Análisis de Reto 3 – Entrega Final

Integrantes:

- Juan David Vasquez Hernández - jd.vasquezh@uniandes.edu.co – 201914782
- Briseth Rodríguez Tovar – b.rodriguezt@uniandes.edu.co – 202116910

Complejidad de cada requerimiento:

Los códigos implementados para la solución de los requerimientos se muestran a continuación acompañados de sus respectivas complejidades.

En el caso de todos los algoritmos se usará **n** para representar el tamaño de UFOs y **k** para representar el número de llaves en el cual se organizan los datos en un árbol ordenado.

Req1:

```
def create_city_index(catalog):
    catalog['city_index'] = om.newMap(omaptype='RBT',
comparefunction=compareNames)
    city_index = catalog['city_index']
    for ufo_data in lt.iterator(catalog['UFOs']):
        city_info = ufo_data['city']
        date_info = ufo_data['datetime']
        if om.contains(city_index,city_info):
            date_index = om.get(city_index,city_info)['value']
            if om.contains(date_index,date_info):
                list_UFOs = om.get(date_index,date_info)['value']
                lt.addLast(list_UFOs, ufo_data)
            else:
                list_UFOs = lt.newList()
                lt.addLast(list_UFOs,ufo_data)
                om.put(date_index,date_info,list_UFOs)
        else:
            date_index = om.newMap(omaptype='RBT',
comparefunction=compareDates)
            list_UFOs = lt.newList()
            lt.addLast(list_UFOs,ufo_data)
            om.put(date_index,date_info,list_UFOs)
            om.put(city_index,city_info,date_index)

def getSightingsByCity(catalog, city):
    city_index = catalog['city_index']
    date_index = om.get(city_index,city)['value']
    return om.size(city_index), date_index
```

Para el primer requerimiento, la función create_city_index( ) posee una complejidad O(n), dado que recorre únicamente una vez la lista de elementos para poder crear la estructura de datos para el problema. Para la función getSightingsByCity( ) la complejidad es O(logk) en el caso en el que el árbol se encuentre balanceado. En caso contrario, la complejidad de búsqueda vendría siendo O(2logk) dado que se usa un árbol RBT. La complejidad total del algoritmo en el peor caso con el árbol no balanceado vendría siendo O(n) + O(2logk). La situación cambiaría si se llegase a usar un árbol BST, dado que la complejidad en el caso no balanceado para buscar un elemento sería O(k).

Req 2 (Briseth Rodríguez):

```
def create_duration_index(catalog):
    catalog['duration_index'] = om.newMap(omaptype='RBT',
comparefunction=compareDuration)
    duration_index = catalog['duration_index']
    for ufo_data in lt.iterator(catalog['UFOs']):
        duration_info = round(float(ufo_data['duration (seconds)']),1)
        country_info = ufo_data['country']
        city_info= ufo_data['city']
        country_city=  city_info + '-' + country_info
        if om.contains(duration_index,duration_info):
            country_city_index =
om.get(duration_index,duration_info)['value']
            if om.contains(country_city_index,country_city):
                list_UFOs = om.get(country_city_index,country_city)['value']
                lt.addLast(list_UFOs, ufo_data)
            else:
                list_UFOs = lt.newList()
                lt.addLast(list_UFOs,ufo_data)
                om.put(country_city_index,country_city,list_UFOs)
        else:
            country_city_index = om.newMap(omaptype='RBT',
comparefunction=compareLocation)
            list_UFOs = lt.newList()
            lt.addLast(list_UFOs,ufo_data)
            om.put(country_city_index,country_city,list_UFOs)
            om.put(duration_index,duration_info,country_city_index)

def getSightingsByDuration(catalog,duration_min,duration_max):
    duration_index = catalog['duration_index']
    ufo_list = lt.newList()
    latest_duration = om.maxKey(duration_index)
    latest_country_city = om.get(duration_index,latest_duration)['value']
    latest_sightings = 0
```

```
    for country_city in lt.iterator(om.keySet(latest_country_city)):
        latest_sightings +=
lt.size(om.get(latest_country_city,country_city)['value'])

    keys_duration = om.keys(duration_index,duration_min,duration_max)
    for key_duration in lt.iterator(keys_duration):
        country_city_index = om.get(duration_index,key_duration)['value']
        for key_country_city in lt.iterator(om.keySet(country_city_index)):
            ufo_info = om.get(country_city_index,key_country_city)['value']
            for ufo in lt.iterator(ufo_info):
                lt.addLast(ufo_list,ufo)
    return latest_duration, latest_sightings, ufo_list
```

