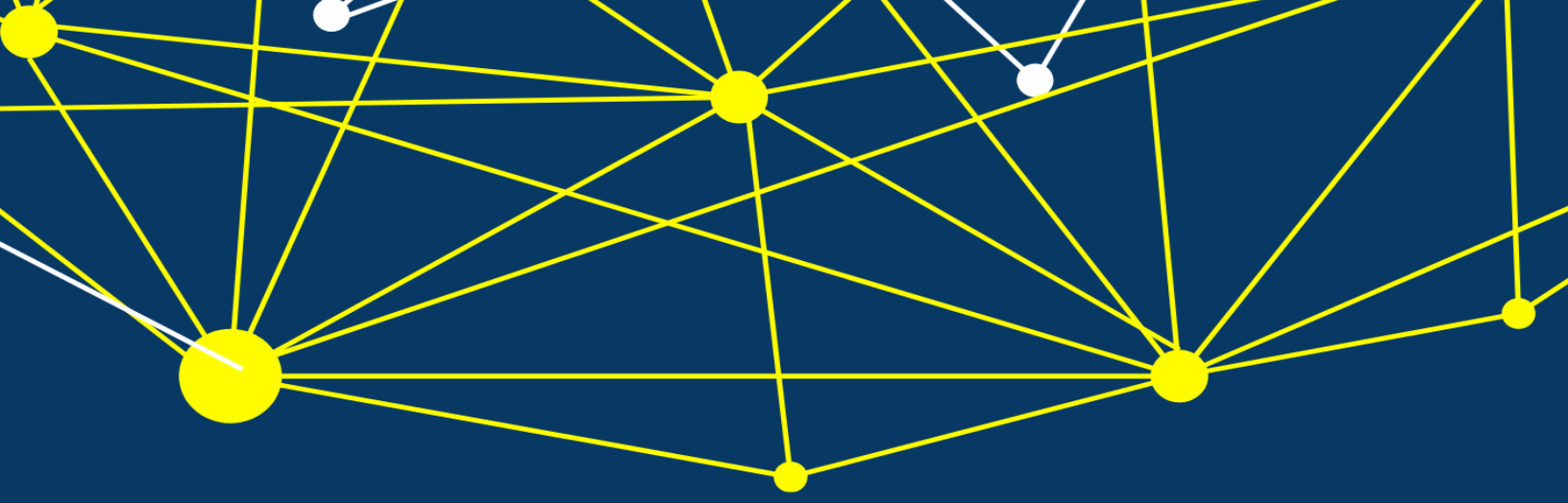

Adrián Borges Cano

Marco González Martínez

Laboratorio: Miércoles 15:00 – 17:00

Algoritmia

Tema 3: Divide y Vencerás

Tabla de contenido

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 mergesort visto en teoría. Por ello, hemos decidido implementar directamente este algoritmo de ordenación, introduciendo leves cambios para adaptarlo al ejercicio propuesto. Este método de ordenación es un algoritmo divide y vencerás, puesto que este divide la lista principal en varias sublistas de menor tamaño, y estas a su vez en otras sublistas, hasta dar con el caso base, el cual ocurre cuando hay solo un elemento de la lista. Consecuentemente, hemos decidido incluir un parámetro de entrada, para que nuestro algoritmo 'inversiones' haga lo pedido, el cual es un entero, y es usado como contador de inversiones del método de ordenación; cada vez que se intercambian un par de datos, este contador se ve incrementado por uno, puesto que es el objetivo del ejercicio.

