Universidad de los Andes
Ingeniería de Sistemas y Computación
Estructuras de Datos y Algoritmos

Reto 1: YouTube Social Analysis

Documento de análisis de los requerimientos propuestos en el Reto 1 del curso

María Paula Alméciga Moreno
m.almeciga@uniandes.edu.co
202023369

Andrés Felipe Vargas Cuadros
af.vargasc1@uniandes.edu.co
202013817

Bogotá, Colombia
Marzo 2021

Índice general

Capítulo 1

Introducción

En el primer reto del curso se presenta una serie de datos sobre los videos que históricamente han sido tendencia en la plataforma YouTube, los cuales pueden ser útiles para estudios como análisis sociales de comportamientos de la población en ámbitos generales y debido a eventos determinados en el año.

Teniendo en cuenta esto, primero es necesario poseer una herramienta que permita consultar y filtrar fácilmente los datos, para facilitar su estudio.

1.1. Objetivos

Para este primer reto del curso se tienen los siguientes objetivos principales:

- Practicar los conceptos aprendidos en clase acerca de las estructuras de datos lineales: Lista, Pila y Cola.
- Practicar los conceptos aprendidos en clase acerca de los algoritmos de ordenamiento y búsquedas eficientes de información.
- Utilizar de manera adecuada el manejo de versiones utilizando Git.

Capítulo 2

Requerimientos

Para la implementación del reto se han propuesto los siguientes cuatro requerimientos:

1. Encontrar buenos videos por categoría y país (Grupal)
2. Encontrar video tendencia por país (Individual)
3. Encontrar video tendencia por categoría (Individual)
4. Buscar los videos con más likes (Grupal)

A continuación se presenta información más detallada sobre cada uno de los requerimientos y un análisis de su complejidad temporal.

2.1. Requerimiento 1

Para el primer requerimiento se quieren conocer los **n** videos con más **Likes** en un **país** y con una **etiqueta** (tag).

Recibe del usuario la categoría, el país, y el número de videos a listar (n). Como respuesta imprime el nombre del video, la fecha en que fue tendencia, nombre del canal, fecha de publicación, reproducciones, likes, y dislikes.

2.1.1. Complejidad temporal

Al analizar el código implementado para completar el requerimiento 1, se concluye que la complejidad temporal (en notación big O) del requerimiento corresponde a:

$$O(n^2)$$

Ya que el crecimiento temporal más significativo en el código es $O(n^2)$, que corresponde al crecimiento temporal del algoritmo de ordenamiento quick sort, implementado para ordenar los datos al final.

Figura 2.1: Código con la complejidad temporal más significativa en el Req. 1.

```python
def firstReq(catalog, data_size, country, category):
    """
    Completa el requerimiento 1
    """
    filtered = catalog.copy()
    i = 1
    t = lt.size(filtered["videos"])
    while i <= t:
        elem = lt.getElement(filtered["videos"], i)
        if (elem["country"].lower()) != (country.lower()) or elem["category_id"] != category:
            lt.deleteElement(filtered["videos"], i)
            t -= 1
            i -= 1
        i += 1
    sorted_list = quick.sort(filtered["videos"], cmpVideosByViews)
    try:
        data_sublist = lt.subList(sorted_list, 1, data_size)
        data_sublist = data_sublist.copy()
        return data_sublist
    except:
        return sorted_list
```

2.2. Requerimiento 2

Para el segundo requerimiento se quiere conocer el video con **más días como tendencia** en un **país**.

Recibe del usuario el país. Como respuesta imprime el nombre del video, nombre del canal, y el número de días como tendencia.

El requerimiento, siendo individual, fue implementado por **Andrés Felipe Vargas Cuadros**.

2.2.1. Complejidad temporal

Al analizar el código implementado para completar el requerimiento 2, se concluye que la complejidad temporal (en notación big O) del requerimiento corresponde a:

$$O(n)$$

Ya que el crecimiento temporal más significativo en el código es $O(n)$, que corresponde a un ciclo con una variable que, en el peor caso, aumenta de forma constante, y va iterando normalmente sobre todos los elementos de la lista dada.

