ANÁLISIS DE RESULTADOS

Estudiante 1 Cod XXXX
David Samuel Rojas Sánchez - Cod 202214621 - ds.rojass1@uniandes.edu.co
Simón Calderón López Cod 202113559

Ambientes de pruebas

	Máquina 1	Máquina 2	Máquina 3
Procesadores	AMD Ryzen 3 2200G with Radeon™ Vega 8 Graphics @3.5 GHz	M1	Intel Core i5 – 10300H CPU @2.50 GHZ
Memoria RAM (GB)	8.0 GB	16.0 GB	8.0 GB
Sistema Operativo	Windows 10 Home Single Language - 64bits	MacOS Monterey 12.5	Microsoft Windows 11 Home

Especificaciones de las máquinas para ejecutar las pruebas de rendimiento.

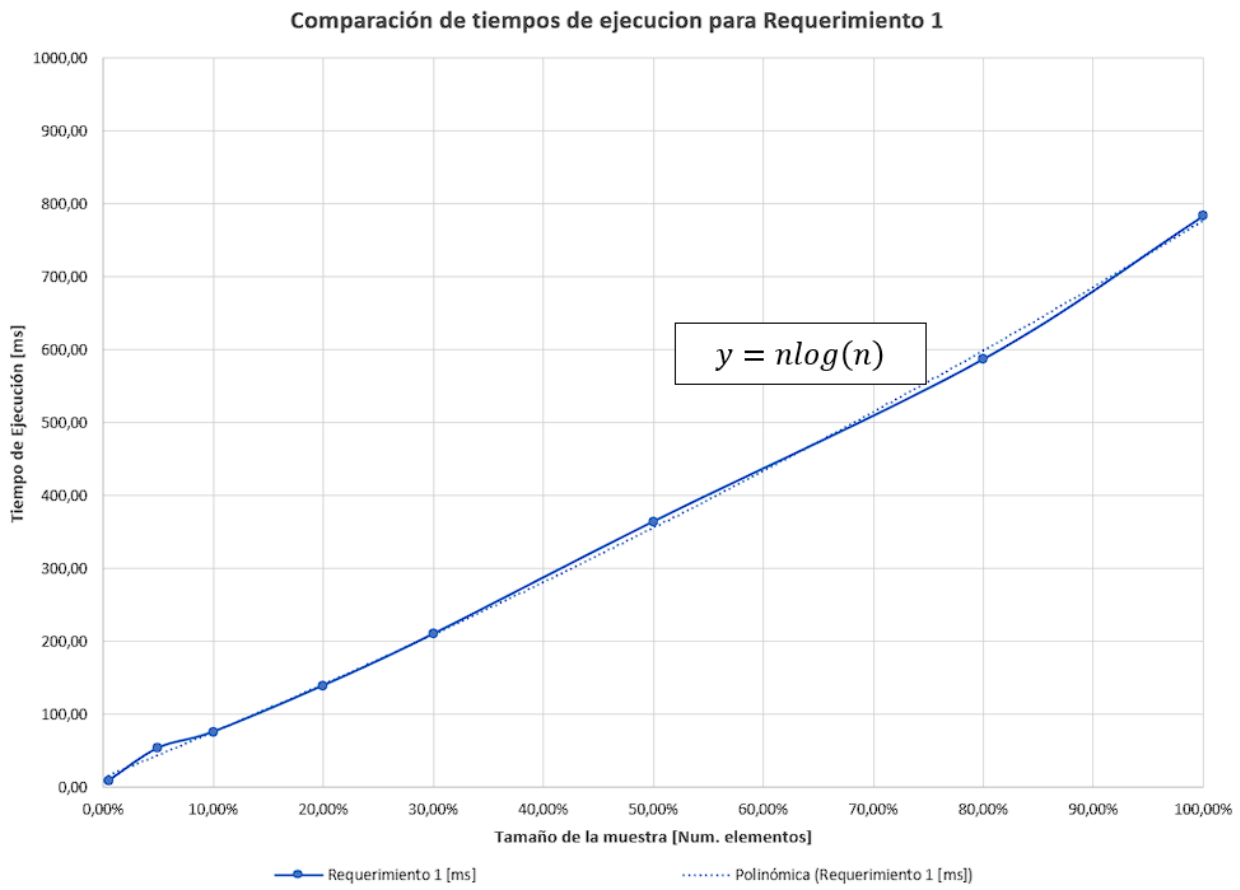
Requerimiento 1: Listar las películas estrenadas en un período de tiempo (Grupal)

```
def getMoviesbyYear(catalog, init_year, final_year):
    sort_movies, time = sortTitles_by_release_year(catalog, 'merge', 'Movie') → nlog(n)
    init_pos = binarySearchMin(sort_movies, init_year, 'release_year') → log(n)
    if init_pos == 0:
        init_pos = 1
    final_pos = binarySearchMax(sort_movies, final_year, 'release_year') → nlog(n)
    cantidad = final_pos - init_pos
    answer_list = lt.subList(sort_movies, init_pos, cantidad)
    return answer_list
```

