Reto 3

Sebastian Guerrero – 202021249

Diego Alejandro Gonzalez – 202110240

Pautas Generales:

De manera general, tras haber llevado a cabo los laboratorios, decidimos llevar a cabo este reto con los árboles rojo negro (RBT), con el objetivo de tener un árbol balanceado y ordenado del cual se puedan extraer valores entre dos límites de manera rápida. Para poder guardar más datos asociados a una misma llave, se decidió utilizar una lista con la estructura de "ARRAY_LIST" y el algoritmo de ordenamiento mergesort. Las pruebas se llevaron a cabo por dos máquinas diferentes para poder entender cómo se comporta la solución en dos diferentes sistemas, y poder evidenciar su eficiencia independientemente de los recursos del computador. Las especificaciones de las diferentes maquinas se evidencia en la siguiente tabla:

	Máquina de prueba	Máquina de Prueba 2
Procesador	Intel (R) Core (TM) i7-10800H CPU @ 3.40 GHz	Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz 2.90 GHz
Memoria RAM (GB)	16 GB	8GB
Sistema Operativo	Windows 10 x64 bits	Windows 10 x64 bits

Requerimiento 1

Para llevar a cabo este requerimiento se decidió implementar un árbol rojo-negro (RBT) como estructura de datos principal, siguiendo los lineamentos del reto. En este árbol, se organizaron las llaves en cuanto a la ciudad del avistamiento. Ya que existe la posibilidad de que dos o mas avistamientos hayan ocurrido en la misma ciudad, se decidió que cada valor corresponde a una lista en la cual se van a guardar los diferentes avistamientos con toda la información necesaria de los mismos. Para poder lograr el árbol propuesto, cada vez que se añade un nuevo elemento, primero se revisa si la ciudad ya existe en los datos cargados, si existe, se extrae la lista de la ciudad y simplemente se añade al ultimo lugar el avistamiento que se esta guardando. Si la ciudad aun no existe en los datos cargados, se crea una nueva lista y se adiciona al árbol con la ciudad como llave y la lista como valor. Este proceso de revisar la existencia en el árbol puede ser tardado con una complejidad O(n) ya que tiene que recorrer todos los elementos del archivo CSV y revisar la existencia de el mismo en el árbol cada iteración, pero este tiempo perdido en la carga de datos va a ser repuesto en el requerimiento en sí. Esta implementación en el catalogo se puede evidenciar en la siguiente imagen:

```
def addCity(catalog, avistamiento):
    """
    Anade una ciudad al mapa si no existe ya
    """
    ciudades = catalog["Ciudad"]

    city = avistamiento["city"]

    existe = om.contains(ciudades,city)

    if existe:
        ci = om.get(ciudades, city)
        ci = me.getValue(ci)

    else:
        ci = lt.newList("ARRAY_LIST")
        om.put(ciudades, city, ci)

    lt.addLast(ci, avistamiento)
```

Tras haber cargado los datos correctamente, podemos comenzar a pensar en el reto en sí. En este caso se puede obtener la respuesta del requerimiento rápidamente, ya que en el cargar datos logramos organizar por ciudad del avistamiento, decidimos simplemente buscar en el árbol por la llave, la cual en este caso corresponde a la ciudad que es ingresada por el usuario. Este tiempo de búsqueda corresponde a O(log n) donde n es el numero de ciudades diferentes que hay en el árbol, al hacer pruebas, podíamos observar que la cantidad de ciudades diferentes en comparación con el total de datos era una parte mínima, por lo que la complejidad de búsqueda se puede considerar como constante O(1). Esto nos devuelve la lista con todos los avistamientos de esta ciudad. La implementación se evidencia a continuación:

