



**Algoritmia**

Adrián Borges Cano

Marco González Martínez

Laboratorio: Miércoles 15:00 – 17:00

Tema 3: Divide y Vencerás

Tabla de contenido

[PROBLEMA 4: 3](#_Toc98607948)

[PROBLEMA 6: 5](#_Toc98607949)

# PROBLEMA 4:

**En una plataforma digital van a analizar los gustos de sus clientes para analizar las similitudes con otros clientes y hacerles recomendaciones personalizadas. Para ello han pedido a los usuarios de dicha plataforma que realicen un ranking de un conjunto de peliculas que ya han visto. Para clientes hay que estudiar cómo de similares son sus gustos Una de las formas de calcular la similitud entre dos rankings es contar el número de inversiones que existen. Dos películas i y j están invertidas en las preferencias de A y B si el usuario A prefiere la serie i antes que la j (aparece antes en su ranking) mientras que el usuario B prefiere la serie j antes que el i. Cuantas menos inversiones existan entre dos rankings, más similares son las preferencias de esos usuarios. Para facilitar este proceso, podemos suponer que el ranking de uno de los clientes siempre es 1, 2, …, n (cualquier par de rankings se puede transformar para que se cumpla esto). De esta forma, podemos trabajar sólo con el otro ranking, y existe una inversión cada vez que un número más pequeño está situado a la derecha de un número más grande. Diseñar un algoritmo, lo más eficiente posible, que permita calcular el número de inversiones entre dos clientes.**

Descripción:

Para poder solucionar el problema propuesto, observamos que las inversiones mencionadas en el enunciado, equivalen a las rotaciones que se producen en la ordenación por el método de burbuja. Este método de ordenación es, implícitamente, un algoritmo divide y vencerás, puesto que, al hacerlo recursivamente, obtenemos subproblemas más simplificados hasta dar con el caso base, el cual es la lista con un único elemento. Por tanto, hemos aplicado el método de burbuja, cuyos parámetros de entrada son la lista del ranking de los clientes correspondientes, y un entero, el cual será utilizado como contador de inversiones.

Casos de prueba:

**Primer caso.**

**Se le dan diez películas a elegir y el orden de los gustos de los dos clientes es el siguiente:**

**Cliente1 = [1, 2, 3, 4, 5, 6, 7, 8]**

**Cliente2 = [6, 4, 3, 1, 8, 7, 2, 5]**

**Segundo caso.**

**Se le dan veinte películas a elegir y el orden de los gustos de los dos clientes es el siguiente:**

**Cliente1 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]**

**Cliente2 = [7, 13, 2, 19, 10, 4, 9, 20, 1, 5, 15, 17, 6, 18, 3, 14, 16, 8, 12, 11]**

Cliente1 = [1, 2, 3, 4, 5, 6, 7, 8]

Cliente2 = [6, 4, 3, 1, 8, 7, 2, 5]

Cliente3 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]

Cliente4 = [7, 13, 2, 19, 10, 4, 9, 20, 1, 5, 15, 17, 6, 18, 3, 14, 16, 8, 12, 11]

print(burbuja(Cliente2)[1])

print(burbuja(Cliente4)[1])

Resultado para Cliente1 (Cliente2)🡪 15

Resultado para Cliente2 (Cliente4) 🡪 87

Cálculo de la complejidad:

Realizaremos el cálculo de la complejidad directamente desde el código, para facilitar la comprensión:

def burbuja(c): #Operación: Coste

contador = 0 #Asignación: 1

n = len(c) #Asignación y función len(): n+1

if n <=1: #Condicional y comparación: 1

return c, contador #Retorno: 2

else:

for i in range(n-1): #Bucle y resta: 1+n\*CosteBucle

if c[i]>c[i+1]: #Comparación y suma: 2

contador+=1 #Asignación y suma: 2

c[i],c[i+1] = c[i+1],c[i] #Asignación y sumas: 4

r = burbuja(c[:-1]) #Asignación y llamada recursiva: 1+T(n-1)

return r[0]+[c[-1]],r[1]+contador #Retorno y sumas: 4

#Análisis de la recursividad:

# 1+n+1+1+max(2,1+n\*max(2+2+4,0)+1+T(n-1)+4) = 3+n+max(2,6+n\*max(8,0)+T(n-1)) =

# = 3+n+6+8n+T(n-1) = 9+9n+T(n-1)

# T(n) = 9+9n + 9+9(n-1) + 9+9(n-2) + 9+9(n-3) + ...

#Se puede apreciar que la expresión recursiva representa el sumatorio desde 1 hasta n,

# cuyo coste es de O(n^2).

Código:

# El número de inversiones se puede obtener mediante la ordenación por burbuja, por que al ir ordenando los elementos,

# cada intercambio ocurre si un elemento con menor valor está situado después de uno con mayor valor. La solución

# implementada utiliza el método de ordenación por burbuja calculando el total de intercambios que ocurren.

def burbuja(c):

contador = 0

n = len(c)

if n <=1:

return c, contador

else:

for i in range(n-1):

if c[i]>c[i+1]:

contador+=1

c[i],c[i+1] = c[i+1],c[i]

# Reducción del problema: Mediante la ordenación por burbuja, el último elemento de la lista ya queda ordenado

# por lo que sólo queda ordenar la lista excluyendo el último elemento de la lista

r = burbuja(c[:-1])

return r[0]+[c[-1]],r[1]+contador

# PROBLEMA 6:

**Mr. Scrooge ha cobrado una antigua deuda, recibiendo una bolsa con n monedas de oro. Su olfato de usurero le asegura que una de ellas es falsa, pero lo único que la distingue de las demás es su peso, aunque no sabe si este es mayor o menor que el de las otras. Para descubrir cuál es la falsa, Mr. Scrooge solo dispone de una balanza con dos platillos para comparar el peso de dos conjuntos de monedas. En cada pesada lo único que puede observar es si la balanza queda equilibrada, si pesan más los objetos del platillo de la derecha o si pesan más los de la izquierda. Diseñar un algoritmo Divide y Vencerás para resolver el problema de Mr. Scrooge (encontrar la moneda falsa y decidir si pesa más o menos que las auténticas).**

Descripción:

Para resolver el problema, hemos basado nuestro planteamiento en la recursividad, donde dividimos la lista inicial en 3 partes, las cuales la 1 y la 2, tienen el mismo tamaño, mientras que la 3, contendrá el resto de elementos. En caso de que el tamaño de la lista original sea múltiplo de 3, las 3 partes tendrán la misma longitud; de lo contrario, su tamaño será distinto. Nuestro algoritmo planteado realiza lo siguiente: comprueba si las listas 1 y 2 pesan lo mismo. En caso afirmativo, la moneda falsa se encuentra en la 3ª parte; si no, entonces está o en la parte 1 o en la 2. Como es recurviso, este procedimiento se divide en numerosas ocasiones hasta llegar al caso trivial. Si la moneda es más pesada que el resto de elementos, la función devuelve ‘True’. De lo contrario, devuelve ‘False’.

Tenemos un caso base, que ocurre cuando la lista tiene tamaño 3, y después hacemos distinciones si la lista tiene longitud 4, 5 o 6. Para estos últimos, lo único que hacemos es dividir las partes de forma que queden 2 listas de 3 elementos, para que puedan entrar al caso elemental.

A la hora de hablar sobre el caso en el que la lista tiene 3 elementos, hacemos un procedimiento similar al del comentado al principio: comparamos el primer elemento con el segundo, si pesan lo mismo, el elemento falso es el 3º, y lo comparamos con el primero para ver si su peso es mayor o menos. En caso de que los dos primeros elementos sean distintos, concretamente, el primero menor que el segundo, se compara si el primero es igual que el tercero; en caso afirmativo, el segundo elemento, el cual es el falso, es de mayor peso; de lo contrario, entonces el elemento falso es el primero, y tendría peso menor.

Adicionalmente, hemos incluído en nuestro código el caso en el que no haya ningún elemento distinto. Por lo tanto, si después nunca cumple todas estas condiciones, el resultado que devolverá será ‘None’.

