ANÁLISIS DEL RETO

María Alejandra Pinzón - 202213956 – ma.pinzonr1

María Alejandra Londoño - 202220983 - m.londonoi

Gabriela Escobar - 2011307663 - *g.escobar23*

Requerimiento 1

Descripción

```
def req_1(data_structs, anio, cse):
    """
    Función que soluciona el requerimiento 3
    """
    lista_del_anio= ""
    if mp.contains(data_structs["map_anio"], anio):
        lista_del_anio= mp.get(data_structs["map_anio"], anio)
    impuestos= lista_del_anio['value']['impuestos']
    lista_cse= lt.newList("ARRAY_LIST")
    for x in lt.iterator(impuestos):
        if cse== x['Código sector económico']:
            lt.addLast(lista_cse, x)
    mayor_s_a_p= -1
    mayor= ""
    for c in lt.iterator(lista_cse):
        if int(c['Total saldo a pagar'])> mayor_s_a_p:
            mayor_s_a_p= int(c['Total saldo a pagar'])
            mayor= c
    dic_final= {'Código actividad económica': [mayor['Código actividad económica']], 'Nombre actividad económica':[mayor['Nombre actividad económica']], 'Código subsector económico':[mayor['Código subsector económico']],
                'Nombre subsector económico':[mayor['Nombre subsector económico']],'Total ingresos netos':[mayor['Total ingresos netos']], 'Total costos y gastos': [mayor['Total costos y gastos']], 'Total saldo a pagar': [mayor['Total saldo a pagar']],
                'Total saldo a favor': [mayor['Total saldo a favor']] }
    return dic_final
```

Este requerimiento se en carga de retornar la actividad económica que tuvo el mayor saldo total de impuestos a pagar (Total saldo a pagar) para un sector y un año específico. Encuentra el año indicado, crea una lista de datos correspondientes al sector económico y retorna un diccionario con los datos de la actividad con el mayor saldo total de impuestos a pagar.

Entrada	Estructuras de datos del modelo, año, sector económico.
Salidas	Diccionario actividad con el mayor saldo total de impuestos a pagar.
Implementado (Sí/No)	Si, Maria Alejandra Londoño.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Paso 1: Encontrar en datastructs el año indicado	O(N)
Paso 2: Crear lista del sector económico.	O(N)
Paso 3: Hallar el mayor salto total	O(N)
TOTAL	***O(N)***

Pruebas Realizadas

Las pruebas realizadas fueron realizadas en una maquina con las siguientes especificaciones. Los datos de entrada fueron 2016, 1.

Procesadores	**11th Gen Intel(R) Core (TM) i7-11370H @ 3.30GHz 3.30 GHz**

Memoria RAM	16 GB
Sistema Operativo	Windows 10

Entrada	**Tiempo (ms)**
small	0.12690000049769878
5 pct	0.13740000128746033
10 pct	0.15359999984502792
20 pct	0.16530000045895576
30 pct	0.17539999820291996
50 pct	0.2239999994635582
80 pct	0.2615000009536743
large	0.32099999859929085

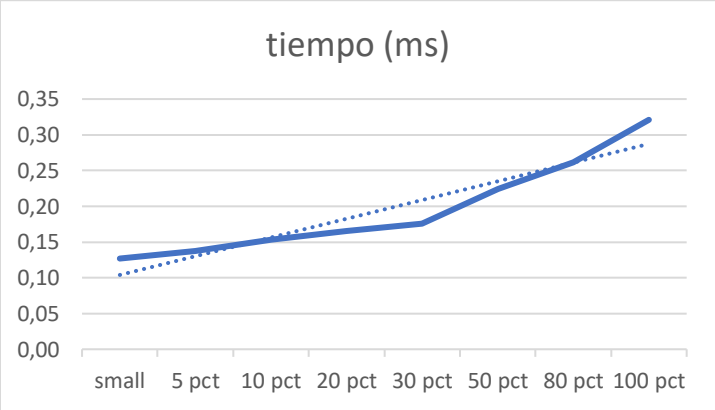
Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

Muestra	**Salida**	**Tiempo (ms)**
small	Dato1	0.12690000049769878
5 pct	Dato2	0.13740000128746033
10 pct	Dato3	0.15359999984502792
20 pct	Dato4	0.16530000045895576
30 pct	Dato5	0.17539999820291996
50 pct	Dato6	0.2239999994635582
80 pct	Dato7	0.2615000009536743
large	Dato8	0.32099999859929085

Graficas

Las gráficas con la representación de las pruebas realizadas.



Análisis

Se puede ver que el requerimiento, como era esperado, si cumple con un orden lineal de O(n). Esto se puede bien evidenciar en la grafica y en los datos de tiempo en las tablas.

Requerimiento 2

Descripción

```python
def req_2(data_structs, year, codigo):
    """
    Función que soluciona el requerimiento 2
    Obtener la actividad económica con mayor saldo a pagar para un sector económico y un año específico.
    """
    # TODO: Realizar el requerimiento 2

    year_entry = mp.get(data_structs['map_anio'], year)
    year_datos = me.getValue(year_entry)
    lista_actecon = year_datos['impuestos']
    aja = lt.newList('ARRAY_LIST')

    for act_econ in lt.iterator(lista_actecon):
        lt.addLast(aja, act_econ)

    lst_act_eco_filt = lt.newList('ARRAY_LIST')
    for act_eco_filt in lt.iterator(aja):
        if act_eco_filt['Código sector económico'] == codigo:
            lt.addLast(lst_act_eco_filt, act_eco_filt)

    merg.sort(lst_act_eco_filt, cmp_saf)
    final_enarray = lt.getElement(lst_act_eco_filt, lt.size(lst_act_eco_filt))
    headers = ['Código actividad económica', 'Nombre actividad económica', 'Nombre subsector económico',
               "Total ingresos netos", "Costos y gastos nómina", "Total saldo a pagar","Total saldo a favor"]
    final = lt.newList('ARRAY_LIST')
    for elemento in headers:
        lt.addLast(final,final_enarray[elemento])


    return final, headers

def cmp_saf (data1, data2):
    return int(data1['Código sector económico']) > int(data2['Código sector económico'])
```

Este requerimiento se encarga de retornar la actividad económica que tuvo el mayor saldo total de impuestos a favor (Total saldo a favor) para un sector y un año específico.

Entrada	El map principal, año elegido por el usuario y el código del sector económico elegido por el usuario.
Salidas	La actividad económica con mayor saldo a favor, junto a las características pedidas en el enunciado.
Implementado (Sí/No)	Si, Gabriela Escobar

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**

`year_entry = mp.get(data_structs['map_anio'], year)`
`year_datos = me.getValue(year_entry)`
`lista_actecon = year_datos['impuestos']`
	O(N)

`for act_econ in lt.iterator(lista_actecon):`	O(N)

`lt.addLast(aja, act_econ)`	

`for act_eco_filt in lt.iterator(aja):`
`  if act_eco_filt['Código sector económico'] == codigo:`
`  lt.addLast(lst_act_eco_filt, act_eco_filt)`
	O(N)

`merg.sort(lst_act_eco_filt, cmp_saf)`	O(NlogN)
`final_enarray = lt.getElement(lst_act_eco_filt, lt.size(lst_act_eco_filt))`	O (N)
TOTAL	***O(NlogN)***

Pruebas Realizadas

Las pruebas realizadas fueron realizadas en una maquina con las siguientes especificaciones. Los datos de entrada fueron para el año 2021, y el código de sector económico 3. El programa se probó utilizando el esquema de colisión chaining, con el factor de carga de 8.

Procesador	**11th Gen Intel(R) Core(TM) i7-1195G7 @ 2.90GHz 2.92 GHz**
Memoria RAM	32GB
Sistema Operativo	Windows 11 Pro- 64 bits

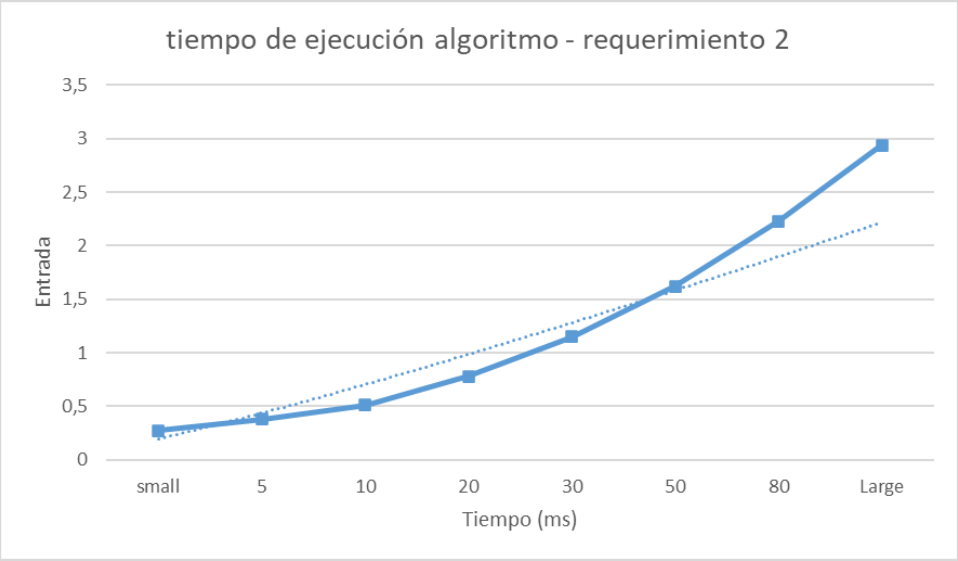
Entrada	**Tiempo (ms)**
small	0,27
5 pct	0,38
10 pct	0,51
20 pct	0,78
30 pct	1,15
50 pct	1,62
80 pct	2,23
large	2,94

