RETO_001

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

Daniel Vargas – 201822068 - REQ 3

Numero de artistas: N

Numero de obras: M

REQUISITO1: listar cronológicamente los artistas

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

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

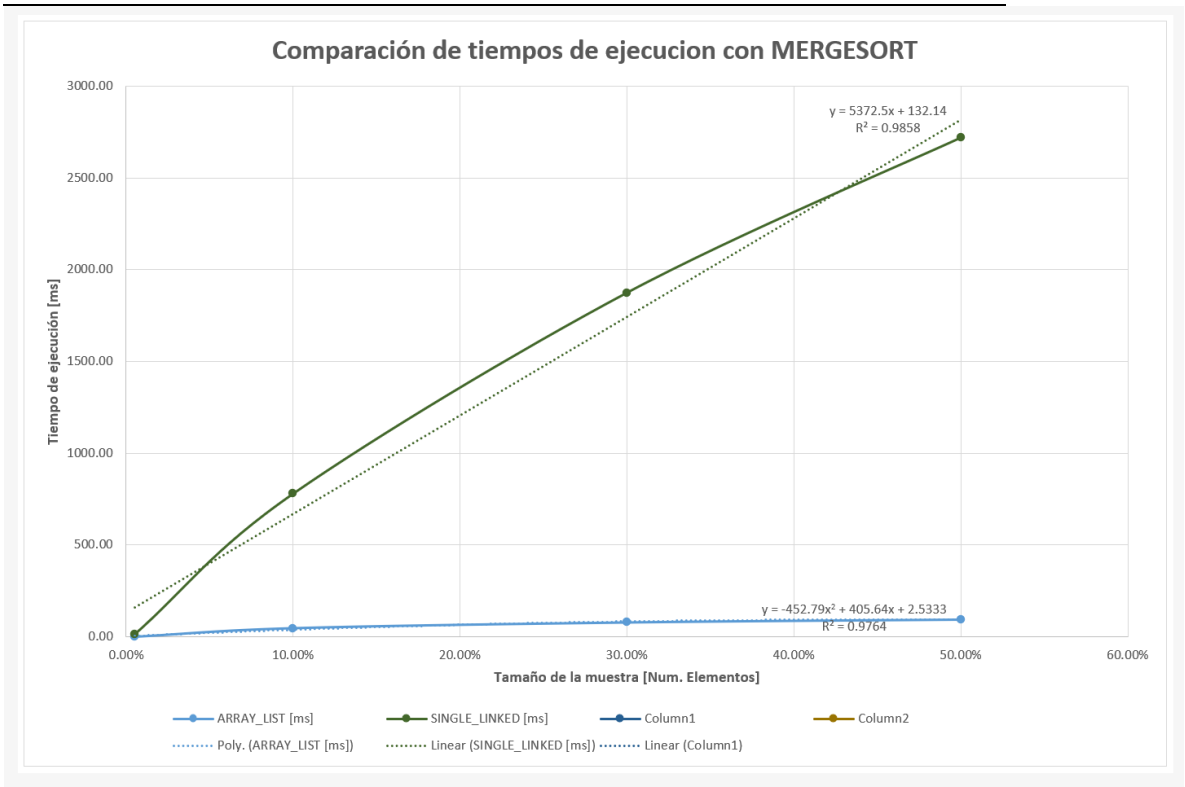
Para la primera parte, se crea una lista vacía con la implementación ARRAY_LIST, ya que esta lista tiene una mejor eficiencia a la hora de implementar un algoritmo de sort. Esta función luego hace un recorrido comparando y añadiendo artistas a la nueva lista vacía, lo cual tiene un orden de crecimiento en el peor de los casos de 0(N) (si todos los artistas están dentro del rango de comparación).

Para la segunda parte, se utiliza el algoritmo de ordenamiento MERGE_SORT para ordenar la sublista ingresada. Se eligió este método de ordenamiento ya que tiene el menor orden en el peor de los casos O(nlog(n)). Vale la pena nota que, al ingresar una sublista gracias a la función anterior, buscamos reducir en el máximo el tiempo de ordenamiento ya que tendría significativamente menos datos para ordenar que si ingresaramos la lista de artistas completos.

