Reto 2

Sebastian Guerrero – 202021249

Diego Alejandro Gonzalez – 202110240

Pautas Generales:

De manera general, tras haber hecho las pruebas del laboratorio 6, decidimos hacer uso de la implementación de mapas con el algoritmo de manejo de colisiones de “PROBING” con un factor de carga de 0.5. Esto se decidió ya que la memoria del computador con el que se realizaron las pruebas soportaba el espacio extra que requería y mejoraba la considerablemente la velocidad de la implementación del TAD. Además de esto, se tuvo que crear listas de ayuda para poder organizar los datos pedidos. Para esto se utilizo la estructura de “ARRAY_LIST” y el método de ordenamiento de merge sort. Para las pruebas, se mantuvo constante la cantidad de datos esperados en el mapa. Por esta razón, la utilización de la memoria se mantuvo constante en su mayor parte. Los datos de la maquina de prueba se pueden ver en la siguiente tabla:

	Máquina de prueba	Maquina de Prueba 2
Procesador	Intel (R) Core (TM) i7-10800H CPU @ 3.40 GHz	Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz 2.90 GHz
Memoria RAM (GB)	16 GB	8GB
Sistema Operativo	Windows 10 x64 bits	Windows 10 x64 bits

Requerimiento 1

Inicialmente, para el caso del reto 1, se cumplió con el requerimiento 1 a partir de la creación de una lista en la función de carga que tenía toda la información de los autores en cada una de sus posiciones. Posteriormente, se organizaba toda esa información por los años de nacimiento de los artistas de forma ascendente, con el algoritmo de ordenamiento de listas MergeSort. Luego, se realizaban 2 búsquedas binarias para la determinación de los límites de los datos útiles dentro del arreglo para un rango de años determinado. Finalmente, se creaba una sublista para el almacenamiento de estos datos. En conclusión, y como producto de la función de ordenamiento del lote de datos completo, se dijo en su momento que **el algoritmo, en notación Big O, tenía una complejidad estimada de O(nlogn), es decir, linearítmica**

Por otra parte, para el caso de este nuevo reto, este código se tuvo que modificar enormemente para mejorar su rendimiento. Así las cosas, se cambió la función cargar, en donde se eliminó la lista que almacenaba la información completa de cada autor.

```
catalog['yearsborn'] = mp.newMap(4000,maptype='PROBING',loadfactor = 0.5)
```

Después de esto, se creó la nueva estructura de datos que correspondería a una tabla de símbolos o mapa. Ahora bien, este mapa tendría por llave los años de nacimiento de cada uno

de los artistas, y por llave, toda la información correspondiente a los artistas nacidos en dicho año dentro de una lista.

```
def addArtistBorn(catalog, artist):
    try:
        years = catalog['yearsborn']
        if (artist['BeginDate'] != ''):
            bornyear = artist['BeginDate']
            bornyear = int(float(bornyear))
        else:
            bornyear = 99999
        existyear = mp.contains(years, bornyear)
        if existyear:
            entry = mp.get(years, bornyear)
            year = me.getValue(entry)
        else:
            year = newYear(bornyear)
            mp.put(years, bornyear, year)
        lt.addLast(year['artists'], artist)
    except Exception:
        return None
```

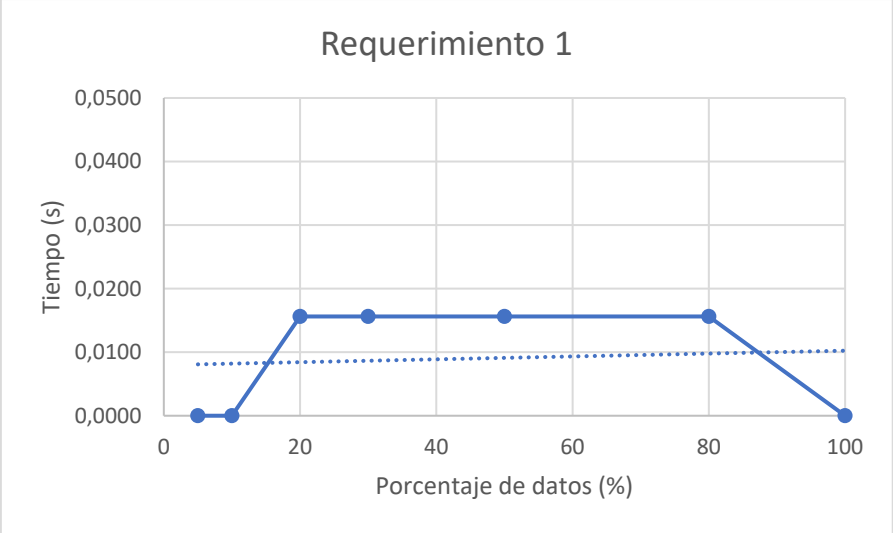
Dicho todo lo anterior, se procedía en este momento a ejecutar la función del requerimiento 1. En este sentido, el único procedimiento necesario era sacar la información correspondiente a las llaves de los años que ocupaban el rango de interés. Para ello, se realizó que recorriese dichos años, sacara las listas de cada uno de los años, y copiara sus elementos a una nueva lista que sería la que se mostraría al usuario.

```
def funcionReqUnoReto(catalog, inicial, final):
    ini = int(float(inicial))
    fin = int(float(final))
    tad_rta = lt.newList("ARRAY_LIST")
    for i in range(ini,fin+1):
        anio = mp.get(catalog['yearsborn'], i)
        if anio:
            artistas = me.getValue(anio)
            a_pasar=artistas["artists"]
            for x in range(1, lt.size(a_pasar)+1):
                elem= lt.getElement(a_pasar,x)
                lt.addLast(tad_rta, elem)
    return tad_rta
```

