Reto 1 – Análisis de algoritmos

Req. 3 - Carlos Arturo Holguín Cárdenas, 202012385, c.holguinc@uniandes.edu.co

Req. 4 - Daniel Hernández Pineda, 202013995, d.hernandez24@uniandes.edu.co

IMPORTANTE: Con la autorización del profesor, se hizo uso de listas nativas de Python en el archivo view.py para poder hacer uso de la librería "tabulate". Para que esta librería funcione adecuadamente, se recomienda usar la versión 3.7 de Python en el intérprete, además de instalarla con el comando "pip install tabulate" en caso de no tenerla instalada

Carga de Datos

Para esta tarea, se tomó como guía los ejemplos dados en los laboratorios para realizar cargar los datos. No obstante, hubo una importante modificación: se implementó la función addArtistsNames().

Esta función se invoca cuando se está cargando cada obra, utilizando los IDs de sus autores para buscar en el catálogo de artistas los nombres correspondientes y cargarlos en el catálogo de obras. Esto se realiza mediante una búsqueda binaria, habiendo primero organizado el catálogo de artistas según IDs.

Al tener en el catálogo de obras los nombres de sus respectivos autores, se reduce significativamente la complejidad de los algoritmos de los requerimientos 2, 3, 4 y 5. Además, se realizó inicialmente un ordenamiento del catálogo de artistas según BeginDate y del catálogo de obras según DateAcquired.

Nota: a menos que se indique lo contrario, para los ordenamientos se utilizó el algoritmo Merge Sort.

Requerimiento 1

Preámbulo

Entiéndase: **m** = size(artists). Es importante recordar que el catálogo de artistas está ordenado según fecha de nacimiento desde la carga de datos.

Análisis de complejidad

Para desarrollar este requerimiento se hizo uso de una búsqueda binaria y de la función getInitPosReq1(). La búsqueda binaria tiene como fin encontrar la posición en la que se encuentra el año inicial (no necesariamente la primera posición en la que aparece, dado que puede haber valores repetidos), realizando **log(m)** recorridos.

Posteriormente, dentro de getInitPosReq1() se realiza un ciclo que busca la primera posición del catálogo de artistas en la que aparece el año dado como inicial, realizando por mucho **m/2** recorridos (no realiza **m** recorridos puesto que, como se utilizó una búsqueda binaria, en el peor de los casos ya se partió el tamaño de los datos a la mitad). Finalmente, se hace uso de la función getArtistsRangeReq1(), que se encarga de guardar los datos relevantes de los artistas que están dentro del rango dado, realizando **m** ciclos en el peor caso.

De esta manera, para el requerimiento 1 se realizan **log(m) + (3/2)m** recorridos en el peor caso. Se reduce entonces la complejidad a **O(m)**.

Pruebas de tiempo

Se utilizaron las siguientes entradas para recrear el peor caso:

- Año Inicial = 0
- Año Final = 2022

A continuación, los resultados obtenidos:

		Tiempo [ms]	
Archivo	**# artistas (m)**	**Daniel**	**Carlos**
small	1948	6.250	15.625
5pct	4996	15.625	15.625
10pct	6656	25.000	15.625
20pct	8724	31.250	23.312
30pct	10063	37.500	23.312
50pct	12137	40.625	31.250
80pct	14143	50.000	31.250
large	15223	53.125	40.625

