Documento análisis reto 3

Hecho por: Daniel Gomez y Jenifer Arce

Contenido

Observaciones generales

- Link repositorio reto github: Reto3-G02
- Usamos la librería Prettytable() para mostrar resultados en el view
- Utilizamos un ordenamiento de selection editado para el reto.
    - Selection.sortEdit es un ordenamiento con selection, pero con unas modificaciones para el reto. Lo que hace este algoritmo editado es ordenar las posiciones finales e iniciales que le digamos por parámetro. Como solamente necesitamos mostrar los 3 primeros y últimos artistas/obras en la mayoría de los requerimientos, solo queremos que estas posiciones quedan ordenadas. Por lo cual, al hacer este ordenamiento con selection significa que la lista se recorrerá 6 veces, es decir que la complejidad será O(6N) = O(N). La finalidad de esta modificación era reducir la complejidad de todos los requerimientos del reto
- Para todos los requerimientos al momento de estar creando los árboles RBT guardamos solamente las posiciones de los avistamientos. Esto para evitar duplicar la información.
- En las complejidades escribimos logN, pero este logN sería en base 2 dado que estamos usando árboles binarios
- Máquinas en las que hicimos pruebas:

	Máquina 1	**Máquina 2**
Procesadores	Intel(R) Core(TM) i5-8265U CPU @ 1.60GHz 1.80 GHz	Intel(R) Core(TM) i7-1185G7 CPU @ 3GHz
Memoria RAM (GB)	8 GB	16 GB
Nombre del SO	Windows 10 64-bits	Windows 10 64-bits

Requerimiento 1 – Grupal

Pruebas de tiempo de ejecución

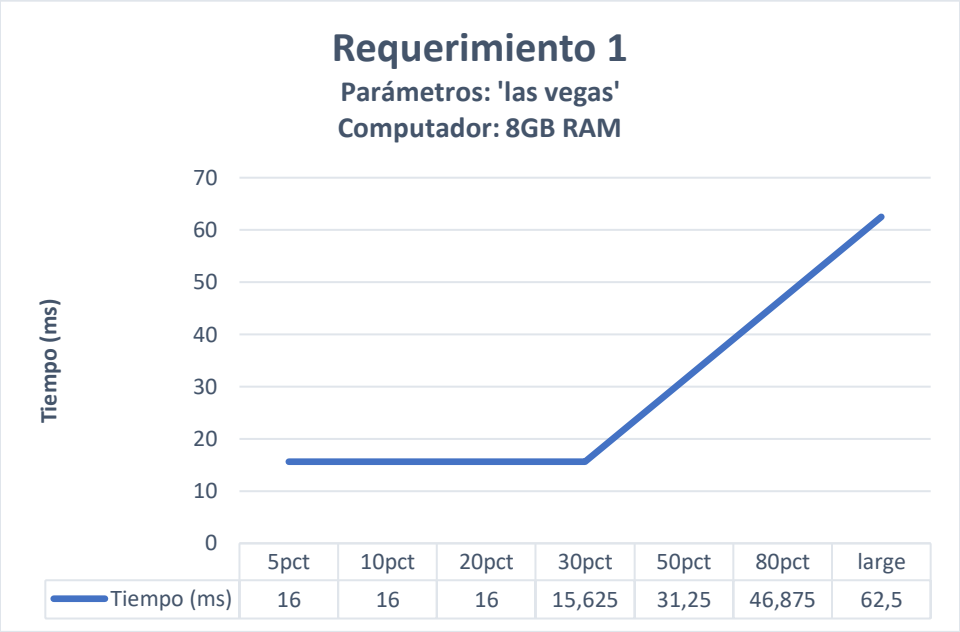


Ilustración 1 Tiempos de ejecución requerimiento 1 – Computador 8gb RAM

Análisis de complejidad

1. Total de ciudades con avistamientos

```
def avistamientosPorCiudad(catalog,ciudad): # Requerimiento Grupal 1
    numeroCiudades=om.size(catalog["cityIndex"])
```

Para saber el total de ciudades de avistamientos usamos la función om.size() en el árbol RBT creado para esta función. Al analizar la librería de DISClib podemos notar que esta función tiene una complejidad de O(1) dado que la raíz del árbol va a contener el tamaño total del árbol.

Complejidad:

O(1)

2. Obtener la información ingresada por parámetro

```
    ufos=om.get(catalog["cityIndex"],ciudad)["value"]
    listaAvistamiento=lt.newList("ARRAY_LIST")
    for indice in lt.iterator(ufos):
        lt.addLast(listaAvistamiento,lt.getElement(catalog["ufos"],indice))
    listaAvistamiento=sortList(listaAvistamiento,compareUFObyDate,1)
    return listaAvistamiento, numeroCiudades
```

Para obtener los avistamientos de una ciudad en específico utilizamos la función om.get(). Como es un árbol RBT la complejidad en el peor caso será O(logN) dado que es un árbol balanceado.

Después de esto hacemos un recorrido sobre los avistamientos para obtener su información, puesto que los elementos que están en ufos serán las posiciones de los avistamientos, por lo cual tenemos que relacionar cada elemento con la lista de UFOs del catálogo. Es así como la complejidad de este recorrido será O(*N*).

Por último, para ordenar de manera cronológica los 3 primeros y últimos avistamientos utilizamos la función de selection sort editado. Por lo cual este ordenamiento tiene una complejidad de 6*O(*N*) = O(*N*).

La complejidad será:

$$O(\log N) + O(N) + O(N)$$

$$O(\log N + N + N)$$

$$O(N)$$

‘* Donde N será el número de avistamientos en la ciudad ingresada por el usuario.