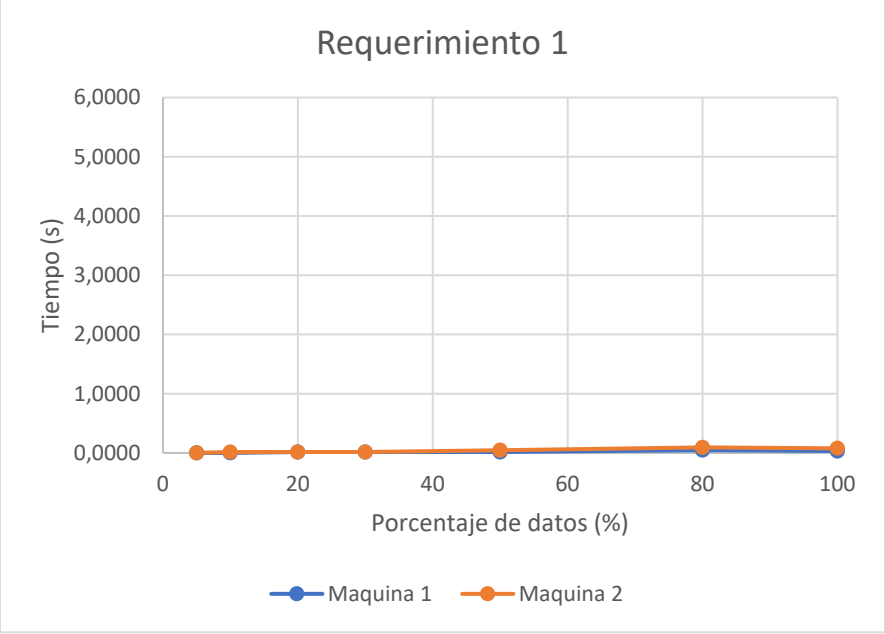
```
def reqUno(catalog, ciudad):

    ciudades = catalog["Ciudad"]

    resp = om.get(ciudades, ciudad)
    resp = me.getValue(resp)
    resp = merge.sort(resp, cmpcronologico)

    return resp
```

Al haber obtenido una lista, también pudimos hacer uso de mergesort con el fin de organizar la respuesta en cuanto a la fecha del avistamiento así, teniendo una complejidad de la comparación como O(m log m) donde m es la cantidad de avistamientos en la ciudad buscada, con las pruebas nos percatamos de que la cantidad de avistamientos en cada ciudad es bastante reducida por lo que este paso también se puede considerar como constante O(1). Para esto se utilizó la siguiente función de comparación:

```
def cmpcronologico(avista1, avista2):
    d1 = datetime.strptime(avista1["datetime"], '%Y-%m-%d %H:%M:%S')
    d2 = datetime.strptime(avista2["datetime"], '%Y-%m-%d %H:%M:%S')
    return (d1<d2)
```

Teniendo en cuenta el análisis previo, **se puede considerar la complejidad del algoritmo constante O(1)** o con muy poco incremento con los datos y su distribución,. Para poder comprobar la complejidad se realizaron las siguientes pruebas:

Porcentaje (%)	Tiempo(s) Maquina 1	Tiempo(s) Maquina 2
5	0,0000	0,0000
10	0,0000	0,0156
20	0,0156	0,0156
30	0,0156	0,0156
50	0,0156	0,0469
80	0,0469	0,0938
100	0,0313	0,0781



Requerimiento 2 (Sebastian Guerrero)

Para la realización de este requerimiento se decidió crear otro árbol rojo-negro. En esta estructura de datos, se decidió tomar como llaves la duración en segundos de los avistamientos. Para esto, se realizo el mismo proceso que en el árbol utilizado para el requerimiento anterior. Del mismo modo, primero se revisaba si la llave, en este caso la duración, ya existía en el árbol, si existía, simplemente se recuperaba la lista guardada en el valor asociado a la llave de esa. Si no existía, se creaba una nueva lista para guardar los avistamientos con la misma duración. En este caso, también se tiene una complejidad elevada de O(n) donde n es la cantidad de avistamientos totales, esto se debe a que es necesario recorrer el archivo completo y recorrer el árbol para poder saber si existe o no una llave en este. A pesar de esta complejidad alta, también es necesario para poder reducir la complejidad del requerimiento como tal.

```
def addDuration(catalog, avistamiento):
    """
    Anade un avistamiento en cuanto a su duracion
    """
    Duration = catalog["Duration"]

    Dura = float(avistamiento["duration (seconds)"])

    existe = om.contains(Duration, Dura)

    if existe:
        sec = om.get(Duration, Dura)
        sec = me.getValue(sec)

    else:
        sec = lt.newList("ARRAY_LIST")
        om.put(Duration, Dura, sec)

    lt.addLast(sec, avistamiento)
```