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

Haciendo cambios en los tamaños de las listas csv, y el tipo de implementación para la sublista, podemos verificar que el ARRAY_LIST es mucho mas eficiente que el SINGLE_LINKED a la hora de hacer el ordenamiento.

Porcentaje de la muestra [pct]	Tamaño de la muestra (artistas)	ARRAY_LIST [ms]	SINGLE_LINKED [ms]
small	1948	0.0	15.625
10%	6656	46.875	781.25
30%	10063	78.125	1875.0
50%	12137	93.750	2718.75



REQUISITO 2: listar cronológicamente las adquisiciones

Para este requisito, se utilizan 3 procedimientos. El primero se utiliza para obtener una sublista de las obras de arte limitadas al rango de fechas indicado. El segundo hace un recorrido por esta sublista evaluando si la obra fue comprada. El último ordena la sublista según la fecha de adquisición de las obras.

```
81  def sortArtworksByBeginDate(catalog, year1, year2):
82
83      artworksInRange = model.getArtworksInDateRange(catalog, year1, year2)
84      purchasedAmount = model.purchasedAmount(artworksInRange)
85      sortedResult = model.sortArtworks(artworksInRange)
86      return sortedResult,purchasedAmount
87
```

Para la primera parte, se crea una lista vacía con la implementación ARRAY_LIST, ya que esta lista tiene una mejor eficiencia a la hora de implementar un algoritmo de sort. Esta función luego hace

un recorrido comparando y añadiendo obras de arte a la nueva lista vacía, lo cual tiene un orden de crecimiento en el peor de los casos de 0(m) (si todas las obras estuvieran dentro del rango de comparación).

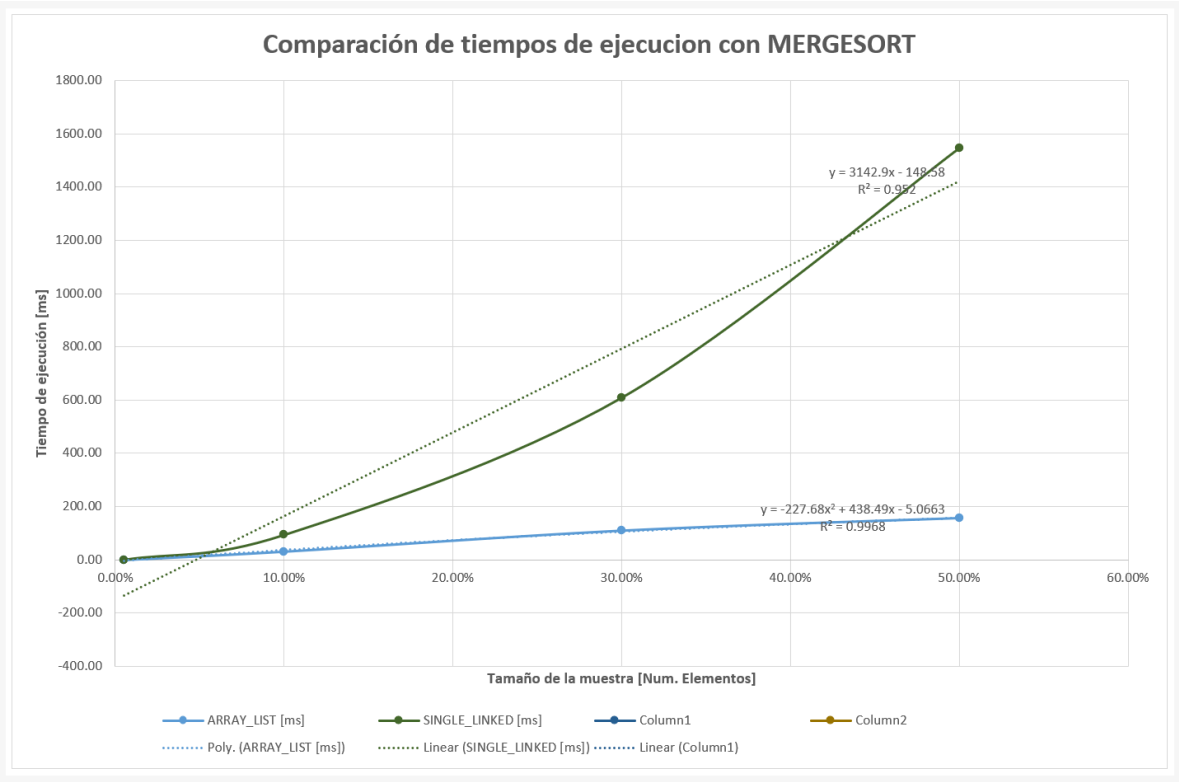
Para la segunda parte, se ingresa la sublista como parámetro, y se recorre para buscar el valor de "CreditLine" de cada obra, comparándolo para saber si fue comprado o no, sumando en un contador cada que esto sea cierto. Como hace un recorrido, en el peor de los casos tendría una complejidad de O(m), en caso de que la sublista tuviera el mismo tamaño que la lista original de obras.

