

Francisco Alba

Ejercicio 1:

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

$T(n)$ es $O(f(n))$ si existen constantes c y n_0 positivas, tales que $T(n) \leq cf(n)$ para todo $n \geq n_0$. Considerando $T(n) = 6n^3$, podemos observar que incluso para alguna constante c muy grande, existe un $n = n_0$ a partir del cual $6n^3 \geq cn^2$ para todo n . Luego $6n^3 \neq O(n^2)$.

Ejercicio 2:

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

Tomando como pivote siempre el elemento del medio del arreglo, la siguiente es una posibilidad:

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

Al estar el arreglo ordenado en forma creciente, en cada división sucesiva la parte izquierda y la derecha se mantienen equilibradas y entonces el algoritmo logra ejecutarse en $O(n \log n)$

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?

Cuando todos los elementos del arreglo tienen el mismo valor, los tiempos de ejecución son los siguientes:

QuickSort: $O(n^2)$

InsertionSort: $O(n)$

MergeSort: $(n \log n)$

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
---	---	---	---	---	---	---	---	----	---

```
# Primero, se ubica la posición el elemento del medio de la lista.
# Dicha posición indica también el número de elementos que hay a la izquierda del mismo.
# Si menos de la mitad de los elementos del lado izquierdo al del medio son menores que él,
# se buscan elementos menores al del medio en el lado derecho y se intercambian con alguno mayor
# que esté en el lado izquierdo.
# Si más de la mitad de los números del lado izquierdo son menores que el del medio,
# se sigue la estrategia opuesta, es decir, se mueven elementos menores al del medio desde el lado
# izquierdo al derecho, y se intercambian por alguno mayor al del medio que esté ubicado
# en el lado derecho.

def halfMinorOrder(L):
    # Obtenemos la posición y valor del elemento del medio
    midElemPos = int(length(L)/2)
    midElemVal = access(L, midElemPos)
    # Contamos el número de elementos menores que el central ubicados antes que él
    numOfMinorElems = 0
    for i in range(0, midElemPos):
        if access(L, i) < midElemVal:
            numOfMinorElems += 1
    print(numOfMinorElems)
    if numOfMinorElems != int(midElemPos/2):
        if numOfMinorElems < int(midElemPos/2):
            # Menos de la mitad de los elementos anteriores al del medio, son menores que el del medio
            minorElemsToAdd = int(midElemPos/2) - numOfMinorElems
            minorElemsToTheRight = 0
            for i in range(midElemPos+1, length(L)):
                if access(L, i) < midElemVal:
                    minorElemsToTheRight += 1
            if minorElemsToTheRight >= minorElemsToAdd:
                # Tomo elementos menores al del medio de la derecha y los muevo a la izquierda
                i = midElemPos+1
                while minorElemsToAdd > 0:
                    if access(L, i) < midElemVal:
                        nodeVal = access(L, i)
                        insert(L, nodeVal, midElemPos)
                        deleteAferPos(L, nodeVal, i+1)
                    # Tomo elementos mayores al del medio de la izquierda, y los muevo a la derecha
                    j = 0
                    while access(L, j) <= midElemVal:
                        j += 1
                    nodeVal = access(L, j)
                    insert(L, nodeVal, i)
                    delete(L, nodeVal)
                    minorElemsToAdd -= 1
                    i += 1
            else:
                print("No hay elementos menores del lado derecho para compensar")
                return
```

```
else:
    # Más de la mitad de los elementos anteriores al del medio, son menores que el del medio
    minorElemsToQuit = numOfMinorElems - int(midElemPos/2)
    majorElemsToTheRight = 0
    for i in range(midElemPos+1, length(L)):
        if access(L, i) > midElemVal:
            majorElemsToTheRight += 1
    if majorElemsToTheRight >= minorElemsToQuit:
        i = 0
        while minorElemsToQuit > 0:
            if access(L, i) < midElemVal:
                insert(L, access(L, i), midElemPos+1)
                delete(L, access(L, i))
                # Tomo elementos mayores al del medio de la derecha, y los muevo a la izquierda
                j = midElemPos+1
                while access(L, j) <= midElemVal:
                    j += 1
                nodeVal = access(L, j)
                insert(L, nodeVal, i)
                deleteAfterPos(L, nodeVal, j)
                minorElemsToQuit -= 1
            i += 1
        else:
            print("No hay elementos mayores del lado derecho para compensar")
            return
    else:
        print("La lista ya está ordenada!")
        return
```

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.