Dado que la línea más relevante dentro del código posee una complejidad linearítmica, para el peor caso, se estima que la complejidad temporal se acota por la función $O(n\log(n))$

Porcentaje de la muestra [pct.]	Tamaño de la muestra (ARRAY_LIST)	Requerimiento 1 [ms]
0,50%	288,00	9,20
5,00%	1148,00	53,98
10,00%	2298,00	76,18
20,00%	4612,00	139,60
30,00%	6914,00	210,36
50,00%	11521,00	364,40
80,00%	18428,00	587,01
100,00%	23036,00	783,29

Tiempos de ejecución del requerimiento 1 ejecutados en la máquina 1.



Gráfica comparativa de los tiempos de ejecución del requerimiento 1.

Análisis: La complejidad real de la función se comporta de acuerdo a la estimación realizada, pues a medida que los datos crecen, la complejidad temporal del código se acota de forma superior por la función $O(n\log(n))$

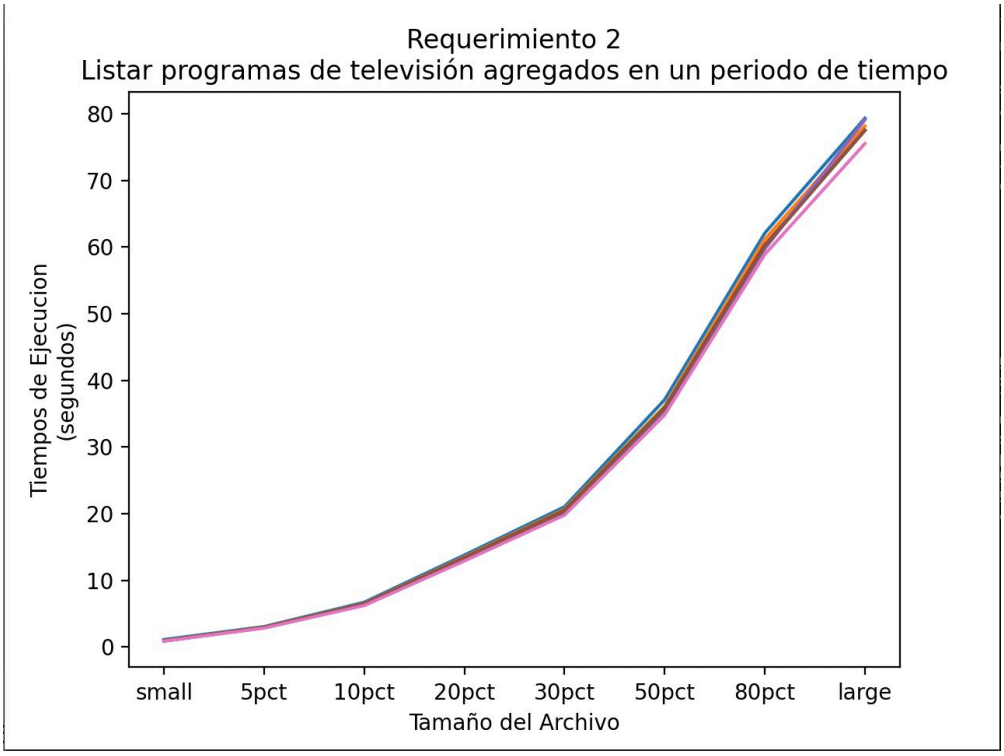
Requerimiento 2: Listar programas de televisión agregados en un periodo de tiempo (Grupal)

```
def getTvSHowsbydate(catalog, init_date, final_date):
    init_date = datetime.datetime.strptime(init_date, "%Y-%m-%d")
    final_date = datetime.datetime.strptime(final_date, "%Y-%m-%d")
    sort_TVShows = sortTV_ShowbyDate(catalog) # se hizo con merge --> nlog(n)
    init_pos = linealSearch(sort_TVShows, final_date, 'date_added') # Busqueda Lineal --> n
    final_pos = linealSearch(sort_TVShows, init_date, 'date_added') # Busqueda Lineal --> n
    cantidad = final_pos - init_pos
    answer_list = lt.subList(sort_TVShows, init_pos, cantidad)
    return answer_list
```

Dado que la línea más relevante dentro del código posee una complejidad linearítmica, para el peor caso, se estima que la complejidad temporal se acota por la función O(n*log(n))

Porcentaje de la muestra [pct.]	Tamaño de la muestra (ARRAY_LIST)	Requerimiento 2 (promedio de pruebas) [ms]
0,50%	288,00	0,96
5,00%	1148,00	2,95
10,00%	2298,00	6,46
20,00%	4612,00	13,34
30,00%	6914,00	20,43
50,00%	11521,00	35,86
80,00%	18428,00	60,35
100,00%	23036,00	77,78

Tiempos de ejecución del requerimiento 2 ejecutados en la máquina 2.