Con los datos cargados y el árbol en cuanto a duración construido, se puede comenzar con el requerimiento como tal. En este caso se utiliza la función values, ya que el usuario nos entrega los limite superiores e inferiores de los datos que desean. Este proceso no tiene complejidad tan alta ya que se hacen dos búsquedas O(log m) donde m es la cantidad de duraciones diferentes, en este caso tampoco se espera un volumen elevado de duraciones diferentes por lo que se puede aproximar esta búsqueda a una complejidad constante O(1). Tras haber recibido una lista de listas, debido a que cada valor es una lista, se itera por estas para añadirlas a una lista auxiliar que se creo previamente para obtener una respuesta más ordenada para poder acceder e imprimir los datos más fácilmente. Este proceso tiene una complejidad de O(i*j) donde i es la cantidad de duraciones

diferentes en el rango dado por el usuario y j es el promedio de avistamientos en cada duración. En este caso, a pesar de que los tamaños de i y j se espera que sean pequeños, al hacer las pruebas si se evidencio una diferencia en cada porcentaje de los datos, a pesar de estas pequeñas fluctuaciones, se evidencia que las diferencias son mínimas y en todos los volúmenes de datos se obtiene un tiempo favorable. Esta implementación se puede ver en la siguiente imagen:

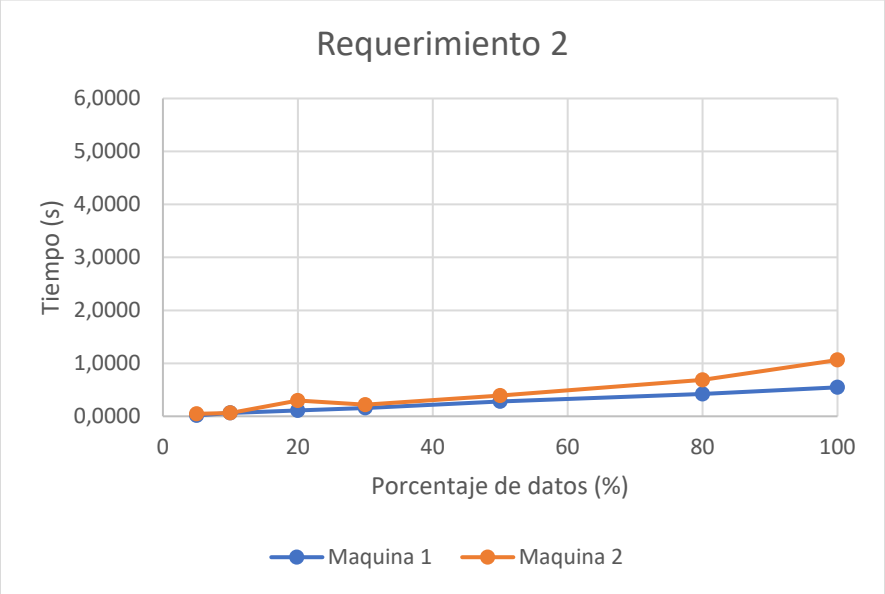
```
def reqDos(catalog, inferior, superior):
    duracion = catalog["Duration"]
    data = lt.newList("ARRAY_LIST")

    resp = om.values(duracion, inferior, superior)
    resp = lt.iterator(resp)
    for i in resp:
        re = lt.iterator(i)
        for j in re:
            lt.addLast(data, j)

    data = merge.sort(data, cmpduration)

    return data
```

Tras haber obtenido la lista de valores completa, se organizaron los datos primero por duración de pequeño a grande, posteriormente, si las duraciones eran las mismas, se organizaba en orden alfabético de la “A” a la “Z”. Este proceso tiene una complejidad de O(m log m) donde m es la cantidad de avistamientos en el rango de duraciones. La función de comparación se puede ver en la siguiente imagen:

```
def cmpduration(avista1, avista2):
    d1 = float(avista1["duration (seconds)"])
    d2 = float(avista2["duration (seconds)"])

    if d1==d2:
        d1 = (avista1["country"] + avista1["city"]).replace(" ","")
        d2 = (avista2["country"] + avista2["city"]).replace(" ","")

    return (d1<d2)
```

En conclusión, se puede decir que este requerimiento se ejecute en un orden temporal según notación Big O de O(1). Para poder comprobar la eficiencia del requerimiento, se realizaron las pruebas en las dos máquinas diferentes, las pruebas se evidencian a continuación:

Porcentaje (%)	Tiempo(s) Maquina 1	Tiempo(s) Maquina 2
5	0,0156	0,0469
10	0,0625	0,0625
20	0,1094	0,2969
30	0,1563	0,2188
50	0,2813	0,3906
80	0,4219	0,6875
100	0,5469	1,0625



