RETO_002

Camilo Garcia – 201728914 – ca.garciar1@uniandes.edu.co REQ 4

Daniel Vargas – 201822068 – d.vargasm@uniandes.edu.co REQ 3

REQUISITO1: listar cronológicamente los artistas

Para este requisito, se utilizan dos procedimientos. El primer es para obtener una sublista que contenga únicamente la información de los artistas dentro del rango de edad deseado. El segundo utiliza esta sublista como input para hacer un ordenamiento dependiendo en la fecha, y retorna la lista ordenada para obtener los resultados necesarios.

```
75  def sortArtistsByBeginDate(catalog, year1, year2):
76
77      artistsInRange = model.getArtistsInDateRange(catalog, year1, year2)
78      sortedResult = model.sortArtists(artistsInRange)
79      return sortedResult
```

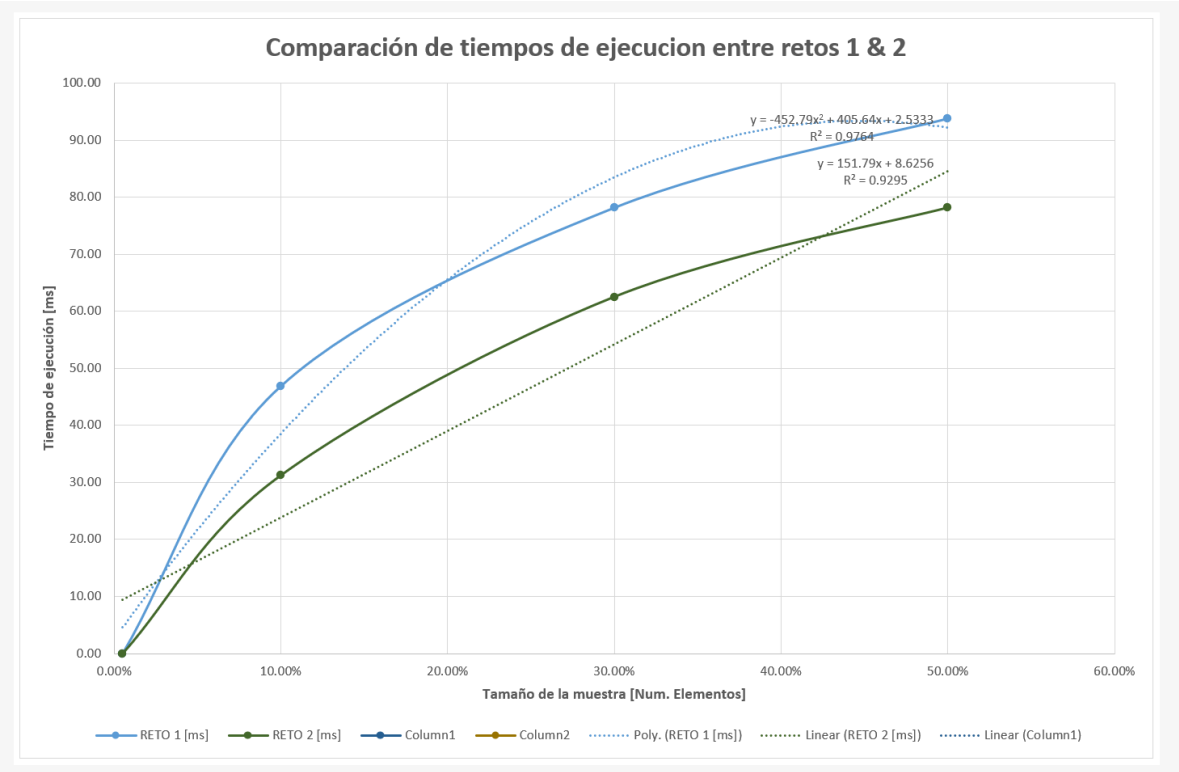
Para la primera parte, se recorren las llaves del mapa 'years', el cual ordena la lista de artistas por año, y si existe un valor asociado a la llave, se compara con el rango de años ingresados. En caso de encontrarse dentro del rango deseado, se añaden los artistas asociados a este año en una lista vacía. Esta operación de comparación tiene un orden de crecimiento O(M), donde M es el tamaño de elementos en la tabla de hash, así estén llenos o no. Aquí podemos ver que, al haber menos años que numero de artistas en total, este recorrido sería más eficiente que el implementado anteriormente, aunque tenga el mismo orden de crecimiento.