En suma, se puede decir que este ciclo de búsqueda será de un valor constante para el cálculo de la complejidad del código, en la medida en que la cantidad de años posibles a buscar son pronunciadamente menores que la cantidad de datos. **De este modo, la complejidad del algoritmo sería O(1) en notación Big O.** Sin embargo, se espera que en la práctica los tiempos sí varíen dependiendo de la cantidad de datos. Esto se debe a los ciclos de carga de datos en la lista de respuesta final, cuyos datos a copiar van a incrementar conforme la cantidad de datos aumente. Aunque, estos datos a copiar se esperan sean ampliamente inferiores a la cantidad de datos total.

Pruebas:

Porcentaje (%)	Tiempo (s)	Memoria (%)
5	0,0000	87
10	0,0000	89
20	0,01562	92
30	0,01562	94
50	0,01562	94
80	0,01562	96
100	0,0000	98



MAQUINA 2:

Porcentaje	Tiempo	Memoria Gastada
5%	0.015seg	6,78gb
10%	0.015seg	7.54gb
20%	0.0315	7.27gb
30%	0.015	7,42gb
50%	0.0315	7,45gb
80%	0.015seg	7,88gb
100%	0,015seg	7,88gb

En esta grafica se evidencia como si se cumple la complejidad constante, se logró una mejora muy grande frente al primer reto. Se evidencia como en todas las pruebas, el tiempo de ejecución es constante y muy cercano a cero. Aquí se muestra como los mapas pudieron mejorar los tiempos, a consecuencia de un mayor uso de memoria, ya que se observa una utilización casi completa de la memoria disponible en el sistema.

Requerimiento 2

Para el reto 1, este requerimiento se abordó con el uso de listas ordenadas (con el algoritmo de ordenamiento merge sort) y la búsqueda binaria para poder extraer los datos entre dos fechas. En este caso, se evidenciaba una complejidad de **O(nlogn)**, debido a las dos búsquedas binarias (mezclada con búsqueda secuencias) para encontrar la posición de las fecha inicial y final con el fin de poder extraer todo lo que se encuentra en el medio. Este método podría parecer un poco lento y se pudo mejorar los tiempos de respuesta con la aplicación del TAD mapa.

Para poder llevar a cabo la implementación con mapas, primero se creó un mapa en el catálogo en el cual las llaves son los años de adquisición de las obras. Se decidió utilizar únicamente el año ya que si se utilizará la fecha entera se comportaría como una lista no ordenada. Ya que puede haber varias obras adquiridas el mismo año, se dividió que el valor de cada llave fuera una lista (utilizando el TAD lista) con las diferentes obras que fueron adquiridas ese año. Como se mencionó previamente, este mapa fue creando, utilizando el manejo de colisiones "PROBING". La creación del mapa se puede ver a continuación. En esta foto se evidencia como primero se verifica la existencia de la llave (año) en el mapa si no existía se creaba una nueva lista para poder guardar las obras. Si la llave si existía, simplemente se añadía la obra a la lista existente.

```
def adquisicion(catalog, obra):

    mapa = catalog["DateAcquired"]
    fecha = obra["DateAcquired"][0:4]

    if fecha == None:
        fecha = 0

    existe = mp.contains(mapa, fecha)
    if existe:
        list = mp.get(mapa, fecha)
        li = me.getValue(list)
    else:
        li = lt.newList("ARRAY_LIST")
        mp.put(mapa, fecha, li)

    lt.addLast(li, obra)
```

Con el mapa ya creado, fue mucho más fácil el extraer la información de los diferentes años en el rango, ya que se extrajeron las llaves del mapa y se itero sobre ellas confirmando si pertenecían al rango. Ya que el año en el que las obras fueron adquiridas no tiene un rango tan grande, se puede entender la complejidad de esta iteración como O(n) donde n es el número de años diferentes que fueron adquiridas las obras. Ya que este número no es tan grande y en nuestras pruebas tan solo vimos unos 89 años diferentes, podemos entender esta complejidad como constante O(1).

Por otro lado, tuvimos una limitación en cuanto al mapa, ya que lo creamos en base al año y no la fecha completa, tuvimos que iterar por todas las obras de los años iniciales y finales, y revisar si pertenecían al rango. A pesar de que esta iteración se puede considerar como O(m) donde m es el número de obras en un año especifico, en las pruebas encontramos que este número tenía una buena distribución por lo cual no había un gran número de obras en cada año (final e inicial), también podemos entender esta complejidad como constante en los mejores casos 2O(1).