Gráfica comparativa de los tiempos de ejecución del requerimiento 2.

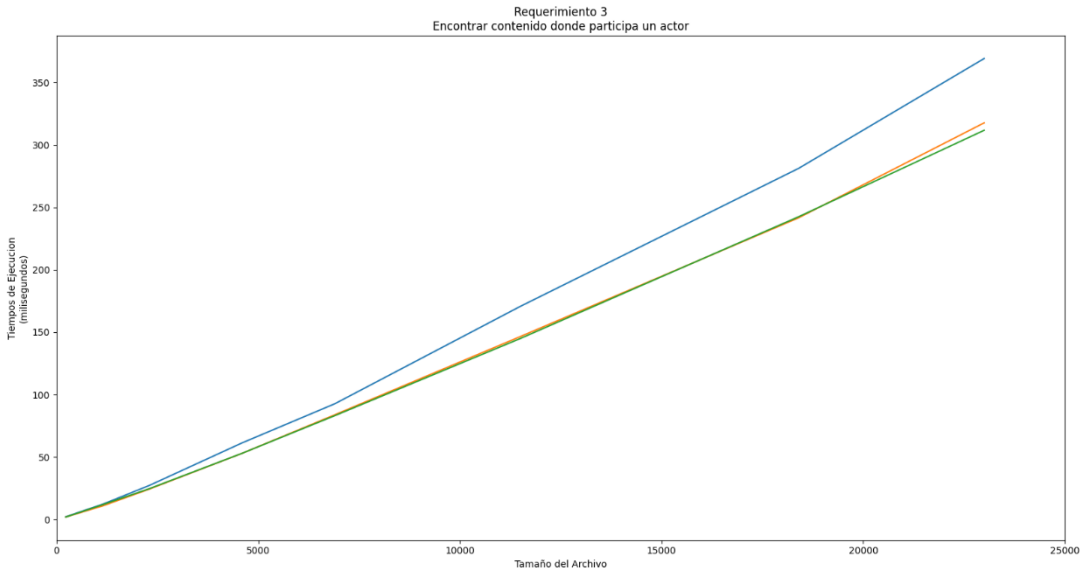
Análisis: La complejidad real de la función se comporta de acuerdo a la estimación realizada, pues a medida que los datos crecen, la complejidad temporal del código se acota de forma superior por la función O(n^2)

Requerimiento 3: Encontrar contenido donde participa un actor (Simón Calderón)

```python
def getActorData(catalog, actor_name):

    actors = catalog['actors']

    pos_inicial = binarySearchMin(actors, actor_name, 'Name') #! → O(logn)
    pos_final = binarySearchMax(actors, actor_name, 'Name') #! → O(logn)

    count = pos_final - pos_inicial
    actor_data = lt.subList(actors, pos_inicial, count)

    actor = organize_actor(actor_data)

    return actor
```

La complejidad más relevante es O(logn), pero la función de organize actor itera sobre una sublista con los elementos del autor, lo cual puede hacer que se haga más grande la complejidad llegando a ser O(nlogn)

Porcentaje de la muestra [pct.]	Tamaño de la muestra (ARRAY_LIST)	Requerimiento 3 (promedio de pruebas) [ms]
0,50%	288,00	2.015
5,00%	1148,00	11.69
10,00%	2298,00	25.397
20,00%	4612,00	55.754
30,00%	6914,00	86.58
50,00%	11521,00	154.089
80,00%	18428,00	255.165
100,00%	23036,00	332.96



Vemos en la gráfica que tiene una tendencia n(logn) lo que concuerda con lo enunciado teóricamente