Para la segunda parte, se utiliza el algoritmo de ordenamiento MERGE_SORT para ordenar la sublista ingresada. Se eligió este método de ordenamiento ya que tiene el menor orden en el peor de los casos O(nlog(n)).

En el paso final, de buscar datos de las primeras 3 y últimas 3 posiciones de la lista ordenada, el hecho que sea un ARRAY_LIST también hace más eficiente la búsqueda según posición, ya que no se requiere recorrer toda la lista para obtener las ultimas posiciones, como tocaría hacer con la implementación SINGLE_LINKED. Teniendo todo esto en mente, el mayor orden de crecimiento en el peor de los casos para este requisito con nuestra implementación sería O(nlog(n)).

Haciendo cambios en los tamaños de las listas csv, podemos verificar que la implementación en el reto 2 es algo más eficiente que la del reto 1 a la hora de ordenar los datos, aunque tengan un orden de crecimiento similar.

Porcentaje de la muestra [pct]	Tamaño de la muestra (artistas)	RETO 1	RETO 2
small	1948	0.0	0.0
10%	6656	46.875	31.25
30%	10063	78.125	62.5
50%	12137	93.750	78.125



REQUISITO 2: listar cronológicamente las adquisiciones

Para este requisito, se utilizan 2 procedimientos. El primero se utiliza para obtener una sublista de las obras de arte limitadas al rango de fechas indicado. El último ordena la sublista según la fecha de adquisición de las obras.

```
def sortArtworksByBeginDate(catalog, year1, year2):

    artworksInRange = model.getArtworksInDateRange(catalog, year1, year2)
    sortedResult = model.sortArtworks(artworksInRange[0])
    return sortedResult,artworksInRange[1]
```

Para la primera parte, se recorren las llaves del mapa ‘dates’ (el cual ordena la lista de obras por fecha en formato AAAA-MM-DD) y si existe un valor asociado a la llave, se compara con el rango de fechas ingresadas. En caso de encontrarse dentro del rango deseado, se añaden las obras asociadas a este año en una lista vacía. Esta operación de comparación tiene un orden de crecimiento O(M), donde M es el tamaño de elementos en la tabla de hash, así estén llenos o no. Aquí podemos ver que, si hay menos fechas que numero de obras en total, este recorrido sería más eficiente que el implementado anteriormente, aunque tenga el mismo orden de crecimiento.

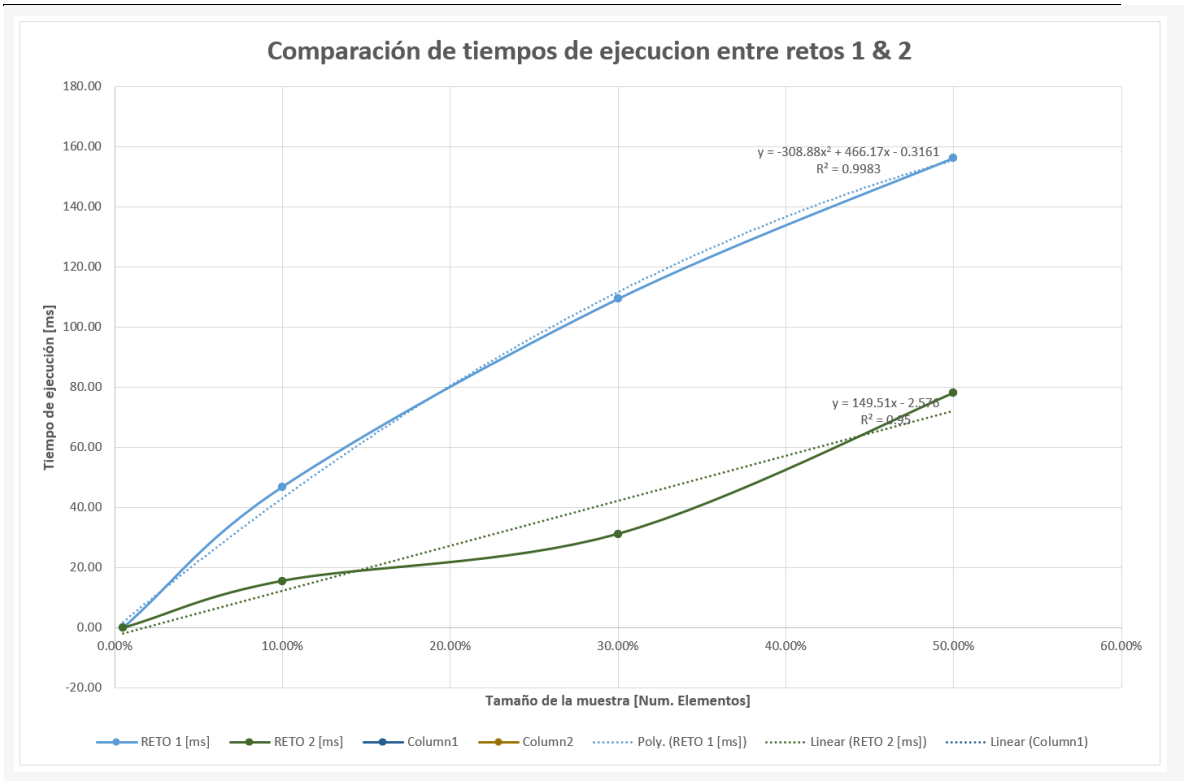
Para obtener el numero de obras que fueron compradas, el orden de crecimiento se reduce respecto al reto anterior, ya que este valor lo guardamos dentro de la estructura del entry de cada fecha a la hora de crear el mapa de ‘dates’, por lo que se ahorra un recorrido por la sublista creada de obras dentro del rango y solo tiene un orden de O(1) por cada fecha confirmada.