La complejidad final del requerimiento 1 es:

$$O(1) + O(N)$$

$$O(N)$$

Requerimiento 2 – Daniel

Pruebas de tiempo de ejecución

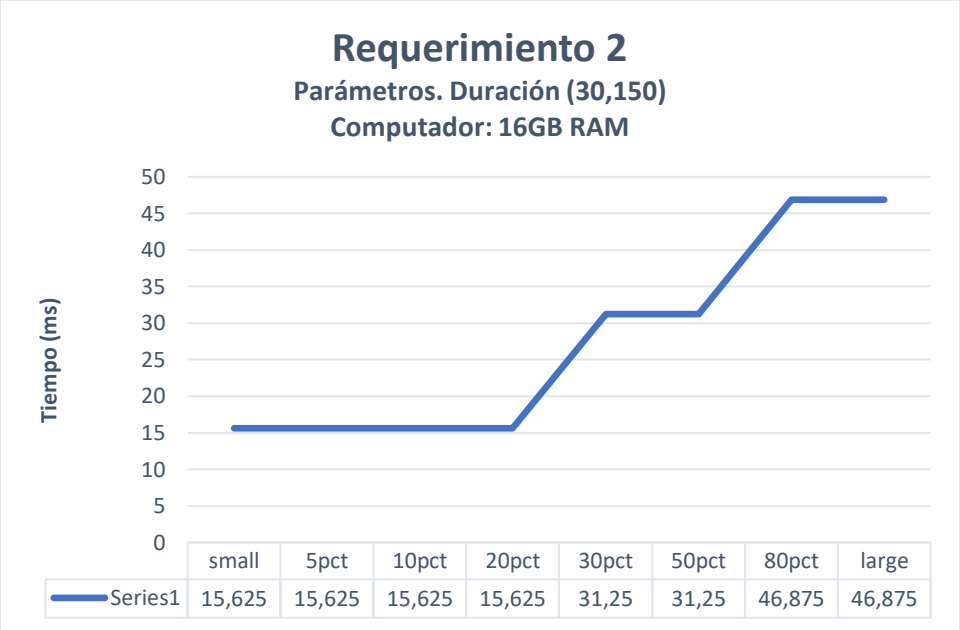


Ilustración 2. Gráfica de tiempo de ejecución para requerimiento 2

Análisis de complejidad

1. Obtención de valores del mapa total de duraciones

```
listaAvistamientos = lt.newList("ARRAY_LIST")
    numeroDuraciones=om.size(catalog["durationIndex"])
    mayorDuracion=om.maxKey(catalog["durationIndex"])
    arbolMayorDuracion=om.get(catalog["durationIndex"],mayorDuracion)["value"]
```

```
cantMayorDuracion=0

    for listaMayor in lt.iterator(om.valueSet(arbolMayorDuracion)):
        cantMayorDuracion+=lt.size(listaMayor)
```

Se obtiene la llave con valor más grande maxValue:

$$O(\log N)$$

Se obtiene el número de duraciones:

$O(k)$[1]

Para obtener el número total de avistamientos toca obtener el árbol de la llave más grande (caso promedio $O(\log N), peor\sim caso: O(N)$) y luego recorrer cada árbol $O(N_a)$, ya que el árbol principal se compone de otros árboles llave ciudad-país. Ahora bien, en la mayoría de los casos cada combinación de duración-ciudad-país tendrá un caso o máximo dos por lo que el recorrido en list se reduce a $O(k)$. Por lo que en el primer caso el caso promedio será.

$$O(\log\{N)) + O(\log N) + O(k) + O(k) \rightarrow O(\log N)$$

2. Obtener avistamientos en un rango de duración (segundos) con segundo criterio de ordenamiento (ciudad-país).

```
for duracion in
lt.iterator(om.values(catalog["durationIndex"],segundos_min,segundos_max)):
        for cc in lt.iterator(om.valueSet(duracion)):
            for indice in lt.iterator(cc):
                lt.addLast(listaAvistamientos, lt.getElement(catalog["ufos"],indice))
```

En la obtención de la lista ordenada no se hace ningún ordenamiento, dado que el RBT entrega los valores ordenados. Ahora bien, como la estructura de datos se compone de un árbol mayor que tiene duraciones y dentro de cada nodo del árbol otro árbol con la llave heterogénea de ciudad-país, al aplicar una vez values y una vez value set se obtiene la lista ordenada por los dos criterios con prioridad de ordenamiento a duraciones.

Se obtiene primero values (complejidad promedio $O(\log N)$ peor caso O(N)).

Luego se hace un valueSet pero sobre otro tamaño de datos (caso promedio O(k), peor caso $O(N_3)$) y con eso se llena la lista.

El ultimo paso es un recorrido en la lista de duración-ciudad-país que será de complejidad promedió O(k)

O(log(N))+O(k)+O(k) -> O(log N)

La complejidad final del requerimiento 2 es:

$O(\log N)$

[1] La implementación en DISCLib nos indica que root tiene una llave con el valor total de nodos que hay, por lo que obtener el om.size() de un RBT es de complejidad O(k)

Y en el peor caso será una multiplicación de O(N) de diferente tamaño. Como los tamaños son diametralmente distintos se obtiene una complejidad de O(N) siendo N el número de UFOS.

Requerimiento 3 – Jenifer

Pruebas de tiempos de ejecución:

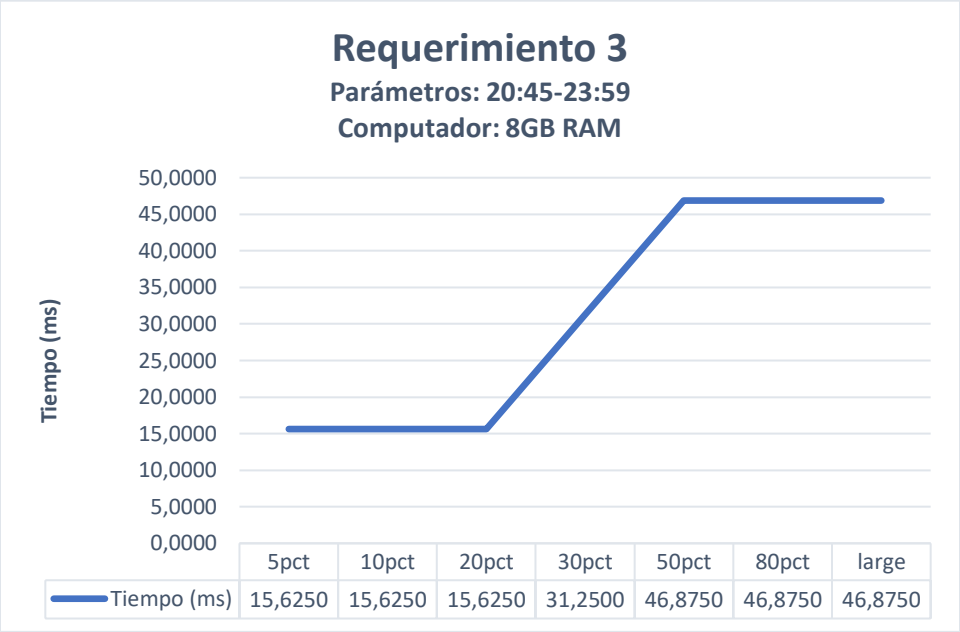


Ilustración 3 Tiempos de ejecución requerimiento 3 – Computador 8gb RAM

Análisis de complejidad:

3. Obtener la hora máxima

```
#Hora máxima
horaMaxima=om.maxKey(catalog["hourIndex"])
ValueUltimaHora=om.get(catalog["hourIndex"],horaMaxima)

UltimaHora=lt.newList("ARRAY_LIST")
lt.addLast(UltimaHora,{"hour":horaMaxima.strftime('%H:%M'),
                       "count":ValueUltimaHora["value"]["size"]})
```

La complejidad de la operación om. maxKey() teniendo un árbol RBT que es balanceado, será equivalente a la altura del árbol *h*, dado que el recorrido será lo más a la derecha posible. En términos de Big O será O(logN).

om.get[2] hará el mismo recorrido que maxKey, por lo tanto, la complejidad será equivalente a O(logN)

Después de esto se realiza una lista para añadir la key máxima y su tamaño, con el objetivo de obtener una vista organizada en el view. La complejidad es *O(K)*

Por ende, la complejidad es:

$$O(logN) + O(logN) + O(K)$$

$$O\ (2logN + k)$$

$$O(logN)$$

4. Obtener avistamientos de UFOs dentro de un rango de tiempo

```
#Avistamientos dentro del rango de horas usuario
horaInicialTime=datetime.datetime.strptime(horaInicial,'%H:%M').time()
horaFinalTime=datetime.datetime.strptime(horaFinal,'%H:%M').time()
rangoHoras=om.values(catalog["hourIndex"],horaInicialTime,horaFinalTime)
listaOrdenadaAvistamientos=ListasRespuesta(catalog,
                           rangoHoras,requerimiento="req3")
listaRespuesta=listaOrdenadaAvistamientos[0]
contadorAvistamientos=listaOrdenadaAvistamientos[1]
```

La función om.values() obtendrá los valores del árbol RBT de horas que estén dentro del rango deseado. Al analizar esta función en la librería de DISClib se puede notar que la complejidad de esta función dependerá del rango de las keys(). Por lo cual, el peor caso de complejidad será O(N) que representará la situación en donde la hora mínima sea 00:00 y la hora máxima sea 23:59, con estos parámetros se tendrá que recorrer a todo el árbol para obtener los valores dentro del rango. El mejor caso será cuando la diferencia entre la hora mínima y máxima no sea tanta, además que el rango esté lo más cerca de la raíz de este árbol, con esto la búsqueda sería O(logN).

[2] La operación de om.get no sería necesaria si maxKey() devolviera todo el nodo, pero esta retorna nodo[“key”]. En el momento de hacer el requerimiento pensé en editar la librería de **DISClib** para que la función de maxKey(), del archivo de bst, devolviera todo el nodo, es decir, que retornará node solamente en la línea 245, esta modificación funcionó, pero decidí no hacer cambios en esta librería debido a las restricciones que me explicaron por discord. En el caso que maxKey() devolviera todo el rango la complejidad sería de O(h+k), donde h representa la altura del árbol.

Complejidad en peor caso om.values:

$$O(N_1)$$

5. Funciones complementarias (listaOrdenadaAvistamientos y selectionPlace():

```
def ListasRespuesta(catalog,tabla,requerimiento): #req 3-4 -
    size=tabla["size"]
    i=1
    ufosPosCadaElemento=tabla
    numeroAvistamientos=0

    #Se obtiene una array lista con los 3 primeros y 3 últimos, además se cuentan los
avistamientos
    if requerimiento=="req3" or requerimiento=="req4":
        ufosPosCadaElemento=lt.newList("ARRAY_LIST")
        for key in lt.iterator(tabla):
            ## ……. más líneas de código
    recorrer=True
    pos=1
    lista_respuesta=lt.newList("ARRAY_LIST")
    while recorrer and lista_respuesta["size"]<=6:
        key_actual=lt.getElement(ufosPosCadaElemento,pos)
        lista_en_pos=key_actual
        #lista_en_pos=om.get(catalog["dateIndex"],key_actual)["value"] #Se obtiene la
lista correspondiente a esa pos
        pos_j=1

        condiciones_elementos= True
        while condiciones_elementos and pos_j<=lista_en_pos["size"]: #Se detendrá el
recorrido de cada key

            pos_UFOlist=lt.getElement(lista_en_pos,pos_j)
            if requerimiento=="req4":
                elemento=lt.getElement(catalog["ufos"],pos_UFOlist) #elemento que se
agrega en la lista de respuesta
            elif requerimiento=="req3":
                if pos<=3:
                    cmp=compareDateHourMin
                else:
                    cmp=compareDateHourMax
                elementoPos=selectionPlace(lista_en_pos,cmp)
                #print(f'**********ElementoPos{elementoPos}')
                elemento=lt.getElement(catalog["ufos"],elementoPos["pos"])
            #elemento=lt.getElement(catalog["ufos"],pos_UFOlist) #elemento que se agrega
en la lista de respuesta
            lt.addLast(lista_respuesta,elemento)
            pos_j+=1
### más líneas de código …..

        if lista_respuesta["size"]>=6:
            recorrer=False
###más línead de código    #SE ORDENAN LOS ÚLTIMOS 3 ELEMENTOS
    sortList(lista_respuesta,compareUFObyDate,sortType=1,
             ordenarInicio=False,ordenarFinal=True)
    return lista_respuesta,numeroAvistamientos
```

El uso de esta función es obtener los 3 primeros y 3 últimos avistamientos de acuerdo con el rango de horas, así mismo, esta función contará el número de avistamientos dentro de la lista de valores. La complejidad para contar el número de avistamientos y obtener las 3 primeras y 3 últimas horas será $O(N_1)$, dado que recorre toda la lista de valores de horas (resultado om.values), que en el mayor caso este N será equivalente a 1390 elementos (horas), (ver anexo análisis datos req3).

