Estructuras de Datos y Algoritmos: 2022-1

Miguel Arturo Reina Rocabado, 202014739, m.reina@uniandes.edu.co

Sergio Andrés Oliveros Barbosa, 202123159, s.oliverosb@uniandes.edu.co

A lo largo del Reto 2 la estructura de datos que ese utilizó para los mapas fue “Linear Probing”, para las listas se utilizó “Array List”.

NOTA: para el desarrollo de los requerimientos, decidimos minimizar los tiempos tanto de cada requerimiento como el tiempo de carga. Esto, porque a la percepción del usuario no podrá diferenciar entre 0.5ms o 50 ms y sí marca una diferencia considerable esperar 4-5 minutos que esperar 1 minuto por la aplicación. Así pues, nuestro desarrollo busca un equilibrio entre ambos tiempos.

Por otro lado, los requerimientos 1, 3 y 5 se realizaron en la máquina 1, mientras que los requerimientos 2, 4 y 6 se realizaron en la máquina 2.

	Máquina 1	**Máquina 2**
Procesadores	AMD Ryzen 7 5800H - 3.20 GHz	AMD Ryzen 5 3500U 2.10Ghz
Memoria RAM (GB)	8.00 GB, 7.35GB usable	8.00GB, 5.95 usable
Sistema Operativo	Windows 11 - 64bits	Windows 11 – 64 bits

Requerimiento 1: Examinar los álbumes en un año de interés.

Análisis carga del requerimiento:

```
#================[R1]================
album["release_date"] = str_to_date(album["release_date"], album["release_date_precision"])
anio_album = str(album["release_date"])[:4]
exists_year = mp.get(catalog["albums_per_year"], anio_album)

if exists_year:
    insort_right_r1(exists_year["value"], album, lo=1,hi=None)

else:
    mp.put(catalog["albums_per_year"], anio_album, lt.newList("ARRAY_LIST"))
    exists_year = mp.get(catalog["albums_per_year"], anio_album)
    lt.addLast(exists_year["value"], album)
```

Durante la carga de datos se crea un mapa donde las llaves son años y los valores son listas que contienen los álbumes lanzados durante el año en cuestión. Crear y agregar llaves este mapa tiene una complejidad temporal de O(1), garantizamos esto debido a que el número de elementos que espera guardar el mapa fue establecido con antelación para asegurarnos de este comportamiento. De igual manera la mayoría de las demás líneas y funciones que se usan también tienen una complejidad de O(1).

```
def insort_right_r1(lista, album, lo=1, hi=None):
    """Insert item x in list a, and keep it sorted assuming a is sorted.

    If x is already in a, insert it to the right of the rightmost x.

    Optional args lo (default 0) and hi (default len(a)) bound the
    slice of a to be searched.
    """
    if hi is None:
        hi = lt.size(lista) + 1
    while lo < hi:
        mid = (lo+hi)//2
        if album["release_date"] < lt.getElement(lista, mid)["release_date"]:
            hi = mid

        else:
            lo = mid+1
    lt.insertElement(lista, album, lo)
```

Sin embargo, consideramos importante tener en cuenta la complejidad de la función *Insort()*. Esta lo que hace es agregar elementos a una lista de manera ordenada según un único criterio al encontrar su lugar adecuado al hacer búsqueda binaria. Por lo que para agregar un nuevo álbum la complejidad que se le asocia es O(log(n)), donde n hace referencia al número de álbumes en una lista sobre la cual vaya a agregar un elemento.

Funciones asociadas a la carga del Requerimiento 1:

```
def str_to_date(string, precision):
    if precision == "year":
        return dt.strptime(string,"%Y").date()
    elif precision == "month":
        string= string[:4] + "19" + string[4:]
        return dt.strptime(string, "%b-%Y").date()
    return dt.strptime(string, "%Y-%m-%d").date()
```

Análisis ejecución del Requerimiento:

```
def answer_r1(catalog, albumes_del_anio):
    """
    Función principal del requerimiento 1
    """
    lista_anio = mp.get(catalog["model"]["albums_per_year"], albumes_del_anio)

    if lista_anio == None:
        return None, None, None
    lista_anio = mp.get(catalog["model"]["albums_per_year"], albumes_del_anio)["value"]
    num_albums = mp.size(lista_anio)
    answer_list_r1 = tomar_primeros_ultimos(lista_anio)


    asc_artists = lt.newList('ARRAY_LIST')
    for album in lt.iterator(answer_list_r1):
        asc_art = find_asc_artist(catalog, album)
        lt.addLast(asc_artists, asc_art)

    return answer_list_r1, asc_artists, num_albums
```

Entradas: Catalogo que contiene los mapas cargados al inicio del programa y el año que el usuario desea ingresar.

Salidas: Una lista de 6 elementos que contiene los 3 primeros álbumes lanzados en el año ingresado cómo parámetro por el usuario y también los últimos 3. Una lista que contiene los

nombres de los artistas que compusieron los álbumes anteriormente mencionados. Y un número que representa la cantidad total de álbumes lanzados en el año que el usuario ingreso.

Operaciones:

1) Se hace *mp.get()* del año ingresado por el usuario para determinar si el año existe. Esta operación tiene una complejidad de **O(1)**. En caso de que no se haye el año ingresado se retorna None y la función culmina con una complejidad de **O(1)**.

O(1)

2) Una vez obtenemos la lista podemos seleccionar los primeros 3 y los últimos 3 elementos de la misma, pues *insort()* ordenó la lista exclusivamente según la fecha durante la carga, y cómo la lista que se usó fue del tipo "Array List" el obtener estos 6 elementos se hace un tiempo constante.

O(1)

```
def find_asc_artist(catalog, album):
    id_artist = album['artist_id']
    artist = mp.get(catalog['model']['id_artists'], id_artist)
    if artist:
        return artist['value']['name']
    else:
        return 'Unknown'
```

3) Finalmente se obtienen los nombres de los artistas que compusieron los 6 álbumes seleccionados anteriormente. Para esto simplemente se recorren los álbumes, se seleccionan los Ids de sus artistas y se usan estos Ids cómo llaves un mapa dónde las llaves son los Ids de los artistas y los valores los diccionarios que contienen su información de estos. En otras palabras, se accede a un mapa 6 veces, operación cuya complejidad temporal es constante.

O(1)

Complejidad Final:

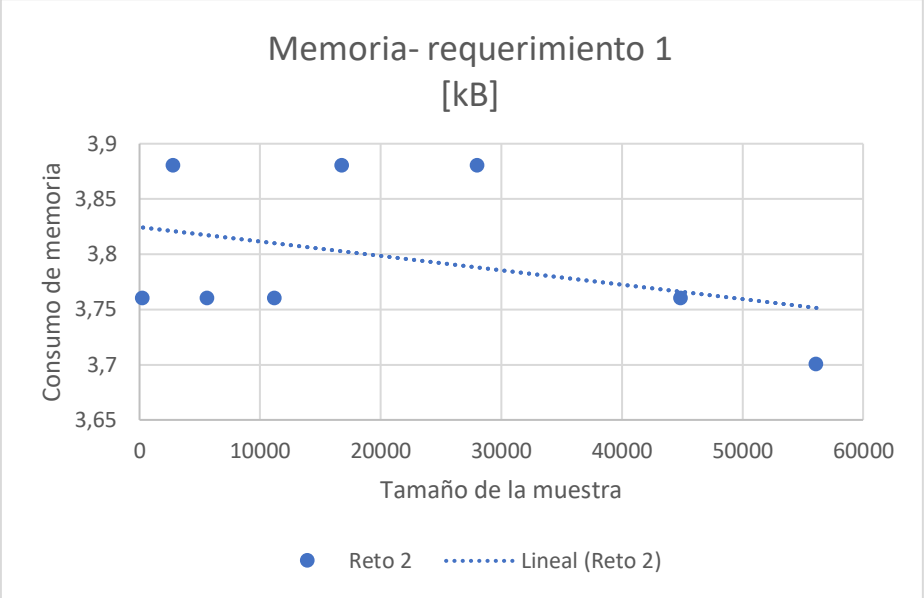
O(1+1+1) = O(1)

En conclusión el Requerimiento 1 tiene una complejidad de O(1).

Toma de datos: tiempos de ejecución y consumo de memoria.

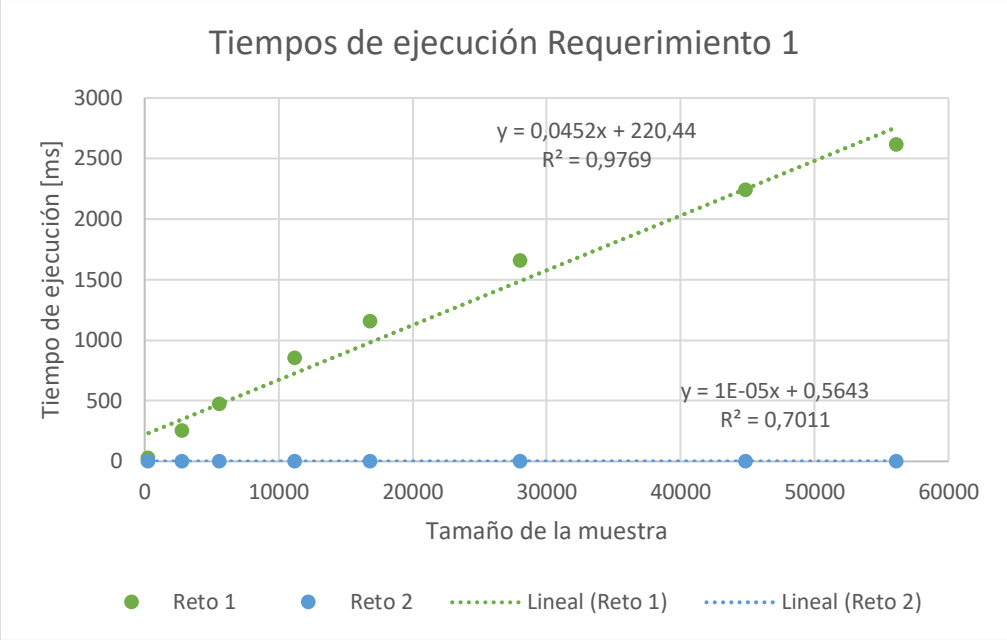
		Reto 1	Reto 2	Reto 2
Porcentaje de la muestra [pct]	Tamaño de la muestra (ARRAY_LIST)	Tiempo de ejecución (Merge Sort) [ms]	Tiempo de ejecución [ms]	Memoria [kB]
0.50%	281	26.24	0.52	3.76
5.00%	2806	254.72	0.51	3.88

10.00%	5613	470.25	0.86	3.76
20.00%	11226	849.42	0.71	3.76
30.00%	16838	1153.8	0.64	3.88
50.00%	28064	1654.67	0.92	3.88
80.00%	44902	2238.94	0.84	3.76
100.00%	56128	2616.16	1.35	3.7



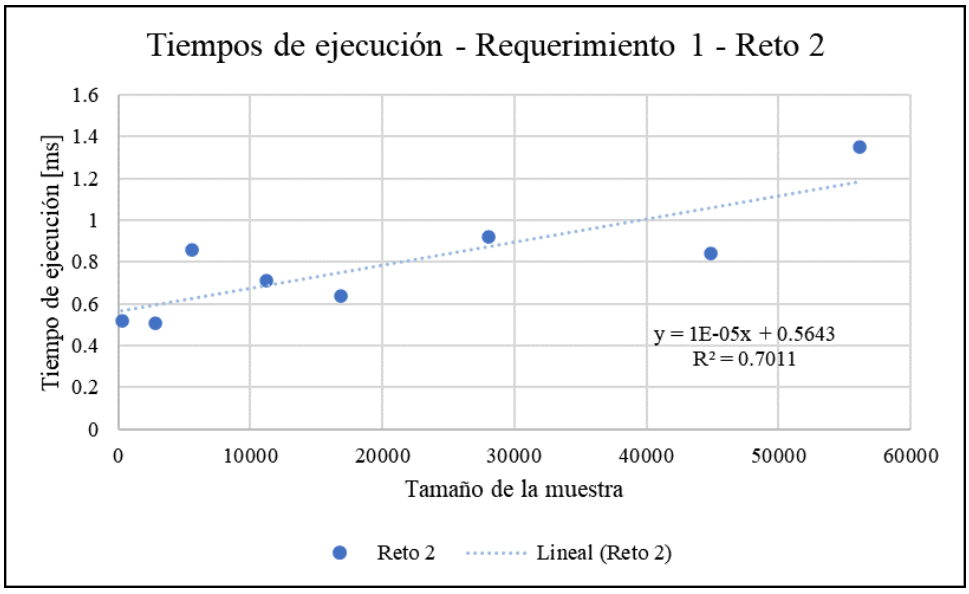
Gráfica 1.1: comparación de consumo de memoria entre reto1 y reto2 para el requerimiento 1.

Para el requerimiento 1 esperábamos que la memoria fuese constante, sin embargo, cómo es posible evidenciar en la gráfica, esto no fue así. Observamos una pequeña variabilidad, de menos de 1 kB en los resultados la cual al parecer no depende del tamaño de la muestra. Por lo que asociamos esta variabilidad posiblemente a especificaciones técnicas del computador sobre el cual se realizaron las pruebas. No obstante, a grandes rasgos la memoria no se desvía mucho entre sus valores, por lo que consideramos que en el gran esquema de las cosas la memoria sí se mantuvo constante.