Para la segunda parte, se utiliza el algoritmo de ordenamiento MERGE_SORT para ordenar la sublista ingresada. Se eligió este método de ordenamiento ya que tiene el menor orden en el peor de los casos O(mlog(m)).

Para poder mostrar el nombre de los artistas de estas 6 obras, se implemento otra función que ingresara los 'ConstituentID' de las obras al mapa de IDs, y retornara el nombre del artista asociado a ese ID. En compasión al reto anterior, esto es mucho más eficiente, ya que se ahorra un recorrido por la lista de artistas y utiliza un método de búsqueda con complejidad O(1). Teniendo todo esto en mente, el mayor orden de crecimiento en el peor de los casos para este requisito con nuestra implementación sería O(nlog(n)).

Haciendo pruebas de ejecución, cambiando el tamaño de datos, podemos verificar que nuevamente la implementación del reto 2 es más eficiente que la del reto 1, aunque tenga un orden de crecimiento similar.

Porcentaje de la muestra [pct]	Tamaño de la muestra (obras)	Reto 1	Reto2
small	768	0.0	0.0
10%	15008	46.875	15.625
30%	43704	109.375	31.25
50%	71432	156.25	78.125



REQUISITO 3 (Daniel Vargas):

REQUISITO 4 (Camilo García): clasificar las obras por la nacionalidad de sus creadores

En el reto anterior, también se creo una estructura que guardara la información de las obras de arte según nacionalidad al momento de cargar los datos y luego hiciera un sort de la lista de paises

Para este requisito, se hicieron dos procedimientos. Primero, se crea un mapa que contiene los datos de las distintas nacionalidades de los artistas, conteniendo la información de las obras pertenecientes a cada país. Luego, se ordena esta estructura dependiendo del número de obras totales de cada país.

```
def addArtworkNationlities(catalog, artwork):

    constituentids = artwork['ConstituentID']
    artists = ArtistByID_v2(catalog, constituentids)

    for artist in lt.iterator(artists):
        addNationality(catalog, artwork, artist)



# Funciones para creacion de datos

def addNationality(catalog, artwork, artist):
    try:
        nationalities = catalog['nationalities']
        if (artist['Nationality'] != ''):
            pubnationality = (artist['Nationality'])
        else:
            pubnationality = 'Nationality unknown'
        existnationality = mp.contains(nationalities, pubnationality)
        if existnationality:
            entry = mp.get(nationalities, pubnationality)
            nationality = me.getValue(entry)
        else:
            nationality = newNationlity(pubnationality)
            mp.put(nationalities, pubnationality, nationality)
        lt.addLast(nationality['artworks'], artwork)
        nationality['artwork_number']+=1
    except Exception:
        return None
```

```python
def ArtistByID_v2(catalog, constituentids):

    ID_catalog = catalog['artistID']
    ID_list = constituentids.strip("[]").split(", ")
    result = lt.newList('ARRAY_LIST')

    for item in ID_list:
        existsID = mp.contains(ID_catalog, item)
        if existsID:
            entry = mp.get(ID_catalog, item)
            info_artistas = me.getValue(entry)
            lista_artistas = info_artistas['artists']

            for artist in lt.iterator(lista_artistas):
                lt.addLast(result, artist)

    return result
```