Para la tercera parte, se utiliza el algoritmo de ordenamiento MERGE_SORT para ordenar la sublista ingresada. Se eligió este método de ordenamiento ya que tiene el menor orden en el peor de los casos O(mlog(m)). Vale la pena nota que, al ingresar una sublista gracias a la primera función, buscamos reducir en el máximo el tiempo de ordenamiento y búsqueda del "purchased" ya que tendría significativamente menos datos para ordenar que si ingresaramos la lista de obras completos.

En el paso final, de buscar datos de las primeras 3 y últimas 3 posiciones de la lista ordenada, el hecho que sea un ARRAY_LIST también hace más eficiente la búsqueda según posición, ya que no se requiere recorrer toda la lista para obtener las ultimas posiciones, como tocaría hacer con la implementación SINGLE_LINKED. Para poder mostrar el nombre de los artistas de estas 6 obras, se implemento otra función que recorriera el catalogo de artistas por cada valor del "constituentID" de cada obra, y retornar el nombre cuando coincidieran. En esta función, nos aseguramos de implementar un quiebre para que el recorrido se detuviera una vez encontrara el nombre, pero en el peor de los casos igual tendría que hacer un recorrido de O(n) por cada constituentID. Teniendo todo esto en mente, el mayor orden de crecimiento en el peor de los casos para este requisito con nuestra implementación sería O(nlog(n)).

Haciendo pruebas de ejecución, cambiando el tamaño de datos y la implementación del TAD lista para la sublista de obras, podemos verificar que nuevamente ARRAY_LIST es mucho mas eficiente con este algoritmo de ordenamiento.

Porcentaje de la muestra [pct]	Tamaño de la muestra (obras)	ARRAY_LIST [ms]	SINGLE_LINKED [ms]
small	768	0.0	0.0
10%	15008	31.25	93.75
30%	43704	109.375	609.375
50%	71432	156.25	1546.875



REQUISITO 3 (Daniel Vargas):

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

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

```python
def newNationlity(nationality):
    """
    Crea una nueva estructura para almacenar las obras de una nacionalidad
    """
    country = {'name': "", "artworks": None, "size" : 0 }
    country['name'] = nationality
    country['artworks'] = lt.newList('ARRAY_LIST')
    return country
```

```python
def sortCountries(catalog):

    countries = catalog['nationalities']
    sortedResult = model.sortCountries(countries)
    return sortedResult