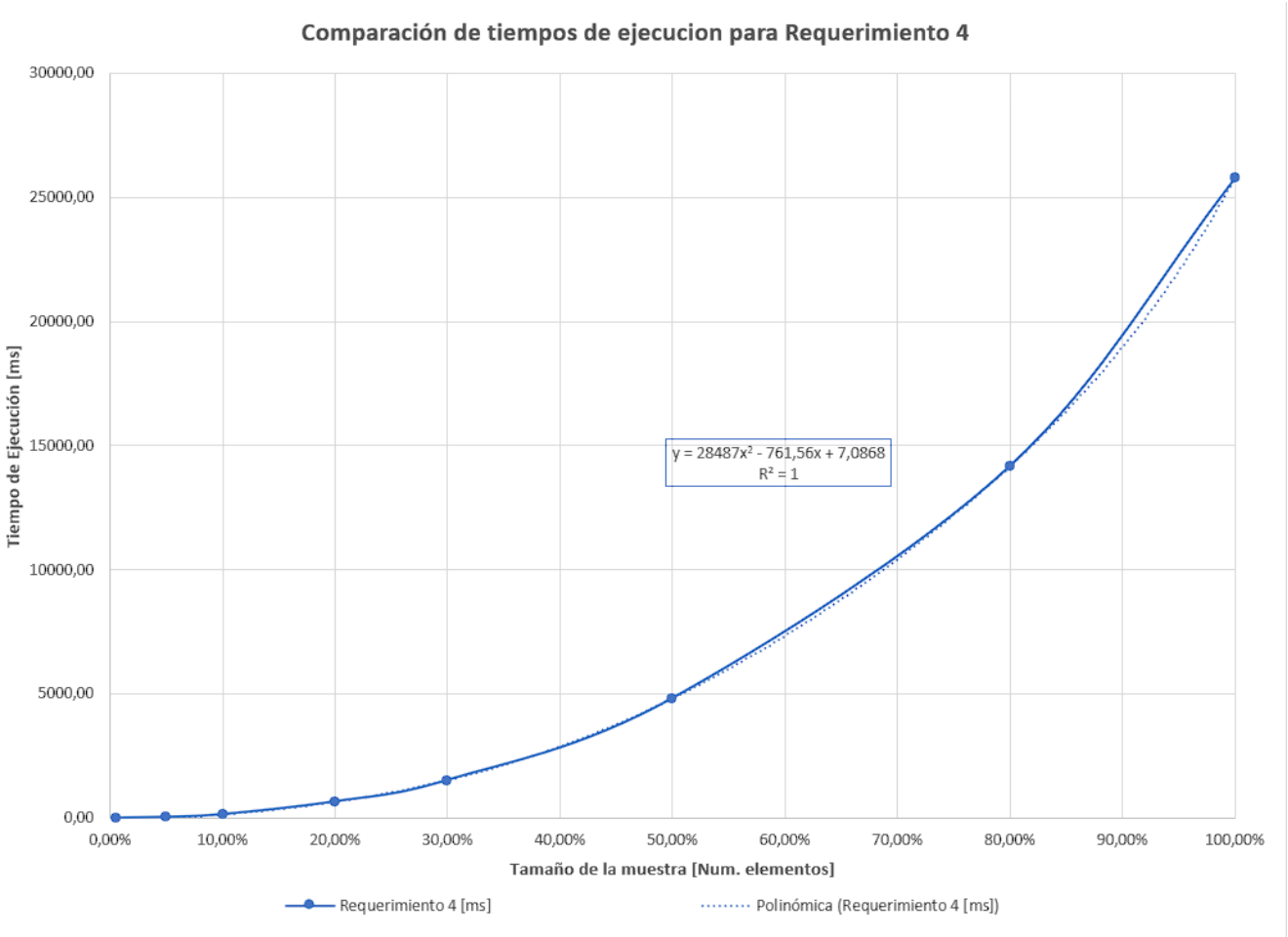
Requerimiento 4: Encontrar contenido por un género específico (David Rojas)

```
def get_titles_by_genre(catalog, genrename):

    sublist = lt.newList("SINGLE_LINKED", cmpTitlesbyTitlesReleaseDirector)
    origin = lt.subList(catalog['titles'], 0, lt.size(catalog['titles']))

    tv_show_count = 0
    movie_count = 0

    for title in lt.iterator(origin): ------------------------------> n^2
        if genrename in title['listed_in']:
            lt.addLast(sublist, title) ---> n
            if title['type'] == "TV Show":
                tv_show_count += 1
            else:
                movie_count += 1

    sublist = sa.sort(sublist, cmpTitlesbyTitlesReleaseDirector) ---> nlog(n)

    return sublist, movie_count, tv_show_count
```

Dado que dentro del bucle hay una línea que añade un elemento al final de una lista encadenada, y que la instrucción de mayor grado es cuadrática, se estima que la complejidad temporal se acota por la función $O(n^2)$

Porcentaje de la muestra [pct.]	Tamaño de la muestra (ARRAY_LIST)	Requerimiento 4 [ms]
0,50%	288,00	1,46
5,00%	1148,00	40,68
10,00%	2298,00	147,30
20,00%	4612,00	662,48
30,00%	6914,00	1527,47
50,00%	11521,00	4825,23
80,00%	18428,00	14173,81
100,00%	23036,00	25767,99

Tiempos de ejecución del requerimiento 4 ejecutados en la máquina 1.



Gráfica comparativa de los tiempos de ejecución del requerimiento 4.

Análisis: La complejidad real de la función se comporta de acuerdo a la estimación realizada, pues a medida que los datos crecen, la complejidad temporal del código se acota de forma superior por la función $O(n^2)$