Tabla 1. Pruebas de rendimiento del primer requerimiento

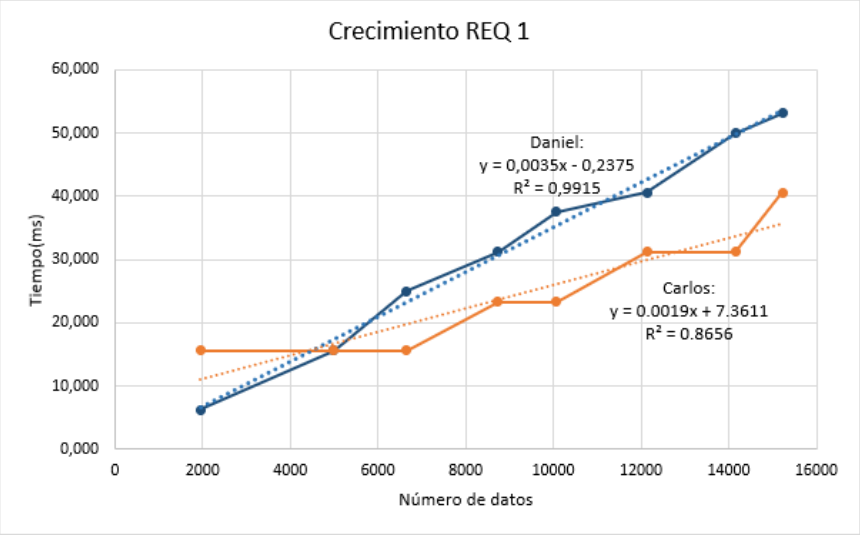

Figura 1. Gráfico del comportamiento del primer requerimiento (regresión lineal)

Se puede observar que, si bien en las pruebas realizadas por Daniel Hernández se obtuvo un comportamiento lineal, no fue así en las pruebas realizadas por Carlos Holguín. Esto se puede atribuir, sin embargo, a la precisión del cronómetro de la librería "time" utilizada para estas mediciones, dado que el mínimo valor que arroja (distinto de cero) son 15.625 milisegundos, aproximando a este los valores que posiblemente habrían mostrado un comportamiento distinto en el gráfico de Carlos. En los requerimientos posteriores no se cuenta con este problema, pues este es el único requerimiento con tiempos de ejecución tan pequeños para los archivos de todos los tamaños.

Nota: Daniel obtuvo una medición de 6.25 milisegundos con el archivo small puesto que se promediaron 5 mediciones de tiempo, de las cuales algunas arrojaron tiempos de 0.0 milisegundos.

Requerimiento 2

Preámbulo

Entiéndase: **n** = size(artworks). Es importante recordar que el catálogo de obras está ordenado por fecha de adquisición desde la carga de datos.

Análisis de complejidad

Para desarrollar este requerimiento se hizo uso de una búsqueda binaria y de la función getArtworksInfoReq2(). La búsqueda binaria tiene como fin encontrar la posición del primer elemento en el rango, realizando **log(n)** recorridos. La búsqueda binaria de este requerimiento tiene una particularidad y es que, puesto que la fecha inicial es tan precisa, muchas veces no se encontrará coincidencia exacta en los datos, por lo que se elige la posición del primer elemento que entra en el rango. Esto no afecta la complejidad del algoritmo (ver código).

La función getArtworksInfoReq2() invoca la búsqueda binaria mencionada anteriormente y, a partir de la posición inicial encontrada en el rango, se recorre el catálogo de obras (**n** recorridos en el peor caso) hasta que se encuentre una obra que ya no pertenece al rango. Durante este ciclo, se va añadiendo la información de cada obra en una lista si fue adquirida en el rango dado como entrada.

En total, se realizan **log(n) + n** ciclos en el peor caso, resumiéndose entonces en una complejidad de **O(n)**.

Nota: Si bien este requerimiento podría haber sido implementado sin necesidad de búsqueda binaria (quitando el log(n) adicional), en la práctica será más rápido el algoritmo con búsqueda binaria en la mayoría de los casos. Las excepciones se presentarán cuando se tengan casos muy cercanos al peor caso, en los que la posición inicial del rango se encuentre entre las primeras posiciones del archivo.

Pruebas de tiempo

Se utilizaron las siguientes entradas para recrear el peor caso:

- Fecha Inicial = 1900-01-01
- Año Final = 2022-12-31

Los resultados obtenidos fueron:

		Tiempo [ms]	
Archivo	# obras (n)	Daniel	Carlos
small	768	6,250	15,625
5pct	7572	71,875	78,125
10pct	15008	137,500	156,250
20pct	29489	271,875	281,250
30pct	43704	421,875	421,875
50pct	71432	700,000	703,125
80pct	111781	1140,625	1187,500
large	138150	1440,625	1500,000

