2 | 3
# -----

from copy import deepcopy
from typing import TypeVar

from hypothesis import given

from src.TAD.cola import (Cola, colaAleatoria, esVacia, inserta, primero,
                           resto, vacia)
from src.transformaciones_colas_listas import colaAlista, listaAcola

A = TypeVar('A')

# 1ª solución
# =====

def extiendeCola(c1: Cola[A], c2: Cola[A]) -> Cola[A]:
    if esVacia(c2):
        return c1
    pc2 = primero(c2)
    rc2 = resto(c2)
    return extiendeCola(inserta(pc2, c1), rc2)

# 2ª solución
# =====

def extiendeCola2(c1: Cola[A], c2: Cola[A]) -> Cola[A]:
    return listaAcola(colaAlista(c1) + colaAlista(c2))

# Las funciones colaAlista y listaAcola están definidas en el ejercicio
# "Transformaciones entre colas y listas" que se encuentra en

```

```
# https://bit.ly/3Xv0oIt
```

```
# 3ª solución
```

```
# =====
```

```
def extiendeCola3Aux(c1: Cola[A], c2: Cola[A]) -> Cola[A]:
    if c2.esVacia():
        return c1
    pc2 = c2.primer()
    c2.resto()
    return extiendeCola(inserta(pc2, c1), c2)

def extiendeCola3(c1: Cola[A], c2: Cola[A]) -> Cola[A]:
    _c2 = deepcopy(c2)
    return extiendeCola3Aux(c1, _c2)
```

```
# 4ª solución
```

```
# =====
```

```
def extiendeCola4Aux(c1: Cola[A], c2: Cola[A]) -> Cola[A]:
    r = c1
    while not esVacia(c2):
        r = inserta(primer(c2), r)
        c2 = resto(c2)
    return r

def extiendeCola4(c1: Cola[A], c2: Cola[A]) -> Cola[A]:
    _c2 = deepcopy(c2)
    return extiendeCola4Aux(c1, _c2)
```

```
# 5ª solución
```

```
# =====
```

```
def extiendeCola5Aux(c1: Cola[A], c2: Cola[A]) -> Cola[A]:
    r = c1
    while not c2.esVacia():
        r.inserta(primer(c2))
        c2.resto()
    return r
```

```

def extiendeCola5(c1: Cola[A], c2: Cola[A]) -> Cola[A]:
    _c1 = deepcopy(c1)
    _c2 = deepcopy(c2)
    return extiendeCola5Aux(_c1, _c2)

# Comprobación de equivalencia
# =====

# La propiedad es
@given(c1=colaAleatoria(), c2=colaAleatoria())
def test_extiendeCola(c1: Cola[int], c2: Cola[int]) -> None:
    r = extiendeCola(c1, c2)
    assert extiendeCola2(c1, c2) == r
    assert extiendeCola3(c1, c2) == r
    assert extiendeCola4(c1, c2) == r

# La comprobación es
#   src> poetry run pytest -q extiendeCola.py
#   1 passed in 0.32s

```

7.11. Intercalado de dos colas

7.11.1. En Haskell

```

-----
-- Utilizando el [tipo abstracto de datos de las colas](https://bit.ly/3QWTsRL),
-- definir la función
--   intercalaColas :: Cola a -> Cola a -> Cola a
-- tal que (intercalaColas c1 c2) es la cola formada por los elementos de
-- c1 y c2 colocados en una cola, de forma alternativa, empezando por
-- los elementos de c1. Por ejemplo,
--   λ> ej1 = inserta 3 (inserta 5 vacia)
--   λ> ej2 = inserta 0 (inserta 7 (inserta 4 (inserta 9 vacia)))
--   λ> intercalaColas ej1 ej2
--   5 | 9 | 3 | 4 | 7 | 0
--   λ> intercalaColas ej2 ej1
--   9 | 5 | 4 | 3 | 7 | 0
-----

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

```

```
module IntercalaColas where
```

```
import TAD.Cola (Cola, vacia, inserta, primero, resto, esVacia)
import Transformaciones_colas_listas (colaAlista, listaAcola)
import ExtiendeCola (extiendeCola)
import Test.QuickCheck
```

```
-- 1ª solución
-- =====
```

```
intercalaColas :: Cola a -> Cola a -> Cola a
intercalaColas c1 c2
  | esVacia c1 = c2
  | esVacia c2 = c1
  | otherwise  = extiendeCola (inserta pc2 (inserta pc1 vacia))
                        (intercalaColas rc1 rc2)

where pc1 = primero c1
      rc1 = resto c1
      pc2 = primero c2
      rc2 = resto c2
```

```
-- La función extiendeCola está definida en el ejercicio
-- "TAD de las colas: Extensión de colas" que se encuentra en
-- https://bit.ly/3XIJJ4m
```

```
-- 2ª solución
-- =====
```

```
intercalaColas2 :: Cola a -> Cola a -> Cola a
intercalaColas2 c1 c2 = aux c1 c2 vacia
  where
    aux d1 d2 c
      | esVacia d1 = extiendeCola c d2
      | esVacia d2 = extiendeCola c d1
      | otherwise  = aux rd1 rd2 (inserta pd2 (inserta pd1 c))
    where pd1 = primero d1
          rd1 = resto d1
          pd2 = primero d2
          rd2 = resto d2
```

```

-- 3ª solución
-- =====

intercalaColas3 :: Cola a -> Cola a -> Cola a
intercalaColas3 c1 c2 =
    listaAcola (intercalaListas (colaAlista c1) (colaAlista c2))

-- (intercalaListas xs ys) es la lista obtenida intercalando los
-- elementos de xs e ys. Por ejemplo,
intercalaListas :: [a] -> [a] -> [a]
intercalaListas []      ys      = ys
intercalaListas xs      []      = xs
intercalaListas (x:xs) (y:ys) = x : y : intercalaListas xs ys

-- Comprobación de equivalencia
-- =====

-- La propiedad es
prop_intercalaColas :: Cola Int -> Cola Int -> Bool
prop_intercalaColas c1 c2 =
    all (== intercalaColas c1 c2)
        [intercalaColas2 c1 c2,
         intercalaColas2 c1 c2]

-- La comprobación es
--    λ> quickCheck prop_intercalaColas
--    +++ OK, passed 100 tests.

```

7.11.2. En Python

```

# -----
# Utilizando el [tipo abstracto de datos de las colas](https://bit.ly/3QWTsRL),
# definir la función
#     intercalaColas : (Cola[A], Cola[A]) -> Cola[A]
# tal que (intercalaColas c1 c2) es la cola formada por los elementos de
# c1 y c2 colocados en una cola, de forma alternativa, empezando por
# los elementos de c1. Por ejemplo,
#     >>> ej1 = inserta(3, inserta(5, vacia()))
#     >>> ej2 = inserta(0, inserta(7, inserta(4, inserta(9, vacia()))))

```

```

#     >>> intercalaColas(ej1, ej2)
#     5 | 9 | 3 | 4 | 7 | 0
#     >>> intercalaColas(ej2, ej1)
#     9 | 5 | 4 | 3 | 7 | 0
# -----

from copy import deepcopy
from typing import TypeVar

from hypothesis import given

from src.extiendeCola import extiendeCola
from src.TAD.cola import (Cola, colaAleatoria, esVacia, inserta, primero,
                           resto, vacia)
from src.transformaciones_colas_listas import colaAlista, listaAcola

A = TypeVar('A')

# 1ª solución
# =====

def intercalaColas(c1: Cola[A], c2: Cola[A]) -> Cola[A]:
    if esVacia(c1):
        return c2
    if esVacia(c2):
        return c1
    pc1 = primero(c1)
    rc1 = resto(c1)
    pc2 = primero(c2)
    rc2 = resto(c2)
    return extiendeCola(inserta(pc2, inserta(pc1, vacia()))),
                       intercalaColas(rc1, rc2))

# La función extiendeCola está definida en el ejercicio
# "TAD de las colas: Extensión de colas" que se encuentra en
# https://bit.ly/3XIJJ4m

# 2ª solución
# =====

```

```

def intercalaColas2(c1: Cola[A], c2: Cola[A]) -> Cola[A]:
    def aux(d1: Cola[A], d2: Cola[A], d3: Cola[A]) -> Cola[A]:
        if esVacia(d1):
            return extiendeCola(d3, d2)
        if esVacia(d2):
            return extiendeCola(d3, d1)
        pd1 = primero(d1)
        rd1 = resto(d1)
        pd2 = primero(d2)
        rd2 = resto(d2)
        return aux(rd1, rd2, inserta(pd2, inserta(pd1, d3)))

    return aux(c1, c2, vacia())

# 3ª solución
# =====

# intercalaListas(xs, ys) es la lista obtenida intercalando los
# elementos de xs e ys. Por ejemplo,
def intercalaListas(xs: list[A], ys: list[A]) -> list[A]:
    if not xs:
        return ys
    if not ys:
        return xs
    return [xs[0], ys[0]] + intercalaListas(xs[1:], ys[1:])

def intercalaColas3(c1: Cola[A], c2: Cola[A]) -> Cola[A]:
    return listaAcola(intercalaListas(colaAlista(c1), colaAlista(c2)))

# 4ª solución
# =====

def intercalaColas4Aux(c1: Cola[A], c2: Cola[A]) -> Cola[A]:
    if c1.esVacia():
        return c2
    if c2.esVacia():
        return c1
    pc1 = c1.primer()
    c1.resto()
    pc2 = c2.primer()

```

```

    c2.resto()
    return extiendeCola(inserta(pc2, inserta(pc1, vacia()))),
                        intercalaColas4Aux(c1, c2))

def intercalaColas4(c1: Cola[A], c2: Cola[A]) -> Cola[A]:
    _c1 = deepcopy(c1)
    _c2 = deepcopy(c2)
    return intercalaColas4Aux(_c1, _c2)

# 5ª solución
# =====

def intercalaColas5Aux(c1: Cola[A], c2: Cola[A]) -> Cola[A]:
    r: Cola[A] = vacia()
    while not esVacia(c1) and not esVacia(c2):
        pc1 = primero(c1)
        c1.resto()
        pc2 = primero(c2)
        c2.resto()
        r = inserta(pc2, inserta(pc1, r))
    if esVacia(c1):
        return extiendeCola(r, c2)
    return extiendeCola(r, c1)

def intercalaColas5(c1: Cola[A], c2: Cola[A]) -> Cola[A]:
    _c1 = deepcopy(c1)
    _c2 = deepcopy(c2)
    return intercalaColas5Aux(_c1, _c2)

# Comprobación de equivalencia
# =====

# La propiedad es
@given(c1=colaAleatoria(), c2=colaAleatoria())
def test_intercalaCola(c1: Cola[int], c2: Cola[int]) -> None:
    r = intercalaColas(c1, c2)
    assert intercalaColas2(c1, c2) == r
    assert intercalaColas3(c1, c2) == r
    assert intercalaColas4(c1, c2) == r
    assert intercalaColas5(c1, c2) == r

```



```
# La comprobación es
#   src> poetry run pytest -q intercalaColas.py
#   1 passed in 0.47s
```

7.12. Agrupación de colas

7.12.1. En Haskell

```
-- -----
-- Utilizando el [tipo abstracto de datos de las colas](https://bit.ly/3QWTsRL),
-- definir la función
--   agrupaColas :: [Cola a] -> Cola a
-- tal que (agrupaColas [c1,c2,c3,...,cn]) es la cola formada mezclando
-- las colas de la lista como sigue: mezcla c1 con c2, el resultado con
-- c3, el resultado con c4, y así sucesivamente. Por ejemplo,
--   λ> ej1 = inserta 2 (inserta 5 vacia)
--   λ> ej2 = inserta 3 (inserta 7 (inserta 4 vacia))
--   λ> ej3 = inserta 9 (inserta 0 (inserta 1 (inserta 6 vacia)))
--   λ> agrupaColas []
--   -
--   λ> agrupaColas [ej1]
--   5 | 2
--   λ> agrupaColas [ej1, ej2]
--   5 | 4 | 2 | 7 | 3
--   λ> agrupaColas [ej1, ej2, ej3]
--   5 | 6 | 4 | 1 | 2 | 0 | 7 | 9 | 3
-- -----
```

```
{-# OPTIONS_GHC -fno-warn-unused-imports #-}
```

```
module AgrupaColas where
```

```
import TAD.Cola (Cola, vacia, inserta)
import IntercalaColas (intercalaColas)
import Test.QuickCheck
```

```
-- 1ª solución
-- =====
```

```

agrupaColas1 :: [Cola a] -> Cola a
agrupaColas1 []           = vacia
agrupaColas1 [c]          = c
agrupaColas1 (c1:c2:colas) = agrupaColas1 ((intercalaColas c1 c2) : colas)

-- La función intercalaColas está definida en el ejercicio
-- "TAD de las colas: Intercalado de dos colas" que se encuentra en
-- https://bit.ly/3XYyjsM

-- 2ª solución
-- =====

agrupaColas2 :: [Cola a] -> Cola a
agrupaColas2 = foldl intercalaColas vacia

-- Comprobación de equivalencia
-- =====

-- La propiedad es
prop_agrupaColas :: [Cola Int] -> Bool
prop_agrupaColas cs =
  agrupaColas1 cs == agrupaColas2 cs

-- La comprobación es
--   λ> quickCheckWith (stdArgs {maxSize=30}) prop_agrupaColas
--   +++ OK, passed 100 tests.

```

7.12.2. En Python

```

# -----
# Utilizando el [tipo abstracto de datos de las colas](https://bit.ly/3QWTsRL),
# definir la función
#   agrupaColas : (list[Cola[A]]) -> Cola[A]
# tal que (agrupaColas [c1,c2,c3,...,cn]) es la cola formada mezclando
# las colas de la lista como sigue: mezcla c1 con c2, el resultado con
# c3, el resultado con c4, y así sucesivamente. Por ejemplo,
#   >>> ej1 = inserta(2, inserta(5, vacia()))
#   >>> ej2 = inserta(3, inserta(7, inserta(4, vacia())))
#   >>> ej3 = inserta(9, inserta(0, inserta(1, inserta(6, vacia()))))
#   >>> agrupaColas([])

```

```
# -
# >>> agrupaColas([ej1])
# 5 | 2
# >>> agrupaColas([ej1, ej2])
# 5 | 4 | 2 | 7 | 3
# >>> agrupaColas([ej1, ej2, ej3])
# 5 | 6 | 4 | 1 | 2 | 0 | 7 | 9 | 3
# -----
```

```
from functools import reduce
from typing import TypeVar
```

```
from hypothesis import given
from hypothesis import strategies as st
```

```
from src.intercalaColas import intercalaColas
from src.TAD.cola import Cola, colaAleatoria, inserta, vacia
```

```
A = TypeVar('A')
```

```
# 1ª solución
# =====
```

```
def agrupaColas1(cs: list[Cola[A]]) -> Cola[A]:
    if not cs:
        return vacia()
    if len(cs) == 1:
        return cs[0]
    return agrupaColas1([intercalaColas(cs[0], cs[1])] + cs[2:])
```

```
# La función intercalaColas está definida en el ejercicio
# "TAD de las colas: Intercalado de dos colas" que se encuentra en
# https://bit.ly/3XYyjsM
```

```
# 2ª solución
# =====
```

```
def agrupaColas2(cs: list[Cola[A]]) -> Cola[A]:
    return reduce(intercalaColas, cs, vacia())
```

```
# Comprobación de equivalencia
# =====

# La propiedad es
@given(st.lists(colaAleatoria(), max_size=4))
def test_agrupaCola(cs: list[Cola[int]]) -> None:
    assert agrupaColas1(cs) == agrupaColas2(cs)

# La comprobación es
#   src> poetry run pytest -q agrupaColas.py
#   1 passed in 0.50s
```

7.13. Pertenencia a una cola

7.13.1. En Haskell

```
-- -----
-- Utilizando el [tipo abstracto de datos de las colas](https://bit.ly/3QWTsRL),
-- definir la función
--   perteneceCola :: Eq a => a -> Cola a -> Bool
-- tal que (perteneceCola x c) se verifica si x es un elemento de la
-- cola c. Por ejemplo,
--   perteneceCola 2 (inserta 5 (inserta 2 (inserta 3 vacia))) == True
--   perteneceCola 4 (inserta 5 (inserta 2 (inserta 3 vacia))) == False
-- -----

{-# OPTIONS_GHC -fno-warn-unused-imports #-}
```

module PerteneceCola where

```
import TAD.Cola (Cola, vacia, inserta, primero, resto, esVacia)
import Transformaciones_colas_listas (colaAlista, listaAcola)
import Test.QuickCheck

-- 1ª solución
-- =====

perteneceCola :: Eq a => a -> Cola a -> Bool
perteneceCola x c
  | esVacia c = False
```

```

    | otherwise = x == primero(c) || perteneceCola x (resto c)

-- 2ª solución
-- =====

perteneceCola2 :: Eq a => a -> Cola a -> Bool
perteneceCola2 x c =
    x `elem` colaAlista c

-- La función colaAlista está definida en el ejercicio
-- "Transformaciones entre colas y listas" que se encuentra en
-- https://bit.ly/3Xv0oIt

-- Comprobación de equivalencia
-- =====

-- La propiedad es
prop_perteneceCola :: Int -> Cola Int -> Bool
prop_perteneceCola x p =
    perteneceCola x p == perteneceCola2 x p

-- La comprobación es
--    λ> quickCheck prop_perteneceCola
--    +++ OK, passed 100 tests.

```

7.13.2. En Python

```

# -----
# Utilizando el [tipo abstracto de datos de las colas](https://bit.ly/3QWTsRL),
# definir la función
#     perteneceCola : (A, Cola[A]) -> bool
# tal que perteneceCola(x, c) se verifica si x es un elemento de la
# cola p. Por ejemplo,
#     >>> perteneceCola(2, inserta(5, inserta(2, inserta(3, vacia()))))
#     True
#     >>> perteneceCola(4, inserta(5, inserta(2, inserta(3, vacia()))))
#     False
# -----

from copy import deepcopy

```

```
from typing import TypeVar

from hypothesis import given
from hypothesis import strategies as st

from src.TAD.cola import (Cola, colaAleatoria, esVacia, inserta, primero,
                          resto, vacia)
from src.transformaciones_colas_listas import colaAlista

A = TypeVar('A')

# 1ª solución
# =====

def perteneceCola(x: A, c: Cola[A]) -> bool:
    if esVacia(c):
        return False
    return x == primero(c) or perteneceCola(x, resto(c))

# 2ª solución
# =====

def perteneceCola2(x: A, c: Cola[A]) -> bool:
    return x in colaAlista(c)

