# OBSERVACIONES DE LA PRÁCTICA

Juan David Salguero, 201923136, J.salguero@uniandes.edu.co
David Molina, 202125176, d.molinad@uniandes.edu.co

| | Máquina |
|---|---|
| **Procesadores** | Intel Core i5 |
| **Memoria RAM (GB)** | 8GB |
| **Sistema Operativo** | macOs Catalina - 64 bits |

*Tabla 1. Especificaciones de las máquinas para ejecutar las pruebas de rendimiento.*

En esta observación sobre la eficacia de cada requerimiento no se comparan los diferentes algoritmos de ordenamiento usados en la carga de datos. Esto es porque, en virtud de un rendimiento más veloz, los elementos cargados en las listas son organizados previamente a la ejecución de los requisitos. Como nota adicional, se usa Mergesort para organizar cada elemento de listas, algoritmo de ordenamiento el cual es, teóricamente, más veloz en una situación promedio.

## Análisis de complejidad

### Requerimiento 1

Complejidad O(N)

*Para este requerimiento se debía construir un algoritmo que, dado un año inicial y uno final, devolviera los álbumes presentes en este rango de años.*

Independientemente del tipo de lista u organización de la lista que guarda los álbumes, la función comparativa, en su peor caso, tarda O(N) en agregar todos los álbumes dentro del rango de años a la lista retornada. Esto se ilustra a continuación:

```
def albumesPorAnio(catalog, anio_o, anio_f):
    """
    retorna una lista de albumes en in periodo de tiempo
    """
    albums = catalog['albums']
    lista_albums = lt.newList("ARRAY_LIST")
    for album in lt.iterator(albums):
        if (album['release_date'] >= anio_o) and \
           (album['release_date'] <= anio_f):
            lt.addLast(lista_albums, album)
    return lista_albums
```

Esta complejidad se mantiene constante a lo largo del programa, gracias a que la estructura de datos utilizada en los álbumes, guarda un apuntador a una lista que contiene los artistas relacionados con cada álbum, como se ilustra en la siguiente imagen. De esta manera se obtienen todos los datos de los artistas

involucrados, sin necesidad de incrementar la complejidad de la función.

```
def addArtist(catalog, artistdic):
    neoArtist = newArtist(artistdic)
    for album in lt.iterator(catalog["albums"]):
        artists = album["artist_id"].lower()
        if neoArtist['id'].lower() in artists:
            lt.addLast(album['artist_dic'], neoArtist)
    lt.addLast(catalog["artists"], neoArtist)
    return catalog
```

## Requerimiento 2 (Hecho por David Molina)

Complejidad O(1)

*Dado un número de artistas (M), se debía construir un programa que retornara los artistas más populares en el Top M.*

La velocidad de este requerimiento consiste en su totalidad en el orden dado a los datos al realizar su carga. El programa usa mergesort para organizar a los artistas según su popularidad, previamente a la ejecución de este requerimiento.

```
def compareArtists(artist1, artist2):
    if float(artist1["artist_popularity"]) > float(artist2["artist_popularity"]):
        return float(artist1["artist_popularity"]) > float(artist2["artist_popularity"])
    else:
        pass
```

De esta manera, al estar los artistas organizados en una lista según su popularidad, la complejidad de esta función siempre resulta O(1). Esto es por que se conoce la posición de cada artista solicitado.

```
def getTopArtists(list, topN):
    artistas = list["artists"]
    lista_artistas = lt.newList("ARRAY_LIST")
    for pos in range(topN):
        lt.addLast(lista_artistas, lt.getElement(artistas, pos+1))
    return lista_artistas
```

Por otra parte, información relacionada a la canción más popular de cada artista se obtiene al iterar sobre una lista de canciones asignada a cada artista. Esta lista es asignada en la carga de datos e iterar por ella tiene una complejidad de O(1), ya que cada lista solo contiene las canciones respectivas a cada artista.

```
  for el in lt.iterator(artistN["all_tracks"]):
      if el["id"] == artistN["track_id"]:
          nombreCancion = el["name"]
          break
```

