Estructuras de Datos y Algoritmos: Reto 01

Juan Sebastián Ortega (Estudiante 1- js.ortergar1@uniandes.edu.co - 202021703) y Yesid Camilo Almanza (Estudiante 2 - y.almanza@uniandes.edu.co - 201921773)

Departamento de Ingeniería de Sistemas y Computación

UNIVERSIDAD DE LOS ANDES

8 de marzo de 2021

I. Introducción

CON base en la información presente en la base de datos *Trending YouTube Video Statistics*, específicamente en los archivos *category-id.csv* y *videos-large.csv*, se planteo realizar una aplicación de consola que permitiera cargar los datos contenidos en los archivos *.csv* y realizar un análisis de estos bajo ciertos requerimientos específicos. Con este objetivo, se implementó la librería *DISClib* para poder: generar listas, filtrar listas y ordenar listas.

Requerimientos a cumplir:

1. Encontrar los n videos con más reproducciones (views) que son tendencia para un país y una categoría especifica.
2. Conocer el video con más días como tendencia en un país.
3. Conocer el video con más días como tendencia según una categoría.
4. Conocer los n videos con más Likes (likes) en un país y con una etiqueta (tag) especifica.

En el momento de cumplir con los requerimientos se tuvo en mente la complejidad temporal que iba a adquirir el algoritmo en cada uno de estos, de forma que se tomaron las siguientes decisiones para optimizar la eficiencia:

- Todos los ordenamientos se realizarán mediante la aplicación de **MERGE SORTING** puesto que es el algoritmo de ordenamiento más efectivo dentro de la *DISClib* de acuerdo a pruebas de eficiencia realizadas previamente.

- Todas las listas serán instanciadas como **ARRAY_LIST** debido a que se prioriza la utilización de las funciones $lt.addLast()$ y $lt.getElement()$ las cuales cuentan con una complejidad de $O(K)$ en este tipo de estructura.

II. Funciones Principales

El algoritmo de la aplicación fue diseñado de tal forma que los diferentes requerimientos comparten las mismas funciones (en su mayoría). Por lo tanto, antes de realizar el análisis general de la complejidad temporal de cada requerimiento es necesario analizar cada función principal individualmente. Las funciones que se mostrarán a continuación se encuentran ubicadas en el $model.py$ y son las encargadas de filtrar y organizar las listas utilizadas por los requerimientos.

II-A. Funciones de filtro

- filterCatalog: **O(n)**

```
def filterCatalog(catalog, column_1, value_1, column_2=None, value_2=None):
    """Filtra el catalogo dejando solo los videos con el valor especificado
    para maximo 2 columnas determinadas."""


    filtered_catalog = lt.newList("ARRAY_LIST")
    filtered_catalog["videos"] = lt.newList("ARRAY_LIST")

    for video in lt.iterator(catalog['videos']):

        if column_2 is not None:

            if video[column_1] == " "+value_1 and video[column_2] == value_2:

                lt.addLast(filtered_catalog["videos"], video)
        else:
            if video[column_1] == value_1:
                lt.addLast(filtered_catalog["videos"], video)

    return filtered_catalog
```

En esta función es posible observar un único *forLoop* que itera a través de todos los videos presentes en el catálogo cargado una única vez, teniendo esta sección una complejidad de **O(n)**. Dentro de este se invoca la función *lt.addLast* se cumpla o no la condición del condicional *if*. Ya que se esta utilizando una *Array_List*, la complejidad de estas funciones internas es de *O(K)*, por lo que se desprecia.

- newUniqueCatalog: **O(2n)**

```
def newUniqueCatalog(catalog):
    unique_dict = {"videos": None}
    unique_dict["videos"] = {}
    unique_catalog = lt.newList("ARRAY_LIST")
    pos = 0
    for video in lt.iterator(catalog):
        pos += 1
        try:
            video_info = unique_dict["videos"][video["title"]]
            new_day = lt.getElement(video_info, 1) + 1
            lt.changeInfo(video_info, 1, new_day)
        except:
            unique_dict["videos"][video["title"]] = lt.newList("ARRAY_LIST")
            lt.addLast(unique_dict["videos"][video["title"]], 1)
            lt.addLast(unique_dict["videos"][video["title"]], pos)
            lt.addLast(unique_dict["videos"][video["title"]], video)

    for i in unique_dict["videos"]:
        lt.addLast(unique_catalog, unique_dict["videos"][i]["elements"])


    return unique_catalog
```

En esta función se itera a través de todos los videos a través de un único *forLoop* lo que establece una

complejidad inicial de *O(n)*. Ya en el interior se utilizan las funciones *lt.addLast*, *lt.getElement* y *lt.changeInfo*. No obstante, ya que estas están siendo aplicadas sobre una *ARRAY_LIST* en una posición ya conocida, la complejidad en cada una es de *O(K)* por lo que no es tenida en cuenta. También existe otro *forLoop* que itera a través de un diccionario que contiene una cantidad reducida de videos denominada m, dando otra complejidad de O(m). Sin embargo, se tomará en cuenta el peor de los casos donde todos los videos sean únicos dando una complejidad total de O(2n).

