Pauta Auxiliar Extra C1

P1

a)

En una frase formal en castellano el invariante del problema seria:

- Dados dos nodos i y j, de la primera y segunda lista respectivamente, que son los que se revisan actualmente conoce la cantidad de nodos con valores distintos entre la lista que precede a i y la lista que precede a j.

b)

```
class Nodo:
    def __init__(self, val, sig):
        self.valor = val
        self.sgte = sig


def elementosDistintos(n1, n2):
    ans = 0
    while(n1 != None or n2 != None):
        if(n1 == None):
            ans += 1
            n2 = n2.sgte

        elif(n2 == None):
            ans += 1
            n1 = n1.sgte

        elif(n1.valor < n2.valor):
            ans += 1
            n1 = n1.sgte

        elif(n1.valor > n2.valor):
            ans += 1
            n2 = n2.sgte

        else:
            n1 = n1.sgte
            n2 = n2.sgte

    return ans

if __name__ == "__main__":
    a = Nodo(12, Nodo(22, Nodo(45, Nodo(67, Nodo(81, Nodo(99, None))))))
    b = Nodo(8, Nodo(17, Nodo(22, Nodo(60, Nodo(81, None)))))
    c = elementosDistintos(a,b)
    print(c)
```

```
7
```

c)

Si tenemos ambas listas, las cuales llamaremos A y B, con largo n y m respectivamente, el peor caso para nuestro algoritmo se generara en el caso de que el ultimo elemento de una lista sea menor al primer elemento de la otra list

Por ejemplo, si el ultimo elemento de la lista A es menor que el primero de B sucedera que recorreremos por comp lista A y luego comenzaremos a recorrer la lista B. Dado esto, la cantidad de pasos que se realizarian es $n + m$, p que concluimos que el algoritmo tiene una cota de tiempo $O(n + m)$.

P2

```
class ArbolBinario:
    def __init__(self, val, izq=None, der=None):
        self.info = val
        self.izq = izq
        self.der = der

def pesos(raiz):
  # caso base si la raiz es nula tiene peso 0
  if(raiz == None):
    return 0

  pesoIzq = pesos(raiz.izq)   # calculamos el peso a la izquierda
  pesoDer = pesos(raiz.der)   # lo mismo a la derecha

  if(pesoIzq == pesoDer):
    print(raiz.info)

  # se retorna el peso de los hijos mas el peso que aporta la raiz
  return pesoIzq + pesoDer + 1

if __name__ == "__main__":


    a = ArbolBinario(5,
                     ArbolBinario(3,
                                  ArbolBinario(2),
                                  ArbolBinario(4)),
                     ArbolBinario(7,
                                  ArbolBinario(6),
                                  ArbolBinario(8,
                                               None,
                                               ArbolBinario(9))))
    b = ArbolBinario(5,
                         ArbolBinario(3,
                                      ArbolBinario(2),
                                      ArbolBinario(16)),
                         ArbolBinario(8,
                                      ArbolBinario(7),
                                      ArbolBinario(9)))
    print("el peso del arbol a es:", pesos(a))
    print()
    print("el peso del arbol b es:", pesos(b))
```

```
2
4
3
6
9
el peso del arbol a es: 8

2
16
3
7
9
8
5
el peso del arbol b es: 7
```

P3

Lo primero que sabemos es que ir desde un punto a el mismo vale 0, por lo que $C(i,i) = 0$. Luego, tambien poder notar que el optimo para ir desde un punto i a un punto j sera el minimo entre $a(i,j)$ y todas las combinaciones po para ir de i a j con un punto k intermedio. Esto lo podemos formalizar en la siguiente expresion:

$C(i,j) = min(a(i,j), \min_{i<k<j}\{C(i,k) + C(k,j)\})$

La cual es la formula recursiva que nos piden encontrar.

Ahora, en esta recursion podemos aplicar tabulacion guardando los valores optimos $C(i,j)$ en forma creciente, de forma, cuando estemos revisando todas las combinaciones $C(i,k) + C(k,j)$ habran algunas que ya tendremos calculadas.

Este algoritmo tendria una cota temporal $O(N^3)$

A continuacion se muestra una implementacion de ejemplo:

```
# Recibe como parámetro una matriz (cuadrada) de n x n con los costos a(i,j) de ir desde
# un lugar i hasta otro lugar j sin pasar por puntos intermedios.
# Retorna una nueva matriz que contiene el costo óptimo C(i,j) utilizando programación dinám
# considerando paradas intermedias. Toma tiempo cúbico.
# optimal = np.zeros((n, n), dtype=int)

import numpy as np
def costo_minimo(matriz):
    n = len(matriz)
    for i in range(0, n):
        for j in range(1, n):
            opti = matriz[i][j]
            for k in range(i + 1, j + 1):
                opti = min(matriz[i][k] + matriz[k][j], opti)
            matriz[i][j] = opti
    return matriz
```