

Federico lucero - 13997

<https://replit.com/@LuceroFederico/1Ejercitacion-Repaso-Complejidad-y-Metodo-Maestro#main.py>

Ejercicio 1:

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

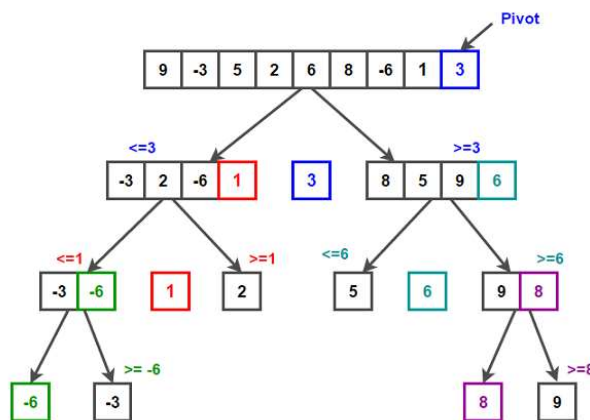
“Una función $f(n) \in \theta(g(n))$ si existen dos constantes reales positivas c_1, c_2 tales que $f(n)$ esté acotada inferiormente por $c_1 g(n)$ y superiormente por $c_2 g(n)$ para un n suficientemente grande.”

Solución: Nótese que c_1 existe para $n_0 \geq 0$. Aquí debemos probar que c_2 no existe.

Prueba por contradicción: Suponemos que c_2 y n_0 existen, por lo que: $6n^3 \leq c_2 n^2$ para todo $n \geq n_0$. Pero entonces $n \leq c_2/6$, lo que no es necesariamente cierto para un n suficientemente grande porque c_2 es constante. Entonces contradice el hecho supuesto $n \geq n_0$.

Ejercicio 2:

¿Cómo sería un array de números (mínimo 10 elementos) para el mejor caso de la estrategia de ordenación Quicksort(n) ?



En el mejor de los casos, la complejidad temporal de Quicksort es $O(n \log(n))$. El mejor caso ocurre cuando el pivote divide la array en dos partes casi iguales. Ahora cada llamada recursiva procesa una lista de la mitad del tamaño. $T(n) = 2 T(n/2) + cn = O(n \log(n))$

Ejercicio 3:

Cuál es el tiempo de ejecución de la estrategia Quicksort(A), Insertion-Sort(A) y Merge-Sort(A) cuando todos los elementos del array A tienen el mismo valor?

Quicksort(A) $O(n^2)$ El peor caso del quicksort ocurre cuando los datos de entrada están ya ordenados o inversamente ordenados. En estos casos, todo el tiempo simplemente se está seleccionando el extremo

Insertion-Sort(A) $O(n)$ En el mejor de los casos, el arreglo está inicialmente en orden, el algoritmo solo hace una pasada

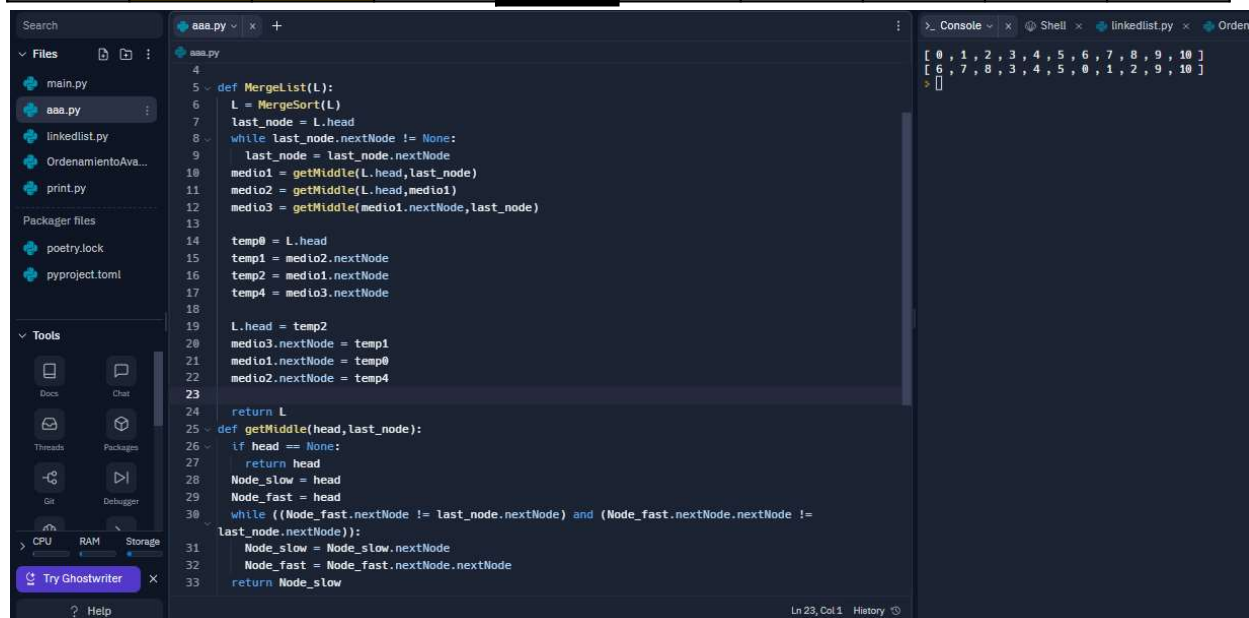
Merge-Sort(A) $O(n \log n)$ En el mejor caso, peor caso y caso promedio siempre es $O(n \log n)$