Tablas de datos

Muestra	**Salida**	**Tiempo (ms)**
small	Dato1	0,27
5 pct	Dato2	0,38
10 pct	Dato3	0,51
20 pct	Dato4	0,78
30 pct	Dato5	1,15
50 pct	Dato6	1,62
80 pct	Dato7	2,23

large	Dato8	2,94

Gráfica



Análisis

Algunos errores que se pueden considerar significativos en la toma de datos pueden ser que el computador tenía otros programas abiertos durante la toma de datos, y, además, que el probar el programa tantas veces seguidas aumentaba el tiempo real de la prueba. Tomando en cuenta lo anterior, se puede notar que se afirma el análisis de complejidad temporal, pues hay un crecimiento de NlogN. Este resultado se puede ver en la línea de tendencia expuesta en la misma gráfica. Esta complejidad temporal se debe en su totalidad al mecanismo de ordenamiento merge sort. Si no hubiese sido por este, el ordenamiento tendía una complejidad de O(N), pues el resto de los pasos del algoritmo son, en su mayoría, lineales. Sin embargo, aunque la complejidad del algoritmo incremente al emplear merge sort, este es uno de los mejores mecanismos de ordenamiento en cuanto a la complejidad temporal.

Requerimiento 3

Descripción

```python
def req_3(data_structs, anio):
    """
    Función que soluciona el requerimiento 1
    """
    lista_del_anio= ""
    if mp.contains(data_structs["map_anio"], anio):
        lista_del_anio= mp.get(data_structs["map_anio"], anio)
    impuestos= lista_del_anio['value']['impuestos']
    map_sub_sec= mp.newMap(numelements= lt.size(impuestos))
    for z in lt.iterator(impuestos):
        subsector= z['Código subsector económico']
        cant_ret= 0
        for x in lt.iterator(impuestos):
            if subsector == x['Código subsector económico']:
                cant_ret += int(x['Total retenciones'])
        if subsector not in  map_sub_sec:
            mp.put(map_sub_sec, subsector, cant_ret)
    cont= 1
    ind= 0
    men_tot_ret= 100000000000000000000000
    for c in lt.iterator(mp.valueSet(map_sub_sec)):
        if int(c)< men_tot_ret:
            men_tot_ret = int(c)
            ind= cont
        cont += 1
    cod_menor_ret= lt.getElement(mp.keySet(map_sub_sec),ind)
    tot_ret= 0
    tot_in_net= 0
    tot_c_y_g= 0
    tot_s_a_p= 0
    tot_s_a_f= 0
    for v in lt.iterator(impuestos):
        if v['Código subsector económico'] == cod_menor_ret:
            cod_sec= v['Código sector económico']
            nom_sec= v['Nombre sector económico']
            cod_s_sec= v['Código subsector económico']
            nom_s_sec= v['Nombre subsector económico']
            tot_ret += int(v['Total retenciones'])
            tot_in_net += int(v['Total ingresos netos'])
            tot_c_y_g += int(v['Total costos y gastos'])
            tot_s_a_p += int(v['Total saldo a pagar'])
            tot_s_a_f += int(v['Total saldo a favor'])
    dic_final= {"Código sector económico": [cod_sec],'Nombre sector económico': [nom_sec], 'código subsector económico' : [cod_s_sec] , 'Nombre subsector económico': [nom_s_sec],'Total retenciones': [tot_ret],
                'Total ingresos netos': [tot_in_net], 'Total costos y gastos': [tot_c_y_g], 'Total saldo a pagar': [tot_s_a_p],'Total saldo a favor': [tot_s_a_f] }
    list_fin= lt.newList('ARRAY_LIST')
    for b in lt.iterator(impuestos):
        if b['Código subsector económico'] == cod_menor_ret:
            lt.addLast(list_fin, {"Código actividad económica": b['Código actividad económica'], 'Nombre actividad económica': b['Nombre actividad económica'], 'Total retenciones': b['Total retenciones'],
                                  'Total ingresos netos': b['Total ingresos netos'], 'Total costos y gastos': b['Total costos y gastos'], 'Total saldo a pagar':b['Total saldo a pagar'], 'Total saldo a favor':b['Total saldo a favor']})
    if lt.size(list_fin)< 6:
        return dic_final, list_fin
    else:
        lista_a_usar= list_fin.copy()
        list_mas= lt.newList("ARRAY_LIST")
        qpa= 3
        while qpa!=0:
            mayor_apoyo= -1
            m= ""
            mayor= ""
            qu = 0
            for x in lt.iterator(lista_a_usar):
                qu += 1
                if int(x["Total retenciones"]) > mayor_apoyo:
                    mayor_apoyo= int(x["Total retenciones"])
                    mayor= qu
                    m= x
            lt.addLast(list_mas, m)
            lt.deleteElement(lista_a_usar, mayor)
            qpa -=1
        qpa2= 3
        list_menos= lt.newList("ARRAY_LIST")
        while qpa2!=0:
            menor_apoyo= 100000000000000000000
            m= ""
            menor= ""
            qu = 0
            for x in lt.iterator(list_fin):
                qu += 1
                if int(x["Total retenciones"]) > menor_apoyo:
                    menor_apoyo= int(x["Total retenciones"])
                    menor= qu
                    m= x
            lt.addLast(list_menos, m)
            lt.deleteElement(list_fin, menor)
            qpa2 -=1
        return dic_final, list_mas, list_menos
```

Este requerimiento se en carga de retornar el subsector económico que tuvo el menor total de retenciones (Total retenciones) para un año especifico. Encuentra el año indicado, crea un mapa de datos correspondientes al subsector económico y retorna un diccionario con los datos del subsector con el menor total de retenciones y una lista de las actividades económicas que más y menos apoyaron en retenciones al subsector.

Entrada	Estructuras de datos del modelo, año.
Salidas	Diccionario subsector con el menor total de retenciones, lista más y menos aportes
Implementado (Sí/No)	Si, Maria Alejandra Londoño.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Paso 1: Encontrar en datastructs el año indicado	O(N)
Paso 2: Crear mapa del subsector económico.	O(N^2)
Paso 3: Hallar el menor total de retenciones.	O(N)
Paso 4: Crear diccionario menor total retenciones.	O(N)
Paso 5: Crear lista más y menos aporte.	O(N)
TOTAL	***O(N^2)***

Pruebas Realizadas

Las pruebas realizadas fueron realizadas en una maquina con las siguientes especificaciones. Los datos de entrada fueron 2013.

Procesadores	**11th Gen Intel(R) Core (TM) i7-11370H @ 3.30GHz 3.30 GHz**
Memoria RAM	16 GB
Sistema Operativo	Windows 10

Entrada	**Tiempo (ms)**
small	0.3752999994903803
5 pct	0.8459999989718199
10 pct	1.6094999983906746
20 pct	2.137899998575449
30 pct	4.244699999690056
50 pct	9.87079999782145
80 pct	19.575300000607967
large	29.473600002005696

Tablas de datos

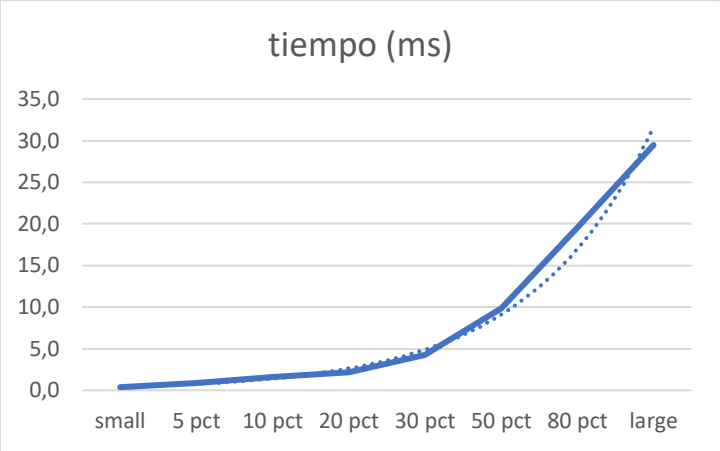
Las tablas con la recopilación de datos de las pruebas.

Muestra	**Salida**	**Tiempo (ms)**
small	Dato1	0.3752999994903803
5 pct	Dato2	0.8459999989718199
10 pct	Dato3	1.6094999983906746
20 pct	Dato4	2.137899998575449
30 pct	Dato5	4.244699999690056
50 pct	Dato6	9.87079999782145
80 pct	Dato7	19.575300000607967

large	Dato8	29.473600002005696

Graficas

Las gráficas con la representación de las pruebas realizadas.



Análisis

Se puede ver que el requerimiento, como era esperado, si cumple con un orden lineal de O(n^2). Esto se puede bien evidenciar en la gráfica y en los datos de tiempo en las tablas.