```

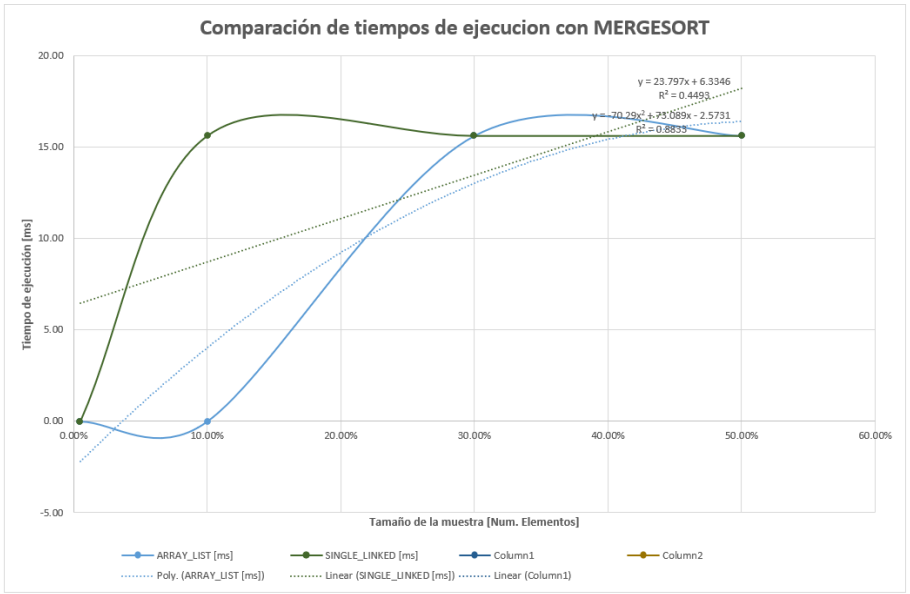
Para la primera parte, decidí cargar los datos a el catalogo de nacionalidades desde la función de carga al inicio de la aplicación, utilizando una estructura de **ARRAY_LIST** para optimizar el eventual proceso de sort. Esto se hace con cada entrada del csv de obras de arte, donde se utiliza una lista de “constituentIDs” para buscar la nacionalidad de cada artista en el catalogo de artistas. Para hacer esto, implemente una función donde solo se tiene que hacer un recorrido parcial por la lista de artistas por cada entrada de obras de arte, haciendo comparaciones en simultaneo con el grupo de IDs en caso de haber más de un artista por obra. De esta manera, se reduce el numero de recorridos, y la longitud de estos, que tienen que hacerse sobre la lista de artistas para cargar la estructura de datos de nacionalidades. Aun con esto, si asumimos el peor caso de que todas las obras incluyan al último artista en el catálogo, la complejidad de esta carga sería de O(M*N), ya que haría un recorrido completo de artistas (N) por cada valor de obras (M).

Para la segunda parte, solo se tiene que hacer un sort sobre la estructura de nacionalidades, la cual ya contiene el numero de obras como una pareja llave-valor, utilizando una función de comparación y el algoritmo MERGE SORT.

Por último, para poder mostrar el nombre de los artistas de 6 obras dentro del país con más obras, se implementó otra función que recorriera el catálogo de artistas por cada valor del “constituentID” de cada obra, y retornar el nombre cuando coincidieran. En esta función, nos aseguramos de implementar un quiebre para que el recorrido se detuviera una vez encontrara el nombre, pero en el peor de los casos igual tendría que hacer un recorrido de O(n) por cada constituentID. Teniendo todo esto en mente, el mayor orden de crecimiento en el peor de los casos para este requisito con nuestra implementación sería O(M*N), al momento de cargar los datos de nacionalidades.

Para analizar si el cambio de la implementación del catalogo [‘nationalities´] afectaba los tiempos de procesamiento para el sort, se realizaron pruebas de rendimiento con ambos tipos de lista.