Ahora bien, para obtener los 3 primeros elementos ordenados se recorre esta lista de 3 primera y 3 últimas horas, y de ahí se comienzan a organizar los avistamientos de acuerdo con su orden cronológico. Como estos elementos son listas se tuvo que hacer un ordenamiento cronológico internamente dependiendo el caso. Para este

ordenamiento interno se hizo una función extra llamada selectionPlace(), lo que hará esta función es tomar el máximo/mínimo elemento de una lista, intercambiarlo con la última posición, y después de eso se elimina la última posición y se retornará este elemento. La complejidad de selectionPlace[3]() será siempre $O(N_2)$, donde N_2 será el tamaño de la lista que esté analizando, donde esta puede contener hasta 4617 avistamientos si la hora es 22:00 o solo un avistamiento si la hora es 11:27.

```
def selectionPlace(lst,cmpfunction): #Funcion complementaria req 3
    size = lt.size(lst)
    pos1 = 1
    minimum = pos1    # minimun tiene el menor elemento
    pos2 = pos1 + 1
    while (pos2 <= size):
        if (cmpfunction(lt.getElement(lst, pos2),
           (lt.getElement(lst, minimum)))):
            minimum = pos2  # minimum = posición elemento más pequeño
        pos2 += 1
    lt.exchange(lst, size, minimum)  # elemento más pequeño -> elem
pos1
    minimo=lt.getElement(lst, size)
    lt.removeLast(lst)
    return minimo
```

Finalmente, se ordenan los 3 últimos elementos de la lista de respuesta (que contiene 6 elementos en total.

Por lo tanto, la complejidad de listaRespuesta() para el requerimiento [4]3será:

$$O(N_1) + 6O(N_2) + O(6)$$

$$O(N_1+6N_2+6)$$

$$O(N_2)$$

La complejidad final del requerimiento 3 es:

$$O(\log N) + O(N_1) + O(N_2)$$

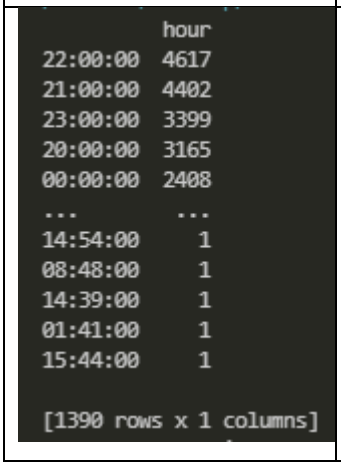
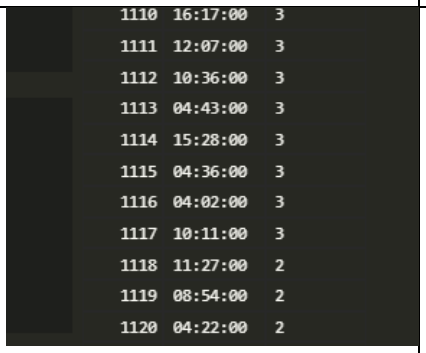
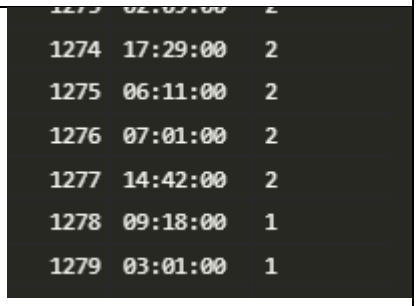
$$O(N)$$

[3] Esta función es basada en el ordenamiento de selection de la disclib

[4] $O(N_2)$ dado que esta es la lista que puede tener mayor cantidad de elementos (4617 avistamientos en el peor caso). $O(N_1)$ como se mencionó anteriormente, en el peor caso el árbol tiene 1390 horas (HH:MM).

Observación análisis de complejidad:

La complejidad de la función se puede ver reducida si se editan algunas funciones de la DISClib, en específico las funciones de om.values(), valuesRange() en árboles RBT. Al ser una función recursiva se puede agregar un contador que vaya sumando los tamaños de las listas que están como value en cada nodo que cumpla con la condición de ((complo <= 0) and (comphi >= 0)). Adicionalmente, si el tipo de TAD es una Array List en vez de una single linked se pueden acceder de una manera más eficiente a las 3 primera y 3 últimas posiciones para tener un resultado en el view.

Análisis datos – Req 3		
Hay 1390 horas distintas	El 83% de las horas tienen una ocurrencia de más de 3 veces	El 11% de las horas tienen una ocurrencia de 2 veces
		