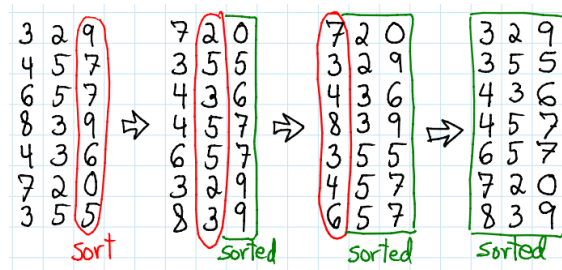
```
def contieneSuma(A, n):
    for i in range(0, length(A)):
        for j in range(0, length(A)):
            if (access(A, i) + access(A, j) == n) and i != j:
                return True
    return False
```

```
# Por cada elemento de la lista de tamaño k, se realizan k sumas para
# comprobar si su resultado es n. Por lo tanto, el costo computacional
# se puede estimar en  $O(k^2)$ 
```

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.

El algoritmo de ordenamiento RadixSort funciona habitualmente ordenando elementos conformados por dígitos, sin compararlos. Para lograr esto, ordena todos los elementos por etapas considerando en primer lugar la cifra menos significativa, ordenando, y repitiendo así cifra por cifra hasta la mayor cifra significativa de todos los elementos.



En la imagen anterior, puede verse ilustrado el funcionamiento del algoritmo. Para un conjunto de 7 números de 3 cifras cada uno, se empieza ordenando la columna ubicada más hacia la derecha, que representa la menor cifra significativa de la base decimal, y se repite el proceso para la columna del medio y la columna izquierda. Para ordenar cada columna, sin embargo, RadixSort se vale de un algoritmo de ordenación intermedio, que debe ser estable.

Generalmente, el algoritmo elegido en este caso es CountingSort. El hecho de que sea estable, significa que en cada paso al ordenar, preserva el orden relativo de elementos de igual valor, es decir, el orden en el que aparecen en la lista.

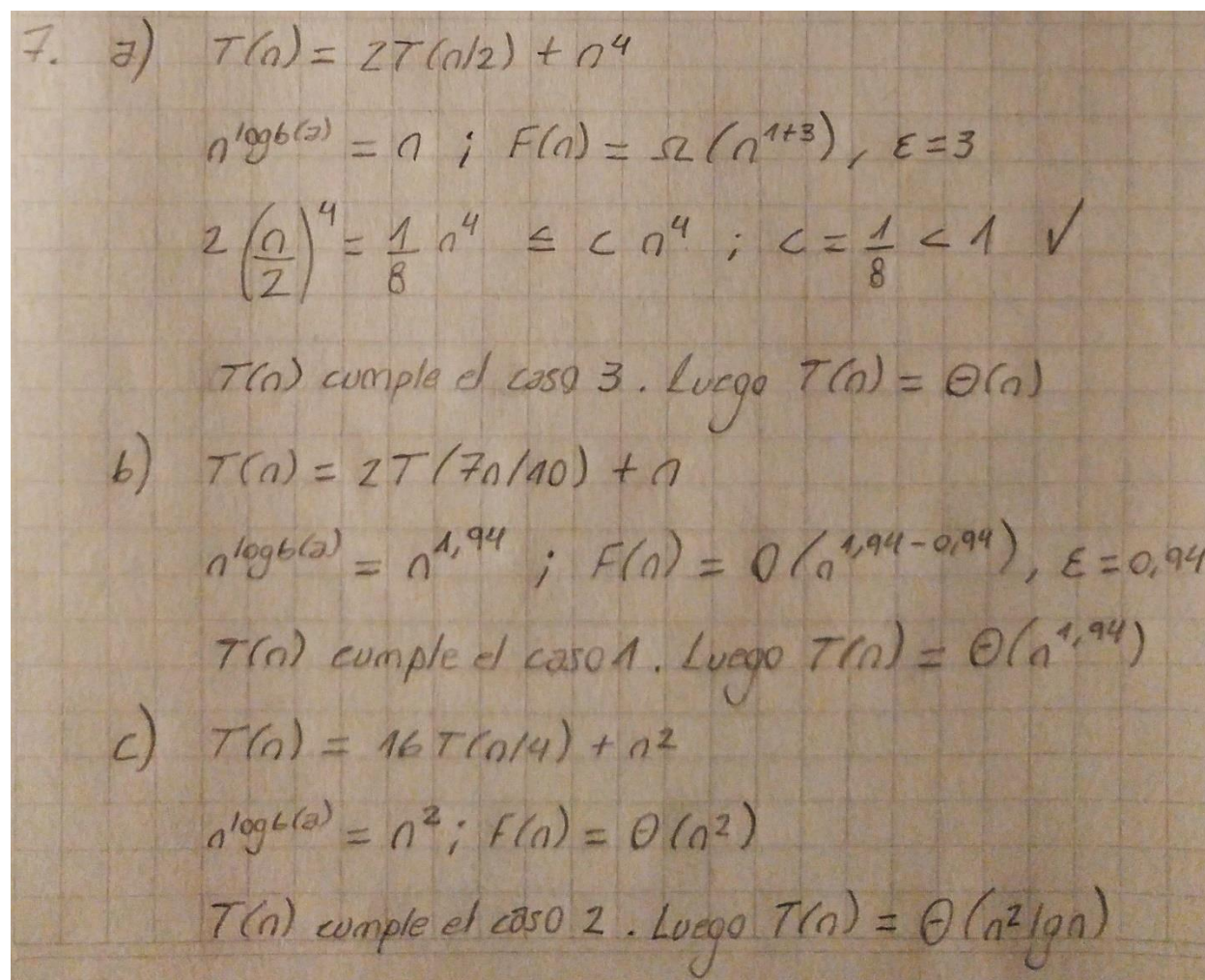
Si los números de la lista a ordenar no tienen todos la misma cantidad de cifras, el algoritmo comienza considerando la cantidad de cifras del mayor elemento de la lista, ya que este determina el número total de pasadas que realizará el algoritmo.