Por otro lado, también se utiliza el método de merge para poder organizar la respuesta del requerimiento, esto significa una complejidad de O(logn)

En ambos casos decidimos hacer una iteración secuencias normal ya que los números de elementos no era tan grande, consideramos que el tiempo que nos ahorramos haciendo una búsqueda binaria, se podría perder en el hecho de que tocaba hacer un ordenamiento antes.

Teniendo en cuenta las dos iteraciones que debemos tener para poder completar con el requerimiento, se entiende la complejidad total como O(n) + O(m-inicial) + O(m-final) + O(logn), donde n es el número de años diferentes, y m es el número de obras en los años iniciales y finales. Como se dijo previamente estas cantidades no son tan grandes y se acerca bastante a ser una complejidad constante. Esta implementación se puede ver en las siguientes imágenes:

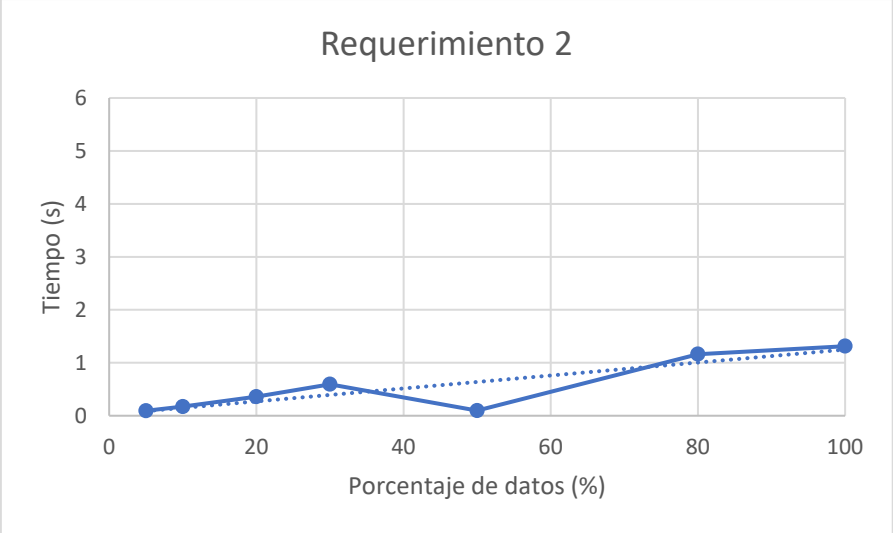
```
for i in range(ini, fin+1):

    if i == ini or i == fin:
        anio = mp.get(catalog["DateAcquired"], str(i))
        if anio:
            anio = me.getValue(anio)
            for x in range(1, lt.size(anio)+1):
                ele = lt.getElement(anio, x)

                if ele["DateAcquired"] >= inicial and ele["DateAcquired"] <= final:

                    if "purchase" in ele["CreditLine"].lower():
                        purhcased += 1

                    autores = "-"
                    au = ele["ConstituentID"].replace("[", "").replace("]", "").replace(" ", "").split(",")
                    for a in au:
                        art = mp.get(catalog["ArtistConstituent"], a)
                        if art:
                            arti = me.getValue(art)
                            autores = autores + arti["DisplayName"] + "-"

                    ele["ConstituentID"] = autores
                    lt.addLast(data, ele)
    else:
        anio = mp.get(catalog["DateAcquired"], str(i))
        if anio:
            obras = me.getValue(anio)
            for j in range(1, lt.size(obras)+1):
                ele = lt.getElement(obras, j)

                if "purchase" in ele["CreditLine"].lower():
                    purhcased += 1

                autores = "-"
                au = ele["ConstituentID"].replace("[", "").replace("]", "").replace(" ", "").split(",")
                for a in au:
                    art = mp.get(catalog["ArtistConstituent"], a)
                    if art:
                        arti = me.getValue(art)
                        autores = autores + arti["DisplayName"] + "-"
                ele["ConstituentID"] = autores
                lt.addLast(data, ele)

data = merge.sort(data, cmpdateadquired)

return [data, purhcased]
```

A parte de la utilización del índice de amo de adquisición, se utilizó el índice de artistas para poder encontrar rápidamente un artista con la búsqueda de un constitudent ID. Esto también logro reducir los tiempos de ejecución y carga de los datos.

Tras cumplir con el requerimiento, se realizaron pruebas con cada uno de los porcentajes de datos para poder mostrar como la implementación responde a volúmenes más grandes de datos y si se mantiene esa complejidad. Además, para verificar las mejoras en cuanto a tiempo comparando con la implementación del reto 1. Además, podemos ver que, en este caso, la complejidad bajo de O (nlogn) a casi constante por los volúmenes tan pequeños que toca organizar O (log n).