- filterTag: **O(n)**

```
def filterTag(catalog, tag):
    filter_tags=lt.newList("ARRAY_LIST")
    filter_tags["videos"]=lt.newList("ARRAY_LIST")

    for video in lt.iterator(catalog['videos']):

        video_tags=video["tags"]

        if tag in video_tags:
            lt.addLast(filter_tags["videos"], video)


    return filter_tags
```

En esta función se utiliza un *forLoop* para iterar a través de todos los videos, lo que le da al algoritmo una complejidad de *O(n)* que depende de la cantidad de videos cargados. Adicionalmente se analiza si un substring (tag) está dentro de un string mayor para lo que se utiliza la función de Python *.split()* la cual dependerá de la cantidad de caracteres (c) en el string, dando una complejidad adicional de *O(c)*. Para efectos prácticos, se tendrá en cuenta el límite de 400 caracteres que tiene la casilla de caracteres en YouTube (siendo este una constante en el peor de los casos), por lo que podrá ser descartada la complejidad O(c).

II-B. Funciones de ordenamiento

- sortVideos: $\mathbf{O(n \cdot log_2(n))}$

```
def sortVideos(catalog, size, cmpFunction):

    if cmpFunction == "sortByViews":
        sub_list = lt.subList(catalog['videos'], 1, size)
        sub_list = sub_list.copy()
        start_time = time.process_time()
        sorted_list = mergesort.sort(sub_list, cmpVideosByViews)
        stop_time = time.process_time()
        elapsed_time_mseg = (stop_time - start_time)*1000
        return elapsed_time_mseg, sorted_list

    elif cmpFunction == "sortByDays":
        sub_list = lt.subList(catalog, 1, size)
        sub_list = sub_list.copy()
        start_time = time.process_time()
        sorted_list = mergesort.sort(sub_list, cmpVideosByDays)
        stop_time = time.process_time()
        elapsed_time_mseg = (stop_time - start_time)*1000
        return elapsed_time_mseg, sorted_list

    elif cmpFunction == "sortByLikes":

        sub_list = lt.subList(catalog['videos'], 1, size)
        sub_list = sub_list.copy()
        start_time = time.process_time()
        sorted_list = mergesort.sort(sub_list, cmpVideosByLikes)
        stop_time = time.process_time()
        elapsed_time_mseg = (stop_time - start_time)*1000
        return elapsed_time_mseg, sorted_list
```

Esta función se encuentra fragmentada en 3 condicionales, no obstante, todos tienen la misma complejidad ya que el único cambio es el valor con el que se realizará la comparación al momento de organizar los datos. Como es posible observar se utiliza **MERGE_SORT** para organizar una lista con n elementos (la cantidad de videos cargados) lo que tiene una complejidad de $O(n \cdot log_2(n))$

III. Requerimiento 01

```
def requerimiento_1(catalog):

    filter_category = filterCategory(catalog)
    filter_country = filterCountry(catalog)

    filtered_catalog = controller.filterCatalog(catalog = catalog,
    column_1 = "category_name", column_2 = "country", value_1 =
    filter_category, value_2 = filter_country)

    n_sample = askSampleList(filtered_catalog)

    top_views = sortVideos(filtered_catalog, lt.size(filtered_catalog["videos"]), "sortByViews")

    printResultsReq1(top_views[1], n_sample)
```

En el Requerimiento 1 se empieza por aplicar dos funciones iterativas *(filterCategory y filterCountry)* cuya única función es la de obtener una entrada válida por parte del usuario por lo que su complejidad se reduce a una constante y puede ser despreciada.

La siguiente función hace referencia a *filterCatalog* la cual tiene una complejidad de **O(n)**.

La función invocada para definir la variable *n_sample* solo busca obtener una entrada válida por parte del usuario por lo que su complejidad es constante y por lo tanto despreciable.

Por último, se invoca la función sortVideos la cual tiene una complejidad de $\mathbf{O(n \cdot log_2(n))}$. Es importante resaltar que debido a que se realizó previamente un filtrado por categoría, la cantidad de datos *n* es mucho menor en el momento de ordenar los datos. Por cuestiones prácticas, se mantendrá está *n* en el peor de los casos donde todos los videos pertenezcan a la misma categoría.