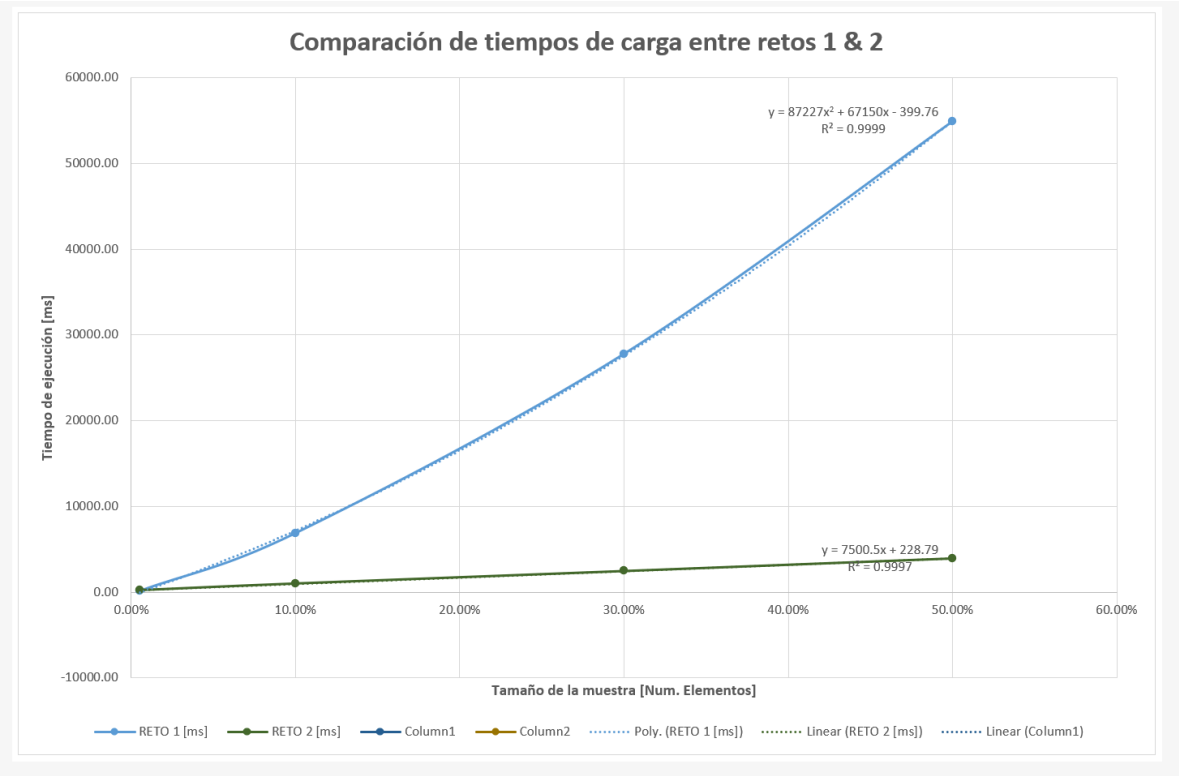
Para la carga de datos, podemos ver que la nueva función para acceder al artista según su ID es mucho más eficiente, ya que permite la búsqueda en complejidad O(1) por cada ID buscado, en vez de la anterior que tenía que recorrer toda la lista de artistas por cada artista de cada obra de arte (O(n)).

Para la segunda parte, se tiene que hacer un sort sobre la estructura de nacionalidades, la cual ya contiene el numero de obras como una pareja llave-valor, utilizando una función de comparación y el algoritmo MERGE SORT. Para esto, es necesario recorrer todas las llaves de la tabla de hash, por lo que la complejidad sería O(m) para una tabla con M posiciones.