Requerimiento 4

Descripción

```python
def req_4(data_structs,anio):
    """
    Función que soluciona el requerimiento 4. Como analista económico Deseo identificar el subsector económico que tuvo los mayores costos y gastos de nómina (Costos y gastos nómina) para un año específico.
    """
    lista_totales =   ["Costos y gastos nómina", "Total ingresos netos","Total costos y gastos","Total saldo a pagar", "Total saldo a favor"]
    entry_anio = mp.get(data_structs["map_anio"], anio)
    datos_anio = me.getValue(entry_anio) #ESTO NO ES LA LISTA, SON LOS DATOS
    lista_impuestos = datos_anio["impuestos"]
    subsectores_del_anio =  mp.newMap(30,
                            maptype='CHAINING',
                            loadfactor=(4),
                            cmpfunction=compareCodigoSubsector)
    cod_mayor_subs, mayor_cost_y_nom  = -1 , 0
    for impuesto in lt.iterator(lista_impuestos):
        codigo_subsector = int(impuesto["Código subsector económico"])
        if cod_mayor_subs == -1 and mayor_cost_y_nom ==0:
            cod_mayor_subs = codigo_subsector
            mayor_cost_y_nom = int(impuesto["Costos y gastos nómina"])
        if not mp.contains(subsectores_del_anio, codigo_subsector):
            actividades_eco = lt.newList(datastructure="ARRAY_LIST", cmpfunction= compareCodigoActividad)
            lt.addLast(actividades_eco,impuesto)
            info_subsector = {"Nombre sector económico":impuesto["Nombre sector económico"],"Código sector económico":impuesto["Código sector económico"],"Código subsector económico": codigo_subsector,
                              "Actividades económicas": actividades_eco}
            for total in lista_totales:
                info_subsector[total] = int(impuesto[total])
            mp.put(subsectores_del_anio, codigo_subsector, info_subsector)
            if mayor_cost_y_nom < info_subsector["Costos y gastos nómina"]:
                cod_mayor_subs = codigo_subsector
                mayor_cost_y_nom = info_subsector["Costos y gastos nómina"]
        else:
            entry_info_subsector = mp.get(subsectores_del_anio,codigo_subsector)
            info_subsector = me.getValue(entry_info_subsector)
            for total in lista_totales:
                info_subsector[total] = int(impuesto[total]) + int(info_subsector[total])
            lt.addLast(info_subsector["Actividades económicas"], impuesto)
            mp.put(subsectores_del_anio, codigo_subsector, info_subsector)
            if mayor_cost_y_nom < info_subsector["Costos y gastos nómina"]:
                cod_mayor_subs = codigo_subsector
                mayor_cost_y_nom = info_subsector["Costos y gastos nómina"]
    entry_mayor_subsector = mp.get(subsectores_del_anio,cod_mayor_subs)
    mayor_subsector = me.getValue(entry_mayor_subsector)
    mayor_subsector["Actividades económicas"] = merg.sort(mayor_subsector["Actividades económicas"], sort_criteria_nomina) #LAS ORDENAMOS PARA LUEGO TOMAR LAS 3 PRIMERAS Y LAS 3 ULTIMAS
    actividades_toreplace = lt.newList(datastructure="ARRAY_LIST", cmpfunction=compareCodigoActividad)
    i = 1
    for actividad in lt.iterator(mayor_subsector["Actividades económicas"]):
        if  (i <=3)  or   (i >=  data_size_lista( mayor_subsector["Actividades económicas"] )-2):
            lt.addLast(actividades_toreplace,actividad)
        i+=1
    mayor_subsector["Actividades económicas"] = actividades_toreplace
    return mayor_subsector
```

Este requerimiento se encarga de hallar el subsector económico con los mayores costos y gastos de nómina en un año en específico. Para encontrar el subsector se realiza un mapa de subsectores a partir del año dado por parámetro (subsectores del año), y mientras se van agregando los subsectores al mapa se identifica cual es el código del subsector que tuvo el mayor total de costos y gastos nómina. Cuando es obtenido el mejor subsector a partir de su código, se organizan sus actividades de acuerdo con el total de costos y gastos nómina y si son menos de 6 se toma esa lista ordenada, si son más, se elegirá la mejor y peor actividad de este subsector y se retornará junto con el resto de sus características.

Entrada	El data structs con el que se puede acceder al mapa de los años y el año elegido por el usuario.
Salidas	El subsector económico con los mayores costos y gastos nomina (Total costos y gastos nómina del subsector) para un año específico, junto a las características de este subsector de acuerdo con las especificaciones del enunciado
Implementado (Sí/No)	Sí, por María Alejandra Pinzón

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

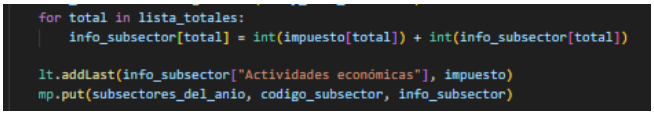
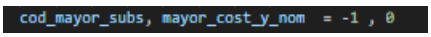
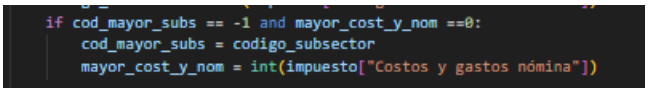
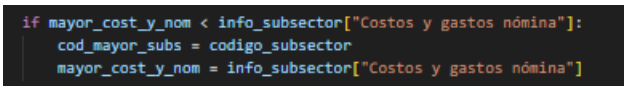
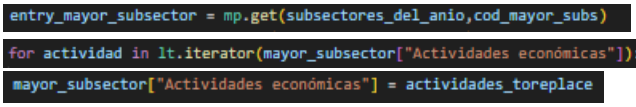
Pasos	Complejidad

Paso 1: Definición del mapa de subsectores del año cuyas llaves serán el código del subsector. Esto, a partir del mapa que entra por parámetro por años, seleccionando el valor del año que ingresa el usuario.
`entry_anio = mp.get(data_structs["map_anio"], anio)`
`datos_anio = me.getValue(entry_anio)`
`lista_impuestos = datos_anio["impuestos"]`
`subsectores_del_anio = mp.newMap(30, maptype='CHAINING', loadfactor=(4), cmpfunction=compareCodigoSubsector)`
	O (1)

Paso 2: Creación del mapa de subsectores del año, en esta parte se recorrerá toda la lista de impuestos del determinado año, para ir añadiéndolas al mapa de subsectores cada vez que encuentra un código diferente, este será una llave.
`for impuesto in lt.iterator(lista_impuestos):`
`codigo_subsector = int(impuesto["Código subsector económico"])`
`if not mp.contains(subsectores_del_anio, codigo_subsector):`
`else:`
`entry_info_subsector = mp.get(subsectores_del_anio,codigo_subsector)`
	O(N)

Paso 3: Mientras se crea el mapa de subsectores del año, cada vez que se encuentra un nuevo código (llave) , se agrega al respectivo valor de la llave un diccionario “info_subsector” el cual contiene los elementos que pide el enunciado como “Nombre sector económico”, y para las actividades se está creando una lista la cual contendrá todas las actividades de ese subsector.
`actividades_eco = lt.newList(datastructure="ARRAY_LIST", cmpfunction= compareCodigoActividad)`
`lt.addLast(actividades_eco,impuesto)`
`info_subsector = {"Nombre sector económico":impuesto["Nombre sector económico"], "Código sector económico":impuesto["Código sector económico"], "Código subsector económico": codigo_subsector, "Actividades económicas": actividades_eco}`
	O(N)

Paso 4: Mientras se crea el mapa de subsectores del año, se realizara un ciclo para hallar la suma de todos los totales que pide el enunciado, en este ciclo se utilizara la misma estructura para calcular la suma. Cada suma se agregara como una llave en el diccionario “info_subsector”, el cual es el valor que corresponde a la llave de código de subsector en el mapa de subsectores por año.
`lista_totales = ["Costos y gastos nómina", "Total ingresos netos","Total costos y gastos", "Total saldo a pagar", "Total saldo a favor"]`
`for total in lista_totales:`
`info_subsector[total] = int(impuesto[total])`
`mp.put(subsectores_del_anio, codigo_subsector, info_subsector)`
	O(N^2)

	
Paso 5: Mientras se esta creando el mapa de subsectores, se aprovecha el recorrido que se realiza para empezar a encontrar el subsector con los mayores costos y gastos de nómina, para lo cual cada vez que se encuentre un nuevo código de subsector mediante las llaves del mapa subsectores del año, se compararan los valores de "Costos y gastos nomina" entre los subsectores para obtener el código del subsector con mayores costos y gastos de nómina de ese año.   	O(N)
Paso 6: Ahora que tenemos el código del subsector con mayores costos y gastos de nómina de ese año, se utilizará para extraer su valor a partir del mapa de subsectores por año. Luego de tener el mayor, se obtienen las actividades y se realiza un ciclo para identificar las 3 mejores y 3 peores. 	O(N)
TOTAL 6	**O(N^2)**

Pruebas Realizadas

Las pruebas realizadas fueron realizadas en una maquina con las siguientes especificaciones. Los datos de entrada fueron 2012. Y al cargar los datos se utilizó chaining con factor de carga 4.