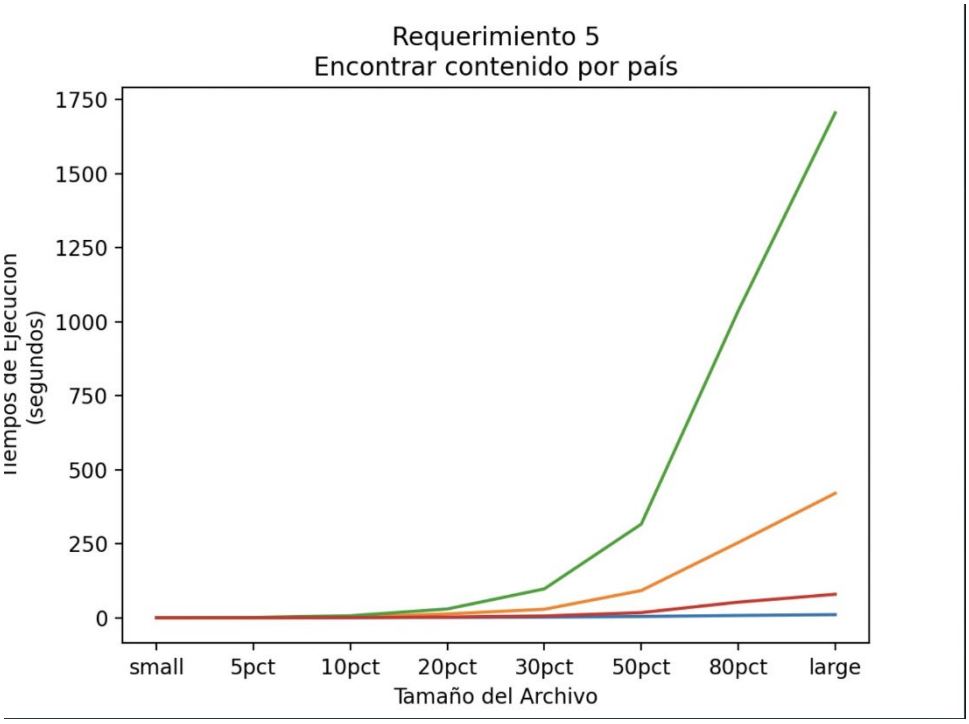
Requerimiento 5: Encontrar contenido producido en un país (Marco Ramírez)

```python
def get_titles_by_country(catalog, countryname):

    sublist = lt.newList("SINGLE_LINKED", cmpMoviesbyTitle)
    origin_copy = lt.subList(catalog['titles'], 0, lt.size(catalog['titles']))

    tv_show_count = 0
    movie_count = 0
    for title in lt.iterator(origin_copy): #--> n
        if (countryname in title['country']) or (countryname.lower() in title['country'].lower()):
            lt.addLast(sublist, title)
            if title['type'] == "TV Show":
                tv_show_count += 1
            else:
                movie_count += 1

    sublist = sa.sort(sublist, cmpMoviesbyTitle) #se ordena con shell sort --> n^(3/2)

    return sublist, movie_count, tv_show_count
```

Dado que la línea más relevante dentro del código posee una complejidad exponencial, para el peor caso, se estima que la complejidad temporal se acota por la función O(n^(3/2))

Porcentaje de la muestra [pct.]	Tamaño de la muestra (ARRAY_LIST)	Requerimiento 5 (promedio de pruebas) [ms]
0,50%	288,00	0,53
5,00%	1148,00	0,88
10,00%	2298,00	2.94
20,00%	4612,00	11,62
30,00%	6914,00	33,83
50,00%	11521,00	104,77
80,00%	18428,00	323,31
100,00%	23036,00	523,411

Tiempos de ejecución del requerimiento 5 ejecutados en la máquina 2.



Gráfica comparativa de los tiempos de ejecución del requerimiento 5.

Análisis: La complejidad real de la función se comporta de acuerdo a la estimación realizada, pues a medida que los datos crecen, la complejidad temporal del código se acota de forma superior por la función O(n^2)