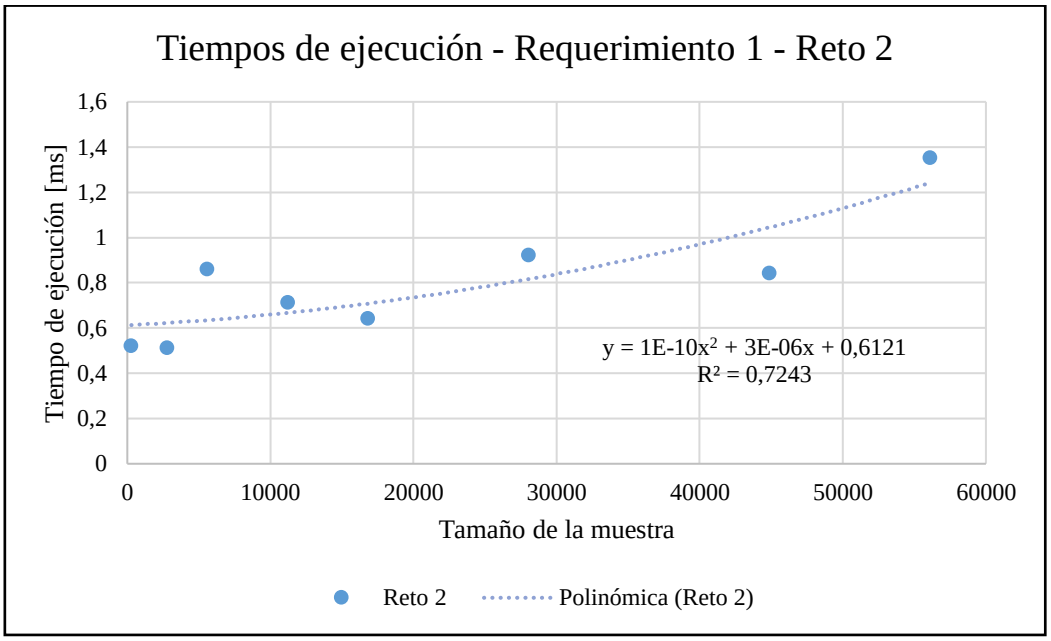
Gráfica 1.2: comparación de tiempos de ejecución entre reto1 y reto2 para el requerimiento 1.

La gráfica confirma nuestras expectativas de que los tiempos de ejecución en Reto 2 deberían ser ordenes de magnitud menores a las del Reto 1, ya que en el Reto 1 la complejidad temporal del requerimiento 1 era de O(nlog(n)), siendo n un valor cercanamente relacionado al tamaño de la muestra. Mientas que en el Reto 2, el Requerimiento 1 tiene una complejidad temporal de O(1) es decir esperábamos que todas nuestras mediciones de tiempo fusen iguales. Sin embargo, al revisar la tabla de datos notamos una variación entre los valores iniciales y finales de alrededor 0.8 ms. Por lo que se presenta una relación directa entre el tamaño de la muestra y el tiempo de ejecución.



Gráfica 1.3: tiempos de ejecución para el requerimiento 1, ajuste lineal.

Sin embargo, al realizar un análisis más directo sobre los datos obtenidos, vemos que a pesar de que no se evidencie un comportamiento constante, tampoco vemos que el R^2 de la regresión lineal sea el más adecuado.



Gráfica 1.4 tiempos de ejecución para el requerimiento 1, ajuste cuadrático.

Y si realizamos una regresión cuadrática para verificar que el comportamiento real no fuese $O(n^2)$, vemos que esta regresión también muestra una correlación regular. Por lo que en este caso vemos que el crecimiento de los datos no se debe al algoritmo utilizado.

Realizando un análisis más profundo sobre la naturaleza de nuestro requerimiento, notamos que desde un inicio nosotros otorgamos un tamaño de tabla capaz de servir para todos los tamaños de archivos que íbamos a manejar. Sin embargo, el número de llaves aumenta entre archivo y archivo. Por lo que notamos que el factor de carga de nuestra tabla (# de llaves / tamaño de la tabla) aumenta con el tamaño de los archivos utilizado. Es decir, a medida que aumenta el archivo es más probable que ocurran más colisiones en nuestra tabla, haciendo que la complejidad de hacer *mp.get()* no se mantenga en O(1). Por ende, las colisiones en la tabla y la variabilidad de nuestros computadores terminan causando un aumento en los tiempos de ejecución de este requerimiento. Sin embargo, tienen un impacto tan minúsculo, aspecto que se evidencia en los coeficientes de las líneas de tendencia, que no es significativo para nuestro análisis de complejidad. A parte de todo esto, tenemos que tener en cuenta variaciones causa de otros programas durante la ejecución, especialmente pues estos tiempos son muy pequeños (1.5 ms máximo).

Requerimiento 2: Encontrar los artistas por popularidad.

Resuelto por: Miguel Arturo Reina Rocabado

Análisis carga del requerimiento:

```
#================[R2]=================
art_popularity = str(artist['artist_popularity'])
exist_pop = mp.get(catalog['artists_per_popularity'], art_popularity)

if exist_pop:
    insort_right_r2(exist_pop['value'], artist, lo = 1, hi = None)
else:
    mp.put(catalog['artists_per_popularity'], art_popularity, lt.newList('ARRAY_LIST'))
    exists_pop = mp.get(catalog["artists_per_popularity"], art_popularity)
    lt.addLast(exists_pop["value"], artist)
```

Para la carga de datos, decidimos crear un TAD map que tenga como llaves la popularidad del artista (como entero) y como valor un TAD lista con los artistas de esta popularidad. Insertamos con insort_right que inserta ordenadamente y quedan ordenados los artistas por el criterio principal (followers)

```
def insort_right_r2(lista, artist, lo=1, hi=None):
    """Insert item x in list a, and keep it sorted assuming a is sorted.

    If x is already in a, insert it to the right of the rightmost x.

    Optional args lo (default 0) and hi (default len(a)) bound the
    slice of a to be searched.
    """
    if hi is None:
        hi = lt.size(lista) + 1
    while lo < hi:
        mid = (lo+hi)//2
        if artist["followers"] > lt.getElement(lista, mid)["followers"]:
            hi = mid
        else:
            lo = mid+1
    lt.insertElement(lista, artist, lo)
```

Análisis de complejidad temporal del Requerimiento:

Entradas: popularidad de los artistas (parte entera).

Salidas: 1. Número de artistas con la popularidad ingresada.

2. TAD lista de artistas ordenados por popularidad y nombre del artista (retornar 3 primeros y 3 últimos).

Operaciones:

```
def answer_r2(catalog, inp_popularity):

    lista_popu = mp.get(catalog['model']['artists_per_popularity'], str(inp_popularity))
    if lista_popu == None:
        return None, None, None

    lista_popu = mp.get(catalog['model']['artists_per_popularity'], str(inp_popularity))['value']
    num_artists = lt.size(lista_popu)
    lista_popu_ord = insertion.sort(lista_popu, cmpR2)
    answer_list_r2 = tomar_primeros_ultimos(lista_popu_ord)

    asc_tracks = lt.newList('ARRAY_LIST')
    for artist in lt.iterator(answer_list_r2):
        asc_track = find_asc_track(catalog, artist)
        lt.addLast(asc_tracks, asc_track)
```

Inmediatamente se selecciona la opción 2, el programa accede al map artist_per_popularity con la llave ingresada y retorna el TAD lista con los artistas con esta popularidad (lo demás es para contención de errores, no se encontró la llave por ejemplo). La complejidad de este paso es:

O(1)

Luego, se toma el valor de esta pareja llave, valor:

O(1)

Se accede al tamaño de este TAD lista:

O(1)

Realizamos *insertion.sort()* ya que necesitabamos un algoritmo adaptativo ya que ordenamos por el criterio principal con *insort()* y necesitamos ahora solo solucionar los conflictos (artistas con la misma cantidad de followers). Por pruebas, es poco probable que dos artistas con la misma popularidad tengan la misma cantidad de seguidores, entonces *insertion.sort()* sigue su mejor caso, una lista ordenada por lo que la complejidad es de aprox.

O(artistas por popularidad dada)

```
def cmpR2(artist1, artist2):
    menor = True
    if artist1['followers'] < artist2['followers']:
        menor = False
    elif artist1['followers'] == artist2['followers']:
        if artist1['name'] < artist2['name']:
            menor = False
    return menor
```

Finalmente, tomamos los tres primeros y tres últimos artistas de la lista de respuesta, como nuestra lista es ‘ARRAY_LIST’ podemos acceder a cada uno en un tiempo constante y tomar estos es independiente del tamaño de artistas, por lo que su complejidad es de:

O(1)

```
def find_asc_track(catalog, artist):
    id_track = artist['track_id']
    artist = mp.get(catalog['model']['id_tracks'], id_track)
    if artist:
        return artist['value']['name']
    else:
        return 'Unknown'
```

Finalmente, buscamos las canciones asociadas a cada uno de estos 6 artistas, como creamos un TAD map con llave artist_id y como valor la ref. a la canción, los tiempos son independientes a la cantidad de artistas por popularidad (6 búsquedas siempre) su complejidad es

O(1)

Finalmente, retornamos cada uno de los datos pedidos. Por lo que la complejidad final del requerimiento 2 esperamos que sea…

O(1) + O(1) + O(1) + O(artistas por popularidad) + O(1) + O(1)

Que es igual a

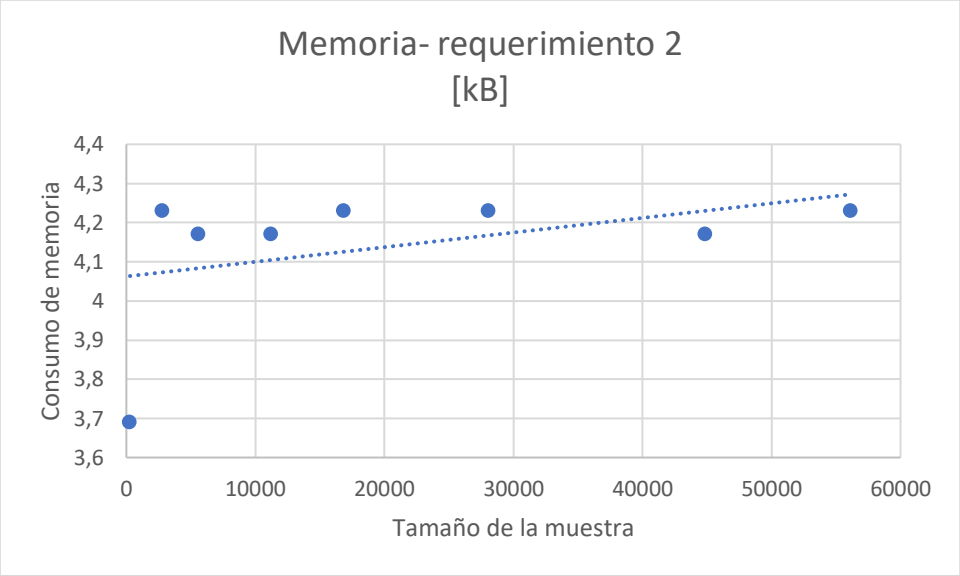
O(artistas por popularidad) o O(n) para n = artistas por popularidad dada

Toma de datos: tiempos de ejecución y consumo de memoria.

		Reto 1	Reto 2	Reto 2
Porcentaje de la muestra [pct]	Tamaño de la muestra	Tiempo de ejecución (Merge Sort) [ms]	Tiempo de ejecución [ms]	Memoria [kB]
0.50%	281	23.62	0.35	3.69
5.00%	2806	257.69	0.42	4.23
10.00%	5613	504.48	0.42	4.17
20.00%	11226	931.06	0.47	4.17
30.00%	16838	1315.26	0.47	4.23
50.00%	28064	1983.51	0.51	4.23
80.00%	44902	2805.33	0.62	4.17
100.00%	56128	3281.64	0.77	4.23

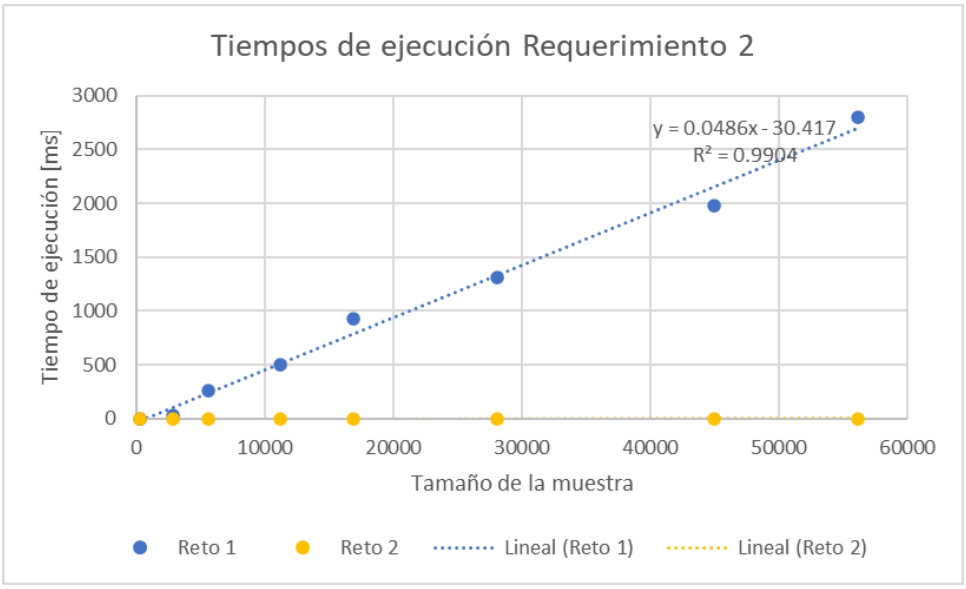
Tabla 2: tiempos de ejecución y consumo de memoria, reto1 y reto2 para el requerimiento 2.

