

# Ejercicios LPC Algoritmos

2-2023

**Ejercicio 1** Extienda la clase Lista escribiendo el método `proc Lista<entero>::reorderOddsAndEvens()`. El cual debe reordenar una lista enlazada de manera que los nodos pares se ubiquen después de los nodos impares manteniendo el orden relativo.

Resolver utilizando SOLO APUNTADORES. El orden de complejidad debe ser a lo sumo  $O(n)$

```
proc Lista<int>::reorderOddAndEven()
Var
    pointer to Nodo<int>: pPar, pImpar, pPrimerPar
Begin
    pImpar <- instance.head
    if pImpar then
        pPar <- pImpar->getNext()
        pPrimerPar <- pPar
    endif

    while pPar ^ pImpar then
        pImpar->setNext(pPar->getNext())
        pImpar <- pImpar->getNext()
        if pImpar then
            pPar->setNext(pImpar->getNext())
            pPar <- pPar->getNext()
        endif
    endwhile

    if pPrimerPar then
        pImpar->setNext(pPrimerPar)
    endif
    if pPar then
        instance.tail <- pPar
    else
        instance.tail <- pImpar
    endif
endproc
```

**Ejercicio 3** Se requiere que cree la función `func sortedIntersect(Lista<entero> a, b): Lista<entero>`. Que reciba dos listas ordenadas ascendentemente y devuelva la intersección de ambas.

NO DEBE UTILIZAR APUNTADORES. Su solución debe ser a lo sumo  $O(n)$ .

$a = (1, 2, 3, 4, 5, 6, 7) \quad b = (4, 5, 9, 12, 15) \quad sortedIntersect(a, b) = (4, 5)$

```

func sortedIntersect(Lista<int>: a, b):Lista<int>
    Var
        Lista<int>: result
        int: valorA, valorB
    Begin
        result.construir()
        while ~a.esVacia() ^ ~b.esVacia() do
            valorA <- a.consultar(1)
            valorB <- b.consultar(1)

            if valorA = valorB then
                result.insertar(valorA, result.getLong() +1)
                a.eliminar(1)
                b.eliminar(1)
            else
                if valorA < valorB then
                    a.eliminar(1)
                else
                    b.eliminar(1)
                endif
            endif
        endwhile
        return result
    endfunc

```

## Parcial 1-2025

**Ejercicio 1** Extienda la clase lista creando el método: `proc Lista<element>::rightShift(int shift)` Que transforme la lista instancia en la misma lista rotada shift espacios a la derecha. Por ejemplo: L = 1 → 2 → 3 → 4 → 5 Entonces, si se hace L.rightShift(2) la lista resultante debería ser: L = 4 → 5 → 1 → 2 → 3

SÓLO PUEDE UTILIZAR APUNTADORES EN SU SOLUCIÓN. Su solución debe ser a lo sumo O(n).

```

proc Lista<Element>::rightShift(int: shift)
    Var
        int: trueShift, primero
        pointer to Nodo<Element>: pUltimo
    Begin
        trueShift <- shift mod instance.long
        primero <- instance.long - trueShift +1
        if trueShift > 0 then
            instance.tail->setNext(instance.primero)

            for i<- 1 to primero do
                if i = primero -1 then
                    pUltimo <- instance.head
                endif

```

```

        instance.head <- instance.head->getNext()
    endfor
    instance.tail <- pUltimo
    instance.tail->setNext(NULL)
endif
endproc

```

**Ejercicio 2** Dada una lista de enteros, se pide que cree la siguiente función: `func compact(Lista<int> list): Lista<Lista<int>>` que dada una lista ordenada con elementos repetidos, devuelva una lista comprimida. Por ejemplo:  $L = (1, 1, 1, 3, 3, 3, 5, 5)$   $\text{compact}(L) = ((1, 3), (3, 3), (5, 2))$  Básicamente, se debe devolver una lista de listas, en donde cada lista componente va a tener dos elementos, el primero es el elemento original, y el segundo es la cantidad de veces que aparece este elemento.