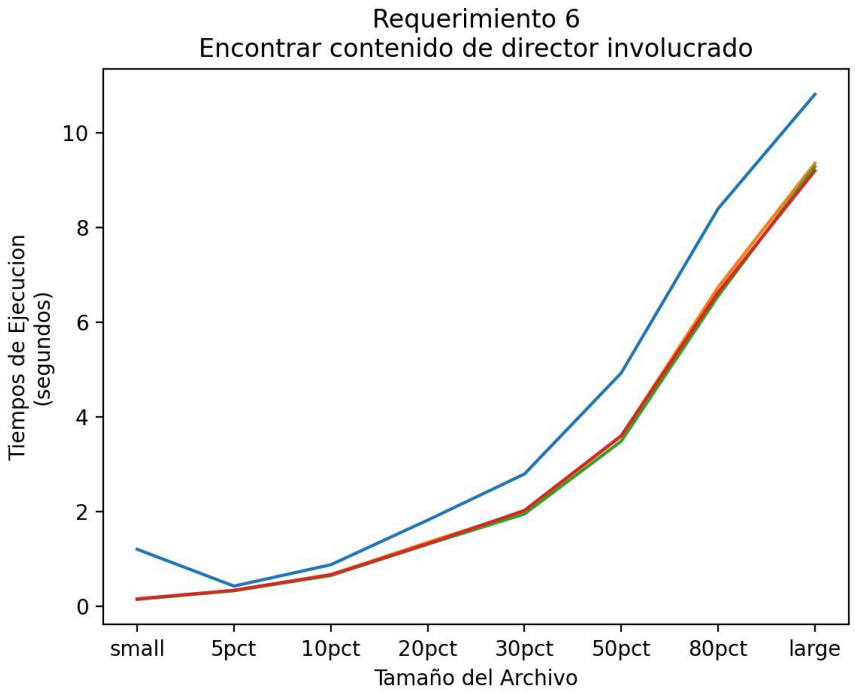
Requerimiento 6: Encontrar contenido con un director involucrado (Grupal)

```python
def get_titles_by_director(catalog, directorname):

    sublist = lt.newList("SINGLE_LINKED", cmpMoviesByReleaseYear)
    origin_copy = lt.subList(catalog['titles'], 0, lt.size(catalog['titles']))

    tv_show_count = 0
    movie_count = 0
    netflix_count_movie= 0
    amazon_count_movie= 0
    hulu_count_movie= 0
    disney_count_movie= 0
    netflix_count_tv= 0
    amazon_count_tv= 0
    hulu_count_tv= 0
    disney_count_tv= 0
    for title in lt.iterator(origin_copy):
        for director_str in title["director"]: #--> n(log(n))
            if (directorname in director_str):
                lt.addLast(sublist, title)
                if title['type'] == "TV Show":
                    tv_show_count += 1
                    if title["platform"]=="netflix":
                        netflix_count_tv +=1
                    elif title["platform"]=="hulu":
                        hulu_count_tv +=1
                    elif title["platform"]=="amazon_prime":
                        amazon_count_tv +=1
                    elif title["platform"]=="disney_plus":
                        disney_count_tv +=1
                else:
                    movie_count += 1
                    if title["platform"]=="netflix":
                        netflix_count_movie +=1
                    elif title["platform"]=="hulu":
                        hulu_count_movie +=1
                    elif title["platform"]=="amazon_prime":
                        amazon_count_movie +=1
                    elif title["platform"]=="disney_plus":
                        disney_count_movie +=1

    sublist = sa.sort(sublist, cmpMoviesByReleaseYear) # Ordenado con shell sort --> n^(3/2)
    #ACÁ SE CREA LA LISTA CON CADA PLATAFORMA Y SUS CONTADAS
    platforms_count = lt.newList("SINGLE_LINKED", cmpMoviesByReleaseYear)
    lt.addLast(platforms_count,{"platform":"Netflix","Movie":netflix_count_movie,"TV Shows":netflix_count_tv})
    lt.addLast(platforms_count,{"platform":"Hulu","Movie":hulu_count_movie,"TV Shows":hulu_count_tv})
    lt.addLast(platforms_count,{"platform":"Amazon Prime","Movie":amazon_count_movie,"TV Shows":amazon_count_tv})
    lt.addLast(platforms_count,{"platform":"Disney Plus","Movie":disney_count_movie,"TV Shows":disney_count_tv})
```

Dado que la línea más relevante dentro del código posee una complejidad exponencial, para el peor caso, se estima que la complejidad temporal se acota por la función O(n^(3/2))

Porcentaje de la muestra [pct.]	Tamaño de la muestra (ARRAY_LIST)	Requerimiento 6 (promedio de pruebas) [ms]
0,50%	288,00	0,42
5,00%	1148,00	0,35
10,00%	2298,00	0,7
20,00%	4612,00	1,45
30,00%	6914,00	2,19
50,00%	11521,00	3,9
80,00%	18428,00	7,0
100,00%	23036,00	9,66

Tiempos de ejecución del requerimiento 6 ejecutados en la máquina 2.



Gráfica comparativa de los tiempos de ejecución del requerimiento 6.

Análisis: La complejidad real de la función **NO** se comporta de acuerdo a la estimación realizada, pues a medida que los datos crecen, la complejidad temporal del código decrece, indicando que, en el caso promedio, su complejidad es O(n*log(n))