Ejercicio 4:

Implementar un algoritmo que ordene una lista de elementos donde siempre el elemento del medio de la lista contiene antes que él en la lista la mitad de los elementos menores que él. Explique la estrategia de ordenación utilizada.

Ejemplo de lista de salida

7	3	2	8	5	4	1	6	10	9
---	---	---	---	---	---	---	---	----	---



```
4
5 def MergeList(L):
6     L = MergeSort(L)
7     last_node = L.head
8     while last_node.nextNode != None:
9         last_node = last_node.nextNode
10    medio1 = getMiddle(L.head, last_node)
11    medio2 = getMiddle(L.head, medio1)
12    medio3 = getMiddle(medio1.nextNode, last_node)
13
14    temp0 = L.head
15    temp1 = medio2.nextNode
16    temp2 = medio1.nextNode
17    temp4 = medio3.nextNode
18
19    L.head = temp2
20    medio3.nextNode = temp1
21    medio1.nextNode = temp0
22    medio2.nextNode = temp4
23
24    return L
25
26 def getMiddle(head, last_node):
27     if head == None:
28         return head
29     Node_slow = head
30     Node_fast = head
31     while ((Node_fast.nextNode != last_node.nextNode) and (Node_fast.nextNode.nextNode !=
32     last_node.nextNode)):
33         Node_slow = Node_slow.nextNode
34         Node_fast = Node_fast.nextNode.nextNode
35     return Node_slow
```

Console output:

```
[ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]
[ 6, 7, 8, 3, 4, 5, 0, 1, 2, 9, 10 ]
> []
```

Primero ordena una lista utilizando mergesort, luego encuentra el medio de la lista, luego encuentra el medio de la primera mitad y después el medio de la segunda mitad, luego se intercambia la mitad de la primera mitad con la mitad de la segunda mitad.



Ejercicio 5:

Implementar un algoritmo `Contiene-Suma(A,n)` que recibe una lista de enteros `A` y un entero `n` y devuelve `True` si existen en `A` un par de elementos que sumados den `n`. Analice el costo computacional.

```
ejercicios.py
34
35 """ Ejercicio 5: Implementar un algoritmo Contiene-Suma(A,n) que recibe una lista de enteros A y
un entero n y devuelve True si existen en A un par de elementos que sumados den n. Analice el
costo computacional. """
36
37 def ContieneSuma(A,n):
38     node1 = A.head
39     node2 = A.head
40
41     while node1.nextNode != None:
42         while node2.nextNode != None:
43             if node1.value + node2.value == n:
44                 return True
45             node2 = node2.nextNode
46         node1 = node1.nextNode
47     return False
```

```
> Console
[ 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , 10 ]
[ 6 , 7 , 8 , 3 , 4 , 5 , 0 , 1 , 2 , 9 , 10 ]
[ 0 , 1 , 2 , 3 , 4 , 5 ]
True
> []
```

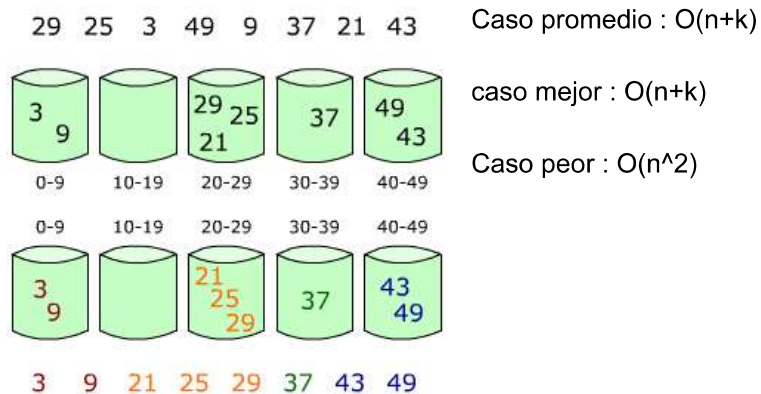
Costo computacional: $O(n^2)$

Ejercicio 6:

Investigar otro algoritmo de ordenamiento como BucketSort, HeapSort o RadixSort, brindando un ejemplo que explique su funcionamiento en un caso promedio. Mencionar su orden y explicar sus casos promedio, mejor y peor.