Tabla 2. Pruebas de rendimiento para el segundo requerimiento

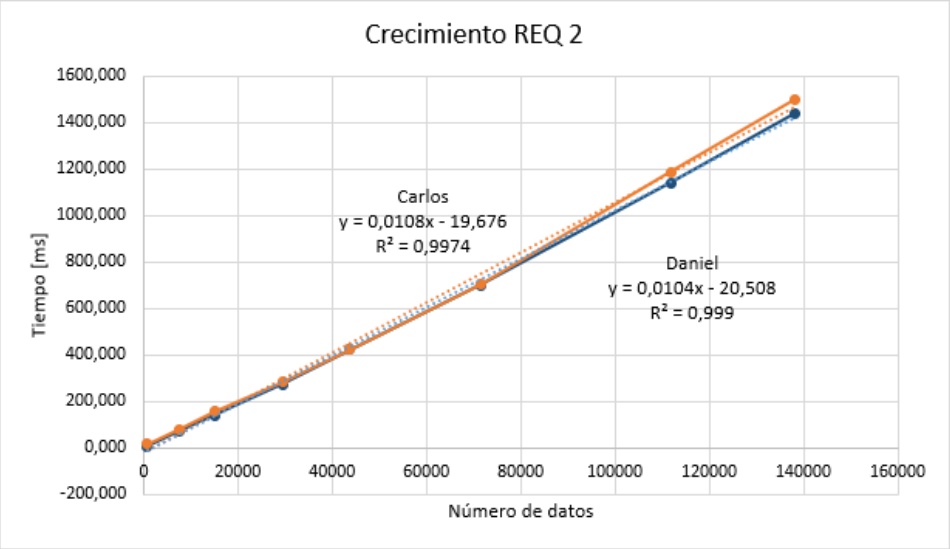

Figura 2. Gráfico del comportamiento del segundo requerimiento

Como se observó en las pruebas de rendimiento, el comportamiento del algoritmo es claramente lineal, confirmando el análisis de complejidad realizado previamente. La distinción entre las pruebas realizadas por cada miembro del grupo tiene que ver con la máquina de la que cada uno disponía, pero para ambos se tuvo un comportamiento acorde con el O(n) anunciado teóricamente.

Requerimiento 3 (implementado por Carlos Holguín)

Para este requerimiento,

Entradas del caso: **Louise Bourgeois**

Preámbulo

Entiéndase: **N** = size(artworks). Es importante recordar que el catálogo de obras está ordenado por fecha de adquisición desde la carga de datos.

Análisis de complejidad

Para desarrollar este requerimiento se hace uso de 4 funciones en *getTechniquesReq3()* se hace un recorrido del catálogo de las obras, este representa **n** de complejidad temporal, ya que, en este ciclo se recorre toda la lista. Posteriormente, se llama la función *operaciones _req3(),* esta se encarga de encontrar las técnicas que aparecen con mayor frecuencia, este representa **NLogN** de complejidad temporal. Posteriormente, se eliminan algunos datos repetidos con la función *eliminar_repetidos()*, esta representa **NLogN** de complejidad temporal. Después se organiza la frecuencia de aparición de las técnicas, se hace uso de Shell sort, lo cual representa una complejidad temporal de $\boldsymbol{N^{\frac{3}{2}}}$. Finalmente, se clasifican los datos de interés por medio de la función *encontrar_obras_con_tec(),* lo cual representa una complejidad temporal de **N**. Además, cabe añadir que estas tres últimas funciones tienden a iterar pocas veces, ya que, la lista con las que se trata tiende a tener muy pocos datos, ya que, estas ya están clasificados.

Obteniendo:

N+ NLogN + NLogN + $\boldsymbol{N^{\frac{3}{2}}}$**+ N**

En notación Big O

O($\boldsymbol{N^{\frac{3}{2}}}$**)**

Pruebas de Tiempo

Archivo	# obras (n)	tiempo [ms]
small	768	0.1
5pct	7572	31.25
10pct	15008	109
20pct	29489	406.25
30pct	43704	828
50pct	71432	2312.5
80pct	111781	5531.25
large	138150	8,625

Tabla 3. Pruebas de rendimiento del tercer requerimiento

Figura 3. Gráfico del comportamiento del tercer requerimiento

Por otro lado, en la Figura 3, se observa un comportamiento que no es lineal pero tampoco corresponde a un comportamiento cuadrático, observando que la complejidad temporal estimada es muy cercana a la real, por otro lado, el algoritmo posee unos tiempos tolerables.