En el caso en que el algoritmo de ordenación intermedio empleado sea CountingSort, la complejidad temporal de RadixSort para los casos mejor, promedio y peor viene dada por $O(d \cdot (n + b))$, donde d es el número de dígitos del mayor elemento, y $O(n + b)$ es la complejidad de CountingSort.

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

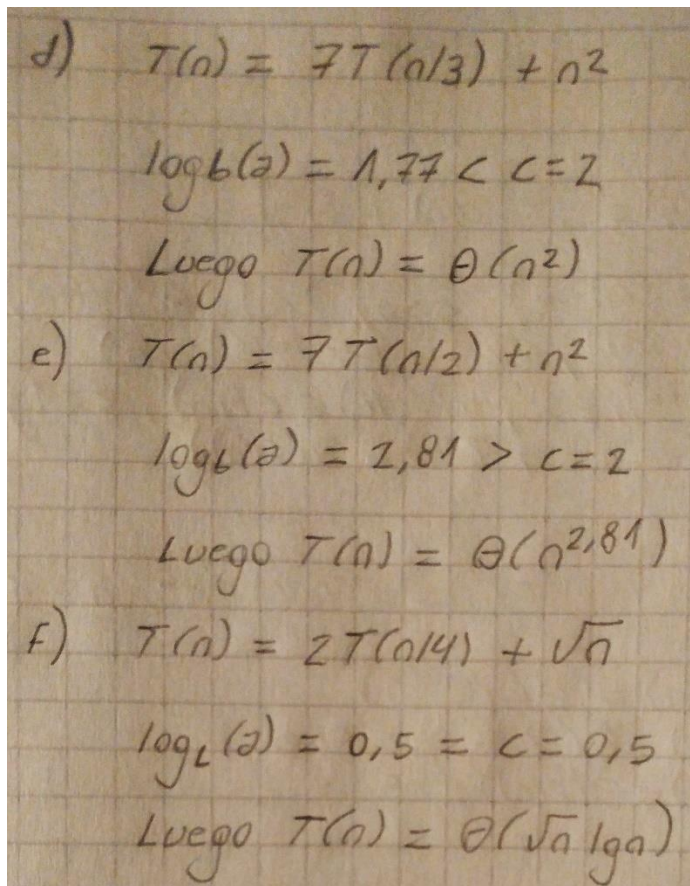
- a. $T(n) = 2T(n/2) + n^4$
- b. $T(n) = 2T(7n/10) + n$
- c. $T(n) = 16T(n/4) + n^2$
- d. $T(n) = 7T(n/3) + n^2$
- e. $T(n) = 7T(n/2) + n^2$
- f. $T(n) = 2T(n/4) + \sqrt{n}$



7. a) $T(n) = 2T(n/2) + n^4$
 $n^{\log_b(a)} = n$; $F(n) = \Omega(n^{1+3})$, $\epsilon = 3$
 $2\left(\frac{n}{2}\right)^4 = \frac{1}{8}n^4 \leq cn^4$; $c = \frac{1}{8} < 1$ ✓
 $T(n)$ cumple el caso 3. Luego $T(n) = \Theta(n^4)$

b) $T(n) = 2T(7n/10) + n$
 $n^{\log_b(a)} = n^{1.94}$; $F(n) = O(n^{1.94-0.94})$, $\epsilon = 0.94$
 $T(n)$ cumple el caso 1. Luego $T(n) = \Theta(n^{1.94})$

c) $T(n) = 16T(n/4) + n^2$
 $n^{\log_b(a)} = n^2$; $F(n) = \Theta(n^2)$
 $T(n)$ cumple el caso 2. Luego $T(n) = \Theta(n^2 \lg n)$



d) $T(n) = 7T(n/3) + n^2$
 $\log_b(a) = 1,77 < c = 2$
Luego $T(n) = \Theta(n^2)$

e) $T(n) = 7T(n/2) + n^2$
 $\log_b(a) = 2,81 > c = 2$
Luego $T(n) = \Theta(n^{2,81})$

f) $T(n) = 2T(n/4) + \sqrt{n}$
 $\log_b(a) = 0,5 = c = 0,5$
Luego $T(n) = \Theta(\sqrt{n} \lg n)$

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.