## Requerimiento 3 (Hecho por Juan David Salguero)

*Dado un número de canciones (M), se debía construir un programa que retornara las canciones más populares en el Top M.*

Complejidad O(1)

De manera similar al anterior requerimiento, la velocidad de este requerimiento consiste en su totalidad en el orden dado a los datos al realizar su carga. El programa usa mergesort para organizar las tracks según su popularidad, previamente a la ejecución de este requerimiento.

```
def compareTracks(track1, track2):
    try:
        if float(track1["popularity"]) > float(track2["popularity"]):
            return float(track1["popularity"]) > float(track2["popularity"])
        else:
            pass
```

De esta manera, al estar los artistas organizados en una lista según su popularidad, la complejidad de esta función siempre resulta O(1). Esto es por que se conoce la posición de cada artista solicitado.

```
def popularTracks(catalog, top):
    """
    retorna una lista con las N canciones mas populares
    """
    tracks = catalog['tracks']
    lista_tracks = lt.newList("ARRAY_LIST")
    for track in lt.iterator(tracks):
        if lt.size(lista_tracks) < top:
            lt.addLast(lista_tracks, track)
        else:
            break
    return lista_tracks
```

## Requerimiento 4

*Dado un artista y país en específico, se debía retornar los álbumes y canciones disponibles en esa región, encontrando a su vez la canción más popular entre estas.*

Complejidad O(N)

Dado que el parámetro de entrada es un artista y país en específico se deben iterar los datos en busca de los álbumes que cumplan estos dos requerimientos: ser del artista y país especificado. Esto tiene una complejidad de O(N).

```
for album in lt.iterator(albumes):
    # NO ES o(n)^2, ya que solo realiza el segundo loop en el caso de
    # encontrar el artista buscado, y la segunda iteracion es o(1)
    if pais in album["available_markets"]:
        for artist in lt.iterator(album['artist_dic']):
            if (artist["name"].lower() in nombre.lower()):
                lt.addLast(validAlbums, album)
                if lt.size(validArtist) == 0:
                    lt.addLast(validArtist, artist)
                    for track in lt.iterator(artist["all_tracks"]):
                        if pais in track["available_markets"]:
                            lt.addLast(validTracks, track)
```

La segunda iteración (for loop) en la imagen no resulta O(N) Esto es porque la lista iterada solo contiene de 1 a 2 elementos, siendo estos apuntadores posibles artistas asociados al álbum. Para ser más efectivo, también solo se itera en el caso de que el álbum cumpla el primer requerimiento: estar disponible en un país en específico. Esta misma lista [“artist-dic”] permite encontrar en una complejidad de O(1) la información respectiva a cada artista, y en consecuencia, también sus canciones, gracias a los apuntadores contenidos en cada lista de artistas, como fue explicado en el requerimiento 2.

```
def addArtist(catalog, artistdic):
    neoArtist = newArtist(artistdic)
    for album in lt.iterator(catalog["albums"]):
        artists = album["artist_id"].lower()
        if neoArtist['id'].lower() in artists:
            lt.addLast(album['artist_dic'], neoArtist)
    lt.addLast(catalog["artists"], neoArtist)
    return catalog
```

## Requerimiento 5

*Dado el nombre de un artista, se debía retornar todos los álbumes asociados a ese artista, además de las canciones más populares de cada álbum.*

Complejidad O(N)

De manera similar al último requerimiento, se itera el catálogo de álbumes, guardando en una nueva lista a los que pertenezcan al artista requerido . La operación, que depende de la cantidad de álbumes, resulta O(N)

```
def discografiaArtista(catalog, nombreArtista):
    nombre = str(nombreArtista).strip().lower()
    lista_albums = lt.newList("ARRAY_LIST")
    for album in lt.iterator(catalog['albums']):
        for artista in lt.iterator(album['artist_dic']):
            if nombre == artista['name'].strip().lower():
                lt.addLast(lista_albums, album)
    return lista_albums
```

Toda la información adicional requerida, se puede acceder en O(1) desde la lista de artistas presente en cada álbum, y la lista de canciones, presente en cada artista.