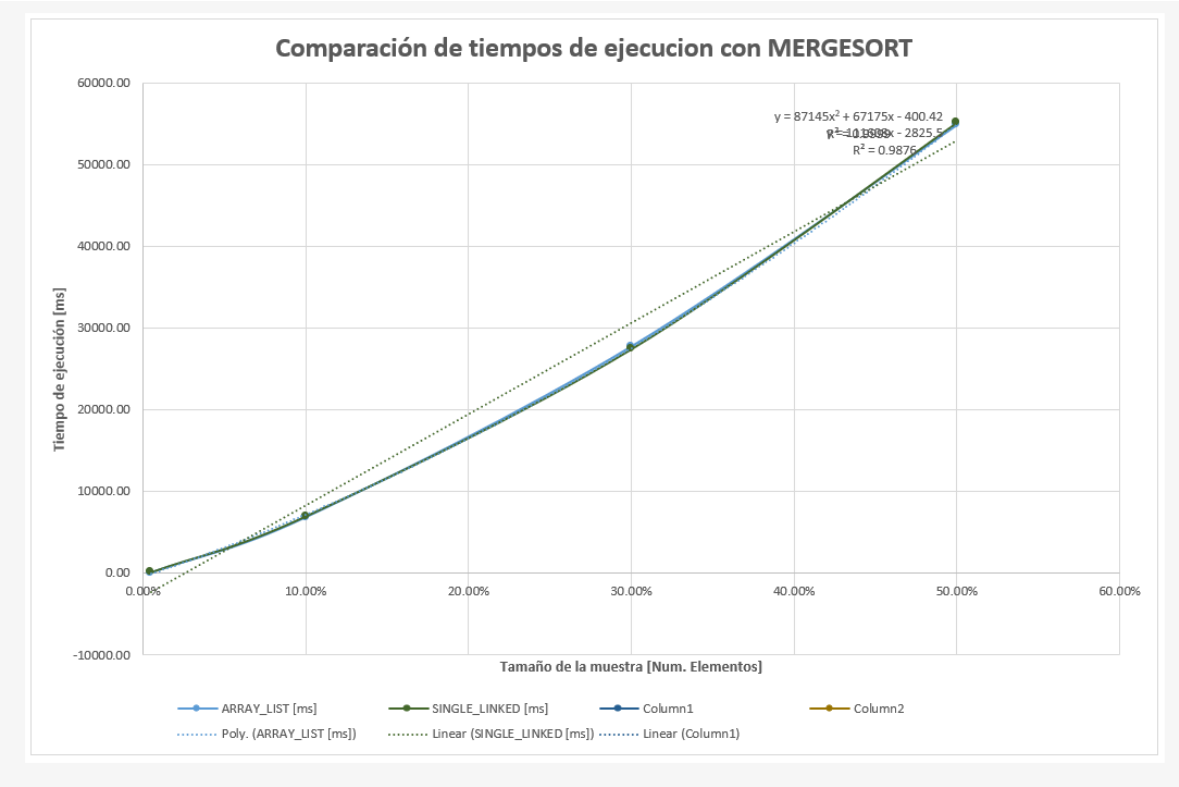
Porcentaje de la muestra [pct]	Tamaño de la muestra (países en el sort)	ARRAY_LIST [ms]	SINGLE_LINKED [ms]
small	48	0.0	0.0
10%	92	0.0	15.625
30%	105	15.625	15.625
50%	112	15.625	15.625



Aquí, podemos ver unos resultados poco conclusivos, en parte porque el tamaño de elementos con el cual se hace el sort (países) es bastante pequeño, y no varía mucho por cada tamaño de datos que se utilizara. Sin embargo, por las conclusiones de requisitos pasados, podemos suponer que, en caso de generar una lista mucho más grande, la implementación de ARRAY_LIST tendría mejores tiempos de ejecución utilizando MERGE_SORT.

Haciendo pruebas de rendimiento con el proceso de cargado de datos, podemos ver que nos dan resultados muy parecidos para ambos tipos de estructuras para el catalogo [‘nationalities’], ya que las únicas operaciones que se hacen sobre esta lista es un recorrido, y un addLast, que tienen la misma complejidad para ambos tipos de implementación.

Porcentaje de la muestra [pct]	Tamaño de la muestra (obras cargadas)	ARRAY_LIST [ms]	SINGLE_LINKED [ms]
small	768	109.375	125.0
10%	15008	6875.0	6984.375
30%	43704	27796.875	27437.5
50%	71432	54921.875	55156.25



REQUISITO 5: transportar obras de un departamento

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

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

Para la primera parte, se genera una sublista de obras con implementación ARRAY_LIST, ya que esta es más eficiente a la hora de hacer un sort. Luego, se hace un recorrido por la lista inicial, y por cada obra que coincida con el departamento buscado se añade a la nueva lista, se le calcula el peso y el costo y se les asocia el costo a los datos de la obra, sumándolos luego a un contador de totales. El calculo de peso y costo por cada obra tiene una complejidad de O(1). El recorrido por la lista original y la inclusión a la nueva lista tiene una complejidad de O(m), siendo m el tamaño de la lista original. Aquí vemos que, en el peor de los casos, si todas las obras fueran del departamento buscado, tendría que hacer muchas más operaciones de búsqueda de costo y peso, sin embargo, al ser estas de orden de crecimiento constante, este componente no tendría un efecto tan grande

en los tiempos de ejecución. Lo que si podemos ver es que el hecho de tener la sublista va a reducir significantemente la cantidad de elementos de los cuales se tendrá que hacer sort más adelante.