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(mergesort(Cliente2)[1])
print(mergesort(Cliente4)[1])
```

Resultado para Cliente1 (Cliente2)→ 15

Resultado para Cliente2 (Cliente4) → 87

Para analizar el paso a paso de los casos de prueba, explicaremos solamente uno de ellos (Cliente1 y Cliente2), ya que el segundo tendrá el mismo procedimiento aplicado:

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

[6, 4, 3, 1] [8, 7, 2, 5] Se dividen progresivamente las listas hasta que estas estén formadas por un único elemento.

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

[6] [4] [3] [1] [8] [7] [2] [5] Una vez damos con el caso base, agruparemos los elementos de la misma forma que se han separado (juntándolos por grupos de elementos 2 a 2). Se estudian los elementos de forma secuencial, siguiendo el orden propio de cada lista, y serán agrupados de forma que el menor siempre irá a la izquierda del mayor. Cada vez que esto ocurra, se sumará 1 al contador general. En el procedimiento paso a paso se indicará con un '(+1)' cada vez que esto ocurra. Si una lista tiene más de un elemento, se irá comprobando uno a uno si es mayor o menor que la lista resultante.

[6] [4] Como el menor está a la derecha del mayor, intercambiamos el orden para que se cumpla lo requerido → [4] [6] Como ahora están colocados según la condición de que deben ir de menor a mayor, serán agrupados → [4, 6] (+1)

[3] [1] → Pasa lo mismo que en el caso anterior → [1] [3] → [1, 3] (+1)

[8] [7] → Ídem → [7] [8] → [7, 8] (+1)

[2] [5] → En este caso, los elementos ya vienen ordenados por defecto como se pide, de forma que no se intercambian, sino que solo se agrupan, y por ello, no se debe incrementar en uno el contador → [2, 5]

Al realizar esta iteración, obtenemos:

[4, 6] [1, 3] [7, 8] [2, 5] Combinamos los elementos 2 a 2 como se ha dicho, siguiendo la propiedad de que definimos previamente. Comparamos los elementos uno a uno para formar las nuevas listas.

[4, 6] [1, 3] → Como 1<4, se pone el 1 primero → [1] [4, 6] [3] (+1) → Como 3<4, se pone primero en la lista resultante → [1, 3] [4,6] [-] (+1) → Como no tenemos elementos en una de las listas (en este caso la segunda), la lista restante se une con la resultante → [1, 3, 4, 6] (+1)

[7, 8] [2, 5] → Realizamos el mismo procedimiento que en el caso anterior →

[2] [7, 8] [5] (+1) → [2, 5] [7, 8] [-] (+1) → [2, 5, 7, 8]

En esta iteración, obtenemos las listas:

[1, 3, 4, 6] [2, 5, 7, 8] Una vez más, realizamos lo mismo que en la iteración anterior, aunque en este caso tenga más elementos.

[1, 3, 4, 6] [2, 5, 7, 8] → [1] [3, 4, 6] [2, 5, 7, 8] (+1) → [1, 2] [3, 4, 6] [5, 7, 8] (+1) → [1, 2, 3] [4, 6] [5, 7, 8] (+1) → [1, 2, 3, 4] [6] [5, 7, 8] (+1) →

[1, 2, 3, 4, 5] [6] [7, 8] (+1) → [1, 2, 3, 4, 5, 6] [-] [7, 8] (+1) → [1, 2, 3, 4, 5, 6, 7, 8] (+1)

Finalmente, el resultado queda:

[1, 2, 3, 4, 5, 6, 7, 8] [15]

Donde 15 es el número de inversiones que se ha ido incrementando.

<u>Cálculo de la complejidad:</u>

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

```
def mergesort(c):
    n = len(c) #n + 1

    # Caso base (Cuándo ya no se puede dividir más se calculan los problemas pequeños por una aproximación lineal)
    if n <=1:
        return c, 0

    # Caso general
    # División del problema, resolvemos el problema primero para la parte izquierda y derecha de la lista y después lo mezclaremos
    a = mergesort(c[:n//2]) # 2 + T(n/2)
    b = mergesort(c[n//2:]) # 2 + T(n/2)
```

```
    l = a[0]
    r = b[0]
    contador = a[1] + b[1]

    res = []
    while len(l)>0 and len(r)>0: # (2n+1) n/2 = max(len(l), len(r))
        if l[0] <= r[0]:
            res.append(l[0])
            l = l[1:]
        else:
            #Si un elemento menor está situado después de uno mayor, hay una inveersión por cada elemento en la lista izquierda.
            contador += len(l) # n · n/2
            res.append(r[0])
            r = r[1:]
    # Restantes (l o r va a ser [])
    res += l + r
    return res, contador

    # Análisis de la recursividad:
    # Según el método maestro
    # T(n) = k * T(n/b) + f(n)              k = 2, b = 2, f(n) = 3/2(n^2), p=2  k<b^p
    # T(n) = 2 · T(n/2) + 3(n^2)/2 => T(n) = O(n^2)
```

Código:

```
# El número de inversiones se puede obtener mediante la ordenación por mergesort, 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 mergesort calculando el total de intercambios que ocurren.

def mergesort(c):
    n = len(c) #n + 1
```

```
    # Caso base (Cuándo ya no se puede dividir más se calculan los problemas pequeños por una aproximación lineal)
    if n <=1:
        return c, 0

    # Caso general
    # División del problema, resolvemos el problema primero para la parte izquierda y derecha de la lista y después lo mezclaremos
    a = mergesort(c[:n//2]) # 2 + T(n/2)
    b = mergesort(c[n//2:]) # 2 + T(n/2)
    l = a[0]
    r = b[0]
    contador = a[1] + b[1]

    res = []
    while len(l)>0 and len(r)>0: # (2n+1) n/2 = max(len(l), len(r))
        if l[0] <= r[0]:
            res.append(l[0])
            l = l[1:]
        else:
            #Si un elemento menor está situado después de uno mayor, hay una inveersión por cada elemento en la lista izquierda.
            contador += len(l) # n · n/2
            res.append(r[0])
            r = r[1:]
    # Restantes (l o r va a ser [])
    res += l + r
    return res, 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 incluido 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

Para analizar el paso a paso de los casos de prueba, explicaremos solamente uno de ellos (caso_1), ya que el segundo tendrá el mismo procedimiento aplicado:

[1,1,1,1,1,2,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]

Dividimos la lista en 3 partes:

[1,1,1,1,1,2,1,1,1] [1,1,1,1,1,1,1,1,1] [1,1,1,1,1,1,1,1,1]

Hacemos la comparación de los dos primeros trozos en la báscula:

10 != 9

Como son desiguales sabemos que la unión de los dos primeros trozos contiene la moneda falsa

[1,1,1,1,1,2,1,1,1,1,1,1,1,1,1,1,1,1]

Dividimos la lista en 3 partes:

[1,1,1,1,1,2] [1,1,1,1,1,1] [1,1,1,1,1,1]

Hacemos la comparación de los dos primeros trozos en la báscula:

7 != 6

Como son desiguales sabemos que la unión de los dos primeros trozos contiene la moneda falsa

[1,1,1,1,1,2,1,1,1,1,1,1]

Dividimos la lista en 3 partes:

[1,1,1,1] [1,2,1,1] [1,1,1,1]

Hacemos la comparación de los dos primeros trozos en la báscula:

4 != 5

Como son desiguales sabemos que la unión de los dos primeros trozos contiene la moneda falsa

[1,1,1,1,1,2,1,1]

Dividimos la lista en 3 partes

[1,1] [1,1] [2,1,1,1]

Hacemos la comparación de los dos primeros trozos en la báscula:

2 == 2

Como son iguales sabemos que la moneda falsa está en el tercer trozo:

[2,1,1,1]

Como ahora la lista ya es muy pequeña, se trata de un caso base en el que se resuelve el problema por una aproximación lineal, tendremos una lista de 6 ó menos elementos, así que buscaremos la moneda falsa buscando entre las 3 primeras monedas, y las 3 últimas (una moneda puede aparecer en ambos grupos). Después teniendo 3 monedas, las compararíamos secuencialmente:

lista[0:3] lista[1:4]

[2,1,1]

Hacemos la comparación en la báscula de los elementos individualmente:

lista[0]=2>1=lista[1] lista[0]=2!=1=lista[2] El elemento 0 es distinto y mayor, por lo que la moneda falsa tiene mayor peso (Según nuestro retorno especificado se devuelve un True)

lista = [1,1,1]

Hacemos la comparación en la báscula de los elementos individualmente:

lista[0]=1==1=lista[1] lista[0]=1==1=lista[2] Los 3 elementos tenían el mismo peso, no se ha encontrado la moneda falsa en este trozo (Según nuestro retorno especificado se devuelve None)

A partir de ahora, se deshace la llamada recursiva pasando el valor de retorno a la llamada anterior, aunque primero se hace una operación para terminar con la aproximación lineal

True and None = True

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 and n>3):
        return moneda_falsa(lista[0:3]) and moneda_falsa(lista[-3:])
    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)
```