ANÁLISIS DEL RETO 2 – GRUPO 04

Integrantes

Sergio Cañar, 202020383, s.canar
Juan Martín Vásquez, 202113214, j.vasquezc
Juan Bernardo Parra, 202021772, j.parrah

Carga de Datos

Para comparar las diferencias de velocidad y consumo de memoria entre linear probing y separate chaining, se realizó una medición de tiempo y memoria para la carga de datos en una máquina con las siguientes características:

Procesador	AMD Ryzen 7 3700U with Radeon Vega Mobile Gfx 2.30 GHz
Memoria RAM (GB)	8.00 GB (6.94 GB usable)
Sistema Operativo	Windows 11 Home Single Language – 64 bits

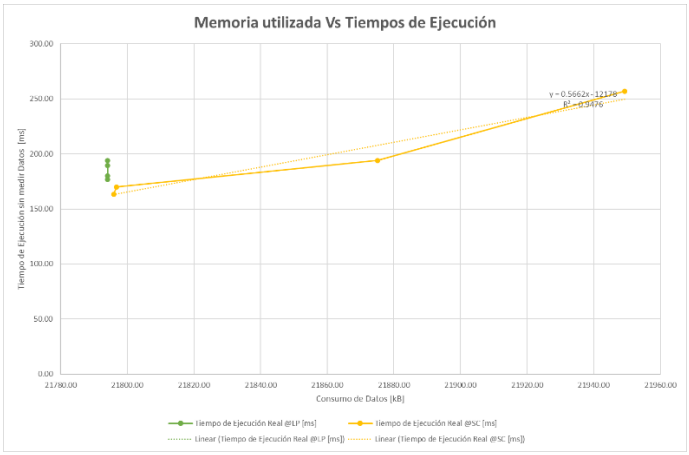
Los resultados de las mediciones son los siguientes:

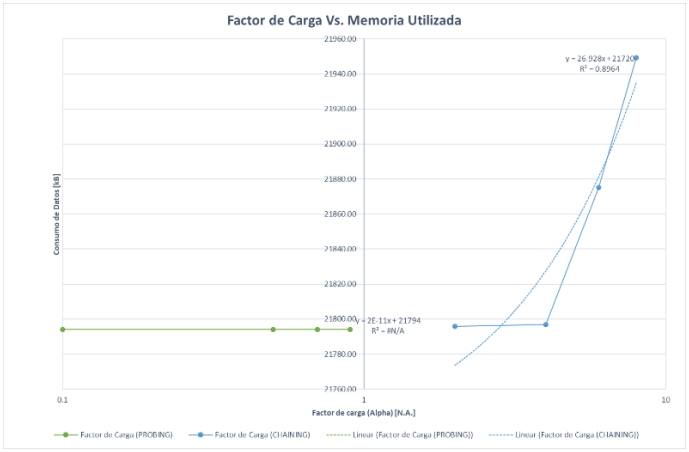
Carga de Catálogo PROBING

Factor de Carga (PROBING)	**Consumo de Datos [kB]**	**Tiempo de Ejecución Real @LP [ms]**
0.1	21794.15	197.52
0.5	21794.15	232.38
0.7	21794.15	295.68
0.9	21794.15	316.87

Carga de Catálogo CHAINING

Factor de Carga (CHAINING)	**Consumo de Datos [kB]**	**Tiempo de Ejecución Real @SC [ms]**
2.00	21795.93	163.39
4.00	21796.81	169.97
6.00	21875.15	193.97
8.00	21949.34	257.06


Memoria utilizada Vs Tiempos de Ejecución
Tiempo de Ejecución sin medir Datos [ms]
300.00
250.00
200.00
150.00
100.00
50.00
0.00
21780.00
21800.00
21820.00
21840.00
21860.00
21880.00
21900.00
21920.00
21940.00
21960.00
Consumo de Datos [kB]
y = 0.5662x - 12178
R² = 0.5476
Tiempo de Ejecución Real @LP [ms]
Tiempo de Ejecución Real @SC [ms]
Linear (Tiempo de Ejecución Real @LP [ms])
Linear (Tiempo de Ejecución Real @SC [ms])


Factor de Carga Vs. Memoria Utilizada
Consumo de Datos [kB]
21960.00
21940.00
21920.00
21900.00
21880.00
21860.00
21840.00
21820.00
21800.00
21780.00
21760.00
0.1
1
10
Factor de carga (Alpha) [N.A.]
y = 26.928x + 21720
R² = 0.8964
y = 2E-11x + 21794
R² = #N/A
Factor de Carga (PROBING)
Factor de Carga (CHAINING)
Linear (Factor de Carga (PROBING))
Linear (Factor de Carga (CHAINING))

Requerimiento 1

Descripción

```python
def req_1(data_structs,yearIn, codeEcoIn):
    """
    Función que soluciona el requerimiento 1
    """
    # TODO: Realizar el requerimiento 1

    impuestosSorted = sortTaxbyYearSAP(data_structs, yearIn)
    listaImpustosCodigoDeseado = lt.newList('ARRAY_LIST')
    respuesta = lt.newList('ARRAY_LIST')
    for impuesto in lt.iterator(impuestosSorted):
        if codeEcoIn == int(impuesto['Código sector económico']):
            lt.addLast(listaImpustosCodigoDeseado, impuesto)
    mayordeseado = lt.getElement(listaImpustosCodigoDeseado, 1)
    lt.addLast(respuesta, mayordeseado)
    return respuesta
```

```python
def sortTaxbyYearSAP(data_structs, yearIn):

    '''
    Funcion que retorna una lista de impuestos ordenados por año y por saldo a pagar
    '''
    mapa_year = mp.get(data_structs['anio'], yearIn)

    if mapa_year:
        year_taxes = me.getValue(mapa_year)['impuesto']
        lista_ordenada = useMergeSort(year_taxes, compareSaldoAPagar)

    return lista_ordenada
```

```python
def compareSaldoAPagar(impuesto1, impuesto2):
    '''
    Funcion que retorna True si el saldo a pagar de la actividad economica de un impuesto1 es menor que el del impuesto2.

    Ademas verifica si dicho valor es numerico.
    '''
    if impuesto1['Total saldo a pagar'].isnumeric() and impuesto2['Total saldo a pagar'].isnumeric():
        return int(impuesto1['Total saldo a pagar']) > int(impuesto2['Total saldo a pagar'])
```

El requerimiento 1 se encarga de obtener la actividad económica con mayor saldo a pagar dados un sector económico y un año que son recibidos como parámetro. Este empieza por filtrar la estructura de datos según el año dado y la ordena de mayor a menor usando la función compareSaldoAPagar. Luego crea una lista auxiliar llamada listaImpuestosCodigoDeseado, en la cual filtra la lista ordenada por el código dado por el usuario. Debido a que la lista está ordenada de mayor a menor, basta con extraer el primer elemento de esta lista para obtener la respuesta.

Entrada	Estructura de datos del modelo, año y sector económico deseados
Salidas	La actividad económica con mayor saldo a pagar, el tiempo en ms y la memoria en kbs.
Implementado (Sí/No)	Si. Implementado por el grupo 04 de EDA

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

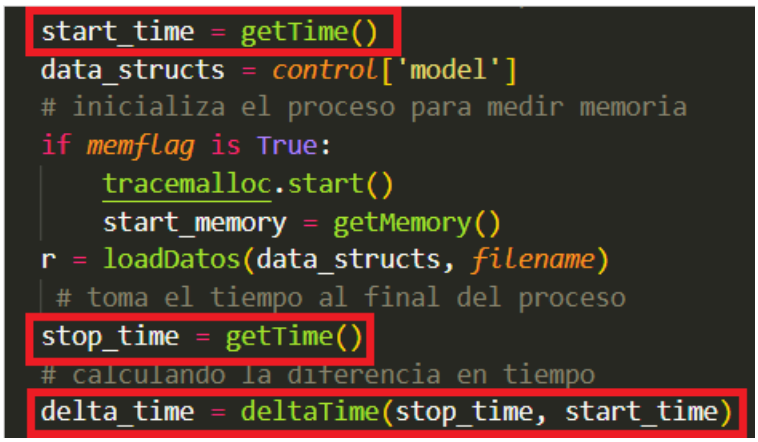
Pasos	**Complejidad**
Ordenar por saldo a pagar en el año dado (Merge)	O ($n \log n$)
Iterar sobre la lista ordenada y usar addLast	O(n)
Obtener el mayor con getElement	O (1)
Añadir el mayor a la lista respuesta	O(1)
TOTAL	***O(n log n)***

Pruebas Realizadas

Las pruebas se realizaron en un computador con las siguientes características:

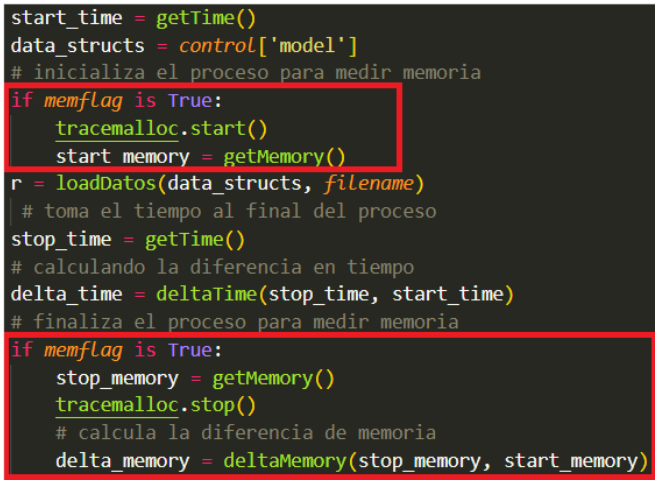
Procesador	Intel(R) Core(TM) i7-1065G7 CPU @ 1.30GHz 1.50 GHz
Memoria RAM (GB)	12.0 GB (11.7 GB utilizable)
Sistema Operativo	Windows 11 Home Single Language – 64 bits

