Universidad: **Universidad de Málaga · UMA**

Grado: **Matemáticas**

Asignatura: **Informática I**

Profesor/a: **Andrea Campano**

Examen Resuelto Febrero 2024

El examen se entrega en un único fichero, donde debe estar claramente indicado cada problema con un comentario tipo #problema 1.

- El alumno pondrá el nombre en la primera línea del fichero que va a entregar - El examen se puede resolver en Jupyter Notebook o Visual Code Studio. El fichero tendrá extensión ipynb, y estará libre de errores.
- No se puede usar ningún fichero adicional, apuntes o consultar internet. Únicamente los ficheros que se han puesto en el campus virtual.
- Se puede usar la chuleta de funciones Python en papel.
- Las funciones deben llamarse con nombre indicado en el enunciado. A las funciones auxiliares pueden tener el nombre que se desee.
- Prográmese cada función según la técnica pedida. Si no se dice nada de qué técnica utilizar para programar, utilícese la que se considere más apropiada. Si hace falta, se pueden usar funciones auxiliares.
- Únicamente se puede programar en Python con los elementos explicados en el curso: listas por comprensión, recursividad, funciones de orden superior y estructuras de datos. Todo lo que se salga del contenido del curso no será evaluado.
- Las funciones deben hacer lo que se pide en el enunciado, y devolver lo que dice el enunciado, y no algo parecido. Las funciones que no compilen NO se corregirán. El alumno tiene que asegurarse que su código compila y es correcto.
- Los comentarios y explicaciones se pondrán en el mismo fichero en la línea del código, encima o debajo.
- La corrección del examen no es automática, se corrige a mano. En la corrección, se valorará la elegancia y eficiencia en la implementación. Es decir, no es suficiente con que funcione y devuelva el valor pedido sino la manera en que se llega a esa solución. La forma de programarlo y la eficiencia de la función. Los comentarios ayudan a la corrección.

Problema 1. Fechas. 2,5 puntos

Las fechas son un tipo de datos que tienen tres valores, día, mes y año. Vamos a trabajar con fechas desde el año 1 hasta el año 3000 , incluidos los años bisiestos. Los tres valores que definen una fecha tienen las siguientes restricciones:

- Los días son números enteros positivos que van desde 1 hasta, como mucho y dependiendo del mes y el año, $28,29,30$ o 31.
- Los meses van desde 1 a 12.
- Los años van desde 1 a 3000, en este problema.

En Python definimos una fecha como una lista de exactamente tres elementos, donde el primero indica el día, el segundo indica el mes y el tercero indica el año. Ejemplos de fechas correctas:

hoy = [19,1 ,2024]
anioNuevo = [1, 1, 2024]

Se pide:

1. Definir la función función esFechaValida tal que dada una fecha indique si es válida, es decir, si cumple todas las restricciones indicadas, teniendo en cuenta además si el año es bisiesto o no. (0,2p)

   esFechaValida([19,1,2024]) → True
   esFechaValida([28,13,2023]) → False
   esFechaValida([29,2,2000]) → True
   esFechaValida([28,6,23]) → True
   esFechaValida([28,6,23,45]) → False (porque tiene cuatro elementos)
   esfechaValida(28,-3,45]) → False, un elemento es negativo

Solución

```
def es_bisiesto(año):
    #Un año es bisiesto si es divisible por 4 pero no por 100, o si es divisible por 400
    return (año % 4 == 0 and año % 100 != 0) or (año % 400 == 0)

def dias_en_mes(mes, año):
    #Devuelve el número de días de un mes dado el año
    if mes == 1 or mes == 3 or mes == 5 or mes == 7 or mes == 8 or mes == 10 or mes == 12:
        return 31
    elif mes == 4 or mes == 6 or mes == 9 or mes == 11:
        return 30
    elif mes == 2:
        if es_bisiesto(año):
            return 29
        else:
            return 28
        else:
            return -1  #Caso no válido

def es_fecha_valida(fecha):
    #Verifica que la fecha sea una lista de exactamente tres elementos
    if len(fecha) != 3:
        return False
    dia, mes, año = fecha
    #Verifica que los elementos sean enteros positivos
    if not (isinstance(dia, int) and isinstance(mes, int) and isinstance(año, int)):
        return False
    if dia <= 0 or mes <= 0 or año <= 0:
        return False
    #Verifica que los meses estén entre 1 y 12
    if mes < 1 or mes > 12:
        return False
    #Verifica que los años estén en el rango adecuado
    if año < 1 or año > 3000:
        return False
```

```
    #Verifica que los días estén en el rango adecuado para el mes y el año
    if dia > dias_en_mes(mes, año):
        return False
    return True

#Ejemplo:
print(es_fecha_valida([19, 1, 2024]))
print(es_fecha_valida([28, 13, 2023]))
print(es_fecha_valida([29, 2, 2000]))
print(es_fecha_valida([28, 6, 23]))
print(es_fecha_valida([28, 6, 23, 45]))
print(es_fecha_valida([28, -3, 45]))
```

