## Ejercicio 1:

Demuestre que $6n^3 \neq \boldsymbol{O}(n^2)$.

## Ejercicio 2:

El mejor caso para Quicksort(n) es de O(n * log(n)) y un ejemplo de array de números para que se cumpla tiene que ser de [1,2,3,4,5,6,7,8,9,10]. En cada iteración se divide el arreglo en dos partes aproximadamente iguales, lo que garantiza un rendimiento óptimo de Quicksort(n).

## Ejercicio 3:

Quicksort(A): $O(n^2)$

Insertion-Sort(A): $O(n)$

Merge-Sort(A): $O(n \,.\, log(n))$

## Ejercicio 4:

```python
def orderHalfMinors(A):
  lenA = lengthList(A)-1
  midPosition = math.trunc(lenA/2)
  midValue = accessList(A, midPosition)
  minorsFirstHalf = 0
  minorsSecondHalf = 0
  listMayorFirstHalf = []
  listMinorFirstHalf = []
  listMayorSecondHalf = []
  listMinorSecondHalf = []

  minorsFirstHalf, minorsSecondHalf = checkMinorsBothSides(A, midValue, midPosition, minorsFirstHalf, minorsSecondHalf)
  listMinorFirstHalf, listMinorSecondHalf, listMayorFirstHalf, listMayorSecondHalf = updatePositionLists(A, midPosition, midValue, listMinorFirstHalf,

  differenceMinors = minorsFirstHalf - minorsSecondHalf
  print("Pivote: ", midValue, ". Posición: ", midPosition)

  #A partir de la diferencia entre los menores que se encuentren a la izquierda y a la derecha del pivote,
  #a partir del lado que tienga mas menores realizo intercambios de estos con los mayores que se encuentren del otro lado
  while differenceMinors != 0 and differenceMinors != 1 and differenceMinors != -1:
    if minorsFirstHalf > minorsSecondHalf:
      swapNodes(A, listMinorFirstHalf[0], listMayorSecondHalf[0])
      listMinorSecondHalf.insert(0, listMinorFirstHalf.pop(0))
      listMayorFirstHalf.insert(0, listMayorSecondHalf.pop(0))
      minorsFirstHalf = 0
      minorsSecondHalf = 0
      minorsFirstHalf, minorsSecondHalf = checkMinorsBothSides(A, midValue, midPosition, minorsFirstHalf, minorsSecondHalf)
      differenceMinors = minorsFirstHalf - minorsSecondHalf
    elif minorsFirstHalf < minorsSecondHalf:
      swapNodes(A, listMayorFirstHalf[0], listMinorSecondHalf[0])
      listMinorFirstHalf.insert(0, listMinorSecondHalf.pop(0))
      listMayorSecondHalf.insert(0, listMayorFirstHalf.pop(0))
      minorsFirstHalf = 0
      minorsSecondHalf = 0
      minorsFirstHalf, minorsSecondHalf = checkMinorsBothSides(A, midValue, midPosition, minorsFirstHalf, minorsSecondHalf)
      differenceMinors = minorsFirstHalf - minorsSecondHalf
    else:
      return A
  else:
    return A
```

```python
def checkMinorsBothSides(A, midValue, midPos, minorsFirstHalf, minorsSecondHalf):
  currentNode1 = A.head
  lenA = lengthList(A)
  #Calculo cantidad de menores al valor del medio que hay en la primera mitad
  for i in range(midPos):
    if currentNode1.value < midValue:
      minorsFirstHalf += 1
    currentNode1 = currentNode1.nextNode
  currentNode1 = currentNode1.nextNode

  #Calculo cantidad de menores al valor del medio que hay en la segunda mitad
  for j in range(midPos+1, lenA):
    if currentNode1.value < midValue:
      minorsSecondHalf += 1
    currentNode1 = currentNode1.nextNode
  currentNode1 = A.head

  return minorsFirstHalf, minorsSecondHalf

def updatePositionLists(A, midPosition, midValue, listMinorFirstHalf, listMinorSecondHalf, listMayorFirstHalf, listMayorSecondHalf):
  currentNode1 = A.head
  position = 0
  lenA = lengthList(A)

  #Asigno a 1 lista las posiciones de los valores mayores al valor del medio de la primera mitad y a la otra lista y los valores menores
  for k in range(midPosition):
    if currentNode1.value < midValue:
      listMinorFirstHalf.append(position)
      position += 1
    elif currentNode1.value >= midValue:
      listMayorFirstHalf.append(position)
      position += 1
    currentNode1 = currentNode1.nextNode

  currentNode1 = currentNode1.nextNode
  position += 1

  #Asigno a 1 lista las posiciones de los valores mayores al valor del medio de la segunda mitad y a la otra lista y los valores menores
  for k in range(midPosition+1, lenA):
    if currentNode1.value < midValue:
      listMinorSecondHalf.append(position)
      position += 1
    elif currentNode1.value >= midValue:
      listMayorSecondHalf.append(position)
      position += 1
    currentNode1 = currentNode1.nextNode
  currentNode1 = A.head

  return listMinorFirstHalf, listMinorSecondHalf, listMayorFirstHalf, listMayorSecondHalf
```

## Ejercicio 5:

```python
def ContieneSuma(A, n):
  lenA = lengthList(A)
  lenB = lengthList(A)
  currentNode = A.head
  currentNode2 = A.head

  for i in range(lenA):
    valor = currentNode2.value
    for j in range(lenB-1):
      currentNode = currentNode.nextNode
      if valor + currentNode.value == n:
        return True
    currentNode2 = currentNode2.nextNode
    currentNode = currentNode2
    lenB = lenB - 1
  return False
```

El costo computacional es $O(n^2)$

## Ejercicio 6:

BucketSort es un algoritmo de ordenamiento por casilleros, en el que distribuye todos los elementos a ordenar en un número finito de “baldes”, siguiendo determinadas condiciones, como el valor del elemento.

Se verifica el valor del elemento y se le asigna a uno de los baldes. Dentro de estos los elementos son ordenados con algún otro algoritmo de ordenamiento, como InsertionSort, y luego son insertados nuevamente en la lista final respetando el orden de los baldes.

En el caso promedio, si lo s elementos se distribuyen uniformemente entre los baldes y se utiliza un algoritmo de clasificación eficiente, la complejidad del caso promedio es O(nlog(n))

Para el mejor caso, si todos los elementos se asignan al mismo balde y se ordenan usando un algoritmo de ordenamiento con complejidad lineal, entonces su orden de complejidad es O(n)

Para el peor casO, si todos los elementos se asignan al mismo balde y se ordenan usando un algoritmo de ordenamiento con mayor complejidad, su orden de complejidad puede llegar a O(n^2)

## Ejercicio 7:

A partir de las siguientes ecuaciones de recurrencia, encontrar la complejidad expresada en Θ(n) y ordenarlas de forma ascendente respecto a la velocidad de crecimiento. Asumiendo que T(n) es constante para n <= 2. Resolver 3 de ellas con el método maestro completo: $T(n) = a\ T(n/b) + f(n)$ y otros 3 con el método maestro simplificado: $T(n) = a\ T(n/b) + n^c$

a. $T(n) = 2T(n/2) + n^4$
b. $T(n) = 2T(7n/10) + n$
c. $T(n) = 16T(n/4) + n^2$
d. $T(n) = 7T(n/3) + n^2$
e. $T(n) = 7T(n/2) + n^2$
f. $T(n) = 2T(n/4) + \sqrt{n}$