Requerimiento 3 (Diego Alejandro Gonzalez Vargas)

Para poder realizar este requerimiento de acuerdo con las estipulaciones de los laboratorios, se pensó en hacer una carga que de datos que contemplara esta estructura de datos de árbol rojo negro con la información necesaria para el requerimiento.

```
catalog["HoraMin"] = om.newMap(omaptype= "RBT")
```

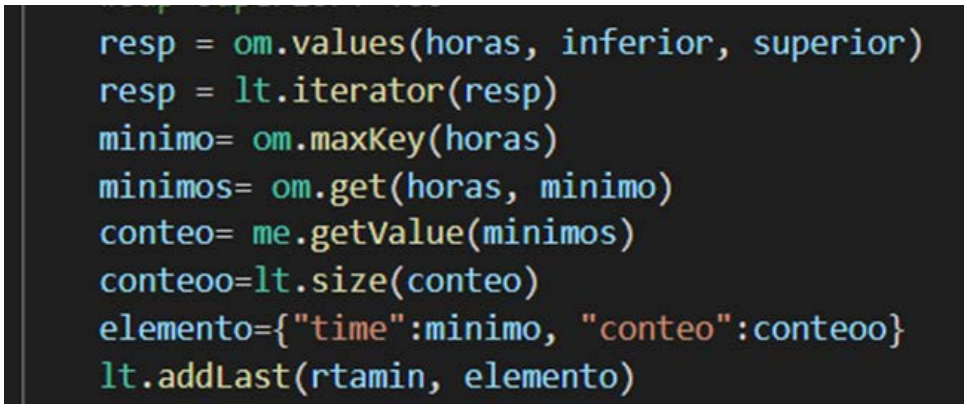
Así, se creó un RBT dentro del catálogo que recibía como llaves cada una de las 1440 posibles combinaciones de horas y minutos [HH:MM], y que como valores tiene listas con la información de todos los avistamientos que se dieron en esa hora y minuto específicos.

```
def addHora(catalog, avistamiento):
    """
    Anade un avistamiento en cuanto a su hora
    """
    horas = catalog["Hora"]

    hora = datetime.strptime(avistamiento["datetime"][11:], '%H:%M:%S')

    existe = om.contains(horas, hora)

    if existe:
        sec = om.get(horas, hora)
        sec = me.getValue(sec)

    else:
        sec = lt.newList("ARRAY_LIST")
        om.put(horas, hora, sec)

    lt.addLast(sec, avistamiento)
```

Sin embargo, como se mencionó, todas estas operaciones, que tienen un orden O(n) por recorrer todos los datos, no deben ser tenidas en cuenta para la complejidad del algoritmo, pues se realizan en el momento de la carga de datos y no en el desarrollo de la función. En consecuencia, una vez se tiene toda esta información organizada, la ejecución del requerimiento queda un poco más ligera. Para empezar, se realiza una serie de asignaciones simples de orden O (1) para poder obtener llamar el RBT del catálogo, la cantidad de horas [HH:MM] diferentes, y crear las listas auxiliares que servirán para almacenar la información respuesta

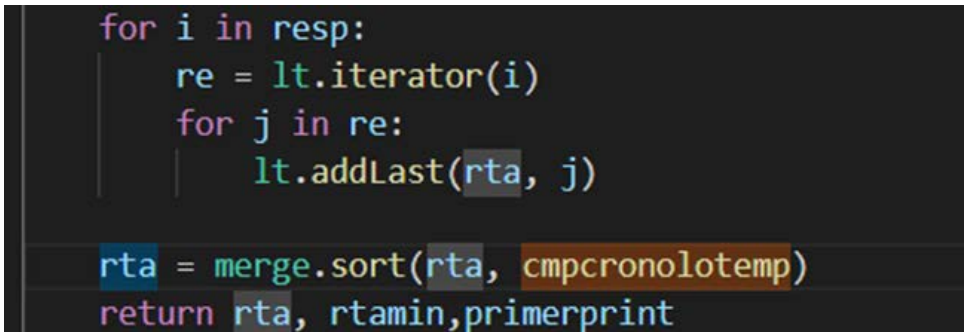
```
def reqTres(catalog, inferior, superior):
    horas = catalog["Hora"]
    primerprint=om.size(horas)
    rta = lt.newList("ARRAY_LIST")
    rtamin=lt.newList("ARRAY_LIST")
```