Para la medición de tiempos se usaron las funciones getTime y deltaTime de la librería time. Estas se ubicaron en el controller para medir el tiempo de ejecución de los requerimientos de la siguiente manera:



```
start_time = getTime()
data_structs = control['model']
# inicializa el proceso para medir memoria
if memflag is True:
    tracemalloc.start()
    start_memory = getMemory()
r = loadDatos(data_structs, filename)
# toma el tiempo al final del proceso
stop_time = getTime()
# calculando la diferencia en tiempo
delta_time = deltaTime(stop_time, start_time)
```

Por otro lado, se usó la librería tracemalloc para rastrear el consumo de memoria. Se usaron las funciones de tracemalloc.start, getMemory, deltaMemory y tracemalloc.stop para el rastreo de memoria. Hay que destacar que este rastreo solo se ejecuta cuando el usuario indica que se haga, por lo que se implementó una variable booleana (llamada memflag) para confirmar esta operación, como se puede ver en la imagen:



```
start_time = getTime()
data_structs = control['model']
# inicializa el proceso para medir memoria
if memflag is True:
    tracemalloc.start()
    start_memory = getMemory()
r = loadDatos(data_structs, filename)
# toma el tiempo al final del proceso
stop_time = getTime()
# calculando la diferencia en tiempo
delta_time = deltaTime(stop_time, start_time)
# finaliza el proceso para medir memoria
if memflag is True:
    stop_memory = getMemory()
    tracemalloc.stop()
    # calcula la diferencia de memoria
    delta_memory = deltaMemory(stop_memory, start_memory)
```

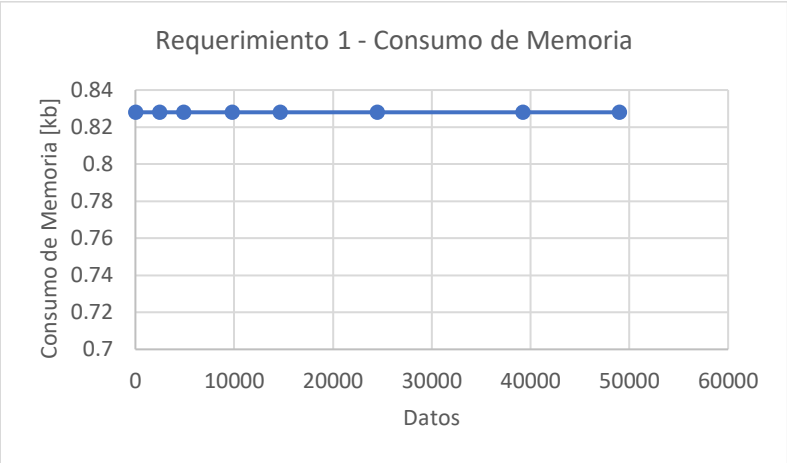
En este ejemplo específico, se hace la medición de memoria y tiempo sobre la función loadDatos, cuyo resultado se almacena en la variable r.

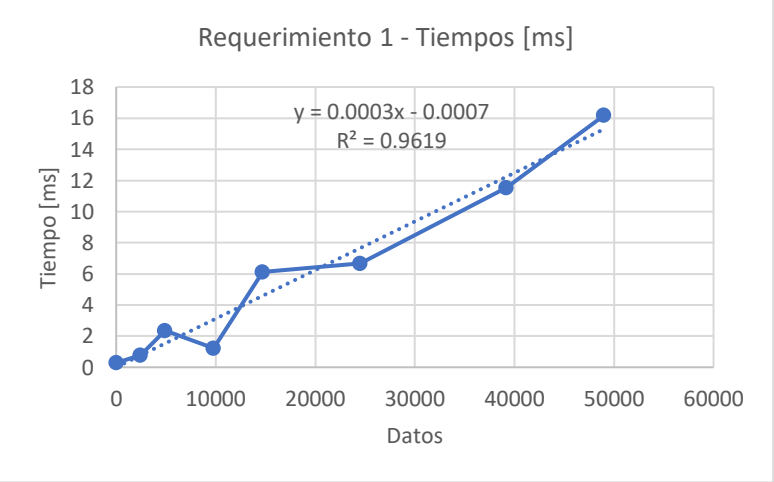
Para las mediciones del requerimiento 1, se ingresaron como parámetros el año 2021 y el subsector 11. Se usó el método de linear probing con un factor de carga de 0.5.

Tablas de datos

Datos	Consumo de Datos [kB]	Tiempo de Ejecución Real @LP [ms]
49	0.828	0.29
2450	0.828	0.784
4900	0.828	2.37
9800	0.828	1.21
14700	0.828	6.11
24510	0.828	6.66
39220	0.828	11.55
49030	0.828	16.19

Graficas





Análisis

En este caso se evidencia cómo los tiempos de ejecución del algoritmo muestran una tendencia lineal por lo que es posible que el algoritmo se comporte basado en las iteraciones realizadas en las listas ordenadas. Sin embargo, se observa una diferencia notoria en el tiempo de ejecución a medida que incrementan la cantidad de datos por lo que es posible en la práctica la complejidad algorítmica del código implementado difiera con el análisis teórico realizado previamente. Por otro lado, se observa que no hubo un cambio en la memoria utilizada durante la ejecución del requerimiento, principalmente esto se deba a que el ordenamiento de los datos no supone una gran diferencia entre los porcentajes de datos cargados. Lo anterior, sucede debido a que si bien incrementa la cantidad de datos totales, cuando se realiza la búsqueda sobre el mapa de años-subsectores, al momento de buscar un sector económico especifico la cantidad de datos que cumplan ambas condiciones no difieran en grandes cantidades entre cada porcentaje de datos.

Requerimiento 2

Descripción

```python
def req_2(data_structs, yearIn, codeEcoIn):
    """
    Función que soluciona el requerimiento 2
    """
    # TODO: Realizar el requerimiento 2

    impuestosSorted = sortTaxbyYearSAF(data_structs, yearIn)
    listaImpustosCodigoDeseado = lt.newList('ARRAY_LIST')
    respuesta = lt.newList('ARRAY_LIST')
    for impuesto in lt.iterator(impuestosSorted):
        if codeEcoIn == int(impuesto['Código sector económico']):
            lt.addLast(listaImpustosCodigoDeseado, impuesto)
    mayordeseado = lt.getElement(listaImpustosCodigoDeseado, 1)
    lt.addLast(respuesta, mayordeseado)
    return respuesta
```

```python
def compareSaldoAFavor(impuesto1, impuesto2):
    '''
    Funcion que retorna True si el saldo a pagar de la actividad economica de un impuesto1 es menor que el del impuesto2.

    Ademas verifica si dicho valor es numerico.
    '''
    if impuesto1['Total saldo a favor'].isnumeric() and impuesto2['Total saldo a favor'].isnumeric():
        return int(impuesto1['Total saldo a favor']) > int(impuesto2['Total saldo a favor'])

def sortTaxbyYearSAF(data_structs, yearIn):

    '''
    Funcion que retorna una lista de impuestos ordenados por año y por saldo a favor
    '''
    mapa_year = mp.get(data_structs['anio'], yearIn)

    if mapa_year:
        year_taxes = me.getValue(mapa_year)['impuesto']
        lista_ordenada = useMergeSort(year_taxes, compareSaldoAFavor)

    return lista_ordenada
```

El requerimiento 2 busca la actividad económica con mayor saldo a favor dados un sector económico y un año. Este funciona de manera similar al requerimiento 1, pues también filtra la estructura de datos según el año dado y la ordena de mayor a menor usando esta vez la función *compareSaldoAFavor*. Luego crea una lista auxiliar llamada *listaImpuestosCodigoDeseado*, en la cual filtra la lista ordenada por el código dado por el usuario. De la misma manera que en el requerimiento 1, basta con extraer el primer elemento de esta lista para obtener la respuesta puesto que la lista ya está ordenada.