En primer lugar, la memoria,



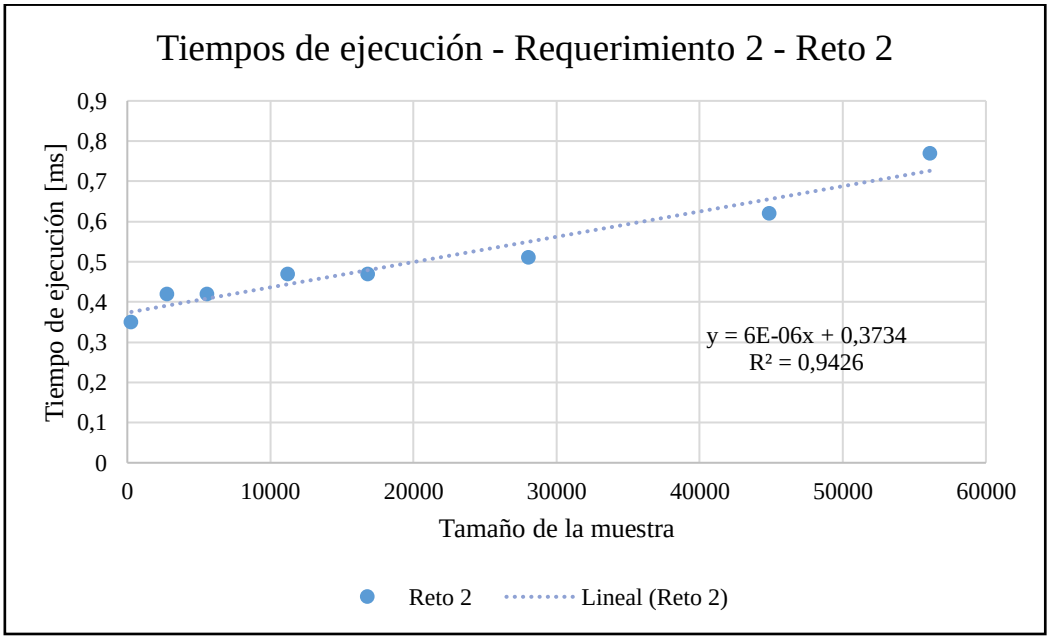
Gráfica 2.1: comparación de consumo de memoria entre reto1 y reto2 para el requerimiento 2

Para la memoria, podemos ver que se conserva alrededor de 4 kB, a excepción para -small. Estas constantes en memoria las podemos explicar porque insertion.sort() es un algoritmo in-place y para -small interpretamos que es por la cantidad de artistas (menos de 6) que se encuentran dada la popularidad y por tanto la lista creada de respuesta tiene menor consumo de memoria.

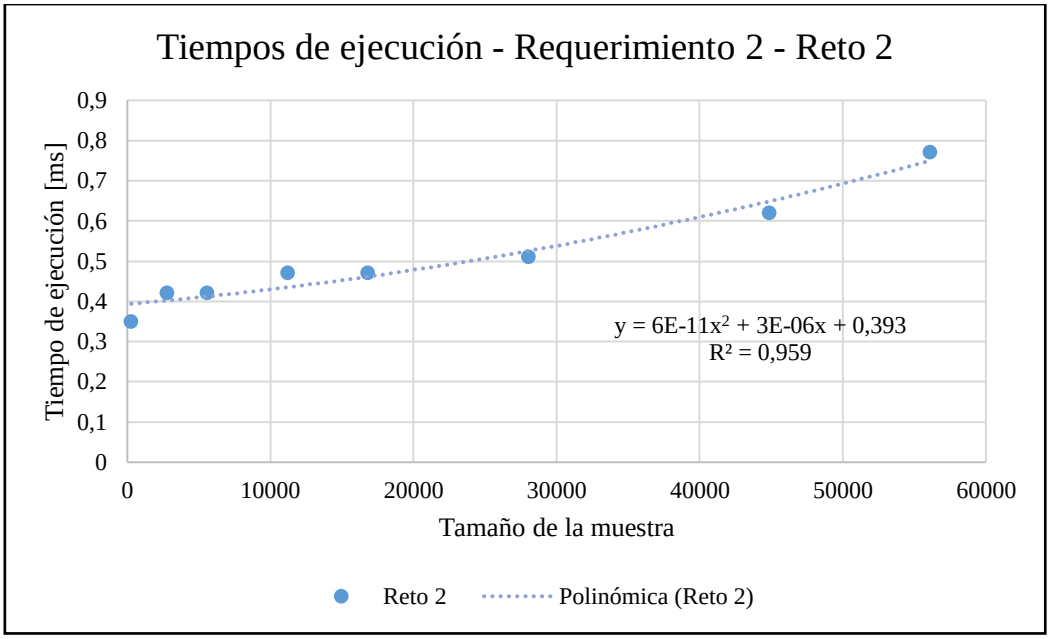


Gráfica 2.2: comparación de tiempos de ejecución entre reto1 y reto2 para el requerimiento 2.

En la gráfica 2.2 es evidente la disminución de tiempos de ejecución del R2 a comparación del R1. Desde 10pct en el R1 solo el requerimiento 2 se demoraba ya un segundo, mientras que con nuestra nueva implementación logramos que siempre fuera menor a 1 ms. Con respecto a la complejidad, para el R1 la complejidad determinamos como **O(n*log n)** con n con todos los artistas, a comparación del R2 con O(m) con m para los artistas dada una popularidad, tanto la expresión matemática que lo describe es menor como la cantidad de elementos (n y m) es menor en el R2 lo cual explica sus tiempos mucho más eficientes.



Gráfica 2.3: Tiempos de ejecución requerimiento 2, ajuste lineal.



Gráfica 2.4: Tiempos de ejecución requerimiento 2, ajuste cuadrático.

Ahora, con respecto a la complejidad observada para el requerimiento 2, podemos hacer tanto la regresión lineal como cuadrática para los datos. Para esta última, el valor de R cuadrado es ligeramente más alto que la lineal. Sin embargo, es común que al aumentar el grado de la ecuación polinómica aumente también el R cuadrado. Este comportamiento entre lineal y cuadrática lo podemos interpretar como una combinación entre el mejor caso de insertion.sort() y el peor caso de insertion.sort() dado por las coincidencias en criterios iniciales iguales. Para poder concluir, determinamos que el coeficiente del factor al cuadrado es 6E-11 muy pequeño, por lo que no debería ser tomado en cuenta y entonces observaríamos una complejidad lineal observada vs. una complejidad lineal esperada.

Requerimiento 3: Encontrar las canciones por popularidad.

Resuelto por: Sergio Andrés Oliveros Barbosa

Análisis carga del requerimiento:

```
#================[R3]================

track_rating = str(round(float(track["popularity"])))
exist_track_rating = mp.get(catalog["tracks_by_popularity"], track_rating)
if exist_track_rating:
    insort_right_r3(exist_track_rating["value"], track, lo=1, hi=None)

else:
    mp.put(catalog["tracks_by_popularity"],track_rating, lt.newList("ARRAY_LIST"))
    exists_track_rating = mp.get(catalog["tracks_by_popularity"], track_rating)
    lt.addLast(exists_track_rating["value"], track)
```

Para el requerimiento 3 se crea un mapa donde las llaves son las popularidades de las canciones y los valores son listas que contienen las canciones con una misma popularidad. La operación más costosa durante este proceso es hacer *Insort()* por lo que durante la carga el Requerimiento

3 tiene una complejidad de O(n), n haciendo referencia al número de canciones con una misma popularidad.

```
def insort_right_r3(lista, track, lo=1, hi=None):
    """Insert item x in list a, and keep it sorted assuming a is sorted.

    If x is already in a, insert it to the right of the rightmost x.

    Optional args lo (default 0) and hi (default len(a)) bound the
    slice of a to be searched.
    """
    if hi is None:
        hi = lt.size(lista) + 1
    while lo < hi:
        mid = (lo+hi)//2
        if float(track["duration_ms"]) > float(lt.getElement(lista, mid)["duration_ms"]):
            hi = mid
        else:
            lo = mid+1
    lt.insertElement(lista, track, lo)
```

Análisis ejecución del Requerimiento:

```
def answer_R3(catalog, inp_track_popularity):
    list_tracks_with_popularity = mp.get(catalog["model"]["tracks_by_popularity"], inp_track_popularity)
    if list_tracks_with_popularity == None:
        return None, None, None, None

    list_tracks_with_popularity =  list_tracks_with_popularity["value"]
    list_tracks_with_popularity_sort = insertion.sort(list_tracks_with_popularity, cmpR3)

    tam_list = lt.size(list_tracks_with_popularity_sort)

    list_resp_R3 = tomar_primeros_ultimos(list_tracks_with_popularity_sort)

    names_albums = lt.newList("ARRAY_LIST")

    names_artists = lt.newList("ARRAY_LIST")
    a = 1
    for track in lt.iterator(list_resp_R3):

        id_album = track["album_id"]
        exist_album = mp.get(catalog["model"]["id_albums"], id_album)

        if exist_album:
            lt.addLast(names_albums, exist_album["value"]["name"])
        else:
            lt.addLast(names_albums, "Not Found")

        artists_ids = track["artists_id"].strip("[").strip('"').strip("]").strip('[').replace("'","").split(",")
        names = lt.newList("ARRAY_LIST")
        for id in artists_ids:
            id = id.strip(" ")
            exists_name = mp.get(catalog["model"]["id_artists"], id)
            if exists_name:
                lt.addLast(names,exists_name["value"]["name"])
            else:
                lt.addLast(names,"Not Found")
        lt.addLast(names_artists, names)
    return tam_list, list_resp_R3, names_albums, names_artists
```

Entradas: Catalogo que contiene los mapas cargados al inicio del programa y la popularidad ingresada por el usuario.

Salidas: El número de canciones que contienen la popularidad ingresada por el usuario. Las primeras 3 y las ultimas 3 canciones dentro de la lista de canciones ordenadas por el criterio compuesto. Los nombres de los álbumes a los que pertenecen las canciones seleccionadas en el punto anterior. Y los artistas asociados, nuevamente, a las canciones previamente seleccionadas.

Operaciones:

1) Se hace *mp.get() usando* cómo llave la popularidad dada por el usuario. En caso de no encontrarse el valor correspondiente se retorna None.

O(1)

```
def cmpR3(track1, track2):
    mayor = False
    if float(track1["duration_ms"]) == float(track2["duration_ms"]):
        if track1["name"] > track2["name"]:
            mayor = True
    elif float(track1["duration_ms"]) > float(track2["duration_ms"]):
        mayor = True
    return mayor
```

2) Si se obtuvo la lista de canciones, se corre el algoritmo *Insertion Sort()* ya que la lista si se ordena con *insort()* solamente se ordena por un criterio, por lo que resulta necesario correr otro algoritmo de ordenamiento para incluir el segundo criterio de ordenamiento, nombre. Sin embargo, debido a que la lista obtenida está casi ordenada, *Insertion Sort()* resulta ideal para terminar de ordenar la lista, pues debido a su carácter adaptativo es capaz de recorrer la lista en un tiempo que el usuario no es capaz de notar. Esto quiere decir que se tiene aproximadamente el mejor caso de *Insertion Sort()* donde su complejidad depende del número de canciones en la lista solicitada por el usuario.

O (Núm. Tracks Lista Usuario)

3) Una vez con la lista ordenada se procede a seleccionar las primeras 3 y las últimas 3 canciones, acción que al estar implementada sobre una lista tipo “Array List” se hace en un tiempo constante.

O(1)

4) Finalmente, a cada una de las 6 canciones seleccionadas se le encuentra el álbum al que pertenecen y los artistas asociados mediante el uso de mapas creados durante la carga de datos. Por lo que el completar estas operaciones se hace en un tiempo constante.

