Trabajo Practico 1

Ejercicio_1:

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

Para demostrar que $6n^3$!= $O(n^2)$, primero debemos entender que significa la notación $O(n^2)$ y como se relaciona con la función $6n^3$.

Notación O(n^2): En análisis de algoritmos, O(n^2) representa una cota superior asintótica para el tiempo de ejecución de un algoritmo. Significa que el tiempo de ejecución del algoritmo crece a lo sumo proporcionalmente a n^2 (donde n es el tamaño del problema). En otras palabras, el algoritmo tiene una complejidad cuadrática.

Función 6n^3: La función 6n^3 es una función cúbica con un coeficiente constante de 6. Su forma general es an^3, donde a es una constante.

1. Definición de O(n^2): Decimos que f(n) = O(n^2) si existe una constante positiva c y un valor de n0 tal que para todo n ≥ n0, se cumple que f(n) ≤ c*n^2.

2. Comparación con 6n^3: Si tomamos f(n) = 6n^3, vemos que f(n) ≠ O(n^2). Esto se debe a que no podemos encontrar una constante c y un valor de n0 tal que 6n^3 ≤ c*n^2 para todo n ≥ n0. La función 6n^3 crece más rápido que cualquier función cuadrática, por lo que no podemos establecer una relación de orden entre ellas. En resumen, 6n^3 no está contenido en O(n^2); por lo tanto, 6n^3 ≠ O(n^2).

Ejercicio_2:

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

Tomemos como pivote el primer elemento del arreglo, luego de resituar los elementos del arreglo el pivote me queda en el centro de la lista entonces me quedaría 2 partes iguales del lado derecho e izquierdo del pivote teniendo una lista en el centro el pivote

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

En el mejor de los casos la complejidad me quedaría 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**: Potencialmente $O(n2)$ a menos que se implementen optimizaciones.
- **Insertion-Sort**: $O(n)$ debido a la falta de intercambios, aunque hace muchas comparaciones.
- **Merge-Sort**: Mantiene su eficiencia en $O(n\log n)$.

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.

```
"""
Estrategia
Para implementar un algoritmo que ordene una lista de elementos segun el criterio especifico,
podemos segir los sigirntes pasos
1-Encontrar el elemento del medio para eso vamos a usar el QuickSelect que dado una posicion k nos
dara el elemento de esa pocicion
si fuese estar ordenada.
2-Una vez encontrada el elemento del medio por QuickSelect tendemos a la derecha de la pocicion k
los elementos menores al valor
y a la derecha los elementos mayores al valor
3-Dividiremos la sub lista izquierda y derecha del valor central a la mitad y colocaremos a la derecha
la mitad mayor y la mitad menor y
hacemos lo mismo a la derecha del valor
"""

def QuickSelect(lst,k):
"""
Encuentra el k-esimo elemento mas pequeño en la lista 'lst' usando el
metodo QuickSelect
"""
if len(lst) == 1:
return lst[0]
lst[len(lst)-1],lst[k]=lst[k],lst[len(lst)-1]
pos = partition(lst, 0, len(lst)-1)
if k == pos:
return lst[pos]
elif k < pos:
return QuickSelect(lst[:pos], k)
else:
return QuickSelect(lst[pos + 1:], k - pos - 1)

def partition(lst,left,right):
"""
Esta funcion toma el ultimo elemento como pivote, coloca el elemento
pivote en su posicion correcta en la lista ordenada,
y coloca todos los elementos menores (menores que el pivote) a la
izquierda del pivote y todos los elementos mayores a la derecha del
pivote
"""
pivot = lst[right]
i = left
for j in range(left,right):
if lst[j] < pivot:
lst[i],lst[j]=lst[j],lst[i]
i=i+1
```

```
lst[i],lst[right]=lst[right],lst[i]
return i

def order_short(lst):
"""
Para sa satida del resultado utilizamos un metodo funcional
"""

if len(lst) > 1:
n=len(lst)
mid = n//2 # Funcion piso
if n%2==0:
med = QuickSelect(lst,mid-1)
else:
med = QuickSelect(lst,mid)
if n > 4:
k=mid//2
lst= lst[mid+1:n-k] + lst[0:k] + lst[mid:mid+1] + lst[k:mid] +lst[n-k:]
return lst

lst_1=[1,2,3,4,5]
print("lista",lst_1)
lst_1=order_short(lst_1)
print(lst_1)
lst_2=[1,2,3,4,5,6]
print("lista",lst_2)
lst_2=order_short(lst_2)
print(lst_2)
```