Entrada	Estructura de datos del modelo, año y sector económico deseados
Salidas	La actividad económica con mayor saldo a favor, el tiempo en ms y la memoria en kbs.
Implementado (Sí/No)	Si. Por el grupo 04 de EDA

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Ordenar por saldo a favor en el año dado (Merge)	O $(n \log n)$
Iterar sobre la lista ordenada y usar addLast	O(n)
Obtener el mayor con getElement	O (1)
Añadir el mayor a la lista respuesta	O(1)
TOTAL	***O(n log n)***

Pruebas Realizadas

Las pruebas se realizaron en un computador con las siguientes características:

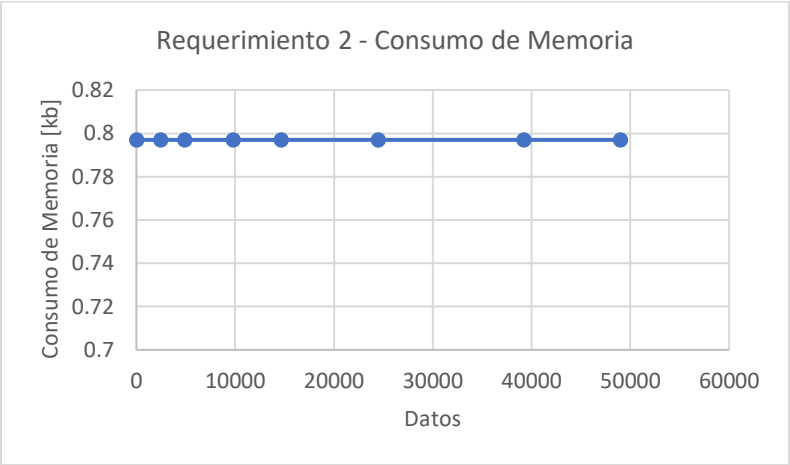
Procesador	Intel(R) Core(TM) i7-1065G7 CPU @ 1.30GHz 1.50 GHz
Memoria RAM (GB)	12.0 GB (11.7 GB utilizable)
Sistema Operativo	Windows 11 Home Single Language – 64 bits

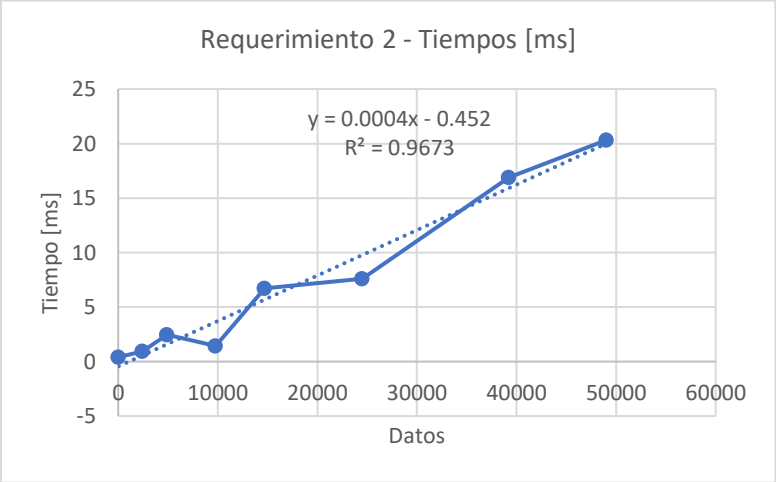
Se realizó el mismo procedimiento descrito en el requerimiento 1 para la medición de tiempo y consumo de memoria. Para las mediciones del requerimiento 2, se ingresaron como parámetros el año 2021 y el subsector 11. Se usó el método de linear probing con un factor de carga de 0.5.

Tablas de datos

Datos	**Consumo de Datos [kB]**	**Tiempo de Ejecución Real @LP [ms]**
49	0.797	0.4
2450	0.797	0.931
4900	0.797	2.45
9800	0.797	1.42
14700	0.797	6.7
24510	0.797	7.6
39220	0.797	16.91
49030	0.797	20.32

Graficas





Análisis

Similar a lo que ocurre con el requerimiento 1, se observa que a medida de que incrementan la cantidad de datos hay un comportamiento lineal del algoritmo, difiriendo del análisis teórico realizado con anterioridad. Esto, puede ocurrir debido a que el algoritmo puede variar en la cantidad de iteraciones realizadas sobre la cantidad de elementos de un año específico y un sector económico específico. En términos de la memoria, hay un comportamiento similar debido a que como se explicó previamente puede que la cantidad de datos que cumplan las condiciones ingresadas por el usuario no difieran significativamente para que haya una diferencia en la memoria empleada para la ejecución del requerimiento

Requerimiento 3 (Sergio Cañar)

Descripción

```
def req_3(data_structs, yearIn, headers):
    '''
    Funcion que retorna una lista de impuestos ordenados por año y por saldo a favor
    '''
    mapa = data_structs['anioSUBSEC']
    llaves = mp.keySet(mapa)
    listaSubSectores = lt.newList('ARRAY_LIST')
    for llave in lt.iterator(llaves):
        if (yearIn in llave) == True:
            impuesto = mp.get(mapa, llave)
            lt.addLast(listaSubSectores, impuesto)
    #Acumulaciones
    lista_r1 = lt.newList('ARRAY_LIST')
    pos_min = 1
    pos = 1
    min_ = 1000000000000000000
    for tax in lt.iterator(listaSubSectores):
        mapaTemporal = mp.newMap(10, maptype='PROBING', loadfactor=0.7)
        for fila in headers:
            mp.put(mapaTemporal, fila, 0)
        for data in lt.iterator(tax['value']['impuesto']):
            if me.getValue(mp.get(mapaTemporal, 'Código sector económico')) == 0:
                mp.put(mapaTemporal,'Código sector económico',data['Código sector económico'])
                mp.put(mapaTemporal,'Nombre sector económico',data['Nombre sector económico'])
                mp.put(mapaTemporal,'Código subsector económico',data['Código subsector económico'])
                mp.put(mapaTemporal,'Nombre subsector económico',data['Nombre subsector económico'])
            mp.put(mapaTemporal,'Total retenciones',me.getValue(mp.get(mapaTemporal,'Total retenciones'))+int(data['Total retenciones']))
            mp.put(mapaTemporal,'Total ingresos netos',me.getValue(mp.get(mapaTemporal,'Total ingresos netos'))+int(data['Total ingresos netos']))
            mp.put(mapaTemporal,'Total costos y gastos',me.getValue(mp.get(mapaTemporal,'Total costos y gastos'))+int(data['Total costos y gastos']))
            mp.put(mapaTemporal,'Total saldo a pagar',me.getValue(mp.get(mapaTemporal,'Total saldo a pagar'))+int(data['Total saldo a pagar']))
            mp.put(mapaTemporal,'Total saldo a favor',me.getValue(mp.get(mapaTemporal,'Total saldo a favor'))+int(data['Total saldo a favor']))
        lt.addLast(lista_r1, mapaTemporal)
        if min_ > me.getValue(mp.get(mapaTemporal, 'Total retenciones')):
            min_ = me.getValue(mp.get(mapaTemporal, 'Total retenciones'))
            pos_min = pos
        pos += 1

    respuesta1 = lt.getElement(lista_r1, pos_min)
    respuesta2 = listaFromMapReq3(respuesta1)

    codeMin = me.getValue(mp.get(respuesta1, 'Código subsector económico'))
    listaActividades = (me.getValue(mp.get(mapa,yearIn+","+str(codeMin))))['impuesto']
    listaActividadesSorted = useMergeSort(listaActividades, compareRetencionTotal)
    flag = None

    if lt.size(listaActividadesSorted) > 6:
        listaActividadesSorted = recortarLista(listaActividadesSorted)
        flag = True
    else:
        listaActividadesSorted = listaActividades
        flag = False

    return respuesta2, flag, listaActividadesSorted, codeMin
```

```
def listaFromMapReq3(mapa):
    respuesta2 = lt.newList('ARRAY_LIST')
    llaves = mp.keySet(mapa)
    for llave in lt.iterator(llaves):
        if llave != None:
            valor = (llave, me.getValue(mp.get(mapa, llave)))
            lt.addLast(respuesta2, valor)
    return respuesta2

def compareRetencionTotal(impuesto1, impuesto2):
    '''
    Funcion que retorna True si la retencion total de la actividad economica de un impuesto1 es menor que el del impuesto2.

    Ademas verifica si dicho valor es numerico.
    '''
    return int(impuesto1['Total retenciones']) < int(impuesto2['Total retenciones'])
```

Con el requerimiento 3 se desea encontrar, para un año dado, el subsector económico con las menores retenciones totales. Posteriormente, se procede a buscar la posición en el mapa del subsector con el mínimo total de retenciones comparándolo que un mínimo inicial (cuyo valor es muy grande). Si las retenciones son menores que el mínimo, este se actualiza y almacena las retenciones actuales como el mínimo. Una vez se obtiene la posición, se obtiene este elemento mediante la función getElement y su resultado se almacena en la variable respuesta1. La variable respuesta2 corresponde a la respuesta 1 modificada para que su estructura sea una lista, con el fin de facilitar su impresión. Para esto se usa la función listafromMapReq3. El código con el subsector obtenido anteriormente se almacena en codeMin, la cual se usa posteriormente para obtener la lista de actividades que corresponden al conjunto año y codeMin. Esta lista de actividades se ordena usando MergeSort. La función retorna el mapa convertido en lista, un indicador que muestra si hay menos de 6 elementos en la lista de actividades, la lista de actividades ordenadas y el código del subsector con menores retenciones para el año indicado.