Requerimiento 4 – Grupal

Pruebas de tiempo de ejecución

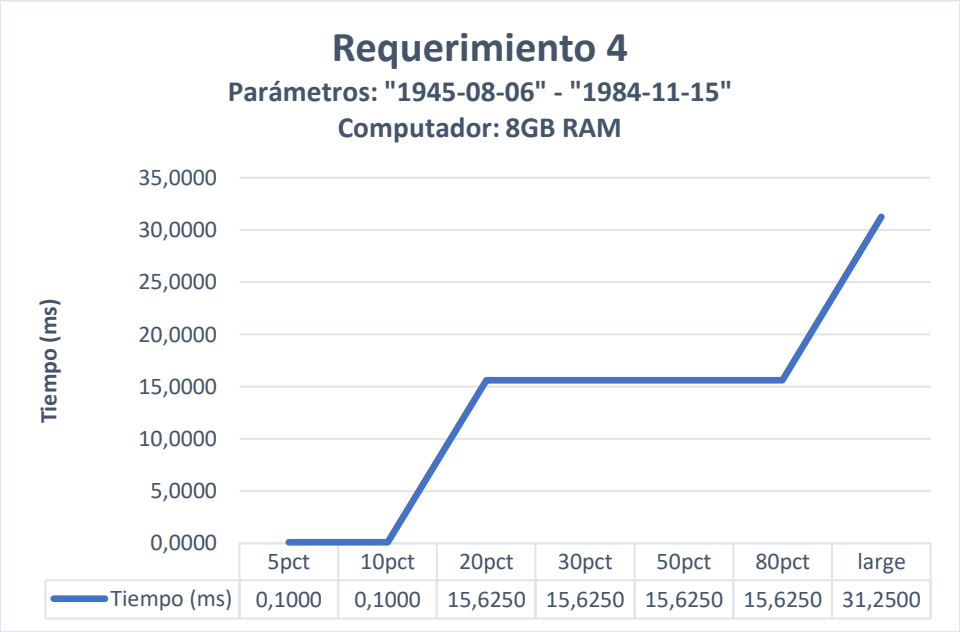


Ilustración 4Tiempos de ejecución requerimiento 4 – Computador 8gb RAM

Análisis de complejidad

1. Obtener la fecha mínima de avistamiento

```
minkey=om.minKey(catalog["dateIndex"])
infoMinKey=om.get(catalog["dateIndex"],minkey)["value"]
respuestaUltimasFechas=lt.newList("ARRAY_LIST")
lt.addLast(respuestaUltimasFechas,{"date":minkey.strftime('%Y-%m-%d'),
                                   "count":infoMinKey["size"]})
```

La complejidad de la función minKey() será O(logN), donde N es el tamaño del árbol, o equivalentemente, la complejidad será la altura del árbol, dado que se tendrá que recorrer al lado más izquierdo del árbol.

Para obtener los avistamientos de la menor fecha se usa om.get(), esto tiene una complejidad O(*logN*) dado que el árbol es balanceado.

Después de esto se crea una lista para mostrar la respuesta en el view, lo cual tiene una complejidad O*(K)*

Por ende, la complejidad es:

$$O(logN) + O(logN) + O(K)$$

$O\ (2logN + k)$

$O(logN)$

2. Avistamientos dentro del rango de fechas dado por el usuario

```
#Fechas dentro del rango brindado por el usuario
avistamientoRango=om.values(catalog["dateIndex"],date1,date2)
diasAvistamientos=avistamientoRango["size"] #Días distintos UFOs
contadorYLista=ListasRespuesta(catalog,avistamientoRango,"req4")
listaRespuestaView=contadorYLista[0] #3 primeros y últimos UFOs
contadorAvistamientos=contadorYLista[1] #Contador total UFOs
```

Para obtener los avistamientos dentro del rango de fechas se utilizó om.values, el peor cao de complejidad será O(*N*), esto pasará cuando el rango de fechas sea la fecha mínima, hasta la fecha máxima, en este caso se tendrá que recorrer todo el árbol. En el mejor caso será O(logN), esto ocurrirá cuando el rango de fechas esté alineado sobre una rama del árbol.

Para conocer el número de avistamientos dentro del rango de fechas, y conseguir los 3 primeros y últimos avistamientos en orden crónologico, se utilizó la función de listasRespuesta(). En este requerimiento la función recorrerá toda la lista del resultado de om.values(), acá va ir llevando un contador del total de avistamientos de acuerdo con el size de cada elementa, esto tiene una complejidad O(*N*), al mismo tiempo agregará la lista con avistamientos de una fecha si su posición es (1,2,3,size-2,size-1,size). Después de esto se recorre esta lista de 6 elementos y se sacan los 3 primeros y 3 últimos avistamientos, la complejidad de esto será siempre O(6).

Por lo cual, la complejidad de listasRespuesta para este requerimiento es:

$O(N) + O(6)$

$O(N)$

**Donde N será el número de fechas del resultado de om.values()*

La complejidad final del requerimiento 4 es:

$O(logN) + O(N_1) + O(N_2)$

$O(N)$

*N_1 representa el peor caso del recorrido de om.values(), N_2 es el resultado de om.values

Requerimiento 5 – Grupal

Pruebas de tiempo de ejecución

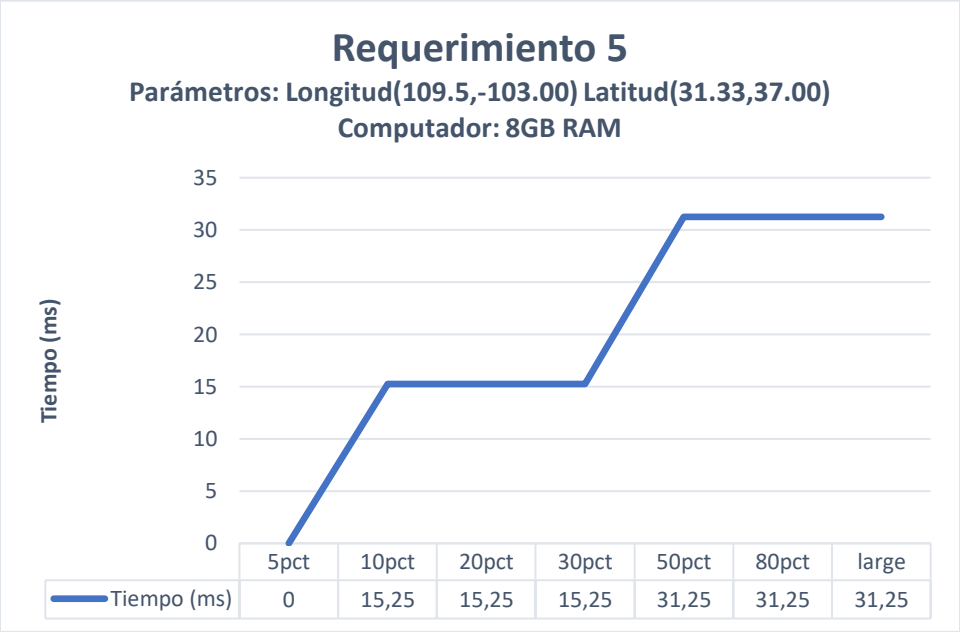


Ilustración 5. Tiempos de ejecución requerimiento 5 – Computador 8gb RAM

Análisis de complejidad

1. Comprobar rangos de longitudes y latitudes (mínimos y máximos)

```
long_min=round(float(long_min),2)
long_max=round(float(long_max),2)
lat_min=round(float(lat_min),2)
lat_max=round(float(lat_max),2)

if(long_max<long_min):
    long_temp=long_min
    long_min=long_max
    long_max=long_temp

if(lat_max<lat_min):
    lat_temp=lat_min
    lat_min=lat_max
    lat_max=lat_temp

lat_max+=0.01
long_max+=0.01
```

Para evitar errores en los rangos de resultados hicimos una comprobación de los valores de las latitudes y longitudes. Así mismo, adicionamos a cada máximo de latitudes y

longitudes +0.01, esto para incluir a todas las latitudes y longitudes que estuvieran antes de ese límite.