Requerimiento 4 (implementado por Daniel Hernández)

Preámbulo

Entiéndase: **m** = size(artists); **n** = size(artworks); **z** = #nacionalidades; **x** = máximo de autores en una obra; **y** = máximo de artistas pertenecientes a la misma nacionalidad. Si bien es teóricamente posible que **x,y,z** tengan el mismo tamaño que **m**, estos valores serán siempre de un orden significativamente menor que **m** por la naturaleza de los datos utilizados (aunque también aumentan a medida que **m** aumenta). Por esta razón, no tiene mucho sentido tratarlos como si fuesen **m**.

Análisis de complejidad

Para este requerimiento se crearon dos funciones: nationalityListReq4() y getNationalityCountReq4().

La primera función crea dos listas de tipo “ARRAY_LIST” en las cuales se almacenan listas llamadas “country” y los nombres de las nacionalidades a modo de directorio. Cada una de estas listas “country” almacena en la primera posición el nombre de la nacionalidad a tratar, y en las demás posiciones los IDs de sus artistas correspondientes. Para esta función, el método lt.getElement() tiene orden de crecimiento constante al tratarse de un arreglo. Por otro lado, se realiza un ciclo que recorre todo el catálogo de artistas, es decir, realiza **m** ciclos. Dentro de cada uno de estos ciclos se invoca el método lt.isPresent(), el cual, en el peor caso, realiza **z** ciclos. Así, la función realiza **mz** ciclos en el peor caso.

La segunda función invoca a nationalityListReq4(). Luego, recorre el directorio de los nombres de las nacionalidades **(z** ciclos**)** y crea una pila en la que se almacenará la información relevante de cada nacionalidad. Posteriormente, recorre el catálogo de obras (**n** ciclos) y, dentro de este ciclo, revisa cada ID de los autores de la obra (**x** ciclos) y verifica la nacionalidad por medio de la lista creada por

nationalityListReq4() (**y** ciclos). Entonces, en total, el requerimiento 4 realiza **mz + nxyz** ciclos. Puesto que **n** se vuelve considerablemente mayor que **m** a medida que aumenta el tamaño de los datos, se puede resumir la complejidad de este algoritmo como **O(nxyz)**.

Es importante resaltar que, si bien **x,y,z** son mucho menores que **n**, vale la pena incluirlos en la complejidad porque aumentan su tamaño a medida que aumenta el tamaño de los archivos. Esto impide que se pueda hablar de comportamiento lineal en el algoritmo ni se pueda escribir simplemente como O(n).

Pruebas de tiempo

Se obtuvieron los siguientes resultados:

Archivo	# obras (n)	tiempo [s]
small	768	0,67
5pct	7572	12,28
10pct	15008	29,69
20pct	29489	70,18
30pct	43704	114,64
50pct	71432	211,50
80pct	111781	372,66
large	138150	485,84

Tabla 4. Pruebas de rendimiento para el cuarto requerimiento

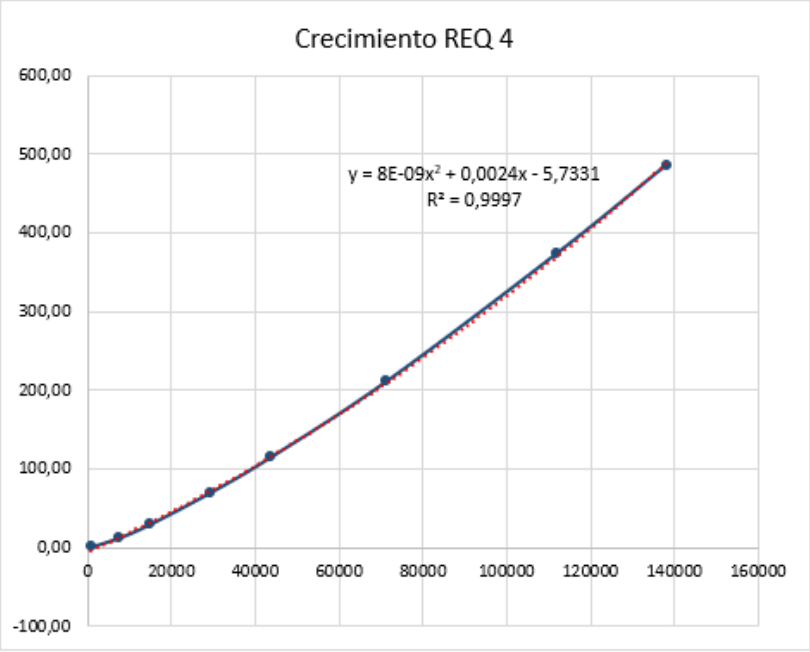

Figura 4.1. Gráfico del comportamiento del cuarto requerimiento (regresión cuadrática)