Procesadores	**11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz 2.42 GHz**
Memoria RAM	12 GB
Sistema Operativo	Windows 11

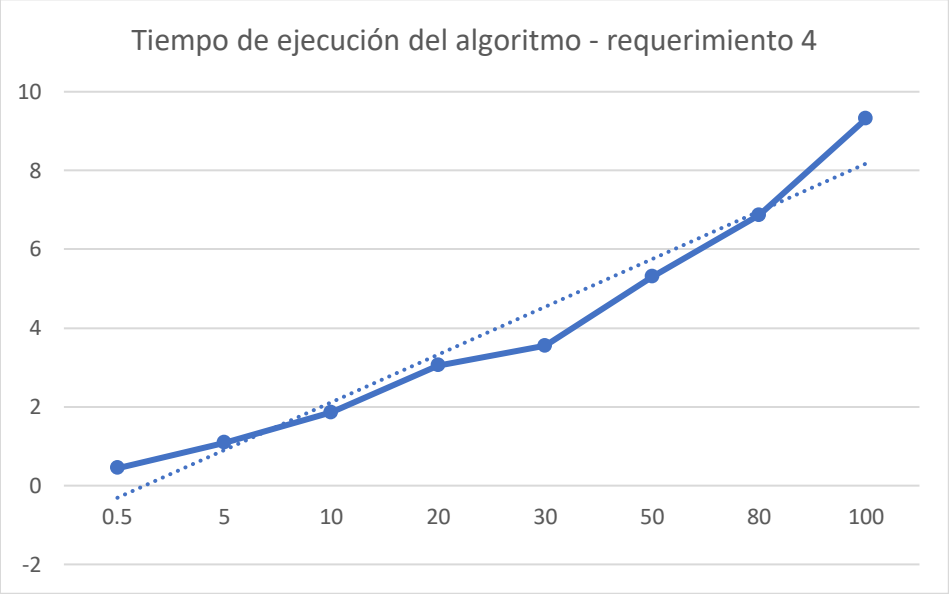
Entrada	**Tiempo (ms)**

small	0.4426999092102051
5 pct	1.09089994430542
10 pct	1.8609001636505127
20 pct	3.059499979019165
30 pct	3.5541000366210938
50 pct	5.302900075912476
80 pct	6.875699758529663
large	9.317300081253052

Tablas de datos

Muestra	Salida	Tiempo (ms)
small	Dato1	0.4426999092102051
5 pct	Dato2	1.09089994430542
10 pct	Dato3	1.8609001636505127
20 pct	Dato4	3.059499979019165
30 pct	Dato5	3.5541000366210938
50 pct	Dato6	5.302900075912476
80 pct	Dato7	6.875699758529663
large	Dato8	9.317300081253052

Gráfica



Análisis

Al realizar la toma de datos variando el tamaño de estos e identificado el tiempo en el que demoran retornando la respuesta del requerimiento, se evidencio una tendencia creciente, lo cual indica que está bien, debido a que es correcto que a medida que aumente el tamaño de los datos, el tiempo de ejecución también lo hará. En cuanto a la respuesta retornada por el requerimiento se evidencia que esta es la deseada y que los elementos que contiene son los adecuados dadas las instrucciones del requerimiento. Su complejidad es la esperada debido a que se realiza un for dentro de otro ciclo de for, pero puede no ser la esperada si se quiere analizar desde la eficiencia, aunque *O(N^2)* es una complejidad que no hace que el requerimiento demore mas de lo esperado, este requerimiento sigue siendo eficiente.

Requerimiento 5

Descripción

```
def req_5(data_structs,year):
    """
    Función que soluciona el requerimiento 5
    """
    sumas_q_me_piden = ['Código sector económico', "Nombre sector económico", "Nombre subsector económico", 'Código subsector económico', "Descuentos tributarios", "Total ingresos netos", "Total costos y gastos",
    |   |   |   |   |   "Total saldo a pagar", "Total saldo a favor"]
    year_entry = mp.get(data_structs['map_anio'], year)
    year_datos = me.getValue(year_entry)
    lista_actecon = year_datos['impuestos']
    aja = lt.newList('ARRAY_LIST')
    for act_econ in lt.iterator(lista_actecon):
    |   lt.addLast(aja, act_econ)
    por_subsector = {}
    for i in lt.iterator(aja):
    |   if i ['Código subsector económico'] not in por_subsector:
    |   |   por_subsector[i ['Código subsector económico']]=lt.newList(datastructure= "ARRAY_LIST")
    |   |   lt.addLast( por_subsector[i['Código subsector económico']] ,i)
    |   else:
    |   |   lt.addLast(por_subsector[i['Código subsector económico']], i)
    lista_sqmepiden = [[[x[sumas_q_me_piden[0]], x[sumas_q_me_piden[1]], x[sumas_q_me_piden[2]],x[sumas_q_me_piden[3]], x[sumas_q_me_piden[4]], x[sumas_q_me_piden[5]], x[sumas_q_me_piden[6]],
    |   |   |   |   |   x[sumas_q_me_piden[7]], x[sumas_q_me_piden[8]]] for x in por_subsector[subsector]['elements']]for subsector in por_subsector]
    final1 = lt.newList("ARRAY_LIST")
    mayor = 0
    mayor_codigosub = 0
    for sub in lista_sqmepiden:
    |   suma_descuentos = 0
    |   for actividades in sub:
    |   |   suma_descuentos += int(actividades[4])
    |   |   suma_porsub = suma_descuentos
    |   |   if suma_porsub > mayor:
    |   |   |   mayor = suma_porsub
    |   |   |   mayor_nombresub = actividades[2]
    |   |   |   mayor_codigosub = actividades[3]
    theoneandonly= mayor_codigosub
    lt.addLast(final1, mayor_nombresub)
    lt.addLast(final1, mayor_codigosub)
    lt.addLast(final1, str(mayor))
    for sub in lista_sqmepiden:
    |   total_ing = 0
    |   total_cyg = 0
    |   total_sap = 0
    |   total_saf = 0
    |   for actividades in sub:
    |   |   if theoneandonly == actividades[3]:
    |   |   |   total_ing += int(actividades[5])
    |   |   |   total_cyg += int(actividades[6])
    |   |   |   total_sap += int(actividades[7])
    |   |   |   total_saf += int(actividades[8])
    |   |   |   mayor_ing = str(total_ing)
    |   |   |   mayor_cyg = str(total_cyg)
    |   |   |   mayor_sap = str(total_sap)
    |   |   |   mayor_saf = str(total_saf)
    lt.addLast(final1, mayor_ing)
    lt.addLast(final1, mayor_cyg)
    lt.addLast(final1, mayor_sap)
    lt.addLast(final1, mayor_saf)
    subsector_actecon= por_subsector[mayor_codigosub]
    merg.sort(subsector_actecon, cmp_descuentostrib)
    mayndmen = lt.newList('ARRAY_LIST')
    if lt.size(subsector_actecon) > 6:
    |   lt.addLast(mayndmen, lt.getElement(subsector_actecon, 1))
    |   lt.addLast(mayndmen, lt.getElement(subsector_actecon, 2))
    |   lt.addLast(mayndmen, lt.getElement(subsector_actecon, 3))
    |   lt.addLast(mayndmen, lt.getElement(subsector_actecon, lt.size(subsector_actecon)))
    |   lt.addLast(mayndmen, lt.getElement(subsector_actecon, lt.size(subsector_actecon)-1))
    |   lt.addLast(mayndmen, lt.getElement(subsector_actecon, lt.size(subsector_actecon)-2))
    else:
    |   for i in lt.iterator(subsector_actecon):
    |   |   lt.addLast(mayndmen, i)
    headz_1 = ["Nombre subsector económico", 'Código subsector económico', "Total descuentos tributarios", "Total ingresos netos", "Total costos y gastos", "Total saldo a pagar", "Total saldo a favor"]
    headz_2 = ["Código actividad económica","Nombre actividad económica","Descuentos tributarios", "Total ingresos netos", "Costos y gastos nómina", "Total saldo a pagar","Total saldo a favor"]
    mayndmen_final = [[x[headz_2[0]], x[headz_2[1]], x[headz_2[2]],x[headz_2[3]], x[headz_2[4]], x[headz_2[5]], x[headz_2[6]]]for x in mayndmen['elements']]
    return final1, headz_1, mayndmen_final, headz_2
def cmp_descuentostrib (data1, data2):
    return int(data1['Descuentos tributarios']) < int(data2['Descuentos tributarios'])
```

Este requerimiento se encarga de encontrar el subsector económico que tuvo los mayores descuentos tributarios (Descuentos tributarios) para un año específico que es dado por el usuario.

Entrada	El map principal y el año elegido por el usuario.
Salidas	el subsector económico que tuvo los mayores descuentos tributarios (Descuentos tributarios) para un año específico, junto a sus características pedidas en el enunciado.
Implementado (Sí/No)	Sí, por Gabriela Escobar

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**

`year_entry = mp.get(data_structs['map_anio'], year)`
`year_datos = me.getValue(year_entry)`
`lista_actecon = year_datos['impuestos']`
	O(N)

`for act_econ in lt.iterator(lista_actecon):` `lt.addLast(aja, act_econ)`	O(N)

`lista_sqmepiden = [[[x[sumas_q_me_piden[0]],`
`x[sumas_q_me_piden[1]], x[sumas_q_me_piden[2]],`
`x[sumas_q_me_piden[3]], x[sumas_q_me_piden[4]],`
`x[sumas_q_me_piden[5]], x[sumas_q_me_piden[6]],`
`x[sumas_q_me_piden[7]], x[sumas_q_me_piden[8]]]`
`for x in por_subsector[subsector]['elements']]`
`for subsector in por_subsector]`
	O(N^2)

`lt.addLast(final1, mayor_nombresub)`	O(NlogN)
`merg.sort(subsector_actecon, cmp_descuentostrib)`	O (N)

`mayndmen_final = [[x[headz_2[0]], x[headz_2[1]],x[headz_2[2]],`
`x[headz_2[3]], x[headz_2[4]], x[headz_2[5]], x[headz_2[6]]]`
`for x in mayndmen['elements']]`
	O (N)

TOTAL	***O (N^2)***

Pruebas Realizadas

Las pruebas realizadas fueron realizadas en una maquina con las siguientes especificaciones. Los datos de entrada fueron para el año 2021. El programa se probó utilizando el esquema de colisión chaining, con el factor de carga de 8.