Pruebas:

Porcentaje (%)	Tiempo (s)	Memoria (%)
5	0,0937	86
10	0,1718	87
20	0,3593	93
30	0,5937	96
50	0,0968	94
80	1,1625	97
100	1,3125	97



MAQUINA 2:

Porcentaje	Tiempo	Memoria Gastada
5%	0.125seg	5,81gb
10%	0.281seg	7,33gb
20%	0.335seg	5.52gb
30%	0.90625seg	7,35gb
50%	1,75seg	7,34gb
80%	1,24seg	7,76gb
100%	0.825seg	7,88gb

En estos resultados se evidencia que el requerimiento no es completamente constante, pero si se logra un mejora considerable y medible en los tiempos de ejecución. Además, se evidencia como el crecimiento es mucho más pequeño al del reto anteriormente, ya que vemos que al 100 por ciento tan solo tarda 1.3 segundos. Como es esperado, si se evidencia una utilización de memoria bastante alto ya que se utiliza casi las 16GB disponibles en el computador de prueba, además se evidencia que esta utilización es constante a lo largo de los diferentes porcentajes, probablemente por la cantidad de valores esperados utilizados en la creación de los mapas, ya

que mantuvieron contantes y un poco altos para evitar rehashing y reducir el tiempo de carga de los archivos.

Requerimiento 3 (Diego Alejandro Gonzalez Vargas)

Para el caso del requerimiento 3, en el primer reto, con el manejo exclusivo de listas, se pensó en la realización del código por secciones: Primero, se realizaba una búsqueda común del nombre del artista en una lista que contenía los datos de todos los artistas del MoMA con sus obras. Esta búsqueda, como se recorría el archivo elemento por elemento, era de orden O(n). Segundo, ya con el artista encontrado, se realizaba la búsqueda de cada una sus obras dentro de una lista que contenía toda la información de todas las obras del MoMA. Estas búsquedas, teniendo en cuenta que se organizaba el arreglo al principio con un algoritmo MergeSort, eran de orden O(mlogm), con m como la cantidad de obras totales. Finalmente, se hacían operaciones internas para el tratamiento de los datos encontrados en las búsquedas, para poder organizar 2 listas con la información que se iba a mostrar al usuario. Sin embargo, ninguna de estas operaciones tenía una frecuencia significativa para el análisis de su complejidad temporal. En conclusión, se podía decir que la complejidad de dicho algoritmo era de O(n + mlogm) donde n era el número total de artistas y m el número total de obras.

Por otra parte, para el caso del reto 2, las formas en que se abordó este requerimiento cambiaron bastante. Para empezar, se prescindió completamente de las listas que almacenaban la información completa de todos los artistas y obras. En su lugar, se creó un mapa que permitía el almacenamiento de toda la información de los autores.

```
catalog['ArtistConsti'] = mp.newMap(40000,maptype = "PROBING",loadfactor = 0.5)
```

Esta tabla de hash se diferenciaba de la creada para el requerimiento 2 en que manejaba por llave el nombre de cada uno de los artistas, y por valor toda la información relativa a este.

```
def addArtistConsti(catalog, artist):

    artistas = catalog['ArtistConsti']
    artist['requetres']= mp.newMap(1000,maptype = "PROBING",loadfactor = 0.5)
    artist['ltrequetres']=lt.newList('ARRAY_LIST')
    mp.put(artistas, artist["DisplayName"], artist)
```

Sin embargo, como se puede ver en el código enseñado, la información que se carga en cada uno de los valores del mapa no se limita a la del artista en el csv por defecto, sino que se crean 2 nuevas variables: un mapa y una lista. Estos elementos van a ser las respuestas a cada una de las necesidades del requerimiento 3. En consecuencia, el mapa va a ser el encargado de, en la carga del csv de las obras, asignar cada obra con su autor.

```
    }
    catalog['Artworks']+=1
    addToAuthor(catalog, obra)
```

Para ello, se asignará una llave de este nuevo mapa interno para cada medio, mientras que los valores de esta tabla de símbolos será la información de cada una de las obras realizadas por el

autor en ese medio específico. Así mismo, dentro de esta misma función se controla la lista de medios, en la que se maneja los contadores de cada uno de los medios utilizados por el artista, como se puede ver a continuación:

```
def addToAuthor(catalog, artwork):
    #requetres = catalog['Nationality']

    artistas = artwork["ConstituentID"].strip("[]").replace(" ", "").split(",")

    for i in artistas:
        nat = mp.get(catalog["ArtistConstituent"], i)
        nat = me.getValue(nat)
        nat = nat["DisplayName"]
        pa_requetres = mp.get(catalog['ArtistConsti'], nat)
        pa_requetres = me.getValue(pa_requetres)
        med_pa_requetres = artwork["Medium"]

        if med_pa_requetres.lower() in (None, "", "unknown"):
            med_pa_requetres = "Unknown"

        existe = mp.contains(pa_requetres['requetres'], med_pa_requetres)
        if existe:
            elmed = mp.get(pa_requetres['requetres'], med_pa_requetres)
            elmedi = me.getValue(elmed)
            lista= pa_requetres['ltrequetres']
            for i in range(1, lt.size(lista)+1):
                ele = lt.getElement(lista, i)
                if ele["Medium"]==med_pa_requetres:
                    ele["Cant"]+=1
        else:
            elmedi = lt.newList("ARRAY_LIST")
            mp.put(pa_requetres['requetres'], med_pa_requetres, elmedi)
            lista= pa_requetres['ltrequetres']
            nuevoele={
                "Medium": med_pa_requetres,
                "Cant":1
            }
            lt.addLast(lista,nuevoele)

        lt.addLast(elmedi,artwork)
```