O(1)

Complejidad Final:

O(1 + Núm. Tracks Lista Usuario + 1 +1) = **O(Núm. Tracks Lista Usuario)**

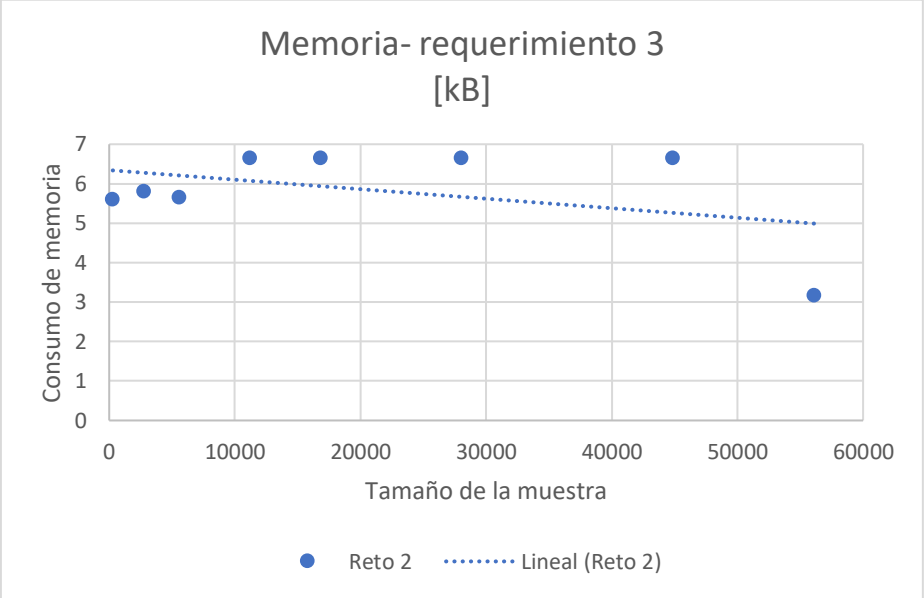
Que es aproximable a la complejidad **O(n).**

Es verdad que queda una complejidad de O(n), sin embargo, debido a la cantidad de datos y la distribución de los mismos, se garantiza que el n es bastante pequeño al punto en que el usuario no es capaz de notar la diferencia entre una complejidad de **O(1)** y **O(n).**

Toma de datos: tiempos de ejecución y consumo de memoria.

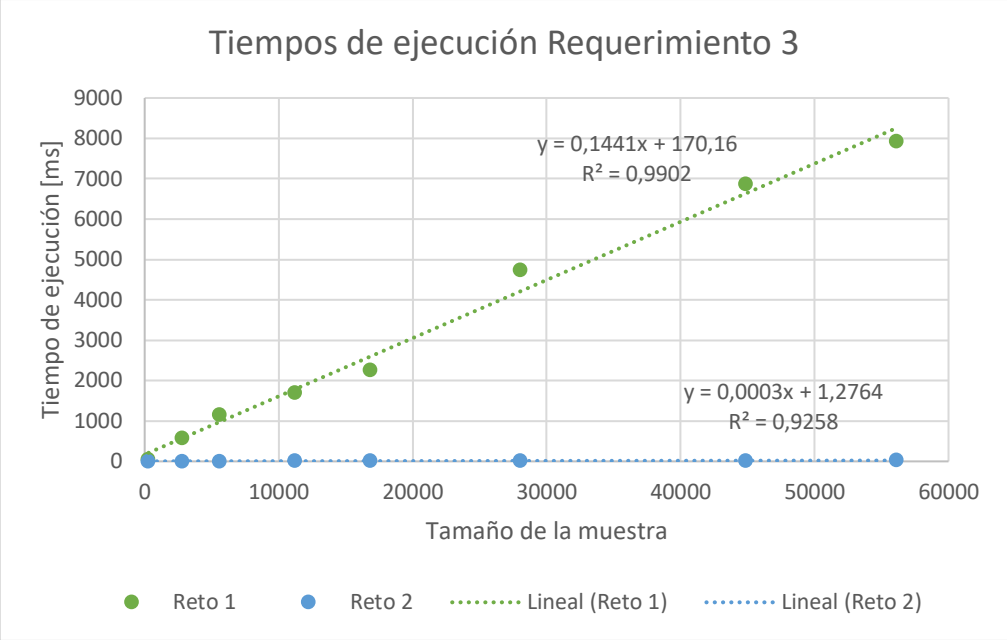
Reto 1	Reto 2	Reto 2

Porcentaje de la muestra [pct]	Tamaño de la muestra (ARRAY_LIST)	Tiempo de ejecución (Merge Sort) [ms]	Tiempo de ejecución [ms]	Memoria [kB]
0.50%	281	50.66	1.91	5.59
5.00%	2806	570.62	3.18	5.8
10.00%	5613	1159.55	2.14	5.64
20.00%	11226	1700.39	7.12	6.64
30.00%	16838	2262.55	6.2	6.64
50.00%	28064	4732.36	9.08	6.64
80.00%	44902	6860.87	13.83	6.64
100.00%	56128	7921.39	23.61	3.16



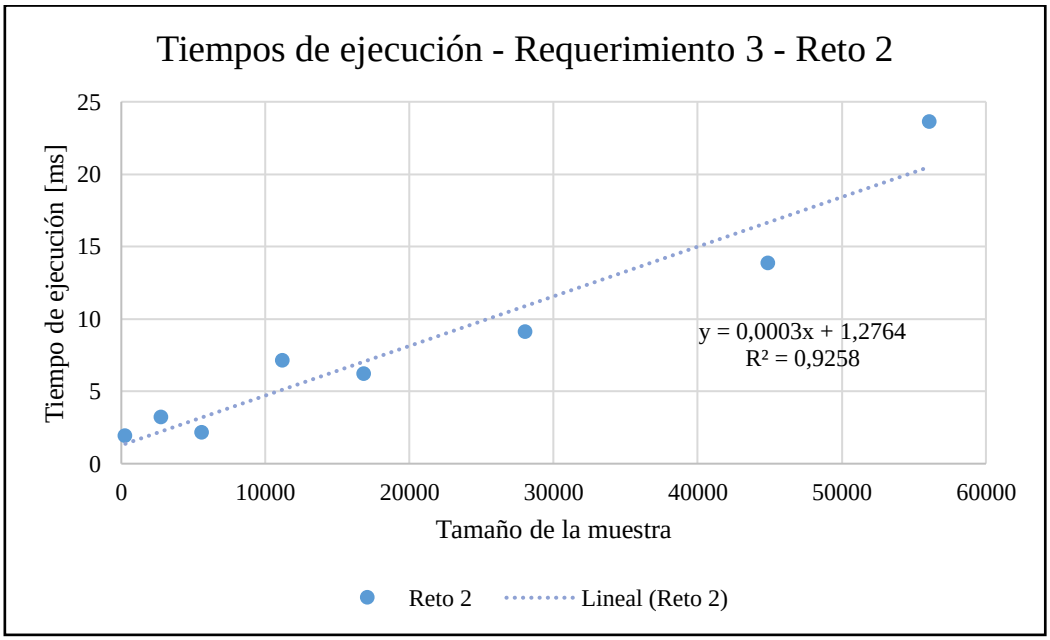
Gráfica 3.1: comparación de consumo de memoria entre reto1 y reto2 para el requerimiento 3.

La única manipulación dentro de la ejecución del requerimiento 3 cuya memoria podría depender del tamaño de la carga, era en el algoritmo de ordenamiento. Sin embargo, al usarse *Inertion Sort()*, el cual es inplace, esperábamos que la memoria se mantuviese constante. Al ver la gráfica este no parece ser una conclusión clara. Sin embargo, notamos que no hay relación entre el consumo de memoria y el tamaño de la muestra. Y también que los datos en lo general no tienden a desviarse más de un kB, siendo una excepción la última medición. Estas sutiles diferencias las atribuimos a características propias del computador dónde se realizaron las pruebas.

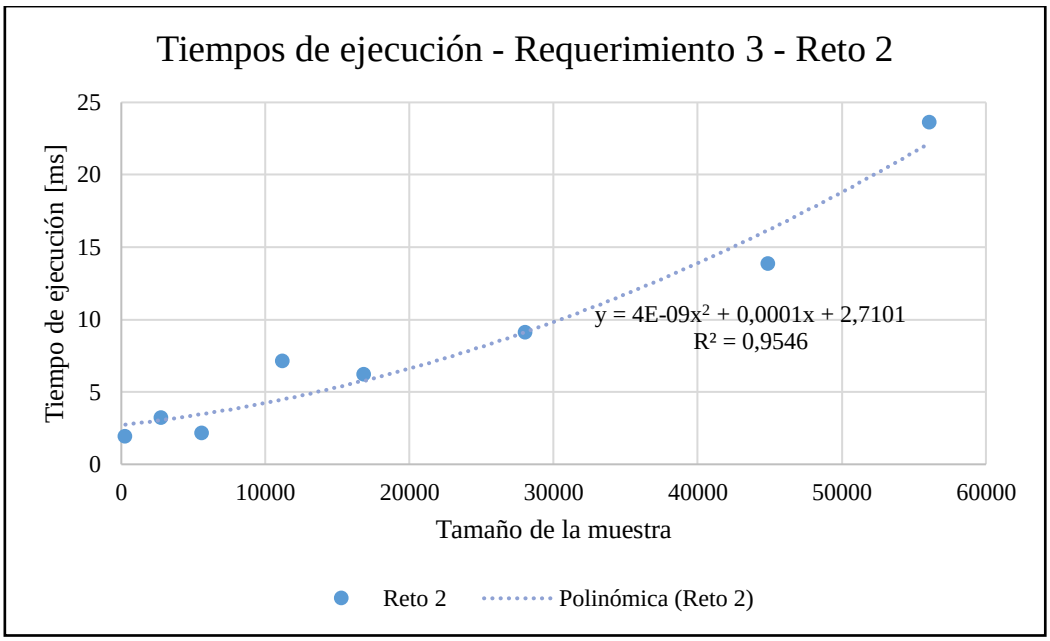


Gráfica 3.2: comparación de tiempos de ejecución entre reto1 y reto2 para el requerimiento 3.

Se demuestra que los tiempos de ejecución para el requerimiento 3 en el Reto 2 son significativamente menores a los necesarios en el Reto 1. Aspecto que puede ser explicado ya que en el Reto 1 este requerimiento contaba con una complejidad temporal de O(nlog(n)), mientras que ahora cuenta con una de O(n), aclarando que en este caso el n no hace referencia directa al Tamaño de la muestra, sino a una fracción diminuta de la misma. Esto se debe a que se usó el algoritmo de ordenamiento *Insertion Sort()*, sin embargo, debido a que se usó en situaciones dónde este algoritmo sobresale, su coste temporal no es visible en los tiempos de ejecución. Por último, notamos que se obtuvo un R^2 bueno, confirmando de esta manera la relación entre el tiempo de ejecución y el tamaño de la muestra para el requerimiento 3 es una tan pequeña que para un usuario no sería capaz de percibirla. Es decir, para un usuario podría considerar que este requerimiento tiene una complejidad O(1).



Gráfica 3.3: Tiempos de ejecución requerimiento 3, ajuste lineal.



Gráfica 3.4: Tiempos de ejecución requerimiento 3, ajuste cuadrático.

Revisando más a profundad las gráficas del requerimiento 3, notamos que al hacer una regresión lineal obtenemos un buen coeficiente de correlación. De igual manera al hacer una regresión polinómica de segundo grado, obtenemos también un buen coeficiente de correlación que, aunque es ligeramente superior, es de esperarse cuando se aumentan los términos en una regresión polinómica. Esto nos da a entender que el comportamiento real o más acertado de nuestro algoritmo debe ser un caso que está entre el peor y el mejor comportamiento de *Insertion Sort()*. Sin embargo, sería ideal continuar el análisis de este algoritmo al aumentar el tamaño de la muestra mucho más, pues de esta manera podríamos ver verdaderamente cómo se comporta nuestro algoritmo cuando la muestra aumenta en ordenes de magnitud.

Requerimiento 4: Encontrar la canción más popular de un artista.

Carga de datos para el requerimiento 4:

```
def tracks_byArtist(catalog, track):
    artists_id = track["artists_id"].strip("[").strip('"').strip("]").strip('[').replace("'","").split(",")

    for artist_id in artists_id:
        artist_id = artist_id.strip(" ")
        if artist_id == '':
            pass
        else:
            artist = mp.get(catalog['id_artists'], artist_id)['value']
            artist_name = artist['name']
            exist_name = mp.get(catalog['tracks_by_artist'], artist_name)


            if exist_name:
                insort_right_r4(exist_name["value"], track, lo=1, hi=None)

            else:           #Si no existe el id, crear la lista
                mp.put(catalog['tracks_by_artist'], artist_name, lt.newList('ARRAY_LIST'))
                exists_name = mp.get(catalog["tracks_by_artist"], artist_name)
                lt.addLast(exists_name["value"], track)
```

Para la carga de datos del requerimiento 4, creamos un TAD map donde la llave es el nombre del artista asociado y como valor un TAD lista con el track que contiene este artista. Para esto, accedemos a 'artists_id' extraemos cada uno de los ids, los pasamos a nombres con ayuda de otro TAD map y la canción queda referenciada en cada lista con artista asociado.

Nuevamente, usamos insort_right para que inserte ordenadamente por el criterio principal (popularidad).

```
def insort_right_r4(lista, track, lo=1, hi=None):
    """Insert item x in list a, and keep it sorted assuming a is sorted.

    If x is already in a, insert it to the right of the rightmost x.

    Optional args lo (default 0) and hi (default len(a)) bound the
    slice of a to be searched.
    """
    if hi is None:
        hi = lt.size(lista) + 1
    while lo < hi:
        mid = (lo+hi)//2
        if track["popularity"] > lt.getElement(lista, mid)["popularity"]:
            hi = mid
        else:
            lo = mid+1
    lt.insertElement(lista, track, lo)
```

Análisis ejecución del Requerimiento:

Entradas: Nombre del artista,

Siglas del país o mercado en el que se encuentra presente la canción

Salidas: El número total de canciones que son del artista y disponibles en el país indicado.

El número total de álbumes que posee un artista dado un mercado de distribución

Canción más popular (ordenamiento por popularidad, duración y nombre subsecuentemente) con la información asociada a esta canción.

