OBSERVACIONES & ANÁLISIS – RETO 3

Req2 - María Alejandra Estrada García Cod. 202021060

Req3 - Santiago Martínez Novoa Cod. 202112020

Análisis de complejidad

Requerimiento 1: (Grupal)

- Este requerimiento tiene como función principal, y el cual se llama en el controller es, getEventsByCity(…). Tiene una complejidad de O(NLogN). En esta función lo que se busca es contar los avistamientos en una ciudad determinada, la cual nos dan como parámetro. Para ello, se hace un ciclo el cual se va a filtrar la lista obtenida de las llaves del mapa ordenado 'city', y luego dentro se saca los eventos los cuales se encuentra en la ciudad, para eso se usó las funciones get(...) y getValues(...). A la lista que se logró obtener de los avistamientos en una ciudad, se ordena por medio de mergesort (de menor a mayor), el cual tiene una complejidad O(NLogN), por lo que la complejidad final del requerimiento es O(NLogN).

```
#Req 1:
def getEventsByCity(analyzer,ciudad):
    start_time = time.process_time()
    ciudadTree = analyzer['city']
    keys = om.keySet(ciudadTree)
    eye_city = ""
    sightings = 0
    for ciudads in lt.iterator(keys):
        par = om.get(analyzer['city'], ciudads)
        city_value = me.getValue(par)
        if lt.size(city_value['events']) > sightings:
            eye_city = ciudads
            sightings = lt.size(city_value['events'])
    city_especific = om.get(analyzer['city'], ciudad)
    specific = me.getValue(city_especific)
    list_specific = specific['events']
    sortDurationHM(list_specific)
    stop_time = time.process_time()
    elapsed_time_mseg = (stop_time - start_time)*1000
    return sightings, list_specific, elapsed_time_mseg, eye_city
```

Requerimiento 2: (Individual)

- En este requerimiento lo que se busca es contar los avistamientos por un rango de duración de segundos. Asimismo, la función principal es, getEventsByDurationS(…). En esta función lo que se busca es la maxima duración, por medio de la función maxKey(...) y sacar los valores (eventos) de la llave, y con la funciòn lt.size(...), contar

la cantidad de avistamientos que tiene la duraciòn màxima. Luego se hace un ciclo con for donde se dé una iteración con for y la función iterator del TAD lista para avanzar en una lista y luego el de los eventos los cuales se encuentran dentro del rango de duración. Posteriormente se organizan de acuerdo a la duración. Finalmente, tiene una complejidad de O(N*M).

```
#Req 2:
def getEventsByDurationS(analyzer, minSeg, maxSeg):
    start_time = time.process_time()
    durationSegTree = analyzer['duration(seconds)']
    maxK = om.maxKey(durationSegTree)
    maxget = om.get(durationSegTree, maxK)
    maxvalues = me.getValue(maxget)
    maxsize = lt.size(maxvalues['events'])

    lst = om.values(analyzer['duration(seconds)'], minSeg, maxSeg)
    lista_duracionSeg = lt.newList('ARRAY_LIST')

    for i in lt.iterator(lst):
        i = i['events']
        for j in lt.iterator(i):
            lt.addLast(lista_duracionSeg, j)

    sortDurationS(lista_duracionSeg) #Se organiza cronologicamente
    stop_time = time.process_time()
    elapsed_time_mseg = (stop_time - start_time)*1000
    return maxsize, lista_duracionSeg, elapsed_time_mseg, maxK
```

Requerimiento 3: (Individual)

- El requerimiento 3 busca contar los avistamientos por duración hora/minuto del día. La función principal de este requerimiento es getEventsByRangeDate(...). En esta función se hace un ciclo for con iterador del TAD lista, con los valores del rango de duración obtenido por medio de la función values(...), y dentro de realiza otro ciclo for con el iterador del TAD lista de los eventos dentro del rango, y cada evento se va añadiendo dentro de la lista vacía previamente hecha, la complejidad sería O(N*M). La lista se organiza por medio de un mergesort.sort() cronologicamente, menor a mayor; la función tiene como complejidad O(NLogN). Por último, la complejidad final del requerimiento es O(N*M).

```python
#Req 3:
def getEventsByRangeDate(analyzer, minDate, maxDate):

    start_time = time.process_time()
    datetimetree = analyzer['datetime']
    maxK = om.maxKey(datetimetree)
    maxget = om.get(datetimetree, maxK)
    maxvalues = me.getValue(maxget)
    maxsize = lt.size(maxvalues['events'])
    fminDate=datetime.datetime.strptime(minDate, '%H:%M:%S')
    fmaxDate=datetime.datetime.strptime(maxDate, '%H:%M:%S')
    lst = om.values(analyzer['time'], fminDate.time(), fmaxDate.time())
    lista_datesinrange = lt.newList('ARRAY_LIST')

    for i in lt.iterator(lst):
        i = i['events']
        for j in lt.iterator(i):
            lt.addLast(lista_datesinrange, j)

    sortDurationHM(lista_datesinrange)
    stop_time = time.process_time()
    elapsed_time_mseg = (stop_time - start_time)*1000
    return maxsize,lista_datesinrange, elapsed_time_mseg, maxK
```

Requerimiento 4: (Grupal)

