OBSERVACIONES DEL RETO 2

#Req 4 Samuel Alejandro Jiménez Ramírez - [redacted]
#Req 3 Marilyn Stephany Joven Fonseca -

Ambientes de pruebas

	Máquina 1	Máquina 2
Procesadores	Intel® Core™ i5 - 10310U CPU @ 1.70 GHz 2.21GHz 64 bits	AMD Ryzen 5 4500U Radeon Graphics 2.38GHz
Memoria RAM (GB)	16 GB (15,6 utilizable)	16 GB
Sistema Operativo	Windows 10 Pro	Windows 10 Home

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

Resultados de las pruebas

Para tomar el tiempo promedio por ejecución de cada requerimiento acá enlistado, usamos el archivo con terminación **small.** Tomamos 3 muestras de cada requerimiento y sacamos la media aritmética que es la que está escrita en la tabla.

El tipo de guardado de datos que encontramos más optimo es **"LINEAR_PROBING "**, esto para el manejo de datos que trabajamos durante cada implementación por requerimiento. Para cada valor de cualquier llave usábamos una lista que contenía los datos. Esta fue la manera más óptima que encontramos de utilizar los dos tipos de estructuras para el trabajo.

```
def newCatalog():
    catalog = {'artists': None,
               'artworks': None,
               }
    catalog["artists"] = mp.newMap(numelements = 3907, maptype="PROBING", loadfactor= 0.5 ) #ID-InfoArtists
    catalog["artworks"] = mp.newMap(numelements = 1543, maptype="PROBING", loadfactor= 0.5 ) #ID-InfoArtworks
    catalog["Medios"] = mp.newMap(numelements = 769, maptype="PROBING", loadfactor= 0.5 ) #Medio-TitleObras
    catalog["Obras"] = mp.newMap(numelements = 1543, maptype="PROBING", loadfactor= 0.5 ) #TitleObra-Fecha
    catalog["NacimientoArtistas"] = mp.newMap(numelements=1667, maptype="PROBING", loadfactor= 0.5) #BeginDate-InfoArtistas
    catalog["DateAcquired"] = mp.newMap(numelements= 191, maptype="PROBING", loadfactor= 0.5) #DateAquired-InfoArtistas
    catalog["Medartist"]= mp.newMap(numelements=3907, maptype="PROBING", loadfactor= 0.5) #ID-Medios
    catalog["ids"] = mp.newMap(numelements=3907, maptype="PROBING", loadfactor= 0.5) #Name-ID
    catalog["Nat"] = mp.newMap(numelements=3907, maptype="PROBING", loadfactor= 0.5) #ID-Nat
    catalog["ID"] = lt.newList(datastructure="ARRAY_LIST") #Todos los ids de artworks
    catalog["ArtNat"] = mp.newMap(numelements=401, maptype="PROBING", loadfactor= 0.5) #Nat-IDs
    catalog["Department"] = mp.newMap(numelements=23, maptype= "PROBING", loadfactor= 0.5) #Catalogo con los medios de las artworks Dept-Obras
    catalog["ConstituentName"] = mp.newMap(numelements=3907, maptype="PROBING", loadfactor= 0.5) #Llave: Constituent ID - Valor: Nombre del artista
    return catalog
```

Para cada mapa que se creó, se tuvo en cuenta la cantidad máxima de datos que pueda contener el mapa, se multiplicó por 2 y se aproximó al numero primo más cercano por encima. El load factor se definió como 0.5 puesto que con base en lo dicho anteriormente no es posible hacer rehash, lo que demoraría el programa más de lo necesario. Cabe aclarar que estos mapas se crean durante la carga de datos al inicio del programa.

Para las muestras no se va a medir la memoria utilizada puesto que esta es constante debido a la carga de datos que se realiza al comienzo del programa, y nunca se elevó a valores preocupantes para el buen funcionamiento de la máquina.

Para el uso de procesador se tiene la misma situación en la que no se llego a dar un valor preocupante, ni a temperaturas muy altas.

Cabe destacar que el tiempo que se demora guardando datos del archivo csv con este tipo de estructura de datos es de **0.265625 s** en la **Máquina 1**, y de **0.21875 s** para la **Máquina 2**.

