# Análisis de Resultados Reto-3

Integrantes:

- E1: Daniela Alvarez Rodriguez-202020209-d.alvarezr@uniandes.edu.co

## ***Requerimiento 1: Caracterizar las reproducciones***

```
def caracterizarrep(cat, carac, minimo, maximo):
(1)         m = cat["features"]
(1)         a = mp.get(m, carac)
(1)         m_carac = me.getValue(a)
(lgN + M)   l_reps = om.values(m_carac, minimo, maximo)

(1)         t_artists = om.newMap(omaptype='RBT',
                                  comparefunction=compareValue)
(1)         num_reps = 0

(M)         for lists in lt.iterator(l_reps):
(1)           size = lt.size(lists)
(1)           num_reps+=size
(P)           for reps in lt.iterator(lists):
(1)              artist = reps["artist_id"]
(lgQ)            if om.contains(t_artists, artist):
(lgQ)                c = om.get(t_artists, artist)
(1)                  l_artist = me.getValue(c)
(1)              else:
(1)                  l_artist = lt.newList(datastructure="SINGLE_LINKED")
(1)              lt.addLast(l_artist, reps)
(lgQ)            om.put(t_artists, artist, l_artist)

(1)         artists = om.size(t_artists)
            return (num_reps, artists, t_artists)
```

### *Análisis de Complejidad (Notación O)*

Cálculo de complejidad:

- N= Tamaño árbol de la característica consultada
- M = Tamaño lista de listas de reproducciones que están dentro de los valores consultados
- P = Tamaño lista de reproducciones que están dentro de los valores consultados
- Q = Tamaño árbol donde se guardan los artistas únicos de las reproducciones

$$O(n) = 3 + (lgN + M) + 2 + \mathrm{M}\big(2 + \mathrm{P}(\,1 + \mathrm{lgQ} + \mathrm{lgQ} + 1 + 1 + \mathrm{lgQ})\big) + 1$$

$$O(n) = 6 + \mathrm{lgN} + \mathrm{M} + \mathrm{M}\big(2 + \mathrm{P}(\,3 + 3\mathrm{lgQ})\big)$$

$$O(n) = 6 + \mathrm{lgN} + \mathrm{M} + \mathrm{M}(2 + 3\mathrm{P} + 3\mathrm{PlgQ})$$

$$O(n) = 6 + \lg N + M + 2M + 3PM + 3PM\lg Q$$

$$O(n) = 6 + \lg N + 3M + 3PM + 6PM\lg Q$$

$$O(n) = 6PM\lg Q$$

$$O(n) = O(PM\lg Q)$$

Discusión

La complejidad de este algoritmo utilizando la notación BIG O es O(PMlgQ) esto quiere decir que su orden de crecimiento se asemeja a uno linearítmico por lo que el tiempo de procesamiento va a aumentar en menor proporción a medida que aumenta la cantidad de datos. Cabe resaltar que en este análisis se tomó el producto PM como una sola variable de tamaño similar a M, lo anterior debido a que se asume que P es un valor cercano a 1. Esto debido a que, por lo que los valores de features son decimales (float), es poco factible que muchas reproducciones coincidan en una misma llave. Por otro lado, es evidente que Q es mucho menor a M, por lo que, siendo mas exactos en la obtención de la complejidad, podemos designar la siguiente acotación:

$$O(Q\log Q) \leq O(PM\lg Q) \leq O(M\log M)$$

***Análisis de tiempo de ejecución y consumo de memoria***

| Archivos cargados [%] | Consumo de Datos [kB] | Tiempo de Ejecución [ms] |
|---|---|---|
| small | ***0.172*** | ***1.759*** |
| 5 | ***0.172*** | ***1.705*** |

***Requerimiento 2 y 3: Música para festejar y estudiar***

```
def musica(cat, carac_1, carac_2, min_1, max_1, min_2, max_2):
(1)         m = cat["features"]
(1)         m_reps = mp.newMap(4000,
                               maptype='PROBING',
                               loadfactor=0.5)
(1)         m_tracks = om.newMap(omaptype='RBT',
                                 comparefunction=compareValue)

(1)         a = mp.get(m, carac_1)
(1)         m_1 = me.getValue(a)
(lgN + M)   l_1 = om.values(m_1, min_1, max_1)
(M)         for lists in lt.iterator(l_1):
(P)           for e in lt.iterator(lists):
(1)             mp.put(m_reps, e["track_id"], "")

(1)         b = mp.get(m, carac_2)
(1)         m_2 = me.getValue(b)
(lgN + W)   l_2 = om.values(m_2, min_2, max_2)
(W)         for lists in lt.iterator(l_2):
(Z)           for e in lt.iterator(lists):
```

```
(1)              if mp.contains(m_reps, e["track_id"]):
(lgT)               om.put(m_tracks, e["track_id"], e)

(1)         n_tracks = om.size(m_tracks)
(1)           if n_tracks >= 5:
(1)             num = 5
(1)           else:
(1)             num = n_tracks

(1)         random_tracks = random.sample(range(0, n_tracks), num)
(1)         keys = lt.newList(datastructure="ARRAY_LIST")
(1)         l_tracks = lt.newList(datastructure="ARRAY_LIST")

(5)         for n in random_tracks:
(lgT)         key = om.select(m_tracks, n)
(1)           lt.addLast(keys, key)

(5)         for a in lt.iterator(keys):
(lgT)         rep = om.get(m_tracks, a)
(1)           rep_info = me.getValue(rep)
(1)           lt.addLast(l_tracks, rep_info)

    return (n_tracks, l_tracks)
```