Procesador	**11th Gen Intel(R) Core(TM) i7-1195G7 @ 2.90GHz 2.92 GHz**
Memoria RAM	32GB
Sistema Operativo	Windows 11 Pro- 64 bits

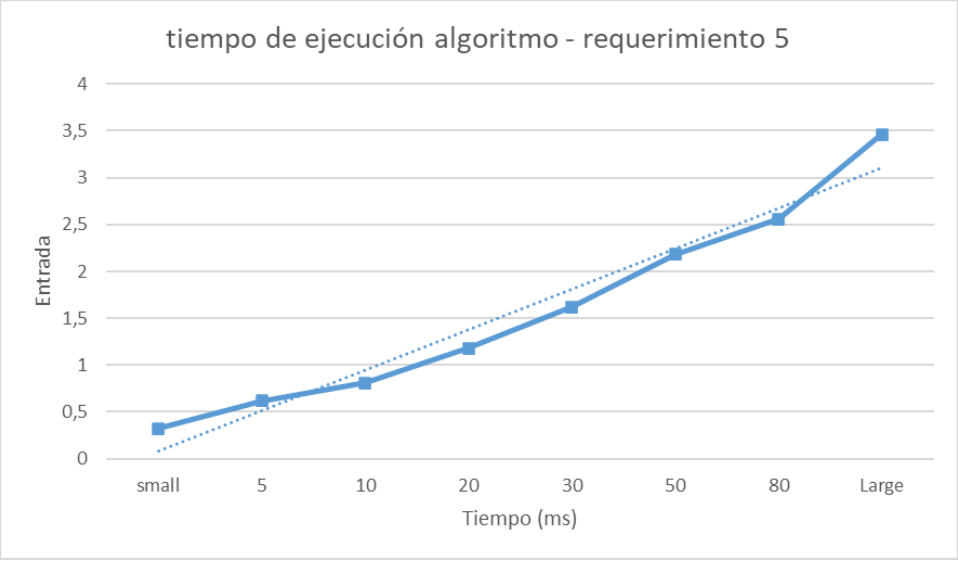
Entrada	**Tiempo (ms)**
small	0,32

5 pct	0,62
10 pct	0,81
20 pct	1,18
30 pct	1,62
50 pct	2,18
80 pct	2.56
large	3,46

Tablas de datos

Muestra	Salida	Tiempo (ms)
small	Dato1	0,32
5 pct	Dato2	0,62
10 pct	Dato3	0,81
20 pct	Dato4	1,18
30 pct	Dato5	1,62
50 pct	Dato6	2,18
80 pct	Dato7	2.56
large	Dato8	3,46

Gráfica



Análisis

Algunos errores que se pueden considerar significativos en la toma de datos pueden ser que el computador tenía otros programas abiertos durante la toma de datos, y, además, que el probar el programa tantas veces seguidas aumentaba el tiempo real de la prueba. Tomando en cuenta lo anterior, se puede notar que se afirma el análisis de complejidad temporal, pues hay un crecimiento cuadrático. Este resultado se puede ver en la línea de tendencia expuesta en la misma gráfica. Esta complejidad temporal se debe en su totalidad al doble iteraciones, una dentro de otra. Aunque esta complejidad podría ser más eficiente, se puede ver un avance, en comparación a requerimientos de similar complejidad, del reto pasado. Esto demuestra que las habilidades aprendidas este nivel fueron empleadas para mejorar en términos de eficiencia.

Requerimiento 6

Descripción

```
def req_6(data_structs,anio):
    """ Función que soluciona el requerimiento 6 """
    lista_totales =   ["Costos y gastos nómina", "Total ingresos netos","Total costos y gastos","Total saldo a pagar", "Total saldo a favor"]
    entry_anio = mp.get(data_structs["map_anio"], anio)
    datos_anio = me.getValue(entry_anio) #ESTO NO ES LA LISTA, SON LOS DATOS
    lista_impuestos = datos_anio["impuestos"]
    sectores_del_anio =  mp.newMap(40,
                                   maptype='CHAINING',
                                   loadfactor=(4),
                                   cmpfunction=compareCodigoSector)
    cod_mayor_sec, mayor_total_net_sec = -1 , 0
    for impuesto in lt.iterator(lista_impuestos):
        codigo_sector = int(impuesto["Código sector económico"])
        codigo_subsector = int(impuesto["Código subsector económico"])
        if cod_mayor_sec == -1:
            cod_mayor_sec = codigo_sector
            mayor_total_net_sec = int(impuesto["Total ingresos netos"])
        if not mp.contains(sectores_del_anio, codigo_sector):
            subsectores_del_anio =  mp.newMap(40,
                                maptype='CHAINING',
                                loadfactor=(4),
                                cmpfunction=compareCodigoSector)
            actividades_eco = lt.newList(datastructure="ARRAY_LIST", cmpfunction= compareCodigoActividad)
            lt.addLast(actividades_eco,impuesto)
            info_subsector = {"Código subsector económico": codigo_subsector, "Actividades económicas": actividades_eco}
            for total in lista_totales:
                info_subsector[total] = int(impuesto[total])
            mp.put(subsectores_del_anio, codigo_subsector, info_subsector)
            info_sector = {"Nombre sector económico":impuesto["Nombre sector económico"],"Código sector económico":impuesto["Código sector económico"],
                           "Subsector económico": subsectores_del_anio}
            for total in lista_totales:
                info_sector[total] = int(impuesto[total])
            mp.put(sectores_del_anio, codigo_sector, info_sector)
        else:
            entry_info_sector = mp.get(sectores_del_anio,codigo_sector)
            info_sector = me.getValue(entry_info_sector)
            for total in lista_totales:
                info_sector[total] = int(impuesto[total]) + int(info_sector[total])
            if not mp.contains(info_sector["Subsector económico"], codigo_subsector):
                actividades_eco = lt.newList(datastructure="ARRAY_LIST", cmpfunction= compareCodigoActividad)
                lt.addLast(actividades_eco,impuesto)
                info_subsector = {"Código subsector económico": codigo_subsector, "Actividades económicas": actividades_eco}
                for total in lista_totales:
                    info_subsector[total] = int(impuesto[total])
                mp.put(info_sector["Subsector económico"], codigo_subsector, info_subsector)
            else:
                entry_info_subsector = mp.get(info_sector["Subsector económico"],codigo_subsector)
                info_subsector = me.getValue(entry_info_subsector)
                for total in lista_totales:
                    info_subsector[total] = int(impuesto[total]) + int(info_subsector[total])
                lt.addLast(info_subsector["Actividades económicas"], impuesto)
                mp.put(info_sector["Subsector económico"], codigo_subsector, info_subsector)
            if mayor_total_net_sec < int(info_sector["Total ingresos netos"]):
                cod_mayor_sec = codigo_sector
                mayor_total_net_sec= int(info_sector["Total ingresos netos"])
            mp.put(sectores_del_anio, codigo_sector, info_sector)
    entry_mejor_sector = mp.get(sectores_del_anio, cod_mayor_sec)
    mejor_sector = me.getValue(entry_mejor_sector)
    cod_mayor_subs, mayor_total_net_subs  = -1 , 0
    cod_menor_subs, menor_total_net_subs  = -1 , 0
    codigos_subs_map_sec = mp.keySet(mejor_sector["Subsector económico"])
    for key in lt.iterator(codigos_subs_map_sec):
        entry_subsetor_get = mp.get(mejor_sector["Subsector económico"], key)
        info_subsector_get = me.getValue(entry_subsetor_get)
        if (cod_mayor_subs == -1) or  mayor_total_net_subs < info_subsector["Total ingresos netos"]:
            cod_mayor_subs = key
```

```
            mayor_total_net_subs = int(info_subsector_get["Total ingresos netos"])
        if (cod_menor_subs == -1) or  menor_total_net_subs > info_subsector["Total ingresos netos"]:
            cod_menor_subs = key
            menor_total_net_subs = int(info_subsector_get["Total ingresos netos"])
    entry_mayor_subsector = mp.get(mejor_sector["Subsector económico"] ,cod_mayor_subs)
    mayor_subsector = me.getValue(entry_mayor_subsector)
    entry_menor_subsector = mp.get(mejor_sector["Subsector económico"] ,cod_menor_subs)
    menor_subsector = me.getValue(entry_menor_subsector)
    # Procedimiento para ajustar el MEJOR SUBSECTOR y eliminar su lista de actividades
    mejor_act_del_mejorsubs = lt.firstElement(mayor_subsector["Actividades económicas"])
    peor_act_del_mejorsubs = lt.firstElement(mayor_subsector["Actividades económicas"])
    for actividad in lt.iterator(mayor_subsector["Actividades económicas"]):
        if  int(mejor_act_del_mejorsubs["Total ingresos netos"]) < int(actividad["Total ingresos netos"]):
            mejor_act_del_mejorsubs = actividad
        if  int(peor_act_del_mejorsubs["Total ingresos netos"]) < int(actividad["Total ingresos netos"]):
            peor_act_del_mejorsubs = actividad
    mayor_subsector["Mejor actividad"] = mejor_act_del_mejorsubs
    mayor_subsector["Peor actividad"] = peor_act_del_mejorsubs
    mayor_subsector.pop("Actividades económicas",-1)
    # PROCEDIMIENTO PARA LO MISMO PERO EN EL PEOR SUBSECTOR
    mejor_act_del_peorsubs = lt.firstElement(menor_subsector["Actividades económicas"])
    peor_act_del_peorsubs = lt.firstElement(menor_subsector["Actividades económicas"])
    for actividad in lt.iterator(menor_subsector["Actividades económicas"]):
        if  int(mejor_act_del_mejorsubs["Total ingresos netos"]) < int(actividad["Total ingresos netos"]):
            mejor_act_del_peorsubs = actividad
        if  int(peor_act_del_mejorsubs["Total ingresos netos"]) < int(actividad["Total ingresos netos"]):
            peor_act_del_peorsubs = actividad
    menor_subsector["Mejor actividad"] = mejor_act_del_peorsubs
    menor_subsector["Peor actividad"] = peor_act_del_peorsubs
    menor_subsector.pop("Actividades económicas",-1)
    #Borrar map de subsectores y meter el mejor y el peor
    mejor_sector["Mejor subsector"] = mayor_subsector
    mejor_sector["Peor subsector"] = menor_subsector
    mejor_sector.pop("Subsector económico",-1)
    return mejor_sector
```