Complejidad: O(*K*)

2. Extraer las coordenadas que estén dentro del rango de latitudes y longitudes

```
avistamientos=lt.newList("ARRAY_LIST")

    for arbolLat in lt.iterator(om.values(catalog["longitudIndex"],Long_min,Long_max)):
        for lat in lt.iterator(om.values(arbolLat,Lat_min,Lat_max)):
            for index in lt.iterator(lat):
                elemento=lt.getElement(catalog["ufos"],index)
                lt.addLast(avistamientos,elemento)

    return avistamientos
```

Observación: Para este requerimiento construimos un árbol RBT que tiene como keys las longitudes de los avistamientos. El valor de cada key será otro árbol RBT que tiene como keys las latitudes. Construimos este árbol de esta manera para lograr mejores complejidades en las búsquedas debido a las ventajas que ofrecen los RBT. El ordenamiento de la respuesta del view está dado por la longitud (de mayor a menor).

Notaciones[5]:

N_1 = Tamaño del árbol de longitudes del catálogo

N_2 = Tamaño del árbol de latitudes de cada una de las posiciones del resultado de om.values() del rango de longitudes

$N_{longitudes}$ = Longitudes que están dentro del rango (size del resultado de om.values de longitudes)

N_3= Avistamientos por cada coordenada que está dentro del rango de latitudes y longitudes

$N_{coordenadas}$: Número de coordenadas que están dentro del rango de latitud y longitud

El recorrido más externo obtendrá los valores que estén dentro del rango de longitudes, para esta parte usamos **om.values**, que en el peor caso tiene una complejidad de $O(N_1)$; en este árbol el peor caso sería que los valores de entrada fueran el mínimo de longitud del

[5] Hicimos una diferenciación en el tamaño de los datos (N) dado que dependiendo del recorrido y la posición el tamaño de la lista será distinto.

árbol y el máximo de longitud del árbol. El mejor caso sería que el rango estuviera sobre una rama estrictamente, lo cual sería $O(\log N_1)$. Para este requerimiento, debido a que estamos analizando una zona geográfica tomamos como complejidad de om.values() su mejor tiempo, que es $O(\log N_1)$.

Después de extraer las longitudes que estén dentro del rango pasamos a recorrer cada uno de estos resultados de longitudes para extraer las latitudes. Nuevamente, utilizamos om.values() para extraer las latitudes que estén dentro del rango dado por el usuario. La complejidad por cada posición de la lista que estamos recorriendo será en promedio $O(\log N_2)$, como explicamos anteriormente, tomamos el mejor caso de om.values() ya que se quiere analizar una zona geográfica en específico.

Por último, hacemos un recorrido de los avistamientos de los UFOs que están dentro del rango deseado para agregarlo a la lista de respuesta. Esto será $O(N_3)$ por cada posición de la latitud, donde N_3 representará los avistamientos vistos en una coordenada exacta.

Complejidad:

$$O(\log N_1) + N_{longitudes} * (O(\log N_2)) + N_{coordenadas} * O(N_3)$$

$$N_{longitudes} * (O(\log N_2))$$

*debido a que son las listas más grandes, ver anexo análisis datos

La complejidad del requerimiento 5 es:

$$N_{longitudes} * (O(\log N_2))$$

Análisis datos req5 y req6:

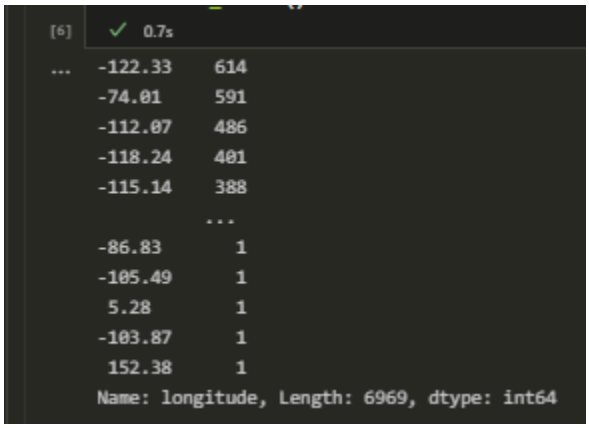


Ilustración 6 Hay 6969 longitudes iguales, la que más se repite tiene una ocurrencia de 614 avistamientos

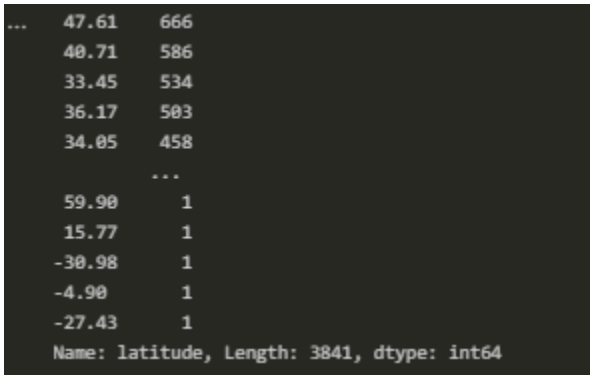


Ilustración 7. Hay 3841 latitudes iguales

Requerimiento 6 – Grupal

Explicación de como mostrar el mapa

Para mostrar el mapa se tiene que ejecutar el archivo desde la ventana interactiva de VS Code. Escogimos esta opción para mostrar el mapa para aprovechar al máximo este IDE. Adicionalmente, de esta manera no se genera ningún archivo extra (HTML) que se tenga que visualizar en otra aplicación.