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.

```
def contiene_suma(A,n):
#Creamos un conjunto para almacenar los elementos vistos
vistos=set()

#iteramos sobre la lista A, y visitamos cada elemento de la lista
for num in A:
#calculamos el complemento esesario para que la suma sea n
complemento = n - num
#Verificamos si el complemento ya esta en la lista de elementos visitados
if complemento in vistos:
#Si seencuentra un par que dado su suma da n
return True
vistos.add(num)
#Si no se a ancontrado entonces retorna falso
return False
```

```
A=[1,2,3,4,5]
n=9
print("Lista: ",A)
print("¿La lista A contiene un par de elementos que suman",n,"? ", contiene_suma(A,n) )
#la compejiad del algoritmo seria O(n) ya que solo recoremos la lista una vez y realiza las operaciones en tiempo constante en cada iteracion.
```

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.

Radix Sort: Es un método de ordenamiento que procesa los dígitos de los números en forma individual empezando por el digito menos significativo.

Lo primero que se tiene que saber para este ordenamiento es el número de veces que se repetirá el algoritmo y este es igual al número de dígitos de el numero con más longitud de la lista

El algoritmo de ordenamiento Radix Sort es un algoritmo no comparativo que funciona ordenando los elementos basándose en los dígitos individuales de sus representaciones numéricas. Funciona bien para ordenar enteros o números con una cantidad fija de dígitos. El Radix Sort puede ser eficiente cuando se trabaja con números que tienen una cantidad fija de dígitos, como números enteros con una cantidad constante de dígitos o cadenas de caracteres con longitudes constantes.

Funcionamiento:

Selecciona la cantidad máxima de dígitos en los números a ordenar.

Itera a través de cada dígito, comenzando desde el dígito menos significativo hasta el más significativo.

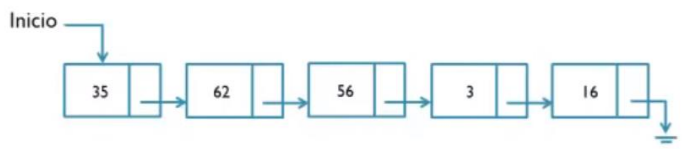
En cada iteración, se distribuyen los elementos en un conjunto de "baldes" o contenedores según el valor del dígito actual.

Luego, se reúnen los elementos de los baldes en orden, formando una nueva lista ordenada.

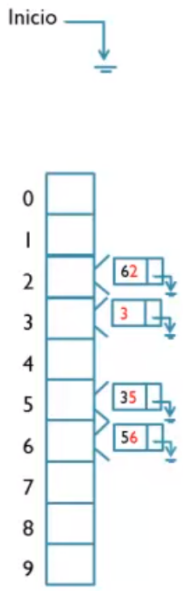
Este proceso se repite para cada dígito, hasta que se hayan considerado todos los dígitos.

Ejemplo:

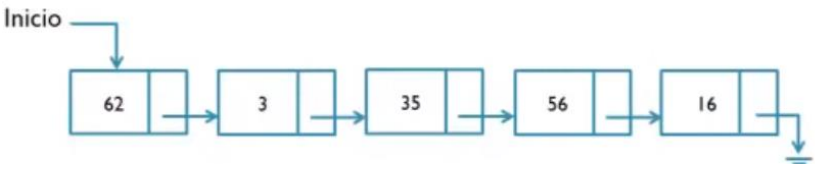
Supongamos que tenemos el siguiente conjunto de números enteros a ordenar:



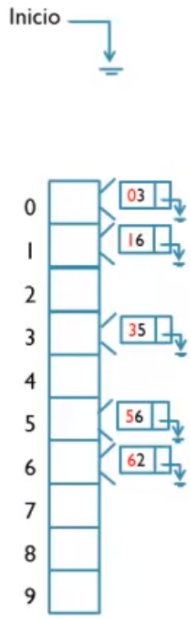
Comenzamos ordenando los números según el dígito menos significativo (unidades). Los números se distribuyen en 10 baldes, uno para cada posible dígito (0-9).



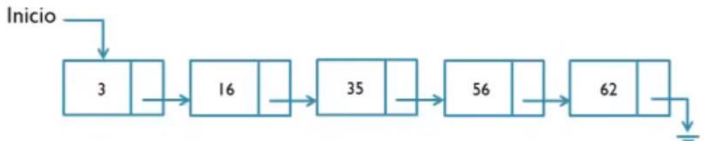
Lista parcialmente ordenada:



Luego, ordenamos los números según el segundo dígito menos significativo (decenas). Los números se redistribuyen en los baldes correspondientes si no se encuentra un número se sustituye con un 0.