# Las función colaAlista está definida en el ejercicio
# "Transformaciones entre colas y listas" que se encuentra en
# https://bit.ly/3Xv0oIt

# 3ª solución
# =====

def perteneceCola3Aux(x: A, c: Cola[A]) -> bool:
    if c.esVacia():
        return False
    pc = c.primer()
    c.resto()
    return x == pc or perteneceCola3Aux(x, c)

def perteneceCola3(x: A, c: Cola[A]) -> bool:
```

```

    c1 = deepcopy(c)
    return perteneceCola3Aux(x, c1)

# 4ª solución
# =====

def perteneceCola4Aux(x: A, c: Cola[A]) -> bool:
    while not c.esVacia():
        pc = c.primer()
        c.resto()
        if x == pc:
            return True
    return False

def perteneceCola4(x: A, c: Cola[A]) -> bool:
    c1 = deepcopy(c)
    return perteneceCola4Aux(x, c1)

# Comprobación de equivalencia de las definiciones
# =====

# La propiedad es
@given(x=st.integers(), c=colaAleatoria())
def test_perteneceCola(x: int, c: Cola[int]) -> None:
    r = perteneceCola(x, c)
    assert perteneceCola2(x, c) == r
    assert perteneceCola3(x, c) == r
    assert perteneceCola4(x, c) == r

# La comprobación es
#   src> poetry run pytest -q perteneceCola.py
#   1 passed in 0.25s

```

7.14. Inclusión de colas

7.14.1. En Haskell

```

-- -----
-- Utilizando el [tipo abstracto de datos de las colas](https://bit.ly/3QWTsRL),
-- definir la función

```

```
--   contenidaCola :: Eq a => Cola a -> Cola a -> Bool
--   tal que (contenidaCola c1 c2) se verifica si todos los elementos de
--   la cola c1 son elementos de la cola c2. Por ejemplo,
--   λ> ej1 = inserta 3 (inserta 2 vacia)
--   λ> ej2 = inserta 3 (inserta 4 vacia)
--   λ> ej3 = inserta 5 (inserta 2 (inserta 3 vacia))
--   λ> contenidaCola ej1 ej3
--   True
--   λ> contenidaCola ej2 ej3
--   False
```

```
-- -----
{-# OPTIONS_GHC -fno-warn-unused-imports #-}
```

```
module ContenidaCola where
```

```
import TAD.Cola (Cola, vacia, inserta, esVacia, primero, resto)
import PerteneceCola (perteneceCola)
import Transformaciones_colas_listas (colaAlista)
import Test.QuickCheck
```

```
-- 1ª solución
-- =====
```

```
contenidaCola1 :: Eq a => Cola a -> Cola a -> Bool
contenidaCola1 c1 c2
  | esVacia c1 = True
  | otherwise  = perteneceCola (primero c1) c2 &&
                  contenidaCola1 (resto c1) c2
```

```
-- La función perteneceCola está definida en el ejercicio
-- "TAD de las colas: Pertenencia a una cola" que se encuentra en
-- https://bit.ly/3RcVgqb
```

```
-- 2ª solución
-- =====
```

```
contenidaCola2 :: Eq a => Cola a -> Cola a -> Bool
contenidaCola2 c1 c2 =
  contenidaLista (colaAlista c1) (colaAlista c2)
```



```
-- La función colaAlista está definida en el ejercicio
-- "TAD de las colas: Transformaciones entre colas y listas" que se
-- encuentra en https://bit.ly/3Xv0oIt
```

```
contenidaLista :: Eq a => [a] -> [a] -> Bool
contenidaLista xs ys =
  all (`elem` ys) xs
```

```
-- Comprobación de equivalencia
-- =====
```

```
-- La propiedad es
prop_contenidaCola :: Cola Int -> Cola Int -> Bool
prop_contenidaCola c1 c2 =
  contenidaCola1 c1 c2 == contenidaCola2 c1 c2
```

```
-- La comprobación es
-- λ> quickCheck prop_contenidaCola
-- +++ OK, passed 100 tests.
```

7.14.2. En Python

```
# -----
# Utilizando el [tipo abstracto de datos de las colas](https://bit.ly/3QWTsRL),
# definir la función
#   contenidaCola : (Cola[A], Cola[A]) -> bool
# tal que contenidaCola(c1, c2) se verifica si todos los elementos de la
# cola c1 son elementos de la cola c2. Por ejemplo,
#   >>> ej1 = inserta(3, inserta(2, vacia()))
#   >>> ej2 = inserta(3, inserta(4, vacia()))
#   >>> ej3 = inserta(5, inserta(2, inserta(3, vacia())))
#   >>> contenidaCola(ej1, ej3)
#   True
#   >>> contenidaCola(ej2, ej3)
#   False
# -----
```

```
from copy import deepcopy
from typing import TypeVar
```

```

from hypothesis import given

from src.perteneceCola import perteneceCola
from src.TAD.cola import (Cola, colaAleatoria, esVacia, inserta, primero,
                          resto, vacia)
from src.transformaciones_colas_listas import colaAlista

A = TypeVar('A')

# 1ª solución
# =====

def contenidaCola1(c1: Cola[A], c2: Cola[A]) -> bool:
    if esVacia(c1):
        return True
    return perteneceCola(primero(c1), c2) and contenidaCola1(resto(c1), c2)

# La función perteneceCola está definida en el ejercicio
# "Perteneencia a una cola" que se encuentra en
# https://bit.ly/3RcVgqb

# 2ª solución
# =====

def contenidaCola2(c1: Cola[A], c2: Cola[A]) -> bool:
    return set(colaAlista(c1)) <= set(colaAlista(c2))

# La función colaAlista está definida en el ejercicio
# "Transformaciones entre colas y listas" que se encuentra en
# https://bit.ly/3Xv0oIt

# 3ª solución
# =====

def contenidaCola3Aux(c1: Cola[A], c2: Cola[A]) -> bool:
    if c1.esVacia():
        return True
    pc1 = c1.primero()
    c1.resto()

```

```

    return perteneceCola(pc1, c2) and contenidaCola1(c1, c2)

def contenidaCola3(c1: Cola[A], c2: Cola[A]) -> bool:
    _c1 = deepcopy(c1)
    return contenidaCola3Aux(_c1, c2)

# 4ª solución
# =====

def contenidaCola4Aux(c1: Cola[A], c2: Cola[A]) -> bool:
    while not c1.esVacia():
        pc1 = c1.primer()
        c1.resto()
        if not perteneceCola(pc1, c2):
            return False
    return True

def contenidaCola4(c1: Cola[A], c2: Cola[A]) -> bool:
    _c1 = deepcopy(c1)
    return contenidaCola4Aux(_c1, c2)

# Comprobación de equivalencia de las definiciones
# =====

# La propiedad es
@given(c1=colaAleatoria(), c2=colaAleatoria())
def test_contenidaCola(c1: Cola[int], c2: Cola[int]) -> None:
    r = contenidaCola1(c1, c2)
    assert contenidaCola2(c1, c2) == r
    assert contenidaCola3(c1, c2) == r
    assert contenidaCola4(c1, c2) == r

# La comprobación es
# src> poetry run pytest -q contenidaCola.py
# 1 passed in 0.44s

```

7.15. Reconocimiento de prefijos de colas

7.15.1. En Haskell

```

-----
-- Utilizando el [tipo abstracto de datos de las colas](https://bit.ly/3QWTsRL),
-- definir la función
--   prefijoCola :: Eq a => Cola a -> Cola a -> Bool
-- tal que (prefijoCola c1 c2) se verifica si la cola c1 es justamente
-- un prefijo de la cola c2. Por ejemplo,
--   λ> ej1 = inserta 4 (inserta 2 vacia)
--   λ> ej2 = inserta 5 (inserta 4 (inserta 2 vacia))
--   λ> ej3 = inserta 5 (inserta 2 (inserta 4 vacia))
--   λ> prefijoCola ej1 ej2
--   True
--   λ> prefijoCola ej1 ej3
--   False
-----

```

```
{-# OPTIONS_GHC -fno-warn-unused-imports #-}
```

```
module PrefijoCola where
```

```
import TAD.Cola (Cola, vacia, inserta, esVacia, primero, resto)
import Transformaciones_colas_listas (colaAlista)
import Data.List (isPrefixOf)
import Test.QuickCheck
```

```
-- 1ª solución
-- =====
```

```
prefijoCola :: Eq a => Cola a -> Cola a -> Bool
prefijoCola c1 c2
  | esVacia c1 = True
  | esVacia c2 = False
  | otherwise  = primero c1 == primero c2 &&
                    prefijoCola (resto c1) (resto c2)
```

```
-- 2ª solución
-- =====
```

```

prefijoCola2 :: Eq a => Cola a -> Cola a -> Bool
prefijoCola2 c1 c2 =
    colaAlista c1 `isPrefixOf` colaAlista c2

-- La función colaAlista está definida en el ejercicio
-- "Transformaciones entre colas y listas" que se encuentra en
-- https://bit.ly/3Xv0oIt

-- Comprobación de equivalencia
-- =====

-- La propiedad es
prop_prefijoCola :: Cola Int -> Cola Int -> Bool
prop_prefijoCola c1 c2 =
    prefijoCola c1 c2 == prefijoCola2 c1 c2

-- La comprobación es
-- λ> quickCheck prop_prefijoCola
-- +++ OK, passed 100 tests.

```

7.15.2. En Python

```

# -----
# Utilizando el [tipo abstracto de datos de las colas](https://bit.ly/3QWTsRL),
# definir la función
#   prefijoCola : (Cola[A], Cola[A]) -> bool
# tal que prefijoCola(c1, c2) se verifica si la cola c1 es justamente
# un prefijo de la cola c2. Por ejemplo,
#   >>> ej1 = inserta(4, inserta(2, vacia()))
#   >>> ej2 = inserta(5, inserta(4, inserta(2, vacia())))
#   >>> ej3 = inserta(5, inserta(2, inserta(4, vacia())))
#   >>> prefijoCola(ej1, ej2)
#   True
#   >>> prefijoCola(ej1, ej3)
#   False
# -----

from copy import deepcopy
from typing import TypeVar

```

```

from hypothesis import given

from src.TAD.cola import (Cola, colaAleatoria, esVacia, inserta, primero,
                          resto, vacia)
from src.transformaciones_colas_listas import colaAlista

A = TypeVar('A')

# 1ª solución
# =====

def prefijoCola(c1: Cola[A], c2: Cola[A]) -> bool:
    if esVacia(c1):
        return True
    if esVacia(c2):
        return False
    return primero(c1) == primero(c2) and prefijoCola(resto(c1), resto(c2))

# 2ª solución
# =====

def esPrefijoLista(xs: list[A], ys: list[A]) -> bool:
    return ys[:len(xs)] == xs

def prefijoCola2(c1: Cola[A], c2: Cola[A]) -> bool:
    return esPrefijoLista(colaAlista(c1), colaAlista(c2))

# La función colaAlista está definida en el ejercicio
# "Transformaciones entre colas y listas" que se encuentra en
# https://bit.ly/3Xv0oIt

# 3ª solución
# =====

def prefijoCola3Aux(c1: Cola[A], c2: Cola[A]) -> bool:
    if c1.esVacia():
        return True
    if c2.esVacia():
        return False
    cc1 = c1.primer()

```

```

    c1.resto()
    cc2 = c2.primer()
    c2.resto()
    return cc1 == cc2 and prefijoCola3(c1, c2)

def prefijoCola3(c1: Cola[A], c2: Cola[A]) -> bool:
    q1 = deepcopy(c1)
    q2 = deepcopy(c2)
    return prefijoCola3Aux(q1, q2)

# 4ª solución
# =====

def prefijoCola4Aux(c1: Cola[A], c2: Cola[A]) -> bool:
    while not c2.esVacia() and not c1.esVacia():
        if c1.primer() != c2.primer():
            return False
        c1.resto()
        c2.resto()
    return c1.esVacia()

def prefijoCola4(c1: Cola[A], c2: Cola[A]) -> bool:
    q1 = deepcopy(c1)
    q2 = deepcopy(c2)
    return prefijoCola4Aux(q1, q2)

# Comprobación de equivalencia de las definiciones
# =====

# La propiedad es
@given(c1=colaAleatoria(), c2=colaAleatoria())
def test_prefijoCola(c1: Cola[int], c2: Cola[int]) -> None:
    r = prefijoCola(c1, c2)
    assert prefijoCola2(c1, c2) == r
    assert prefijoCola3(c1, c2) == r
    assert prefijoCola4(c1, c2) == r

# La comprobación es
# src> poetry run pytest -q prefijoCola.py
# 1 passed in 0.29s

```

7.16. Reconocimiento de subcolas

7.16.1. En Haskell

```

-----
-- Utilizando el [tipo abstracto de datos de las colas](https://bit.ly/3QWTsRL),
-- definir la función
--   subCola :: Eq a => Cola a -> Cola a -> Bool
-- tal que (subCola c1 c2) se verifica si c1 es una subcola de c2. Por
-- ejemplo,
--   λ> ej1 = inserta 2 (inserta 3 vacia)
--   λ> ej2 = inserta 7 (inserta 2 (inserta 3 (inserta 5 vacia)))
--   λ> ej3 = inserta 2 (inserta 7 (inserta 3 (inserta 5 vacia)))
--   λ> subCola ej1 ej2
--   True
--   λ> subCola ej1 ej3
--   False
-----

```

```
{-# OPTIONS_GHC -fno-warn-unused-imports #-}
```

```
module SubCola where
```

```

import TAD.Cola (Cola, vacia, inserta, esVacia, primero, resto)
import Transformaciones_colas_listas (colaAlista)
import PrefijoCola (prefijoCola)
import Data.List (isPrefixOf, tails)
import Test.QuickCheck

```

```

-- 1ª solución
-- =====

```

```

subCola1 :: Eq a => Cola a -> Cola a -> Bool
subCola1 c1 c2
  | esVacia c1 = True
  | esVacia c2 = False
  | pc1 == pc2 = prefijoCola rc1 rc2 || subCola1 c1 rc2
  | otherwise  = subCola1 c1 rc2
  where pc1 = primero c1
        rc1 = resto c1
        pc2 = primero c2

```



```

rc2 = resto c2

-- La función PrefijoCola está definida en el ejercicio
-- "Reconocimiento de prefijos de colas" que se encuentra en
-- https://bit.ly/3HaK20x

-- 2ª solución
-- =====

subCola2 :: Eq a => Cola a -> Cola a -> Bool
subCola2 c1 c2 =
  sublista (colaAlista c1) (colaAlista c2)

-- La función colaAlista está definida en el ejercicio
-- "Transformaciones entre colas y listas" que se encuentra en
-- https://bit.ly/3Xv0oIt

-- (sublista xs ys) se verifica si xs es una sublista de ys. Por
-- ejemplo,
--   sublista [3,2] [5,3,2,7] == True
--   sublista [3,2] [5,3,7,2] == False
sublista :: Eq a => [a] -> [a] -> Bool
sublista xs ys =
  any (xs `isPrefixOf`) (tails ys)

-- Comprobación de equivalencia
-- =====

-- La propiedad es
prop_subCola :: Cola Int -> Cola Int -> Bool
prop_subCola c1 c2 =
  subCola1 c1 c2 == subCola2 c1 c2

-- La comprobación es
--   λ> quickCheck prop_subCola
--   +++ OK, passed 100 tests.

```

7.16.2. En Python

```
# -----
# Utilizando el [tipo abstracto de datos de las colas](https://bit.ly/3QWTsRL),
# definir la función
#     subCola : (Cola[A], Cola[A]) -> bool
# tal que subCola(c1, c2) se verifica si c1 es una subcola de c2. Por
# ejemplo,
#     >>> ej1 = inserta(2, inserta(3, vacia()))
#     >>> ej2 = inserta(7, inserta(2, inserta(3, inserta(5, vacia()))))
#     >>> ej3 = inserta(2, inserta(7, inserta(3, inserta(5, vacia()))))
#     >>> subCola(ej1, ej2)
#     True
#     >>> subCola(ej1, ej3)
#     False
# -----
```

```
from copy import deepcopy
from typing import TypeVar
```

```
from hypothesis import given
```

```
from src.prefijoCola import prefijoCola
from src.TAD.colas import Cola, colaAleatoria, esVacia, inserta, primero,
                        resto, vacia)
from src.transformaciones_colas_listas import colaAlista
```

```
A = TypeVar('A')
```

```
# 1ª solución
# =====
```

```
def subCola1(c1: Cola[A], c2: Cola[A]) -> bool:
    if esVacia(c1):
        return True
    if esVacia(c2):
        return False
    pc1 = primero(c1)
    rc1 = resto(c1)
    pc2 = primero(c2)
    rc2 = resto(c2)
```

```

    if pc1 == pc2:
        return prefijoCola(rc1, rc2) or subCola1(c1, rc2)
    return subCola1(c1, rc2)

# La función prefijoCola está definida en el ejercicio
# "Reconocimiento de prefijos de colas" que se encuentra en
# https://bit.ly/3HaK20x

# 2ª solución
# =====

# sublista(xs, ys) se verifica si xs es una sublista de ys. Por
# ejemplo,
# >>> sublista([3,2], [5,3,2,7])
# True
# >>> sublista([3,2], [5,3,7,2])
# False
def sublista(xs: list[A], ys: list[A]) -> bool:
    return any(xs == ys[i:i+len(xs)] for i in range(len(ys) - len(xs) + 1))

def subCola2(c1: Cola[A], c2: Cola[A]) -> bool:
    return sublista(colaAlista(c1), colaAlista(c2))

# La función colaAlista está definida en el ejercicio
# "Transformaciones entre colas y listas" que se encuentra en
# https://bit.ly/3Xv0oIt

# 3ª solución
# =====

def subCola3Aux(c1: Cola[A], c2: Cola[A]) -> bool:
    if c1.esVacia():
        return True
    if c2.esVacia():
        return False
    if c1.primer() != c2.primer():
        c2.resto()
        return subCola3Aux(c1, c2)
    q1 = deepcopy(c1)
    c1.resto()

```

```

    c2.resto()
    return prefijoCola(c1, c2) or subCola3Aux(q1, c2)

def subCola3(c1: Cola[A], c2: Cola[A]) -> bool:
    q1 = deepcopy(c1)
    q2 = deepcopy(c2)
    return subCola3Aux(q1, q2)

# Comprobación de equivalencia de las definiciones
# =====

# La propiedad es
@given(c1=colaAleatoria(), c2=colaAleatoria())
def test_subCola(c1: Cola[int], c2: Cola[int]) -> None:
    r = subCola1(c1, c2)
    assert subCola2(c1, c2) == r
    assert subCola3(c1, c2) == r