Pasos:

1. Dar click izquierdo sobre el archivo de view, después dar run current file in interactive window.

Ilustración 8. Click izquierdo sobre el archivo de view > Run Current File in Interactive Window

2. Después de que cargue la ventana interactiva (Jupyter) hay que seleccionar las opciones deseadas.

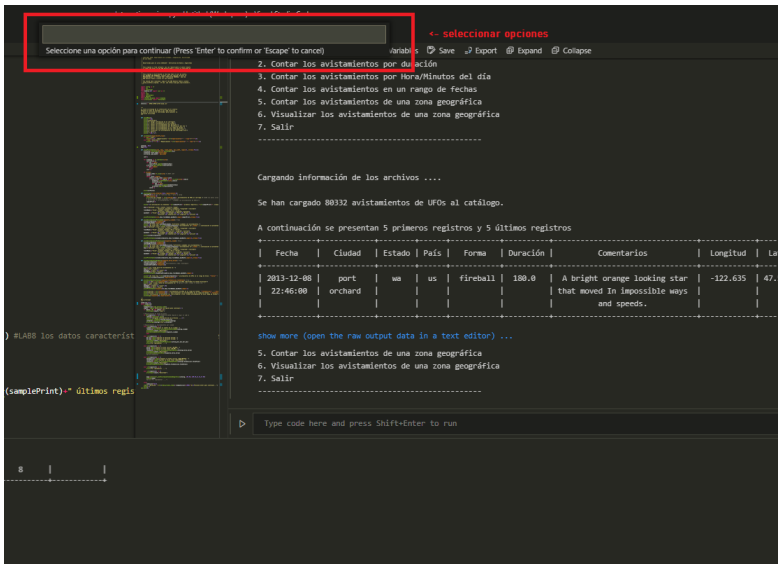


Ilustración 9 Interactuar con la barra superior para seleccionar las opciones deseadas. (Primero se debe cargar el catálogo)

3. Al seleccionar la opción 6 y digitar los valores correspondientes a altura y latitud, aparecerá el mapa

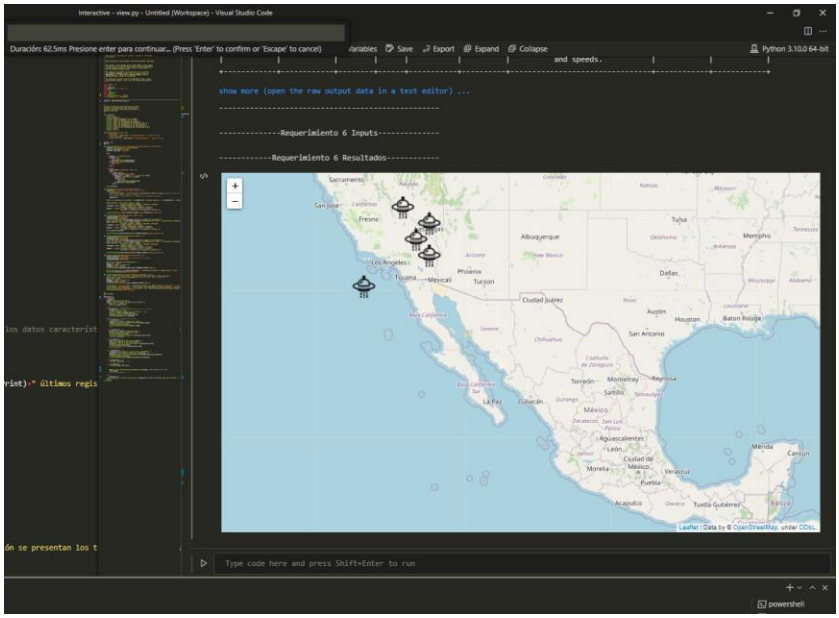


Ilustración 10. Seleccionar opción 6 y escribir los valores de longitud y latitud que se desee. Después de esto se muestra el mapa.

Grabación pantalla mostrar mapa:

En el siguiente vídeo hay una grabación de pantalla de como mostrar el mapa en el view: Link One Drive Grabación de Pantalla

Pruebas de tiempo de ejecución

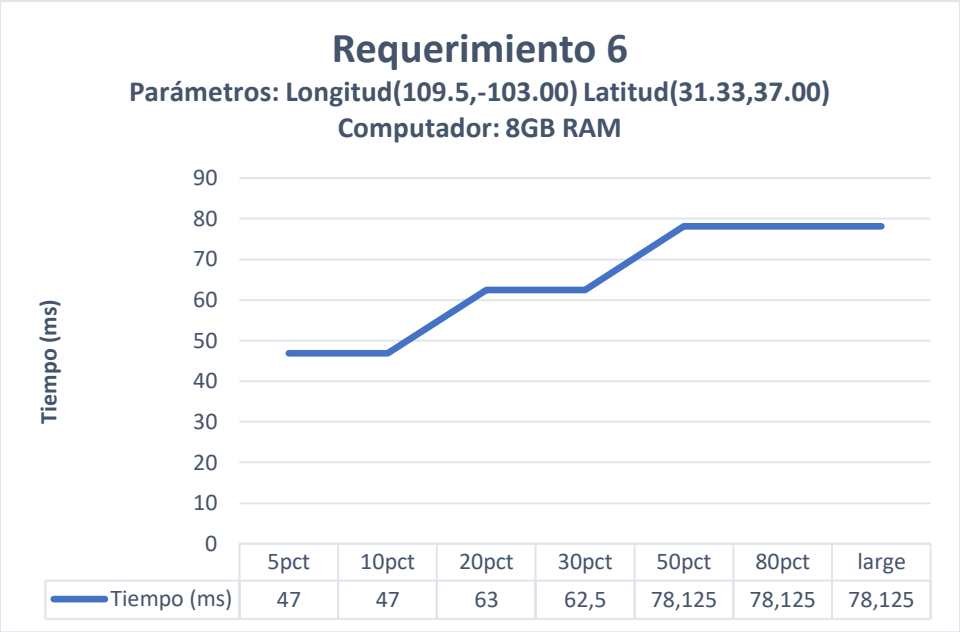


Ilustración 11.Tiempos de ejecución requerimiento 6 – Computador 8gb RAM

Análisis de complejidad

1. Llamar al resultado del requerimiento 5 y generar una lista con 10 avistamientos
`` if avistamientosCargados==None: avistamientosArea=contarAvistamientosZonaGeografica(catalog, long_min,long_max,lat_min,lat_max) else: avistamientosArea=avistamientosCargados listaCoordenadas=listaReq6(avistamientosArea) ``` La complejidad de esta operación será $N_{longitudes} * (O(logN_2))$

En caso de que el resultado ya esté cargado (es decir, que el usuario acepté visualizar el mapa después de haber utilizado el req5) la complejidad sería O(1). Después de esto, se llama a una función para obtener los 5 primeros y últimos avistamientos en esta área, lo cual tiene una complejidad de O(10).