Lista parcialmente ordenada:



Finalmente, ordenamos los números según el dígito más significativo (centenas). Como todos los números tienen el mismo dígito más significativo (cero), permanecen en el mismo orden.

Orden de Radix Sort:

El orden de Radix Sort es O(nk), donde n es el número de elementos y k es la cantidad de dígitos máxima en los números a ordenar.

Casos:

Mejor caso: El mejor caso ocurre cuando todos los números tienen la misma cantidad de dígitos y están uniformemente distribuidos en los baldes en cada iteración. En este caso, el rendimiento es óptimo y el algoritmo ejecuta en tiempo lineal, es decir, O(n).

Caso promedio: El caso promedio es similar al mejor caso, pero puede haber algunas variaciones en la distribución de los números en los baldes. Aun así, el rendimiento sigue siendo bueno y el orden es O(nk).

Peor caso: El peor caso ocurre cuando los números tienen dígitos desbalanceados y hay una distribución no uniforme en los baldes en cada iteración. En este caso, el rendimiento del algoritmo se degrada, pero sigue siendo O(nk).

Ejercicio_7:

Algoritmos y Estructuras de Datos II:

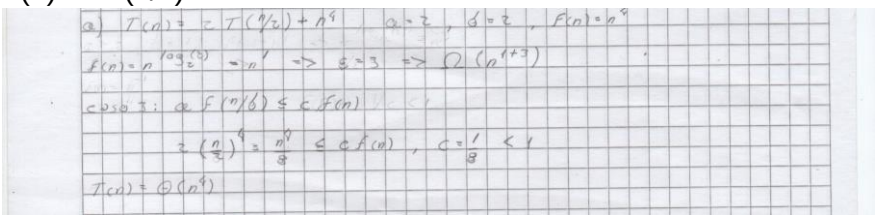
Ejercitación: Análisis de Complejidad por casos

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

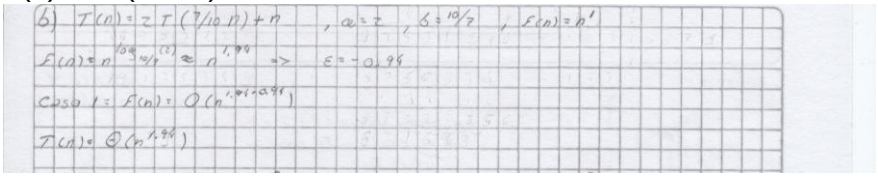
T(n) es constante para n <= 2. Resolver 3 de ellas con el método maestro completo: T(n) = a

T(n/b) + f(n) y otros 3 con el método maestro simplificado: T(n) = a T(n/b) + n^c

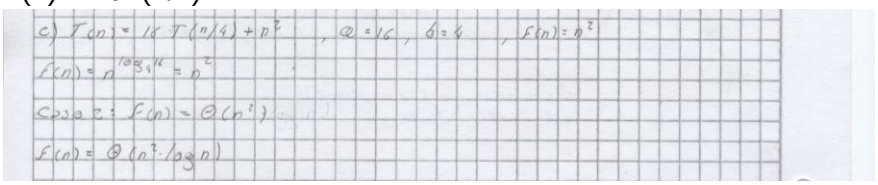
a. T(n) = 2T(n/2) + n^4



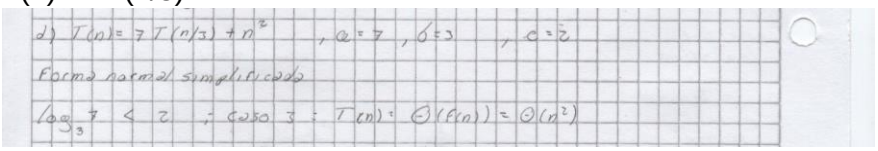
b. T(n) = 2T(7n/10) + n



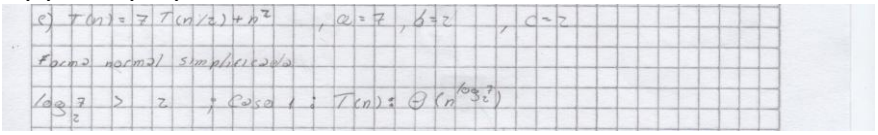
c. $T(n) = 16T(n/4) + n^2$



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



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



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