# La comprobación es
#   src> poetry run pytest -q subCola.py
#   1 passed in 0.31s

```

7.17. Reconocimiento de ordenación de colas

7.17.1. En Haskell

```

-- -----
-- Utilizando el [tipo abstracto de datos de las colas](https://bit.ly/3QWTsRL),
-- definir la función
--   ordenadaCola :: Ord a => Cola a -> Bool
-- tal que (ordenadaCola c) se verifica si los elementos de la cola c
-- están ordenados en orden creciente. Por ejemplo,
--   ordenadaCola (inserta 6 (inserta 5 (inserta 1 vacia))) == True
--   ordenadaCola (inserta 1 (inserta 0 (inserta 6 vacia))) == False
-- -----

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module OrdenadaCola where

```

```

import TAD.Cola (Cola, vacia, inserta, esVacia, primero, resto)
import Transformaciones_colas_listas (colaAlista)
import Test.QuickCheck

-- 1ª solución
-- =====

ordenadaCola :: Ord a => Cola a -> Bool
ordenadaCola c
  | esVacia c    = True
  | esVacia rc   = True
  | otherwise    = pc <= prc && ordenadaCola rc
  where pc      = primero c
        rc      = resto c
        prc     = primero rc

-- 2ª solución
-- =====

ordenadaCola2 :: Ord a => Cola a -> Bool
ordenadaCola2 =
  ordenadaLista . colaAlista

-- (ordenadaLista xs) se verifica si la lista xs está ordenada de menor
-- a mayor. Por ejemplo,
ordenadaLista :: Ord a => [a] -> Bool
ordenadaLista xs =
  and [x <= y | (x,y) <- zip xs (tail xs)]

-- La función colaAlista está definida en el ejercicio
-- "Transformaciones entre colas y listas" que se encuentra en
-- https://bit.ly/3Xv0oIt

-- Comprobación de equivalencia
-- =====

-- La propiedad es
prop_ordenadaCola :: Cola Int -> Bool
prop_ordenadaCola c =
  ordenadaCola c == ordenadaCola2 c

```

```
-- La comprobación es
--   λ> quickCheck prop_ordenadaCola
--   +++ OK, passed 100 tests.
```

7.17.2. En Python

```
# -----
# Utilizando el [tipo abstracto de datos de las colas](https://bit.ly/3QWTsRL),
# definir la función
#   ordenadaCola : (Cola[A]) -> bool
# tal que ordenadaCola(c) se verifica si los elementos de la cola c
# están ordenados en orden creciente. Por ejemplo,
#   >>> ordenadaCola(inserta(6, inserta(5, inserta(1, vacia()))))
#   True
#   >>> ordenadaCola(inserta(1, inserta(0, inserta(6, vacia()))))
#   False
# -----
```

```
from copy import deepcopy
from typing import TypeVar
```

```
from hypothesis import given
```

```
from src.TAD.colas import (Cola, colaAleatoria, esVacia, inserta, primero,
                           resto, vacia)
from src.transformaciones_colas_listas import colaAlista
```

```
A = TypeVar('A', int, float, str)
```

```
# 1ª solución
# =====
```

```
def ordenadaCola(c: Cola[A]) -> bool:
    if esVacia(c):
        return True
    pc = primero(c)
    rc = resto(c)
    if esVacia(rc):
        return True
```

```

    prc = primero(rc)
    return pc <= prc and ordenadaCola(rc)

# 2ª solución
# =====

# ordenadaLista(xs, ys) se verifica si xs es una lista ordenada. Por
# ejemplo,
# >>> ordenadaLista([2, 5, 8])
# True
# >>> ordenadaLista([2, 8, 5])
# False
def ordenadaLista(xs: list[A]) -> bool:
    return all((x <= y for (x, y) in zip(xs, xs[1:])))

def ordenadaCola2(p: Cola[A]) -> bool:
    return ordenadaLista(colaALista(p))

# La función colaALista está definida en el ejercicio
# "Transformaciones entre colas y listas" que se encuentra en
# https://bit.ly/3Xv0oIt

# 3ª solución
# =====

def ordenadaCola3Aux(c: Cola[A]) -> bool:
    if c.esVacia():
        return True
    pc = c.primer()
    c.resto()
    if c.esVacia():
        return True
    return pc <= c.primer() and ordenadaCola3Aux(c)

def ordenadaCola3(c: Cola[A]) -> bool:
    _c = deepcopy(c)
    return ordenadaCola3Aux(_c)

# 4ª solución
# =====

```

```

def ordenadaCola4Aux(c: Cola[A]) -> bool:
    while not c.esVacia():
        pc = c.primer()
        c.resto()
        if not c.esVacia() and pc > c.primer():
            return False
    return True

def ordenadaCola4(c: Cola[A]) -> bool:
    _c = deepcopy(c)
    return ordenadaCola4Aux(_c)

# Comprobación de equivalencia de las definiciones
# =====

# La propiedad es
@given(p=colaAleatoria())
def test_ordenadaCola(p: Cola[int]) -> None:
    r = ordenadaCola(p)
    assert ordenadaCola2(p) == r
    assert ordenadaCola3(p) == r
    assert ordenadaCola4(p) == r

# La comprobación es
#   src> poetry run pytest -q ordenadaCola.py
#   1 passed in 0.27s

)

```

7.18. Máximo elemento de una cola

7.18.1. En Haskell

```

-----
-- Utilizando el [tipo abstracto de datos de las colas](https://bit.ly/3QWTsRL),
-- definir la función
--   maxCola :: Ord a => Cola a -> a
-- tal que (maxCola c) sea el mayor de los elementos de la cola c. Por
-- ejemplo,

```



```

--      λ> maxCola (inserta 3 (inserta 5 (inserta 1 vacia)))
--      5
--      -----

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module MaxCola where

import TAD.Cola (Cola, vacia, inserta, esVacia, primero, resto)
import Transformaciones_colas_listas (colaAlista)
import Test.QuickCheck

-- 1ª solución
-- =====

maxCola1 :: Ord a => Cola a -> a
maxCola1 c
  | esVacia rc = pc
  | otherwise  = max pc (maxCola1 rc)
  where pc = primero c
        rc = resto c

-- 2ª solución
-- =====

maxCola2 :: Ord a => Cola a -> a
maxCola2 =
  maximum . colaAlista

-- La función colaAlista está definida en el ejercicio
-- "Transformaciones entre colas y listas" que se encuentra en
-- https://bit.ly/3Xv0oIt

-- Comprobación de equivalencia
-- =====

-- La propiedad es
prop_maxCola :: Cola Int -> Property
prop_maxCola c =
  not (esVacia c) ==> maxCola1 c == maxCola2 c

```

```
-- La comprobación es
--   λ> quickCheck prop_maxCola
--   +++ OK, passed 100 tests; 16 discarded.
```

7.18.2. En Python

```
# -----
# Utilizando el [tipo abstracto de datos de las colas](https://bit.ly/3QWTsRL),
# definir la función
#   maxCola : (Cola[A]) -> A
# tal que maxCola(c) sea el mayor de los elementos de la cola c. Por
# ejemplo,
#   >>> maxCola(inserta(3, inserta(5, inserta(1, vacia()))))
#   5
# -----
```

```
from copy import deepcopy
from typing import TypeVar
```

```
from hypothesis import assume, given
```

```
from src.TAD.cola import (Cola, colaAleatoria, esVacia, inserta, primero,
                          resto, vacia)
from src.transformaciones_colas_listas import colaAlista
```

```
A = TypeVar('A', int, float, str)
```

```
# 1ª solución
# =====
```

```
def maxCola1(c: Cola[A]) -> A:
    pc = primero(c)
    rc = resto(c)
    if esVacia(rc):
        return pc
    return max(pc, maxCola1(rc))
```

```
# 2ª solución
# =====
```

```
# Se usará la función colaAlista del ejercicio
# "Transformaciones entre colas y listas" que se encuentra en
# https://bit.ly/3ZHewQ8
```

```
def maxCola2(c: Cola[A]) -> A:
    return max(colaAlista(c))
```

```
# 3ª solución
# =====
```

```
def maxCola3Aux(c: Cola[A]) -> A:
    pc = c.primer()
    c.resto()
    if esVacia(c):
        return pc
    return max(pc, maxCola3Aux(c))
```

```
def maxCola3(c: Cola[A]) -> A:
    _c = deepcopy(c)
    return maxCola3Aux(_c)
```

```
# 4ª solución
# =====
```

```
def maxCola4Aux(c: Cola[A]) -> A:
    r = c.primer()
    while not esVacia(c):
        pc = c.primer()
        if pc > r:
            r = pc
        c.resto()
    return r
```

```
def maxCola4(c: Cola[A]) -> A:
    _c = deepcopy(c)
    return maxCola4Aux(_c)
```

```
# Comprobación de equivalencia de las definiciones
# =====
```

```
# La propiedad es
@given(c=colaAleatoria())
def test_maxCola(c: Cola[int]) -> None:
    assume(not esVacia(c))
    r = maxCola1(c)
    assert maxCola2(c) == r
    assert maxCola3(c) == r
    assert maxCola4(c) == r

# La comprobación es
# src> poetry run pytest -q maxCola.py
# 1 passed in 0.30s
```

Capítulo 8

El tipo abstracto de datos de los conjuntos

8.1. El tipo abstracto de datos de los conjuntos

8.1.1. En Haskell

```
-- Un conjunto es una estructura de datos, caracterizada por ser una
-- colección de elementos en la que no importe ni el orden ni la
-- repetición de elementos.
--
-- Las operaciones que definen al tipo abstracto de datos (TAD) de los
-- conjuntos (cuyos elementos son del tipo a) son las siguientes:
--     vacio      :: Conj a
--     inserta    :: Ord a => a -> Conj a -> Conj a
--     menor      :: Ord a => Conj a -> a
--     elimina    :: Ord a => a -> Conj a -> Conj a
--     pertenece  :: Ord a => a -> Conj a -> Bool
--     esVacio    :: Conj a -> Bool
--
-- tales que
--     + vacio es el conjunto vacío.
--     + (inserta x c) es el conjunto obtenido añadiendo el elemento x al
--       conjunto c.
--     + (menor c) es el menor elemento del conjunto c.
--     + (elimina x c) es el conjunto obtenido eliminando el elemento x
--       del conjunto c.
--     + (pertenece x c) se verifica si x pertenece al conjunto c.
--     + (esVacio c) se verifica si c es el conjunto vacío.
--
```

```
-- Las operaciones tienen que verificar las siguientes propiedades:
--   + inserta x (inserta x c) == inserta x c
--   + inserta x (inserta y c) == inserta y (inserta x c)
--   + not (pertenece x vacio)
--   + pertenece y (inserta x c) == (x==y) || pertenece y c
--   + elimina x vacio == vacio
--   + Si x == y, entonces
--       elimina x (inserta y c) == elimina x c
--   + Si x /= y, entonces
--       elimina x (inserta y c) == inserta y (elimina x c)
--   + esVacio vacio
--   + not (esVacio (inserta x c))
--
-- Para usar el TAD hay que usar una implementación concreta. En
-- principio, consideraremos las siguientes:
--   + mediante listas no ordenadas con duplicados,
--   + mediante listas no ordenadas sin duplicados,
--   + mediante listas ordenadas sin duplicados y
--   + mediante la librería Data.Set.
-- Hay que elegir la que se desee utilizar, descomentándola y comentando
-- las otras.
```

```
{-# OPTIONS_GHC -fno-warn-unused-imports #-}
```

module TAD.Conjunto

```
(Conj,
 vacio,      -- Conj a
 inserta,    -- Ord a => a -> Conj a -> Conj a
 menor,      -- Ord a => Conj a -> a
 elimina,    -- Ord a => a -> Conj a -> Conj a
 pertenece,  -- Ord a => a -> Conj a -> Bool
 esVacio     -- Conj a -> Bool
) where

-- import TAD.ConjuntoConListasNoOrdenadasConDuplicados
-- import TAD.ConjuntoConListasNoOrdenadasSinDuplicados
import TAD.ConjuntoConListasOrdenadasSinDuplicados
-- import TAD.ConjuntoConLibreria
```

8.1.2. En Python

```
# Un conjunto es una estructura de datos, caracterizada por ser una
# colección de elementos en la que no importe ni el orden ni la
# repetición de elementos.
#
# Las operaciones que definen al tipo abstracto de datos (TAD) de los
# conjuntos (cuyos elementos son del tipo a) son las siguientes:
#   vacio      :: Conj a
#   inserta    :: Ord a => a -> Conj a -> Conj a
#   menor      :: Ord a => Conj a -> a
#   elimina    :: Ord a => a -> Conj a -> Conj a
#   pertenece  :: Ord a => a -> Conj a -> Bool
#   esVacio    :: Conj a -> Bool
# tales que
#   + vacio es el conjunto vacío.
#   + (inserta x c) es el conjunto obtenido añadiendo el elemento x al
#     conjunto c.
#   + (menor c) es el menor elemento del conjunto c.
#   + (elimina x c) es el conjunto obtenido eliminando el elemento x
#     del conjunto c.
#   + (pertenece x c) se verifica si x pertenece al conjunto c.
#   + (esVacio c) se verifica si c es el conjunto vacío.
#
# Las operaciones tienen que verificar las siguientes propiedades:
#   + inserta x (inserta x c) == inserta x c
#   + inserta x (inserta y c) == inserta y (inserta x c)
#   + not (pertenece x vacio)
#   + pertenece y (inserta x c) == (x==y) || pertenece y c
#   + elimina x vacio == vacio
#   + Si x == y, entonces
#     elimina x (inserta y c) == elimina x c
#   + Si x /= y, entonces
#     elimina x (inserta y c) == inserta y (elimina x c)
#   + esVacio vacio
#   + not (esVacio (inserta x c))
#
# Para usar el TAD hay que usar una implementación concreta. En
# principio, consideraremos las siguientes:
#   + mediante listas no ordenadas con duplicados,
#   + mediante listas no ordenadas sin duplicados,
```

```
# + mediante listas ordenadas sin duplicados y
# + mediante la librería Data.Set.
# Hay que elegir la que se desee utilizar, descomentándola y comentando
# las otras.

__all__ = [
    'Conj',
    'vacio',
    'inserta',
    'menor',
    'elimina',
    'pertenece',
    'esVacio',
    'conjuntoAleatorio'
]

# from src.TAD.conjuntoConListasNoOrdenadasConDuplicados import (
#     Conj, conjuntoAleatorio, elimina, esVacio, inserta,
#     menor, pertenece, vacio)

# from src.TAD.conjuntoConListasNoOrdenadasSinDuplicados import (
#     Conj, conjuntoAleatorio, elimina, esVacio, inserta, menor, pertenece,
#     vacio)

from src.TAD.conjuntoConListasOrdenadasSinDuplicados import (Conj,
                                                                conjuntoAleatorio,
                                                                elimina, esVacio,
                                                                inserta, menor,
                                                                pertenece, vacio)

# from src.TAD.conjuntoConLibreria import (
#     Conj, conjuntoAleatorio, elimina, esVacio, inserta, menor, pertenece,
#     vacio)
```


8.2. El tipo de datos de los conjuntos mediante listas no ordenadas con duplicados

8.2.1. En Haskell

```
{-# OPTIONS_GHC -fno-warn-unused-top-binds #-}
```

```
module TAD.ConjuntoConListasNoOrdenadasConDuplicados
```

```
  (Conj,
   vacio,      -- Conj a
   inserta,    -- Ord a => a -> Conj a -> Conj a
   menor,      -- Ord a => Conj a -> a
   elimina,    -- Ord a => a -> Conj a -> Conj a
   pertenece,  -- Ord a => a -> Conj a -> Bool
   esVacio     -- Conj a -> Bool
  ) where
```

```
import Data.List (intercalate, nub, sort)
```

```
import Test.QuickCheck
```

```
-- Conjuntos como listas no ordenadas con repeticiones:
```

```
newtype Conj a = Cj [a]
```

```
-- (escribeConjunto c) es la cadena correspondiente al conjunto c. Por
-- ejemplo,
```

```
-- λ> escribeConjunto (Cj [])
```

```
-- "{}"
```

```
-- λ> escribeConjunto (Cj [5])
```

```
-- "{5}"
```

```
-- λ> escribeConjunto (Cj [2, 5])
```

```
-- "{2, 5}"
```

```
-- λ> escribeConjunto (Cj [5, 2, 5])
```

```
-- "{2, 5}"
```

```
escribeConjunto :: (Show a, Ord a) => Conj a -> String
```

```
escribeConjunto (Cj xs) =
```

```
  "{" ++ intercalate ", " (map show (sort (nub xs))) ++ "}"
```

```
-- Procedimiento de escritura de conjuntos.
```

```
instance (Show a, Ord a) => Show (Conj a) where
```

```
  show = escribeConjunto
```

```
-- Nota: Aunque el conjunto no está ordenado y tenga repeticiones, al
-- escribirlo se hará sin repeticiones y ordenando sus elementos.
```

```
-- vacio es el conjunto vacío. Por ejemplo,
--   λ> vacio
--   {}
```

```
vacio :: Conj a
vacio = Cj []
```

```
-- (inserta x c) es el conjunto obtenido añadiendo el elemento x al
-- conjunto c. Por ejemplo,
```

```
--   λ> inserta 5 vacio
--   {5}
--   λ> inserta 2 (inserta 5 vacio)
--   {2, 5}
--   λ> inserta 5 (inserta 2 vacio)
--   {2, 5}
```

```
inserta :: Eq a => a -> Conj a -> Conj a
inserta x (Cj ys) = Cj (x:ys)
```

```
-- (menor c) es el menor elemento del conjunto c. Por ejemplo,
```

```
--   λ> menor (inserta 5 (inserta 2 vacio))
--   2
```

```
menor :: Ord a => Conj a -> a
menor (Cj []) = error "conjunto vacío"
menor (Cj xs) = minimum xs
```

```
-- (elimina x c) es el conjunto obtenido eliminando el elemento x
-- del conjunto c. Por ejemplo,
```

```
--   λ> elimina 2 (inserta 5 (inserta 2 vacio))
--   {5}
```

```
elimina :: Eq a => a -> Conj a -> Conj a
elimina x (Cj ys) = Cj (filter (/= x) ys)
```

```
-- (esVacio c) se verifica si c es el conjunto vacío. Por ejemplo,
```

```
--   λ> esVacio (inserta 5 (inserta 2 vacio))
--   False
--   λ> esVacio vacio
--   True
```

```

esVacio :: Conj a -> Bool
esVacio (Cj xs) = null xs

-- (pertenece x c) se verifica si x pertenece al conjunto c. Por ejemplo,
--   λ> pertenece 2 (inserta 5 (inserta 2 vacio))
--   True
--   λ> pertenece 4 (inserta 5 (inserta 2 vacio))
--   False
pertenece :: Eq a => a -> Conj a -> Bool
pertenece x (Cj xs) = x `elem` xs

