DOCUMENTO DE ANÁLISIS

Est 1 Nicholas Barake 202020664 (n.barake)

Est 2 Jesed Domínguez 202011992 (j.dominguezp)

NOTA: El comentario en verde es la respectiva complejidad de la línea.

REQUERIMIENTO 1

```
def sightings_by_city(catalog,city):
    list_sightings_city= om.get(catalog["cities"],city)["value"] #o(logn)
    merge.sort(list_sightings_city,cmpByDatetime) #o(nlogn)
    primeros3= lt.subList(list_sightings_city,1,3) #o(1)
    ultimos3= lt.subList(list_sightings_city,-2,3) #o(1)

    return om.size(catalog["cities"]),lt.size(list_sightings_city),primeros3,ultimos3
```

Complejidad temporal: O(nlogn), donde n es el tamaño del mapa

La complejidad es O(nlogn), donde n es el número de nodos en el mapa de ciudades, porque al sumar todas las complejidades podemos ver que: logn + nlogn. Si factorizamos tenemos que logn(n+1). Ignorando las constantes, nos queda nlogn. Lo cual tiene sentido porque el algoritmo con mayor complejidad en la función es el mergesort con O(nlogn).

REQUERIMIENTO 2 (Realizado por Nicholas)

```
def sightings_by_duration(catalog,min,max):
    max_duration= om.maxKey(catalog["duration_UFO"]) #o(1)
    total_max= lt.size(me.getValue(om.get(catalog["duration_UFO"],max))) #o(logn)+o(1)+o(1)

    duraciones= om.values(catalog["duration_UFO"],min,max) #o(logn)
    total_max_rango= lt.size(lt.lastElement(duraciones)) #o(1)

    sightings_in_range= lt.newList("ARRAY_LIST")
    for duracion in lt.iterator(duraciones): #o(m)
        merge.sort(duracion,cmpByCity) #o(klogk)
        for sighting in lt.iterator(duracion): #o(k)
            lt.addLast(sightings_in_range,sighting) #o(1)

    total_rango= lt.size(sightings_in_range) #o(1)

    return max_duration, total_max, total_rango, total_max_rango, sightings_in_range
```

Complejidad temporal: O(logn+m*k), donde n es el tamaño del mapa, m de la lista de duraciones, y k de los avistamientos con la misma duración

Se tienen dos complejidades de logn, donde n es el tamaño del mapa de duración, antes de entrar a los ciclos, por lo que se tienen en standby. Luego, tenemos un ciclo que se itera m veces (el número de duración diferentes que hay dentro del rango). Y para cada m, se hace un mergesort de complejidad klogk y otro ciclo de complejidad k, donde k es el número de avistamientos con la misma duración.

Haciendo la matemática tenemos que m*(k+klogk)= m*k+m*klogk, y tomando solo los factores dominantes nos queda que m*k+m*k=2mk. Regresando la primera complejidad, los sumamos, factorizamos, y dejamos el factor dominante: 2logn+2mk= logn+m*k.

REQUERIMIENTO 3 (Realizado por Jesed)

```
def older_hour(orderedmap):
    dates_tree = orderedmap['hour_UFO']
    #Para que imprima la hora más tardía con el número de avistamientos
    older_hour = traversal.inorder(dates_tree) #o(1)
    return older_hour
    #Para listar avistamientos dentro de horas
def hours_in_range(orderedmap, lowhour, highhour):
    dates_tree = orderedmap['hour_UFO']
    hours_sightings = lt.newList('ARRAY_LIST')
    mindate = datetime.datetime.strptime(lowhour, '%H:%M:%S')
    maxdate = datetime.datetime.strptime(highhour, '%H:%M:%S')
    mindate = mindate.time()
    maxdate = maxdate.time()
    lst_range = om.values(dates_tree,mindate,maxdate) #o(logn)
    for i in lt.iterator(lst_range): #o(m)
        merge.sort(i,cmpByDatetime) #o(klogk)
        for j in lt.iterator(i): #o(r)
            lt.addLast(hours_sightings, j) #o(1)
    return hours_sightings
```

Complejidad temporal: O(logn+m*k), donde n es el tamaño del mapa, m la de la lista en el rango, y k cada elemento en esa lista.