Posteriormente, aprovechando que los datos se encuentran ordenados dentro del RBT, se utiliza la función values para obtener toda la información que pidió el usuario. Adicionalmente, se utiliza otra de las bondades del RBT como lo es el KeyMax para poder hallar la hora [HH:MM] más tardía con registros, y a partir de la extracción del valor de dicha llave, y la obtención del size de la lista correspondiente, se organiza el TAD lista que contenga toda esta información de manera tabulada sencilla para la impresión.



```
resp = om.values(horas, inferior, superior)
resp = lt.iterator(resp)
minimo= om.maxKey(horas)
minimos= om.get(horas, minimo)
conteo= me.getValue(minimos)
conteoo=lt.size(conteo)
elemento={"time":minimo, "conteo":conteoo}
lt.addLast(rtamin, elemento)
```

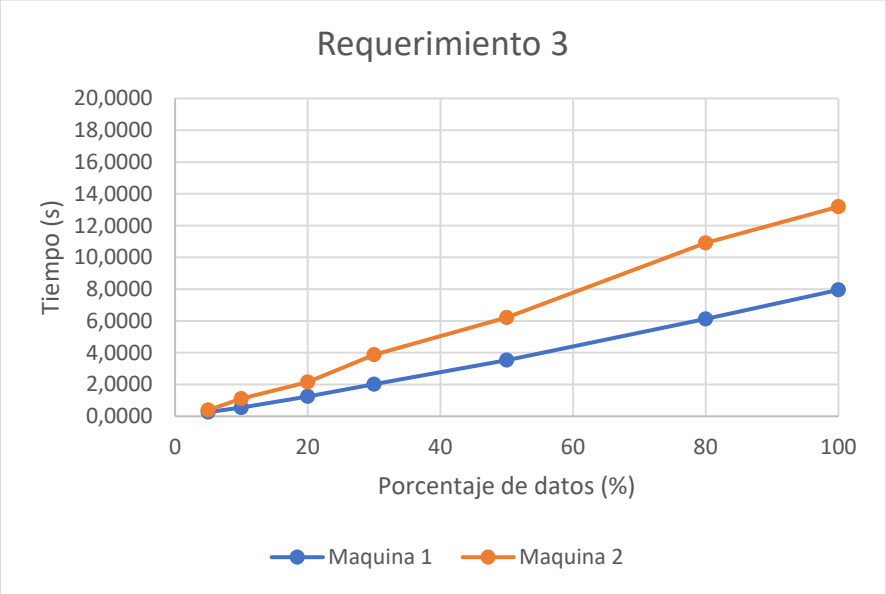
Más adelante, se procede a llenar el TAD lista creado para almacenar la información en el rango horario establecido por parámetro. Para ello, se itera la lista dada por la función values. Esta función tiene una complejidad de O(n), pero este n hace referencia a los datos en el rango horario, que en promedio será muy inferior al total de datos, por lo que no se toma en cuenta para la complejidad algorítmica.



```
for i in resp:
    re = lt.iterator(i)
    for j in re:
        lt.addLast(rta, j)

rta = merge.sort(rta, cmpcronolotemp)
return rta, rtamin,primerprint
```

Finalmente, se realiza un ordenamiento de la lista resultante para cumplir con el requisito de mostrar en orden cronológico aquellos avistamientos en orden cronológico si tienen la misma hora [HH:MM], con una tradicional función MergeSort del TAD lista con complejidad O(nlogn), pero como ya se mencionó, se espera que en casos promedio estas n sean bastante inferiores a la totalidad de los datos. **En conclusión, se puede decir que la complejidad del algoritmo para satisfacer esta función será O(1)**

Porcentaje (%)	Tiempo(s) Maquina 1	Tiempo(s) Maquina 2
5	0,2656	0,3906
10	0,5469	1,1094
20	1,2500	2,1563
30	2,0156	3,8750
50	3,5313	6,2188
80	6,1250	10,9063
100	7,9531	13,1875



Requerimiento 4:

Para poder dar solución a este requerimiento de acuerdo con las indicaciones dadas en la guía de trabajo, se pensó en dedicar una sección de la carga de datos que contemplara la estructura de datos de árbol rojo negro con la información necesaria para el requerimiento.

```
catalog["Fecha"] = om.newMap(omaptype= "RBT")
```