-- (subconjunto c1 c2) se verifica si c1 es un subconjunto de c2. Por
-- ejemplo,
--   subconjunto (Cj [1,3,2,1]) (Cj [3,1,3,2]) == True
--   subconjunto (Cj [1,3,4,1]) (Cj [3,1,3,2]) == False
subconjunto :: Ord a => Conj a -> Conj a -> Bool
subconjunto (Cj xs) (Cj ys) = sublista xs ys
  where sublista [] _      = True
        sublista (z:zs) vs = elem z vs && sublista zs vs

-- (igualConjunto c1 c2) se verifica si los conjuntos c1 y c2 son
-- iguales. Por ejemplo,
--   igualConjunto (Cj [1,3,2,1]) (Cj [3,1,3,2]) == True
--   igualConjunto (Cj [1,3,4,1]) (Cj [3,1,3,2]) == False
igualConjunto :: Ord a => Conj a -> Conj a -> Bool
igualConjunto c c' =
  subconjunto c c' && subconjunto c' c

--- Los conjuntos son comparables por igualdad.
instance Ord a => Eq (Conj a) where
  (==) = igualConjunto

-- Generador de conjuntos
-- =====

-- genConjunto es un generador de conjuntos. Por ejemplo,
--   λ> sample (genConjunto :: Gen (Conj Int))
--   {}
--   {1}
--   {0, 2, 3}

```

```

-- {-6, 5}
-- {2, 5}
-- {-9, -6, 4, 8}
-- {0, 1}
-- {-13, -11, -5, -2, -1, 0, 4, 6, 7, 8, 9, 14}
-- {-7, -5, -2, -1, 1, 2, 10, 13, 15}
-- {-18, -17, -16, -10, -9, 0, 1, 3, 4, 13, 16}
-- {-20, -15, -7, -1, 2, 8, 10, 15, 20}
genConjunto :: (Arbitrary a, Ord a) => Gen (Conj a)
genConjunto = do
  xs <- listOf arbitrary
  return (foldr inserta vacio xs)

-- Los conjuntos son concreciones de los arbitrarios.
instance (Arbitrary a, Ord a) => Arbitrary (Conj a) where
  arbitrary = genConjunto

-- Propiedades de los conjuntos
-- =====

prop_conjuntos :: Int -> Int -> Conj Int -> Bool
prop_conjuntos x y c =
  inserta x (inserta x c) == inserta x c &&
  inserta x (inserta y c) == inserta y (inserta x c) &&
  not (pertenece x vacio) &&
  pertenece y (inserta x c) == (x == y) || pertenece y c &&
  elimina x vacio == vacio &&
  elimina x (inserta y c) == (if x == y
                             then elimina x c
                             else inserta y (elimina x c)) &&
  esVacio (vacio :: Conj Int) &&
  not (esVacio (inserta x c))

-- Comprobación
-- λ> quickCheck prop_conjuntos
-- +++ OK, passed 100 tests.

```

8.2.2. En Python

```
# Se define la clase Conj con los siguientes métodos:
#   + inserta(x) añade x al conjunto.
#   + menor() es el menor elemento del conjunto.
#   + elimina(x) elimina las ocurrencias de x en el conjunto.
#   + pertenece(x) se verifica si x pertenece al conjunto.
#   + esVacia() se verifica si la cola es vacía.
# Por ejemplo,
#   >>> c = Conj()
#   >>> c
#   {}
#   >>> c.inserta(5)
#   >>> c.inserta(2)
#   >>> c.inserta(3)
#   >>> c.inserta(4)
#   >>> c.inserta(5)
#   >>> c
#   {2, 3, 4, 5}
#   >>> c.menor()
#   2
#   >>> c.elimina(3)
#   >>> c
#   {2, 4, 5}
#   >>> c.pertenece(4)
#   True
#   >>> c.pertenece(3)
#   False
#   >>> c.esVacio()
#   False
#   >>> c = Conj()
#   >>> c.esVacio()
#   True
#   >>> c = Conj()
#   >>> c.inserta(2)
#   >>> c.inserta(5)
#   >>> d = Conj()
#   >>> d.inserta(5)
#   >>> d.inserta(2)
#   >>> d.inserta(5)
#   >>> c == d
```

```

#     True
#
# Además se definen las correspondientes funciones. Por ejemplo,
#     >>> vacio()
#     {}
#     >>> inserta(5, inserta(3, inserta(2, inserta(5, vacio()))))
#     {2, 3, 5}
#     >>> menor(inserta(5, inserta(3, inserta(2, inserta(5, vacio())))))
#     2
#     >>> elimina(5, inserta(5, inserta(3, inserta(2, inserta(5, vacio())))))
#     {2, 3}
#     >>> pertenece(5, inserta(5, inserta(3, inserta(2, inserta(5, vacio())))))
#     True
#     >>> pertenece(1, inserta(5, inserta(3, inserta(2, inserta(5, vacio())))))
#     False
#     >>> esVacio(inserta(5, inserta(3, inserta(2, inserta(5, vacio())))))
#     False
#     >>> esVacio(vacio())
#     True
#     >>> inserta(5, inserta(2, vacio())) == inserta(2, inserta(5, (inserta(2, vacio()))))
#     True
#
# Finalmente, se define un generador aleatorio de conjuntos y se
# comprueba que los conjuntos cumplen las propiedades de su
# especificación.

```

```

from __future__ import annotations

```

```

__all__ = [
    'Conj',
    'vacio',
    'inserta',
    'menor',
    'elimina',
    'pertenece',
    'esVacio',
    'conjuntoAleatorio'
]

```

```

from abc import abstractmethod

```

```

from copy import deepcopy
from dataclasses import dataclass, field
from typing import Any, Generic, Protocol, TypeVar

```

```

from hypothesis import given
from hypothesis import strategies as st

```

```

class Comparable(Protocol):
    @abstractmethod
    def __lt__(self: A, otro: A) -> bool:
        pass

```

```

A = TypeVar('A', bound=Comparable)

```

```

# Clase de los conjuntos mediante listas no ordenadas con duplicados
# =====

```

```

@dataclass

```

```

class Conj(Generic[A]):

```

```

    _elementos: list[A] = field(default_factory=list)

```

```

    def __repr__(self) -> str:
        """

```

```

        Devuelve una cadena con los elementos del conjunto entre llaves
        y separados por ", ".
        """

```

```

        return '{' + ', '.join(str(x) for x in sorted(list(set(self._elementos))))

```

```

    def __eq__(self, c: Any) -> bool:
        """

```

```

        Se verifica si el conjunto es igual a c; es decir, tienen los
        mismos elementos sin importar el orden ni las repeticiones.
        """

```

```

        return sorted(list(set(self._elementos))) == sorted(list(set(c._elementos)))

```

```

    def inserta(self, x: A) -> None:
        """

```

```

        Añade el elemento x al conjunto.
        """

```

```

        self._elementos.append(x)

    def menor(self) -> A:
        """
        Devuelve el menor elemento del conjunto
        """
        return min(self._elementos)

    def elimina(self, x: A) -> None:
        """
        Elimina el elemento x del conjunto.
        """
        while x in self._elementos:
            self._elementos.remove(x)

    def esVacio(self) -> bool:
        """
        Se verifica si el conjunto está vacío.
        """
        return not self._elementos

    def pertenece(self, x: A) -> bool:
        """
        Se verifica si x pertenece al conjunto.
        """
        return x in self._elementos

# Funciones del tipo conjunto
# =====

def vacio() -> Conj[A]:
    """
    Crea y devuelve un conjunto vacío de tipo A.
    """
    c: Conj[A] = Conj()
    return c

def inserta(x: A, c: Conj[A]) -> Conj[A]:
    """
    Inserta un elemento x en el conjunto c y devuelve un nuevo conjunto

```



```

        con el elemento insertado.
        """
        _aux = deepcopy(c)
        _aux.inserta(x)
        return _aux

def menor(c: Conj[A]) -> A:
    """
    Devuelve el menor elemento del conjunto c.
    """
    return c.menor()

def elimina(x: A, c: Conj[A]) -> Conj[A]:
    """
    Elimina las ocurrencias de c en c y devuelve una copia del conjunto
    resultante.
    """
    _aux = deepcopy(c)
    _aux.elimina(x)
    return _aux

def pertenece(x: A, c: Conj[A]) -> bool:
    """
    Se verifica si x pertenece a c.
    """
    return c.pertenece(x)

def esVacio(c: Conj[A]) -> bool:
    """
    Se verifica si el conjunto está vacío.
    """
    return c.esVacio()

# Generador de conjuntos
# =====

def conjuntoAleatorio() -> st.SearchStrategy[Conj[int]]:
    """
    Genera una estrategia de búsqueda para generar conjuntos de enteros
    de forma aleatoria.
    """

```

```

Utiliza la librería Hypothesis para generar una lista de enteros y
luego se convierte en una instancia de la clase cola.
"""
    return st.lists(st.integers()).map(Conj)

# Comprobación de las propiedades de los conjuntos
# =====

# Las propiedades son
@given(c=conjuntoAleatorio(), x=st.integers(), y=st.integers())
def test_conjuntos(c: Conj[int], x: int, y: int) -> None:
    v: Conj[int] = vacio()
    assert inserta(x, inserta(x, c)) == inserta(x, c)
    assert inserta(x, inserta(y, c)) == inserta(y, inserta(x, c))
    assert not pertenece(x, v)
    assert pertenece(y, inserta(x, c)) == (x == y) or pertenece(y, c)
    assert elimina(x, v) == v

    def relacion(x: int, y: int, c: Conj[int]) -> Conj[int]:
        if x == y:
            return elimina(x, c)
        return inserta(y, elimina(x, c))

    assert elimina(x, inserta(y, c)) == relacion(x, y, c)
    assert esVacio(vacio())
    assert not esVacio(inserta(x, c))

# La comprobación es
# > poetry run pytest -q conjuntoConListasNoOrdenadasConDuplicados.py
# 1 passed in 0.33s

```

8.3. El tipo de datos de los conjuntos mediante listas no ordenadas sin duplicados

8.3.1. En Haskell

```
{-# OPTIONS_GHC -fno-warn-unused-top-binds #-}
```

module TAD.ConjuntoConListasNoOrdenadasSinDuplicados

```

(Conj,
 vacio,      -- Conj a
 inserta,    -- Ord a => a -> Conj a -> Conj a
 menor,      -- Ord a => Conj a -> a
 elimina,    -- Ord a => a -> Conj a -> Conj a
 pertenece,  -- Ord a => a -> Conj a -> Bool
 esVacio     -- Conj a -> Bool
) where

import Data.List (intercalate, sort)
import Test.QuickCheck

-- Los conjuntos como listas no ordenadas sin repeticiones.
newtype Conj a = Cj [a]

-- (escribeConjunto c) es la cadena correspondiente al conjunto c. Por
-- ejemplo,
--   λ> escribeConjunto (Cj [])
--   "{}"
--   λ> escribeConjunto (Cj [5])
--   "{5}"
--   λ> escribeConjunto (Cj [2, 5])
--   "{2, 5}"
--   λ> escribeConjunto (Cj [5, 2])
--   "{2, 5}"
escribeConjunto :: (Show a, Ord a) => Conj a -> String
escribeConjunto (Cj xs) =
    "{" ++ intercalate ", " (map show (sort xs)) ++ "}"

-- Procedimiento de escritura de conjuntos.
instance (Show a, Ord a) => Show (Conj a) where
    show = escribeConjunto

-- vacio es el conjunto vacío. Por ejemplo,
--   λ> vacio
--   {}
vacio :: Conj a
vacio = Cj []

```

```

-- (inserta x c) es el conjunto obtenido añadiendo el elemento x al
-- conjunto c. Por ejemplo,
--   λ> inserta 5 vacio
--   {5}
--   λ> inserta 2 (inserta 5 vacio)
--   {2, 5}
--   λ> inserta 5 (inserta 2 vacio)
--   {2, 5}
inserta :: Eq a => a -> Conj a -> Conj a
inserta x s@(Cj xs) | pertenece x s = s
                    | otherwise     = Cj (x:xs)

-- (menor c) es el menor elemento del conjunto c. Por ejemplo,
--   λ> menor (inserta 5 (inserta 2 vacio))
--   2
menor :: Ord a => Conj a -> a
menor (Cj []) = error "conjunto vacío"
menor (Cj xs) = minimum xs

-- (elimina x c) es el conjunto obtenido eliminando el elemento x
-- del conjunto c. Por ejemplo,
--   λ> elimina 2 (inserta 5 (inserta 2 vacio))
--   {5}
elimina :: Eq a => a -> Conj a -> Conj a
elimina x (Cj s) = Cj [y | y <- s, y /= x]

-- (esVacio c) se verifica si c es el conjunto vacío. Por ejemplo,
--   λ> esVacio (inserta 5 (inserta 2 vacio))
--   False
--   λ> esVacio vacio
--   True
esVacio :: Conj a -> Bool
esVacio (Cj xs) = null xs

-- (pertenece x c) se verifica si x pertenece al conjunto c. Por ejemplo,
--   λ> pertenece 2 (inserta 5 (inserta 2 vacio))
--   True
--   λ> pertenece 4 (inserta 5 (inserta 2 vacio))
--   False
pertenece :: Eq a => a -> Conj a -> Bool

```

```

pertenece x (Cj xs) = x `elem` xs

-- (subconjunto c1 c2) se verifica si c1 es un subconjunto de c2. Por
-- ejemplo,
--     subconjunto (Cj [1,3,2]) (Cj [3,1,2])    == True
--     subconjunto (Cj [1,3,4,1]) (Cj [1,3,2]) == False
subconjunto :: Ord a => Conj a -> Conj a -> Bool
subconjunto (Cj xs) (Cj ys) = sublista xs ys
  where sublista [] _      = True
        sublista (z:zs) vs = elem z vs && sublista zs vs

-- (igualConjunto c1 c2) se verifica si los conjuntos c1 y c2 son
-- iguales. Por ejemplo,
--     igualConjunto (Cj [3,2,1]) (Cj [1,3,2]) == True
--     igualConjunto (Cj [1,3,4]) (Cj [1,3,2]) == False
igualConjunto :: Ord a => Conj a -> Conj a -> Bool
igualConjunto c c' =
  subconjunto c c' && subconjunto c' c

--- Los conjuntos son comparables por igualdad.
instance Ord a => Eq (Conj a) where
  (==) = igualConjunto

-- Generador de conjuntos
-- =====

-- genConjunto es un generador de conjuntos. Por ejemplo,
--     λ> sample (genConjunto :: Gen (Conj Int))
--     {}
--     {-1, 0}
--     {-4, 1, 2}
--     {-3, 0, 2, 3, 4}
--     {-7}
--     {-10, -7, -5, -2, -1, 2, 5, 8}
--     {5, 7, 8, 10}
--     {-9, -6, -3, 8}
--     {-8, -6, -5, -1, 7, 9, 14}
--     {-18, -15, -14, -13, -3, -2, 1, 2, 4, 8, 12, 17}
--     {-17, -16, -13, -12, -11, -9, -6, -3, 0, 1, 3, 5, 6, 7, 16, 18}
genConjunto :: (Arbitrary a, Ord a) => Gen (Conj a)

```

```

genConjunto = do
  xs <- listOf arbitrary
  return (foldr inserta vacio xs)

-- Los conjuntos son concreciones de los arbitrarios.
instance (Arbitrary a, Ord a) => Arbitrary (Conj a) where
  arbitrary = genConjunto

-- Propiedades de los conjuntos
-- =====

prop_conjuntos :: Int -> Int -> Conj Int -> Bool
prop_conjuntos x y c =
  inserta x (inserta x c) == inserta x c &&
  inserta x (inserta y c) == inserta y (inserta x c) &&
  not (pertenece x vacio) &&
  pertenece y (inserta x c) == (x == y) || pertenece y c &&
  elimina x vacio == vacio &&
  elimina x (inserta y c) == (if x == y
                              then elimina x c
                              else inserta y (elimina x c)) &&
  esVacio (vacio :: Conj Int) &&
  not (esVacio (inserta x c))

-- Comprobación
--   λ> quickCheck prop_conjuntos
--   +++ OK, passed 100 tests.

```

8.3.2. En Python

```

# Se define la clase Conj con los siguientes métodos:
#   + inserta(x) añade x al conjunto.
#   + menor() es el menor elemento del conjunto.
#   + elimina(x) elimina las ocurrencias de x en el conjunto.
#   + pertenece(x) se verifica si x pertenece al conjunto.
#   + esVacia() se verifica si la cola es vacía.
# Por ejemplo,
#   >>> c = Conj()
#   >>> c
#   {}

```

```
# >>> c.inserta(5)
# >>> c.inserta(2)
# >>> c.inserta(3)
# >>> c.inserta(4)
# >>> c.inserta(5)
# >>> c
# {2, 3, 4, 5}
# >>> c.menor()
# 2
# >>> c.elimina(3)
# >>> c
# {2, 4, 5}
# >>> c.pertenece(4)
# True
# >>> c.pertenece(3)
# False
# >>> c.esVacio()
# False
# >>> c = Conj()
# >>> c.esVacio()
# True
# >>> c = Conj()
# >>> c.inserta(2)
# >>> c.inserta(5)
# >>> d = Conj()
# >>> d.inserta(5)
# >>> d.inserta(2)
# >>> d.inserta(5)
# >>> c == d
# True
#
# Además se definen las correspondientes funciones. Por ejemplo,
# >>> vacio()
# {}
# >>> inserta(5, inserta(3, inserta(2, inserta(5, vacio()))))
# {2, 3, 5}
# >>> menor(inserta(5, inserta(3, inserta(2, inserta(5, vacio()))))
# 2
# >>> elimina(5, inserta(5, inserta(3, inserta(2, inserta(5, vacio()))))
# {2, 3}
```

```
# >>> pertenece(5, inserta(5, inserta(3, inserta(2, inserta(5, vacio())))))
# True
# >>> pertenece(1, inserta(5, inserta(3, inserta(2, inserta(5, vacio())))))
# False
# >>> esVacio(inserta(5, inserta(3, inserta(2, inserta(5, vacio())))))
# False
# >>> esVacio(vacio())
# True
# >>> inserta(5, inserta(2, vacio())) == inserta(2, inserta(5, (inserta(2, vacio()))))
# True
#
# Finalmente, se define un generador aleatorio de conjuntos y se
# comprueba que los conjuntos cumplen las propiedades de su
# especificación.
```