Es importante detallar de la Figura 4.1 que, si bien un modelo cuadrático se ajusta al gráfico, el coeficiente de la variable cuadrática es supremamente pequeño, dado que, como se dijo anteriormente, el tamaño de **x,y,z** no es comparable con el tamaño de **n.** De hecho, una regresión lineal se adapta al gráfico de manera satisfactoria, aunque no lo suficientemente buena como la regresión cuadrática:

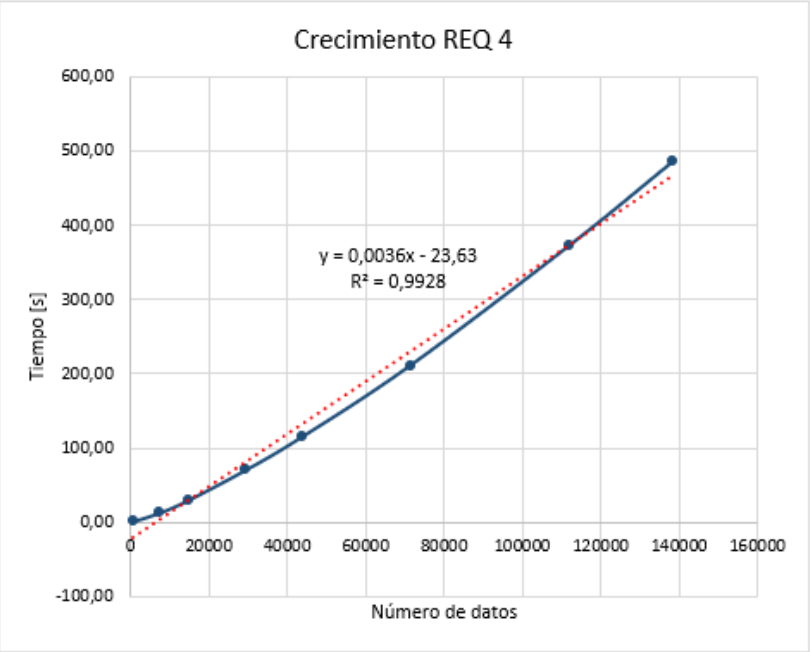

Figura 4.2. Gráfico del comportamiento del cuarto requerimiento (regresión lineal)

Los resultados obtenidos, evidentemente, comprueban lo propuesto en el análisis teórico de la complejidad de los algoritmos.

Requerimiento 5

Preámbulo

Entiéndase: **n** = size(artworks); **x =** número de obras del departamento.

Aunque teóricamente **x** pudiese ser del mismo tamaño de **n**, los datos con los que se está trabajando hacen que **x** sea considerablemente menor que **n** para archivos de cualquier tamaño.

Análisis de complejidad

Para este requerimiento se hace uso de seis funciones: calculateDimensionsReq5(), calculateSingularCostReq5(), getInitPosReq5(), addInfoReq5(), addTOP5Req5(), moveArtworksReq5().

La función calculateDimensionsReq5() calcula las dimensiones físicas (área o volumen) de cada obra dependiendo la información que se brinde (calculará volumen siempre que se tenga información suficiente, y área cuando solo se tengan dos valores de longitud o el diámetro). En esta se tiene una complejidad de **O(1),** ya que, solo hay un ciclo y este solo itera 5 veces, por tanto, se aproxima.

La función calculateSingularCostReq5() se encarga de calcular el costo de la obra dadas sus dimensiones físicas y su peso. Su complejidad es de orden de **O(1),** ya que, solo hay condicionales y operaciones matemáticas que permiten determinar el costo. Otras funciones con orden de crecimiento constante son addInfoReq5(), que no realiza siquiera iteraciones, y addTOP5Req5(), que realiza 5 iteraciones, en cualquier caso.