Operaciones:

```python
#====================[R4]==================================
def answer_r4(catalog, artist_name, country):
    catalog = catalog['model']

    exist_name = mp.get(catalog['tracks_by_artist'], artist_name)
    if exist_name == None:
        return None, None, None
    else:

        list_artist = exist_name['value']

        rta_list_havoc = searchCountry(list_artist, country)

        rta_list_ord = insertion.sort(rta_list_havoc, cmpfunction=cmpR4)

        popular_track = lt.getElement(rta_list_ord, 1)
        info_r4(catalog, popular_track)
        num_tracks = uniqueTracks(rta_list_ord)
        num_albums = uniqueAlbums(rta_list_ord)



    return popular_track, num_tracks, num_albums
```

Inmediatamente que se selecciona la opción 4, se busca el nombre en el map *'tracks_by_artist'* asumiendo que existe, accede el valor y a través de iteraciones, busca que coincida el país ingresado.

```python
def searchCountry(lista, inp_country):
    rta_list = lt.newList('ARRAY_LIST')

    for track in lt.iterator(lista):
        countries = track["available_markets"].strip("[").strip('"').strip("]").strip('[').replace("'","").split(",")

        for country in countries:
            country = country.strip(' ')
            if country == inp_country:
                lt.addLast(rta_list, track)

    return rta_list
```

Para la complejidad de *seachCountry()*:

O(tracks por artista)

Ya extraídas las canciones que cumplen nuestras condiciones, realizamos un algoritmo de ordenamiento adaptativo (*insertion.sort()*) para que se cumplan todas las condiciones de ordenamiento dadas en el R4.

```
def cmpR4(track1, track2):
    menor = True
    # De mayor a menor la popularidad
    if track1['popularity'] < track2['popularity']:
        menor = False

    elif track1['popularity'] == track2['popularity']:

        # De mayor a menor la duración
        if track1['duration_ms'] < track2['duration_ms']:
            menor = False

        elif track1['duration_ms'] == track2['duration_ms']:

            if track1['name'] > track2['name']:
                menor = False

    return menor
```

La complejidad de este paso es aproximadamente el mejor caso de insertion.sort por lo que

O(tracks por artista y país disponible)

Luego a través de *info_r4()* buscamos la información asociada a la primera canción únicamente, esto es independiente de cuántas canciones carguemos pues utilizamos maps por lo que su complejidad es

O(1)

```
def info_r4(catalog, popular_track):
    album_id = popular_track['album_id']
    album = mp.get(catalog['id_albums'], album_id)['value']
    album_name = album['name']
    album_date = album['release_date']
    #1
    popular_track['album_name'] = album_name
    #2
    popular_track['release_date'] = album_date

    artists_id = popular_track["artists_id"].strip("[").strip('"').strip("]").strip('[').replace("'","").split(",")

    artists = ''
    for artist_id in artists_id:
        artist_id = artist_id.strip(" ")
        if artist_id == '':
            pass
        else:
            artist = mp.get(catalog['id_artists'], artist_id)['value']
            artist_name = artist['name']

            artists += artist_name + ', '

    #3
    popular_track['artists_name'] = artists
```

Luego, de la lista ordenada, hacemos una transformación de TAD lista a TAD map para que colapsen las canciones repetidas y así al tomar *mp.size()* contar tanto los álbumes como las canciónes únicas.

```python
def uniqueAlbums(list_tracks):

    my_map = mp.newMap(maptype='PROBING',
                       loadfactor=our_loadfactor,
                       numelements=60)

    for track in lt.iterator(list_tracks):
        album_id = track['album_id']
        mp.put(my_map, album_id, True)

    return mp.size(my_map)
```

```python
def uniqueTracks(list_tracks):

    my_map = mp.newMap(maptype='PROBING',
                       loadfactor=our_loadfactor,
                       numelements=60)

    for track in lt.iterator(list_tracks):
        track_id = track['id']
        mp.put(my_map, track_id, True)

    return mp.size(my_map)
```

Su complejidad asociada es

O(tracks por artista y país disponible)

Por tanto, su complejidad final es…

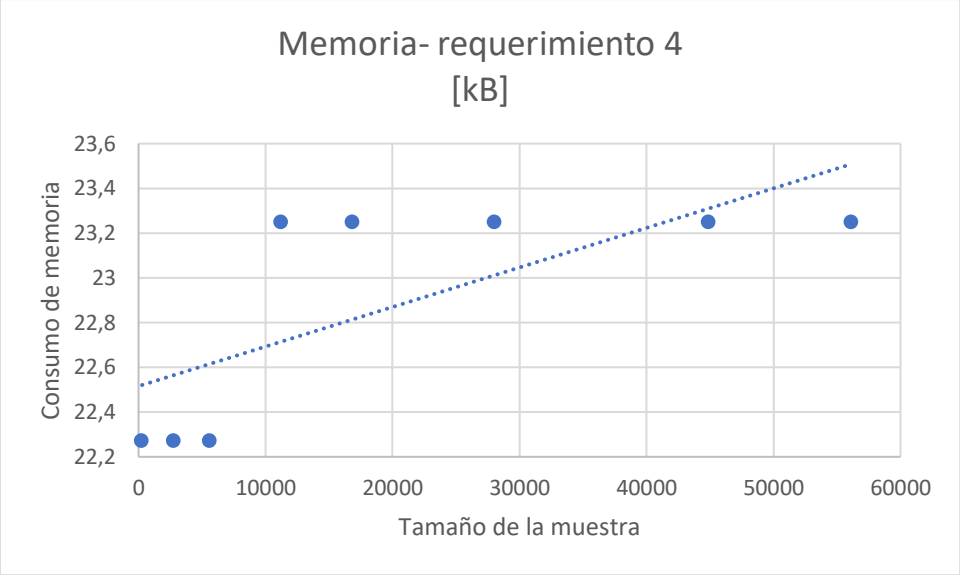
O(tracks por artista) + … + O(tracks por artista y país) =

Como la cantidad de canciones por artista es mayor que la cantidad de canciones dado un artista y país, la complejidad final está determinada por la cantidad de canciones por artista.

Por tanto, la complejidad final es igual a … **O(tracks por artista)**

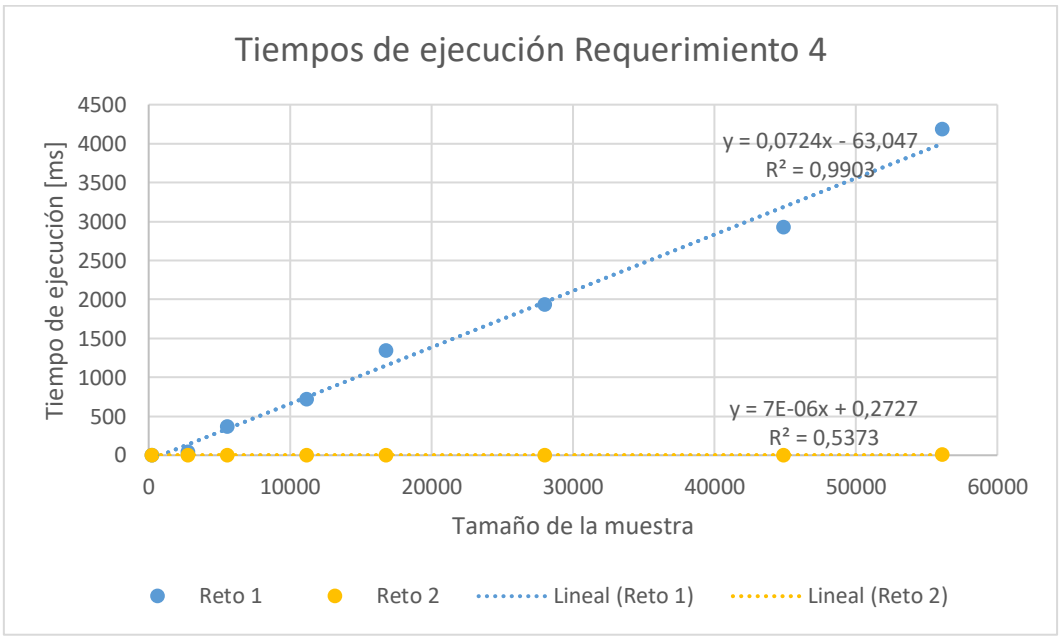
		Reto 1	Reto 2	Reto 2
Porcentaje de la muestra [pct]	Tamaño de la muestra (ARRAY_LIST)	Tiempo de ejecución (Merge Sort) [ms]	Tiempo de ejecución [ms]	Memoria [kB]
0.50%	281	37.43	2.18	22.27
5.00%	2806	365.02	2.95	22.27
10.00%	5613	717.58	3.23	22.27
20.00%	11226	1343.96	3.95	23.25
30.00%	16838	1932.5	3.21	23.25

50.00%	28064	2925.63	4.15	23.25
80.00%	44902	4180.72	4.05	23.25
100.00%	56128	5019.7	5.01	23.25



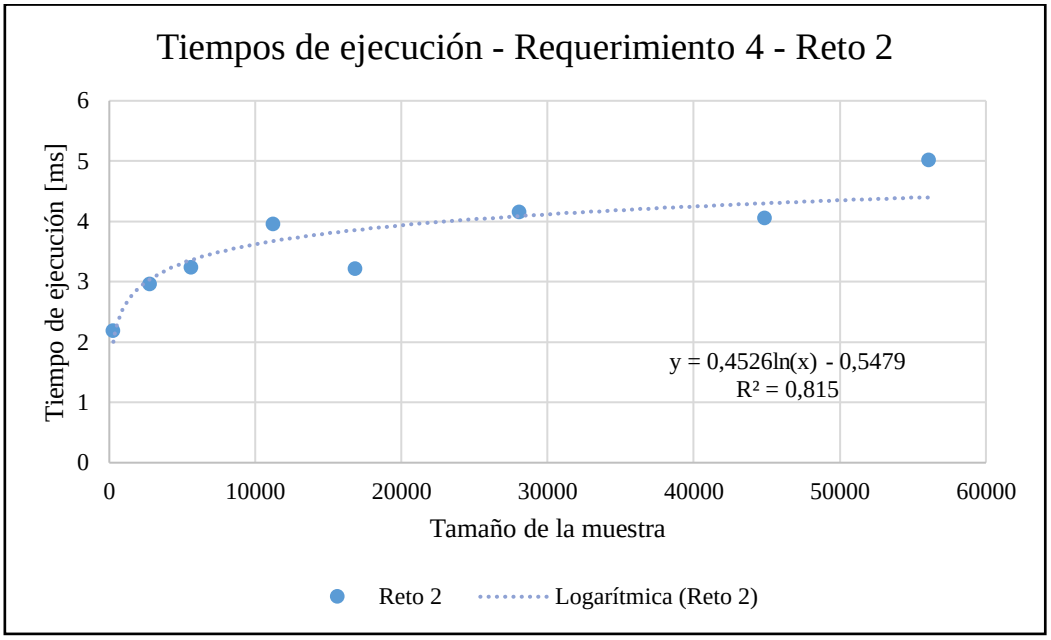
Gráfica 4.1: comparación de consumo de memoria entre reto1 y reto2 para el requirimiento 4.

Para el consumo de memoria, nos dimos cuenta de que la distribución sigue un patrón por escaleras. Explicamos este comportamiento por el aumento de canciones por artista a partir de 20pct dada la combinación de artista/país.



Gráfica 4.2: comparación de tiempos de ejecución entre reto1 y reto2 para el requerimiento 4.

Para los tiempos de ejecución, nos damos cuenta de que los tiempos del R2 siempre son menores a 50 ms y para el R1 a partir del 5 pct ya pasamos este umbral. Además, en términos de complejidad, recordemos que el R1 tenía una complejidad final de O(n log n) mientras nuestra nueva solución es O(n) con un n mucho menor no siendo la cantidad total de canciones sino la cantidad de canciones por el artista dado. A continuación, presentamos la gráfica de tiempos de ejecución del requerimiento 4:



Gráfica 4.3: Tiempos de ejecución requerimiento 4, ajuste logarítmico.

En la gráfica, claramente podemos evidenciar un comportamiento logarítmico. Esto lo podemos explicar si profundizamos en la distribución de canciones con respecto a un artista a medida que aumenta el tamaño de la muestra. Si la cantidad de canciones de un mismo artista promedio aumenta logarítmicamente, nuestro algoritmo reflejará ese comportamiento pues estamos iterando sobre estos resultantes. Para poder confirmar esta hipótesis y verificar que nuestro algoritmo es efectivamente O(tracks por artista) necesitaríamos asegurar una distribución lineal a medida que aumenta el tamaño de la muestra o darle directamente al algoritmo en cantidades directamente proporcionales, canciones de un mismo artista.

Requerimiento 5: Encontrar la discografía de un artista.