La complejidad por lo tanto será:

$$O(K)$$

2. Generar una tabla de símbolos para las coordenadas de la lista

Al hacer esta función notamos que si habían 2 o más avistamientos en una coordenada, la visualización no sería la más óptima, dado que solo se veía el avistamiento más superficial.

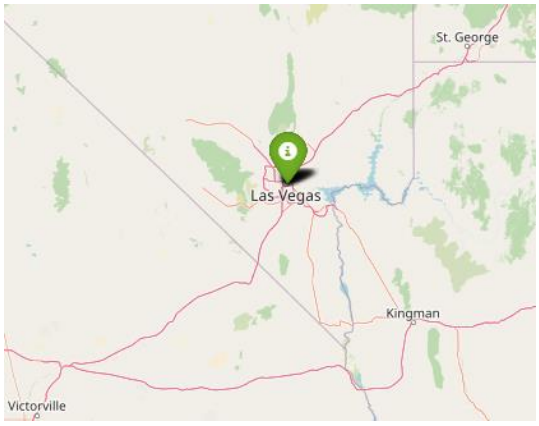


Ilustración 12. Varios avistamientos en una misma locación

Por lo tanto, se creó una tabla tipo linear probing, con tamaño de entrada de 11 elementos. De esta manera podíamos agrupar los avistamientos que tuvieran una misma coordenada.

```
mapProbing=mp.newMap(numelements=11,maptype="PROBING")

for avistamientoUFO in lt.iterator(listaCoordenadas):
        latitud=avistamientoUFO["latitude"]
        longitud=avistamientoUFO["longitude"]
        tuplaCoordenada=(latitud,longitud)
        existsCoordenada=mp.contains(mapProbing,tuplaCoordenada)
        if existsCoordenada:
            coordenada=mp.get(mapProbing,tuplaCoordenada)
            coordenadaInMap=me.getValue(coordenada)
        else:
            coordenadaInMap=lt.newList("ARRAY_LIST")
            mp.put(mapProbing,tuplaCoordenada,coordenadaInMap)
        lt.addLast(coordenadaInMap,avistamientoUFO)


coordenadasAgrupadas=mp.valueSet(mapProbing)
```

La complejidad de crear la tabla de símbolos será:

$$O(numelements*factor\ de\ carga) = O(2N) = O(N)$$

Para recorrer los 10 avistamientos e ingresar estos avistamientos a la tabla de símbolos será:

$$O(10) = O(K)$$ *suponiendo que las colisiones son mínimas

Por último, para agrupar las coordenadas se tiene que recorrer toda la tabla de símbolos. Lo cual tiene una complejidad de.

$$O(N)$$ *donde N es el tamaño de la tabla*

Es así como la complejidad será:

$$O(N) + O(N) + O(K)$$

$$O(2N + K)$$

$$O(N)$$

3. Crear el mapa de folium y añadir los marcadores

```
    media_longitud=(float(long_min)+float(long_max))/2
    media_latitud=(float(lat_min)+float(lat_max))/2
    mapgraf=folium.Map(location=[media_latitud,media_longitud],zoom_start=6) #Se
crea el mapa
    for avistamientoTupla in lt.iterator(coordenadasAgrupadas):
        nAvistamiento="Avistamientos:"
        qAvistamientos=0
        avistamientosEnLocacion=""
        for UFO in lt.iterator(avistamientoTupla):
            latitud=UFO["latitude"]
            longitud=UFO["longitude"]
            fecha=UFO["datetime"]
            nAvistamiento+=" #" +str(n)
            duracion=UFO["duration (seconds)"]
            forma= UFO["shape"]
            ciudadPais=UFO["city"] +", " + UFO["country"]

            infoPorAvistamiento=str("<br><b> Avistamiento: #"+str(n)+ "</b>"
                                    +"<br><b>Fecha y hora:    </b>"+fecha
                                    +" <br><b>Duración: </b>" +duracion
                                    + "<br></b>"+"<b>Forma: </b>" +forma
                                    + "<br> ")

            avistamientosEnLocacion+=infoPorAvistamiento
            n+=1
            qAvistamientos+=1
        infoAvistamientoTupla=str("<br><h4> <b>"+nAvistamiento+ " "+
"</b></h4>"
                            + "<br><b> Cantidad de
avistamientos:"+str(qAvistamientos)+ " "+ "</b> "
                            + "<br><b> Ciudad, País: </b>"+ciudadPais
                            + "<br><br><b> Info por cada avistamiento:</b><br>"
                            + avistamientosEnLocacion)

        infoHTML=folium.Html(infoAvistamientoTupla,script=True)

        fileImage=cf.data_dir+ "UFOS//ufoSpace.png"
        ufoIcon=folium.features.CustomIcon(fileImage,icon_size=(40,40))
        folium.Marker(location=[latitud, longitud], popup=folium.Popup(infoHTML,
parse_html=False),icon=ufoIcon,tooltip=nAvistamiento).add_to(mapgraf)
```

Para crear el mapa tomamos esta operación como O(*K*), dado que es una función de la librería de folium.

Para añadir los marcadores al mapa tenemos que recorrer la lista de coordenadas agrupadas (resultado de la agrupación de valores de la tabla de símbolos), lo cual es O(*N*). Después de esto se tiene que generar labels para los marcadores del mapa y otras asignaciones que tienen complejidad O(1), lo cual sería O(*K*) por cada avistamiento.

Es así como la complejidad sería:

$$O(K) + O(N*K)$$

$$O(N)$$

Finalmente, la complejidad del requerimiento 6 sería:

1. Caso de tener cargado el resultado cargado del requerimiento 5:

$$O(K) + O(N) + O(N)$$

$$O(N)$$

2. Caso en el que se tenga que llamar al requerimiento 5:

$$N_{longitudes} * (O(\log N_2)) + O(K) + O(N) + O(N)$$

$$N_{longitudes} * (O(\log N_2))$$