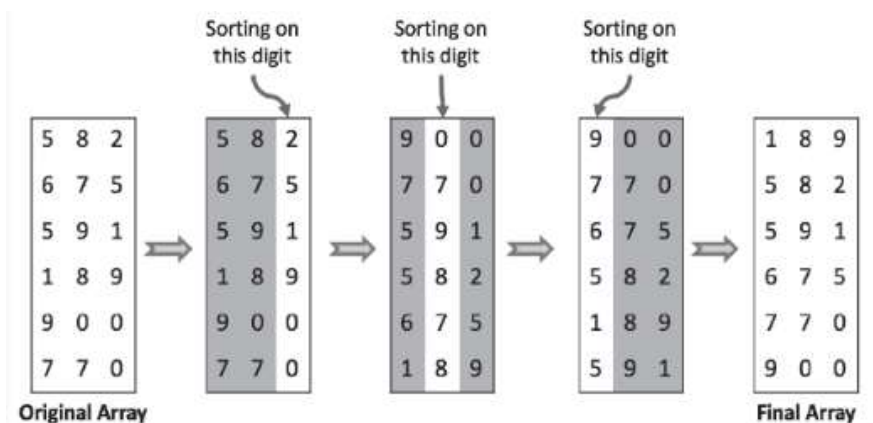
BucketSort

El ordenamiento por casilleros (bucket sort o bin sort, en inglés) es un algoritmo de ordenamiento que distribuye todos los elementos a ordenar entre un número finito de casilleros. Cada casillero solo puede contener los elementos que cumplan unas determinadas condiciones. En el ejemplo esas condiciones son intervalos de números



RadixSort

El ordenamiento Radix es un algoritmo de ordenación no comparativo. Este algoritmo evita las comparaciones insertando elementos en cubos de acuerdo con el radix (Radix/Base es el número de dígitos únicos utilizados para representar números. Por ejemplo, los números decimales tienen diez dígitos únicos). Ordena los elementos basándose en los dígitos de los elementos individuales. Realiza la ordenación por conteo de los dígitos desde el menos significativo hasta el más significativo.



Caso promedio : $O(nk)$

caso mejor : $O(nk)$

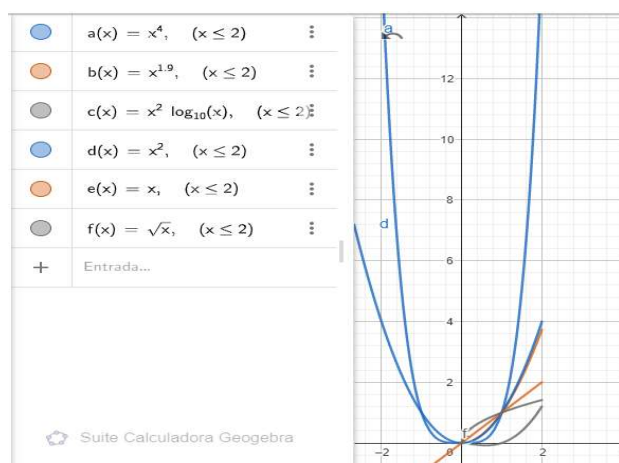
Caso peor : $O(nk)$

Ejercicio 7:

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

$$T(n) = aT(n/b) + n^c$$

ordenadas de forma ascendente respecto a la velocidad de crecimiento: c,f,e,b,d,a



método maestro completo:

a. $T(n) = 2T(n/2) + n^4$



b. $T(n) = 2T(7n/10) + n$



c. $T(n) = 16T(n/4) + n^2$



método maestro simplificado:

d. $T(n) = 7T(n/3) + n^2$



e. $T(n) = 7T(n/2) + n^2$



f. $T(n) = 2T(n/4) + \sqrt{n}$



A tener en cuenta:

1. Usen lápiz y papel primero
2. ~~No se puede utilizar otra Biblioteca mas alla de algo1.py y linkedlist.py~~
3. Hacer una análisis por cada algoritmo implementado del caso mejor, el caso peor y una perspectiva del caso promedio.