2. Definir la función fechaDe que dado un número menor que 366 nos diga a qué día/mes corresponde. Se indicará con un parámetro si el año es bisiesto o no. Por defecto, el año será no bisiesto. (0,7p)

   fechaDe (60 , True) →[29,2] #año bisiesto
   fechaDe (60) → [1,3] #año no bisiesto
   fechaDe (366, True) → [31,12]
   fechaDe (366) → error número no valido #(por defecto es "no bisiesto")
   fechaDe (100) → [10,4]

Solución

```
def fechaDe(dia_del_año, es_bisiesto=False):
    #Definimos los días en cada mes para años no bisiestos y bisiestos
    dias_en_meses_no_bisiesto = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
    dias_en_meses_bisiesto = [31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
    #Seleccionamos la lista de días en meses según si es bisiesto o no
    dias_en_meses = dias_en_meses_bisiesto if es_bisiesto else dias_en_meses_no_bisiesto
    #Calculamos el número máximo de días en el año
    max_dias = 366 if es_bisiesto else 365
    #Verificamos si el día del año es válido
    if dia_del_año < 1 or dia_del_año > max_dias:
        return "Error: número no válido"
    #Iteramos sobre los meses para encontrar el mes y día correspondientes
    mes = 1
    for dias_en_mes in dias_en_meses:
        if dia_del_año <= dias_en_mes:
            return [dia_del_año, mes]
        dia_del_año -= dias_en_mes
        mes += 1

#Ejemplo:
print(fechaDe(60, True))
print(fechaDe(60))
print(fechaDe(366, True))
print(fechaDe(366))
print(fechaDe(100))
```

3. Definir la función numeroDeDia que, dada una fecha, si es válida, nos diga a qué día corresponde. Si la fecha no es válida se devolverá -1. La función tendrá un parámetro opcional bisiesto que por defecto vale False. (0,3 p)

numeroDeDia ([29,2,2020]) → 60
numeroDeDia ([1,3,2034]) → 60
numeroDeDia ([14,5,2034]) → 134
numeroDeDia ([28,-3,45]) → 1

Solución

```
def numeroDeDia(fecha, bisiesto=False):
    #Primero verificamos si la fecha es válida
    if not es_fecha_valida(fecha):
        return -1
    dia, mes, año = fecha
    #Si el año de la fecha dada es bisiesto, el parámetro bisiesto será True
    bisiesto = es_bisiesto(año)
    #Definimos el número de días en cada mes
    dias_en_meses_no_bisiesto = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
    dias_en_meses_bisiesto = [31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
    #Seleccionamos la lista de días en meses adecuada
    dias_en_meses = dias_en_meses_bisiesto if bisiesto else dias_en_meses_no_bisiesto
    #Calculamos el número de día del año sumando los días de los meses anteriores y el día actual
    numero_dia = sum(dias_en_meses[:mes - 1]) + dia
    return numero_dia

#Ejemplos:
print(numeroDeDia([29, 2, 2020]))
print(numeroDeDia([1, 3, 2034]))
print(numeroDeDia([14, 5, 2034]))
print(numeroDeDia([28, -3, 45]))
```

4. Definir la función entreFechas que nos diga cuantos días hay entre dos fechas válidas. Si alguna fecha es no válida, se devolverá -1. En la entrada de la función, puede ser que la primera fecha sea mayor que la segunda o viceversa. (1p)

entreFechas ([31,1,2010], [28,2,2010]) → 28
entreFechas ([32,1,2010], [28,2,2010]) → -1 #la primera fecha es no válida
entreFechas ([1,1,2000], [31,12,2000]) → 365
entreFechas ([1, 2, 1990], [4, 3,1995]) → 1857
entreFechas ([1, 2,1990], [4, 3,2015]) → 9162
entreFechas ([4, 3,2015], [1, 2,1990]) → 9162
entreFechas ([1,2,2024], [19,8,1900]) → 45091