- Este requerimiento busca contar la cantidad de avistamientos en un rango de fechas. La función principal es geteventsByDatetime(…), el cual tiene una complejidad de O(N*M), ya que se realizan dos ciclos con for y la función del iterator de TAD lista y luego otro ciclo for con la función iterator para los eventos dentro del rango de fechas. Asimismo, por medio de la función size, se busca contar los avistamientos en el rango de fechas. Posteriormente, se organiza la lista de menor a mayor cronológicamente, por medio de mergesort.sort() el cual tiene complejidad O(NLogN). Por último, la complejidad final del requerimiento es O(N*M).

```python
def geteventsByDatetime(analyzer, datemin, datemax):
    start_time = time.process_time()
    datetimetree = analyzer['datetime']
    minK = om.minKey(datetimetree)
    minget = om.get(datetimetree, minK)
    minvalues = me.getValue(minget)
    minsize = lt.size(minvalues['events'])
    fminDate = datetime.datetime.strptime(datemin, '%Y-%m-%d')
    fmaxDate = datetime.datetime.strptime(datemax, '%Y-%m-%d')
    lst = om.values(analyzer['datetime'], fminDate.date(), fmaxDate.date())
    lista_datesinrange = lt.newList('ARRAY_LIST')

    for i in lt.iterator(lst):
        i = i['events']
        for j in lt.iterator(i):
            lt.addLast(lista_datesinrange, j)

    sortDurationHM(lista_datesinrange)
    stop_time = time.process_time()
    elapsed_time_mseg = (stop_time - start_time)*1000
    return minsize,lista_datesinrange, elapsed_time_mseg, minK
```

Requerimiento 5: (Grupal)

- Para este requerimiento, se utilizó la función principal evetLongLat(...). Tiene una complejidad estimada es O(N^2), ya que en el ciclo más significativo se vale de una iteración con for y la función iterator del TAD lista para avanzar en una lista y a medida que avanza va buscando si elemento ya se encuentra en otra lista que se pretende retornar, de manera que, se tiene un ciclo dentro de otro. Adicionalmente, se organiza la función por medio de la función mergesort.sort() y se organiza de mayor a menor por latitud, esta función tiene una complejidad de O(NLogN). Po rúltimo, la complejidad final del requerimiento es O(N^2).

```
#Req 5:

def eventLongLat(analyzer, latmin, latmax, longmin, longmax):
    start_time = time.process_time()
    longmap = analyzer['longitud']
    lst1 = om.values(longmap, longmin, longmax)
    comb = om.newMap(omaptype='RBT', comparefunction=compareLL)
    lista_filtrada = lt.newList('ARRAY_LIST')

    for i in lt.iterator(lst1):
        i = i['events']
        for j in lt.iterator(i):
            addLatitud(comb, j)

    listasemifiltrada =  om.values(comb, latmin, latmax)

    for i in lt.iterator(listasemifiltrada):
        i = i['events']
        for j in lt.iterator(i):
            lt.addLast(lista_filtrada, j)

    sortLatitud(lista_filtrada)
    total = lt.size(lista_filtrada)
    stop_time = time.process_time()
    elapsed_time_mseg = (stop_time - start_time)*1000
    return lista_filtrada, elapsed_time_mseg, total
```

Requerimiento 6: (Bono)

- En este requerimiento se busca visualizar los avistamientos/eventos de una zona geográfica, en este caso del requerimiento 5, por medio de la librería folium. La función principal de este requerimiento es visualizarEventosZonaG(). En esta función se crea un archivo html donde se presentan los avistamientos dentro de la zona geográfica ingresada (latitud mínima y máxima, y longitud mínima y máxima). Esto se hace mediante el uso de la librería folium, la cual crea un mapa con los datos ingresados por el usuario para latitud y longitud, luego de eso se implementa un for loop se añaden todos los avistamientos de la zona que se quiere visualizar, y además se añade la información de dichos avistamientos en el rectángulo del mapa creado por el folium.Map() y folium.Rectangle(). Por este ciclo la complejidad del requerimiento es O(N).

```
def visualizarEventoZonaG(eventLL, latmin, latmax, longmin, longmax):
    promedio_lat = (latmax + latmin)/ 2
    latP = round(promedio_lat, 2)
    promedio_long = (longmax + longmin)/ 2
    longP = round(promedio_long, 2)

    mapaF = folium.Map(location=(latP, longP), zoom_start = 8)

    folium.Rectangle([(latmin, latmax), (longmin, longmax)],
                     fill=True,
                     weight=5,
                     fill_color="orange",
                     color="green").add_to(mapaF)

    cityL = []
    datetimeL = []
    durationSL = []
    shapeL = []
    commentsL = []
    countryL = []
    latL = []
    longL = []
    for element in lt.iterator(eventLL[0]):
        cityL.append(element['city'])
        datetimeL.append(element['datetime'])
        durationSL.append(element['duration (seconds)'])
        shapeL.append(element['shape'])
        commentsL.append(element['comments'])
        countryL.append(element['country'])
        latL.append(element['latitude'])
        longL.append(element['longitude'])
```

```
    for  city, datetime, durationS, shape, comments, country, lat, long in zip(cityL, datetimeL, durationSL, shapeL, commentsL, country

        info = 'Datetime: ' + str(datetime) + ', City: ' + str(city) + ', Country:' + str(country) + ', Shape: ' + str(shape) + ', Dur

        folium.Marker(
                    location=[lat,long],
                    popup=folium.Popup(info, max_widht=500),
                    icon=folium.Icon(color='green', icon_color='white', icon='info-sign', angle=0, prefix='glyphicon')).add_to(mapaF)


    mapaF.save(outfile='eventos.html')
```