```
from __future__ import annotations
```

```
__all__ = [
    'Conj',
    'vacio',
    'inserta',
    'menor',
    'elimina',
    'pertenece',
    'esVacio',
    'conjuntoAleatorio'
]
```

```
from abc import abstractmethod
from copy import deepcopy
from dataclasses import dataclass, field
from typing import Any, Generic, Protocol, TypeVar
```

```
from hypothesis import given
from hypothesis import strategies as st
```

```
class Comparable(Protocol):
    @abstractmethod
    def __lt__(self: A, otro: A) -> bool:
```



```

    pass

A = TypeVar('A', bound=Comparable)

# Clase de los conjuntos mediante listas no ordenadas sin duplicados
# =====

@dataclass
class Conj(Generic[A]):
    _elementos: list[A] = field(default_factory=list)

    def __repr__(self) -> str:
        """
        Devuelve una cadena con los elementos del conjunto entre llaves
        y separados por ", ".
        """
        return '{' + ', '.join(str(x) for x in sorted(self._elementos)) + '}'

    def __eq__(self, c: Any) -> bool:
        """
        Se verifica si el conjunto es igual a c; es decir, tienen los
        mismos elementos sin importar el orden ni las repeticiones.
        """
        return sorted(self._elementos) == sorted(c._elementos)

    def inserta(self, x: A) -> None:
        """
        Añade el elemento x al conjunto.
        """
        if x not in self._elementos:
            self._elementos.append(x)

    def menor(self) -> A:
        """
        Devuelve el menor elemento del conjunto
        """
        return min(self._elementos)

    def elimina(self, x: A) -> None:
        """

```

```

        Elimina el elemento x del conjunto.
        """
        if x in self._elementos:
            self._elementos.remove(x)

    def esVacio(self) -> bool:
        """
        Se verifica si el conjunto está vacío.
        """
        return not self._elementos

    def pertenece(self, x: A) -> bool:
        """
        Se verifica si x pertenece al conjunto.
        """
        return x in self._elementos

# Funciones del tipo conjunto
# =====

def vacio() -> Conj[A]:
    """
    Crea y devuelve un conjunto vacío de tipo A.
    """
    c: Conj[A] = Conj()
    return c

def inserta(x: A, c: Conj[A]) -> Conj[A]:
    """
    Inserta un elemento x en el conjunto c y devuelve un nuevo conjunto
    con el elemento insertado.
    """
    _aux = deepcopy(c)
    _aux.inserta(x)
    return _aux

def menor(c: Conj[A]) -> A:
    """
    Devuelve el menor elemento del conjunto c.
    """

```

```

    return c.menor()

def elimina(x: A, c: Conj[A]) -> Conj[A]:
    """
    Elimina las ocurrencias de c en c y devuelve una copia del conjunto
    resultante.
    """
    _aux = deepcopy(c)
    _aux.elimina(x)
    return _aux

def pertenece(x: A, c: Conj[A]) -> bool:
    """
    Se verifica si x pertenece a c.
    """
    return c.pertenece(x)

def esVacio(c: Conj[A]) -> bool:
    """
    Se verifica si el conjunto está vacío.
    """
    return c.esVacio()

# Generador de conjuntos
# =====

def sin_duplicados(xs: list[int]) -> list[int]:
    return list(set(xs))

def conjuntoAleatorio() -> st.SearchStrategy[Conj[int]]:
    """
    Estrategia de búsqueda para generar conjuntos de enteros de forma
    aleatoria.
    """
    xs = st.lists(st.integers()).map(sin_duplicados)
    return xs.map(Conj)

# Comprobación de las propiedades de los conjuntos
# =====

```

```

# Las propiedades son
@given(c=conjuntoAleatorio(), x=st.integers(), y=st.integers())
def test_conjuntos(c: Conj[int], x: int, y: int) -> None:
    assert inserta(x, inserta(x, c)) == inserta(x, c)
    assert inserta(x, inserta(y, c)) == inserta(y, inserta(x, c))
    v: Conj[int] = vacio()
    assert not pertenece(x, v)
    assert pertenece(y, inserta(x, c)) == (x == y) or pertenece(y, c)
    assert elimina(x, v) == v

    def relacion(x: int, y: int, c: Conj[int]) -> Conj[int]:
        if x == y:
            return elimina(x, c)
        return inserta(y, elimina(x, c))

    assert elimina(x, inserta(y, c)) == relacion(x, y, c)
    assert esVacio(vacio())
    assert not esVacio(inserta(x, c))

# La comprobación es
# > poetry run pytest -q conjuntoConListasNoOrdenadasSinDuplicados.py
# 1 passed in 0.26s

```

8.4. El tipo de datos de los conjuntos mediante listas ordenadas sin duplicados

8.4.1. En Haskell

```

{-# OPTIONS_GHC -fno-warn-unused-top-binds #-}

module TAD.ConjuntoConListasOrdenadasSinDuplicados
  (Conj,
   vacio,      -- Conj a
   inserta,    -- Ord a => a -> Conj a -> Conj a
   menor,      -- Ord a => Conj a -> a
   elimina,    -- Ord a => a -> Conj a -> Conj a
   pertenece,  -- Ord a => a -> Conj a -> Bool
   esVacio     -- Conj a -> Bool
  ) where

```

```
import Data.List (intercalate)
import Test.QuickCheck

-- Los conjuntos como listas ordenadas sin repeticiones.
newtype Conj a = Cj [a]
    deriving Eq

-- (escribeConjunto c) es la cadena correspondiente al conjunto c. Por
-- ejemplo,
--   λ> escribeConjunto (Cj [])
--   "{}"
--   λ> escribeConjunto (Cj [5])
--   "{5}"
--   λ> escribeConjunto (Cj [2, 5])
--   "{2, 5}"
--   λ> escribeConjunto (Cj [5, 2])
--   "{2, 5}"
escribeConjunto :: Show a => Conj a -> String
escribeConjunto (Cj xs) =
    "{" ++ intercalate ", " (map show xs) ++ "}"

-- Procedimiento de escritura de conjuntos.
instance Show a => Show (Conj a) where
    show = escribeConjunto

-- vacio es el conjunto vacío. Por ejemplo,
--   λ> vacio
--   {}
vacio :: Conj a
vacio = Cj []

-- (inserta x c) es el conjunto obtenido añadiendo el elemento x al
-- conjunto c. Por ejemplo,
--   λ> inserta 5 vacio
--   {5}
--   λ> inserta 2 (inserta 5 vacio)
--   {2, 5}
--   λ> inserta 5 (inserta 2 vacio)
--   {2, 5}
```

```

inserta :: Ord a => a -> Conj a -> Conj a
inserta x (Cj s) = Cj (agrega x s)
  where agrega z [] = [z]
        agrega z s'@(y:ys) | z > y = y : agrega z ys
                           | z < y = z : s'
                           | otherwise = s'

-- (menor c) es el menor elemento del conjunto c. Por ejemplo,
--   λ> menor (inserta 5 (inserta 2 vacio))
--   2
menor :: Ord a => Conj a -> a
menor (Cj []) = error "conjunto vacío"
menor (Cj (x:_)) = x

-- (elimina x c) es el conjunto obtenido eliminando el elemento x
-- del conjunto c. Por ejemplo,
--   λ> elimina 2 (inserta 5 (inserta 2 vacio))
--   {5}
elimina :: Ord a => a -> Conj a -> Conj a
elimina x (Cj s) = Cj (elimina' x s)
  where elimina' _ [] = []
        elimina' z s'@(y:ys) | z > y = y : elimina' z ys
                           | z < y = s'
                           | otherwise = ys

-- (esVacio c) se verifica si c es el conjunto vacío. Por ejemplo,
--   λ> esVacio (inserta 5 (inserta 2 vacio))
--   False
--   λ> esVacio vacio
--   True
esVacio :: Conj a -> Bool
esVacio (Cj xs) = null xs

-- (pertenece x c) se verifica si x pertenece al conjunto c. Por ejemplo,
--   λ> pertenece 2 (inserta 5 (inserta 2 vacio))
--   True
--   λ> pertenece 4 (inserta 5 (inserta 2 vacio))
--   False
pertenece :: Ord a => a -> Conj a -> Bool
pertenece x (Cj s) = x `elem` takeWhile (<= x) s

```

```

-- Generador de conjuntos                                     --
-- =====

-- genConjunto es un generador de conjuntos. Por ejemplo,
--   λ> sample (genConjunto :: Gen (Conj Int))
--   {}
--   {1}
--   {2}
--   {}
--   {-5, -1}
--   {-9, -8, 2, 3, 10}
--   {4}
--   {-13, -7, 1, 14}
--   {-12, -10, -9, -4, 1, 2, 5, 14, 16}
--   {-18, -15, -14, -13, -10, -7, -6, -4, -1, 1, 10, 11, 12, 16}
--   {-16, -9, -6, -5, -4, -2, 3, 6, 9, 13, 17}
genConjunto :: (Arbitrary a, Ord a) => Gen (Conj a)
genConjunto = do
  xs <- listOf arbitrary
  return (foldr inserta vacio xs)

-- Los conjuntos son concreciones de los arbitrarios.
instance (Arbitrary a, Ord a) => Arbitrary (Conj a) where
  arbitrary = genConjunto

-- Propiedades de los conjuntos                                --
-- =====

prop_conjuntos :: Int -> Int -> Conj Int -> Bool
prop_conjuntos x y c =
  inserta x (inserta x c) == inserta x c &&
  inserta x (inserta y c) == inserta y (inserta x c) &&
  not (pertenece x vacio) &&
  pertenece y (inserta x c) == (x == y) || pertenece y c &&
  elimina x vacio == vacio &&
  elimina x (inserta y c) == (if x == y
                              then elimina x c
                              else inserta y (elimina x c)) &&
  esVacio (vacio :: Conj Int) &&

```

```
not (esVacio (inserta x c))
```

```
-- Comprobación
--   λ> quickCheck prop_conjuntos
--   +++ OK, passed 100 tests.
```

8.4.2. En Python

```
# Se define la clase Conj con los siguientes métodos:
#   + inserta(x) añade x al conjunto.
#   + menor() es el menor elemento del conjunto.
#   + elimina(x) elimina las ocurrencias de x en el conjunto.
#   + pertenece(x) se verifica si x pertenece al conjunto.
#   + esVacia() se verifica si la cola es vacía.
# Por ejemplo,
#   >>> c = Conj()
#   >>> c
#   {}
#   >>> c.inserta(5)
#   >>> c.inserta(2)
#   >>> c.inserta(3)
#   >>> c.inserta(4)
#   >>> c.inserta(5)
#   >>> c
#   {2, 3, 4, 5}
#   >>> c.menor()
#   2
#   >>> c.elimina(3)
#   >>> c
#   {2, 4, 5}
#   >>> c.pertenece(4)
#   True
#   >>> c.pertenece(3)
#   False
#   >>> c.esVacio()
#   False
#   >>> c = Conj()
#   >>> c.esVacio()
#   True
#   >>> c = Conj()
```



```

#     >>> c.inserta(2)
#     >>> c.inserta(5)
#     >>> d = Conj()
#     >>> d.inserta(5)
#     >>> d.inserta(2)
#     >>> d.inserta(5)
#     >>> c == d
#     True
#
# Además se definen las correspondientes funciones. Por ejemplo,
#     >>> vacio()
#     {}
#     >>> inserta(5, inserta(3, inserta(2, inserta(5, vacio()))))
#     {2, 3, 5}
#     >>> menor(inserta(5, inserta(3, inserta(2, inserta(5, vacio()))))
#     2
#     >>> elimina(5, inserta(5, inserta(3, inserta(2, inserta(5, vacio()))))
#     {2, 3}
#     >>> pertenece(5, inserta(5, inserta(3, inserta(2, inserta(5, vacio()))))
#     True
#     >>> pertenece(1, inserta(5, inserta(3, inserta(2, inserta(5, vacio()))))
#     False
#     >>> esVacio(inserta(5, inserta(3, inserta(2, inserta(5, vacio()))))
#     False
#     >>> esVacio(vacio())
#     True
#     >>> inserta(5, inserta(2, vacio())) == inserta(2, inserta(5, (inserta(2, vacio())))
#     True
#
# Finalmente, se define un generador aleatorio de conjuntos y se
# comprueba que los conjuntos cumplen las propiedades de su
# especificación.

```

```

from __future__ import annotations

```

```

__all__ = [
    'Conj',
    'vacio',
    'inserta',
    'menor',

```

```

        'elimina',
        'pertenece',
        'esVacio',
        'conjuntoAleatorio'
    ]

    from abc import abstractmethod
    from bisect import bisect_left, insort_left
    from copy import deepcopy
    from dataclasses import dataclass, field
    from itertools import takewhile
    from typing import Generic, Protocol, TypeVar

    from hypothesis import given
    from hypothesis import strategies as st

    class Comparable(Protocol):
        @abstractmethod
        def __lt__(self: A, otro: A) -> bool:
            pass

    A = TypeVar('A', bound=Comparable)

    # Clase de los conjuntos mediante listas ordenadas sin duplicados
    # =====

    @dataclass
    class Conj(Generic[A]):
        _elementos: list[A] = field(default_factory=list)

        def __repr__(self) -> str:
            """
            Devuelve una cadena con los elementos del conjunto entre llaves
            y separados por ", ".
            """
            return '{' + ', '.join(str(x) for x in self._elementos) + '}'

        def inserta(self, x: A) -> None:
            """

```

```

    Añade el elemento x al conjunto.
    """
    if x not in self._elementos:
        insort_left(self._elementos, x)

def menor(self) -> A:
    """
    Devuelve el menor elemento del conjunto
    """
    return self._elementos[0]

def elimina(self, x: A) -> None:
    """
    Elimina el elemento x del conjunto.
    """
    pos = bisect_left(self._elementos, x)
    if pos < len(self._elementos) and self._elementos[pos] == x:
        self._elementos.pop(pos)

def esVacio(self) -> bool:
    """
    Se verifica si el conjunto está vacío.
    """
    return not self._elementos

def pertenece(self, x: A) -> bool:
    """
    Se verifica si x pertenece al conjunto.
    """
    return x in takewhile(lambda y: y < x or y == x, self._elementos)

# Funciones del tipo conjunto
# =====

def vacio() -> Conj[A]:
    """
    Crea y devuelve un conjunto vacío de tipo A.
    """
    c: Conj[A] = Conj()
    return c

```

```
def inserta(x: A, c: Conj[A]) -> Conj[A]:
    """
    Inserta un elemento x en el conjunto c y devuelve un nuevo conjunto
    con el elemento insertado.
    """
    _aux = deepcopy(c)
    _aux.inserta(x)
    return _aux

def menor(c: Conj[A]) -> A:
    """
    Devuelve el menor elemento del conjunto c.
    """
    return c.menor()

def elimina(x: A, c: Conj[A]) -> Conj[A]:
    """
    Elimina las ocurrencias de c en c y devuelve una copia del conjunto
    resultante.
    """
    _aux = deepcopy(c)
    _aux.elimina(x)
    return _aux

def pertenece(x: A, c: Conj[A]) -> bool:
    """
    Se verifica si x pertenece a c.
    """
    return c.pertenece(x)

def esVacio(c: Conj[A]) -> bool:
    """
    Se verifica si el conjunto está vacío.
    """
    return c.esVacio()

# Generador de conjuntos
# =====
```

```

def sin_duplicados_y_ordenado(xs: list[int]) -> list[int]:
    xs = list(set(xs))
    xs.sort()
    return xs

def conjuntoAleatorio() -> st.SearchStrategy[Conj[int]]:
    """
    Estrategia de búsqueda para generar conjuntos de enteros de forma
    aleatoria.
    """
    xs = st.lists(st.integers()).map(sin_duplicados_y_ordenado)
    return xs.map(Conj)

# Comprobación de las propiedades de los conjuntos
# =====

# Las propiedades son
@given(c=conjuntoAleatorio(), x=st.integers(), y=st.integers())
def test_conjuntos(c: Conj[int], x: int, y: int) -> None:
    assert inserta(x, inserta(x, c)) == inserta(x, c)
    assert inserta(x, inserta(y, c)) == inserta(y, inserta(x, c))
    v: Conj[int] = vacio()
    assert not pertenece(x, v)
    assert pertenece(y, inserta(x, c)) == (x == y) or pertenece(y, c)
    assert elimina(x, v) == v

    def relacion(x: int, y: int, c: Conj[int]) -> Conj[int]:
        if x == y:
            return elimina(x, c)
        return inserta(y, elimina(x, c))

    assert elimina(x, inserta(y, c)) == relacion(x, y, c)
    assert esVacio(vacio())
    assert not esVacio(inserta(x, c))

# La comprobación es
# > poetry run pytest -q conjuntoConListasOrdenadasSinDuplicados.py
# 1 passed in 0.13s

```

8.5. El tipo de datos de los conjuntos mediante librería

8.5.1. En Haskell

```
{-# OPTIONS_GHC -fno-warn-unused-top-binds #-}
{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module TAD.ConjuntoConLibreria
  ( Conj,
    vacio,      -- Conj a
    inserta,    -- Ord a => a -> Conj a -> Conj a
    menor,      -- Ord a => Conj a -> a
    elimina,    -- Ord a => a -> Conj a -> Conj a
    pertenece,  -- Ord a => a -> Conj a -> Bool
    esVacio     -- Conj a -> Bool
  ) where

import Data.Set as S (Set, empty, insert, findMin, delete, member, null,
                      fromList, toList)
import Data.List (intercalate)
import Test.QuickCheck

-- Los conjuntos como conjuntos de la librería Data.Set
newtype Conj a = Cj (Set a)
  deriving (Eq, Ord)

-- (escribeConjunto c) es la cadena correspondiente al conjunto c. Por
-- ejemplo,
--   λ> escribeConjunto (Cj (fromList []))
--   "{}"
--   λ> escribeConjunto (Cj (fromList [5]))
--   "{5}"
--   λ> escribeConjunto (Cj (fromList [2, 5]))
--   "{2, 5}"
--   λ> escribeConjunto (Cj (fromList [5, 2]))
--   "{2, 5}"
escribeConjunto :: Show a => Conj a -> String
escribeConjunto (Cj s) =
  "{" ++ intercalate ", " (map show (toList s)) ++ "}"
```

```

-- Procedimiento de escritura de conjuntos.
instance Show a => Show (Cj a) where
    show = escribeConjunto

-- vacio es el conjunto vacío. Por ejemplo,
--     λ> vacio
--     {}
vacio :: Cj a
vacio = Cj empty