Entrada	El mapa con los datos, un año específico y las categorías a imprimir
Salidas	Lista con las actividades del subsector con menores retenciones en el año ingresado, el tiempo en ms y la memoria en kbs.
Implementado (Sí/No)	Si, Sergio Cañar

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Recorrer la variable *llave*	O(n)
Recorrer la variable *listaSubSectores*	O(n)
Recorrer la información de cada subsector	O(k) donde k son el número de actividades del subsector
Ordenar la lista de actividades mediante Merge	O(n log n)
TOTAL	***O(n log n)***

Pruebas Realizadas

Las pruebas se realizaron en un computador con las siguientes características:

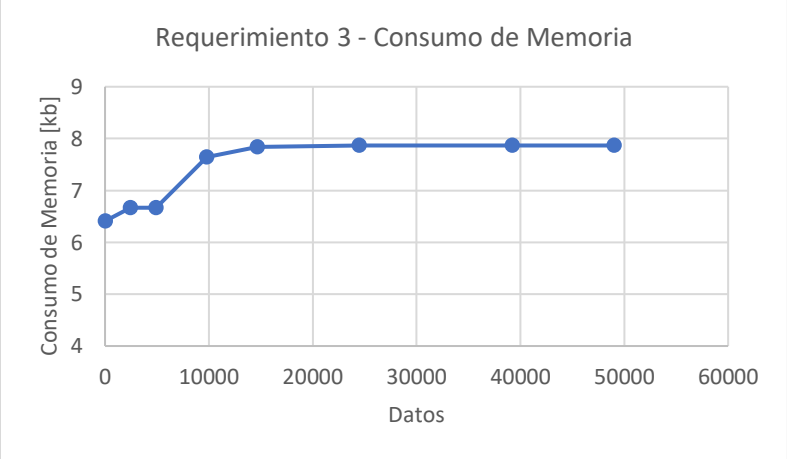
Procesador	Intel(R) Core(TM) i7-1065G7 CPU @ 1.30GHz 1.50 GHz
Memoria RAM (GB)	12.0 GB (11.7 GB utilizable)
Sistema Operativo	Windows 11 Home Single Language – 64 bits

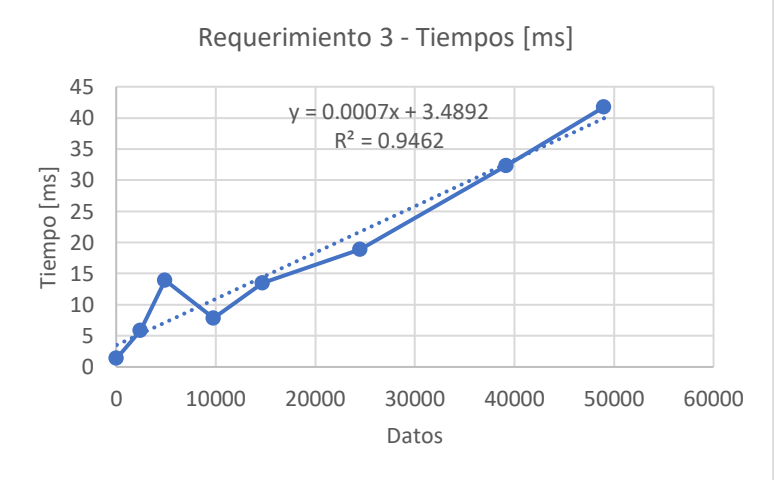
Se realizó el mismo procedimiento descrito en el requerimiento 1 para la medición de tiempo y consumo de memoria. Para las mediciones del requerimiento 3, se ingresó como parámetros el año 2021. Se usó el método de linear probing con un factor de carga de 0.5.

Tablas de datos

Datos	Consumo de Datos [kB]	Tiempo de Ejecución Real @LP [ms]
49	6.414	1.39
2450	6.672	5.88
4900	6.672	13.89
9800	7.648	7.83
14700	7.844	13.48
24510	7.867	18.9
39220	7.867	32.37
49030	7.867	41.77

Graficas





Análisis

Basándose en los resultados de las pruebas realizadas y en el análisis de complejidad temporal teórico realizado se puede apreciar que el tiempo de ejecución del algoritmo presentó un comportamiento lineal

lo cual no era esperado teniendo en cuenta el tipo de ordenamiento empleado para el manejo de los natos (mergeSort). Por otro lado, en términos de memoria se aprecia que hubo un comportamiento esperado dado que fue aumentando considerablemente a medida de que la cantidad de datos fue incrementando.

Requerimiento 4 (Martín Vásquez)

Descripción

```
def req_4(data_structs,anio,headers):
    """
    Función que soluciona el requerimiento 4
    """
    map = data_structs['anioSUBSEC']
    llaves  = mp.keySet(map)
    listaSS = lt.newList('ARRAY_LIST')
    for i in lt.iterator(llaves):
        if (anio in i) == True:
            imp = mp.get(map,i)
            lt.addLast(listaSS, imp)
    #acumlaciones
    '''
    headers=['Código sector económico','Nombre sector económico','Código subsector económico',
                    'Nombre subsector económico','Costos y gastos nómina','Total ingresos netos',
                    'Total costos y gastos','Total saldo a pagar','Total saldo a favor']
    '''
    #mapa
    r1 = lt.newList('ARRAY_LIST')
    pos_max = 1
    pos = 1
    max_ = 0
    for i in lt.iterator(listaSS):
        mapA = mp.newMap(10,maptype='PROBING',loadfactor=0.7)
        for j in headers:
            mp.put(mapA,j,0)
        for k in lt.iterator(i['value']['impuesto']):
            if me.getValue(mp.get(mapA,'Código subsector económico'))==0:
                mp.put(mapA,'Código sector económico',k['Código sector económico'])
                mp.put(mapA,'Nombre sector económico',k['Nombre sector económico'])
                mp.put(mapA,'Código subsector económico',k['Código subsector económico'])
                mp.put(mapA,'Nombre subsector económico',k['Nombre subsector económico'])
            mp.put(mapA,'Costos y gastos nómina',me.getValue(mp.get(mapA,'Costos y gastos nómina'))+int(k['Costos y gastos nómina']))
            mp.put(mapA,'Total ingresos netos',me.getValue(mp.get(mapA,'Total ingresos netos'))+int(k['Total ingresos netos']))
            mp.put(mapA,'Total costos y gastos',me.getValue(mp.get(mapA,'Total costos y gastos'))+int(k['Total costos y gastos']))
            mp.put(mapA,'Total saldo a pagar',me.getValue(mp.get(mapA,'Total saldo a pagar'))+int(k['Total saldo a pagar']))
            mp.put(mapA,'Total saldo a favor',me.getValue(mp.get(mapA,'Total saldo a favor'))+int(k['Total saldo a favor']))
        lt.addLast(r1,mapA)
        if max_ < me.getValue(mp.get(mapA,'Costos y gastos nómina')):
                max_ = me.getValue(mp.get(mapA,'Costos y gastos nómina'))
                pos_max = pos
        pos += 1
        rta1 = lt.getElement(r1, pos_max)
        rta2 = ListaFromMap(rta1)
    codeMAX =  me.getValue(mp.get(rta1,'Código subsector económico'))
    listaActs =  (me.getValue(mp.get(map,anio+","+str(codeMAX))))['impuesto']
    listaActsOrd = useMerge(listaActs, cmp_req_4)
    flag = None
    if lt.size(listaActsOrd) > 6:
        listaActsOrdRect = recortarLista(listaActsOrd)
        flag = True
    else:
        listaActsOrdRect = listaActsOrd
        flag = False
    return rta2, flag, listaActsOrdRect, codeMAX
```

```
def ListaFromMap(map):
    f = lt.newList('ARRAY_LIST')
    keys = mp.keySet(map)
    for i in lt.iterator(keys):
        if i != None:
            x = (i, me.getValue(mp.get(map,i)))
            lt.addLast(f,x)
    return f
```

La solución del requerimiento 4 es similar al proceso del requerimiento 3, en el sentido de que se crean las mismas variables y se usan las mismas estructuras de datos. Sin embargo, por las características del requerimiento es necesario cambiar algunos detalles. Debido a que ahora se busca el subsector con mayores costos y gastos de nómina en un año específico, se debe cambiar los headers para incluir esta característica del subsector al problema y se debe cambiar la información que se añade al mapa auxiliar. Además, se pasa de buscar un mínimo a un máximo, por lo que el signo en las comparaciones cambia.

Entrada	El mapa con los datos, un año específico y las categorías a imprimir
Salidas	Lista con las actividades del subsector con menores retenciones en el año ingresado, el tiempo en ms y la memoria en kbs.
Implementado (Sí/No)	Si, Martín Vásquez

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Recorrer la variable *llave*	O(n)
Recorrer la variable *listaSS*	O(n)
Recorrer la información de cada subsector	O(k) donde k son el número de actividades del subsector
Ordenar la lista de actividades mediante Merge	O(n log n)
TOTAL	***O(n log n)***

Pruebas Realizadas

Las pruebas se realizaron en un computador con las siguientes características:

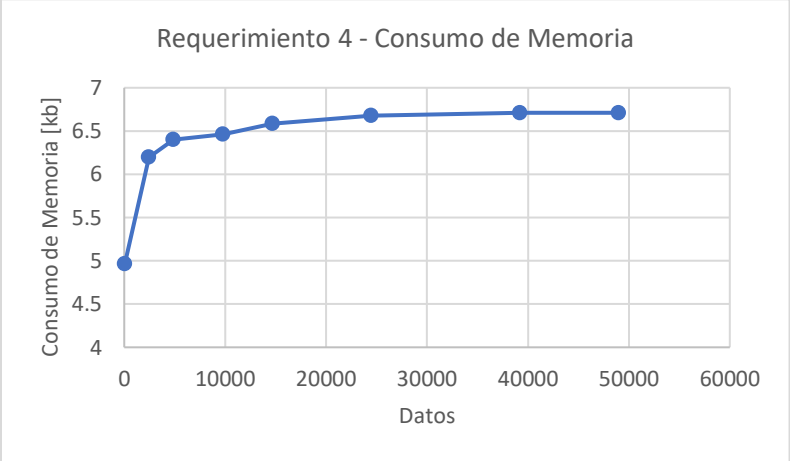
Procesador	Intel(R) Core(TM) i7-1065G7 CPU @ 1.30GHz 1.50 GHz
Memoria RAM (GB)	12.0 GB (11.7 GB utilizable)
Sistema Operativo	Windows 11 Home Single Language – 64 bits

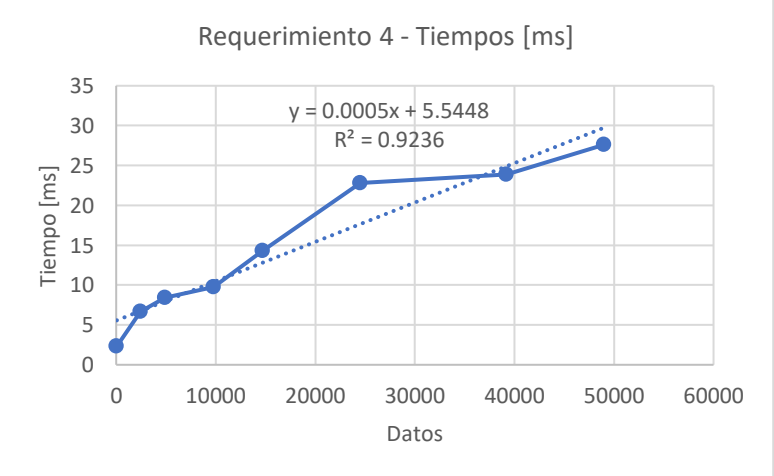
Se realizó el mismo procedimiento descrito en el requerimiento 1 para la medición de tiempo y consumo de memoria. Para las mediciones del requerimiento 4, se ingresó como parámetro el año 2021. Se usó el método de linear probing con un factor de carga de 0.5.