Este requerimiento se encarga de encontrar el sector que contiene los mayores ingresos netos. En la respuesta se incluye el mayor y menor subsector de este, y para cada uno se incluirá su mayor y menor actividad.

Entrada	El data structs con el que se puede acceder al mapa de los años y el año elegido por el usuario.
Salidas	
Implementado (Sí/No)	Sí, por María Alejandra Pinzón

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**

Paso 1: Definición del mapa de sectores del año cuyas llaves serán el código del sector. Esto, a partir del mapa que entra por parámetro por años, seleccionando el valor del año que ingresa el usuario.
`entry_anio = mp.get(data_structs["map_anio"], anio)`
`datos_anio = me.getValue(entry_anio) #ESTO NO ES LA LISTA, SON LOS DATOS`
`lista_impuestos = datos_anio["impuestos"]`
`sectores_del_anio =  mp.newMap(40, maptype='CHAINING', loadfactor=(4), cmpfunction=compareCodigoSector)`
	O (1)

Paso 2: Creación del mapa de sectores del año, en esta parte se recorrerá toda la lista de impuestos del	O(N)

determinado año, para ir añadiéndolas al mapa de sectores cada vez que encuentra un código diferente, este será una llave.
`for impuesto in lt.iterator(lista_impuestos):`
`  codigo_subsector = int(impuesto["Código subsector económico"])`
`if not mp.contains(sectores_del_anio, codigo_sector):`
`else:`
`  entry_info_sector = mp.get(sectores_del_anio,codigo_sector)`	

Paso 3: Mientras se crea el mapa de sectores del año, cada vez que se encuentra un nuevo código (llave), se agrega al respectivo valor de la llave un diccionario "info_sector" el cual contiene los elementos que pide el enunciado como "Nombre sector económico", para los subsectores, se estará creando un mapa subsectores de ese año y de ese subsector, el cual se agregará como un valor en ese mismo diccionario con la llave "Subsector económico", y para las actividades se está creando una lista la cual contendrá todas las actividades de ese subsector, estas estarán ubicadas en el map de subsectores del año del subsector.
`subsectores_del_anio = mp.newMap(40,`
`  maptype='CHAINING',`
`  loadfactor=(4),`
`  cmpfunction=compareCodigoSector)`
`actividades_eco = lt.newList(datastructure="ARRAY_LIST", cmpfunction= compareCodigoActividad)`
`lt.addLast(actividades_eco,impuesto)`
`info_subsector = {"Código subsector económico": codigo_subsector, "Actividades económicas": actividades_eco}`
`info_sector = {"Nombre sector económico":impuesto["Nombre sector económico"],`
`  "Código sector económico":impuesto["Código sector económico"],`
`  "Subsector económico": subsectores_del_anio}`
	O(N)

Paso 4: Mientras se crea el mapa de sectores del año, se realizará un ciclo para hallar la suma de todos los totales que pide el enunciado, en este ciclo se utilizara la misma estructura para calcular la suma. Cada suma se agregará como una llave en el diccionario "info_sector", el cual es el valor que corresponde a la llave de código de sector en el mapa de sectores por año.
`lista_totales = ["Costos y gastos nómina", "Total ingresos netos","Total costos y gastos",`
`  "Total saldo a pagar", "Total saldo a favor"]`
`for total in lista_totales:`
`  info_sector[total] = int(impuesto[total])`
`mp.put(sectores_del_anio, codigo_sector, info_sector)`
`for total in lista_totales:`
`  info_sector[total] = int(impuesto[total]) + int(info_sector[total])`
	O(N^2)

Paso 5: Mientras se crea el mapa de sectores del año cuyos valores son mapas de subsectores, se	O(N)

encuentra el mejor sector económico a partir de su código.
`if mayor_total_net_sec < int(info_sector["Total ingresos netos"]):`
`  cod_mayor_sec = codigo_sector`
`  mayor_total_net_sec= int(info_sector["Total ingresos netos"])`	

Paso 5: Ahora que se tiene el mejor sector, es posible encontrar el mejor y peor subsector de este económico mediante un ciclo que utiliza las llaves del mapa de subsectores y así identificar el subsector con mayor y menor total de ingresos netos de ese mejor sector.
`for key in lt.iterator(codigos_subs_map_sec):`
`mayor_total_net_subs = int(info_subsector_get["Total ingresos netos"])`
`menor_total_net_subs = int(info_subsector_get["Total ingresos netos"])`
	O(N)

Paso 6: Ahora que se tiene el mayor y menor sector de acuerdo con el total de ingresos netos, se realiza un ciclo que encontrara para cada uno su mejor y peor actividad económica. `for actividad in lt.iterator(mayor_subsector["Actividades económicas"]):`	O(N)
TOTAL 6	**O(N^2)**

Pruebas Realizadas

Las pruebas realizadas fueron realizadas en una maquina con las siguientes especificaciones. Los datos de entrada fueron 2012. Y al cargar los datos se utilizó chaining con factor de carga 4.

Procesadores	**11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz 2.42 GHz**
Memoria RAM	12 GB
Sistema Operativo	Windows 11

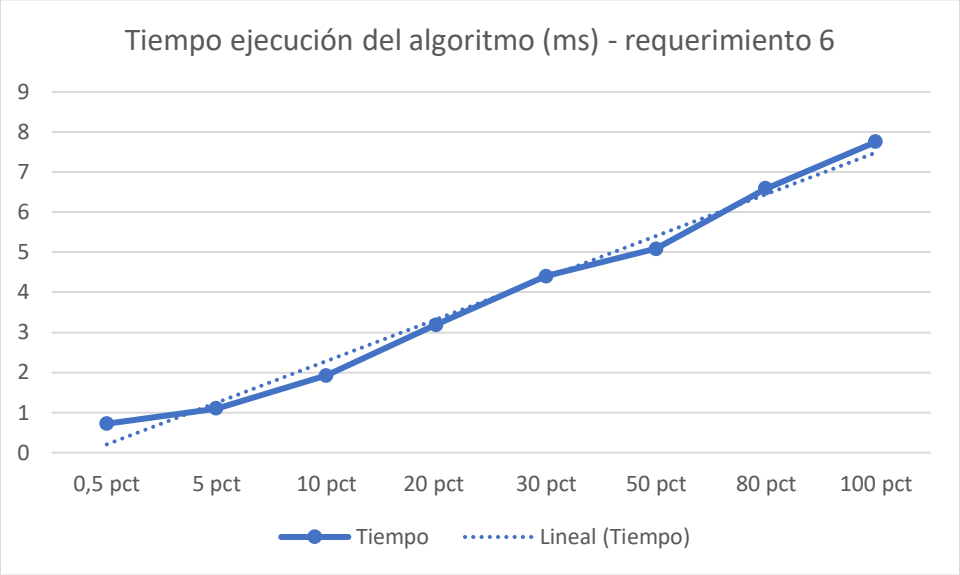
Entrada	**Tiempo (ms)**
small	0.7245998382568359
5 pct	1.1068997383117676
10 pct	1.9271998405456543
20 pct	3.1966001987457275
30 pct	4.4040000438690186
50 pct	5.085900068283081
80 pct	6.591900110244751
large	7.754800081253052

Tablas de datos

Muestra	**Salida**	**Tiempo (ms)**

small	Dato1	0.7245998382568359
5 pct	Dato2	1.1068997383117676
10 pct	Dato3	1.9271998405456543
20 pct	Dato4	3.1966001987457275
30 pct	Dato5	4.4040000438690186
50 pct	Dato6	5.085900068283081
80 pct	Dato7	6.591900110244751
large	Dato8	7.754800081253052

Gráfica



Análisis

Al realizar pruebas con diferentes tamaños de datos, se observó que el tiempo de ejecución aumentaba a medida que se incrementaba el tamaño de los datos. Este comportamiento es esperado, ya que a medida que los datos aumentan, también lo hace la cantidad de operaciones que se deben realizar para procesarlos. Además, se pudo comprobar que la respuesta obtenida del requerimiento era la esperada y que los elementos incluidos en la respuesta eran adecuados de acuerdo con las instrucciones del requerimiento. El algoritmo utilizado en el requerimiento tiene una complejidad O(N^2) debido a que se utiliza un ciclo de for anidado dentro de otro ciclo de for. Aunque esto puede no ser la opción más eficiente, el tiempo de ejecución del requerimiento sigue siendo razonable. En resumen, los resultados obtenidos sugieren que el requerimiento cumple con su objetivo y que su implementación es adecuada, aunque existen posibilidades de mejorar la eficiencia del algoritmo utilizado.