	Tiempo promedio Máquina 1 (s)	Tiempo promedio Máquina 2 (s)
Requerimiento 1	0.0s	0.015625s
Requerimiento 2	0.046875 s	0.109375s
Requerimiento 3	0.03125 s	0.109375s
Requerimiento 4	0.03125 s	0.09375s
Requerimiento 5	0.015625 s	0.0625s
Requerimiento 6	0.015625 s	0.03125s

Análisis complejidad temporal de cada requerimiento

Requerimiento 1:

La complejidad temporal de este algoritmo sería de O(n) inicialmente, donde n es el número de años que hay entre el año mayor y el año menor, dentro de esta iteración se hacen mp.get y lt.addlast pero como esto agregaría constantes no las tomo en cuenta para dar la complejidad, aparte ambas funciones tienen como complejidad temporal O(1) en ambos casos. También hacemos un ordenamiento con mergesort, pero este ordenamiento es sobre la lista que creamos al hacer el recorrido y su complejidad es **O(n Log(n)),** esta sería una complejidad más acertada para el algoritmo ya que es la mayor en las complejidades que arroja cada paso. A comparación del reto 1, este algoritmo es más rápido puesto que omite un O(n) si se toma desde la complejidad O(n*Log(n)).

Requerimiento 2:

La complejidad temporal de este algoritmo sería de **O(n Log(n))**, donde n es el número de obras que se encuentran en el rango a evaluar dado por el usuario ya que se realiza un merge sort a estos datos y esto arroja la mayor complejidad en todos los pasos que realiza el algoritmo. En este caso se tiene la misma situación que en el requerimiento 1, donde se omite un O(n), a comparación del reto 1.

Requerimiento 3:

Para este requerimiento se presenta una complejidad de **O(n*Log(n))** (donde n es la cantidad de medios utilizados por el artista), por lo que se utilizaron 2 de los mapas creados en la carga de datos, se creó uno que uniera la información de los dos anteriores, y en este procedimiento se realizó un ordenamiento a una lista de llaves que fue de referencia para retornar los valores ordenados. Para las funciones de búsqueda de los datos están dadas por O(1), ya que están guardados sobre un mapa que es creado al cargar los datos, pero no tomamos ese valor debido a que solo ocurre una vez al principio del programa. A comparación del reto 1 no se realizan 3 recorridos a los diferentes archivos, pues la complejidad es mucho menor por el tamaño de la muestra de n.

Requerimiento 4:

En este requerimiento la complejidad sería de **O(n)** (donde n es la cantidad de obras) pues se realiza principalmente un recorrido a una lista que contiene todos los ids(así estén repetidos) que se encuentran en el archivo de obras, para luego ver cuantas veces se repiten las nacionalidades, contarlas, ordenarlas y retornar las veces que aparecen las nacionalidades. A comparación del reto 1 es significativamente menor, ya que en este se tenía un recorrido por las obras y por cada ConstituentID del archivo de obras se realizaba un recorrido al archivo de artists.

Requerimiento 5:

La complejidad temporal de este algoritmo sería de **O(n)**, donde n es el número de obras en el departamento a evaluar. Se hacen ordenamientos y búsquedas en el mapa, pero estos no arrojan la mayor complejidad. A comparación del reto 1 se reduce el tamaño de n puesto que este ya no es la cantidad de obras totales, sino la cantidad de obras que pertenecen al departamento a evaluar.

Requerimiento 6:

En este requerimiento se utiliza el mismo planteamiento que en los requerimientos 1 y 3, pues se toma un rango de fechas como en el requerimiento 1 y luego se ordenan por total de obras por artista. Por esto, la complejidad sería de **O(n*Log(n))** donde n es la cantidad de artistas en el rango, el resto de los procesos es despreciable para la toma de la complejidad.

Conclusiones

- Pudimos concluir que el mejor factor de carga es el de 0.5 debido a que es el valor intermedio entre consumo de espacio y de tiempo, pues un factor de carga muy alto puede llegar a ocupar menos espacio, pero tardaría más, y un factor muy bajo puede ser rápido, pero puede llegar a ocupar mucho espacio.
- Se puede comprobar que la carga de datos es más eficiente y ayuda en el tiempo de ejecución de cada requerimiento si se implementa al principio del programa, durante el recorrido que se hace a la lista arrojada por csv.DictReader.
- El uso de nuevas estructuras de datos ayudó a reducir la complejidad temporal de cada requerimiento a comparación del reto 1. Esto se puede evidenciar tanto en la complejidad como en las pruebas de rendimiento.