La solución debe ser a lo sumo  $O(n)$ . NO USAR APUNTADORES.

```

func compact(Lista<int>: target): Lista<Lista<int>>
Var
    Lista<Lista<int>>: result
    Lista<int>: aux
    int: counter
Begin
    counter <- 1
    result.construir()
    while ¬target.esVacia() do
        if target.getLong() > 1 then
            if target.consultar(1) = target.consultar(2) then
                counter <- counter +1
            else
                aux.insertar(target.consultar(2), 1)
                aux.insertar(counter, 2)
                result.insertar(aux, result.getLong() +1)
                aux.vaciar //como aux siempre tiene solo 2 elementos es O(2)
                counter <- 1
            endif
        else
            aux.insertar(target.consultar(1), 1)
            aux.insertar(counter, 2)
            result.insertar(aux, result.getLong() +1)
            aux.vaciar()
        endif
        target.eliminar(1)
    endwhile
endfunc

```

**Ejercicio 3** Dada una lista de floats y un entero positivo (window) cree la función:  $\$A=[a_1, a_2, \dots, a_n]\$$  `func rollingStandardDeviation(List<float> target, int: window): List<float>` Que calcule la desviación estándar en modalidad de ventana deslizante. Se define la desviación estándar como:  $\sigma = \sqrt{\frac{1}{n} \sum (x_i - \bar{x})^2}$

$\{k\} \sum_{i=1}^k (x_i - \mu)^2$  donde:  $\mu$  es la media de los  $k$  elementos de la ventana actual y  $x_i$  es cada elemento dentro de la ventana.

La función debe ser a lo sumo  $O(n)$ . Asuma que cuenta con la función: `func sqrt(float: x): float`  
//devuelve la raíz cuadrada de un número, complejidad  $O(1)$  NO PUEDE USAR APUNTADORES.

```
func rollingStandardDeviation(List<float> target, int: window): List<float>
    Var
        Lista<float>: result, mediaMovil
        Cola<float>: colaAux
        float: sum, num, cuadrado
    Begin
        result.construir()
        mediaMovil <- mediaMovil(target, window)
        colaAux.construir()
        sum <- 0
        while !target.esVacia() do
            num <- target.consultar(1)
            cuadrado <- num - mediaMovil.consultar(1)
            cuadrado <- cuadrado * cuadrado
            sum <- sum + cuadrado
            colaAux.encolar(cuadrado)

            if colaAux.getLong() < window then
                mediaMovil.eliminar(1)
            endif
            if colaAux.getLong() >= window then
                sum <- sum - colaAux.getFrente()
                colaAux.desencolar()
            endif
            if colaAux.getLong() = window then
                raiz <- sum/window
                raiz <- sqrt(raiz)
                result.insertar(raiz, result.getLong() + 1)
            endif
            target.eliminar(1)
        endwhile

        return result
    endfunc
```

## Parcial 2-2025 mod1

**Ejercicio 1** Dada una lista enlazada que contiene números enteros. De dicha lista se sabe que:

- a. Todos los elementos en posiciones pares (2, 4, ...) ya están ordenados de menor a mayor.
- b. Todos los elementos en posiciones impares (1, 3, 5, ...) también están ordenados de menor a mayor.

Extienda la clase lista con el método: `func Lista<int>::partialReorder(): Lista<int>` Que

devuelva una lista nueva con los elementos de la lista original, pero ordenados de menor a mayor.

La complejidad de esta función debe ser a lo sumo de O(n). DEBE USAR SOLO APUNTADORES EN SU SOLUCIÓN.

```
func Lista<int>::partialReorder():Lista<int>
Var
    pointer to Nodo<int>: pPar, pImpar
    Lista<int>: result
    int: i
Begin
    //Verificación si la lista está vacía
    if ~instance.head then
        return result
    endif

    result.construir()
    i <- 1
    pImpar <- instance.head
    pPar <- pPar->.getNext()

    //Comprobación para toda la lista en caso promedio
    while pImpar ^ pPar do
        if pImpar->.getInfo() <= pPar->.getInfo() then
            result.insertar(pImpar, i)
            pImpar <- pImpar->.getNext()
            if pImpar then
                pImpar <- pImpar->.getNext()
            endif
        else
            if pPar->.getInfo() < pImpar->.getInfo() then
                result.insertar(pPar,i)
                pPar <- pPar->.getNext()
                if pPar then
                    pPar <- pPar->.getNext()
                endif
            endif
        endif
        i <- i+1
    endwhile

    // Continuación en casos borde posiblemente opcionales porque el enunciado no
    // lo especifica: La lista solo tiene un elemento, la lista tiene mas numeros pares que
    // impares o viceversa
    if result.long = 0 then
        result.insertar(pImpar, 0)
    else
        while pImpar do
            result.insertar(pImpar, i)
```

```

pImpar <- pImpar->getNext()
if pImpar then
    pImpar <- pImpar->getNext()
endif
i <- i+1
endWhile
while pPar do
    result.insertar(pPar, i)
    pPar <- pPar->getNext()
    if pPar then
        pPar <- pPar->getNext()
    endif
    i <- i+1
endWhile
return result
endfunc

Private proc Lista<int>::insertar(pointer to Nodo<int>: p, int: i)
Var
    pointer to Nodo<int>: pNew, pNewActual
Begin
    if i=0 then
        instance.head <- new(Nodo<int>)
        instance.head->setInfo(p->getInfo())
        instance.tail <- instance.head
        instance.long <- 1
    else
        pNew <- new(Nodo<int>)
        pNew->setInfo(p->getInfo())
        instance.tail->setNext(pNew)
        instance.tail <- pNew
        instance.long <- instance.long +1
    endif
endproc