Requerimiento 7

Descripción

```python
def req_7(data_structs, top, anio, csse):
    """
    Función que soluciona el requerimiento 7
    """
    lista_del_anio= ""
    if mp.contains(data_structs["map_anio"], anio):
        lista_del_anio= mp.get(data_structs["map_anio"], anio)
    impuestos= lista_del_anio['value']['impuestos']
    lista_csse= lt.newList("ARRAY_LIST")
    for x in lt.iterator(impuestos):
        if csse== x['Código subsector económico']:
            lt.addLast(lista_csse, x)
    map_act_e= mp.newMap(numelements= lt.size(lista_csse))
    for c in lt.iterator(lista_csse):
        act= c['Código actividad económica']
        t_c_y_g= 0
        for v in lt.iterator(lista_csse):
            if act == v['Código actividad económica']:
                t_c_y_g += int(v["Total costos y gastos"])
        if act not in map_act_e:
            mp.put(map_act_e, act, t_c_y_g)
    lista_fin= lt.newList("ARRAY_LIST")
    if int(top) >= mp.size(map_act_e):
        t= mp.size(map_act_e)
        while t!=0:
            cont= 1
            ind= 0
            men_tot_c_y_g= 10000000000000000000000
            for c in lt.iterator(mp.valueSet(map_act_e)):
                if int(c)< men_tot_c_y_g:
                    men_tot_c_y_g = int(c)
                    ind= cont
                cont += 1
            cod_menor_c_y_g= lt.getElement(mp.keySet(map_act_e),ind)
            tot_ret= 0
            tot_in_net= 0
            tot_c_y_g= 0
            tot_s_a_p= 0
            tot_s_a_f= 0
            for v in lt.iterator(lista_csse):
                if v['Código actividad económica'] == cod_menor_c_y_g:
                    cod_sec= v['Código sector económico']
                    nom_sec= v['Nombre sector económico']
                    cod_s_sec= v['Código subsector económico']
                    nom_s_sec= v['Nombre subsector económico']
                    tot_ret += int(v['Total retenciones'])
                    tot_in_net += int(v['Total ingresos netos'])
                    tot_c_y_g += int(v['Total costos y gastos'])
                    tot_s_a_p += int(v['Total saldo a pagar'])
                    tot_s_a_f += int(v['Total saldo a favor'])
            dic_final= {"Código sector económico": cod_sec,'Nombre sector económico': nom_sec, 'Código subsector económico' : cod_s_sec , 'Nombre subsector económico': nom_s_sec,
                        'Total ingresos netos': tot_in_net, 'Total costos y gastos': tot_c_y_g, 'Total saldo a pagar': tot_s_a_p,'Total saldo a favor': tot_s_a_f }
            lt.addLast(lista_fin, dic_final)
            mp.remove(map_act_e, cod_menor_c_y_g)
            t -=1
    else:
        t= int(top)
        while t!=0:
            cont= 1
            ind= 0
            men_tot_c_y_g= 10000000000000000000000
            for c in lt.iterator(mp.valueSet(map_act_e)):
                if int(c)< men_tot_c_y_g:
                    men_tot_c_y_g = int(c)
                    ind= cont
                cont += 1
            cod_menor_c_y_g= lt.getElement(mp.keySet(map_act_e),ind)
            tot_ret= 0
            tot_in_net= 0
            tot_c_y_g= 0
            tot_s_a_p= 0
            tot_s_a_f= 0
            for v in lt.iterator(lista_csse):
                if v['Código actividad económica'] == cod_menor_c_y_g:
                    cod_sec= v['Código sector económico']
                    nom_sec= v['Nombre sector económico']
                    cod_s_sec= v['Código subsector económico']
                    nom_s_sec= v['Nombre subsector económico']
                    tot_ret += int(v['Total retenciones'])
                    tot_in_net += int(v['Total ingresos netos'])
                    tot_c_y_g += int(v['Total costos y gastos'])
                    tot_s_a_p += int(v['Total saldo a pagar'])
                    tot_s_a_f += int(v['Total saldo a favor'])
            dic_final= {"Código sector económico": cod_sec,'Nombre sector económico': nom_sec, 'Código subsector económico' : cod_s_sec , 'Nombre subsector económico': nom_s_sec,
                        'Total ingresos netos': tot_in_net, 'Total costos y gastos': tot_c_y_g, 'Total saldo a pagar': tot_s_a_p,'Total saldo a favor': tot_s_a_f }
            lt.addLast(lista_fin, dic_final)
            mp.remove(map_act_e, cod_menor_c_y_g)
            t -=1
    return lista_fin
```

Este requerimiento se en carga de retornar el TOP (N) de las actividades económicas con el menor total de costos y gastos para un subsector y un año específicos. Encuentra el año indicado, crea una lista con el código del subsector, crea un mapa de datos correspondientes a las actividades económicas y retorna una lista de N posiciones con los datos de las actividades económicas con el menor total de costos y gastos.

Entrada	Estructuras de datos del modelo, top (N), año, subsector económico.
Salidas	Lista de N posiciones con los datos de las actividades económicas con el menor total de costos y gastos.
Implementado (Sí/No)	Si, Maria Alejandra Londoño.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Paso 1: Encontrar en datastructs el año indicado	O(N)
Paso 2: Crear lista código subsector económico.	O(N)
Paso 3: Crear mapa del subsector económico.	O(N^2)
Paso 4: Hallar el top de menor total de costos y gastos.	O(N^2)
TOTAL	***O(N^2)***

Pruebas Realizadas

Las pruebas realizadas fueron realizadas en una maquina con las siguientes especificaciones. Los datos de entrada fueron 9, 2020, 11.

Procesadores	**11th Gen Intel(R) Core (TM) i7-11370H @ 3.30GHz 3.30 GHz**
Memoria RAM	16 GB
Sistema Operativo	Windows 10

Entrada	**Tiempo (ms)**
small	0.21559999883174896
5 pct	1.2704000025987625
10 pct	1.417399998754263
20 pct	1.6011999994516373
30 pct	2.2956000007689
50 pct	2.4495999962091446
80 pct	2.8308999985456467
large	2.8463000021874905

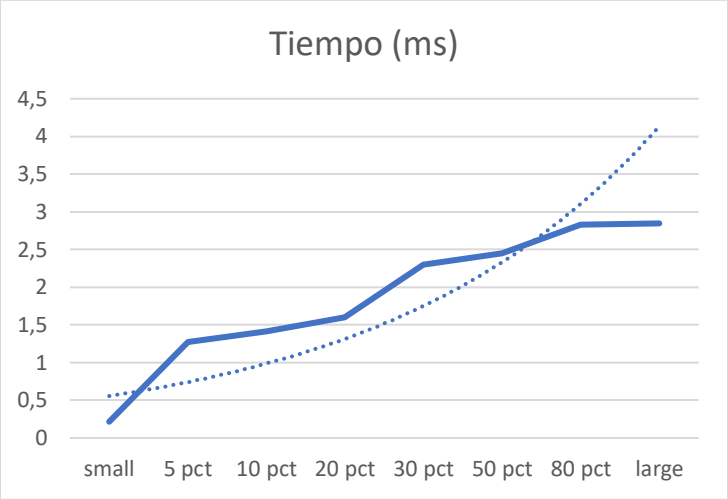
Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

Muestra	Salida	Tiempo (ms)
small	Dato1	0.21559999883174896
5 pct	Dato2	1.2704000025987625
10 pct	Dato3	1.417399998754263
20 pct	Dato4	1.6011999994516373
30 pct	Dato5	2.2956000007689
50 pct	Dato6	2.4495999962091446
80 pct	Dato7	2.8308999985456467
large	Dato8	2.8463000021874905

Graficas

Las gráficas con la representación de las pruebas realizadas.



Análisis

Se puede ver que el requerimiento, como era esperado, si cumple con un orden lineal de O(n^2). Esto se puede bien evidenciar en la gráfica y en los datos de tiempo en las tablas. Sin embargo se puede ver que la curvatura de la grafica en realidad es hacia abajo y no hacia arriba como normalmente seria en una gráfica de orden lineal de O(n^2), esto puede esta sucediendo pues a la hora de hacer un top con un numero indicado de datos en datos pequeños como small y 5 pct es muy probable que se tengan que usar todos los datos para poder completar la lista; en datos más grandes, la lista del top si se puede completar con los datos necesarios según el top indicado, esto disminuyendo el tiempo del requerimiento un poco.