-- (inserta x c) es el conjunto obtenido añadiendo el elemento x al
-- conjunto c. Por ejemplo,
--     λ> inserta 5 vacio
--     {5}
--     λ> inserta 2 (inserta 5 vacio)
--     {2, 5}
--     λ> inserta 5 (inserta 2 vacio)
--     {2, 5}
inserta :: Ord a => a -> Cj a -> Cj a
inserta x (Cj s) = Cj (insert x s)

-- (menor c) es el menor elemento del conjunto c. Por ejemplo,
--     λ> menor (inserta 5 (inserta 2 vacio))
--     2
menor :: Ord a => Cj a -> a
menor (Cj s) = findMin s

-- (elimina x c) es el conjunto obtenido eliminando el elemento x
-- del conjunto c. Por ejemplo,
--     λ> elimina 2 (inserta 5 (inserta 2 vacio))
--     {5}
elimina :: Ord a => a -> Cj a -> Cj a
elimina x (Cj s) = Cj (delete x s)

-- (esVacio c) se verifica si c es el conjunto vacío. Por ejemplo,
--     λ> esVacio (inserta 5 (inserta 2 vacio))
--     False
--     λ> esVacio vacio
--     True

```

```

esVacio :: Conj a -> Bool
esVacio (Cj s) = S.null s

-- (pertenece x c) se verifica si x pertenece al conjunto c. Por ejemplo,
--   λ> pertenece 2 (inserta 5 (inserta 2 vacio))
--   True
--   λ> pertenece 4 (inserta 5 (inserta 2 vacio))
--   False
pertenece :: Ord a => a -> Conj a -> Bool
pertenece x (Cj s) = member x s

-- Generador de conjuntos
-- =====

-- genConjunto es un generador de conjuntos. Por ejemplo,
--   λ> sample (genConjunto :: Gen (Conj Int))
--   {}
--   {-2, 0}
--   {-1, 3}
--   {-3, 2}
--   {-5, -4, -3, 2, 4, 6, 7}
--   {-4, 4}
--   {-9, -6, -3, 1, 5, 11, 12}
--   {-10, -8, -7, -3, 1, 2, 8, 9, 10, 13}
--   {-13, -8, -7, -6, -1, 0, 1, 6, 7, 9, 11, 14, 16}
--   {-15, -12, -9, 1, 2, 9, 13, 15, 16, 18}
--   {-16}
genConjunto :: (Arbitrary a, Ord a) => Gen (Conj a)
genConjunto = do
  xs <- listOf arbitrary
  return (Cj (fromList xs))

-- Los conjuntos son concreciones de los arbitrarios.
instance (Arbitrary a, Ord a) => Arbitrary (Conj a) where
  arbitrary = genConjunto

-- Propiedades de los conjuntos
-- =====

prop_conjuntos :: Int -> Int -> Conj Int -> Bool

```



```

prop_conjuntos x y c =
  inserta x (inserta x c) == inserta x c &&
  inserta x (inserta y c) == inserta y (inserta x c) &&
  not (pertenece x vacio) &&
  pertenece y (inserta x c) == (x == y) || pertenece y c &&
  elimina x vacio == vacio &&
  elimina x (inserta y c) == (if x == y
                              then elimina x c
                              else inserta y (elimina x c)) &&
  esVacio (vacio :: Conj Int) &&
  not (esVacio (inserta x c))

-- Comprobación
--   λ> quickCheck prop_conjuntos
--   +++ OK, passed 100 tests.

```

8.5.2. En Python

```

# Se define la clase Conj con los siguientes métodos:
#   + inserta(x) añade x al conjunto.
#   + menor() es el menor elemento del conjunto.
#   + elimina(x) elimina las ocurrencias de x en el conjunto.
#   + pertenece(x) se verifica si x pertenece al conjunto.
#   + esVacia() se verifica si la cola es vacía.
# Por ejemplo,
#   >>> c = Conj()
#   >>> c
#   {}
#   >>> c.inserta(5)
#   >>> c.inserta(2)
#   >>> c.inserta(3)
#   >>> c.inserta(4)
#   >>> c.inserta(5)
#   >>> c
#   {2, 3, 4, 5}
#   >>> c.menor()
#   2
#   >>> c.elimina(3)
#   >>> c
#   {2, 4, 5}

```

```

#     >>> c.pertenece(4)
#     True
#     >>> c.pertenece(3)
#     False
#     >>> c.esVacio()
#     False
#     >>> c = Conj()
#     >>> c.esVacio()
#     True
#     >>> c = Conj()
#     >>> c.inserta(2)
#     >>> c.inserta(5)
#     >>> d = Conj()
#     >>> d.inserta(5)
#     >>> d.inserta(2)
#     >>> d.inserta(5)
#     >>> c == d
#     True
#
# Además se definen las correspondientes funciones. Por ejemplo,
#     >>> vacio()
#     {}
#     >>> inserta(5, inserta(3, inserta(2, inserta(5, vacio()))))
#     {2, 3, 5}
#     >>> menor(inserta(5, inserta(3, inserta(2, inserta(5, vacio()))))
#     2
#     >>> elimina(5, inserta(5, inserta(3, inserta(2, inserta(5, vacio()))))
#     {2, 3}
#     >>> pertenece(5, inserta(5, inserta(3, inserta(2, inserta(5, vacio()))))
#     True
#     >>> pertenece(1, inserta(5, inserta(3, inserta(2, inserta(5, vacio()))))
#     False
#     >>> esVacio(inserta(5, inserta(3, inserta(2, inserta(5, vacio()))))
#     False
#     >>> esVacio(vacio())
#     True
#     >>> inserta(5, inserta(2, vacio())) == inserta(2, inserta(5, (inserta(2, vacio())))
#     True
#
# Finalmente, se define un generador aleatorio de conjuntos y se

```

```
# comprueba que los conjuntos cumplen las propiedades de su
# especificación.
```

```
from __future__ import annotations
```

```
__all__ = [
    'Conj',
    'vacio',
    'inserta',
    'menor',
    'elimina',
    'pertenece',
    'esVacio',
    'conjuntoAleatorio'
]
```

```
from abc import abstractmethod
from copy import deepcopy
from dataclasses import dataclass, field
from typing import Generic, Protocol, TypeVar
```

```
from hypothesis import given
from hypothesis import strategies as st
```

```
class Comparable(Protocol):
    @abstractmethod
    def __lt__(self: A, otro: A) -> bool:
        pass
```

```
A = TypeVar('A', bound=Comparable)
```

```
# Clase de los conjuntos mediante librería
# =====
```

```
@dataclass
class Conj(Generic[A]):
    _elementos: set[A] = field(default_factory=set)

    def __repr__(self) -> str:
```

```

        xs = [str(x) for x in self._elementos]
        return "{" + ", ".join(xs) + "}"

    def inserta(self, x: A) -> None:
        """
        Añade el elemento x al conjunto.
        """
        self._elementos.add(x)

    def menor(self) -> A:
        """
        Devuelve el menor elemento del conjunto
        """
        return min(self._elementos)

    def elimina(self, x: A) -> None:
        """
        Elimina el elemento x del conjunto.
        """
        self._elementos.discard(x)

    def esVacio(self) -> bool:
        """
        Se verifica si el conjunto está vacío.
        """
        return not self._elementos

    def pertenece(self, x: A) -> bool:
        """
        Se verifica si x pertenece al conjunto.
        """
        return x in self._elementos

# Funciones del tipo conjunto
# =====

def vacio() -> Conj[A]:
    """
    Crea y devuelve un conjunto vacío de tipo A.
    """

```

```
c: Conj[A] = Conj()
return c

def inserta(x: A, c: Conj[A]) -> Conj[A]:
    """
    Inserta un elemento x en el conjunto c y devuelve un nuevo conjunto
    con el elemento insertado.
    """
    _aux = deepcopy(c)
    _aux.inserta(x)
    return _aux

def menor(c: Conj[A]) -> A:
    """
    Devuelve el menor elemento del conjunto c.
    """
    return c.menor()

def elimina(x: A, c: Conj[A]) -> Conj[A]:
    """
    Elimina las ocurrencias de x en c y devuelve una copia del conjunto
    resultante.
    """
    _aux = deepcopy(c)
    _aux.elimina(x)
    return _aux

def pertenece(x: A, c: Conj[A]) -> bool:
    """
    Se verifica si x pertenece a c.
    """
    return c.pertenece(x)

def esVacio(c: Conj[A]) -> bool:
    """
    Se verifica si el conjunto está vacío.
    """
    return c.esVacio()

# Generador de conjuntos
```

```
# =====

def conjuntoAleatorio() -> st.SearchStrategy[Conj[int]]:
    """
    Estrategia de búsqueda para generar conjuntos de enteros de forma
    aleatoria.
    """
    return st.builds(Conj, st.lists(st.integers()).map(set))

# Comprobación de las propiedades de los conjuntos
# =====

# Las propiedades son
@given(c=conjuntoAleatorio(), x=st.integers(), y=st.integers())
def test_conjuntos(c: Conj[int], x: int, y: int) -> None:
    assert inserta(x, inserta(x, c)) == inserta(x, c)
    assert inserta(x, inserta(y, c)) == inserta(y, inserta(x, c))
    v: Conj[int] = vacio()
    assert not pertenece(x, v)
    assert pertenece(y, inserta(x, c)) == (x == y) or pertenece(y, c)
    assert elimina(x, v) == v

    def relacion(x: int, y: int, c: Conj[int]) -> Conj[int]:
        if x == y:
            return elimina(x, c)
        return inserta(y, elimina(x, c))

    assert elimina(x, inserta(y, c)) == relacion(x, y, c)
    assert esVacio(vacio())
    assert not esVacio(inserta(x, c))

# La comprobación es
# > poetry run pytest -q conjuntoConLibreria.py
# 1 passed in 0.22s
```

8.6. Transformaciones entre conjuntos y listas

8.6.1. En Haskell

```

-----
-- Utilizando el [tipo abstracto de datos de los conjuntos](https://bit.ly/3HbB7f
-- definir las funciones
--     listaAconjunto :: [a] -> Conj a
--     conjuntoAlista :: Conj a -> [a]
-- tales que
-- + (listaAconjunto xs) es el conjunto formado por los elementos de xs.
--   Por ejemplo,
--     λ> listaAconjunto [3, 2, 5]
--     {2, 3, 5}
-- + (conjuntoAlista c) es la lista formada por los elementos del
--   conjunto c. Por ejemplo,
--     λ> conjuntoAlista (inserta 5 (inserta 2 (inserta 3 vacio)))
--     [2,3,5]
--
-- Comprobar con QuickCheck que ambas funciones son inversa; es decir,
--     conjuntoAlista (listaAconjunto xs) = sort (nub xs)
--     listaAconjunto (conjuntoAlista c) = c
-----

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module TAD_Transformaciones_conjuntos_listas where

import TAD.Conjunto (Conj, vacio, inserta, menor, elimina, pertenece, esVacio)
import Data.List (sort, nub)
import Test.QuickCheck

-- 1ª definición de listaAconjunto
-- =====

listaAconjunto :: Ord a => [a] -> Conj a
listaAconjunto [] = vacio
listaAconjunto (x:xs) = inserta x (listaAconjunto xs)

-- 2ª definición de listaAconjunto
-- =====

```

```

listaAconjunto2 :: Ord a => [a] -> Conj a
listaAconjunto2 = foldr inserta vacio

-- Comprobación de equivalencia
-- =====

-- La propiedad es
prop_listaAconjunto :: [Int] -> Bool
prop_listaAconjunto xs =
    listaAconjunto xs == listaAconjunto2 xs

-- La comprobación es
--    λ> quickCheck prop_listaAconjunto
--    +++ OK, passed 100 tests.

-- Definición de conjuntoAlista
-- =====

conjuntoAlista :: Ord a => Conj a -> [a]
conjuntoAlista c
    | esVacio c = []
    | otherwise = mc : conjuntoAlista rc
    where mc = menor c
          rc = elimina mc c

-- Comprobación de las propiedades
-- =====

-- La primera propiedad es
prop_1_listaAconjunto :: [Int] -> Bool
prop_1_listaAconjunto xs =
    conjuntoAlista (listaAconjunto xs) == sort (nub xs)

-- La comprobación es
--    λ> quickCheck prop_1_listaAconjunto
--    +++ OK, passed 100 tests.

-- La segunda propiedad es
prop_2_listaAconjunto :: Conj Int -> Bool

```



```
prop_2_listaAconjunto c =
  listaAconjunto (conjuntoAlista c) == c

-- La comprobación es
--   λ> quickCheck prop_2_listaAconjunto
--   +++ OK, passed 100 tests.
```

8.6.2. En Python

```
# -----
# Utilizando el [tipo abstracto de datos de los conjuntos](https://bit.ly/3HbB7fo)
# definir las funciones
#   listaAconjunto : (list[A]) -> Conj[A]
#   conjuntoAlista : (Conj[A]) -> list[A]
# tales que
# + listaAconjunto(xs) es el conjunto formado por los elementos de xs.
#   Por ejemplo,
#   >>> listaAconjunto([3, 2, 5])
#   {2, 3, 5}
# + conjuntoAlista(c) es la lista formada por los elementos del
#   conjunto c. Por ejemplo,
#   >>> conjuntoAlista(inserta(5, inserta(2, inserta(3, vacio()))))
#   [2, 3, 5]
#
# Comprobar con Hypothesis que ambas funciones son inversa; es decir,
#   conjuntoAlista (listaAconjunto xs) = sorted(list(set(xs)))
#   listaAconjunto (conjuntoAlista c) = c
# -----

from __future__ import annotations

from abc import abstractmethod
from copy import deepcopy
from functools import reduce
from typing import Protocol, TypeVar

from hypothesis import given
from hypothesis import strategies as st

from src.TAD.conjunto import (Conj, conjuntoAleatorio, elimina, esVacio,
```

inserta, menor, pertenece, vacio)

```

class Comparable(Protocol):
    @abstractmethod
    def __lt__(self: A, otro: A) -> bool:
        pass

A = TypeVar('A', bound=Comparable)

# 1ª definición de listaAconjunto
# =====

def listaAconjunto(xs: list[A]) -> Conj[A]:
    if not xs:
        return vacio()
    return inserta(xs[0], listaAconjunto(xs[1:]))

# 2ª definición de listaAconjunto
# =====

def listaAconjunto2(xs: list[A]) -> Conj[A]:
    return reduce(lambda ys, y: inserta(y, ys), xs, vacio())

# 3ª solución de listaAconjunto
# =====

def listaAconjunto3(xs: list[A]) -> Conj[A]:
    c: Conj[A] = Conj()
    for x in xs:
        c.inserta(x)
    return c

# Comprobación de equivalencia
# =====

# La propiedad es
@given(st.lists(st.integers()))
def test_listaAconjunto(xs: list[int]) -> None:
    r = listaAconjunto(xs)

```

```

    assert listaAconjunto2(xs) == r
    assert listaAconjunto3(xs) == r

# 1ª definición de conjuntoAlista
# =====

def conjuntoAlista(c: Conj[A]) -> list[A]:
    if esVacio(c):
        return []
    mc = menor(c)
    rc = elimina(mc, c)
    return [mc] + conjuntoAlista(rc)

# 2ª definición de conjuntoAlista
# =====

def conjuntoAlista2Aux(c: Conj[A]) -> list[A]:
    if c.esVacio():
        return []
    mc = c.menor()
    c.elimina(mc)
    return [mc] + conjuntoAlista2Aux(c)

def conjuntoAlista2(c: Conj[A]) -> list[A]:
    c1 = deepcopy(c)
    return conjuntoAlista2Aux(c1)

# 3ª definición de conjuntoAlista
# =====

def conjuntoAlista3Aux(c: Conj[A]) -> list[A]:
    r = []
    while not c.esVacio():
        mc = c.menor()
        r.append(mc)
        c.elimina(mc)
    return r

def conjuntoAlista3(c: Conj[A]) -> list[A]:
    c1 = deepcopy(c)

```

```

    return conjuntoAlista3Aux(c1)

# Comprobación de equivalencia de las definiciones de conjuntoAlista
# =====

@given(c=conjuntoAleatorio())
def test_conjuntoAlista(c: Conj[int]) -> None:
    r = conjuntoAlista(c)
    assert conjuntoAlista2(c) == r
    assert conjuntoAlista3(c) == r

# Comprobación de las propiedades
# =====

# La primera propiedad es
@given(st.lists(st.integers()))
def test_1_listaAconjunto(xs: list[int]) -> None:
    assert conjuntoAlista(listaAconjunto(xs)) == sorted(list(set(xs)))

# La segunda propiedad es
@given(c=conjuntoAleatorio())
def test_2_listaAconjunto(c: Conj[int]) -> None:
    assert listaAconjunto(conjuntoAlista(c)) == c

# La comprobación de las propiedades es
# > poetry run pytest -v TAD_Transformaciones_conjuntos_listas.py
#     test_listaAconjunto PASSED
#     test_conjuntoAlista PASSED
#     test_1_listaAconjunto PASSED
#     test_2_listaAconjunto PASSED

```

8.7. Reconocimiento de subconjunto

8.7.1. En Haskell

```

-- -----
-- Utilizando el [tipo abstracto de datos de los conjuntos](https://bit.ly/3HbB7f
-- definir la función
--     subconjunto :: Ord a => Conj a -> Conj a -> Bool
-- tal que (subconjunto c1 c2) se verifica si todos los elementos de c1

```

```

-- pertenecen a c2. Por ejemplo,
--   λ> ej1 = inserta 5 (inserta 2 vacio)
--   λ> ej2 = inserta 3 (inserta 2 (inserta 5 vacio))
--   λ> ej3 = inserta 3 (inserta 4 (inserta 5 vacio))
--   λ> subconjunto ej1 ej2
--   True
--   λ> subconjunto ej1 ej3
--   False
-- -----

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module TAD_subconjunto where

import TAD.Conjunto (Conj, vacio, inserta, menor, elimina, pertenece, esVacio)
import TAD.Transformaciones_conjuntos_listas (conjuntoAlista)
import Test.QuickCheck

-- 1ª solución
-- =====

subconjunto :: Ord a => Conj a -> Conj a -> Bool
subconjunto c1 c2
  | esVacio c1 = True
  | otherwise  = pertenece mcl c2 && subconjunto rc1 c2
  where mcl = menor c1
        rc1 = elimina mcl c1

-- 2ª solución
-- =====

subconjunto2 :: Ord a => Conj a -> Conj a -> Bool
subconjunto2 c1 c2 =
  and [pertenece x c2 | x <- conjuntoAlista c1]

-- La función conjuntoAlista está definida en el ejercicio
-- "Transformaciones entre conjuntos y listas" que se encuentra en
-- https://bit.ly/3RexzxH