Casos de prueba:

**Primer caso**

**27 monedas, cuyos pesos está representados por un vector:**

**-Moneda falsa: Peso [5]=2**

**-Restos de monedas: Peso[i]=1; para i distinto de 5**

**Segundo caso**

**38 monedas, cuyos pesos está representados por un vector:**

**-Moneda falsa: Peso [25]=1**

**-Resto de monedas: Peso[i]=2; para i distinto de 25**

caso\_1 = []

for i in range(27):

if i ==5:

caso\_1.append(2)

else:

caso\_1.append(1)

caso\_2 = []

for i in range(38):

if i ==25:

caso\_2.append(1)

else:

caso\_2.append(2)

print(moneda\_falsa(caso\_1))

print(moneda\_falsa(caso\_2))

Resultado para Caso\_1🡪 True

Resultado para Caso\_2 🡪 False

Cálculo de la complejidad:

Realizaremos el cálculo de la complejidad directamente desde el código, para facilitar la comprensión:

n = len(lista)

if (n==6):

return moneda\_falsa(lista[0:3]) and moneda\_falsa(lista[3:6])

elif (n==5):

return moneda\_falsa(lista[0:3]) and moneda\_falsa(lista[2:5])

elif (n==4):

return moneda\_falsa(lista[0:3]) and moneda\_falsa(lista[1:4])

elif (n==3):

if lista[0]<lista[1]:

if lista[0]==lista[2]:

return True

else:

return False

elif lista[0]>lista[1]:

if lista[0]==lista[2]:

return False

else:

return True

else:

if lista[0]<lista[2]:

return True

elif lista[0]>lista[2]:

return False

else:

return None

#Caso general

elif (n>6):

# División del problema en 3 partes (Debido a que las dos partes a introducir en la báscula deben tener el mismo número de elementos)

k = n//3 # División entera y Asignación: 2

parte1 = lista[0:k] # Asignación: 1

parte2 = lista[k:2\*k] # Multiplicación y Asignación: 2

parte3 = lista[2\*k:] # Multiplicación y Asignación: 2

a=sum(parte1) # Asignación y Función sum: 1+n

b=sum(parte2) # Asignación y Función sum: 1+n

# Si las dos primeras parte estaban en equilibrio, la moneda estará en la tercera parte

if a==b: # Comparación: 1

return moneda\_falsa(parte3) # Función recursiva: T(n//3)

# Si por el contrario, estaban desequilibradas, estará en la primera o segunda parte

else:

return moneda\_falsa(parte1+parte2) # Función recursiva: T(2\*n//3)

#Análisis de la Recursión:

# Según el método maestro

# T(n) = k \* T(n/b) + f(n) k = 2, b = 3/2, f(n) = 2n + 9 k>b^p

# T(n) = 2 \* T(2n/3) + 2n + 9 => T(n) = O(n^(2log\_(3/2)))

Código:

def moneda\_falsa(lista):

"""

list(num)-->bool

True si la moneda falsa es de mayor peso, False si menor, None si todos los elementos son iguales

OBJ: Determinar si la moneda falsa de un conjunto es mayor o menor

PRE: len(lista)>=3

"""

# Casos base (Cuándo ya no se puede dividir más se calculan los problemas pequeños por una aproximación lineal)

n = len(lista)

if (n==6):

return moneda\_falsa(lista[0:3]) and moneda\_falsa(lista[3:6])

elif (n==5):

return moneda\_falsa(lista[0:3]) and moneda\_falsa(lista[2:5])

elif (n==4):

return moneda\_falsa(lista[0:3]) and moneda\_falsa(lista[1:4])

elif (n==3):

if lista[0]<lista[1]:

if lista[0]==lista[2]:

return True

else:

return False

elif lista[0]>lista[1]:

if lista[0]==lista[2]:

return False

else:

return True

else:

if lista[0]<lista[2]:

return True

elif lista[0]>lista[2]:

return False

else:

return None

elif (n>6):

# División del problema en 3 partes (Debido a que las dos partes a introducir en la báscula deben tener el mismo número de elementos)

k = n//3

parte1 = lista[0:k]

parte2 = lista[k:2\*k]

parte3 = lista[2\*k:]

a=sum(parte1)

b=sum(parte2)

# Si las dos primeras parte estaban en equilibrio, la moneda estará en la tercera parte

if a==b:

return moneda\_falsa(parte3)

# Si por el contrario, estaban desequilibradas, estará en la primera o segunda parte

else:

return moneda\_falsa(parte1+parte2)