```
from algo1 import *
from linkedlist import LinkedList, Node,delete, print_LL, add, length,insert, search,access

#Cambia un nodo de lugar con su siguiente
def swap(L, node):
   aux = node
   next = node.nextNode
   delete(L, node.value)
   aux.nextNode = next.nextNode
   next.nextNode = aux
   return 0

#Algoritmo de ordenamiento -bubblesort
def BubbleSort(L):
   current = L.head
   n = length(L)
   #Recorremos el arreglo n veces haciendo los "swaps" necesarios en el trayecto
   if (current is not None and current.nextNode is not None):
       for i in range(n):
           current = L.head
           while (current.nextNode is not None):
               if (current.value > current.nextNode.value):
                   swap(L,current)
               else:
                   current = current.nextNode

       return L
   else:
       return L

#borra el nodo en la posición indicada
def delete_in_pos(L,position):
   current = L.head
   for i in range(position-1):
       current = current.nextNode

   current.nextNode = current.nextNode.nextNode

#Algoritmo de ordenamiento selection sort
def SelectionSort(L):
   n = length(L)
   if n > 1:
       #Recorremos la lista y cuando encontramos el menor de todos lo trasladamos hacia la
posicion inicial
       #i = posicion inicial, i+1 por cada iteracion/valor encontrado
       for i in range(n):
           min_value = access(L,i)
           for j in range(i+1,n):
               aux = access(L,j)
               if aux <= min_value:
                   min_value = aux
                   min_position = j
           if min_position != i:
```

```
                delete_in_pos(L,min_position)
                insert(L,min_value,i)

    return L

#Algoritmo de busqueda InsertionSort
def InsertionSort(L):
    n = length(L)
    #Recorre la lista desde la posicion 0 hasta la correspondiente en cada iteracion(i)
    for i in range(1,n):
        current_val = access(L,i)
        insert_position = -1
        for j in range(i):
            aux = access(L,j)

            #Guardamos la posicion en la que debemos insertar en caso de encontrar
            #la posicion a la que pertenece
            #Usamos break para que no itere por los otros elementos, ya que podrian ser
mayores
            #y no funcionaria el algoritmo.
            if current_val <= aux:
                insert_position = j
                break
        if insert_position != -1:
            delete_in_pos(L,i)
            insert(L,current_val,insert_position)
    return L
```

EJERCICIO 2	Estabilidad	T. inPlace	Online	Rendimiento
BubbleSort	Es estable, ya que no intercambia elementos iguales si no es necesario	No requiere estructura adicional significativa. Cantidad constante de memoria extra.	Requiere tener todos los elementos disponibles desde el principio.	Mejor caso: O(n)(Lista ordenada) Peor caso: O(n^2) Caso promedio: O(n^2)
InsertionSort	Los elementos iguales también conservan su orden relativo.	No requiere estructura adicional significativa. Cantidad constante de memoria extra.	Sí, puede ordenar elementos a medida que los recibe.	Mejor caso: O(n)(Lista ordenada) Peor caso: O(n^2) Caso promedio: O(n^2), aunque es más eficiente que bubblesort para listas parcialmente ordenadas
SelectionSort	No es estable, ya que intercambia elementos incluso cuando no es necesario.(Puede alterar el orden relativo de elementos iguales)	No requiere estructura adicional significativa. Cantidad constante de memoria extra.	Requiere tener todos los elementos disponibles desde el principio.	O(n^2) para todos los casos, ya que siempre hace la misma cantidad de comparaciones

Ejercicio 3:

La respuesta correcta sería InsertionSort. Como la lista está ordenada de mayor a menor, no deberá recorrer el arreglo en cada iteración, ya que siempre deberá insertar el nodo actual en la posición 0, es decir, tendría una complejidad temporal de O(n)