Análisis carga del requerimiento:

Para cumplir con el Requerimiento 5 se crearon 2 mapas durante la carga.

```python
#===============[R5]===============
artist_name_for_key = mp.get(catalog["id_artists"], album["artist_id"])

if artist_name_for_key != None:
    artist_name_for_key = artist_name_for_key["value"]["name"]

if artist_name_for_key == None:
    pass
else:
    list_artist_albums = mp.get(catalog["albums_by_artist"], artist_name_for_key)

    if list_artist_albums:
        insort_right_r5(list_artist_albums["value"],album, lo=1,hi=None)

    else:
        mp.put(catalog["albums_by_artist"], artist_name_for_key, lt.newList("ARRAY_LIST"))
        list_artist_albums = mp.get(catalog["albums_by_artist"], artist_name_for_key)
        lt.addLast(list_artist_albums["value"], album)
```

```python
def insort_right_r5(lista, album, lo=1, hi=None):
    """Insert item x in list a, and keep it sorted assuming a is sorted.

    If x is already in a, insert it to the right of the rightmost x.

    Optional args lo (default 0) and hi (default len(a)) bound the
    slice of a to be searched.
    """
    if hi is None:
        hi = lt.size(lista) + 1
    while lo < hi:
        mid = (lo+hi)//2
        if album["release_date"] > lt.getElement(lista, mid)["release_date"]:
            hi = mid

        else:
            lo = mid+1
    lt.insertElement(lista, album, lo)
```

En el primero las llaves están compuestas por los nombres de los artistas, y los valores de estas, son listas que contienen los álbumes compuestos por un mismo artista ordenados de acuerdo a la fecha de lanzamiento. De todas las operaciones que se realizan, consideramos que únicamente es valioso mencionar la complejidad de utilizar la función *Insort()*, que, como ya hemos mencionado, es de O(log(n)), donde n es el tamaño de la lista dónde está tratando de agregar un elemento. El resto de las acciones son O(1).

```python
#===============[R5]===================
album_name_for_key = mp.get(catalog["id_albums"], track["album_id"])

if album_name_for_key != None:
    album_name_for_key = album_name_for_key["value"]["name"]

if album_name_for_key == None:
    pass
else:
    list_album_tracks = mp.get(catalog["tracks_in_album"], album_name_for_key)

    if list_album_tracks:
        insort_right_r5_2(list_album_tracks["value"], track, lo=1,hi=None)

    else:
        mp.put(catalog["tracks_in_album"], album_name_for_key , lt.newList("ARRAY_LIST"))
        list_album_tracks = mp.get(catalog["tracks_in_album"], album_name_for_key)
        lt.addLast(list_album_tracks["value"], track)
```

```
def insort_right_r5_2(lista, track, lo=1, hi=None):
    """Insert item x in list a, and keep it sorted assuming a is sorted.

    If x is already in a, insert it to the right of the rightmost x.

    Optional args lo (default 0) and hi (default len(a)) bound the
    slice of a to be searched.
    """
    if hi is None:
        hi = lt.size(lista) + 1
    while lo < hi:
        mid = (lo+hi)//2
        if float(track["popularity"]) > float(lt.getElement(lista, mid)["popularity"]):
            hi = mid

        else:
            lo = mid+1
    lt.insertElement(lista, track, lo)
```

En el segundo mapa las llaves son los nombres de álbumes y los valores a los que llevan esas llaves son listas que contienen las canciones pertenecientes a dichos álbumes, aclarando que las canciones dentro de las listas están ordenadas según popularidad. Nuevamente, la mayoría de las instrucciones en la carga son O(1), la complejidad más grande es la de utilizar *Insort()* y esta es O(log(n)) donde n es el tamaño de la lista sobre la cual desea agregar un elemento.

Análisis ejecución del Requerimiento:

```
def answer_R5(catalog, artist_name):

    list_albums = mp.get(catalog["model"]["albums_by_artist"], artist_name)

    if list_albums == None:
        return None, None, None, None

    list_albums = list_albums["value"]
    album_types = count_album_type(list_albums)

    insertion.sort(list_albums,  albums_by_date)

    resp_albums = tomar_primeros_ultimos(list_albums)


    resp_tracks = lt.newList("ARRAY_LIST")
    names_artists = lt.newList("ARRAY_LIST")

    for album in lt.iterator(resp_albums):
        name_album = album["name"]
        tracks_of_album = mp.get(catalog["model"]["tracks_in_album"], name_album)["value"]
        insertion.sort(tracks_of_album ,tracks_by_popularity)
        most_popular_track = lt.getElement(tracks_of_album, 1)
        lt.addLast(resp_tracks, most_popular_track )

        artists_ids = most_popular_track["artists_id"].strip("[").strip('"').strip("]").strip('[').replace("'","").split(",")

        names = lt.newList("ARRAY_LIST")
        for id in artists_ids:
            id = id.strip(" ")
            exists_name = mp.get(catalog["model"]["id_artists"], id)
            if exists_name:
                lt.addLast(names, exists_name["value"]["name"])
            else:
                lt.addLast(names, "Not Found")
        lt.addLast(names_artists, names)

    return album_types, resp_albums, resp_tracks, names_artists
```

Entradas: Catalogo que contiene los mapas cargados al inicio del programa y el nombre de un artista ingresado por el usuario.

Salidas: Una tupla que indica el número de los diferentes tipos de álbumes que contiene el artista en su discografía. Una lista que contiene los últimos 3 y los primeros 3 álbumes lanzados por un artista. Una lista que contiene la canción más famosa de los 6 álbumes previamente seleccionados. Y una lista que contiene los nombres de los artistas asociados a las canciones previamente seleccionadas.

Operaciones:

1) Se hace mp.get() para saber si el artista existe, en caso de no hacerlo se retorna None.

O(1)

```
def count_album_type(list_albums):
    singles = 0
    compilations = 0
    albumes = 0
    total = lt.size(list_albums)

    for album in lt.iterator(list_albums):
        if album["album_type"] == "single":
            singles += 1
        elif album["album_type"] == "compilation":
            compilations += 1
        elif album["album_type"] == "album":
            albumes += 1

    return (singles, compilations, albumes, total)
```

2) Se recorren los álbumes del artista ingresado por el usuario para saber cuantos álbumes de cada tipo posee dicho artista. Por lo que se itera sobre el tamaño de los álbumes del artista.

O(Núm. Álbumes Artista)

```
def albums_by_date(album1, album2):
    """
    Devuelve verdadero (True) si el año de publicación del album1 es menor que los del album2
    Args:
    album1: informacion del primer album que incluye su fecha en time
    album2: informacion del segundo artista que incluye su fecha en time
    """
    menor = False
    if album1["release_date"] == album2["release_date"]:
        if album1["name"] < album2["name"]:

            menor = True
    else:
        if album1["release_date"] > album2["release_date"]:
            menor = True

    return menor
```

3) Se ordena la lista de los álbumes del artista primero por fecha, y en caso de que estas sean iguales se ordena por nombre. El algoritmo de ordenamiento usado es *Insertion Sort()* y debido a que esta lista está casi ordenada, ya que durante la carga esta se ordenó por el primer criterio con *Insort()*, se trabaja con *Insertion Sort()* en su mejor caso.

O(Núm. Álbumes Artista)

4) Seguido de esto, se eligen los 3 últimos y los 3 primeros álbumes lanzados por el artista.

O(1)

```
def tracks_by_popularity(track1, track2):
    menor = False
    if float(track1['popularity']) == float(track2['popularity']):
        if float(track1["duration_ms"]) == float(track2["duration_ms"]):
            if track1["name"] < track2["name"]:
                menor = True
        else:
            if float(track1["duration_ms"]) > float(track2["duration_ms"]):
                menor = True
    else:
        if float(track1['popularity']) > float(track2['popularity']):
            menor = True
    return menor
```

5) Una vez con los 6 álbumes necesarios, se accede a sus canciones mediante el mapa creado durante la carga. De igual manera, estas listas de canciones están casi ordenadas debido a que en la carga se hizo *Insort()* de acuerdo a la popularidad de las canciones, por lo que correr *Insertion Sort()* para ordenar por los criterios restantes no resulta ser una acción costosa temporalmente, sin contar el hecho de que por lo general el número de canciones en un álbum no es grande.

O(Núm. Tracks en Álbum)

6) Después de ordenar las canciones se hace *lt.getElement()* de la primera canción en cada álbum, debido a que por el ordenamiento está debe ser la canción más popular.

O(1)

7) Finalmente, al contar con las canciones más populares de los 6 álbumes elegidos, solo queda encontrar los nombres de los artistas asociados a estas canciones. Acción que se realiza mediante mapas creados durante la carga. Por lo que esta acción se realiza en un tiempo constante.

O(1)

Complejidad Final:

O(1 + 2xNúm. Álbumes Artista + 6xNúm. Tracks en Álbum + 1 + 1)

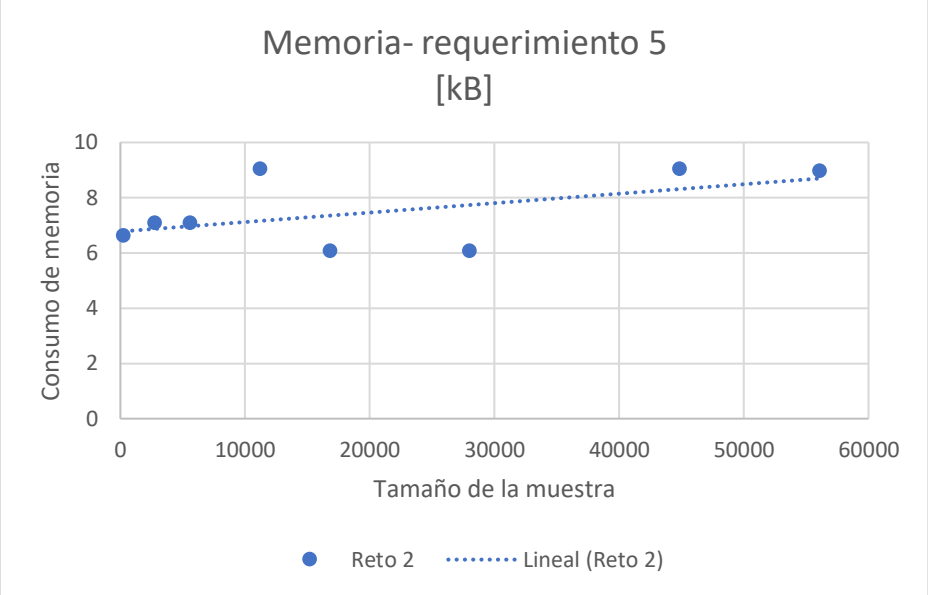
Que de manera simplificada queda:

O(Núm. Álbumes Artista + Núm. Tracks en Álbum)

Complejidad que se puede aproximar a **O(n)**. Pero que cómo ya hemos dicho, debido a la naturaleza de los datos, es un n supremamente pequeño que es percibido por el usuario cómo **O(1)**.

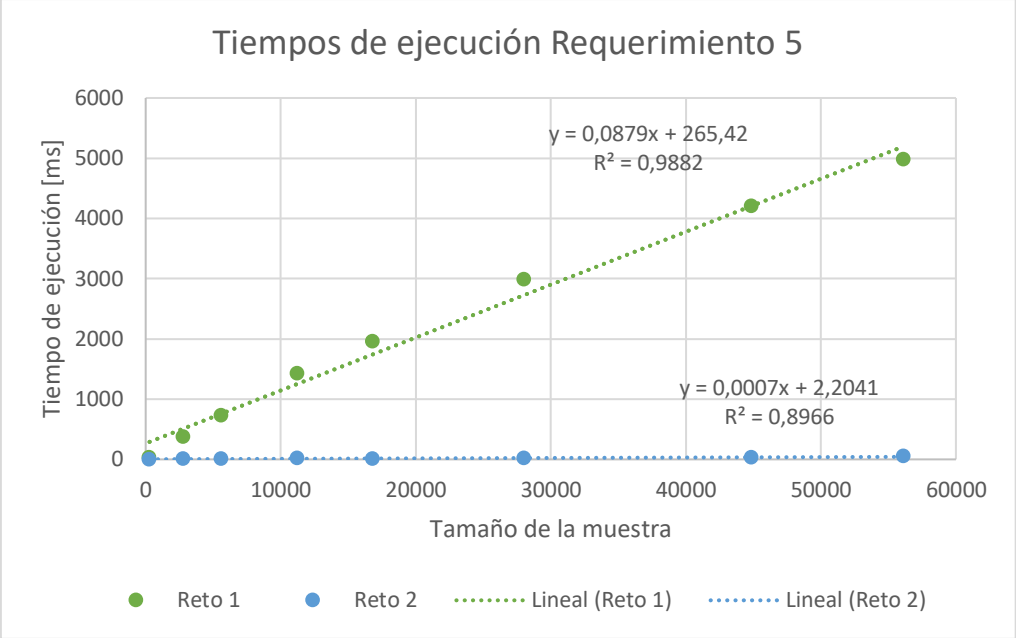
		Reto 1	Reto 2	Reto 2
Porcentaje de la muestra [pct]	Tamaño de la muestra (ARRAY_LIST)	Tiempo de ejecución (Merge	Tiempo de ejecución [ms]	Memoria [kB]

		Sort) [ms]		
0.50%	281	33.42	1.14	6.61
5.00%	2806	372.09	3.6	7.07
10.00%	5613	725.39	5.64	7.07
20.00%	11226	1430.12	18.4	9.02
30.00%	16838	1956.69	12.06	6.07
50.00%	28064	2985.14	19.31	6.07
80.00%	44902	4208.66	26.87	9.02
100.00%	56128	4982.57	48.18	8.96



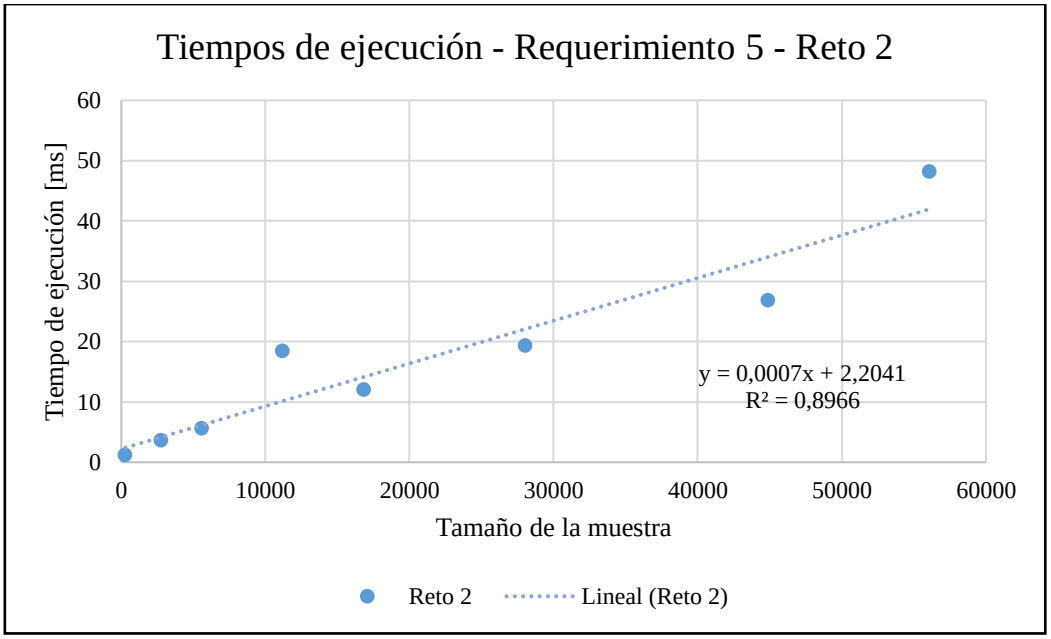
Gráfica 5.1: comparación de consumo de memoria entre reto1 y reto2 para el requerimiento 5.