Figura 2.2: Código con la complejidad temporal más significativa en el Req. 2.

```
def secondReq(catalog, country):
    """
    Completa el requerimiento 2
    """
    dicc = {}
    filtered = catalog.copy()
    i = 1
    t = lt.size(filtered["videos"])
    while i <= t:
        elem = lt.getElement(filtered["videos"], i)
        if (elem["country"].lower()) != (country.lower()):
            lt.deleteElement(filtered["videos"], i)
            t -= 1
            i -= 1
        i += 1
    i = 1
    t = lt.size(filtered["videos"])
    x = 0
    while i <= t:
        elem = lt.getElement(filtered["videos"], i)
        titulo = (elem["title"] + "#,@,#" + elem["channel_title"])
        if titulo not in dicc:
            dicc[titulo] = 1
            x += 1
        else:
            dicc[titulo] += 1
        i += 1
    dicc_sort = sorted(dicc.items(), key=operator.itemgetter(1), reverse=True)
    mayor = dicc_sort[0]
    primerosdatos = mayor[0].split("#,@,#")
    resultado = [primerosdatos[0], primerosdatos[1], mayor[1]]
    return resultado
```

2.3. Requerimiento 3

Para el tercer requerimiento se quiere conocer el video con **más días como tendencia** según una **categoría**.

Recibe del usuario la categoría. Como respuesta imprime el nombre del video, nombre del canal, el identificador de categoría, y el número de días como tendencia.

El requerimiento, siendo individual, fue implementado por **María Paula Alméciga Moreno**.

2.3.1. Complejidad temporal

Al analizar el código implementado para completar el requerimiento 3, se concluye que la complejidad temporal (en notación big O) del requerimiento corresponde a:

$$O(n)$$

Ya que el crecimiento temporal más significativo en el código es $O(n)$, que corresponde a un ciclo con una variable que, en el peor caso, aumenta de forma constante, y va iterando normalmente sobre todos los elementos de la lista dada.

Figura 2.3: Código con la complejidad temporal más significativa en el Req. 3.

```
def thirdReq(catalog, category):
    """
    Completa el requerimiento 3
    """
    dicc = {}
    filtered = catalog.copy()
    i = 1
    t = lt.size(filtered["videos"])
    while i <= t:
        elem = lt.getElement(filtered["videos"], i)
        if elem["category_id"] != category:
            lt.deleteElement(filtered["videos"], i)
            t -= 1
            i -= 1
        i += 1
    i = 1
    t = lt.size(filtered["videos"])
    x = 0
    while i <= t:
        elem = lt.getElement(filtered["videos"], i)
        titulo = (elem["title"] + "#,@,#" + elem["channel_title"])
        if titulo not in dicc:
            dicc[titulo] = 1
            x += 1
        else:
            dicc[titulo] += 1
        i += 1
    dicc_sort = sorted(dicc.items(), key=operator.itemgetter(1), reverse=True)
    mayor = dicc_sort[0]
    primerosdatos = mayor[0].split("#,@,#")
    resultado = [primerosdatos[0], primerosdatos[1], mayor[1], category]
    return resultado
```

2.4. Requerimiento 4

Para el cuarto requerimiento se quieren conocer los **n** videos con más **likes** en un **país** y con una **etiqueta** (tag) específica.

Recibe del usuario el país, el número de videos para listar (n), y la etiqueta (tag) del video. Como respuesta imprime el nombre del video, nombre del canal, la fecha de publicación, las reproducciones, los likes, dislikes, y las categorías (tags).

2.4.1. Complejidad temporal

Al analizar el código implementado para completar el requerimiento 4, se concluye que la complejidad temporal (en notación big O) del requerimiento corresponde a:

$$O(n^2)$$

Ya que el crecimiento temporal más significativo en el código es $O(n^2)$, que corresponde al crecimiento temporal del algoritmo de ordenamiento quick sort, implementado para ordenar los datos al final.

Figura 2.4: Código con la complejidad temporal más significativa en el Req. 4.

```
def fourthReq(catalog, data_size, country, tag):
    """
    Completa el requerimiento 4
    """
    filtered = catalog.copy()
    i = 1
    t = lt.size(filtered["videos"])
    while i <= t:
        elem = lt.getElement(filtered["videos"], i)
        if (elem["country"].lower()) != (country.lower()) or tag not in elem["tags"]:
            lt.deleteElement(filtered["videos"], i)
            t -= 1
            i -= 1
        i += 1
    sorted_list = quick.sort(filtered["videos"], cmpVideosByLikes)
    try:
        data_sublist = lt.subList(sorted_list, 1, data_size)
        data_sublist = data_sublist.copy()
        return data_sublist
    except:
        return sorted_list
```