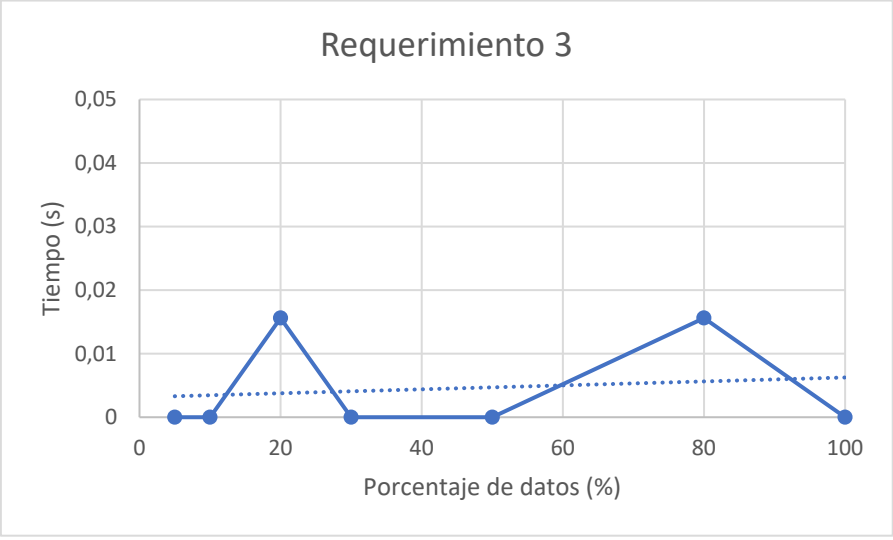
Por consiguiente, con toda esta organización realizada directamente en la función cargar, en el requerimiento el procedimiento tiene una complejidad temporal mucho más conveniente. Para ello, se obtiene la información del valor de la pareja llave-valor para el nombre del autor buscado por el usuario, proceso de complejidad O(1). Después de esto, se ordena la lista con los contadores de los medios utilizados por el artista. A pesar de que esta operación tiene una complejidad O(nlogn), como la cantidad de elementos es muy pequeña, no se toma en cuenta para el cálculo general. Posteriormente, para poder mostrar al usuario la cantidad de obras en el mapa, se realiza un ciclo que recorre los contadores de los medios, para generar una sumatoria de los mismos. Nuevamente, esta función tiene una complejidad no considerable para el cálculo general. Finalmente, se envía una tupla con la lista ordenada, el mapa con los elementos de cada medio (Aunque solo se va a imprimir la lista que corresponda a la llave del medio más utilizado) y el contador de todas las obras de ese artista.

```
def funcionReqTres(catalog, nombre):
    autor = mp.get(catalog['ArtistConsti'], nombre)
    autor = me.getValue(autor)
    sinorden= autor['ltrequetres']
    ordenado= merge.sort(sinorden,cmpcount)
    elmapa = autor["requetres"]
    contador = 0
    for i in range(1,lt.size(ordenado)+1):
        ele= lt.getElement(ordenado,i)
        contador+= ele['Cant']
    tuplarta = ordenado, elmapa, contador
    return tuplarta
```

En suma, se puede decir que este requerimiento también se ejecuta con una complejidad temporal de O(1), gracias a que la todo el procesamiento necesario ya había sido evacuado en la carga.

Pruebas:

Porcentaje (%)	Tiempo (s)	Memoria (%)
5	0	88
10	0	90
20	0,0156	92
30	0	94
50	0	96
80	0,0156	97
100	0	96



MÁQUINA 2:

Porcentaje	Tiempo	Memoria gastada
5%	0,015seg	6,78
10%	0,001seg	7,40gb
20%	0,001seg	5,70gb
30%	0.015seg	7,40gb
50%	0.015	7,29gb
80%	0,015seg	7,65gb
100%	0,015gb	7,88gb

Requerimiento 4 (Sebastian Guerrero)

En el Reto 1 se abordó este problema por medio de una lista creada en la carga de datos que relacionaba las nacionalidades de los artistas y la obra. A partir de esta lista se realizaba un conteo y se creaban listas para cada nacionalidad, para posteriormente utilizar el comando de size y ordenarlo en base a este criterio. Con esta implementación, se obtuvo una complejidad de O(nlogn).

En este reto, se implementaron los mapas para poder obtener una complejidad menor. Para esto se creó un índice en el cual las llaves era la nacionalidad del artista, se guardaba la misma obra por la nacionalidad de cada artista. Como varias obras podían tener la misma nacionalidad, los valores de cada elemento del mapa (cada nacionalidad) es una lista implementada con el TAD lista donde se guardaban las diferentes obras que compartieran la misma nacionalidad. Esta implementación se puede evidenciar en la siguiente imagen:

```
def addNationality(catalog, obra):

    nationality = catalog['Nationality']

    artistas = obra["ConstituentID"].strip("[]").replace(" ", "").split(",")

    for i in artistas:
        nat = mp.get(catalog["ArtistConstituent"], i)
        nat = me.getValue(nat)
        nat = nat["Nationality"]

        autores = "-"
        for j in artistas:
            art = mp.get(catalog["ArtistConstituent"], j)
            art = me.getValue(art)["DisplayName"]

            autores = autores + art + "-"

        obra["ConstituentID"] = autores

        if nat.lower() in (None, "", "unknown", "nationality unknown"):
            nat = "Unknown"

        existe = mp.contains(nationality, nat)
        if existe:
            nal = mp.get(nationality, nat)
            na = me.getValue(nal)
        else:
            na = lt.newList("ARRAY_LIST")
            mp.put(nationality, nat, na)

        lt.addLast(na, obra)
```

A partir de este mapa, se hizo una iteración de las keys y la utilización de el TAD lista para consultar la cantidad de obras por cada nacionalidad. Esta información se guardo en una lista para poder ser ordenada utilizando el algoritmo de merge y teniendo en cuenta la cantidad de elementos en cada nacionalidad.