## Tabla de pruebas

Teniendo en cuenta la estructura y procedimiento de ejecución relatada previamente, se realiza una prueba empírica donde se busca probar la eficiencia del algoritmo teórica pragmáticamente. Para ello, usando valores de entrada constantes, se ejecutó cada requerimiento 5 veces, en una estructura de datos basada en ARRAY_LIST y de nuevo en LINKED_LIST..

### Resultados

| Número de prueba | Tamaño de la muestra (ARRAY_LIST) | Requerimiento 1 (Tiempo-ms) | Requerimiento 2 (Tiempo-ms) | Requerimiento 3 (Tiempo-ms) | Requerimiento 4 (Tiempo-ms) | Requerimiento 5 (Tiempo-ms) |
|---|---|---|---|---|---|---|
| 1 | 75511 Álbumes | 57.465 | 0.391 | 105.881 | 269.067 | 192.499 |
| 2 | 56129 Artistas | 59.181 | 0.430 | 105.842 | 372.225 | 201.139 |
| 3 | 101940 Canciones | 53.450 | 0.147 | 105.394 | 269.039 | 200.044 |
| 4 | | 55.515 | 0.417 | 104.318 | 271.564 | 201.232 |
| 5 | | 58.506 | 0.413 | 106.135 | 269.370 | 198.808 |

*Tabla 2. Comparación de tiempos de ejecución para los requerimientos 1-6 en la representación arreglo.*

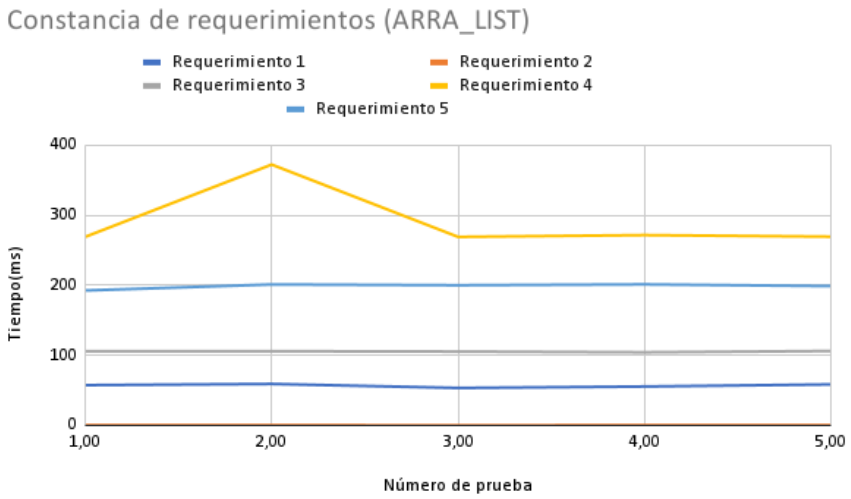
| Número de prueba | Tamaño de la muestra (LINKED_LIST) | Requerimiento 1 | Requerimiento 2 | Requerimiento 3 | Requerimiento 4 | Requerimiento 5 |
|---|---|---|---|---|---|---|
| 1 | 75511 Álbumes | 67.086 | 0.496 | 116.791 | 279.903 | 213.213 |
| 2 | 56129 Artistas | 66.656 | 0.215 | 116.285 | 290.288 | 212.067 |
| 3 | 101940 Canciones | 67.129 | 0.209 | 119.129 | 281.857 | 210.525 |
| 4 | | 65.170 | 0.377 | 122.614 | 285.918 | 210.659 |
| 5 | | 65.571 | 0.207 | 117.283 | 281.369 | 211.834 |