Solución

```
def dias_entre_años(año1, año2):
#Calcula la cantidad de días entre dos años a través de un bucle for que recorra los años entre dos fechas y vamos acumulando los días de cada año en un acumulador llamado día
    dias = 0
    for año in range(año1, año2):
        if es_bisiesto(año):
            dias += 366
        else:
            dias += 365
    return dias
```

```
def entreFechas(fecha1, fecha2):
#Verifica si ambas fechas son válidas
    if not es_fecha_valida(fecha1) or not es_fecha_valida(fecha2):
        return -1
    #Extraemos día, mes y año de ambas fechas
    dia1, mes1, año1 = fecha1
    dia2, mes2, año2 = fecha2
    #Si las fechas están en el mismo año restamos el número de días de ambas fechas
    if año1 == año2:
        return abs(numeroDeDia(fecha1) - numeroDeDia(fecha2))
    #Si las fechas no están en el mismo año, distinguimos 2 casos, el primero cuando el año1<año2 y
    el segundo cuando año1>año2 y calculamos los días completos en cada parte
    if año1 < año2:
        dias_desde_fecha1_hasta_fin_año1 = 365 + es_bisiesto(año1) - numeroDeDia(fecha1)
        dias_hasta_fecha2_desde_inicio_año2 = numeroDeDia(fecha2)
        dias_entre_años = dias_entre_años(año1 + 1, año2)
        return dias_desde_fecha1_hasta_fin_año1 + dias_entre_años + dias_hasta_fecha2_desde_inicio_año2
    else:
        dias_desde_fecha2_hasta_fin_año2 = 365 + es_bisiesto(año2) - numeroDeDia(fecha2)
        dias_hasta_fecha1_desde_inicio_año1 = numeroDeDia(fecha1)
        dias_entre_años = dias_entre_años(año2 + 1, año1)
        return dias_desde_fecha2_hasta_fin_año2 + dias_entre_años + dias_hasta_fecha1_desde_inicio_año1

#Ejemplo:
print(entreFechas([31, 1, 2010], [28, 2, 2010]))
print(entreFechas([32, 1, 2010], [28, 2, 2010]))
print(entreFechas([1, 1, 2000], [31, 12, 2000]))
print(entreFechas([1, 2, 1990], [4, 3, 1995]))
print(entreFechas([1, 2, 1990], [4, 3, 2015]))
print(entreFechas([4, 3, 2015], [1, 2, 1990]))
print(entreFechas([1, 2, 2024], [19, 8, 1900]))
```

5. Definir la función ordenarFechas que, dada una lista de fechas, si son válidas, las ordene de más reciente a más antigua y si alguna no es válida, devuelva la lista vacía. (0,3)

   ordenarFechas ([[1,1,2045],[8,3,2056],[5,3,21],[23,3,21],[4,4,4]]) →
   [[8,3,2056],[1,1,2045],[23,3,21],[5,3,21],[4,4,4]]
   ordenarFechas ([[1,15,2045],[8,3,2056],[5,3,21],[23,3,21],[4,4,4]])→[]# hay una fecha no valida

Solución

```
def convertir_a_dias(fecha):
#Convierte una fecha a un número de días desde una fecha base (1 de enero del año 1)
    dia, mes, año = fecha
    #Calcula los días completos en los años anteriores
    dias_totales = 0
    for a in range(1, año):
        dias_totales += 366 if es_bisiesto(a) else 365
    #Añade los días de los meses anteriores en el año actual
    for m in range(1, mes):
        dias_totales += dias_en_mes(m, año)
    #Añade los días del mes actual
    dias_totales += dia
    return dias_totales
```

```
def ordenarFechas(fechas):
#Verifica si todas las fechas son válidas
    for fecha in fechas:
        if not es_fecha_valida(fecha):
            return []
        #Convierte las fechas a días y las ordena
    fechas_ordenadas = sorted(fechas, key=convertir_a_dias, reverse=True)
    return fechas_ordenadas
#Ejemplos:
print(ordenarFechas([[1, 1, 2045], [8, 3, 2056], [5, 3, 21], [23, 3, 21], [4, 4, 4]]))
print(ordenarFechas([[1, 15, 2045], [8, 3, 2056], [5, 3, 21], [23, 3, 21], [4, 4, 4]]))
```

Problema 2. Cifrado afín. 2,5 puntos

1. Dos números enteros positivos son primos relativos (o coprimos) si no tienen ningún factor primo en común excepto el 1 ; es decir, si 1 es su único divisor común. Definir la función coprimos que dados dos números enteros devuelva True si son coprimos y False si no lo son. (0,3p)

   sonCoprimos (6,19) → True
   sonCoprimos (10,12) → False #tienen en común 1 y 2
   sonCoprimos (3,12) → False #tienen en común 1 y 3
   sonCoprimos (7,12) → True
   sonCoprimos (7,38) → True

Solución

```
def mcd(a, b):
#Aplicamos el algoritmo de Euclides para encontrar el máximo común divisor (MCD)
    while b != 0:
        a, b = b, a % b
    return a

def sonCoprimos(a, b):
#Dos números son coprimos si su MCD es 1
    return mcd(a, b) == 1

#Ejemplo:
print(sonCoprimos(6, 19))
print(sonCoprimos(10, 12))
print(sonCoprimos(3, 12))
print(sonCoprimos(7, 12))
print(sonCoprimos(7, 38))
```