Para la primera función, la función de inorder tiene complejidad constante, por lo que se descarta. Se tiene una complejidad de logn para la función de values, donde n es el tamaño del mapa ordenado por horas. Luego como hay un ciclo, se itera m veces un mergesort de complejidad klogk, donde m es el tamaño de la lista del rango, y k es el número de elementos dentro de la lista del rango. Luego se itera r veces los elementos que están en i, pero como la única operación dentro de ese ciclo es constante, se descarta. Sumando todo tenemos que logn+m*k.

REQUERIMIENTO 4

```
def older_sightings(orderedmap):
    dates_tree = orderedmap['datetime_UFO']
    #Para que imprima la fecha más antigua con el número de avistamientos
    older_date = traversal.inorder(dates_tree) #o(1)
    return older_date
    #Para listar avistamientos dentro de fechas
def dates_in_range(orderedmap, lowdate, highdate):
    dates_tree = orderedmap['datetime_UFO']
    mindate = datetime.datetime.strptime(lowdate, '%Y-%m-%d')
    maxdate = datetime.datetime.strptime(highdate, '%Y-%m-%d')
    mindate = mindate.date()
    maxdate = maxdate.date()
    lst_range = om.values(dates_tree,mindate,maxdate) #o(logn)
    sub_dates = lt.subList(lst_range,0,lt.size(lst_range)+1) #o(1)
    Primeros = lt.subList(lst_range,1,3) #o(1)
    Ultimos = lt.newList('ARRAY_LIST') #o(1)
    j = 0
    while j < 3: #o(3)
        last = lt.removeLast(sub_dates) #o(1)
        lt.addLast(Ultimos, last) #o(1)
        j += 1
    Ultimos = merge.sort(Ultimos, cmpDatetolst) #o(mlogm)
    return lst_range,Primeros,Ultimos
```

Complejidad temporal: O(logn), donde n es el tamaño del mapa de fechas.

La primera complejidad temporal distinta a constante es con el values, que es logn, donde n es el tamaño de la lista generada según el rango ingresado. Luego, hay un ciclo en forma de while que se itera 3 veces, pero como todas operaciones dentro de la estructura tienen complejidades constantes, se descarta. Finalmente hay un mergesort de complejidad mlogm donde m es el tamaño de la lista de los últimos tres datos. Pero como la lista únicamente por adelantado se sabe que tiene tres elementos, se puede descartar de la complejidad por ser constante. Por lo que se concluye que la complejidad de todo es: O(logn).

REQUERIMIENTO 5

```
def sightings_by_zone(catalog,min_long,max_long,min_lat,max_lat):
    data_tree = catalog["longitudes"]
    mapas_latitud_en_rango = om.values(data_tree,min_long,max_long) #o(logn)
    final_range_lst = lt.newList('ARRAY_LIST')
    for data1 in lt.iterator(mapas_latitud_en_rango): #o(m)
        for data2 in lt.iterator(data1): #o(k)
            cmpdata = round(float(data2['latitude']),2)
            if cmpdata >= min_lat and cmpdata <= max_lat:
                lt.addLast(final_range_lst, data2) #o(1)
    final_range_lst = merge.sort(final_range_lst, cmpByDatetime) #o(r)
    sightings_size = size_in_range(final_range_lst) #o(1)
    sub_dates = lt.subList(final_range_lst,0,lt.size(final_range_lst)+1) #o(1)
    Primeros = lt.subList(final_range_lst,1,5) #o(1)
    Ultimos = lt.newList('ARRAY_LIST')
    j = 0
    while j < 5: #(5)
        last = lt.removeLast(sub_dates) #o(1)
        lt.addLast(Ultimos, last) #o(1)
        j += 1
    Ultimos = merge.sort(Ultimos, cmpByDatetime) #o(hlogh)
    return final_range_lst, sightings_size, Primeros, Ultimos
```

Complejidad temporal: O(logn+m*k), donde n es el tamaño del mapa de longitudes, m es la lista dentro del rango, y k es cada uno de sus elementos

La primera complejidad está en el values, que crea una lista con las longitudes del rango, y es logn, donde n es el tamaño del mapa de longitudes. Luego se itera esa lista, y otra vez para llegar los avistamientos. Esto agrega complejidades de m y k, donde m es el tamaño de la primera lista dentro del rango, y k es cada uno de los elementos dentro de ella. Este doble ciclo genera uan complejidad de m*k, luego hay un mergesort de complejidad r, donde r es el tamaño de la lista final de los avistamientos dentro del área. Finalmente, hay un while con operaciones constantes, por lo que se descarta y un mergesort con tamaño constante, por lo que también se descarta. La complejidad sumada es: O(logn+m*k).