Nuestra hipótesis con respecto al consumo de memoria era que está se mantendría constante, pues la funciones y algoritmos usados no deberían mostrar una relación entre el consumo de memoria y el tamaño de muestra. En este caso, vemos que, aunque no se evidencia una variación significativa en la memoria, puede que exista una muy ligera correspondencia entre la memoria y el tamaño de muestra. Sin embargo, la información recolectada y la poca correlación entre la línea de tendencia y los datos obtenidos nos lleva a concluir que no podemos ni confirmar o refutar nuestra hipótesis. A grandez rasgos si existe una relación esta puede que sea lo suficientemente pequeña para ser catalogada cómo irrelevante.

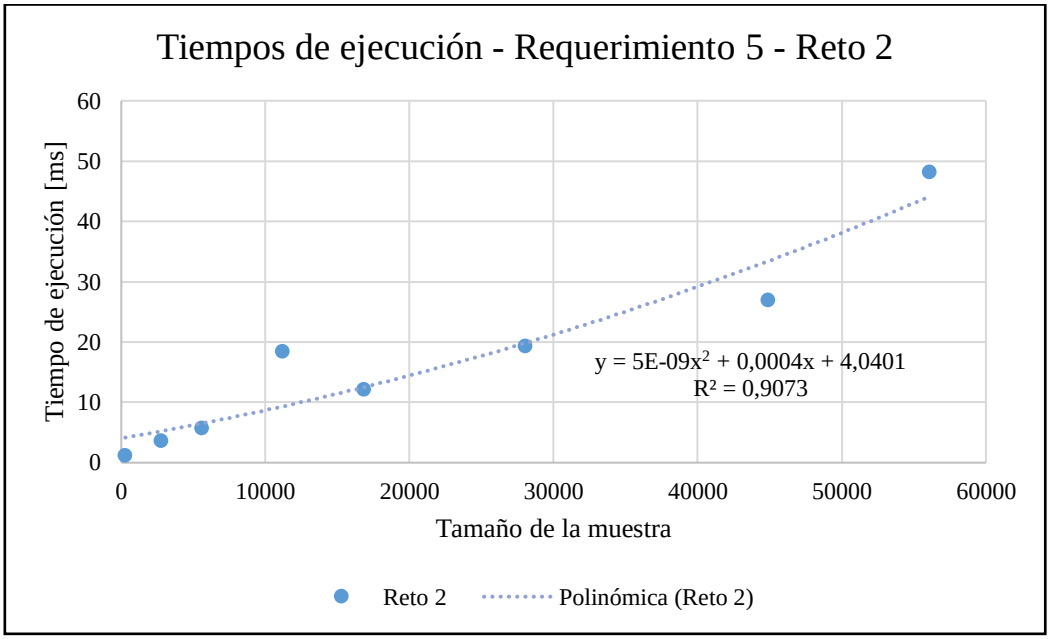


Gráfica 5.2: comparación de tiempos de ejecución entre reto1 y reto2 para el requerimiento 5.

Se demuestra que los tiempos de ejecución para el requerimiento 5 en el Reto 2 son significativamente menores a los necesarios en el Reto 1. Hecho que era esperado ya que en el reto 1 el Requerimiento 5 tenía una complejidad temporal asociada de O(nlog(n)), mientras que en Reto 2 es de O(n), aclarando que n en este caso es ordenes de magnitud menores al n del Reto 1. Adicionalmente, es clara la presencia de una relación directa entre el tamaño de la muestra y el tiempo de ejecución del requerimiento, pues durante la ejecución del requerimiento 5 se hace uso de *Insertion Sort()*, sin embargo, se hace uso del mismo bajo una situación restringida dónde el comportamiento que presenta el algoritmo es una combinación entre el mejor y el peor de caso de *Insertion Sort()*. Aspecto que podemos notar ya que el R^2 de la regresión lineal y la regresión polinómica en ambos casos es considerablemente buena, y además bastante cercana. Sin embargo, si deseamos ver cómo se continuaría comportando este algoritmo a medida que crece aún más el tamaño de muestra, sería necesario realizar pruebas con más datos, de manera que se pueda revelar de manera más precisa el comportamiento de este algoritmo.



Gráfica 5.3: Tiempos de ejecución requerimiento 5, ajuste lineal.



Gráfica 5.4: Tiempos de ejecución requerimiento 5, ajuste cuadrático.

Requerimiento 6: Clasificar las canciones de los artistas con mayor distribución.

Análisis carga del requerimiento:

```
#================[R6]====================
available_markets = track['available_markets']
str_lista_markets = available_markets[2:(len(available_markets)-2)]
lista_markets = str_lista_markets.split("', '")


distribution = len(lista_markets)
track['distribution'] = distribution
```

Para la carga de datos para el requerimiento 6, primero extraemos la cantidad de paises para cada uno de los tracks y se guarda en una nueva pareja valor, llave la cantidad de paises en los que está disponible. Además, también como entradas un artista y un país de distribución, entonces el map creado para el req. 4 nos sirve sin tener que gastar más memoria (revisar req. 4).

Análisis ejecución del Requerimiento:

Entradas: 1. Nombre de un artista

2. País/mercado de distribución

3. Top N a buscar.

Salidas:

1. Los N países de mayor distribución
2. Los tres primeros y tres últimos elementos del Top N ingresado para una lista de canciones dado el nombre de un artista y un país, su ordenamiento es por popularidad, duración y nombre subsecuentemente).

Operaciones:

```python
#==================[R6]==================================
def answer_r6(catalog, artist_name, country, n):
    catalog = catalog['model']

    exist_name = mp.get(catalog['tracks_by_artist'], artist_name)
    if exist_name == None:
        return None, None
    else:

        list_artist = exist_name['value']

        rta_list_havoc = searchCountry(list_artist, country)
        rta_list_ord = insertion.sort(rta_list_havoc, cmpfunction=cmpR6)
        num_tracks = lt.size(rta_list_ord)



    if num_tracks < n:
        top_n = rta_list_ord
    else:
        top_n = lt.subList(rta_list_ord, 1, n)

    answer_list = tomar_primeros_ultimos(top_n)
    info_r6(catalog, answer_list)

    return answer_list, num_tracks
```

Inmediatamente después de que el usuario ingresa la opción 6 se busca el nombre ingresado en el map creado para el req. 4. Asumiendo que la llave existe, se accede a value y se obtienen las canciones asociadas al artista

O(1)

Nuevamente tomamos la metodología de recorrer esta lista buscando individualmente si contenía el país de interés y guardándolo en otra lista (revisar *seachCountry()* del req. 4).

```python
def searchCountry(lista, inp_country):
    rta_list = lt.newList('ARRAY_LIST')

    for track in lt.iterator(lista):
        countries = track["available_markets"].strip("[").strip('"').strip("]").strip('[').replace("'","").split(",")

        for country in countries:
            country = country.strip(' ')
            if country == inp_country:
                lt.addLast(rta_list, track)

    return rta_list
```

Por tanto, la complejidad es de …

O(tracks por artista)

Posteriormente se realiza insertion.sort() para asegurar un ordenamiento dados los múltiples criterios de ordenamiento dados.

```python
def cmpR6(track1, track2):
    menor = True
    # De mayor a menor la popularidad
    if track1['popularity'] < track2['popularity']:
        menor = False

    elif track1['popularity'] == track2['popularity']:

        # De mayor a menor la duración
        if track1['duration_ms'] < track2['duration_ms']:
            menor = False

        elif track1['duration_ms'] == track2['duration_ms']:

            if track1['name'] > track2['name']:
                menor = False

    return menor
```

Nuevamente, esperamos lo más cercano al mejor caso de *insertion.sort()* lista parcialmente-ordenada por lo que la complejidad de este paso sería:

O(tracks por artista y país)

Posteriormente, determinamos que por cada combinación de artista/país todas las canciones tenían exactamente la misma distribución o mercados disponibles, por lo que para la primera salida es suficiente con lt.size().

O(1)

(Obviando la corrección de errores, caso de que sean menos 6 el top escogido o menos de 6 canciones que cumplen) Se crea una sublista de n (input) elementos y esta se pasa por la función tomar_primeros_ultimos() que retorna los tres primeros y tres últimos elementos. Como esta función siempre tendrá un tiempo constante para más de 6 canciones podemos asegurar que para tamaños grandes de datos la complejidad de este paso es…

O(1)

Finalmente, la información pedida para los tres primeros y tres últimos elementos es exactamente la misma que para la canción más popular del R4 por tanto iteramos buscando la información de cada uno de las 6 canciones como máximo.

```python
def info_r6(catalog, lista):
    for i in lt.iterator(lista):
        info_r4(catalog, i)
```

Por tanto, para datos muy grandes, este paso será

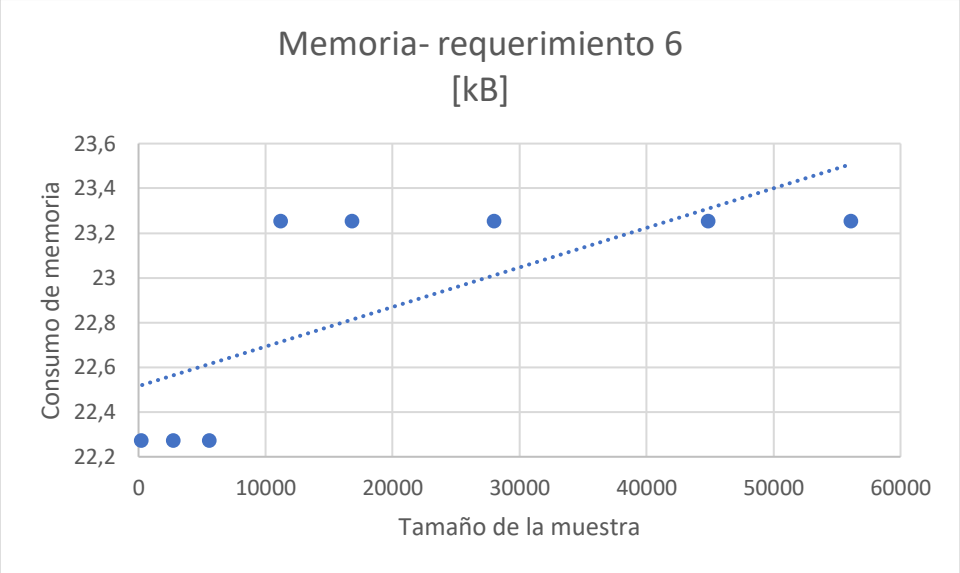
O(1)

Así pues, la complejidad final será de

O(1) + … + O(1) = O(tracks por artista)

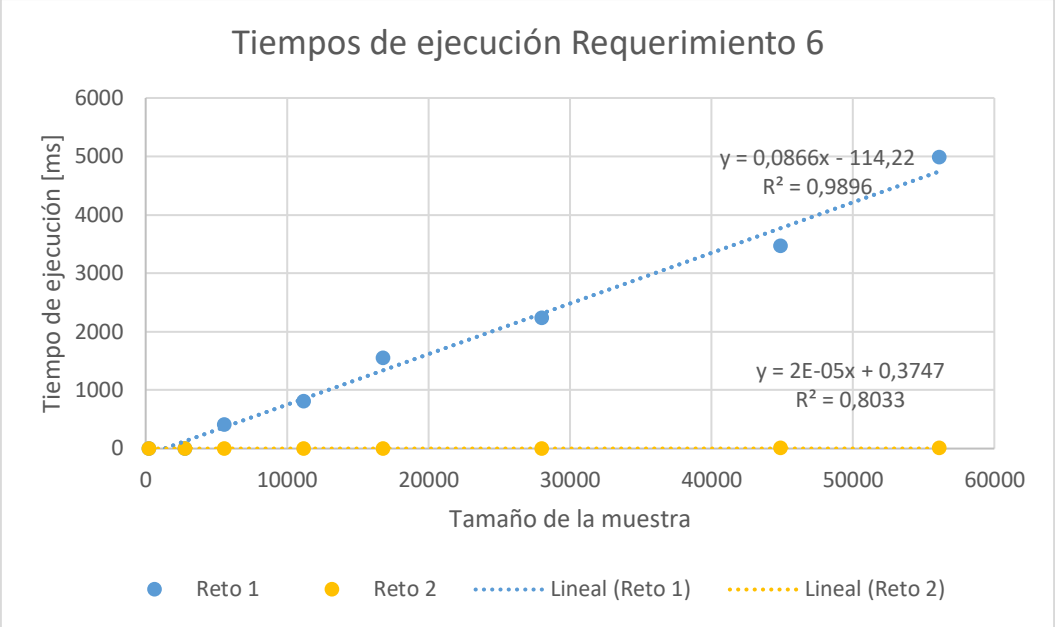
Particularmente, porque las canciones por artista son mayores a las canciones para una combinación de artista/país dado.