Así las cosas, se creó el RBT dentro del catálogo que recibía como llaves cada una de las posibles fechas de los avistamientos del archivo [AAAA-MM-DD], y que como valores tiene listas con la información de todos los avistamientos que se dieron en esa fecha específica.

```
def addFecha(catalog, avistamiento):
    """
    Anade un avistamiento en cuanto a su fecha
    """
    fechas = catalog["Fecha"]

    fecha = datetime.strptime(avistamiento["datetime"][:10], '%Y-%m-%d')

    existe = om.contains(fechas, fecha)

    if existe:
        sec = om.get(fechas, fecha)
        sec = me.getValue(sec)

    else:
        sec = lt.newList("ARRAY_LIST")
        om.put(fechas, fecha, sec)

    lt.addLast(sec, avistamiento)
```

No obstante, como se mencionó anteriormente, todas las operaciones de la imagen anterior, que tienen un orden O(n) por recorrer todos los datos, no deben ser tenidas en cuenta para la complejidad del algoritmo, pues se realizan en el momento de la carga de datos y no en el desarrollo de la función. En consecuencia, una vez se tiene toda esta información organizada, la ejecución del requerimiento se realiza de una forma más dinámica. En primer lugar, se realiza una serie de asignaciones simples de orden O (1) para poder obtener y llamar el RBT del catálogo, y crear la lista auxiliar que servirá para guardar una respuesta fácil de imprimir:

```
def reqCuatro(catalog, inferior, superior):
    fechas = catalog["Fecha"]
    data = lt.newList("ARRAY_LIST")
```

Luego, aprovechando que los datos se encuentran ordenados dentro del RBT, se utiliza la función values para obtener toda la información que pidió el usuario en el rango de fechas [AAAA-MM-DD] que llega por parámetro. Además, toda esta información que queda en un TAD lista de listas

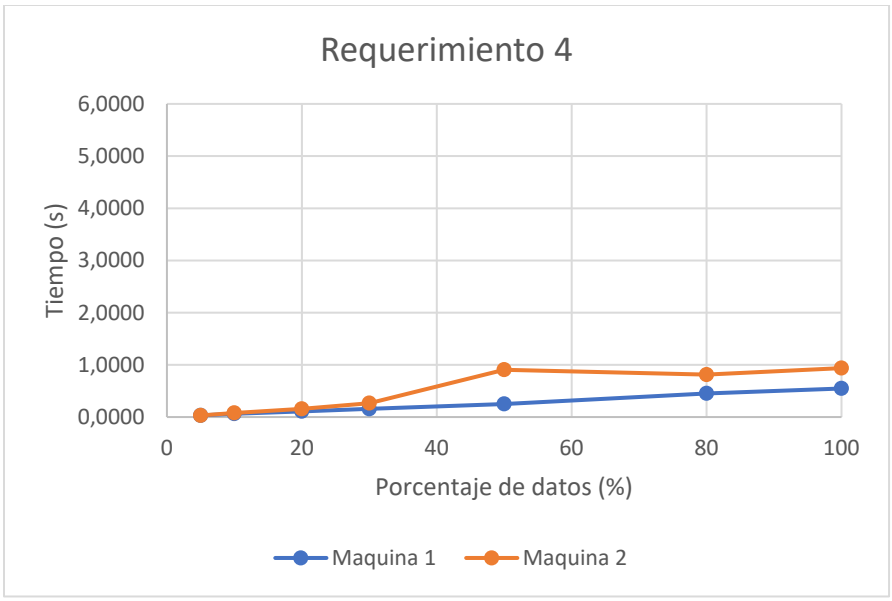
se itera para poder pasar los datos a una sola lista compacta con el TAD auxiliar creado previamente. Este recorrido de cada uno de los TAD lista tendrá una complejidad O(m*n) donde m es la cantidad de fechas diferentes en el rango y n la cantidad de obras promedio en cada fecha [AAAA-MM-DD]. Sin embargo, se espera que el resultado de esta multiplicación para casos promedio sea bastante inferior a la totalidad de datos, por lo que no se toma en cuenta para la complejidad algorítmica:

```
resp = om.values(fechas, inferior, superior)
resp = lt.iterator(resp)
for i in resp:
    re = lt.iterator(i)
    for j in re:
        lt.addLast(data, j)
```