Complejidad temporal final: $\mathbf{O(n \cdot log_2(n)) + O(n)}$

IV. Requerimiento 02 - Estudiante 1

```
filter_country = filterCountry(catalog)
    filtered_catalog = controller.filterCatalog(catalog = catalog,
    column_1 = "country", value_1 = filter_country)

    max_videos = lt.newList()

    unique_catalog = controller.initUniqueCatalog(filtered_catalog["videos"])

    top_days = sortVideos(unique_catalog, lt.size(unique_catalog), "sortByDays")


    first_video = lt.firstElement(top_days[1])

    max_days = first_video[0]

    pos = 1

    while lt.getElement(top_days[1], pos)[0] == max_days:

        lt.addLast(max_videos, lt.getElement(top_days[1], pos)[2])
        pos += 1

    printResultsReq2(max_videos, str(max_days))
```

Como se observó en el requerimiento anterior, la función filterCountry cuenta con una complejidad constante por lo que no será tomada en cuenta.

Se aplica la función filterCatalog (complejidad O(n)) que reduce el catálogo que se utilizará a continuación a tan solo los videos que contengan el país específicado en filterCountry.

En este caso se aplica la función newUniqueCatalog (complejidad *O(2n)* y se obtiene un catalogo reducido. Posteriormente, este catálogo reducido es organizado mediante la función sortVideos (complejidad de $O(n \cdot log_2(n))$).

En las lineas finales se accede a los primeros valores del catálogo ordenado (dependiendo del máximo de días), ya que se accede a una posición conocida, la complejidad es constante y por lo tanto despreciable.

Complejidad temporal final:
$\mathbf{O(n \cdot log_2(n)) + O(n) + O(2n)}$

V. Requerimiento 03 - Estudiante 2

```
filter_category ="- " + filterCategory(catalog)
filtered_catalog = controller.filterCatalog(catalog = catalog,
column_1 = "category_name", value_1 = filter_category)

max_videos = lt.newList()
unique_catalog = controller.initUniqueCatalog(filtered_catalog["videos"])
top_days = sortVideos(unique_catalog, lt.size(unique_catalog), "sortByDays")


first_video = lt.firstElement(top_days[1])
max_days = first_video[0]
pos = 1

while lt.getElement(top_days[1], pos)[0] == max_days:

    lt.addLast(max_videos, lt.getElement(top_days[1], pos)[2])
    pos += 1

printResultsReq2(max_videos, str(max_days))
```

Es posible observar que el requerimiento 3 es idéntico al requerimiento 2, la única diferencia es que este en vez de filtrar con base en un país, filtra basándose en el nombre de única categoría, pero su complejidad y funcionamiento es el mismo.

Complejidad temporal final:
$\mathbf{O(n \cdot log_2(n)) + O(n) + O(2n)}$

VI. Requerimiento 04

```
filter_country = filterCountry(catalog)
    filtered_catalog = controller.filterCatalog(catalog = catalog,
    column_1 = "country", value_1 = filter_country)

    filter_tag=filterTag(filtered_catalog)
    top_likes =sortVideos(filter_tag,lt.size(filter_tag["videos"]), "sortByLikes")
    unique_catalog = controller.initUniqueCatalog(top_likes[1])
    n_sample = askSampleList(filter_tag)

    printResultsReq4(unique_catalog, n_sample)
```

En el requerimiento 4 se utiliza la ya mencionada función filterCountry con complejidad constante y la función filterCatalog (complejidad O(n)). La gran diferencia es que existe una función adicional de filtrado, filterTag, la cual cuenta con una complejidad de O(n). Posterior a esta se realiza nuevamente un ordenamiento mediante sortVideos (complejidad $O(n \cdot log_2(n))$), y a la lista ya ordenada se le vuelve a filtrar a través de la función newUniqueCatalog (complejidad O(2n)).

Complejidad temporal final:
$\mathbf{O(n \cdot log_2(n)) + O(n) + O(2n) + O(n)}$

VII. Conclusión

En conclusión, es posible observar en todos los requerimientos que los polinomios que modelan su complejidad temporal siempre tendrán como término mayor el referente a la complejidad de la función *sortVideos*. Debido a esto la complejidad aproximada de cada requerimiento se reduce a:
$\mathbf{O(n \cdot log_2(n))}$

El algoritmo creado fue implementado con éxito teniendo en cuenta la eficiencia temporal ya que ninguna función de filtrado adquirió una complejidad aproximada mayor a $O(n)$, demostrando que es el ordenamiento el proceso más tardado en este tipo de aplicaciones que operan con grandes volúmenes de datos. En adición, otra ventaja que presenta el algoritmo presentado es que se prioriza primero la filtración de los datos y luego su ordenamiento. De esta forma, la cantidad de datos a ordenar (que determina finalmente el tiempo de ejecución) es menor tras cada filtro.

Referencias

[1] R. Sedgewick and K. Wayne, Algorithms, 4th Edition. Addison-Wesley, 2011.