		Reto 1	Reto 2	Reto 2
Porcentaje de la muestra [pct]	Tamaño de la muestra (ARRAY_LIST)	Tiempo de ejecución (Merge Sort) [ms]	Tiempo de ejecución [ms]	Memoria [kB]
0.50%	281	37.37	0.43	1.54
5.00%	2806	409.42	0.65	1.65
10.00%	5613	805.01	0.73	6.26
20.00%	11226	1550.94	0.88	2.9
30.00%	16838	2229.8	0.88	2.9
50.00%	28064	3461.47	1.08	2.86
80.00%	44902	4985.55	1.42	2.86
100.00%	56128	5799.75	1.42	2.86



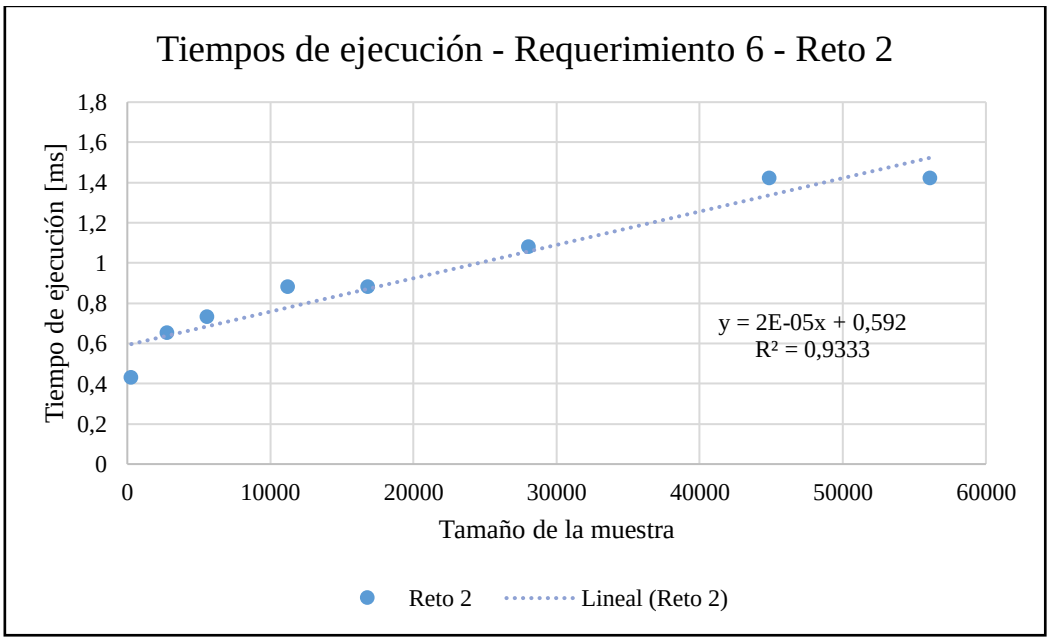
Gráfica 6.1: comparación de consumo de memoria entre reto1 y reto2 para el requirimiento 6

Para la memoria, nos damos cuenta que sigue una distribución por escalonada, nosotros explicamos esto porque en los tamaños de carga más pequeños se trabajaron con 3 canciones mientra que en los otros se trabajaba al final con 6 canciones. Esta hipótesis explica porque los tiempos dado cierto punto son constantes y antes de este punto también.

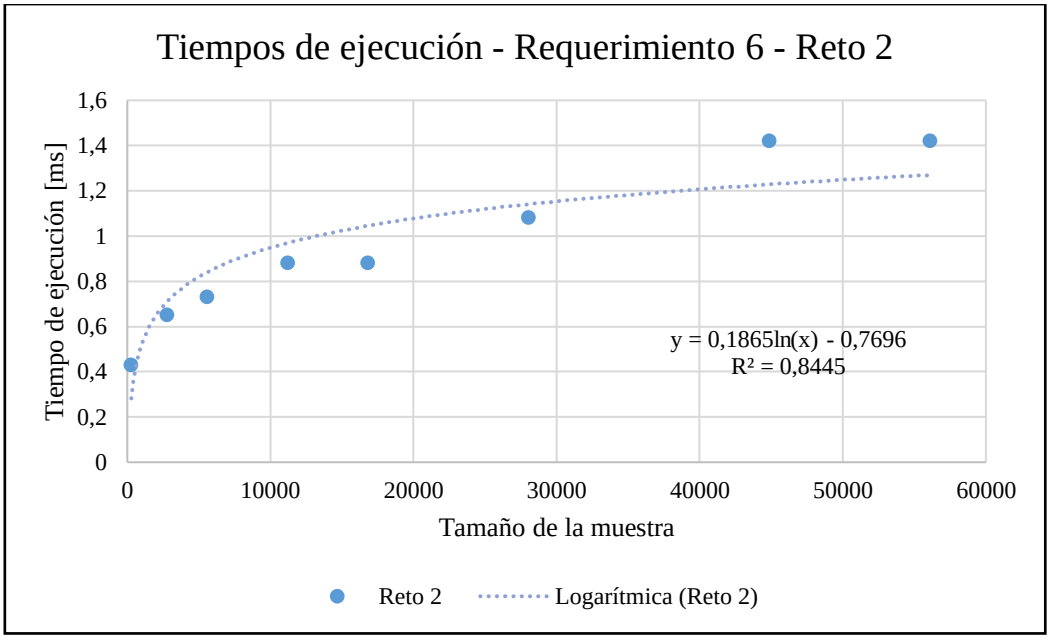


Gráfica 6.2: comparación de tiempos de ejecución entre reto1 y reto2 para el requerimiento 6.

De manera similar a los otros requerimientos, los tiempos de ejecución para este reto son siempre menores a 50ms a comparación de los del R1 que ya en 5pct alcanzaba los 400 ms. Al comparar la complejidad, recordamos que la complejidad del R1 era de O(n log n), nosotros en nuestro análisis determinamos que es O(m) con m mucho menor que n por lo que claramente vemos una complejidad temporal menor en nuestro R2 a comparación del R1. Con respecto a la complejidad del bono, se muestra a continuación su respectiva gráfica:



Gráfica 6.3: Tiempo de ejecución requerimiento 6 (bono), ajuste lineal.



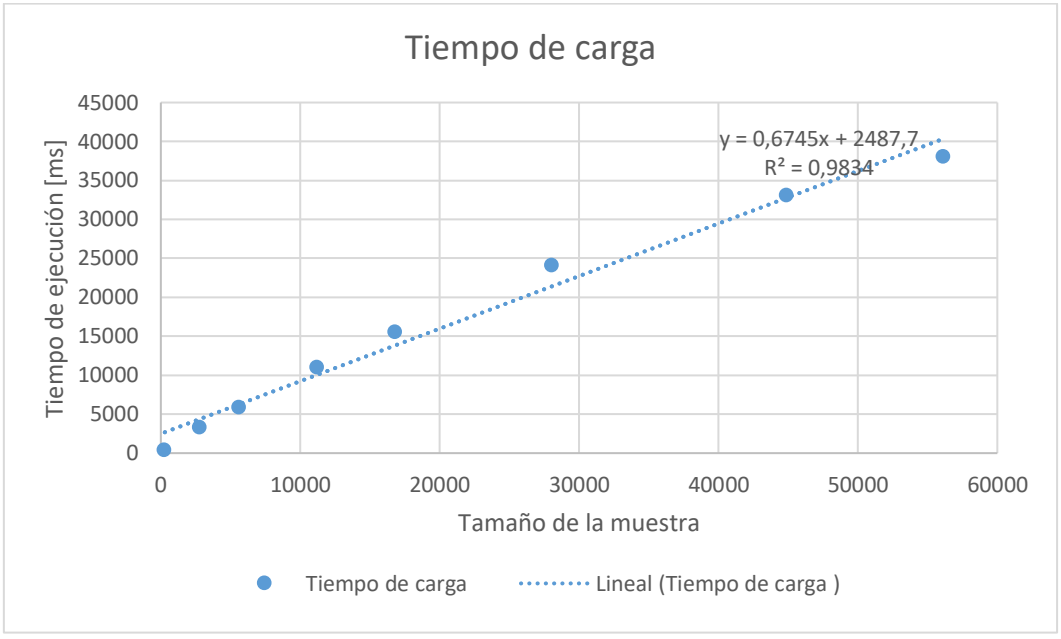
Gráfica 6.4: Tiempo de ejecución requerimiento 6 (bono), ajuste cuadrático.

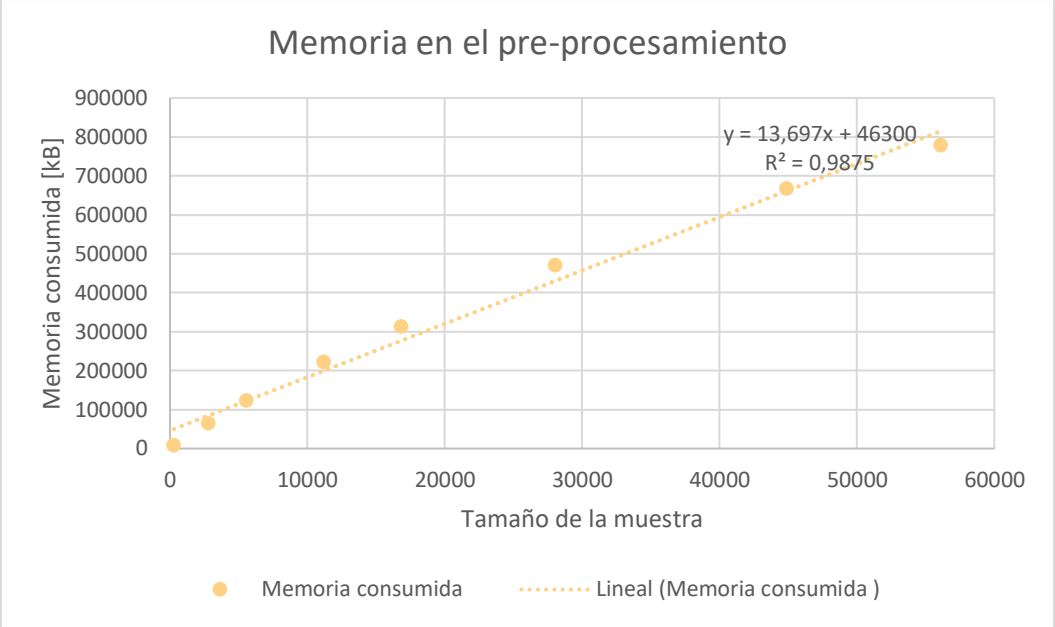
Para estas dos gráficas, pudimos determinar un mejor ajuste lineal que logarítmico, al igual que el requerimiento 4, estimamos una complejidad O(n) siendo n un subconjunto del tamaño de muestra, por lo que la relación entre las canciones de un artista y el total de canciones es lo que determina la complejidad en base del total del tamaño de la muestra. En este sentido, podemos explicar este comportamiento si asumimos que la distribución de canciones por artista es entre logarítmica y lineal. Únicamente evidenciando el valor del R cuadrado podríamos establecer una relación más lineal que logarítmica, sin embargo, esto no es absoluto pues los datos se encuentran entre estas dos proporciones. En cualquier caso, estamos cumpliendo con tener una complejidad aceptable (peor caso lineal) y esto se evidencia, nuevamente, en la comparación con respecto a los tiempos del reto 1.

ANEXOS

Anexo 1: tiempos y memoria en el pre-procesamiento

		Reto 2	Reto 2
Porcentaje de la muestra [pct]	Tamaño de la muestra (ARRAY_LIST)	Tiempo de ejecución [ms]	Memoria [kB]
0.50%	281	460.88	7415.72
5.00%	2806	3369.03	64092.36
10.00%	5613	5951.03	121739.93
20.00%	11226	11055.74	221933.93
30.00%	16838	15608.99	312883.15
50.00%	28064	24112.45	470214.78
80.00%	44902	33152.44	666218.89
100.00%	56128	38058.21	777592.13




Memoria en el pre-procesamiento
Memoria consumida [kB]
900000
800000
700000
600000
500000
400000
300000
200000
100000
0
0
10000
20000
30000
40000
50000
60000
Tamaño de la muestra
y = 13,697x + 46300
R² = 0,9875
Memoria consumida
Lineal (Memoria consumida)