```

**Ejercicio 3** Dada una lista de números enteros, escriba la función: `func trendChangesInWindows(List<int>: target, int k): Lista<int>` Que reciba un entero k y devuelva una nueva lista donde cada posición indique cuántas veces la secuencia de números dentro de la ventana cambia de tendencia, es decir, de creciente a decreciente o de decreciente a creciente. Ejemplo: Entrada: [1, 3, 2, 4, 3, 5], k = 4 Salida: [2, 2, 2]

NO DEBE USAR APUNTADORES EN SU SOLUCIÓN. Su solución debe ser a lo sumo O(n)

```

func trendChangesInWindow(List<int>: target, int: k): Lista<int>
Var
    Lista<int>: result
    Cola<int>: window
    int: i, actual, counter
    Cola<int>: signos

```

```

    int: signoActual, s1, s2
Begin
    result.construir()
    window.construir()
    signos.construir()
    counter <- 0

    //Calcular la primera ventana
    for i <- 1 to k do
        actual <- target.consultar(1)
        target.eliminar(1)

        if window.getLong() >= 1 then
            signoActual <- 0
            //Condición de aumento del contador
            if window.getUltimo() < actual then
                signoActual <- 1
            endif
            if window.getUltimo() > actual then
                signoActual <- -1
            endif
            if signos.getLong() > 0 then
                s1 <- signos.getUltimo()
                if (s1 = 1 ^ signoActual = -1) v (s1 = -1 ^ signoActual = 1) then
                    counter <- counter + 1
                endif
            endif
            signos.encolar(signoActual)
            window.encolar(actual)

        endif
    endfor
    result.insertar(counter, 1)

    while ¬target.esVacia() do
        window.desencolar()

        s1 <- signos.getFrente()
        signos.desencolar()
        s2 <- signos.getFrente()

        if (s1 = 1 ^ s2 = -1) v (s1 = -1 ^ s2 = 1) then
            counter <- counter - 1
        endif

        actual <- target.consultar(1)
        target.eliminar(1)
        signoActual <- 0
        if window.getUltimo() < actual then
            signoActual <- 1

```

```

        endif
        if window.getUltimo() > actual then
            signoActual <- -1
        endif

        s1 <- signos.getUltimo()
        if (s1 = 1 ^ signoActual = -1) v (s1 = -1 ^ signoActual = 1) then
            counter <- counter +1
        endif

        signos.encolar(signoActual)
        window.encolar(actual)
        result.insertar(counter, result.getLong() +1)
    endwhile

    return result
endfunc

```

## Parcial 2-2025 mod2

**Ejercicio 1** Dada una lista enlazada que contiene números enteros. De dicha lista se sabe que:

- a. La primera mitad de la lista (primeros  $n/2$  elementos) está ordenada de menor a mayor.
- b. La segunda mitad de la lista (últimos  $n/2$  elementos) está ordenada de menor a mayor. Extienda la clase lista con el método: `func Lista<int>::partialReorder(): Lista<int>` Que devuelva una lista nueva con los elementos de la lista original, pero ordenados de menor a mayor.

