Análisis de Complejidad
Reto 3

Nombre: Sergio David López Becerra
Código: 202110260
Requerimiento: 2
Nombre: Sergio Montoya Ramirez
Código: 202112171
Requerimiento: 3

1.

```
def avistamiento_ciudad(analyser,ciudad):
    start_time = chronos.process_time()
    lista = me.getValue(om.get(analyser["ciudad"],ciudad))
    lista = lista["lstUFOS"]
    lista_sorted = merge_sort(lista,lt.size(lista),cmpdatetime)
    stop_time = chronos.process_time()
    time = (stop_time - start_time)*1000
    print("se demoro: ", time)
    return lista_sorted
```

Las primeras dos variables como se puede ver son simplemente asignación, sin embargo la primera en concreto es ligeramente más interesante pues dado que debe recorrer el árbol su complejidad no es O(1). La complejidad de esta primera variable es O(2 logn) Dado que nosotros usamos árboles RBT para poder asegurar esto. Además de eso el resultado de ambas variables es una lista con los avistamientos de una ciudad específica (dado que lo sacamos de un árbol que tiene como llaves las ciudades)

Ahora bien, la siguiente variable es ya aprovechando que tenemos una lista esta en concreto lo que hace es que organiza la lista que sacamos. Esta organización lo hace con un merge sort que como sabemos tiene una complejidad de O(nLogn).

Ahora bien como vemos que tenemos dos complejidades nos quedamos con la más grande de las dos y entonces quedaria asi:
Mejor caso:
O(2 logn)
Peor caso:
O(nlogn)

2.

```
def avistamientos_segundos(analyser,s_min,s_max):
    """
    Primera parte del Req-2
    """
    start_time = chronos.process_time()
    high_key = om.maxKey(analyser['segundos'])
    tupla = om.get(analyser["segundos"],high_key)
    mayores = tupla["value"]["lstUFOS"]
    mayor_cantidad = lt.size(mayores)
```

Para el requerimiento dos nos piden dos consultas que realizamos de forma independiente. En la primera, hacemos uso de om.maxKey (cuya complejidad es O(logN), debido a que hace un recorrido yéndose sólo por las ramas de derecha) y culminamos obteniendo el tamaño de una lista, cuya complejidad es de O(k). Por consiguiente tanto para el mejor como para el peor caso es: O(logN).

```
"""
Segunda parte del req-2
"""
avistamientos = lt.newList(datastructure="ARRAY_LIST")
s_inicio = s_min
while s_inicio <= s_max:
    ufos = om.get(analyser['segundos'],s_inicio)
    if ufos != None:
        ufos = me.getValue(ufos)
        ufos = ufos["lstUFOS"]
        for avistamiento in lt.iterator(ufos):
            lt.addLast(avistamientos,avistamiento)
    s_inicio += 0.5

primeras_3 = lt.newList(datastructure="ARRAY_LIST")
for posicion in range(4):
    lt.addLast(primeras_3,lt.getElement(avistamientos,posicion))
ultimas_3 = lt.newList(datastructure="ARRAY_LIST")
for posicion in range(lt.size(avistamientos)-3,lt.size(avistamientos)):
    lt.addLast(ultimas_3,lt.getElement(avistamientos,posicion))
```

Para la segunda parte, tenemos un while que busca una llave específica (O(k)) dentro de un rango que varía según lo que solicite el usuario, por ello, llamemos a la cardinalidad del rango N, tal que el while tiene complejidad O(N); ahora bien, cómo se realiza un for dentro del while para cada uno de los avistamientos en una llave específica,esta nueva longitud la llamaremos M, tal que para el peor caso la complejidad será de: O(NM). Notemos que el mejor caso será si el rango a buscar es de un sólo valor y la cantidad de avistamientos para esa llave sea de uno, teniendo así, O(k).

3.

```
def avistamientos_hora(analyser,hora_inicio,hora_fin):
    start_time = chronos.process_time()
    mas_tarde = str(om.maxKey(analyser['hora']))[11:]
    avistamientos = lt.newList(datastructure="ARRAY_LIST")
    hora = hora_inicio
    while hora <= hora_fin:
        avistamiento = om.get(analyser["hora"],hora)
        if avistamiento != None:
            avistamiento = me.getValue(avistamiento)
            avistamiento = avistamiento["lstUFOS"]
            for evento in lt.iterator(avistamiento):
                lt.addLast(avistamientos,evento)
        hora += datetime.timedelta(minutes = 1)
    avistamientos_sorted = merge_sort(avistamientos,lt.size(avistamientos),cmptime)
    stop_time = chronos.process_time()
    time = (stop_time - start_time)*1000
    print("se demoro: ", time)
    return avistamientos_sorted, mas_tarde
```

En este iniciamos buscando cual es el elemento más grande de un árbol que está organizado por horas y nos quedamos con sus últimos elementos (desde el 11) esto para que solo sea la hora y no fecha - hora (fecha hora por que el objeto está guardado como datetime y por ende debía tener un date). Este proceso, dado que estamos trabajando con un arbol rojo-negro, tiene una complejidad de O(logn) (pues se reparten balanceados los datos y para llegar al max debe buscar repetidamente a la derecha)

Después creamos una lista (en concreto una arraylist) que dado que es solo su creación tiene una complejidad de O(k)

Proseguimos con una asignación poco interesante (O(k))

Luego de esto pasamos por un while al cual visitaremos una vez por cada minuto entre las dos horas pedidas. Lo que significa que tiene una complejidad de O(m)

Luego de eso sacamos un valor del árbol que es O(2logn) pero como esta orden está dentro del while se repite m veces (siendo m el número de minutos en el rango) por ende la complejidad es (2mlogn).