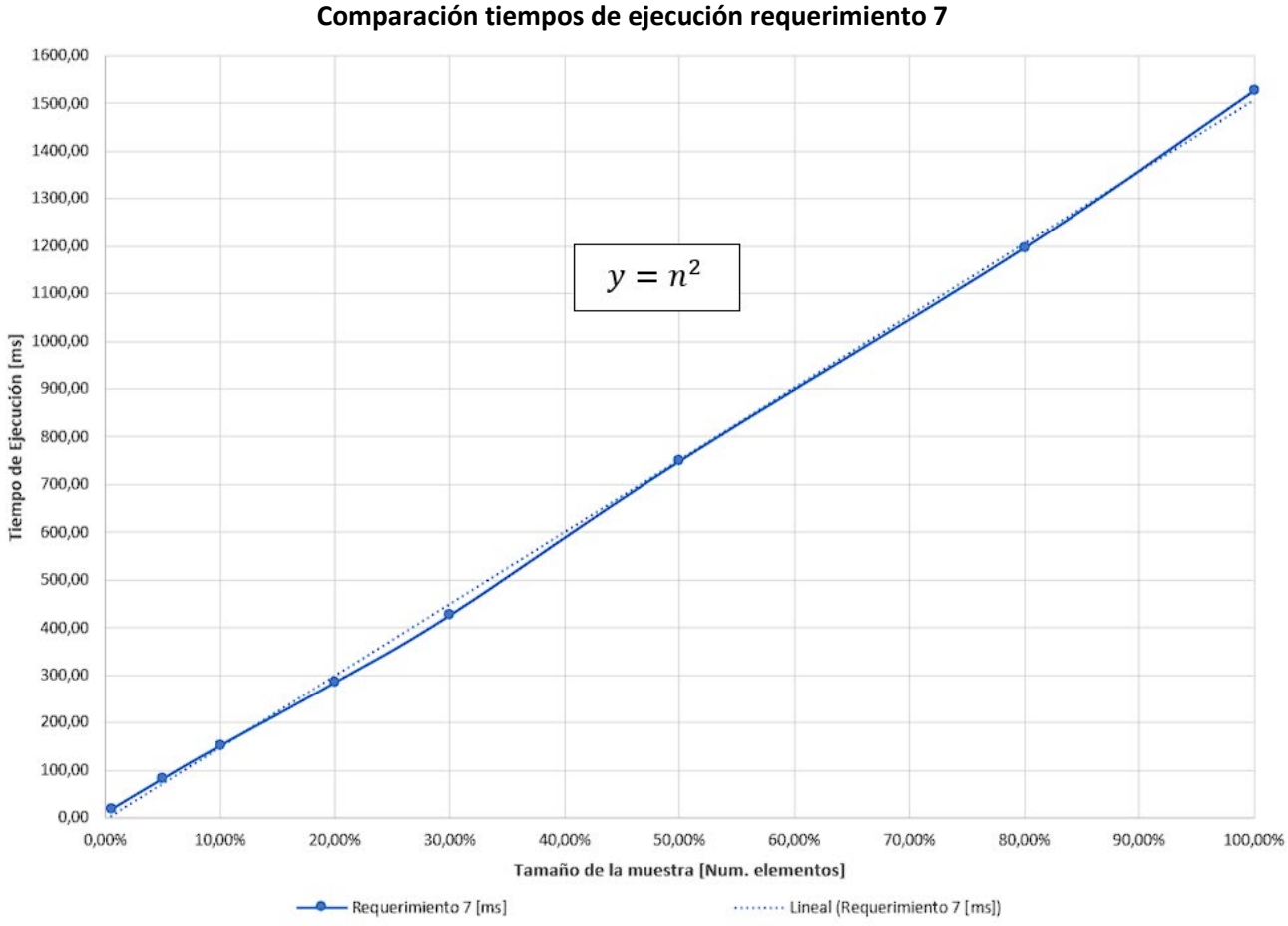
Requerimiento 7: Listar el TOP (N) de los géneros con más contenido (Grupal)

```
def getTopGenres(catalog, topSize):

    sortedGenres, time = merge_sort(catalog["genres"], cmpPersonbyName) ⟶ nlog(n)
    genreData = lt.getElement(sortedGenres, 1)
    genreName = genreData['Name']

    genreInfo = {"Name" : genreData['Name'],
                 "count" : 0,
                 "Movie" : 0,
                 "TV Show" : 0,
                 "amazon_prime" : 0,
                 "netflix" : 0,
                 "hulu" : 0,
                 "disney_plus" : 0}

    topGenres = lt.newList(datastructure='SINGLE_LINKED', cmpfunction=cmpbyName)

    for genre in lt.iterator(sortedGenres): ⟶ n^2
        if genreName == genre['Name']:
            genreInfo['count'] += 1
            genreInfo[genre['type']] += 1
            genreInfo[genre['platform']] += 1
        else:
            lt.addLast(topGenres, genreInfo) ⟶ n
            genreName = genre['Name']
            genreInfo = {"Name" : genre['Name'],
                         "count" : 1,
                         "Movie" : 0,
                         "TV Show" : 0,
                         "amazon_prime" : 0,
                         "netflix" : 0,
                         "hulu" : 0,
                         "disney_plus" : 0}
            genreInfo[genre['type']] += 1
            genreInfo[genre['platform']] += 1

    topGenres, time = merge_sort(topGenres, cmp_function=cmpActorsbyCount) ⟶ nlog(n)

    return lt.subList(topGenres, 1, topSize)
```

Dado que dentro del bucle cabe la posibilidad de agregar un elemento al final de una lista encadenada, existe un crecimiento temporal cuadrático. Por lo tanto, para el peor caso, se estima que la complejidad temporal se acota por la función $O(n^2)$

Porcentaje de la muestra [pct.]	Tamaño de la muestra (ARRAY_LIST)	Requerimiento 7 [ms]
0,50%	288,00	18,55
5,00%	1148,00	83,16
10,00%	2298,00	152,30
20,00%	4612,00	285,24
30,00%	6914,00	426,48
50,00%	11521,00	749,44
80,00%	18428,00	1196,50
100,00%	23036,00	1526,34

Tiempos de ejecución del requerimiento 7 ejecutados en la máquina 1.