Tablas de datos

Datos	Consumo de Datos [kB]	Tiempo de Ejecución Real @LP [ms]
49	4.961	2.306
2450	6.195	6.665
4900	6.398	8.39
9800	6.461	9.77
14700	6.586	14.34
24510	6.678	22.8
39220	6.711	23.9
49030	6.711	27.61

Graficas





Análisis

A partir de los resultados de las pruebas realizadas para el requerimiento 5, se puede apreciar en la gráfica de los tiempos de ejecución con respecto a la cantidad de datos que tiene un comportamiento de nlog(n) dado que dicha curva se asemeja a la curva teórica. Lo anterior, se debe a que para solucionar este

requerimiento se emplear el tipo de ordenamiento MergeSort que tiene una complejidad temporal de O(nlog(n)). Adicionalmente, en términos de memoria se logra observar que esta incrementa a medida que la cantidad de datos aumenta, lo cual era completamente esperado.

Requerimiento 5 (Juan Bernardo Parra)

Descripción

```
def req_5(data_structs, anio, headers):
    """
    Función que soluciona el requerimiento 5
    """
    # TODO: Realizar el requerimiento 5
    #lista con los subsectores de un anio
    map = data_structs['anioSUBSEC']
    llaves  = mp.keySet(map)
    listaSS = lt.newList('ARRAY_LIST')
    for i in lt.iterator(llaves):
        if (anio in i) == True:
            imp = mp.get(map,i)
            lt.addLast(listaSS, imp)
    #mapa
    r1 = lt.newList('ARRAY_LIST')
    pos_max = 1
    pos = 1
    max_ = 0
    for i in lt.iterator(listaSS):
        mapA = mp.newMap(10,maptype='PROBING',Loadfactor=0.7)
        for j in headers:
            mp.put(mapA,j,0)
        for k in lt.iterator(i['value']['impuesto']):
            if me.getValue(mp.get(mapA,'Código subsector económico'))==0:
                mp.put(mapA,'Código sector económico',k['Código sector económico'])
                mp.put(mapA,'Nombre sector económico',k['Nombre sector económico'])
                mp.put(mapA,'Código subsector económico',k['Código subsector económico'])
                mp.put(mapA,'Nombre subsector económico',k['Nombre subsector económico'])
            mp.put(mapA,'Descuentos tributarios',me.getValue(mp.get(mapA,'Descuentos tributarios'))+int(k['Descuentos tributarios']))
            mp.put(mapA,'Total ingresos netos',me.getValue(mp.get(mapA,'Total ingresos netos'))+int(k['Total ingresos netos']))
            mp.put(mapA,'Total costos y gastos',me.getValue(mp.get(mapA,'Total costos y gastos'))+int(k['Total costos y gastos']))
            mp.put(mapA,'Total saldo a pagar',me.getValue(mp.get(mapA,'Total saldo a pagar'))+int(k['Total saldo a pagar']))
            mp.put(mapA,'Total saldo a favor',me.getValue(mp.get(mapA,'Total saldo a favor'))+int(k['Total saldo a favor']))
        lt.addLast(r1,mapA)
        if max_ < me.getValue(mp.get(mapA,'Descuentos tributarios')):
                max_ = me.getValue(mp.get(mapA,'Descuentos tributarios'))
                pos_max = pos
        pos += 1
        rta1 = lt.getElement(r1, pos_max)
        rta2 = ListaFromMap(rta1)
    codeMAX =  me.getValue(mp.get(rta1,'Código subsector económico'))
    listaActs =  (me.getValue(mp.get(map,anio+","+str(codeMAX))))['impuesto']
    listaActsOrd = useMerge(listaActs, cmp_req_5)
    flag = None
    if lt.size(listaActsOrd) > 6:
        listaActsOrdRect = recortarLista(listaActsOrd)
        flag = True
    else:
        listaActsOrdRect = listaActsOrd
        flag = False
    return rta2, flag, listaActsOrdRect, codeMAX
```

El proceso de solución del requerimiento 5 cumple la misma estructura de la solución del requerimiento 4. Sin embargo, este punto debe retornar el subsector económico con mayores descuentos tributarios en un año, no los costos y gastos de nómina. Por esta razón, se deben cambiar los headers para incluir los descuentos tributarios. Además, al realizar la acumulación de datos se debe incluir los descuentos tributarios de los subsectores al mapa auxiliar.

Entrada	El mapa con los datos, un año específico y las categorías a imprimir

Salidas	Lista con las actividades del subsector con menores retenciones en el año ingresado, el tiempo en ms y la memoria en kbs.
Implementado (Sí/No)	Si, Juan Bernardo Parra

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Recorrer la variable *llave*	O(n)
Recorrer la variable *listaSS*	O(n)
Recorrer la información de cada subsector	O(k) donde k son el número de actividades del subsector
Ordenar la lista de actividades mediante Merge	O(n log n)
TOTAL	***O(n log n)***

Pruebas Realizadas

Las pruebas se realizaron en un computador con las siguientes características:

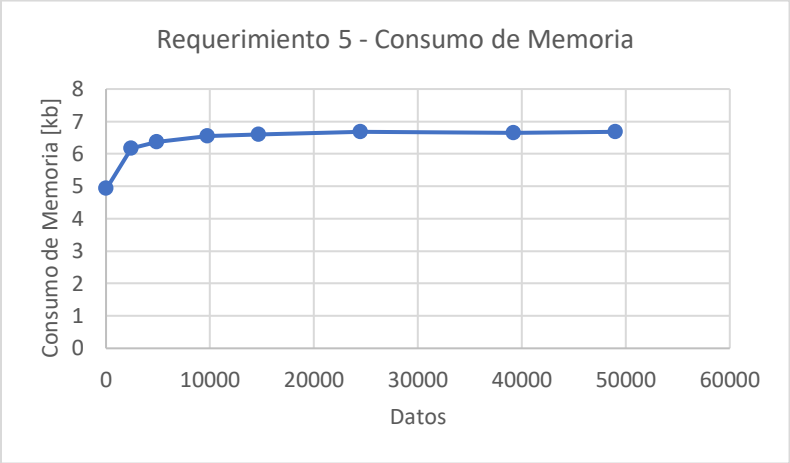
Procesador	Intel(R) Core(TM) i7-1065G7 CPU @ 1.30GHz 1.50 GHz
Memoria RAM (GB)	12.0 GB (11.7 GB utilizable)
Sistema Operativo	Windows 11 Home Single Language – 64 bits

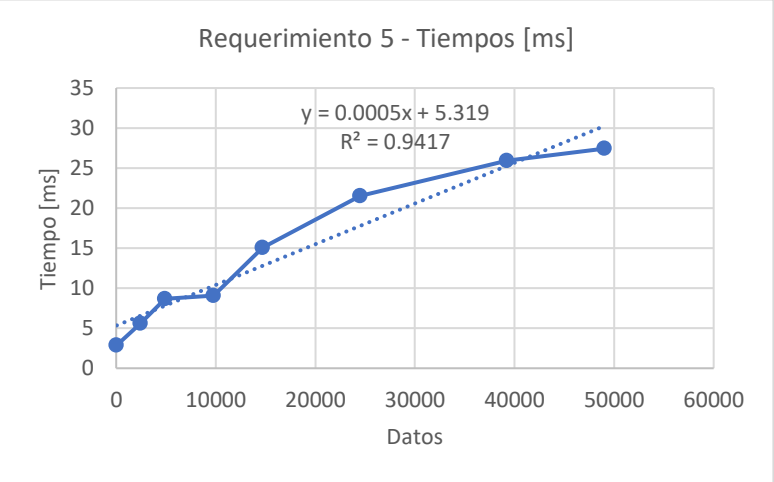
Se realizó el mismo procedimiento descrito en el requerimiento 1 para la medición de tiempo y consumo de memoria. Para las mediciones del requerimiento 2, se ingresó como parámetro el año 2021. Se usó el método de linear probing con un factor de carga de 0.5.

Tablas de datos

Datos	Consumo de Datos [kB]	Tiempo de Ejecución Real @LP [ms]
49	4.93	2.875
2450	6.164	5.63
4900	6.367	8.66
9800	6.555	9.1
14700	6.602	15.04
24510	6.68	21.5
39220	6.64	25.9
49030	6.68	27.43

Graficas





Análisis

Similar a lo ocurrido en el requerimiento 4 se observa que la gráfica tiene un comportamiento de O(nlog(n)) por lo que concuerda con el análisis de complejidad teórico realizado previamente. Lo anterior, se debe a la implementación del tipo de ordenamiento MergeSort que tiene como complejidad temporal O(nlog(n)). En términos de memoria, se aprecia un comportamiento esperado dado que la memoria incrementa considerablemente a medida de que la cantidad de datos es mayor.