-- 3ª solución

```

```

-- =====

subconjunto3 :: Ord a => Conj a -> Conj a -> Bool
subconjunto3 c1 c2 =
  all (`pertenece` c2) (conjuntoAlista c1)

-- 4ª solución
-- =====

subconjunto4 :: Ord a => Conj a -> Conj a -> Bool
subconjunto4 c1 c2 =
  sublista (conjuntoAlista c1) (conjuntoAlista c2)

-- (sublista xs ys) se verifica si xs es una sublista de ys. Por
-- ejemplo,
--   sublista [5, 2] [3, 2, 5] == True
--   sublista [5, 2] [3, 4, 5] == False
sublista :: Ord a => [a] -> [a] -> Bool
sublista [] _ = True
sublista (x:xs) ys = elem x ys && sublista xs ys

-- Comprobación de equivalencia
-- =====

-- La propiedad es
prop_subconjunto :: Conj Int -> Conj Int -> Bool
prop_subconjunto c1 c2 =
  all (== subconjunto c1 c2)
    [subconjunto2 c1 c2,
     subconjunto3 c1 c2,
     subconjunto4 c1 c2]

-- La comprobación es
--   λ> quickCheck prop_subconjunto
--   +++ OK, passed 100 tests.

```

8.7.2. En Python

```

# -----
# Utilizando el [tipo abstracto de datos de los conjuntos](https://bit.ly/3HbB7fo

```

```

# definir la función
#     subconjunto :: Ord a => Conj a -> Conj a -> Bool
# tal que (subconjunto c1 c2) se verifica si todos los elementos de c1
# pertenecen a c2. Por ejemplo,
#     >>> ej1 = inserta(5, inserta(2, vacio()))
#     >>> ej2 = inserta(3, inserta(2, inserta(5, vacio())))
#     >>> ej3 = inserta(3, inserta(4, inserta(5, vacio())))
#     >>> subconjunto(ej1, ej2)
#     True
#     >>> subconjunto(ej1, ej3)
#     False
# -----

from __future__ import annotations

from abc import abstractmethod
from copy import deepcopy
from typing import Protocol, TypeVar

from hypothesis import given

from src.TAD.conjunto import (Conj, conjuntoAleatorio, elimina, esVacio,
                              inserta, menor, pertenece, vacio)
from src.TAD.Transformaciones_conjuntos_listas import conjuntoAlista


class Comparable(Protocol):
    @abstractmethod
    def __lt__(self: A, otro: A) -> bool:
        pass

A = TypeVar('A', bound=Comparable)

# 1ª solución
# =====

def subconjunto(c1: Conj[A], c2: Conj[A]) -> bool:
    if esVacio(c1):
        return True
    mc1 = menor(c1)

```

```

    rc1 = elimina(mc1, c1)
    return pertenece(mc1, c2) and subconjunto(rc1, c2)

# 2ª solución
# =====

def subconjunto2(c1: Conj[A], c2: Conj[A]) -> bool:
    return all((pertenece(x, c2) for x in conjuntoAlista(c1)))

# La función conjuntoAlista está definida en el ejercicio
# "Transformaciones entre conjuntos y listas" que se encuentra en
# https://bit.ly/3RexzxH

# 3ª solución
# =====

# (sublista xs ys) se verifica si xs es una sublista de ys. Por
# ejemplo,
#   sublista [5, 2] [3, 2, 5] == True
#   sublista [5, 2] [3, 4, 5] == False
def sublista(xs: list[A], ys: list[A]) -> bool:
    if not xs:
        return True
    return xs[0] in ys and sublista(xs[1:], ys)

def subconjunto3(c1: Conj[A], c2: Conj[A]) -> bool:
    return sublista(conjuntoAlista(c1), conjuntoAlista(c2))

# 4ª solución
# =====

def subconjunto4(c1: Conj[A], c2: Conj[A]) -> bool:
    while not esVacio(c1):
        mc1 = menor(c1)
        if not pertenece(mc1, c2):
            return False
        c1 = elimina(mc1, c1)
    return True

# 5ª solución

```



```

# =====

def subconjunto5Aux(c1: Conj[A], c2: Conj[A]) -> bool:
    while not c1.esVacio():
        mc1 = c1.menor()
        if not c2.pertenece(mc1):
            return False
        c1.elimina(mc1)
    return True

def subconjunto5(c1: Conj[A], c2: Conj[A]) -> bool:
    _c1 = deepcopy(c1)
    return subconjunto5Aux(_c1, c2)

# Comprobación de equivalencia
# =====

# La propiedad es
@given(c1=conjuntoAleatorio(), c2=conjuntoAleatorio())
def test_subconjunto(c1: Conj[int], c2: Conj[int]) -> None:
    r = subconjunto(c1, c2)
    assert subconjunto2(c1, c2) == r
    assert subconjunto3(c1, c2) == r
    assert subconjunto4(c1, c2) == r
    assert subconjunto5(c1, c2) == r

# La comprobación de las propiedades es
# > poetry run pytest -q TAD_subconjunto.py
# 1 passed in 0.37s

```

8.8. Reconocimiento de subconjunto propio

8.8.1. En Haskell

```

-- -----
-- Utilizando el [tipo abstracto de datos de los conjuntos](https://bit.ly/3HbB7f
-- definir la función
--     subconjuntoPropio :: Ord a => Conj a -> Conj a -> Bool
-- tal (subconjuntoPropio c1 c2) se verifica si c1 es un subconjunto
-- propio de c2. Por ejemplo,

```

```
-- λ> ej1 = inserta 5 (inserta 2 vacio)
-- λ> ej2 = inserta 3 (inserta 2 (inserta 5 vacio))
-- λ> ej3 = inserta 3 (inserta 4 (inserta 5 vacio))
-- λ> ej4 = inserta 2 (inserta 5 vacio)
-- λ> subconjuntoPropio ej1 ej2
-- True
-- λ> subconjuntoPropio ej1 ej3
-- False
-- λ> subconjuntoPropio ej1 ej4
-- False
-- -----
```

```
{-# OPTIONS_GHC -fno-warn-unused-imports #-}
```

```
module TAD_subconjuntoPropio where
```

```
import TAD.Conjunto (Conj, vacio, inserta)
```

```
import TAD_subconjunto (subconjunto)
```

```
subconjuntoPropio :: Ord a => Conj a -> Conj a -> Bool
```

```
subconjuntoPropio c1 c2 =
  subconjunto c1 c2 && c1 /= c2
```

```
-- La función subconjunto está definida en el ejercicio
-- "Reconocimiento de subconjuntos" que se encuentra en
-- https://bit.ly/3wPBtU5
```

8.8.2. En Python

```
# -----
# Utilizando el [tipo abstracto de datos de los conjuntos](https://bit.ly/3HbB7fo)
# definir la función
#   subconjuntoPropio : (Conj[A], Conj[A]) -> bool
# tal subconjuntoPropio(c1, c2) se verifica si c1 es un subconjunto
# propio de c2. Por ejemplo,
#   >>> ej1 = inserta(5, inserta(2, vacio()))
#   >>> ej2 = inserta(3, inserta(2, inserta(5, vacio())))
#   >>> ej3 = inserta(3, inserta(4, inserta(5, vacio())))
#   >>> ej4 = inserta(2, inserta(5, vacio()))
#   >>> subconjuntoPropio(ej1, ej2)
```

```

#     True
#     >>> subconjuntoPropio(ej1, ej3)
#     False
#     >>> subconjuntoPropio(ej1, ej4)
#     False
# -----

from __future__ import annotations

from abc import abstractmethod
from typing import Protocol, TypeVar

from src.TAD.conjunto import (Conj, conjuntoAleatorio, elimina, esVacio,
                              inserta, menor, pertenece, vacio)
from src.TAD_subconjunto import subconjunto

class Comparable(Protocol):
    @abstractmethod
    def __lt__(self: A, otro: A) -> bool:
        pass

A = TypeVar('A', bound=Comparable)

def subconjuntoPropio(c1: Conj[A], c2: Conj[A]) -> bool:
    return subconjunto(c1, c2) and c1 != c2

# La función subconjunto está definida en el ejercicio
# "Reconocimiento de subconjuntos" que se encuentra en
# https://bit.ly/3wPBtU5

```

8.9. Conjunto unitario

8.9.1. En Haskell

```

-- -----
-- Utilizando el [tipo abstracto de datos de los conjuntos](https://bit.ly/3HbB7f
-- definir la función
--     unitario :: Ord a => a -> Conj a
-- tal que (unitario x) es el conjunto {x}. Por ejemplo,

```

```
--    unitario 5 == {5}
```

```
module TAD_Conjunto_unitario where
```

```
import TAD.Conjunto (Conj, vacio, inserta)
```

```
unitario :: Ord a => a -> Conj a
```

```
unitario x = inserta x vacio
```

8.9.2. En Python

```
# -----
# Utilizando el [tipo abstracto de datos de los conjuntos](https://bit.ly/3HbB7fo)
# definir la función
#    unitario :: Ord a => a -> Conj a
# tal que (unitario x) es el conjunto {x}. Por ejemplo,
#    unitario 5 == {5}
# -----
```

```
from __future__ import annotations
```

```
from abc import abstractmethod
```

```
from typing import Protocol, TypeVar
```

```
from src.TAD.conjunto import Conj, inserta, vacio
```

```
class Comparable(Protocol):
```

```
    @abstractmethod
```

```
    def __lt__(self: A, otro: A) -> bool:
```

```
        pass
```

```
A = TypeVar('A', bound=Comparable)
```

```
def unitario(x: A) -> Conj[A]:
```

```
    return inserta(x, vacio())
```

8.10. Número de elementos de un conjunto

8.10.1. En Haskell

```

-----
-- Utilizando el [tipo abstracto de datos de los conjuntos](https://bit.ly/3HbB7f
-- definir la función
--   cardinal :: Conj a -> Int
-- tal que (cardinal c) es el número de elementos del conjunto c. Por
-- ejemplo,
--   cardinal (inserta 4 (inserta 5 vacio))           == 2
--   cardinal (inserta 4 (inserta 5 (inserta 4 vacio))) == 2
-----

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module TAD_Numero_de_elementos_de_un_conjunto where

import TAD.Conjunto (Conj, vacio, inserta, menor, elimina, esVacio)
import TAD.Transformaciones_conjuntos_listas (conjuntoAlista)
import Test.QuickCheck

-- 1ª solución
-- =====

cardinal :: Ord a => Conj a -> Int
cardinal c
  | esVacio c = 0
  | otherwise = 1 + cardinal (elimina (menor c) c)

-- 2ª solución
-- =====

cardinal2 :: Ord a => Conj a -> Int
cardinal2 = length . conjuntoAlista

-- Comprobación de equivalencia
-- =====

-- La propiedad es
prop_cardinal :: Conj Int -> Bool

```

```
prop_cardinal c =
  cardinal c == cardinal2 c

-- La comprobación es
--   λ> quickCheck prop_cardinal
--   +++ OK, passed 100 tests.
```

8.10.2. En Python

```
# -----
# Utilizando el [tipo abstracto de datos de los conjuntos](https://bit.ly/3HbB7fo)
# definir la función
#   cardinal : (Conj[A]) -> int
# tal que cardinal(c) es el número de elementos del conjunto c. Por
# ejemplo,
#   cardinal(inserta(4, inserta(5, vacio()))) == 2
#   cardinal(inserta(4, inserta(5, inserta(4, vacio())))) == 2
# -----

from __future__ import annotations

from abc import abstractmethod
from copy import deepcopy
from typing import Protocol, TypeVar

from hypothesis import given

from src.TAD.conjunto import (Conj, conjuntoAleatorio, elimina, esVacio,
                              inserta, menor, vacio)
from src.TAD.Transformaciones_conjuntos_listas import conjuntoAlista

class Comparable(Protocol):
    @abstractmethod
    def __lt__(self: A, otro: A) -> bool:
        pass

A = TypeVar('A', bound=Comparable)

# 1ª solución
```

```
# =====

def cardinal(c: Conj[A]) -> int:
    if esVacio(c):
        return 0
    return 1 + cardinal(elimina(menor(c), c))

# 2ª solución
# =====

def cardinal2(c: Conj[A]) -> int:
    return len(conjuntoAlista(c))

# 3ª solución
# =====

def cardinal3(c: Conj[A]) -> int:
    r = 0
    while not esVacio(c):
        r = r + 1
        c = elimina(menor(c), c)
    return r

# 4ª solución
# =====

def cardinal4Aux(c: Conj[A]) -> int:
    r = 0
    while not c.esVacio():
        r = r + 1
        c.elimina(menor(c))
    return r

def cardinal4(c: Conj[A]) -> int:
    _c = deepcopy(c)
    return cardinal4Aux(_c)

# Comprobación de equivalencia
# =====
```

```

@given(c=conjuntoAleatorio())
def test_cardinal(c: Conj[int]) -> None:
    r = cardinal(c)
    assert cardinal2(c) == r
    assert cardinal3(c) == r
    assert cardinal3(c) == r

# La comprobación es
#     src> poetry run pytest -q TAD_Numero_de_elementos_de_un_conjunto.py
#     1 passed in 0.33s

```

8.11. Unión de dos conjuntos

8.11.1. En Haskell

```

-- -----
-- Utilizando el [tipo abstracto de datos de los conjuntos](https://bit.ly/3HbB7f
-- definir la función
--     union :: Ord a => Conj a -> Conj a -> Conj a
-- tal (union c1 c2) es la unión de ambos conjuntos. Por ejemplo,
--     λ> ej1 = inserta 3 (inserta 5 vacio)
--     λ> ej2 = inserta 4 (inserta 3 vacio)
--     λ> union ej1 ej2
--     {3, 4, 5}
-- -----

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module TAD_Union_de_dos_conjuntos where

import TAD.Conjunto (Conj, vacio, inserta, menor, elimina, esVacio)
import TAD.Transformaciones_conjuntos_listas (conjuntoAlista, listaAconjunto)
import qualified Data.List as L (union)
import Test.QuickCheck

-- 1ª solución
-- =====

union :: Ord a => Conj a -> Conj a -> Conj a
union c1 c2

```



```

    | esVacio c1 = c2
    | otherwise = inserta mc1 (rc1 `union` c2)
  where mc1 = menor c1
        rc1 = elimina mc1 c1

-- 2ª solución
-- =====

union2 :: Ord a => Conj a -> Conj a -> Conj a
union2 c1 c2 =
  foldr inserta c2 (conjuntoAlista c1)

-- La función conjuntoAlista está definida en el ejercicio
-- "Transformaciones entre conjuntos y listas" que se encuentra en
-- https://bit.ly/3RexzxH

-- 3ª solución
-- =====

union3 :: Ord a => Conj a -> Conj a -> Conj a
union3 c1 c2 =
  listaAconjunto (conjuntoAlista c1 `L.union` conjuntoAlista c2)

-- La función listaAconjunto está definida en el ejercicio
-- "Transformaciones entre conjuntos y listas" que se encuentra en
-- https://bit.ly/3RexzxH

-- Comprobación de equivalencia
-- =====

-- La propiedad es
prop_union :: Conj Int -> Conj Int -> Bool
prop_union c1 c2 =
  all (== union c1 c2)
    [union2 c1 c2,
     union3 c1 c2]

-- La comprobación es
--   λ> quickCheck prop_union
--   +++ OK, passed 100 tests.

```

8.11.2. En Python

```
# -----
# Utilizando el [tipo abstracto de datos de los conjuntos](https://bit.ly/3HbB7fo)
# definir la función
#     union : (Conj[A], Conj[A]) -> Conj[A]
# tal (union c1 c2) es la unión de ambos conjuntos. Por ejemplo,
#     >>> ej1 = inserta(3, inserta(5, vacio()))
#     >>> ej2 = inserta(4, inserta(3, vacio()))
#     >>> union(ej1, ej2)
#     {3, 4, 5}
# -----
```

```
from __future__ import annotations
```

```
from abc import abstractmethod
from copy import deepcopy
from functools import reduce
from typing import Protocol, TypeVar
```

```
from hypothesis import given
```

```
from src.TAD.conjunto import (Conj, conjuntoAleatorio, elimina, esVacio,
                              inserta, menor, vacio)
from src.TAD.Transformaciones_conjuntos_listas import conjuntoAlista
```

```
class Comparable(Protocol):
    @abstractmethod
    def __lt__(self: A, otro: A) -> bool:
        pass
```

```
A = TypeVar('A', bound=Comparable)
```

```
# 1ª solución
# =====
```

```
def union(c1: Conj[A], c2: Conj[A]) -> Conj[A]:
    if esVacio(c1):
        return c2
    mcl = menor(c1)
```

```
    rc1 = elimina(mc1, c1)
    return inserta(mc1, union(rc1, c2))

# 2ª solución
# =====

def union2(c1: Conj[A], c2: Conj[A]) -> Conj[A]:
    return reduce(lambda c, x: inserta(x, c), conjuntoAlista(c1), c2)

# La función conjuntoAlista está definida en el ejercicio
# "Transformaciones entre conjuntos y listas" que se encuentra en
# https://bit.ly/3RexzxH

# 3ª solución
# =====

def union3(c1: Conj[A], c2: Conj[A]) -> Conj[A]:
    r = c2
    while not esVacio(c1):
        mc1 = menor(c1)
        r = inserta(mc1, r)
        c1 = elimina(mc1, c1)
    return r

# 4ª solución
# =====

def union4Aux(c1: Conj[A], c2: Conj[A]) -> Conj[A]:
    while not c1.esVacio():
        mc1 = menor(c1)
        c2.inserta(mc1)
        c1.elimina(mc1)
    return c2

def union4(c1: Conj[A], c2: Conj[A]) -> Conj[A]:
    _c1 = deepcopy(c1)
    _c2 = deepcopy(c2)
    return union4Aux(_c1, _c2)

# Comprobación de equivalencia
```

```
# =====

# La propiedad es
@given(c1=conjuntoAleatorio(), c2=conjuntoAleatorio())
def test_union(c1: Conj[int], c2: Conj[int]) -> None:
    r = union(c1, c2)
    assert union2(c1, c2) == r
    assert union3(c1, c2) == r
    assert union4(c1, c2) == r

# La comprobación de las propiedades es
# > poetry run pytest -q TAD_Union_de_dos_conjuntos.py
# 1 passed in 0.35s
```

8.12. Unión de varios conjuntos

8.12.1. En Haskell

```
-- -----
-- Utilizando el [tipo abstracto de datos de los conjuntos](https://bit.ly/3HbB7f
-- definir la función
-- unionG:: Ord a => [Conj a] -> Conj a
-- tal (unionG cs) calcule la unión de la lista de conjuntos cd. Por
-- ejemplo,
-- λ> ej1 = inserta 3 (inserta 5 vacio)
-- λ> ej2 = inserta 5 (inserta 6 vacio)
-- λ> ej3 = inserta 3 (inserta 6 vacio)
-- λ> unionG [ej1, ej2, ej3]
-- {3, 5, 6}
-- -----