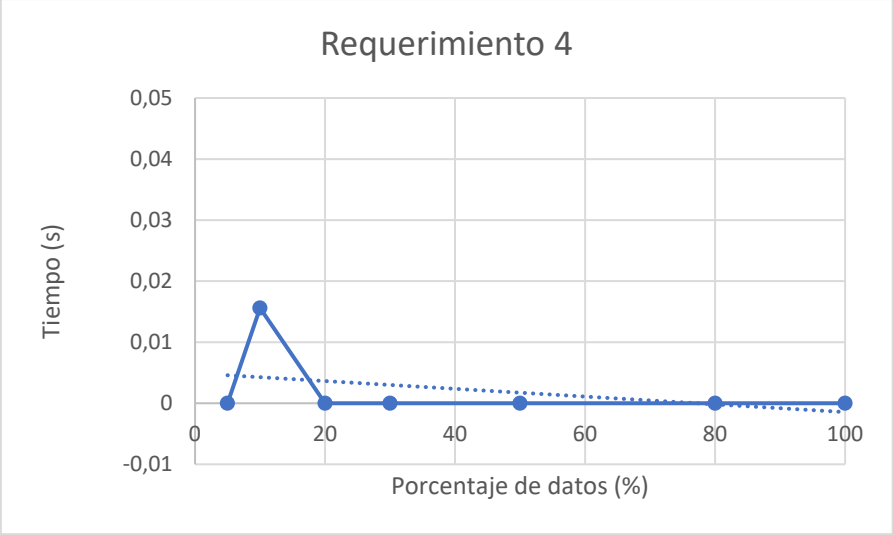
La cantidad de nacionalidades diferentes en el mapa son una cantidad muy pequeña por lo que la complejidad O(n) donde n es la cantidad de nacionalidades diferentes pueden ser tomadas como una constante O(1). Lo mismo sucede al momento de hacer el merge sort. Ya que la cantidad de nacionalidades diferentes, la complejidad O(logn), puede ser tomada como constante O(1).

Teniendo en cuenta las dos operaciones con mayor complejidad de la implementación se tiene una complejidad total de O(n) + O(logn). Considerando lo discutido previamente, nos percatamos de que esta complejidad se puede descartar y aproximarla a una complejidad contante. Esta implementación se puede ver en la siguiente imagen. Al igual que los otros requisitos, se realizaron pruebas con todos los porcentajes de datos para poder entender cómo funciona el algoritmo con volúmenes de datos mas grandes.

```
def ReqCuatro(catalog):

    nationality = catalog["Nationality"]

    keys = mp.keySet(nationality)
    keys = lt.iterator(keys)
    data = lt.newList("ARRAY_LIST")

    for i in keys:
        elem = mp.get(nationality, i)
        elem = me.getValue(elem)
        size = lt.size(elem)

        nat = {"Nationality": i,
               "size": size,
               "Artworks": elem}

        lt.addLast(data, nat)

    return merge.sort(data, cmpsize)
```

Pruebas:

Porcentaje (%)	Tiempo (s)	Memoria (%)
5	0	87
10	0,0156	89
20	0	92
30	0	94
50	0	94
80	0	96
100	0	98



MAQUINA 2:

Porcentaje	Tiempo	Memoria gastada
5%	0,015625seg	7,18
10%	0.015625seg	7,35gb
20%	0.03125seg	5,72gb
30%	0.015seg	7,57gb
50%	0.001seg	7,54gb
80%	0,015seg	7,65gb
100%	0,03125seg	7,81gb

En la gráfica se evidencia que el tiempo de ejecución del algoritmo es constante a lo largo de todos los porcentajes de datos. Esto confirma la complejidad constante del algoritmo y muestra la inmensa mejora frente a la implementación anterior.

Requerimiento 5

Para el caso del requerimiento 5, en el reto 1 se pensó en la utilización de una lista, por lo que se vio la necesidad de un ordenamiento con el algoritmo Merge de una lista con la información de todas las obras por sus departamentos, este ordenamiento dejaba las obras en “bloques” por cada uno de los departamentos, pero tenía una complejidad de O(nlogn). Luego de ello, se recorría cada una de las obras en este rango, se les asignaba el costo de transporte a cada una de ellas, y se agregaba simultáneamente a 2 listas que iban a ser posteriormente organizadas de acuerdo con las 2 necesidades del requerimiento. En suma, esta forma de resolver el requerimiento 5 tenía una complejidad temporal general de O(nlogn).