En la función getInitPosReq5() se consigue la primera posición de la lista en que aparece una obra del departamento dado. Para ello se hace una búsqueda binaria de **log(n)** iteraciones y posteriormente un ciclo de **n/2** de iteraciones en el peor caso, de igual manera como se explicó para la función getInitPosReq1() en el primer requerimiento.

Por último, la función moveArtworksReq5() realiza varias operaciones. Primero, ordena el catálogo de obras según departamento con el fin de agrupar las obras del mismo departamento, con una complejidad de **nlog(n)**. Posteriormente, se invoca la función getInitPosReq5() para encontrar la posición de la primera obra que pertenece al departamento dado. Más adelante, se da inicio a un recorrido del catálogo de obras (**n** recorridos en el peor caso) para extraer la información de las obras pertenecientes al departamento y agregarlo a una lista, y se detiene cuando encuentra una obra de un departamento distinto. Luego, se ordena dos veces la lista que contiene la información extraída según distintos criterios, con el fin de obtener las obras más costosas de transportar y las más antiguas. Estos ordenamientos sumarían entonces **xlog(x)** recorridos cada uno.

De esta manera, el algoritmo realiza **nlog(n) + log(n) + (3/2)n + 2xlog(x)** recorridos en el peor caso. Esto se puede resumir en una complejidad de **O(nlogn)**

Pruebas de tiempo

Se utilizó la siguiente entrada:

- Departamento: Drawings & Prints

Los resultados obtenidos se muestran a continuación:

		Tiempo [ms]	
Archivo	**# obras (n)**	**Daniel**	**Carlos**
small	768	50,000	46,875
5pct	7572	778,125	671,875
10pct	15008	1618,750	1046,250
20pct	29489	3418,750	3093,750
30pct	43704	5331,250	4484,375
50pct	71432	9096,875	7625,000
80pct	111781	14971,875	12546,875
large	138150	18959,375	16078,125

Tabla 5. Pruebas de rendimiento para el quinto requerimiento

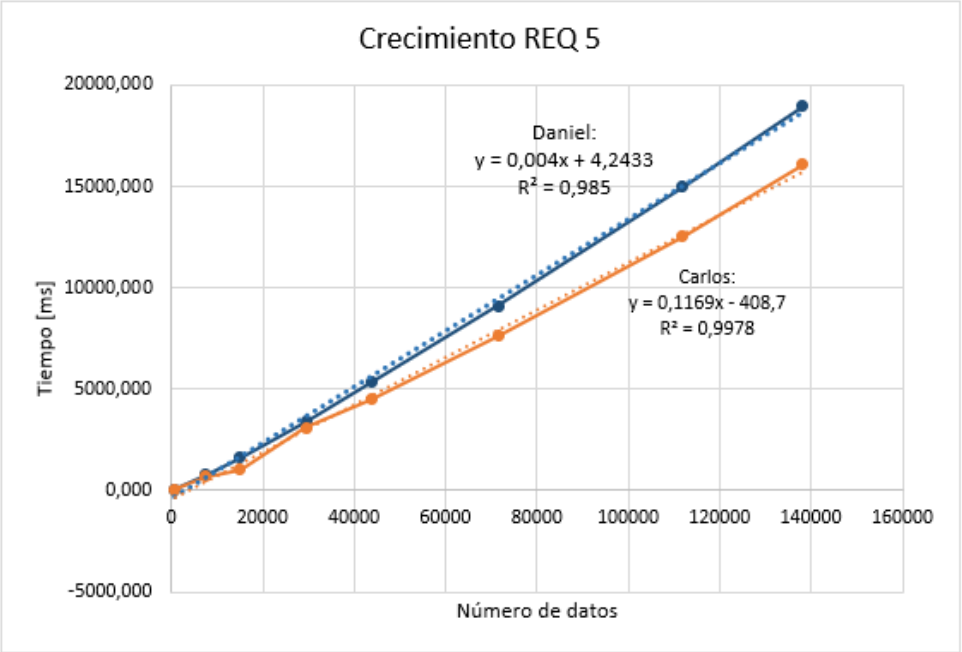

Figura 5. Gráfico del comportamiento del quinto requerimiento

Como era de esperarse, en las pruebas realizadas por ambos estudiantes se observó un orden de crecimiento compatible con el linearítmico propuesto en el análisis teórico, aunque se evaluó mediante regresiones lineales por el tipo de herramientas con las que se cuenta.