Después, hacemos un if que es O(k) pues simplemente prueba que la variable anterior no esté vacía y que se repite n veces O(K*n).

Luego sacamos la información de la pareja llave valor O(k) lo hacemos p veces (con p siendo cantidad de veces en las que entramos al if) O(P*K)

y luego tenemos un for que va por cada una de las obras que hay en una hora concreta. Este for se repite n número de avistamientos por minuto pero es que además for ocurre la cantidad p de veces en la que se ingresa al if. Por ende su complejidad es O(n*p)

posteriormente se agregan cada uno de los elementos O(1*n*p) lo cual es un O(n*p)

luego de eso, tenemos una suma O(k) para que el while no sea un ciclo infinito pero que se repite n veces O(k*n)

por último, con la lista ya acomodada hacemos un merge_sort (con una complejidad O(n*mlogn*m) dado que mergesort tiene una complejidad base de (log n) pero nuestro n en este caso es la cantidad de días

Por todo esto, en el peor caso se daría una complejidad de:
O(n*mlogn*m)
4.

```
def avistamientos_fecha(analyser,fecha_inicias,fecha_final):
    start_time = chronos.process_time()
    fecha = fecha_inicias
    avistamientos = lt.newList(datastructure="ARRAY_LIST")
    while fecha <= fecha_final:
        avistamiento = om.get(analyser["fecha"],fecha)
        if avistamiento != None:
            avistamiento = me.getValue(avistamiento)
            avistamiento = avistamiento["lstUFOS"]
            for evento in lt.iterator(avistamiento):
                lt.addLast(avistamientos,evento)
        fecha += datetime.timedelta(1,0,0)
    avistamientos_sorted = merge_sort(avistamientos,lt.size(avistamientos),cmpdatetime)
    stop_time = chronos.process_time()
    time = (stop_time - start_time)*1000
    print("se demoro: ", time)
    return avistamientos_sorted
```

Al igual que en el requerimiento 2 y 3, se hace un while basado en un rango establecido por el usuario (llamaremos a la cardinalidad de dicho rango N) y un for para cada avistamiento que se tiene por la llave solicitada (esta longitud será M), con lo que tendremos una complejidad de O(NM). Por otro lado, a esos datos que acabamos de extraer, los ordenaremos haciendo uso de merge sort (cuya complejidad es O(Plog P), siendo P = MN). Por consiguiente, la complejidad del requerimiento será de O(P logP).

5.

```
def avistamientos_lugar(analyser,latitud_max,latitud_min,longitud_max,longitud_min):
    start_time = chronos.process_time()
    latitud = latitud_min
    avistamientos = lt.newList(datastructure='ARRAY_LIST')
    while latitud <= latitud_max:
        avistamiento = om.get(analyser['lugar'],latitud)
        if avistamiento != None:
            avistamiento = me.getValue(avistamiento)
            avistamiento = avistamiento["lstUFOS"]
            for evento in lt.iterator(avistamiento):
                if float(evento['longitude']) >= longitud_min and float(evento['longitude']) <= longitud_max:
                    lt.addLast(avistamientos,evento)
        latitud += 0.01
        latitud = round(latitud,2)
    avistamientos_sorted = merge_sort(avistamientos,lt.size(avistamientos),cmpPlace)
```

Iniciamos con una asignación O(k)

Continuamos con la creación de una lista ordenada O(k)

Ahora pasamos por un while al cual se pasa una cantidad m de veces. Siendo m = |altitud max - latitud min| /0.01

luego sacamos la información de un árbol O(2 logn) pero que se repite m veces por ende es O(2 logn)

posteriormente vamos a un if que se comprobado m veces por ende es O(m*k)

Ingresaremos a este if una cantidad p de veces y por ende para sacar la información de la pareja llave valor se usa O(p*k)

ahora tenemos un for que que pasara por cada avistamiento en esa latitud O(j) y que se repetirá cada vez que entremos al if O(j*p)

en este for hay un if que se repite j*p veces pero que además en su comparación requiere hacer 2 O(2) y cada una de esas encuentra un dato en el elemento dado, por ende es: f(j*p*2*1)

agregamos un elemento al final O(1) pero como se repite j*p veces entonces es O(j*p)

le sumamos a la latitud y redondeamos O(1)

por último organizamos todo con un merge sort que es O(j*plogj*p) dado que el n en el merge es j*p pues es el tamaño de la lista a ordenar.

por ende la complejidad es:
O(j*plogj*p)
6.

```
#REQ 6
latit_def = (latitud_max+latitud_min)/2
long_def = (longitud_max+longitud_min)/2

mapa = folium.Map(location=[latit_def,long_def],zoom_start=4)

for avistamiento in lt.iterator(avistamientos_sorted):
    folium.Marker(location=[float(avistamiento["latitude"]),float(avistamiento['longitude'])]).add_to(mapa)

mapa.save(cf.data_dir +"/UFOS/mapita.html")
webbrowser.open("https://www.youtube.com/watch?v=dQw4w9WgXcQ")
webbrowser.open(cf.data_dir +"/UFOS/mapita.html")
```

Este requerimiento visualiza la respuesta del anterior, por ello decidimos dejar los dos como uno solo. Ahora bien, cómo únicamente se le está sumando un ciclo for de los lugares a marcar en el mapa, entonces la complejidad es de O(j*plogj*p)

Nota: Para todos los ejercicios se asumirán como nulos los aportes a la complejidad de las variables usadas para el análisis de tiempo.