Requerimiento 6

Descripción

```python
def req_6(data_structs, anio,headers, headers2):
    """
    Función que soluciona el requerimiento 6
    """
    # TODO: Realizar el requerimiento 6
    map = data_structs['anioSEC']
    llaves  = mp.keySet(map)
    listaSS = lt.newList('ARRAY_LIST')
    for i in lt.iterator(llaves):
        if (anio in i) == True:
            imp = mp.get(map,i)
            lt.addLast(listaSS, imp)
    #acumilaciones
    '''
    headers = ['Código sector económico', 'Nombre sector económico','Total ingresos netos',
            'Total costos y gastos','Total saldo a pagar', 'Total saldo a favor','Subsector que mas aporto','Subsector que menos aporto']
    '''
    #mapa
    r1 = lt.newList('ARRAY_LIST')
    pos_max = 1
    pos = 1
    max_ = 0
    for i in lt.iterator(listaSS):
        mapA = mp.newMap(10,maptype='PROBING',loadfactor=0.7)
        for j in headers:
            mp.put(mapA,j,0)
        for k in lt.iterator(i['value']['impuesto']):
            if me.getValue(mp.get(mapA,'Código sector económico'))==0:
                mp.put(mapA,'Código sector económico',k['Código sector económico'])
                mp.put(mapA,'Nombre sector económico',k['Nombre sector económico'])
            mp.put(mapA,'Total ingresos netos',me.getValue(mp.get(mapA,'Total ingresos netos'))+int(k['Total ingresos netos']))
            mp.put(mapA,'Total costos y gastos',me.getValue(mp.get(mapA,'Total costos y gastos'))+int(k['Total costos y gastos']))
            mp.put(mapA,'Total saldo a pagar',me.getValue(mp.get(mapA,'Total saldo a pagar'))+int(k['Total saldo a pagar']))
            mp.put(mapA,'Total saldo a favor',me.getValue(mp.get(mapA,'Total saldo a favor'))+int(k['Total saldo a favor']))
        lt.addLast(r1,mapA)
        if max_ < me.getValue(mp.get(mapA,'Total ingresos netos')):
                max_ = me.getValue(mp.get(mapA,'Total ingresos netos'))
                pos_max = pos
        pos += 1
        rta1 = lt.getElement(r1, pos_max)
    codeMAX =  me.getValue(mp.get(rta1,'Código sector económico'))
    #,ActMxSSma, ActMxSSmi, ActMiSSma, ActMiSSmi
    code2max, code2min, subSecmax, subSecmin =  req6_p2(data_structs, anio, headers2,codeMAX)
    mp.put(rta1,'Subsector que mas aporto',code2max)
    mp.put(rta1,'Subsector que menos aporto',code2min)
    rta2 = ListaFromMap(rta1)

    return rta2, subSecmax, subSecmin
```

```python
def req6_p2(data_structs, anio, headers,codeMAX):
    map = data_structs['anio_SEC_SUBSEC']
    llaves  = mp.keySet(map)
    listaSS = lt.newList('ARRAY_LIST')
    for i in lt.iterator(llaves):
        if (anio+","+codeMAX in i) == True:
            imp = mp.get(map,i)
            lt.addLast(listaSS, imp)
    #acumilaciones
    '''
     headers2 = ['Código subsector económico','Nombre subsector económico','Total ingresos netos',
                 'Total costos y gastos','Total saldo a pagar','Total saldo a favor', 'Actividad economica que mas aporto', 'Actividad economica que menos aporto']
    '''
    #mapa
    r1 = lt.newList('ARRAY_LIST')
    pos_max = 1
    pos = 1
    max_ = 0
    #------
    pos_min = 1
    pos2 = 1
    min_= 100_000_000_000_000
    for i in lt.iterator(listaSS):
        mapA = mp.newMap(10,maptype='PROBING',loadfactor=0.7)
        for j in headers:
            mp.put(mapA,j,0)
        for k in lt.iterator(i['value']['impuesto']):
            if me.getValue(mp.get(mapA,'Código subsector económico'))==0:
                mp.put(mapA,'Código subsector económico',k['Código subsector económico'])
                mp.put(mapA,'Nombre subsector económico',k['Nombre subsector económico'])
            mp.put(mapA,'Total ingresos netos',me.getValue(mp.get(mapA,'Total ingresos netos'))+int(k['Total ingresos netos']))
            mp.put(mapA,'Total costos y gastos',me.getValue(mp.get(mapA,'Total costos y gastos'))+int(k['Total costos y gastos']))
            mp.put(mapA,'Total saldo a pagar',me.getValue(mp.get(mapA,'Total saldo a pagar'))+int(k['Total saldo a pagar']))
            mp.put(mapA,'Total saldo a favor',me.getValue(mp.get(mapA,'Total saldo a favor'))+int(k['Total saldo a favor']))
        lt.addLast(r1,mapA)
        if max_ < me.getValue(mp.get(mapA,'Total ingresos netos')):
                max_ = me.getValue(mp.get(mapA,'Total ingresos netos'))
                pos_max = pos
        pos += 1
        if min_ > me.getValue(mp.get(mapA,'Total ingresos netos')):
                min_ = me.getValue(mp.get(mapA,'Total ingresos netos'))
                pos_min = pos2
        pos2 += 1
        rta1_p2 = lt.getElement(r1, pos_max)
        rta1_1 = lt.getElement(r1, pos_min)
    codeMAX2 =  me.getValue(mp.get(rta1_p2,'Código subsector económico'))
    codeMIN =  me.getValue(mp.get(rta1_1,'Código subsector económico'))
    listaActs =  (me.getValue(mp.get(map,anio+","+codeMAX+","+str(codeMAX2))))['impuesto']
    listaActs2 =  (me.getValue(mp.get(map,anio+","+codeMAX+","+str(codeMIN))))['impuesto']
    listaActsOrd = useMerge(listaActs, cmp_req_6)
    listaActsOrd2 = useMerge(listaActs2, cmp_req_6)
    mp.put(rta1_p2,'Actividad economica que mas aporto',lt.firstElement(listaActsOrd))
    mp.put(rta1_p2,'Actividad economica que menos aporto',lt.lastElement(listaActsOrd))
    mp.put(rta1_1,'Actividad economica que mas aporto',lt.firstElement(listaActsOrd2))
    mp.put(rta1_1,'Actividad economica que menos aporto',lt.lastElement(listaActsOrd2))
    rta2 = ListaFromMap(rta1_p2)
    rta2_1 = ListaFromMap(rta1_1)
    return  codeMAX2, codeMIN,rta2, rta2_1
```

El requerimiento 6 debe encontrar el sector económico con el mayor total de ingresos netos para un año específico. Sin embargo, en la respuesta se deben incluir los subsectores económicos que más aportaron y que menos aportaron a este total de ingresos. Por esta razón, el requerimiento se dividió en dos: una función para buscar los máximos y otra para los mínimos. Cada una de estas funciones trabaja de manera similar a las funciones de los requerimientos 3, 4 y 5.

Entrada	Parámetros necesarios para resolver el requerimiento.
Salidas	Respuesta esperada del algoritmo.
Implementado (Sí/No)	Si se implementó y quien lo hizo.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Recorrer dos veces la variable *llave*	O(n)
Recorrer dos veces la variable *listaSS*	O(n)

Recorrer dos veces la información de cada subsector	O(k) donde k son el número de actividades del subsector
Ordenar dos veces la lista de actividades mediante Merge	O(n log n)
TOTAL	***O(n log n)***

Pruebas Realizadas

Las pruebas se realizaron en un computador con las siguientes características:

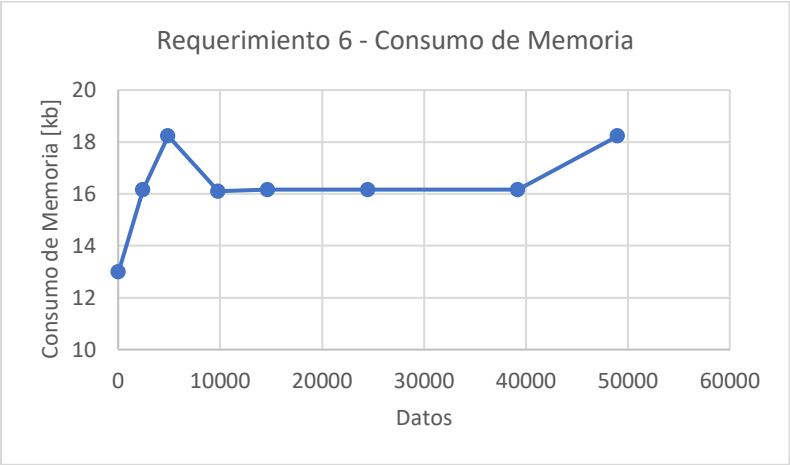
Procesador	Intel(R) Core(TM) i7-1065G7 CPU @ 1.30GHz 1.50 GHz
Memoria RAM (GB)	12.0 GB (11.7 GB utilizable)
Sistema Operativo	Windows 11 Home Single Language – 64 bits

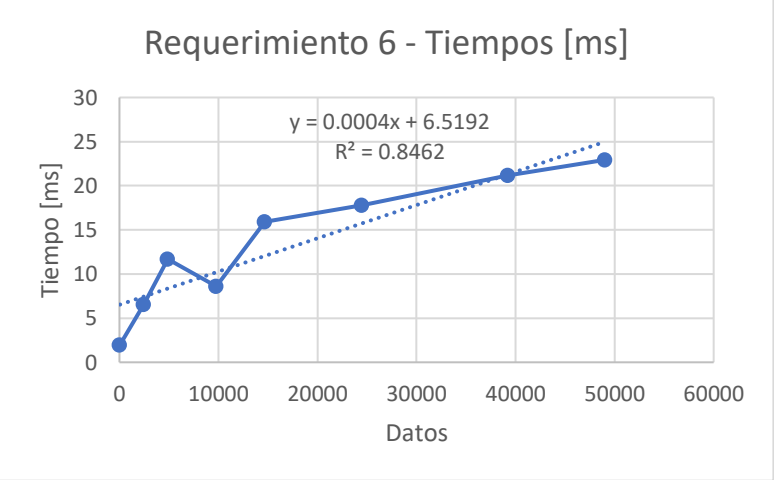
Se realizó el mismo procedimiento descrito en el requerimiento 1 para la medición de tiempo y consumo de memoria. Para las mediciones del requerimiento 6, se ingresó como parámetro el año 2021. Se usó el método de linear probing con un factor de carga de 0.5.