Para el segundo requerimiento, la función create_duration_index( ) posee una complejidad O(n), dado que recorre únicamente una vez la lista de elementos para poder crear la estructura de datos para el problema. Para la función getSightingsByDuration( ) la complejidad es $O(\log k_1 + \log k_1 * \log k_2)$ donde k_1 son las llaves pertenecientes al árbol ordenado por duración (segundos) y k_2 las llaves de cada sub árbol ordenado por la combinación ciudad-país – en el caso en el que el árbol y los subárboles se encuentren balanceados. En caso contrario, la complejidad de búsqueda vendría siendo $O(2\log k_1 + 4 * \log k_1 * \log k_2)$ dado que se usa un árbol RBT. La complejidad total del algoritmo en el peor caso con el árbol no balanceado vendría siendo $O(n) + O(2\log k_1 + 4 * \log k_1 * \log k_2)$. La situación cambiaría si se llegase a usar un árbol BST, dado que la complejidad en el caso no balanceado para buscar un elemento sería O(k). Esto en el caso no balanceado daría un valor de $O(n) + O(k_1 + k_1 * k_2)$

Req 3 (Juan Vásquez):

```
def create_time_index(catalog):
    catalog['time_index'] = om.newMap(omaptype='RBT',
comparefunction=compareTime)
    time_index = catalog['time_index']
    for ufo_data in lt.iterator(catalog['UFOs']):
        time_info = ufo_data['datetime'][11:19]
        date_info = ufo_data['datetime'][0:10]
        if om.contains(time_index,time_info):
            date_index = om.get(time_index,time_info)['value']
            if om.contains(date_index,date_info):
                list_UFOs = om.get(date_index,date_info)['value']
                lt.addLast(list_UFOs, ufo_data)
            else:
                list_UFOs = lt.newList()
                lt.addLast(list_UFOs,ufo_data)
                om.put(date_index,date_info,list_UFOs)
```

```
        else:
            date_index = om.newMap(omaptype='RBT',
comparefunction=compareTime)
            list_UFOs = lt.newList()
            lt.addLast(list_UFOs,ufo_data)
            om.put(date_index,date_info,list_UFOs)
            om.put(time_index,time_info,date_index)

def getSightingsByTime(catalog,time_min,time_max):
    time_index = catalog['time_index']
    ufo_list = lt.newList()
    latest_time = om.maxKey(time_index)
    latest_dates = om.get(time_index,latest_time)['value']
    latest_sightings = 0
    for date in lt.iterator(om.keySet(latest_dates)):
        latest_sightings += lt.size(om.get(latest_dates,date))
    keys_time = om.keys(time_index,time_min,time_max)
    for key_time in lt.iterator(keys_time):
        date_index = om.get(time_index,key_time)['value']
        for key_date in lt.iterator(om.keySet(date_index)):
            ufo_info = om.get(date_index,key_date)['value']
            for ufo in lt.iterator(ufo_info):
                lt.addLast(ufo_list,ufo)
    return latest_time, latest_sightings, ufo_list
```

Para el tercer requerimiento, la función create_time_index( ) posee una complejidad O(n), dado que recorre únicamente una vez la lista de elementos para poder crear la estructura de datos para el problema. Para la función getSightingsByTime( ) la complejidad es $O(\log k_1 + \log k_1 * \log k_2)$ donde k_1 son las llaves pertenecientes al árbol ordenado por tiempo HH:MM y k_2 las llaves de cada sub árbol ordenado por fecha – en el caso en el que el árbol y los subárboles se encuentren balanceados. En caso contrario, la complejidad de búsqueda vendría siendo $O(2\log k_1 + 4*\log k_1 * \log k_2)$ dado que se usa un árbol RBT. La complejidad total del algoritmo en el peor caso con el árbol no balanceado vendría siendo $O(n) + O(2\log k_1 + 4*\log k_1 * \log k_2)$. La situación cambiaría si se llegase a usar un árbol BST, dado que la complejidad en el caso no balanceado para buscar un elemento sería O(k). Esto en el caso no balanceado daría un valor de $O(n) + O(k_1 + k_1 * k_2)$