Por otra parte, para el caso de la solución dada para el requerimiento 2 se utilizaron nuevamente los mapas.

```
catalog['Depts'] = mp.newMap(2000,maptype='PROBING',loadfactor = 0.5)
```

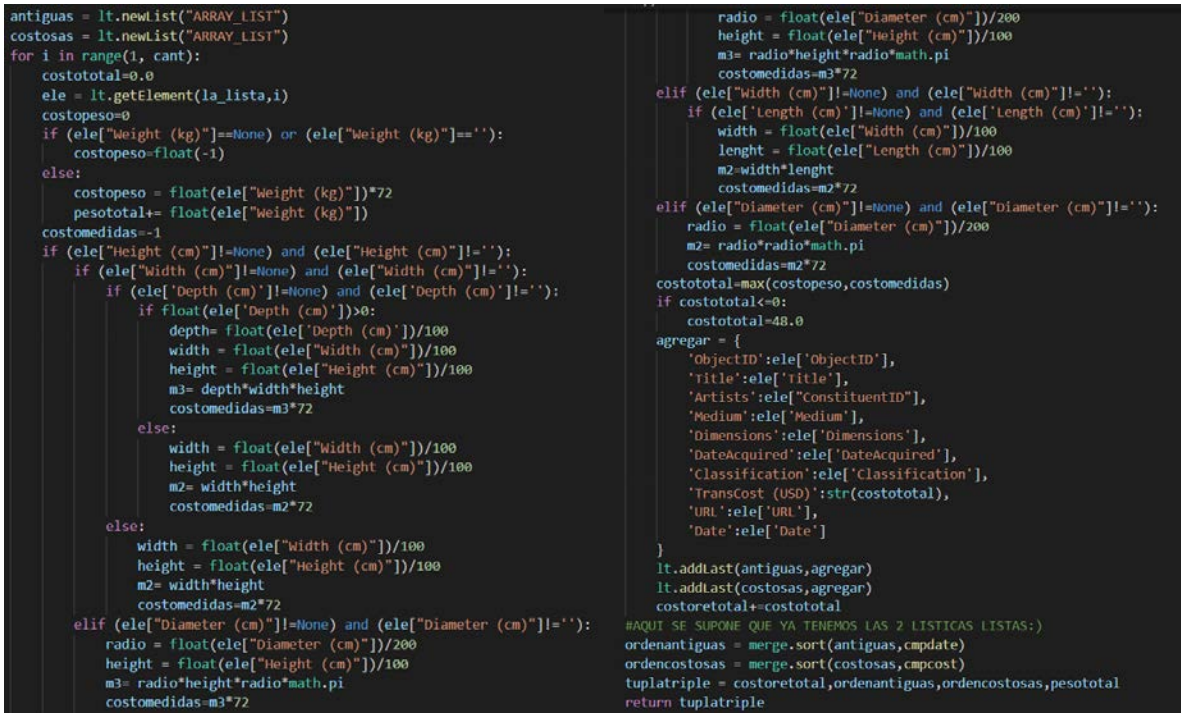
Así las cosas, se creó un nuevo mapa que almacenara toda la información de todas las obras separadas por departamentos, de modo que las llaves de este mapa serían los nombres de los departamentos y los valores serían toda la información de las obras de dicho departamento en una lista.

```
def addToDept(catalog, obra):
    departamento = obra['Department']
    depts = catalog["Depts"]
    if departamento.lower() in (None, "", "unknown"):
        departamento= "unknown"
    existe = mp.contains(depts,departamento)
    if existe:
        parja= mp.get(depts,departamento)
        listica = me.getValue(parja)
    else:
        listica=lt.newList("ARRAY_LIST")
        mp.put(depts,departamento,listica)
    lt.addLast(listica, obra)
```

En consecuencia, para la ejecución del requerimiento, ya no se debía realizar el ordenamiento que le daba peso a la complejidad temporal al algoritmo. En su lugar, la función llama el valor correspondiente a la llave del departamento solicitado por el usuario.

```
def funcionReqCin(catalog, nombre):
    la_lista=mp.get(catalog["Depts"],nombre)
    la_lista=me.getValue(la_lista)
    cant=lt.size(la_lista)
    costoretotal=0.0
    pesototal=0.0
    cant+=1
    antiguas = lt.newList("ARRAY_LIST")
    costosas = lt.newList("ARRAY_LIST")
```

Con esta lista obtenida, se procede a realizar las mismas funcionalidades que para el requerimiento del reto 1: un recorrido de la lista agregándole los precios a cada una de ellas y generando duplicados de cada uno de estos elementos para agregarlos a 2 listas nuevas para su posterior ordenamiento y satisfacción de cada uno de los 2 sub-requerimientos del requerimiento 5.

