TP 1 - Complejidad - Algoritmos 2 - LCC

Tomás Rando - 14004

1)

Si el orden de complejidad de una función es $O(n^2)$ significa que para todo valor de X, la función estará por debajo del nivel del orden de complejidad. Por ello, se plantea la inecuación:

$6n^3 <= cn^2$

Resolviendo

$n <= c/6$

Es decir, no se cumple para todo n >= 0 y solo se cumplirá cuando n sea menor a c/6. Por ello, no es de $O(n^2)$.

2)

[5, 2, 8, 3, 6, 9, 1, 4, 7, 10]

Tomando como pivote el primer elemento, el orden de complejidad será de O(n log n) ya que se dividirá a la lista siempre en dos listas iguales

3)

InsertionSort tendrá complejidad de O(n)

QuickSort tendrá complejidad de $O(n^2)$

MergeSort tendrá complejidad de O(n log n)

4)

```
#Queda de complejidad O(n^2) ya que en la
#ultima parte hay un for que tiene dentro un swap.
#Por lo que quedaria de n * n, es decir, n cuadrado
def ordenamiento(L):
    if L.head == None:
        return None
    len = length(L)

    if len == 1:
        return L

    currentNode = L.head
    #n/2
    medio = round(len/2)
    for i in range(0, medio):
        if i == medio - 1:
            valor = currentNode.value
        currentNode = currentNode.nextNode
    currentNode = L.head
    menorBef = []
    menorAft = []
    mayorBef = []
    mayorAft = []
    contBefore = 0
    contAfter = 0
    #n
    for i in range(0, len):
        if currentNode.value < valor:
            if i < medio - 1:
                menorBef.append(i)
                contBefore = contBefore + 1
            else:
                if i > medio - 1:
                    menorAft.append(i)
                    contAfter = contAfter + 1
        if (currentNode.value >= valor) and (i != medio - 1):
            if i < medio - 1:
                mayorBef.append(i)
            else:
                mayorAft.append(i)
        currentNode = currentNode.nextNode
    difference = abs(contBefore - contAfter)
```

```
#n^2 (Al ser la funcion swap n)
if difference == 0 or difference == 1:
    return L
else:
    iteraciones = math.trunc(difference / 2)
    for i in range(0, iteraciones):
        if contBefore > contAfter:
            swap(L, menorBef[0], mayorAft[0])
            menorBef.pop(0)
            mayorAft.pop(0)
        else:
            swap(L, menorAft[0], mayorBef[0])
            menorAft.pop(0)
            mayorBef.pop(0)
return L
```

Básicamente, se recorre la mitad de la lista para encontrar el elemento del medio. Ese elemento se guarda. Se vuelve a recorrer la lista nuevamente, pero comparando el valor de cada nodo con el elemento guardado anteriormente. Se van guardando los índices de los nodos según sea mayor o menor y según esté antes o después del elemento central. Además, se incrementan dos contadores si se encuentran elementos menores al valor central (antes o después de este). Finalizado esto se restan los dos contadores, si en valor absoluto la resta da 0 o 1, se devuelve la lista, caso contrario, se "arregla" la lista con la operación swap aplicada una cantidad de veces definida por la resta de los contadores dividido 2.

5)

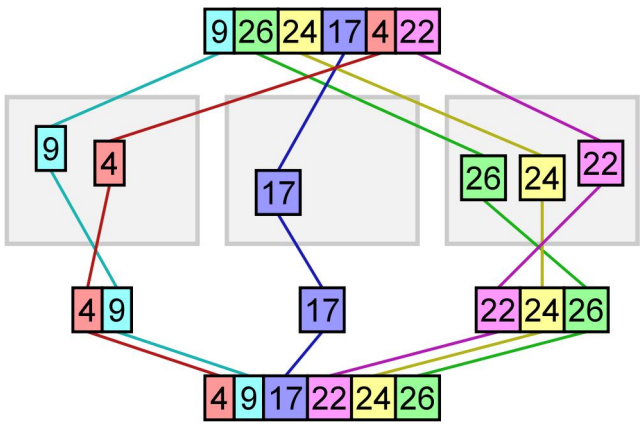
```
#Es O(n^2) ya que contiene dos bucles que recorren la lista uno dentro de otro.
def ContieneSuma(A, n):
    len = length(A)
    currentNode = A.head
    for i in range(0, len):
        number = n - currentNode.value
        currentNode2 = currentNode.nextNode
        for j in range(i + 1, len):
            if currentNode2.value == number:
                return True
            else:
                if currentNode2 != None:
                    currentNode2 = currentNode2.nextNode
        if currentNode != None:
            currentNode = currentNode.nextNode

    return False
```

6)

BucketSort:
Se dividen rangos de valores en varios baldes y se distribuyen los elementos que se desea ordenar en los diferentes baldes dependiendo del valor que tengan.
Se verifica para cada número el valor y se determina el balde que corresponda. A lo último, cada balde se ordena individualmente usando otro algoritmo de ordenamiento (Se pueden utilizar los ya vistos en algo1). Al final, se unen los elementos para obtener la lista ordenada

Siendo n el número de elementos a ordenar y k el número de baldes.
Para el mejor caso se tiene una complejidad de O(n + k). Este caso se da
Para el peor caso se tiene una complejidad de $O(n^2)$. Este caso se da cuando todos los elementos son puestos en el mismo balde y se tiene que utilizar otro algoritmo para ordenarlo
Para el caso promedio se tiene una complejidad de O(n + k)

El mejor caso y el caso promedio se dan cuando en los baldes quedan insertados los valores uniformemente. La diferencia es que el mejor caso se da cuando dentro de cada balde quedan los elementos ya ordenados.



En la imagen anterior es posible observar como se van dividiendo los elementos en los 3 “baldes” correspondientes. Se ordena cada uno por separado y al final se unen los 3, dando como resultado la lista ordenada.

7)