2. Cuando tenemos una lista de caracteres, podemos hacer una cadena uniendo esos caracteres en una cadena utilizando el operador +. Así 'A' + 'B' devuelve 'AB'. Utilizando plegado (función reduce) implementar la función unirC que dada una lista de caracteres devuelva una cadena. (0,2p)

   unirc (['H', 'O', 'L', 'A']) → 'HOLA'
   unirC (['A']) → 'A'

Solución

```
from functools import reduce

def unirC(lista):
    #reduce toma una función y una secuencia y aplica la función acumulativa a los elementos de la secuencia.
    return reduce(lambda x, y: x + y, lista)

#Ejemplo:
print(unirC(['H', 'O', 'L', 'A']))
print(unirC(['A']))
```

El **cifrado Afín** es un tipo de cifrado por sustitución en el que a cada símbolo del alfabeto x se le da un valor, este valor se sustituye en una expresión del tipo y=(ax+b)%m y a lo que resulte se le asigna el símbolo correspondiente.

Para ello, se asigna a cada símbolo del alfabeto un número n

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	Ñ	P	Q	R	S	T	U	V	W	X	Y	Z		0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37

- m es el número de caracteres del alfabeto y a, b dos valores que definen la clave del código. Para el alfabeto de la tabla, m vale 38.
- para garantizar que a cada letra le correspondan una letra distinta en el cifrado, se tiene que cumplir la condición de que a sea coprimo de m.

Ejemplo:

Mensaje: ENERO m → 38 a → 7 b → 20 sonCoprimos (7,38) → True			
mx	x	y = ((7 * x) + 20) % 38	my
E	4	10	K
N	13	35	7
R	4	10	K
R	18	32	4
0	15	11	L

3. Se pide, definir la función codificarAfin que dado un mensaje, un alfabeto y dos valores clave a y b, devuelva el mensaje codificado siguiendo el método descrito. Para implementar la función suponed que siempre llega un mensaje con letras del alfabeto dado. (1p)

```
codificarAfin ("ENERO", letras, 7, 20) -> 'K7K4L'
codificarAfin ("ENERO", letras, 7, 12) -> 'C CXD
codificarAfin ("ENERO", letras, 12, 7) -> ERROR. No se puede codificar: a y m no son coprimos
```

Solución

```
def codificarAfin(mensaje, alfabeto, a, b):
    m = len(alfabeto)
    if not sonCoprimos(a, m):
        return "ERROR. No se puede codificar: a y m no son coprimos"
    #Crear un diccionario para mapear cada letra a su índice
    char_to_index = {char: idx for idx, char in enumerate(alfabeto)}
```

```
    #Función para cifrar un carácter
    def cifrar_char(char):
        x = char_to_index[char]
        y = (a * x + b) % m
        return alfabeto[y]
    #Cifrar el mensaje
    mensaje_cifrado = [cifrar_char(char) for char in mensaje]
    #Usar unirC para juntar los caracteres cifrados en una cadena
    return unirC(mensaje_cifrado)

#Ejemplos:
letras = "ABCDEFGHIJKLMNÑOPQRSTUVWXYZ 0123456789"
print(codificarAfin("ENERO", letras, 7, 20))
print(codificarAfin("ENERO", letras, 7, 12))
print(codificarAfin("ENERO", letras, 12, 7))
```

Para descifrar un mensaje Afin, la función de desencriptación es $y=(a^{(-1)}(x-b))\%m$ donde x es el carácter encriptado e y es el carácter ya desencriptado.

Por ejemplo, para el ejemplo anterior, si quiere desencriptar el carácter x= 'K' con $a=7$ y $b=20$, el valor de y sería 'E'.

Para poder aplicar la función de desencriptado, hay que calcular el módulo inverso de a (que representamos con a^{-1}) para revertir el proceso de codificación, a^{-1} es la inversa multiplicativa de a modulo m, es decir, el 1 en Z_m; en nuestro ejemplo el 1 en Z_{38}.

Cálculo del inverso

Por ejemplo, para si $a=7$ y m vale 38 (que son coprimos) busco a^{-1} de 7 en Z_{38}. Para ello busco un número menor que 38, que multiplicado a y dividido entre 38 devuelva como resto 1. En este ejemplo, es 11, ya que $7*11=77=1$ modulo $38 (77\%38 \rightarrow 1)$.

Entonces la función de desencriptación será: $y=(11*(x-20))\%38$ que se aplicará igual que en el caso del encriptado para descifrar el mensaje.