Req 4:

```
def create_date_index(catalog):
    catalog['date_index'] = om.newMap(omaptype='RBT',
comparefunction=compareDates)
    date_index = catalog['date_index']
```

```
    for ufo_data in lt.iterator(catalog['UFOs']):
        date_info = ufo_data['datetime']
        if om.contains(date_index,date_info):
            list_UFOs = om.get(date_index,date_info)['value']
            lt.addLast(list_UFOs, ufo_data)
        else:
            list_UFOs = lt.newList()
            lt.addLast(list_UFOs,ufo_data)
            om.put(date_index,date_info,list_UFOs)

def getSightingsByDate(catalog,initial_date,final_date):
    date_index = catalog['date_index']
    dates = lt.newList()
    keyMax = om.floor(date_index,final_date)
    final_dates = om.get(date_index,keyMax)['value']
    for date_list in
lt.iterator(om.values(date_index,initial_date,final_date)):
        for date_info in lt.iterator(date_list):
            lt.addLast(dates,date_info)
    return dates, final_dates
```

Para el cuarto requerimiento, la función create_date_index( ) posee una complejidad O(n), dado que recorre únicamente una vez la lista de elementos para poder crear la estructura de datos para el problema. Para la función getSightingsByDate( ) la complejidad es O(logk) en el caso en el que el árbol se encuentre balanceado. En caso contrario, la complejidad de búsqueda vendría siendo O(2logk) dado que se usa un árbol RBT. La complejidad total del algoritmo en el peor caso con el árbol no balanceado vendría siendo O(n) + O(2logk). La situación cambiaría si se llegase a usar un árbol BST, dado que la complejidad en el caso no balanceado para buscar un elemento sería O(k).

Req 5:

```
def create_coord_index(catalog):
    catalog['coord_index'] = om.newMap(omaptype='RBT',
comparefunction=compareCoord)
    coord_index = catalog['coord_index']
    for ufo_data in lt.iterator(catalog['UFOs']):
        latitude_info = str(round(float(ufo_data['latitude']),2))
        longitude_info = str(round(float(ufo_data['longitude']),2))
        if om.contains(coord_index,latitude_info):
            longitude_index = om.get(coord_index,latitude_info)['value']
            if om.contains(longitude_index,longitude_info):
                list_UFOs = om.get(longitude_index,longitude_info)['value']
```

```
                lt.addLast(list_UFOs, ufo_data)
            else:
                list_UFOs = lt.newList()
                lt.addLast(list_UFOs,ufo_data)
                om.put(longitude_index,longitude_info,list_UFOs)
        else:
            longitude_index = om.newMap(omaptype='RBT',
comparefunction=compareCoord)
            list_UFOs = lt.newList()
            lt.addLast(list_UFOs,ufo_data)
            om.put(longitude_index,longitude_info,list_UFOs)
            om.put(coord_index,latitude_info,longitude_index)

def
getSightingsByGeography(catalog,longitude_min,longitude_max,latitude_min,lat
itude_max):
    coord_index = catalog['coord_index']
    ufo_list = lt.newList()
    keys_latitude = om.keys(coord_index,latitude_min,latitude_max)
    for key_latitude in lt.iterator(keys_latitude):
        latitude_index = om.get(coord_index,key_latitude)['value']
        keys_longitude = om.keys(latitude_index,longitude_min,longitude_max)
        for key_longitude in lt.iterator(keys_longitude):
            ufo_info = om.get(latitude_index,key_longitude)['value']
            for ufo in lt.iterator(ufo_info):
                lt.addLast(ufo_list,ufo)
    return ufo_list
```

Para el quinto requerimiento, la función create_time_index( ) posee una complejidad O(n), dado que recorre únicamente una vez la lista de elementos para poder crear la estructura de datos para el problema. Para la función getSightingsByTime( ) la complejidad es $O(\log k_1 + \log k_1 * \log k_2)$ donde k_1 son las llaves pertenecientes al árbol ordenado por latitudes y k_2 las llaves de cada sub árbol ordenado por longitudes – en el caso en el que el árbol y los subárboles se encuentren balanceados. En caso contrario, la complejidad de búsqueda vendría siendo $O(2\log k_1 + 4 * \log k_1 * \log k_2)$ dado que se usa un árbol RBT. La complejidad total del algoritmo en el peor caso con el árbol no balanceado vendría siendo $O(n) + O(2\log k_1 + 4 * \log k_1 * \log k_2)$. La situación cambiaría si se llegase a usar un árbol BST, dado que la complejidad en el caso no balanceado para buscar un elemento sería O(k). Esto en el caso no balanceado daría un valor de $O(n) + O(k_1 * k_2)$