*Tabla 3. Comparación de tiempos de ejecución para los requerimientos 1-6 en la representación lista enlazada.*

## Graficas

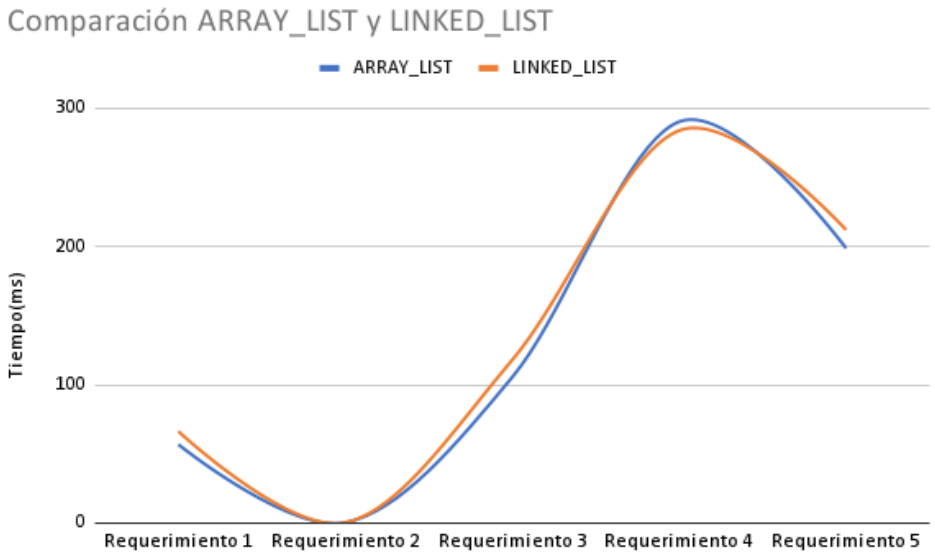
Las siguientes gráficas ilustran los datos planteados en la anterior tabla.

- Comparación de rendimiento ARRAY_LIST.




  - Comparación de rendimiento LINKED_LIST.


  - Comparación de tendencia promedial




## Conclusión y análisis

La diferencia entre usar ARRAY_LIST o LINKED_LIST es mínima en lo que respecta a la efectividad de los requerimientos, como se ilustró en la gráfica anterior. La ejecución de los requerimientos mantienen una velocidad casi idéntica independientemente de la estructura de datos en la que se maneje. No obstante, en consecuencia a la complejidad de las LINKED_LIST, se decidió usar ARRAY_LIST para organizar la información del catálogo, esto se debe al menor tiempo de carga que tiene organizar los datos en ARRAY_LIST.

Por otra parte, los tiempos de ejecución de todos los requerimientos resultan instantáneos, ninguna cuesta más de medio segundo en ser realizado. No obstante, es notable una diferencia, hablando en milisegundos, entre la eficiencia de ejecución de los requerimientos. Por ejemplo, los requerimientos 4 y 5 toman casi 4 veces más tiempo en realizarse que el requerimiento 1, aún cuando, en teoría, los tres requerimientos se deberían comportar como O(n) e iteran sobre la lista de álbumes. Esta diferencia probablemente se debe a la complejidad de la función comparativa aplicada al filtrar cada elemento de la lista iterado.

En comparación con el análisis teórico, en su mayoría los resultados de las pruebas se mantienen idóneos. Los requerimientos 4 y 5, de complejidad O(N) son los algoritmos que toman más tiempo en ejecutarse, en comparación con el requerimiento 2, hipotetizado como O(1). Sin embargo, el requerimiento 3, el cual se cree O(1), toma más tiempo en terminar que el requerimiento 1, creído O(N). Se cree que esto se debe al proceso posterior de datos que se realiza en el requerimiento 3, más complejo que la recolección de datos del requerimiento 1.