Mensaje: KEK4L m → 38 a → 7 a_{-1}→11 b → 20			
m*x*	x	y=(11*(x-20))%38	my
K	10	4	E
7	35	13	N
K	10	4	E
4	32	18	R
L	11	15	0

Se pide:

4. Definir la función inversoDe que dado un número y el tamaño del conjunto, devuelva el inverso de ese número. El inverso de a en Z_m, si a y m son coprimos, se puede calcular probando con todos los números del conjunto hasta encontrar uno que multiplicado por a devuelva un número cuyo resto con m sea 1. Implementar con listas por comprensión. (0,4p)

   inversoDe (27,56) → 27
   inversoDe (9,23) → 18
   inversoDe (7,38) → 11 #es el de nuestro ejemplo

Solución

```
def inversoDe(a, m):
    #Verificamos que a y m sean coprimos
    if not sonCoprimos(a, m):
        return None
    #Utilizamos una lista por comprensión para encontrar el inverso
    inversos = [x for x in range(1, m) if (a * x) % m == 1]
    return inversos[0] if inversos else None
#Ejemplos:
print(inversoDe(27, 56))
print(inversoDe(9, 23))
print(inversoDe(7, 38))
```

5. Definir la funcion decodificarAfin que dado un mensaje, un alfabeto y los valores de a y b devuelva el mensaje decodificado. (0,2p)

   decodificarAfin('K7K4L', letras, 7, 20) → 'ENERO'
   decodificarAfin ((codificarAfin ('ENERO', letras, 7,12)), letras, 7, 12) → 'ENERO'

Solución

```
def decodificarAfin(mensaje, alfabeto, a, b):
    m = len(alfabeto)
    #En primer lugar verificamos si son coprimos
    if not sonCoprimos(a, m):
        return "ERROR. No se puede descifrar: a y m no son coprimos"
    #Calcular la inversa multiplicativa de a
    a_inv = inversoDe(a, m)
    #Crear un diccionario para mapear cada letra a su índice
    char_to_index = {char: idx for idx, char in enumerate(alfabeto)}
    #Crear un diccionario para mapear cada índice a su letra
    index_to_char = {idx: char for idx, char in enumerate(alfabeto)}
    #Función para descifrar un carácter
    def descifrar_char(char):
        x = char_to_index[char]
        y = (a_inv * (x - b)) % m
        return index_to_char[y]
    #Descifrar el mensaje
    mensaje_descifrado = [descifrar_char(char) for char in mensaje]
    #Unir los caracteres descifrados en una cadena
    return ''.join(mensaje_descifrado)

#Ejemplo:
letras = "ABCDEFGHIJKLMNÑOPQRSTUVWXYZ 0123456789"
print(decodificarAfin('K7K4L', letras, 7, 20))
print(decodificarAfin(codificarAfin('ENERO', letras, 7, 12), letras, 7, 12))
```

6. Si analizamos el problema de codificado Afin, puede verse que el algoritmo de cifrado y descifrado es el mismo pero la diferencia está en la función que se aplica en el codificado y en el decodificado

   Codificado → $y = ((a * x) + b) \% m$ (I)
   Decodificado → $y = (a^{-1} * (x - b)) \% m$ (II)

Utilizando Funciones de Orden Superior, definir la función `afin` que tiene como parámetros un mensaje, un alfabeto, las claves `a` y `b` y un parámetro que por defecto vale `True` indicando que la

función debe codificar. Cuando vale False la función deberá decodificar el mensaje. Para cada caso se utilizará I o II según corresponda. (0,4p)

Nota de implementación. En la definición de la función afin no se puede hacer una llamada a codificarAfin y a decodificar Afin según el valor del parámetro, sino que deberá definirse una única función que aplique la función de codificado (I) o decodificado (II) respectivamente según el tercer parámetro de la función, usando funciones de orden superior.

afin ("ENERO", letras, 7, 20) → 'K7K4L'
afin ('K7K4L', letras, 7, 20, False) → 'ENERO
afin (afin('ENERO', letras, 7, 20), letras, 7, 20, False) → 'ENERO'

Solución

```
def afin(mensaje, alfabeto, a, b, codificar=True):
    m = len(alfabeto)
    if codificar:
        #Primero ciframos un carácter
        def transformar_char(char):
            x = alfabeto.index(char)
            y = (a * x + b) % m
            return alfabeto[y]
    else:
        #Verificamos si `a` y `m` son coprimos para descifrar
        if not sonCoprimos(a, m):
            return "ERROR. No se puede descifrar: a y m no son coprimos"
        #Calculamos la inversa de `a`
        a_inv = inversoDe(a, m)
        #Función para descifrar un carácter
        def transformar_char(char):
            x = alfabeto.index(char)
            y = (a_inv * (x - b)) % m
            return alfabeto[y]
        #Aplicar la transformación a cada carácter del mensaje
        mensaje_transformado = [transformar_char(char) for char in mensaje]

        #Unir los caracteres transformados en una cadena
        return ''.join(mensaje_transformado)

#Ejemplos:
letras = "ABCDEFGHIJKLMNÑOPQRSTUVWXYZ 0123456789"
print(afin("ENERO", letras, 7, 20))
print(afin('K7K4L', letras, 7, 20, False))
print(afin(afin('ENERO', letras, 7, 20), letras, 7, 20, False))
```

Problema 3. Colas en el banco. 2 puntos

En un banco hay varias cajas atendiendo clientes. En cada una de las cajas se ha formado una cola de personas según su orden de llegada. En determinado momento se decide cerrar todas las cajas y abrir una única para atender a todas las personas que están esperando. Para ello se forma una única cola intercalando las colas existentes según el siguiente criterio: se atenderán primero a aquellos que llegaron en primer lugar a cada una de las cajas y así sucesivamente.