Finalmente, se realiza un ordenamiento de la lista resultante para cumplir con el requisito de mostrar en orden cronológico aquellas obras en orden cronológico si tienen la misma fecha [AAAA-MM-DD], con una tradicional función MergeSort del TAD lista con complejidad O(nlogn), pero como ya se dijo anteriormente, se espera que en casos promedio estas n sean exponencialmente inferiores a la totalidad de los datos. **Concluyendo, se puede una vez más decir que la complejidad del algoritmo para satisfacer esta función será O(1)**

```
data = merge.sort(data, cmpcronologico)

return data
```

Porcentaje (%)	Tiempo(s) Maquina 1	Tiempo(s) Maquina 2
5	0,0313	0,0313
10	0,0625	0,0781
20	0,1094	0,1563
30	0,1563	0,2656
50	0,2500	0,9063
80	0,4531	0,8125
100	0,5469	0,9375


Requerimiento 4
6,0000
5,0000
4,0000
3,0000
2,0000
1,0000
0,0000
Tiempo (s)
0
20
40
60
80
100
Porcentaje de datos (%)
Maquina 1
Maquina 2

Requerimiento 5:

Para poder dar correcta respuesta a este requerimiento de acuerdo con las indicaciones dadas en la guía pdf conductora, se determinó una sección de la carga de datos que contemplara la estructura de datos de árbol rojo negro con la información necesaria para el requerimiento.

```
catalog["Longitud"] = om.newMap(omaptype= "RBT")
```

Así, se creó un RBT dentro del catálogo que recibía como llaves las posibles longitudes racionales que contenga el archivo. Estas llaves tienen a su vez como valores nuevos árboles rojo-negro. Cada uno de estos árboles internos tienen como llaves las posibles latitudes como números racionales, y como valores, tienen listas con las obras que cumplen tanto con la latitud como la longitud de esa sección del almacenamiento.

```
longitudes = catalog["Longitud"]

longitud = float(avistamiento["longitude"])
latitud = float(avistamiento["latitude"])

existe = om.contains(longitudes,longitud)

if existe:
    lon = om.get(longitudes, longitud)
    lon = me.getValue(lon)
    existex2=om.contains(lon,latitud)
    if existex2:
        lat = om.get(lon,latitud)
        lat = me.getValue(lat)
    else:
        lat=lt.newList("ARRAY_LIST")
        om.put(lon,latitud,lat)

    lt.addLast(lat,avistamiento)

else:
    lon = om.newMap(omaptype= "RBT")
    lat=lt.newList("ARRAY_LIST")
    lt.addLast(lat,avistamiento)
    om.put(lon, latitud,lat)
    om.put(longitudes, longitud, lon)
```

Sin embargo, como se mencionó, todas estas operaciones, que tienen un orden O(n) por recorrer todos los datos, no deben ser tenidas en cuenta para la complejidad del algoritmo, pues se realizan en el momento de la carga de datos y no en el desarrollo de la función. En consecuencia, una vez

se tiene toda esta información organizada, la ejecución del requerimiento queda un poco más veloz. Antes que nada, se realiza una serie de dos asignaciones simples de orden O (1) para poder obtener tanto el RBT del catálogo correspondiente a las longitudes, como para crear el TAD lista auxiliar que servirá para almacenar la información de la respuesta.

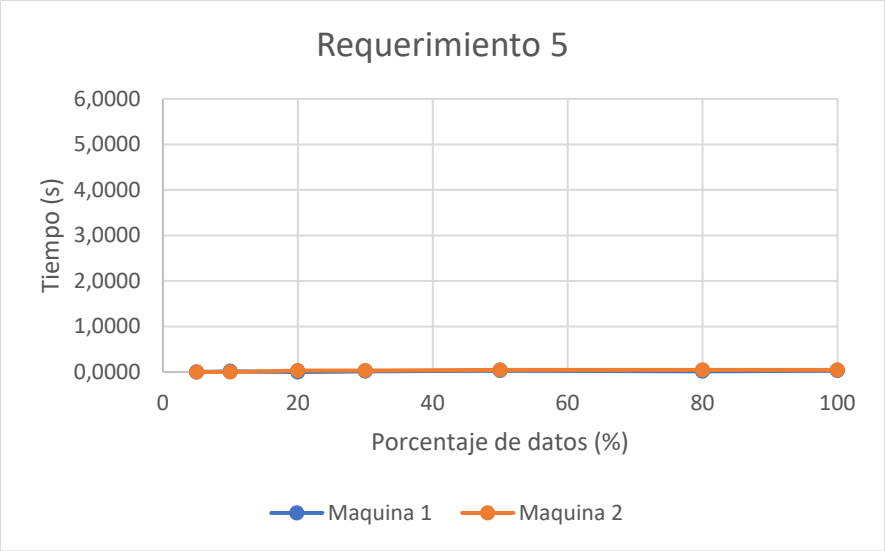
```
def reqCin(catalog, loninferior, lonsuperior,latinferior,latsuperior):
    longitudes = catalog["Longitud"]
    rta = lt.newList("ARRAY_LIST")
```