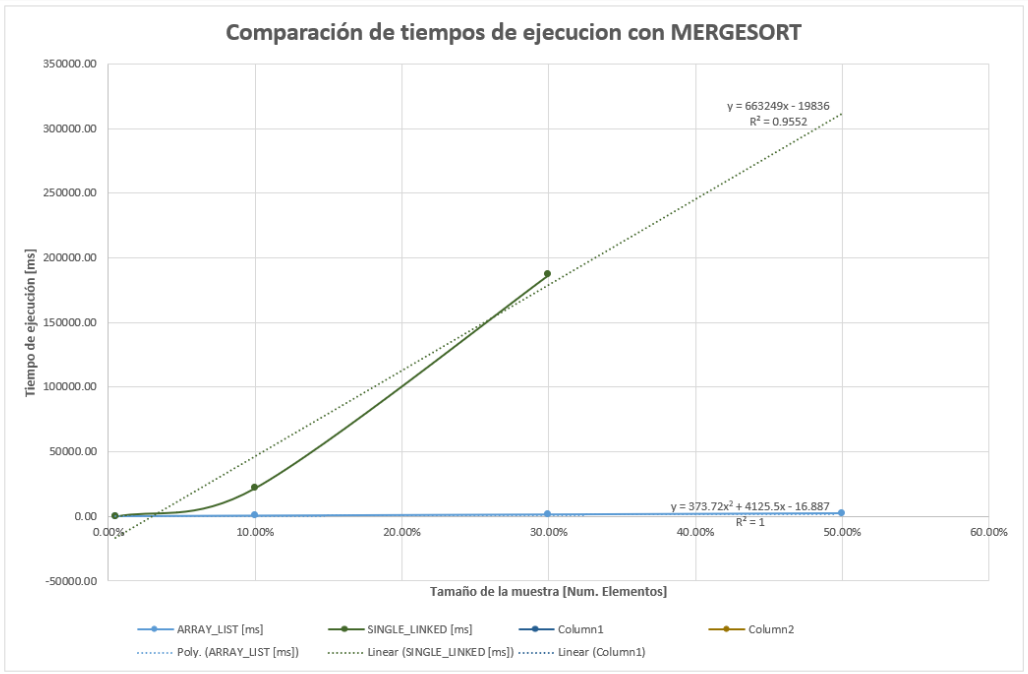
Para la segunda parte y tercera parte, se utiliza el algoritmo de ordenamiento MERGE_SORT para ordenar la sublista ingresada. Se eligió este método de ordenamiento ya que tiene el menor orden en el peor de los casos O(mlog(m)).

En el paso final, de buscar datos de las primeras 5 posiciones de la lista ordenada, el hecho que sea un ARRAY_LIST también hace ligeramente más eficiente la búsqueda según posición, ya que no se requiere recorrer la lista para obtener las posiciones posteriores a la primera, como tocaría hacer con la implementación SINGLE_LINKED. Para poder mostrar el nombre de los artistas de estas 5 obras, se utilizó la función que recorriera el catálogo de artistas por cada valor del "constituentID" de cada obra, y retornar el nombre cuando coincidieran. En esta función, nos aseguramos de implementar un quiebre para que el recorrido se detuviera una vez encontrara el nombre, pero en el peor de los casos igual tendría que hacer un recorrido de O(n) por cada constituentID.

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

Para verificar esto, hicimos pruebas de ejecución sobre archivos de distintos tamaños, cambiando el tipo de implementación de la sublista de obras, donde pudimos verificar que ARRAY_LIST es mucho mas eficiente que SINGLE_LINKED.

Porcentaje de la muestra [pct]	**Tamaño de la muestra (obras en departamento)**	**ARRAY_LIST [ms]**	**SINGLE_LINKED [ms]**
small	394	0.0	62.5
10%	8133	406.25	22031.25
30%	23709	1250.0	187015.625
50%	38888	2140.625	-


Comparación de tiempos de ejecucion con MERGESORT
350000.00
300000.00
250000.00
200000.00
150000.00
100000.00
50000.00
0.00
-50000.00
Tiempo de ejecución [ms]
y = 663249x - 19836
R² = 0.9552
y = 373.72x² + 4125.5x - 16.887
R² = 1
0.00%
10.00%
20.00%
30.00%
40.00%
50.00%
60.00%
Tamaño de la muestra [Num. Elementos]
ARRAY_LIST [ms]
SINGLE_LINKED [ms]
Column1
Column2
Poly. (ARRAY_LIST [ms])
Linear (SINGLE_LINKED [ms])
Linear (Column1)