Implementar la función fusionarColas que recibe como entrada una lista de colas de diferente tamaño, y devuelve como resultado una única cola donde los individuos se atienden en el mismo

orden que estaban en las colas originales, pero intercalados, empezando por la primera cola de la lista, luego la segunda y así sucesivamente. Por ejemplo, si hemos ejecutado

c1 – Cola()
c2 – Cola()
c3 – Cola()
c1 - Cola.llenar_cola(['Ana', 'Bea', 'Carmen', 'Diego'])
c2 - Cola.llenar_cola(['Paco', 'Andres', 'Carolina', 'Juan', 'Pepe'])
c3 - Cola.llenar_cola(['Luis', 'Fco']))

La fusión de las colas sale

cola1 - Cola()
cola1 - fusionarColas ([c1, c2, c3], cola1)

CAJA C1	CAJA C2		CAJA CN	→	CAJA FINAL C1-C2-C3
Ana	Paco		Luis		Ana
Bea	Andrés		Fco		Paco
Carmen	Carolina				Luis
Diego	Juan				Bea
	Pepe				Andrés
					Fco
					Carmen
					Carolina
					Diego
					Juan
					Pepe

cola1.ver_cola() ->
->fin -> ['Pepe', 'Juan', 'Diego', 'Carolina', 'Carmen', 'Fco', 'Andres', 'Bea', 'Luis', 'Paco', 'Ana'] ->
nuevoCola2 – Cola()
nuevaCola2 = fusionarColas ([c3 ,c1 ,c2], nuevaCola2)

CAJA C1	CAJA C2		CAJA CN	→	CAJA FINAL C3-C1-C2
Ana	Paco		Luis		Luis
Bea	Andrés		Fco		Ana
Carmen	Carolina				Paco
Diego	Juan				Fco
	Pepe				Bea
					Andrés
					Carmen
					Carolina
					Diego
					Juan
					Pepe

nuevaCola2.ver_cola() →
→ fin → ['Pepe', 'Juan', 'Diego', 'Carolina', 'Carmen', 'Andrés', 'Bea', 'Fco', 'Paco', 'Ana', 'Luis'] →

Solución

```
class Cola:
    def __init__(self):
    #Inicializa una lista vacía para almacenar los elementos de la cola
    self.items = []

def llenar_cola(cls, elementos):
    #Método de clase para crear una nueva instancia de Cola y llenarla con los elementos
    proporcionados
    cola = cls()  #Crea una nueva instancia de Cola
    for elemento in elementos:
        #Añade cada elemento a la cola
        cola.encolar(elemento)
    return cola  #Devuelve la instancia de Cola con los elementos añadidos

def encolar(self, elemento):
    #Añade un elemento al final de la cola mediante la función append
    self.items.append(elemento)

def desencolar(self):
    #Elimina y retorna el primer elemento de la cola si no está vacía
    if not self.esta_vacia():
        return self.items.pop(0)  #Retira y devuelve el primer elemento de la cola
    return None  #Devuelve None si la cola está vacía

def esta_vacia(self):
    #Verifica si la cola está vacía
    return len(self.items) == 0

def ver_cola(self):
    #Devuelve una representación de la cola como una lista de elementos
    return self.items

def fusionarColas(lista_de_colas, cola_final):
    #Bucle principal que sigue ejecutándose mientras alguna de las colas en lista_de_colas no esté vacía
    while any(not cola.esta_vacia() for cola in lista_de_colas):
        #Itera sobre cada cola en lista_de_colas
        for cola in lista_de_colas:
            #Si la cola actual no está vacía
            if not cola.esta_vacia():
                #Desencola el primer elemento de la cola actual y lo encola en cola_final
                cola_final.encolar(cola.desencolar())
    #Devuleve la cola final fusionada
  return cola_final

#Ejemplos:
#Crear las colas con elementos
c1 = Cola.llenar_cola(['Ana', 'Bea', 'Carmen', 'Diego'])
c2 = Cola.llenar_cola(['Paco', 'Andrés', 'Carolina', 'Juan', 'Pepe'])
c3 = Cola.llenar_cola(['Luis', 'Fco'])
```

```
#Fusionar las colas en el orden [c1, c2, c3]
cola1 = Cola()
cola1 = fusionarColas([c1, c2, c3], cola1)
print(cola1.ver_cola())

#Crear de nuevo las colas (ya que se vaciaron en el primer fusionar)
c1 = Cola.llenar_cola(['Ana', 'Bea', 'Carmen', 'Diego'])
c2 = Cola.llenar_cola(['Paco', 'Andrés', 'Carolina', 'Juan', 'Pepe'])
c3 = Cola.llenar_cola(['Luis', 'Fco'])

#Fusionar las colas en el orden [c3, c1, c2]
nuevaCola2 = Cola()
nuevaCola2 = fusionarColas([c3, c1, c2], nuevaCola2)
print(nuevaCola2.ver_cola())
```