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module TAD_Union_de_varios_conjuntos where

import TAD.Conjunto (Conj, vacio, inserta)
import TAD_Union_de_dos_conjuntos (union)
import Test.QuickCheck

-- 1ª solución
```

```

-- =====

unionG :: Ord a => [Conj a] -> Conj a
unionG []          = vacio
unionG (c:cs) = c `union` unionG cs

-- La función union está definida en el ejercicio
-- "Unión de dos conjuntos" que se encuentra en
-- https://bit.ly/3Y1jBl8

-- 2ª solución
-- =====

unionG2 :: Ord a => [Conj a] -> Conj a
unionG2 = foldr union vacio

-- Comprobación de equivalencia
-- =====

-- La propiedad es
prop_unionG :: [Conj Int] -> Bool
prop_unionG cs =
  unionG cs == unionG2 cs

-- La comprobación es
--   λ> quickCheck prop_unionG
--   +++ OK, passed 100 tests.

```

8.12.2. En Python

```

# -----
# Utilizando el [tipo abstracto de datos de los conjuntos](https://bit.ly/3HbB7fo)
# definir la función
#   unionG : (list[Conj[A]]) -> Conj[A]
# tal unionG(cs) calcule la unión de la lista de conjuntos cd. Por
# ejemplo,
#   >>> ej1 = inserta(3, inserta(5, vacio()))
#   >>> ej2 = inserta(5, inserta(6, vacio()))
#   >>> ej3 = inserta(3, inserta(6, vacio()))
#   >>> unionG([ej1, ej2, ej3])

```

```

# {3, 5, 6}
# -----

from __future__ import annotations

from abc import abstractmethod
from functools import reduce
from typing import Protocol, TypeVar

from hypothesis import given
from hypothesis import strategies as st

from src.TAD.conjunto import Conj, conjuntoAleatorio, inserta, vacio
from src.TAD.Union_de_dos_conjuntos import union

class Comparable(Protocol):
    @abstractmethod
    def __lt__(self: A, otro: A) -> bool:
        pass

A = TypeVar('A', bound=Comparable)

# 1ª solución
# =====

def unionG(cs: list[Conj[A]]) -> Conj[A]:
    if not cs:
        return vacio()
    return union(cs[0], unionG(cs[1:]))

# La función union está definida en el ejercicio
# "Unión de dos conjuntos" que se encuentra en
# https://bit.ly/3Y1jBl8

# 2ª solución
# =====

def unionG2(cs: list[Conj[A]]) -> Conj[A]:
    return reduce(union, cs, vacio())

```

```

# 3ª solución
# =====

def unionG3(cs: list[Conj[A]]) -> Conj[A]:
    r: Conj[A] = vacio()
    for c in cs:
        r = union(c, r)
    return r

# Comprobación de equivalencia
# =====

# La propiedad es
@given(st.lists(conjuntoAleatorio(), max_size=10))
def test_union(cs: list[Conj[int]]) -> None:
    r = unionG(cs)
    assert unionG2(cs) == r
    assert unionG3(cs) == r

# La comprobación de las propiedades es
# > poetry run pytest -q TAD_Union_de_varios_conjuntos.py
# 1 passed in 0.75s

```

8.13. Intersección de dos conjuntos

8.13.1. En Haskell

```

-----
-- Utilizando el [tipo abstracto de datos de los conjuntos](https://bit.ly/3HbB7f
-- definir la función
--   interseccion :: Ord a => Conj a -> Conj a -> Conj a
-- tal que (interseccion c1 c2) es la intersección de los conjuntos c1 y
-- c2. Por ejemplo,
--   λ> ej1 = inserta 3 (inserta 5 (inserta 2 vacio))
--   λ> ej2 = inserta 2 (inserta 4 (inserta 3 vacio))
--   λ> interseccion ej1 ej2
--   {2, 3}
-----

```

```

{-# OPTIONS_GHC -fno-warn-unused-imports #-}

module TAD_Interseccion_de_dos_conjuntos where

import TAD.Conjunto (Conj, vacio, inserta, menor, elimina, pertenece, esVacio)
import TAD.Transformaciones_conjuntos_listas (conjuntoAlista, listaAconjunto)
import Data.List (intersect)
import Test.QuickCheck

-- 1ª solución
-- =====

interseccion :: Ord a => Conj a -> Conj a -> Conj a
interseccion c1 c2
  | esVacio c1      = vacio
  | pertenece mc1 c2 = inserta mc1 (interseccion rc1 c2)
  | otherwise       = interseccion rc1 c2
  where mc1 = menor c1
        rc1 = elimina mc1 c1

-- 2ª solución
-- =====

interseccion2 :: Ord a => Conj a -> Conj a -> Conj a
interseccion2 c1 c2 =
  listaAconjunto [x | x <- conjuntoAlista c1, x `pertenece` c2]

-- Las funciones conjuntoAlista y listaAconjunto está definida en el
-- ejercicio Transformaciones entre conjuntos y listas" que se encuentra
-- en https://bit.ly/3RexzxH

-- 3ª solución
-- =====

interseccion3 :: Ord a => Conj a -> Conj a -> Conj a
interseccion3 c1 c2 =
  listaAconjunto (conjuntoAlista c1 `intersect` conjuntoAlista c2)

-- Comprobación de equivalencia
-- =====

```



```
-- La propiedad es
prop_interseccion :: Conj Int -> Conj Int -> Bool
prop_interseccion c1 c2 =
    all (== interseccion c1 c2)
        [interseccion2 c1 c2,
         interseccion3 c1 c2]

-- La comprobación es
--    λ> quickCheck prop_interseccion
--    +++ OK, passed 100 tests.
```

8.13.2. En Python

[illegible]

```
class Comparable(Protocol):
    @abstractmethod
    def __lt__(self: A, otro: A) -> bool:
        pass
```

```
A = TypeVar('A', bound=Comparable)
```

```
# 1ª solución
```

```
# =====
```

```
def interseccion(c1: Conj[A], c2: Conj[A]) -> Conj[A]:
    if esVacio(c1):
        return vacio()
    mcl = menor(c1)
    rcl = elimina(mcl, c1)
    if pertenece(mcl, c2):
        return inserta(mcl, interseccion(rcl, c2))
    return interseccion(rcl, c2)
```

```
# 2ª solución
```

```
# =====
```

```
def interseccion2(c1: Conj[A], c2: Conj[A]) -> Conj[A]:
    return listaAconjunto([x for x in conjuntoAlista(c1)
                           if pertenece(x, c2)])
```

```
#
```

```
# Las funciones conjuntoAlista y listaAconjunto está definida en el
# ejercicio Transformaciones entre conjuntos y listas" que se encuentra
# en https://bit.ly/3RexzxH
```

```
# 3ª solución
```

```
# =====
```

```
def interseccion3(c1: Conj[A], c2: Conj[A]) -> Conj[A]:
    r: Conj[A] = vacio()
    while not esVacio(c1):
        mcl = menor(c1)
        c1 = elimina(mcl, c1)
        if pertenece(mcl, c2):
            r = inserta(mcl, r)
```

```

    return r

# 4ª solución
# =====

def interseccion4Aux(c1: Conj[A], c2: Conj[A]) -> Conj[A]:
    r: Conj[A] = vacio()
    while not c1.esVacio():
        mc1 = c1.menor()
        c1.elimina(mc1)
        if c2.pertenece(mc1):
            r.inserta(mc1)
    return r

def interseccion4(c1: Conj[A], c2: Conj[A]) -> Conj[A]:
    _c1 = deepcopy(c1)
    return interseccion4Aux(_c1, c2)

# Comprobación de equivalencia
# =====

# La propiedad es
@given(c1=conjuntoAleatorio(), c2=conjuntoAleatorio())
def test_interseccion(c1: Conj[int], c2: Conj[int]) -> None:
    r = interseccion(c1, c2)
    assert interseccion2(c1, c2) == r
    assert interseccion3(c1, c2) == r
    assert interseccion4(c1, c2) == r

# La comprobación de las propiedades es
# > poetry run pytest -q TAD_Interseccion_de_dos_conjuntos.py
# 1 passed in 0.30s

```


Apéndices

Apéndice A

Método de Pólya para la resolución de problemas

A.1. Método de Pólya para la resolución de problemas matemáticos

Para resolver un problema se necesita:

Paso 1: Entender el problema

- ¿Cuál es la incógnita?, ¿Cuáles son los datos?
- ¿Cuál es la condición? ¿Es la condición suficiente para determinar la incógnita? ¿Es insuficiente? ¿Redundante? ¿Contradictoria?

Paso 2: Configurar un plan

- ¿Te has encontrado con un problema semejante? ¿O has visto el mismo problema planteado en forma ligeramente diferente?
- ¿Conoces algún problema relacionado con éste? ¿Conoces algún teorema que te pueda ser útil? Mira atentamente la incógnita y trata de recordar un problema que sea familiar y que tenga la misma incógnita o una incógnita similar.
- He aquí un problema relacionado al tuyo y que ya has resuelto ya. ¿Puedes utilizarlo? ¿Puedes utilizar su resultado? ¿Puedes emplear su método? ¿Te hace falta introducir algún elemento auxiliar a fin de poder utilizarlo?

- ¿Puedes enunciar al problema de otra forma? ¿Puedes plantearlo en forma diferente nuevamente? Recurre a las definiciones.
- Si no puedes resolver el problema propuesto, trata de resolver primero algún problema similar. ¿Puedes imaginarte un problema análogo un tanto más accesible? ¿Un problema más general? ¿Un problema más particular? ¿Un problema análogo? ¿Puede resolver una parte del problema? Considera sólo una parte de la condición; descarta la otra parte; ¿en qué medida la incógnita queda ahora determinada? ¿En qué forma puede variar? ¿Puedes deducir algún elemento útil de los datos? ¿Puedes pensar en algunos otros datos apropiados para determinar la incógnita? ¿Puedes cambiar la incógnita? ¿Puedes cambiar la incógnita o los datos, o ambos si es necesario, de tal forma que estén más cercanos entre sí?
- ¿Has empleado todos los datos? ¿Has empleado toda la condición? ¿Has considerado todas las nociones esenciales concernientes al problema?

Paso 3: Ejecutar el plan

- Al ejecutar tu plan de la solución, comprueba cada uno de los pasos
- ¿Puedes ver claramente que el paso es correcto? ¿Puedes demostrarlo?

Paso 4: Examinar la solución obtenida

- ¿Puedes verificar el resultado? ¿Puedes el razonamiento?
- ¿Puedes obtener el resultado en forma diferente? ¿Puedes verlo de golpe? ¿Puedes emplear el resultado o el método en algún otro problema?

G. Polya "Cómo plantear y resolver problemas" (Ed. Trillas, 1978) p. 19

A.2. Método de Pólya para resolver problemas de programación

Para resolver un problema se necesita:

Paso 1: Entender el problema

- ¿Cuáles son las *argumentos*? ¿Cuál es el *resultado*? ¿Cuál es *nombre* de la función? ¿Cuál es su *tipo*?
- ¿Cuál es la *especificación* del problema? ¿Puede satisfacerse la especificación? ¿Es insuficiente? ¿Redundante? ¿Contradictoria? ¿Qué restricciones se suponen sobre los argumentos y el resultado?
- ¿Puedes descomponer el problema en partes? Puede ser útil dibujar diagramas con ejemplos de argumentos y resultados.

Paso 2: Diseñar el programa

- ¿Te has encontrado con un problema semejante? ¿O has visto el mismo problema planteado en forma ligeramente diferente?
- ¿Conoces algún problema *relacionado* con éste? ¿Conoces alguna función que te pueda ser útil? Mira atentamente el tipo y trata de recordar un problema que sea familiar y que tenga el mismo tipo o un tipo similar.
- ¿Conoces algún problema familiar con una *especificación* similar?
- He aquí un problema *relacionado* al tuyo y que ya has resuelto. ¿Puedes utilizarlo? ¿Puedes utilizar su resultado? ¿Puedes emplear su método? ¿Te hace falta introducir alguna función auxiliar a fin de poder utilizarlo?
- Si no puedes resolver el problema propuesto, trata de resolver primero algún problema similar. ¿Puedes imaginarte un problema análogo un tanto más *accesible*? ¿Un problema más *general*? ¿Un problema más *particular*? ¿Un problema *análogo*?
- ¿Puede resolver una *parte* del problema? ¿Puedes deducir algún elemento útil de los datos? ¿Puedes pensar en algunos otros datos apropiados para determinar la incógnita? ¿Puedes cambiar la incógnita? ¿Puedes cambiar la incógnita o los datos, o ambos si es necesario, de tal forma que estén más cercanos entre sí?
- ¿Has empleado todos los datos? ¿Has empleado todas las restricciones sobre los datos? ¿Has considerado todas los requisitos de la especificación?

Paso 3: Escribir el programa

- Al escribir el programa, comprueba cada uno de los pasos y funciones auxiliares.
- ¿Puedes ver claramente que cada paso o función auxiliar es correcta?
- Puedes escribir el programa en *etapas*. Piensas en los diferentes casos en los que se divide el problema; en particular, piensas en los diferentes casos para los datos. Puedes pensar en el cálculo de los casos independientemente y *unirlos* para obtener el resultado final
- Puedes pensar en la solución del problema descomponiéndolo en problemas con datos más simples y uniendo las soluciones parciales para obtener la solución del problema; esto es, por *recursión*.
- En su diseño se puede usar problemas más generales o más particulares. Escribe las soluciones de estos problemas; ellas pueden servir como guía para la solución del problema original, o se pueden usar en su solución.
- ¿Puedes apoyarte en otros problemas que has resuelto? ¿Pueden usarse? ¿Pueden modificarse? ¿Pueden guiar la solución del problema original?

Paso 4: Examinar la solución obtenida

- ¿Puedes comprobar el funcionamiento del programa sobre una colección de argumentos?
- ¿Puedes comprobar propiedades del programa?
- ¿Puedes escribir el programa en una forma diferente?
- ¿Puedes emplear el programa o el método en algún otro programa?

Simon Thompson [How to program it](#), basado en G. Polya *Cómo plantear y resolver problemas*.

Bibliografía

- [1] C. Allen, J. Moronuki, and S. Syrek. *Haskell programming from first principles*. Lorepub LLC, 2016.
- [2] J. A. Alonso. *Temas de programación funcional con Haskell*. Technical report, Univ. de Sevilla, 2019.
- [3] J. A. Alonso and als. *Exámenes de programación funcional con Haskell*. Technical report, Univ. de Sevilla, 2021.
- [4] J. A. Alonso and M. J. Hidalgo. *Piensa en Haskell (Ejercicios de programación funcional con Haskell)*. Technical report, Univ. de Sevilla, 2012.
- [5] J. A. Alonso and M. J. Hidalgo. *Ejercicios de programación funcional con Haskell*. Technical report, Univ. de Sevilla, 2022.
- [6] R. Bird. *Introducción a la programación funcional con Haskell*. Prentice-Hall, 1999.
- [7] R. Bird. *Pearls of functional algorithm design*. Cambridge University Press, 2010.
- [8] R. Bird. *Thinking functionally with Haskell*. Cambridge University Press, 2014.
- [9] R. Bird and J. Gibbons. *Algorithm design with Haskell*. Cambridge University Press, 2020.
- [10] A. Casamayou-Boucau, P. Chauvin, and G. Connan. *Programmation en Python pour les mathématiques*. Dunod, 2012.
- [11] A. Downey, J. Elkner, and C. Meyers. *Aprenda a pensar como un programador con Python*. Green Tea Press, 2002.
- [12] M. Goodrich, R. Tamassia, and M. Goldwasser. *Data structures and algorithms in Python*. Wiley, 2013.

- [13] J. Guttag. *Introduction to computation and programming using python, second edition*. MIT Press, 2016.
- [14] T. Hall and J. Stacey. *Python 3 for absolute beginners*. Apress, 2010.
- [15] M. Hetland. *Python algorithms: Mastering basic algorithms in the Python language*. Apress, 2011.
- [16] P. Hudak. *The Haskell school of music (From signals to symphonies)*. Technical report, Yale University, 2012.
- [17] J. Hunt. *A beginners guide to Python 3 programming*. Springer International Publishing, 2019.
- [18] J. Hunt. *Advanced guide to Python 3 programming*. Springer International Publishing, 2019.
- [19] G. Hutton. *Programming in Haskell (2nd ed.)*. Cambridge University Press, 2016.
- [20] W. Kurt. *Get programming with Haskell*. Manning Publications, 2018.
- [21] M. Lipovača. ¡Aprende Haskell por el bien de todos! En <http://aprendehaskell.es>.
- [22] S. Lott. *Functional Python programming, 2nd Edition*. Packt Publishing, 2018.
- [23] C. Okasaki. *Purely functional data structures*. Cambridge University Press, 1999.
- [24] B. O’Sullivan, D. Stewart, and J. Goerzen. *Real world Haskell*. O’Reilly, 2008.
- [25] T. Padmanabhan. *Programming with Python*. Springer Singapore, 2017.
- [26] G. Pólya. *Cómo plantear y resolver problemas*. Editorial Trillas, 1965.
- [27] F. Rabhi and G. Lapalme. *Algorithms: A functional programming approach*. Addison-Wesley, 1999.
- [28] M. Rubio-Sanchez. *Introduction to recursive programming*. CRC Press, 2017.
- [29] B. C. Ruiz, F. Gutiérrez, P. Guerrero, and J. Gallardo. *Razonando con Haskell (Un curso sobre programación funcional)*. Thompson, 2004.

- [30] A. Saha. *Doing Math with Python: Use Programming to explore algebra, statistics, calculus, and more!* No Starch Press, 2015.
- [31] Y. Sajanikar. *Haskell cookbook*. Packt Publishing, 2017.
- [32] D. Sannella, M. Fourman, H. Peng, and P. Wadler. *Introduction to computation: Haskell, logic and automata*. Springer International Publishing, 2022.
- [33] A. Serrano. *Beginning Haskell: A project-based approach*. Apress, 2014.
- [34] N. Shukla. *Haskell data analysis cookbook*. Packt Publishing, 2014.
- [35] B. Stephenson. *The Python workbook: A brief introduction with exercises and solutions*. Springer International Publishing, 2015.
- [36] S. Thompson. *Haskell: The craft of functional programming*. Addison-Wesley, third edition, 2011.
- [37] R. van Hattem. *Mastering Python: Write powerful and efficient code using the full range of Python's capabilities*. Packt Publishing, 2022.