Corrigió: Mauro P

Nota: 8 (ocho)

Ej. 1	Ej. 2	Ej. 3
B=	B	B-
Falló	Ok	Ok

(completó el método en otro lado)

107587 - P3

Gallino, Pedro - Práctica Alan

» ej1.py

```
# Dada la clase `ListaEnlazada` con únicamente una referencia al primer nodo
# donde los datos son números enteros, implementar el método `completar_huecos()`,
# que agrega los nodos necesarios de forma tal de que no queden "huecos",
# o sea que todos los números sean consecutivos. La función debe ser O(N) en tiempo.
# Por ejemplo, para la lista 1 -> 4 -> 3 -> 1 -> 3, luego de completar_huecos() debería quedar:
# 1 -> 2 -> 3 -> 4 -> 3 -> 2 -> 1 -> 2 -> 3.

class _Nodo:
    def __init__(self, dato, prox=None):
        self.prox = prox
        self.dato = dato

class ListaEnlazada:

    def completar_huecos(self):

        actual = self.prim

        while actual.prox:
```

Si no hay actual (lista vacía), esto explota... lo repetimos muchas veces en clase!

```
            if actual.dato + 1 != actual.prox.dato and actual.prox.dato > actual.dato or actual.dato - 1 !=
actual.prox.dato and actual.prox.dato < actual.dato:

                if actual.prox.dato > actual.dato:
                    nodo_nuevo = _Nodo((actual.dato) + 1)
                    nodo_nuevo.prox = actual.prox
                    actual.prox = nodo_nuevo
                    actual = actual.prox
                elif actual.prox.dato < actual.dato:
                    nodo_nuevo = _Nodo((actual.dato) - 1)
                    nodo_nuevo.prox = actual.prox
                    actual.prox = nodo_nuevo
                    actual = actual.prox
            else:
                actual = actual.prox


    def __init__(self):
        self.prim = None

    # Este método **no** es parte de la lista enlazada y sólo está
    # para simplificar las pruebas
    def crear_desde(self, lista):
      for elemento in lista:
        self.append(elemento)

    # Este método **no** es parte de la lista enlazada y sólo está
    # para simplificar las pruebas
    def append(self, elemento):
        if not self.prim:
            self.prim = _Nodo(elemento)
            return

        actual = self.prim
        while actual.prox:
            actual = actual.prox
        actual.prox = _Nodo(elemento)
```

```
    # Este método **no** es parte de la lista enlazada y sólo está
    # para simplificar las pruebas
    def __eq__(self, other):
        act = self.prim
        act_other = other.prim

        while act and act_other:
            if act.dato != act_other.dato:
                return False
            act = act.prox
            act_other = act_other.prox

        if not act and act_other or not act_other and act:
            return False

        return True

    # Este método **no** es parte de la lista enlazada y sólo está
    # para simplificar las pruebas
    def __str__(self):
        '''
        Devuelve una representación en la forma {[elem_1] -> [elem_2] -> ... }
        de la lista enlazada.
        '''
        actual = self.prim

        s = '{'
        while actual:
            s += f'[{actual.dato}] -> '
            actual = actual.prox

        return s.rstrip(' -> ') + '}'

    # Completar el siguiente metodo
    def completar_huecos(self):
        pass
```

Tenías que completar el método acá. Las pruebas no te pasan porque este def te está pisando tu implementación...

```
le = ListaEnlazada()
le_res = ListaEnlazada()
le.crear_desde([1,4,3,1,3])
le_res.crear_desde([1,2,3,4,3,2,1,2,3])
le.completar_huecos()
assert le == le_res
```

» ej2.py

```
# Escribir una función `invertir_primeros_k` que recibe una cola y un numero `k`,
# e invierta el orden de los primeros `k` elementos a salir de la cola.
# Ejemplos:
# Para la cola:
# sale <| 1  2  3  4  5 |< entra
# y un k = 3, debería resultar en:
# sale <| 3  2  1  4  5 |< entra
# Para la cola:
# sale <| 1  2  3  4  5 |< entra
# y un k = 7, debería resultar en:
# sale <| 5  4  3  2  1 |< entra

from tda import Cola, Pila

#Completar la siguiente funcion
def invertir_primeros_k(cola, k):

    pila_aux = Pila()
    cola_aux = Cola()
```

```
    for i in range(k):
        if cola.esta_vacia():
            break
        pila_aux.apilar(cola.desencolar())

    while not cola.esta_vacia():
        cola_aux.encolar(cola.desencolar())

    while not pila_aux.esta_vacia():
        cola.encolar(pila_aux.desapilar())

    while not cola_aux.esta_vacia():
        cola.encolar(cola_aux.desencolar())


cola = Cola()
cola.encolar_desde([1, 2, 3, 4, 5])
cola_res = Cola()
cola_res.encolar_desde([3, 2, 1, 4, 5])
invertir_primeros_k(cola, 3)
assert cola == cola_res
```

Ni un comentario para hacer, ejercicio perfecto, felicitaciones :)

No bueno mentira, sí, uno para que aprendas algo: en el for, fijate que "i" no lo usas (el valor de la variable). Cuando es así que necesitas iterar una cantidad de veces pero no necesitas el valor, podés hacer "for _ in range(k):". Eso itera k veces, pero no guarda el valor innecesariamente

» ej3.py

```
# Dado un arreglo de enteros y un número `k`, escribir una función `obtener_suma_maxima`
# que devuelve la mayor suma posible entre `k` elementos contiguos del arreglo.
# Ejemplos:
# obtener_suma_maxima([3, 1, 9, 2, 3, 6], 3)  =>  Devuelve 14, por la suma de [9, 2, 3]
# obtener_suma_maxima([3, 1, 9, 2, 3, 6], 4)  =>  Devuelve 20, por la suma de [9, 2, 3, 6]
# obtener_suma_maxima([3, 1, 9, 2, 3, 6], 15)  =>  Devuelve 24, por la suma de todo el arreglo

from tda import Cola, Pila

#Completar la siguiente funcion
def obtener_suma_maxima(arr, k):
    cola= Cola()
    semi_cola = Cola()
    mayor_suma = 0
    contador = 0

    for indice in arr:
        cola.encolar(indice)

    for i in range(k):
        if cola.esta_vacia():
            break
        contador += 1
        semi_cola.encolar(cola.desencolar())

    if contador < k:
        while not semi_cola.esta_vacia():
            mayor_suma += semi_cola.desencolar()

        return mayor_suma

    while not cola.esta_vacia():

        suma = 0

        for i in range(k):

            dato = semi_cola.desencolar()
            semi_cola.encolar(dato)
```

```
            suma += dato

        semi_cola.desencolar()
        semi_cola.encolar(cola.desencolar())

        if suma > mayor_suma:
            mayor_suma = suma

    suma = 0

    while not semi_cola.esta_vacia():
        suma += semi_cola.desencolar()

    if suma > mayor_suma:
        mayor_suma = suma

    return mayor_suma
```

La idea general es la que buscabamos, y la utilización de la cola también. Te rebuscaste bastante con algunas cosas en el medio, pero aún así está bien logrado en general. Felicitaciones!

```
res1 = obtener_suma_maxima([3, 1, 9, 2, 3, 6], 3)
res2 = obtener_suma_maxima([3, 1, 9, 2, 3, 6], 4)
res3 = obtener_suma_maxima([3, 1, 9, 2, 3, 6], 15)
assert res1 == 14 and res2 == 20 and res3 == 24
```