Evaluacion continua. Polinomios y Método de Horner. (3 puntos)

Un polinomio de grado n $p(x)=a_0+a_1\,x+a_2x^2+a_3x^3+\cdots+a_nx^n$ se le puede representar en Python como una lista de longitud $n+1$ con los coeficientes de todos los términos del polinomio, de menor a mayor grado. Si alguno de los términos no existe (es nulo) aparecerá un 0 en la posición correspondiente de la lista $[a_0,a_1,a_2,a_3,\cdots,a_n]$

Por ejemplo, $f(x)=x^4+3x^3+5x^2+7x+9$ se representa como $[9,7,5,3,1]$

1. Implementar la función evaluaPolLPC que devuelva el resultado de evaluar el polinomio en para un x dado, utilizando listas por comprensión. (0,3p)

   evaluaPolLPC ([9,7,5,3,1],2) → 83

Solución

```
def evaluaPolLPC(coefs, x):
    #Devuelve el resultado de evaluar el polinomio en x sumando una lista por comprensión.
    #El parámetro x es el valor en el que se evalúa el polinomio.
    #El parámetro coef es una lista con los coeficientes del polinomio de menor a mayor grado.
    return sum(coef * (x ** i) for i, coef in enumerate(coefs))

#Ejemplo:
print(evaluaPolLPC([9, 7, 5, 3, 1], 2))
```

2. Implementar la funcion evaluaPolR que devuelva el resultado de evaluar el polinomio para un x dado, utilizando recursividad normal. (0,3p)

   evaluaPolR ([9,7,5,3,1],2) → 83

Solución

```
def evaluaPolR(coefs, x):
    #Igual que la anterior pero usando recursividad
    def _evalua(coefs, x, i):
        if i == len(coefs):
            return 0
        return coefs[i] * (x ** i) + _evalua(coefs, x, i + 1)
    return _evalua(coefs, x, 0)

#Ejemplo:
print(evaluaPolR([9, 7, 5, 3, 1], 2))
```

3. Utilizando FOS implementar la función evaluaPolFOS que devuelva el resultado de evaluar el polinomio para un x dado, utilizando funciones de orden superior. (0,3p)

   evaluaPolFOS ([9,7,5,3,1],2) → 83

Solución

```
def evaluaPolFOS(coefs, x):
    # Usa map y enumerate para multiplicar cada coeficiente por x elevado a su índice
    return sum(map(lambda i_coef: i_coef[1] * (x ** i_coef[0]), enumerate(coefs)))

#Ejemplo:
print(evaluaPolFOS([9, 7, 5, 3, 1], 2))
```

Un polinomio $p(x)=a_0+a_1 x+a_2x^2+a_3x^3+\cdots+a_nx^n$ puede escribirse de forma alternativa del siguiente modo:

$p(x)=a_0+x(a_1+a_2x^1+a_3x^2+\cdots+a_nx^{n-1})$

donde se ha sacado factor común x para los términos con grado mayor a cero. Si repetimos este proceso para el término entre paréntesis, obtenemos

$p(x)=a_0+x(a_1+x(a_2+a_3x^1+\cdots+a_nx^{n-2}))$

Haciéndolo un número suficiente de veces, llegamos a

$p(x)=a_0+x(a_1+x(a_2+x(a_3+\cdots+xa_n)))$

que se conoce como la fórmula de Horner.

Este método permite evaluar un polinomio en un punto, de forma eficiente. Un algoritmo que permite evaluar un polinomio por la fórmula de Horner es el siguiente, donde los coeficientes polinomio están ordenados de menor a mayor grado:

1. $k = n$. El grado del polinomio
2. $b_k = a_k$ Se inicializa con el coeficiente de mayor grado, b_k hace de acumulador
3. $b_{k-1} = a_{k-1} + (b_k \cdot x_0)$
4. $k = k - 1$
5. Si $k <= 0$ devuelvo b_k
6. Si no se repite a partir del paso 3

Por ejemplo, para el polinomio $f(x)=x^4+3x^3+5x^2+7x+9$ que representamos en este problema como [9,7,5,3,1] para $x \leftarrow 2$