Para poder mostrar el nombre de los artistas de 6 obras dentro del país con más obras, se utiliza la misma función de búsqueda en el mapa de IDs, con complejidad de búsqueda de O(1). Teniendo todo esto en mente, el mayor orden de crecimiento en el peor de los casos para este requisito con nuestra implementación sería O(M), al momento de recorrer los datos de la tabla de hash.

Para analizar si el cambio de la implementación del catálogo [‘nationalities´] afectaba los tiempos de carga, se hicieron pruebas comparando los tiempos de carga de todo el catálogo.

Porcentaje de la muestra [pct]	Tamaño de la muestra (obras cargadas)	ARRAY_LIST [ms]	SINGLE_LINKED [ms]
small	768	109.375	234.375
10%	15008	6875.0	1015.625
30%	43704	27796.875	2484.375
50%	71432	54921.875	3968.75



Aquí podemos ver que los tiempos de carga son mucho más eficientes para la implementación en el reto 2, ya que agregar un dato a la estructura tendría una complejidad de O(1), al igual que la comparación entre los constituendID, que antes era O(M*N)

REQUISITO 5: transportar obras de un departamento

Para este requisito se utilizaron tres procedimientos. El primero, genera una sublista de obras de arte usando como criterio de inclusión si el departamento coincide con el input dado por el usuario, simultáneamente calculando el peso y precio de cada elemento en la sublista. El segundo procedimiento hace un sort sobre esta sublista de acuerdo con la fecha de las obras. El tercer procedimiento hace un sort sobre la sublista de acuerdo con el costo de transporte calculado en el primer procedimiento.

```
94  def sortArtworksByDeparment(catalog, department):
95
96      artowrksInDeptResult = model.getArtworksInDeparment(catalog, department)
97      artworksInDept = artowrksInDeptResult[0]
98      sortedResultByDate = model.sortArtworksInDeptByDate(artworksInDept)
99      sortedResultByTransportCost = model.sortArtworksInDeptByTransportCost(artworksInDept)
100     return artowrksInDeptResult, sortedResultByDate, sortedResultByTransportCost
```

Para la primera parte, se utiliza el departamento ingresado por el usuario para obtener la información asociada a la llave que coincide con el departamento dentro del mapa de ‘departments’. Aquí, se puede hacer referencia a la lista de obras guardadas en el departamento que se esta buscando con complejidad de O(1). Luego, recorre las obras dentro de dicho departamento y calcula su peso y costo, añadiendo estos datos a un contador de totales, también de complejidad O(1) por cada obra. Respecto a la implementación del reto 1, podemos ver que la

complejidad para obtener la sublista se reduce significativamente, de O(n) a O(1), aunque en el peor de los casos igual tendría una complejidad de O(n) si fuera necesario calcularle el costo a todas las obras.

Para la segunda parte y tercera parte, se utiliza el algoritmo de ordenamiento MERGE_SORT para ordenar la sublista ingresada. Se eligió este método de ordenamiento ya que tiene el menor orden en el peor de los casos O(mlog(m)).

Para poder mostrar el nombre de los artistas de 5 obras, se utiliza la misma función de búsqueda en el mapa de IDs, con complejidad de búsqueda de O(1).

Teniendo todo esto en mente, el mayor orden de crecimiento en el peor de los casos para este requisito con nuestra implementación sería O(mlog(m)) por los ordenamientos del MERGE_SORT.

Para verificar esto, hicimos pruebas de ejecución sobre archivos de distintos tamaños, cambiando el tipo de implementación. Aquí, podemos ver que el tiempo de ejecución es casi el mismo para ambas implementaciones. Esto puede ser ya que, aunque se ahora un recorrido de O(n), igual tiene que calcular los costos y pesos para cada obra dentro del departamento, al igual que llevar a cabo dos procesos de sort, que tienen la misma complejidad que la obtenida en el primer reto.

Porcentaje de la muestra [pct]	Tamaño de la muestra (obras cargadas)	ARRAY_LIST [ms]	SINGLE_LINKED [ms]
small	768	15.625	15.625
10%	15008	406.25	375.0
30%	43704	1281.25	1312.5
50%	71432	2187.5	2234.375