Gráfica comparativa de los tiempos de ejecución del requerimiento 7.

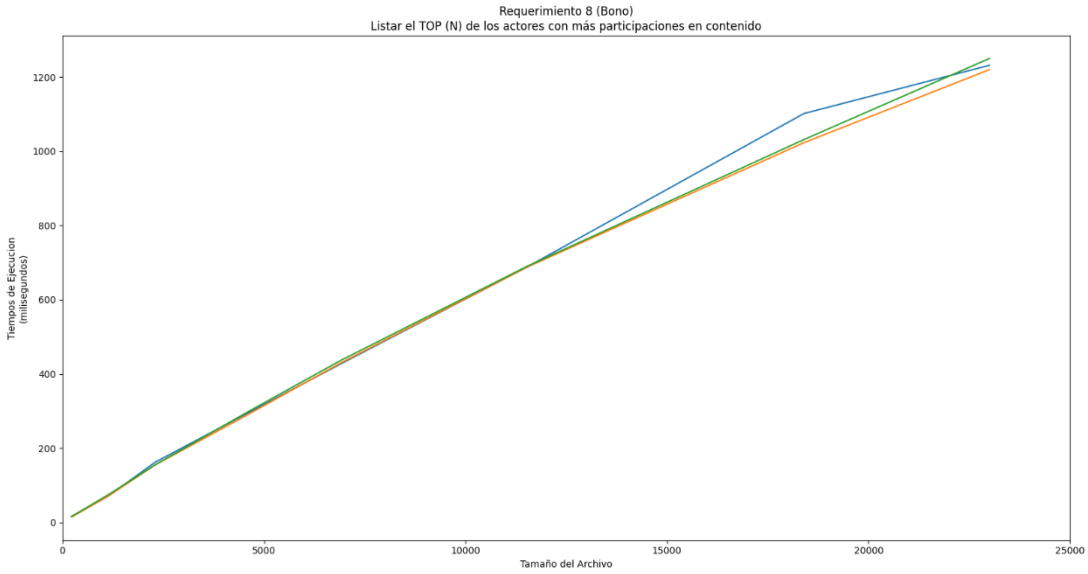
Análisis: La función no parece ser acotada del todo por el orden de complejidad $O(n^2)$**.** Esto puede deberse a que en la ejecución del requerimiento no se alcanzó el peor caso, pues para que este se dé se deberían tener n géneros distintos y solicitar el TOP n de géneros con más contenido. Tal caso no es posible con los datos de las plataformas digitales. Sin embargo, gracias al análisis de complejidad podemos tener la certeza de que en el peor caso la función que acota el crecimiento temporal en función del volumen de datos será $O(n^2)$.

Requerimiento 8: Listar el TOP (N) de los actores con más participaciones en contenido (Bono)

```python
def getTopActors(catalog, topSize):

    sort_Actors = catalog['actors']
    actor_data = lt.getElement(sort_Actors, 1)
    actor_name = actor_data['Name']

    actor_info = {"Name": actor_name,
                  "count": 0}

    top_actors = lt.newList(datastructure='ARRAY_LIST', cmpfunction=cmpbyName)

    for actor in lt.iterator(sort_Actors): #! → O(n)
        if actor_name == actor["Name"]:
            actor_info['count'] += 1
        else:
            lt.addLast(top_actors,actor_info)
            actor_name = actor["Name"]
            actor_info = {"Name" : actor_name, "count": 1}


    top_actors, time = merge_sort(top_actors, cmp_function=cmpActorsbyCount) #! → O(nlogn)
    size = lt.size(top_actors) + 1
    top_actors = lt.subList(top_actors, 1, topSize)
    datatop = lt.newList(datastructure='SINGLE_LINKED', cmpfunction=cmpbyName)
    for actor in lt.iterator(top_actors): #! → O(n)
        actor_name = actor['Name']
        data_actor = getActorData(catalog, actor_name) #→ O(logn)
        top_listed_in, time = shell_sort(data_actor['categories'], cmpActorsbyCount)
        top_listed_in = lt.getElement(top_listed_in, 1)
        data_actor['top_listed_in'] = top_listed_in
        data_actor['count'] = actor['count']
        lt.addLast(datatop, data_actor)
    return datatop, size
```

La línea con la complejidad más relevante es O(nlogn) por un ordenamiento con merge sort de todos los datos de los actores

Porcentaje de la muestra [pct.]	Tamaño de la muestra (ARRAY_LIST)	Requerimiento 8 [ms]
0,50%	288,00	15.48
5,00%	1148,00	72.894
10,00%	2298,00	157.258
20,00%	4612,00	294.216
30,00%	6914,00	430.649
50,00%	11521,00	686.991
80,00%	18428,00	1051.965
100,00%	23036,00	1233.705



Vemos que el comportamiento, efectivamente parece un n(logn) por tanto el análisis es correcto