Requerimiento 8

Descripción

```python
def req_8(data_structs, top, year):
    year_entry = mp.get(data_structs['map_anio'], year)
    year_data = me.getValue(year_entry)
    lista_actecon = year_data['impuestos']
    map_subsect = mp.newMap(numelements= lt.size(lista_actecon))
    sumas =['Total Impuesto a cargo',"Total ingresos netos", "Costos y gastos nómina", "Total saldo a pagar","Total saldo a favor"]
    for cadauna in lt.iterator(lista_actecon):
        subsector = int(cadauna['Código subsector económico'])
        if not mp.contains(map_subsect, subsector):
            actividades_eco = lt.newList(datastructure="ARRAY_LIST", cmpfunction= compareCodigoActividad)
            lt.addLast(actividades_eco,cadauna)
            info_subsector = {"Nombre sector económico":cadauna["Nombre sector económico"],
                              "Código sector económico":cadauna["Código sector económico"],"Código subsector económico": subsector, "Actividades económicas": actividades_eco}
            for total in sumas:
                info_subsector[total] = int(cadauna[total])
            mp.put(map_subsect, subsector, info_subsector)
        else:
            entry_info_subsector = mp.get(map_subsect,subsector)
            info_subsector = me.getValue(entry_info_subsector)
            for total in sumas:
                info_subsector[total] = int(cadauna[total]) + int(info_subsector[total])
            lt.addLast(info_subsector["Actividades económicas"], cadauna)
            mp.put(map_subsect, subsector, info_subsector)
    llaves = mp.keySet(map_subsect)
    dict_sumas = {}
    dict_act = {}
    final_final = {}
    for llave in lt.iterator(llaves):
        subsect_entry = mp.get(map_subsect, llave)
        subsect_data = me.getValue(subsect_entry)
        lista_totales = lt.newList('ARRAY_LIST')
        for titulo in sumas:
            lt.addLast(lista_totales, str(subsect_data[titulo]))
        for caract in lt.iterator(subsect_data['Actividades económicas']):
            lt.addFirst(lista_totales, caract['Nombre subsector económico'])
            lt.addFirst(lista_totales, caract['Código subsector económico'])
            lt.addFirst(lista_totales, caract['Nombre sector económico'])
            lt.addFirst(lista_totales, caract['Código sector económico'])
        dict_sumas[llave] = lista_totales
        lista_actecon_sub = subsect_data['Actividades económicas']
        dict_act[llave] = lista_actecon_sub
        merg.sort(dict_act[llave], cmp_totimpacarg)
        if int(top) > lt.size(dict_act[llave]):
            final = dict_act[llave]
            final_final[subsect_data['Código subsector económico']] = final
        else:
            final = lt.subList(dict_act[llave], 1, int(top))
            final_final[subsect_data['Código subsector económico']] = final
    return dict_sumas, final_final
def cmp_totimpacarg(data1, data2):
    return int(data1['Total Impuesto a cargo']) > int(data2['Total Impuesto a cargo'])
```

Este requerimiento se encarga de listar el top N de actividades económicas de cada subsector con los mayores totales de impuestos a cargo para un año en específico.

Entrada	El map principal, el año y top que son elegidos por el usuario.
Salidas	1. Todos los subsectores con las respectivas sumas 2. Tablas por cada uno de los subsectores del top N de actividades económicas con más “peso” en el total de impuestos a cargo
Implementado (Sí/No)	Sí, por Gabriela Escobar

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**

`year_entry = mp.get(data_structs['map_anio'], year)`
`year_datos = me.getValue(year_entry)`
`lista_actecon = year_datos['impuestos']`
	O(N)

`for cadauna in lt.iterator(lista_actecon):`
`  subsector = int(cadauna['Código subsector económico'])`
`  if not mp.contains(map_subsect, subsector):`
`  actividades_eco = lt.newList(datastructure="ARRAY_LIST", cmpfunction= compareCodigoActividad)`
`  lt.addLast(actividades_eco,cadauna)`
`  info_subsector = {"Nombre sector económico":cadauna["Nombre sector económico"],`
`  "Código sector económico":cadauna["Código sector económico"],"Código subsector económico": subsector, "Actividades económicas": actividades_eco}`
`  for total in sumas:`
`  info_subsector[total] = int(cadauna[total])`
`  mp.put(map_subsect, subsector, info_subsector)`
`  else:`
`  entry_info_subsector = mp.get(map_subsect,subsector)`
`  info_subsector = me.getValue(entry_info_subsector)`
`  for total in sumas:`
`  info_subsector[total] = int(cadauna[total]) + int(info_subsector[total])`
`  lt.addLast(info_subsector["Actividades económicas"], cadauna)`
`  mp.put(map_subsect, subsector, info_subsector)`
	O(N ^2)

`lista_sqmepiden = [[[x[sumas_q_me_piden[0]], x[sumas_q_me_piden[1]], x[sumas_q_me_piden[2]], x[sumas_q_me_piden[3]], x[sumas_q_me_piden[4]], x[sumas_q_me_piden[5]], x[sumas_q_me_piden[6]], x[sumas_q_me_piden[7]], x[sumas_q_me_piden[8]]]`
`for x in por_subsector[subsector]['elements']]`
`for subsector in por_subsector]`
	O(N^2)

`merg.sort(dict_act[llave], cmp_totimpacarg)`	O(NlogN)
`lt.addLast(lista_totales, str(subsect_data[titulo]))`	O (1)

`lt.addFirst(lista_totales, caract['Nombre subsector económico'])`
`lt.addFirst(lista_totales, caract['Código subsector económico'])`
`lt.addFirst(lista_totales, caract['Nombre sector económico'])`
`lt.addFirst(lista_totales, caract['Código sector económico'])`
	O (4N)
	O (N)

TOTAL	***O (N^2)***

Pruebas Realizadas

Las pruebas realizadas fueron realizadas en una maquina con las siguientes especificaciones. Los datos de entrada fueron para el año 2021, con top 3. El programa se probó utilizando el esquema de colisión chaining, con el factor de carga de 8.

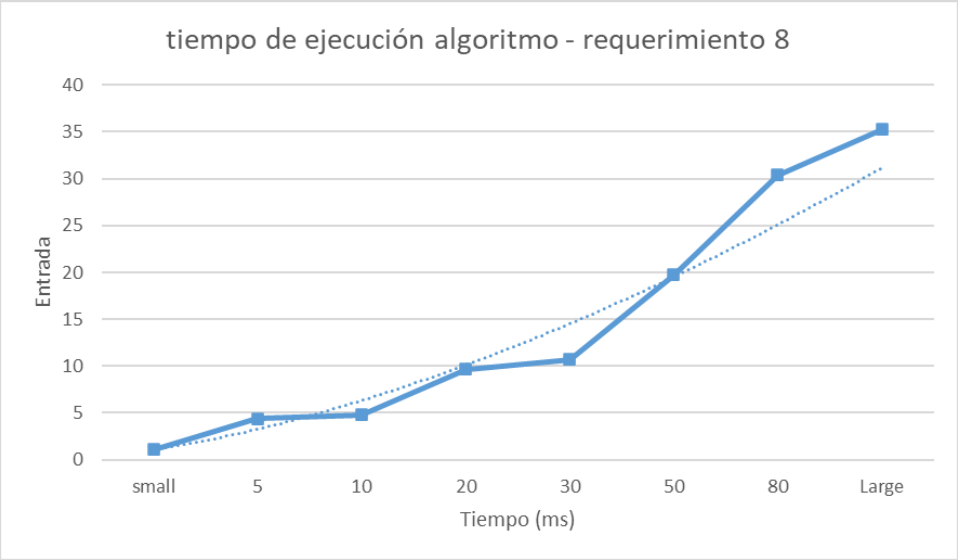
Procesador	**11th Gen Intel(R) Core(TM) i7-1195G7 @ 2.90GHz 2.92 GHz**
Memoria RAM	32GB
Sistema Operativo	Windows 11 Pro- 64 bits

Entrada	Tiempo (ms)
small	1,07
5 pct	4,36
10 pct	4,80
20 pct	9,68
30 pct	10,71
50 pct	19,72
80 pct	30,41
large	35,28

Tablas de datos

Muestra	Salida	Tiempo (ms)
small	Dato1	1,07
5 pct	Dato2	4,36
10 pct	Dato3	4,80
20 pct	Dato4	9,68
30 pct	Dato5	10,71
50 pct	Dato6	19,72
80 pct	Dato7	30,41
large	Dato8	35,28

Gráfica



Análisis

Algunos errores que se pueden considerar significativos en la toma de datos pueden ser que el computador tenía otros programas abiertos durante la toma de datos, y, además, que el probar el programa tantas veces seguidas aumentaba el tiempo real de la prueba. Tomando en cuenta lo anterior, se puede notar que se afirma el análisis de complejidad temporal, pues hay un crecimiento cuadrático. Este resultado se puede ver en la línea de tendencia expuesta en la misma gráfica. Esta complejidad temporal se debe en su totalidad al doble iteraciones, una dentro de otra. Aunque esta complejidad podría ser más eficiente, se puede ver un avance, en comparación a requerimientos de similar complejidad, del reto pasado. Esto demuestra que las habilidades aprendidas este nivel fueron empleadas para mejorar en términos de eficiencia.