Cálculo de complejidad:

- N= Tamaño de ambos arboles (RBT) de las características consultadas
- M = Tamaño lista de listas de reproducciones que están dentro de los valores consultados para la primera característica
- P = Tamaño lista de reproducciones que están dentro de los valores consultados para la primera característica
- W = Tamaño lista de listas de reproducciones que están dentro de los valores consultados para la segunda característica
- Z = Tamaño lista de reproducciones que están dentro de los valores consultados para la segunda característica
- T = Tamaño árbol (RBT) de tracks únicos con las características solicitadas

$$O(n) = 5 + (lgN + M) + \mathrm{M}(P) + 2 + (lgN + W) + \mathrm{W}(Z(lgT)) + 8 + 5(lgT + 1) + 5(lgT + 2)$$

$$O(n) = 30 + \mathrm{lgN} + \mathrm{M} + \mathrm{M}P + \mathrm{lgN} + \mathrm{W} + \mathrm{WZlgT} + 5lgT + 5lgT$$

$$O(n) = 30 + 2\mathrm{lgN} + 10lgT + \mathrm{M} + \mathrm{W} + \mathrm{M}P + \mathrm{WZlgT}$$

$$O(n) = \mathrm{O}(\mathrm{WZlgT})$$

***Requerimiento 2: Música para festejar***

```
def musicafestejar(cat, minEnergy, maxEnergy, minDanceability, maxDanceability):
(WZlgT)    return musica(cat, "energy", "danceability", minEnergy, maxEnergy, minDanceability, maxDa
nceability)
```

Cálculo de complejidad:

- W = Tamaño lista de listas de reproducciones que están dentro de los valores consultados para “danceability”
- Z = Tamaño lista de reproducciones que están dentro de los valores consultados para “danceability”
- T = Tamaño árbol (RBT) de tracks únicos con las características solicitadas

$$O(n) = O(WZlgT)$$

Discusión

La complejidad de este algoritmo utilizando la notación BIG O es O(WZlgT) esto quiere decir que su orden de crecimiento se asemeja a uno linearítmico por lo que el tiempo de procesamiento va a aumentar en menor proporción a medida que aumenta la cantidad de datos. Cabe resaltar que en este análisis se tomó el producto WZ como una sola variable de tamaño similar a W, lo anterior debido a que se asume que Z es un valor cercano a 1. Esto debido a que, por lo que los valores de features son decimales (float), es poco factible que muchas reproducciones coincidan en una misma llave. Por otro lado, es evidente que T es mucho menor a W, por lo que, siendo más exactos en la obtención de la complejidad, podemos designar la siguiente acotación:

$$O(TlogT) \leq O(WZlgT) \leq O(WlogW)$$

***Análisis de tiempo de ejecución y consumo de memoria***

| Archivos cargados | Consumo de Datos [kB] | Tiempo de Ejecución [ms] |
|---|---|---|
| small | ***0.172*** | ***0.068*** |
| 5 | ***0.172*** | ***0.040*** |

## ***Requerimiento 3: Música para estudiar***

```
def musicaestudiar(cat, minInstru, maxInstru, minTempo, maxTempo):
(WZlgT)    return musica(cat, "tempo", "instrumentalness", minTempo, maxTempo, minInstru, maxInstru)
```

Cálculo de complejidad:

- W = Tamaño lista de listas de reproducciones que están dentro de los valores consultados para “instrumentalness”
- Z = Tamaño lista de reproducciones que están dentro de los valores consultados para “instrumentalness”
- T = Tamaño árbol (RBT) de tracks únicos con las características solicitadas

$$O(n) = O(WZlgT)$$

Discusión

La complejidad de este algoritmo utilizando la notación BIG O es O(WZlgT) esto quiere decir que su orden de crecimiento se asemeja a uno linearítmico por lo que el tiempo de procesamiento va a aumentar en menor proporción a medida que aumenta la cantidad de datos. Cabe resaltar que en este análisis se tomó el producto WZ como una sola variable de tamaño similar a W, lo anterior debido a que se asume que Z es un valor cercano a 1. Esto debido a que, por lo que los valores de features son decimales (float), es poco factible que muchas reproducciones coincidan en una misma llave. Por otro lado, es evidente que T es mucho menor a W, por lo que, siendo más exactos en la obtención de la complejidad, podemos designar la siguiente acotación:

$$O(TlogT) \leq O(WZlgT) \leq O(WlogW)$$

***Análisis de tiempo de ejecución y consumo de memoria***

| Archivos cargados | Consumo de Datos [kB] | Tiempo de Ejecución [ms] |
|---|---|---|
| small | ***0.172*** | ***0.075*** |
| 5 | ***0.172*** | ***0.055*** |

***Requerimiento 4: Estudiar los géneros musicales***

```
def generosmusicales(cat, listgenres):
(1)    l_genres = lt.newList(datastructure="ARRAY_LIST")
(1)    tot_reps = 0

(10)    for gen in listgenres:
(PMlgQ)    info = caracterizarrep(cat, "tempo", gen["min_tempo"], gen["max_tempo"])
(1)        gen["escuchas"] = info[0]
(1)        tot_reps+= gen["escuchas"]
(1)        gen["artistas"] = info[1]
(1)        tree = info[2]
(1)        l_id_artists = lt.newList(datastructure="ARRAY_LIST")
(10)       for i in range(1, 11):
(lgQ)        id_artist = om.select(tree, i)
(1)          lt.addLast(l_id_artists, id_artist)
(1)        gen["id_artistas"] = l_id_artists
(1)        lt.addLast(l_genres, gen)

    return (tot_reps, l_genres)
```

Cálculo de complejidad:

- M = Tamaño lista de listas de reproducciones que están dentro de los parámetros de “tempo” para el género
- P = Tamaño lista de reproducciones que están dentro de los parámetros de “tempo” para el género

- Q = Tamaño árbol donde se guardan los artistas únicos de las reproducciones de los géneros

$$O(n) = 2 + 10(\text{PMlgQ} + 5 + 10(\text{lgQ} + 1) + 2)$$

$$O(n) = 2 + 10\text{PMlgQ} + 50 + 100(\text{lgQ} + 1) + 20$$

$$O(n) = 72 + 10\text{PMlgQ} + 100\text{lgQ} + 100$$

$$O(n) = 172 + 100\text{lgQ} + 10\text{PMlgQ}$$

$$O(n) = 10\text{PMlgQ}$$

$$O(n) = \text{O}(\text{PMlgQ})$$

Discusión

La complejidad de este algoritmo utilizando la notación BIG O es O(PMlgQ) esto quiere decir que su orden de crecimiento se asemeja a uno linearítmico por lo que el tiempo de procesamiento va a aumentar en menor proporción a medida que aumenta la cantidad de datos. Cabe resaltar que en este análisis se tomó el producto PM como una sola variable de tamaño similar a M, lo anterior debido a que se asume que P es un valor cercano a 1. Esto debido a que, por lo que los valores de features son decimales (float), es poco factible que muchas reproducciones coincidan en una misma llave. Por otro lado, es evidente que Q es mucho menor a M, por lo que, siendo mas exactos en la obtención de la complejidad podemos designar la siguiente acotación:

$$\text{O}(\text{QlogQ}) \leq \text{O}(\text{PMlgQ}) \leq \text{O}(\text{MlogM})$$

***Análisis de tiempo de ejecución y consumo de memoria***

| Archivos cargados | Consumo de Datos [kB] | Tiempo de Ejecución [ms] |
|---|---|---|
| small | ***0.375*** | ***0.184*** |
| 5 | ***0.375*** | ***0.068*** |