K	
4	b4 = **1** b3 = a3 + b_4*x = 3 1*2 → 5
3	b2 = a_2 + b_3*2 = **5** + 5*2 = 5 + 10 = 15
2	b1 = a1 + b2*2 = **7** + 15*2 = 7 + 30 =37
1	b0 = a0 + b1*2 = **9** +37*2 = 9+ 74 = 83
0	Devolver b0 → 83

4. Implementar la funcion eValPoliH siguiendo la fórmula de Horner, utilizando recursividad con acumuladores, para un polinomio y un punto x, según el método descrito. (0,6p)

   evalPoliH ([1,2,3],5) → 86
   evalPoliH ([9,7,5,3,1],2) → 83

Solución

```
def evalPoliH(coefs, x, k=None, acumulador=0):
    #parametro coefs: Lista de coeficientes del polinomio, de menor a mayor grado.
    #parametro x: Valor en el que se evalúa el polinomio.
    #parametro k: Índice del coeficiente actual (utilizado en la recursión).
    #parametro acumulador: Acumulador para el resultado parcial (utilizado en la recursión).

    #Inicializar k con el índice del coeficiente de mayor grado si no se proporciona un valor inicial.
    #Establece k al índice del último elemento en la lista coefs.

    if k is None:
        k = len(coefs) - 1
    #Si k es menor que 0, significa que hemos procesado todos los coeficientes.
    #Devuelve el valor del acumulador, que contiene el resultado final de la evaluación del polinomio.
    if k < 0:
        return acumulador
    else:
        #Actualiza el acumulador usando la fórmula de Horner.
        #Multiplica el acumulador actual por x y suma el coeficiente actual coefs[k].
        acumulador = coefs[k] + x * acumulador
        #Llama a la función recursivamente con el índice k decrecido en 1 y el acumulador
        actualizado.
        return evalPoliH(coefs, x, k - 1, acumulador)

#Ejemplo:
print(evalPoliH([1, 2, 3], 5))
print(evalPoliH([9, 7, 5, 3, 1], 2))
```

5. De una función evalPoliFold que evalúe un polinomio con la fórmula de Horn utilizando la función reduce, y sin recursividad. (0,5p)

   evalPoliFold ([1,2,3],5)→86

   evalPoliFold ([9,7,5,3,1],2)→83

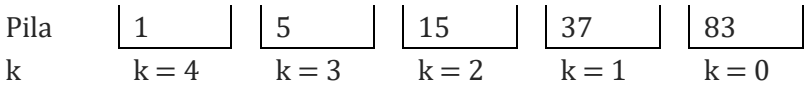
Solución

```
from functools import reduce
def evalPoliFold(coefs, x):
    #reversed(coefs): Genera una lista de los coeficientes en orden inverso (del coeficiente de mayor
    grado al de menor grado).
    #coef + x * acumulador:Implementa el paso de la fórmula de Horner, donde se multiplica el valor
    acumulado por x y se suma el coeficiente actual
    #reduce: Aplica la función lambda a los coeficientes en orden inverso, acumulando el resultado en
    cada paso
    return reduce(lambda acumulador, coef: coef + x * acumulador, reversed(coefs))

#Ejemplo:
print(evalPoliFold([1, 2, 3], 5))
print(evalPoliFold([9, 7, 5, 3, 1], 2))
```

6. Implementar la función `evalPoliHPila` que, dado un polinomio, evaluarlo utilizando el método de Horner descrito, utilizando una Pila. La función tiene como entrada un polinomio y un punto `x` donde se evalúa el polinomio. (1p)

En la pila vamos a guardar la evaluación para cada paso k del polinomio. Cuando, k valga 0, se devuelve el valor de la cima de la pila. En la figura se muestran los sucesivos valores que va tomando la pila



evalPoliHPila ([1,2,3],5) → 86
evalPoliHPila (([9,7,5,3,1],2) → 83

Solución

```
def evalPoliHPila(coefs, x):
    #Inicializamos una pila vacía para almacenar los resultados intermedios.
    pila = []
    #Iteramos sobre los coeficientes del polinomio en orden inverso (desde el término de mayor grado al de menor grado).
    for coef in reversed(coefs):
        #Si la pila está vacía, simplemente apilamos el coeficiente actual.
        if not pila:
            pila.append(coef)
        else:
            #Si la pila no está vacía, calculamos el nuevo valor apilado como el coeficiente actual + x * valor en la cima de la pila.
            pila.append(coef + x * pila[-1])
        #Al final, la cima de la pila contiene el resultado de la evaluación del polinomio.
        return pila[-1]

#Ejemplo:
print(evalPoliHPila([1, 2, 3], 5))
print(evalPoliHPila([9, 7, 5, 3, 1], 2))
print(evalPoliHPila([1, 2, 3], -5))
```