Tablas de datos

Datos	Consumo de Datos [kB]	Tiempo de Ejecución Real @LP [ms]
49	13.01	1.987
2450	16.156	6.54
4900	18.231	11.67
9800	16.094	8.6
14700	16.156	15.93
24510	16.156	17.8
39220	16.156	21.17
49030	18.231	22.931

Graficas





Análisis

A partir de los resultados de las pruebas realizados, se aprecia que hay un comportamiento lineal del algoritmo al probar este con una cantidad de datos en aumento. Lo anterior, no responde al analisis de complejidad teórico enunciado previamente, por lo que, es posible que el algoritmo se comporte dependiendo de la cantidad de datos contenidos en cada subsector económico. Por otro lado, en términos de memoria esta posee un comportamiento poco habitual dado que no tiene un crecimiento constante a medida que aumentan los datos cargados.

Requerimiento 7

Descripción

```
def req_7(data_structs, SS, anio,top):
    """
    Función que soluciona el requerimiento 7
    """
    # TODO: Realizar el requerimiento 7
    map = data_structs['anioSUBSEC']
    KEY = anio+","+SS
    imp = mp.get(map,KEY)
    lista1 = me.getValue(imp)
    lista2 = lista1['impuesto']
    lista3 = useMerge(lista2, cmp_req_7_tot)
    flag = None
    if int(top) <= lt.size(lista3):
        lista4 = lt.subList(lista3,1,int(top))
        flag = True
        return lista4, flag
    else:
        flag = False
        return lista3, flag
```

```
def cmp_req_7(impuesto1, impuesto2):

    '''
    Funcion que retorna True si los Total costos y gastos de un impuesto1 es menor que el del impuesto2.
    '''


    return int(impuesto1['Total costos y gastos']) < int(impuesto2['Total costos y gastos'])
def cmp_req_7_1(impuesto1, impuesto2):
    return int(impuesto1['Código actividad económica']) > int(impuesto2['Código actividad económica'])
def cmp_req_7_tot(impuesto1, impuesto2):
    """
    Devuelve verdadero si el año de impuesto1 es menor que el de impuesto2 en el caso de que
    sean iguales, se revisa el código de actividad ecoonómica, de lo contrario retorna false.

    Arg:

    Impuesto1: Información del primer registro que incluye el año y el código de la actividad economica
    Impuesto2: Información del primer registro que incluye el año y el código de la actividad economica
    """
    if int(impuesto1['Año']) == int(impuesto2['Año']):
        return cmp_req_7(impuesto1, impuesto2)
    else:
        return cmp_req_7_1(impuesto1, impuesto2)
```

En el requerimiento 7 se deben listar N actividades económicas con el menor total de costos y gastos para un subsector y un año específicos. Para empezar, se almacenan la pareja año y subsector ingresados por parámetro en la variable KEY. Esta se usa para obtener la lista de actividades que se realizan en el subsector y año dados y se almacenan en lista1. Posteriormente, se crea la lista2 para ingresar a la lista de todas las actividades económicas del año y subsector dados. Por último, en la lista3 se ordena la lista2. Si el tamaño de la lista 3 es mayor al número de actividades por listar ingresada por parámetro, se realiza un sublist con el parámetro ingresado.

Entrada	El año y subsector deseados, los datos y el n de actividades que se quieren listar
Salidas	Las N actividades económicas con el menor total de costos y gastos para un subsector y un año específicos

Implementado (Sí/No)	Si se implementó y quien lo hizo.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Obtener la variable imp mediante mp.get	O(1)
Usar me.getValue	O(n)
Ordenar la lista con Merge	O(n log n)
TOTAL	***O(n log n)***

Pruebas Realizadas

Las pruebas se realizaron en un computador con las siguientes características:

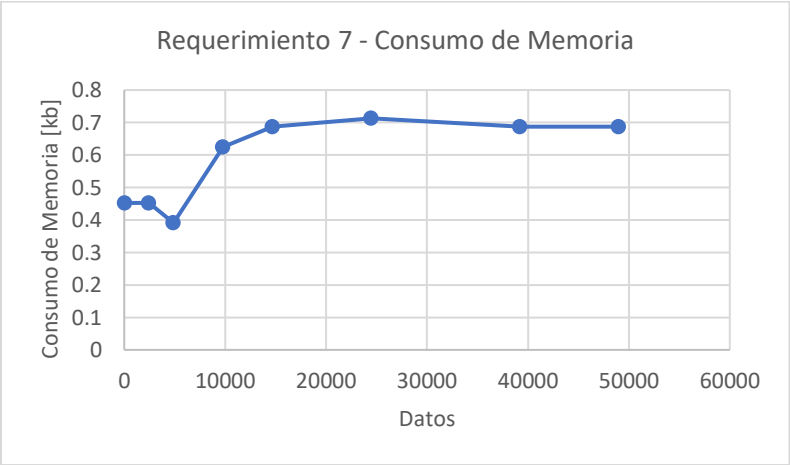
Procesador	Intel(R) Core(TM) i7-1065G7 CPU @ 1.30GHz 1.50 GHz
Memoria RAM (GB)	12.0 GB (11.7 GB utilizable)
Sistema Operativo	Windows 11 Home Single Language – 64 bits

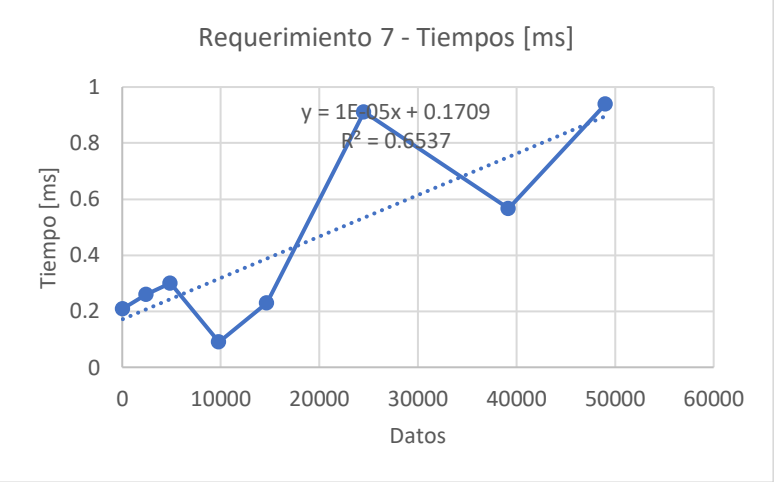
Se realizó el mismo procedimiento descrito en el requerimiento 1 para la medición de tiempo y consumo de memoria. Para las mediciones del requerimiento 7, se ingresaron como parámetros el año 2021, el subsector 11 y se pidió listar el top 3 de actividades. Se usó el método de linear probing con un factor de carga de 0.5.

Tablas de datos

Datos	Consumo de Datos [kB]	Tiempo de Ejecución Real @LP [ms]
49	0.453	0.209
2450	0.453	0.26
4900	0.391	0.301
9800	0.625	0.09
14700	0.688	0.23
24510	0.713	0.912
39220	0.688	0.566
49030	0.688	0.939

Graficas





Análisis

Basándose en los resultados de las pruebas realizadas se logra apreciar que la curva no tiene una tendencia muy clara. Sin embargo, la tendencia que mejor se le ajusta a la curva es una de comportamiento lineal, por lo que, el algoritmo adquiere dicha complejidad en la práctica. Es posible que esto se deba a que puede que el ordenamiento de los datos sea más eficiente o menos dependiendo de la forma en que los datos de cierto año estén entrando en el modelo. Por otro lado, en términos de memoria como en la mayoría de requerimientos se esperaba un incremento notable a medida de que la cantidad de datos era mayor.