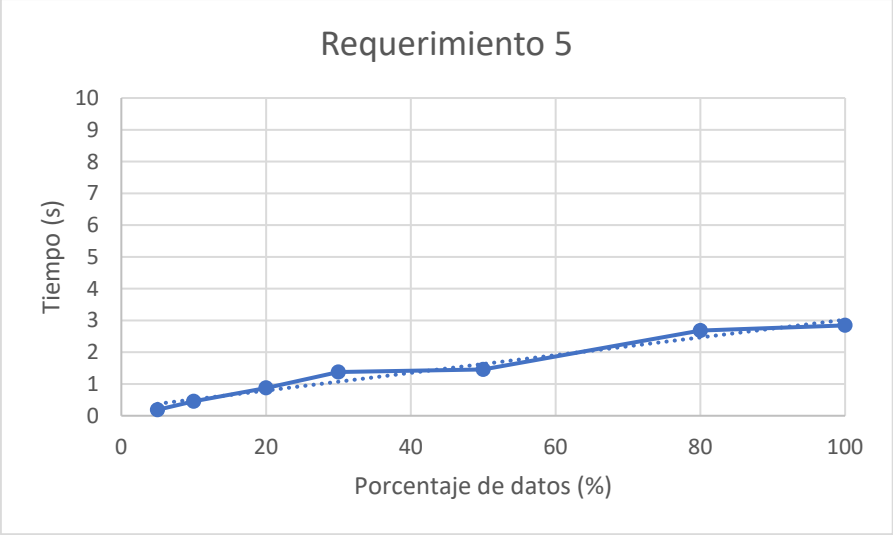


```
antiguas = lt.newList("ARRAY_LIST")
costosas = lt.newList("ARRAY_LIST")
for i in range(1, cant):
    costototal=0.0
    ele = lt.getElement(la_lista,i)
    costopeso=0
    if (ele["Weight (kg)"]==None) or (ele["Weight (kg)"]==''):
        costopeso=float(-1)
    else:
        costopeso = float(ele["Weight (kg)"])*72
        pesototal+= float(ele["Weight (kg)"])
    costomedidas=-1
    if (ele["Height (cm)"]!=None) and (ele["Height (cm)"]!=''):
        if (ele["Width (cm)"]!=None) and (ele["Width (cm)"]!=''):
            if (ele['Depth (cm)']!=None) and (ele['Depth (cm)']!=''):
                if float(ele['Depth (cm)'])>0:
                    depth= float(ele['Depth (cm)'])/100
                    width = float(ele["Width (cm)"])/100
                    height = float(ele["Height (cm)"])/100
                    m3= depth*width*height
                    costomedidas=m3*72
                else:
                    width = float(ele["Width (cm)"])/100
                    height = float(ele["Height (cm)"])/100
                    m2= width*height
                    costomedidas=m2*72
            else:
                width = float(ele["Width (cm)"])/100
                height = float(ele["Height (cm)"])/100
                m2= width*height
                costomedidas=m2*72
        elif (ele["Diameter (cm)"]!=None) and (ele["Diameter (cm)"]!=''):
            radio = float(ele["Diameter (cm)"])/200
            height = float(ele["Height (cm)"])/100
            m3= radio*height*radio*math.pi
            costomedidas=m3*72
    elif (ele["Width (cm)"]!=None) and (ele["Width (cm)"]!=''):
        if (ele['Length (cm)']!=None) and (ele['Length (cm)']!=''):
            width = float(ele["Width (cm)"])/100
            lenght = float(ele["Length (cm)"])/100
            m2=width*lenght
            costomedidas=m2*72
    elif (ele["Diameter (cm)"]!=None) and (ele["Diameter (cm)"]!=''):
        radio = float(ele["Diameter (cm)"])/200
        m2= radio*radio*math.pi
        costomedidas=m2*72
    costototal=max(costopeso,costomedidas)
    if costototal<=0:
        costototal=48.0
    agregar = {
        'ObjectID':ele['ObjectID'],
        'Title':ele['Title'],
        'Artists':ele["ConstituentID"],
        'Medium':ele['Medium'],
        'Dimensions':ele['Dimensions'],
        'DateAcquired':ele['DateAcquired'],
        'Classification':ele['Classification'],
        'TransCost (USD)':str(costototal),
        'URL':ele['URL'],
        'Date':ele['Date']
    }
    lt.addLast(antiguas,agregar)
    lt.addLast(costosas,agregar)
    costoretotal+=costototal
#AQUI SE SUPONE QUE YA TENEMOS LAS 2 LISTICAS LISTAS:)
ordenantiguas = merge.sort(antiguas,cmpdate)
ordencostosas = merge.sort(costosas,cmpcost)
tuplatriple = costoretotal,ordenantiguas,ordencostosas,pesototal
return tuplatriple
```

A pesar de que cada uno de estos ordenamientos tiene una complejidad O(nlogn) y que el ciclo tiene una complejidad temporal de O(n), estas n hacen referencia a las obras del departamento, un número significativamente inferior a las de las obras totales. En conclusión, se puede decir que la complejidad temporal del algoritmo nuevamente sería de O(1).

Pruebas:

Porcentaje (%)	Tiempo (s)	Memoria (%)
5	0,1875	89
10	0,4531	90
20	0,8752	95
30	1,3752	92
50	1,4564	95
80	2,6813	98
100	2,8438	96



MAQUINA 2:

Porcentaje	Tiempo	Memoria
5%	0.406seg	7,06gb
10%	0.625seg	3,94gb
20%	2,5seg	7,27gb
30%	2,14seg	7,38gb
50%	3,82seg	7,32gb
80%	4,8125seg	7,63gb
100%	5,015seg	7,88gb