Consecutivamente, aprovechando que los datos se encuentran ordenados dentro del RBT, se utiliza la función values para obtener toda la información en el rango de longitudes de interés. Adicionalmente, se utiliza un iterador para la lista que arroja esta función values. Dentro de cada uno de los valores de esta lista se encuentran los mapas con las latitudes, por lo que es pertinente reutilizar la función values para obtener todos los datos de las latitudes especificadas. Posteriormente, se itera nuevamente dentro de esta nueva lista resultante, y se van añadiendo los elementos correspondientes al TAD lista de respuesta, como se muestra:

```
    resp = om.values(longitudes, loninferior, lonsuperior)
    resp = lt.iterator(resp)
    for i in resp:
        respuesta= om.values(i,latinferior,latsuperior)
        respuesta = lt.iterator(respuesta)
        for j in respuesta:
            re = lt.iterator(j)
            for k in re:
                lt.addLast(rta,k)
```

Para terminar, se realiza un ordenamiento de la lista resultante para cumplir con el requisito de mostrar en ascendente de latitudes aquellos datos que compartan las especificaciones indicadas por el usuario, con una confiable función MergeSort del TAD lista con complejidad O(nlogn), pero como ya se mencionó, se espera que en casos promedio estas n sean bastante inferiores a la totalidad de los datos. **Así las cosas, se puede decir que la complejidad del algoritmo para satisfacer esta función será O(1).**

Porcentaje (%)	Tiempo(s) Maquina 1	Tiempo(s) Maquina 2
5	0,0000	0,0000
10	0,0156	0,0000
20	0,0000	0,0313
30	0,0156	0,0313
50	0,0313	0,0469
80	0,0156	0,0469
100	0,0313	0,0469


Requerimiento 5
6,0000
5,0000
4,0000
3,0000
2,0000
1,0000
0,0000
Tiempo (s)
0
20
40
60
80
100
Porcentaje de datos (%)
Maquina 1
Maquina 2

Requerimiento 6:

En este requerimiento, se utilizó la respuesta de datos que da la función 5 por lo que **se puede considerar que la parte de procesamiento la complejidad continua siendo constante O(1)**. Por otro lado, para crear el mapa dependemos de la librería folium por lo que es difícil entender la complejidad del requerimiento en total. Igualmente, se realizaron pruebas en las maquinas para entender el tiempo que se tarda para el requerimiento. La implementación y las pruebas se evidencian a continuación:

```
def reqSels(catalog, lonInferior, lonsuperior, latInferior, latsuperior):
    avista = reqCin(catalog, lonInferior, lonsuperior,latinferior,latsuperior)

    avistamientos = lt.iterator(avista)

    m = folium.Map(location=[(latinferior+latsuperior)/2, (loninferior+lonsuperior)/2], tiles="Stamen Terrain", zoom_start=7)

    for i in avistamientos:

        latitud = i["latitude"]
        longitud = i["longitude"]


        iframe = folium.IFrame("Duracion: " + str(i["duration (seconds)"]) + "\n"+
                               "Fecha: " + str(i["datetime"]) + "\n"
                               "Forma: " + str(i["shape"]))
        popup = folium.Popup(iframe,
                             min_width=150,
                             max_width=150)

        folium.Marker(location=[latitud, longitud],
                      popup=popup,
                      icon=folium.Icon(color="blue")
        ).add_to(m)

    m.save("mapa_avistamientos.html")

    return avista
```

Porcentaje (%)	Tiempo(s) Maquina 1	Tiempo(s) Maquina 2
5	0,0625	0,1875
10	0,1406	0,1875
20	0,2813	0,3750
30	0,3750	0,6406
50	0,7347	1,3125
80	1,1563	1,6250
100	1,4063	2,2656