La complejidad de esta función debe ser a lo sumo de  $O(n)$ . DEBE USAR SOLO APUNTADORES EN SU SOLUCIÓN.

```

func Lista<int>::partialReorder(): Lista<int>
    Var
        Lista<int>: result
        pointer to Nodo<int>: p1, p2
        int: i, mitad
    Begin
        result.construir()
        if ~instance.head then
            return result
        endif

        mitad <- instance.long / 2

        p1 <- instance.head
        p2 <- instance.head
        for i <- 1 to mitad do
            p2 <- p2->getNext()
        endfor

```

```

// Merge de [head .. antes de p2] con [p2 .. tail]
while p1 != p2 ^ p2 do
    if p1->getInfo() <= p2->getInfo() then
        result.insertar(p1, result.getLong() + 1)
        p1 <- p1->getNext()
    else
        result.insertar(p2, result.getLong() + 1)
        p2 <- p2->getNext()
    endif
endwhile

while p1 != p2 do
    result.insertar(p1, result.getLong() + 1)
    p1 <- p1->getNext()
endwhile

while p2 do
    result.insertar(p2, result.getLong() + 1)
    p2 <- p2->getNext()
endwhile

return result
endfunc

```

**Ejercicio 2** Dada una lista enlazada de números enteros, escriba la función: `func`

`longestParityAlternatingSublist(Lista<int>: target): int` Que reciba una lista de enteros target y devuelva la longitud de la sublista contigua más larga donde los números alternan entre ser pares e impares.

Ejemplo: Entrada: [1, 3, 2, 4, 3, 5, 5, 2] Salida: 6

**Ejercicio 3** Dada una lista de números enteros, escriba la función: `func`

`localInversionsInWindows(Lista<int>: target, int k): Lista<int>` Que reciba un entero k y devuelva una nueva lista donde cada posición indique cuántas **inversiones locales** existen dentro de la ventana contigua de tamaño k. Una inversión local es un par de elementos consecutivos donde el primero es mayor que el segundo. Ejemplo: Entrada: [4, 3, 2, 1, 2, 3], k = 4 Salida: [3, 2, 1]

NO DEBE USAR APUNTADORES EN SU SOLUCIÓN. Su solución debe ser a lo sumo O(n)

### Resolución del ejercicio

```

func localInversionsInWindows(Lista<int>: target, int: k): Lista<int>
Var
    Lista<int>: result
    Cola<int>: window

```

```

        int: i, actual, counter, primeraVentana
Begin
    window.construir()
    //Calcular la primera inversion
    counter <- 0
    for i <- 1 to k do
        actual <- target.consultar(1)
        target.eliminar(1)
        if window.getLong() >= 1 then
            if window.getUltimo() > actual then
                counter <- counter +1
            endif
        endif

        window.encolar(actual)
    endfor
    result.insertar(counter, 1)

    //Calcular las demas inversiones
    while ¬target.esVacia() do
        primeraVentana <- window.getFrente()
        actual <- target.consultar(1)
        window.desencolar()
        target.eliminar(1)

        if primeraVentana > window.getFrente() then
            counter <- counter -1
        endif
        if window.getUltimo() > actual then
            counter <- counter +1
        endif

        window.encolar(actual)
        result.insertar(counter, result.getLong()+1)
    endwhile

    return result
endfunc

```

## Método expand

Método contrario al compact que convierte una lista de listas en una lista donde unifica los valores repetidos de las sublistas... Pasa de L=( (1,3), (3,2), (5,1), (6,3) ) a L=( 1, 1, 1, 3, 3, 5, 6, 6, 6 )

```

func expand(Lista<Lista<int>>: target): Lista<int>
Var
    Lista<int>: result
    int: elem, reps

```

```

Begin
    result.construir()
    while ¬target.esVacia() do
        elem <- target.consultar(1).consultar(1)
        reps <- target.consultar(1).consultar(2)

        for i <- 1 to reps do
            result.insertar(elem, result.getLong() +1)
        endfor
        target.eliminar(1)
    endwhile

    return result
endfunc

```

## Método media móvil

Método explicado en clase para resolver la desviación estándar móvil en un parcial

```

func mediaMovil(Lista<float>: target, int: window): Lista<float>
Var
    Lista<float>: result
    Cola<float>: cola
    float: sum, num
    int: i
Begin
    result.construct()
    cola.construct()
    sum <- 0
    i <- 1
    while ¬target.esVacia() do
        num <- target.consultar(1)
        cola.encolar(num)
        sum <- sum + num
        if cola.getLong() > window then
            sum <- sum-cola.getFrente()
            cola.desencolar()
        endif
        if cola.getLong() = window then
            result.insertar(sum/window, result.getLong() +1)
        endif

        target.eliminar(1)
    endwhile
    return result
endfunc

```