Requerimiento 8

Descripción

```python
def req_8(data_structs, anio,headers,TOP):
    """
    Función que soluciona el requerimiento 8
    """
    # TODO: Realizar el requerimiento 8
    map = data_structs['anioSUBSEC']
    llaves  = mp.keySet(map)
    listaSS = lt.newList('ARRAY_LIST')
    for i in lt.iterator(llaves):
        if (anio in i) == True:
            imp = mp.get(map,i)
            lt.addLast(listaSS, imp)
    #acumilaciones
    '''
    headers=['Código sector económico','Nombre sector económico','Código subsector económico',
                    'Nombre subsector económico','Costos y gastos nómina','Total ingresos netos',
                    'Total costos y gastos','Total saldo a pagar','Total saldo a favor']
    '''
    #mapa
    r1 = lt.newList('ARRAY_LIST')

    for i in lt.iterator(listaSS):
        mapA = mp.newMap(10,maptype='PROBING',loadfactor=0.7)
        for j in headers:
            mp.put(mapA,j,0)
        for k in lt.iterator(i['value']['impuesto']):
            if me.getValue(mp.get(mapA,'Código subsector económico'))==0:
                mp.put(mapA,'Código sector económico',k['Código sector económico'])
                mp.put(mapA,'Nombre sector económico',k['Nombre sector económico'])
                mp.put(mapA,'Código subsector económico',k['Código subsector económico'])
                mp.put(mapA,'Nombre subsector económico',k['Nombre subsector económico'])
            mp.put(mapA,'Total Impuesto a cargo',me.getValue(mp.get(mapA,'Total Impuesto a cargo'))+int(k['Total Impuesto a cargo']))
            mp.put(mapA,'Total ingresos netos',me.getValue(mp.get(mapA,'Total ingresos netos'))+int(k['Total ingresos netos']))
            mp.put(mapA,'Total costos y gastos',me.getValue(mp.get(mapA,'Total costos y gastos'))+int(k['Total costos y gastos']))
            mp.put(mapA,'Total saldo a pagar',me.getValue(mp.get(mapA,'Total saldo a pagar'))+int(k['Total saldo a pagar']))
            mp.put(mapA,'Total saldo a favor',me.getValue(mp.get(mapA,'Total saldo a favor'))+int(k['Total saldo a favor']))
        lt.addLast(r1,mapA)
```

```
    listaMapsOrd = useMerge(r1, cmp_req_8 )
    listaRespuesta = lt.newList("ARRAY_LIST")
    listarespuesta2 = lt.newList("ARRAY_LIST")
    LC = lt.newList("ARRAY_LIST")
    for i in lt.iterator(listaMapsOrd):
        lt.addLast(listaRespuesta,ListaFromMap(i))
        codigoSS = me.getValue(mp.get(i,'Código subsector económico'))
        lt.addLast(LC, codigoSS)
        listaActs = (me.getValue(mp.get(map,anio+","+str(codigoSS))))['impuesto']
        listaActsOrd = useMerge(listaActs, cmp_req_8_1)
        if lt.size(listaActsOrd) >= int(TOP):
            listaActsOrd2 = lt.subList(listaActsOrd,1,int(TOP))
        else:
            listaActsOrd2 = listaActsOrd
        flag = None
        if int(TOP) >= 12:
            if lt.size(listaActsOrd2) > 6:
                listaActsOrdRect = recortarLista(listaActsOrd2)
                flag = True
            else:
                listaActsOrdRect = listaActsOrd2
                flag = False
            lt.addLast(listarespuesta2,listaActsOrdRect)
        else:
            listaActsOrdRect = listaActsOrd2
            flag = False
            lt.addLast(listarespuesta2,listaActsOrdRect)
    return listaRespuesta, listarespuesta2, flag, LC #rta2, flag, listaActsOrdRect, codeMAX
```

Para la solución del requerimiento 8 se empieza por obtener el mapa anioSUBSEC, el cual contiene los datos del reto y tiene como llaves el año y el código del subsector. Estas llaves se almacenan en la variable llaves mediante la función mp.keySet. Posteriormente, se crea una lista llamada listaSubSectores, en donde se almacenan todas las llaves que contengan el año ingresado por parámetro (para esto es necesario recorrer todo llaves). Posteriormente se recorre todo listaSS y se añade su información en un mapa auxiliar. A este mapa auxiliar se le añaden todos los subsectores presentes en listaSS y se itera para acumular los impuestos a cargo, los ingresos netos, los costos y gastos, el saldo a favor y el saldo a pagar. Una vez se realizan las acumulaciones, se ordena mediante MergeSort. Se crea una *listaRespuesta*, a la cual se añade cada elemento del mapa previamente ordenado y se convierte a una lista de tuplas usando la función ListfromMap. Se crea la listaActs, la cual contiene las actividades económicas del subsector en el año especificado y se ordena usando Merge. Si el tamaño de la listaActs es mayor al top indicado por el usuario, se recorta usando sublist.

Entrada	El mapa con los datos, el año a analizar, el top deseado y los headers que se quieren visualizar en la impresión
Salidas	Una lista con el top de actividades de cada subsector para el año especificado con los mayores impuestos a cargo.
Implementado (Sí/No)	Si se implementó y quien lo hizo.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	Complejidad
Recorrer la variable *llave*	O(n)
Recorrer la variable *listaSS*	O(n)
Recorrer la información de cada subsector	O(k) donde k son el número de actividades del subsector
Ordenar la lista de actividades mediante Merge	O(n log n)
Recorrer la lista ordenada	O(n)
TOTAL	***O(n log n)***

Pruebas Realizadas

Las pruebas se realizaron en un computador con las siguientes características:

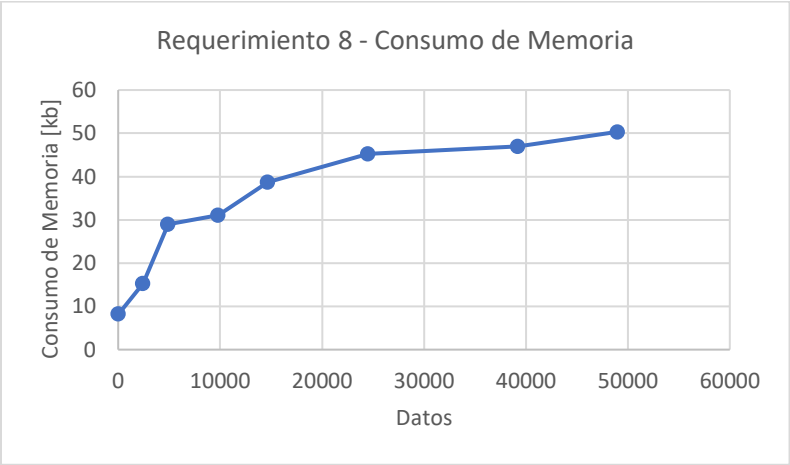
Procesador	Intel(R) Core(TM) i7-1065G7 CPU @ 1.30GHz 1.50 GHz
Memoria RAM (GB)	12.0 GB (11.7 GB utilizable)
Sistema Operativo	Windows 11 Home Single Language – 64 bits

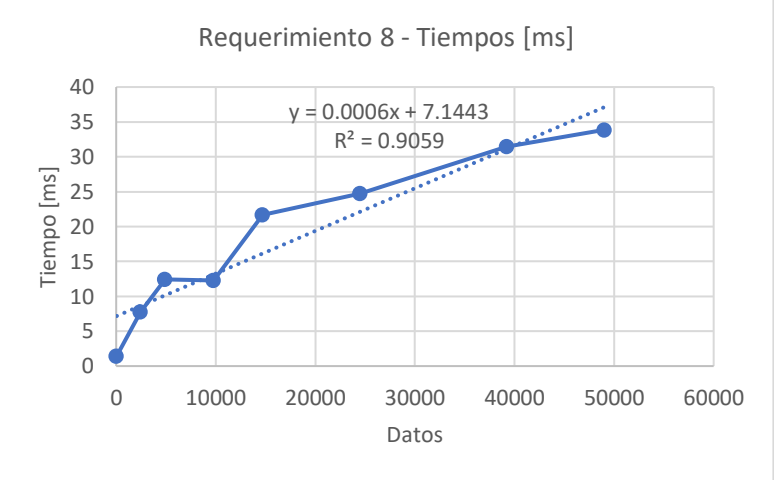
Se realizó el mismo procedimiento descrito en el requerimiento 1 para la medición de tiempo y consumo de memoria. Para las mediciones del requerimiento 2, se ingresaron como parámetros el año 2021 y el subsector 11. Se usó el método de linear probing con un factor de carga de 0.5.

Tablas de datos

Datos	Consumo de Datos [kB]	Tiempo de Ejecución Real @LP [ms]
49	8.266	1.44
2450	15.336	7.73
4900	28.979	12.39
9800	31.099	12.25
14700	38.767	21.71
24510	45.312	24.74
39220	47	31.48
49030	50.359	33.87

Graficas





Análisis

El comportamiento del consumo de memoria del requerimiento 8 es esperado, pues crece de manera directamente proporcional con el crecimiento de los datos. Por otro lado, la gráfica de tiempos muestra un comportamiento lineal, algo un poco alejado de lo esperado según la teoría, pues se esperaba un comportamiento de la forma $x * \log x$

Comparación de resultados con el reto 1

Con el fin de comparar la efectividad temporal de los requerimientos del reto 1 y del reto 2, se tomaron los resultados de tiempos tomados cuando se usó Array List y un 100% de los datos y se compararon a cuando se uso probing y un 100% de los datos. El resultado es el siguiente:

Requerimiento	Reto 1	Reto 2	Diferencia
1	131.45	16.19	115.26
2	202.076	20.32	181.756
3	1887.4	41.77	1845.63
4	2663	27.61	2635.39
5	1048.32	27.43	1020.89

6	194	22.931	171.069
7	910	0.939	909.061
8	-	33.87	-

Como se puede observar, todos los requerimientos del reto 2 son considerablemente más efectivos temporalmente que los requerimientos del reto 1. Lo anterior, debido a que especialmente en los requerimientos individuales y avanzados al realizar unas acumulaciones es mucho más rápido acceder realizar sumatorias implementando mapas dado que estos permiten tener parejas llave-valor que son más eficientes al momento de acceder a la información necesitada.