***Requerimiento 5: Género mas escuchado en un tiempo***

```
def generotiempo(cat, hora_1, hora_2):
(1)     m_g = cat["genres"]
(9)     l_gen_name = mp.keySet(m_g)
(1)     l_gen_reps = lt.newList(datastructure="ARRAY_LIST")
(1)     l_max_tracks_genres = ""
(1)     max_reps_genre = 0
(1)     tot_reps = 0
(9)     for gen in lt.iterator(l_gen_name):
(1)         a = mp.get(m_g, gen)
(1)         m_gen = me.getValue(a)
(lgM + Q)   l_reps = om.values(m_gen, hora_1, hora_2)
(1)         reps = 0
(1)         l_gen = lt.newList(datastructure="ARRAY_LIST", cmpfunction=compareValue)
(Q)         for e in lt.iterator(l_reps):
(1)             size = lt.size(e)
(1)             reps+=size
```

```
(P)            for i in lt.iterator(e):
(W)                if lt.isPresent(l_gen, i["track_id"]) == 0:
(1)                    lt.addLast(l_gen, i["track_id"])
(1)        if reps > max_reps_genre:
(1)            max_reps_genre = reps
(1)            l_max_tracks_genres = l_gen
(1)        tot_reps+=reps
(1)        dic = {"nombre": gen, "reps": reps}
(1)        lt.addLast(l_gen_reps, dic)

(1)    unique_tracks = lt.size(l_max_tracks_genres)
(9^3/2)sorted_gen = sortReps(l_gen_reps, compareReps)
(1)    top_genre = lt.getElement(sorted_gen, 0)

(1)    r = lt.newList(datastructure="ARRAY_LIST")
(1)    m_h = cat["hashtags"]
(1)    m_s = cat["sentiment"]
(W)    for i in lt.iterator(l_max_tracks_genres):
(1)        num_hashtags = 0
(1)        sum_vader = 0
(1)        a = mp.get(m_h, i)
(1)        l_hashtags = me.getValue(a)
(H)        for e in lt.iterator(l_hashtags):
(1)            h = e["hashtag"].lower()
(1)            if (e["created_at"] >= hora_1) and (e["created_at"] <= hora_2):
(1)                if mp.contains(m_s, h):
(1)                    b = mp.get(m_s, h)
(1)                    vader = me.getValue(b)
(1)                    if vader != 100:
(1)                        num_hashtags+=1
(1)                        sum_vader+=vader
(1)        if num_hashtags == 0:
(1)            num_hashtags = 1
(1)        vader_avg = sum_vader/num_hashtags
(1)        dic = {}
(1)        dic["track_id"] = i
(1)        dic["numero hashtags"] = num_hashtags
(1)        dic["vader promedio"] = vader_avg
(1)        lt.addLast(r, dic)

(W^3/2)sorted_ht = sortReps(r, compareTrackHashtags)
(10)   top_tracks = lt.subList(sorted_ht, 1, 10)

    return (tot_reps, sorted_gen, unique_tracks, top_tracks)
```

Cálculo de complejidad:

- M= Tamaño de árbol (RBT) de cada genero
- Q = Tamaño lista de listas de reproducciones que están dentro de las horas consultadas para el género
- P = Tamaño lista de reproducciones que están dentro de las horas consultadas para el género
- W = Tamaño lista de tracks únicos que están dentro de las horas consultadas para el género
- H = Lista de hashtags para el track único

$$O(n) = 14 + 9\left(10 + (\lg M + Q) + Q\left(2 + P(W + 1)\right)\right) + 2 + 9\hat{}\frac{3}{2} + 3 + W(4 + H(8) + 8) + W\hat{}\frac{3}{2} + 10$$

$$O(n) = 56 + 9\left(10 + \lg M + Q + Q(2 + P + PW)\right) + 12W + 8WH + W\hat{}\frac{3}{2}$$

$$O(n) = 56 + 90 + 9\lg M + 9Q + 9Q(2 + P + PW) + 12W + 8WH + W\hat{}\frac{3}{2}$$

$$O(n) = 146 + 9\lg M + 27Q + 9QP + 9QPW + 12W + 8WH + W\hat{}\frac{3}{2}$$

$$O(n) = O(QPW)$$

Discusión

La complejidad de este algoritmo utilizando la notación BIG O es O(QPW) esto quiere decir que su orden de crecimiento se asemeja a uno cuadrático por lo que el tiempo de procesamiento va a aumentar considerablemente a medida que aumenta la cantidad de datos. Cabe resaltar que en este análisis se tomó el producto QP como una sola variable de tamaño similar a Q, lo anterior debido a que se asume que P es un valor cercano a 1. Esto debido a que, por lo que los valores de horas son tan exactos(con horas, minutos y segundos), es poco factible que muchas reproducciones coincidan en una misma llave. Por otro lado, es evidente que W es mucho menor a Q, por lo que, siendo mas exactos en la obtención de la complejidad, podemos designar la siguiente acotación:

$$O(w^2) \leq O(QPW) \leq O(Q^2)$$

***Análisis de tiempo de ejecución y consumo de memoria***

| Archivos cargados | Consumo de Datos [kB] | Tiempo de Ejecución [ms] |
|---|---|---|
| small | ***0.172*** | ***0.082*** |
| 5 | ***0.172*** | ***0.586*** |