ANÁLISIS DEL RETO

Sergio Alejandro Castaño Arcila, 202310390, sa.castanoa1@uniandes.edu.co

Sebastián Rojas Restrepo, 202317694, s.rojasr2@unaindes.edu.co

Julián Roberto Ramírez Alemán, 202310826, jr.ramireza1@uniandes.edu.co

Requerimiento 1

Plantilla para el documentar y analizar cada uno de los requerimientos.

```python
def req_1(data_structs, initialDate, finalDate):
    """
    Función que soluciona el requerimiento 1
    """
    ini=initialDate+":00.000Z"
    fin=finalDate+":59.999Z"
    llaves= bs.keys(data_structs, ini, fin)
    lst = om.values(data_structs, ini, fin)
    map = folium.Map(location=[0, 0], zoom_start=2)
    marker_cluster = MarkerCluster().add_to(map)

    llavesImp=["mag","lat","long","depth","sig","gap","nst","title", "cdi","mmi","magType","type","code"]
    if lt.size(llaves) !=0:
        tabulat=[]
        if lt.size(llaves) > 6:
            firstMatches = newSublist(llaves, 1, 3)
            lastMatches = newSublist(llaves, int(lt.size(llaves)) - 2, 3)
            sampleMatches = mergelists(firstMatches, lastMatches)
            for llave in lt.iterator(sampleMatches):
                value=sacar_informacion(data_structs,llave)
                retorno=dicinformacion(value,llavesImp)
                tabla=tabulate(retorno, headers="keys", tablefmt="fancy_grid",maxcolwidths=[None,None,5, None,9,7,13,6,5,None,6,7,17], showindex=False)
                fi={"Time":llave, "events":len(retorno),"details":tabla}
                tabulat.append(fi)
        else:
            for llave in lt.iterator(llaves):
                value=sacar_informacion(data_structs,llave)
                retorno=dicinformacion(value,llavesImp)
                tabla=tabulate(retorno, headers="keys", tablefmt="fancy_grid",maxcolwidths=[None,None,5, None,9,7,13,6,5,None,6,7,17], showindex=False)
                fi={"Time":llave, "events":len(retorno),"details":retorno}
                tabulat.append(fi)

        for event_list in lt.iterator(lst):
            for event in lt.iterator(event_list):
                coords = [event['lat'], event['long']]
                info = str(event)

                folium.Marker(location=coords, popup=info).add_to(marker_cluster)

        map_file = cf.maps_path + "/req1.html"
        map.save(map_file)

    else:
        return None
    return tabulat
```

Descripción

La función recibe como parámetros "data_structs", una fecha inicial y una fecha final. Su propósito es explorar el rango de fechas mediante la función "keys" de la estructura de datos "bst", generando así una lista que incluye la información de todas las fechas entre la inicial y la final. En caso de que esta lista tenga más de 6 elementos, la función selecciona las primeras y últimas 3 fechas. Posteriormente, extrae

la información asociada a cada fecha, presentando los resultados en forma tabulada. Finalmente, retorna un diccionario que contiene la fecha, la cantidad de eventos y una tabla detallada de los mismos. Este diccionario se agrega a una lista previamente preparada para facilitar su tabulación en la interfaz visual ("view").

Entrada	Data_struct, fecha de inicio, fecha final
Salidas	Lista con diccionarios, size
Implementado (Sí/No)	Si se implementó y todos.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Sacar la lista con las llaves	O(1)
Sacar menor a 6	O(1)
Sacar informacion	O(log(N))
TOTAL	***O(log(N))***

Pruebas Realizadas

Descripción de las pruebas de tiempos de ejecución y memoria utilizada. Incluir descripción del procedimiento, las condiciones, las herramientas y recursos utilizados (librerías, computadores donde se ejecutan las pruebas, entre otros).

Entrada	**Tiempo (s)**

Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

Graficas

Las gráficas con la representación de las pruebas realizadas.

Análisis

Análisis de resultados de la implementación, tener cuenta las pruebas realizadas y el analisis de complejidad.

Requerimiento 2

```
def req_2(data_structs, initialmag, finalmag):
    """
    Función que soluciona el requerimiento 2
    """
    llavesImp=["time","lat","long","depth","sig","gap","nst","title", "cdi","mmi","magType","type","code"]
    llaves= bs.keys(data_structs, initialmag, finalmag)
    map = folium.Map(location=[0, 0], zoom_start=2)
    marker_cluster = MarkerCluster().add_to(map)
    lst = om.values(data_structs, initialmag, finalmag)
    event=0
    if lt.size(llaves) !=0:
        tabulat=[]
        if lt.size(llaves) > 6:
            firstMatches = newSublist(llaves, 1, 3)
            lastMatches = newSublist(llaves, int(lt.size(llaves)) - 2, 3)
            sampleMatches = mergelists(firstMatches, lastMatches)
            for llave in lt.iterator(sampleMatches):
                value=sacar_informacion(data_structs,llave)
                retorno=dicinformacion(value,llavesImp)
                tabla=tabulate(retorno, headers="keys", tablefmt="fancy_grid",maxcolwidths=[None,None,5, None,9,7,13,6,5,None,6,7,17], showindex=False)
                fi={"mag":llave, "events":len(retorno),"details":tabla}
                tabulat.append(fi)
                event+=len(retorno)
        else:
            for llave in lt.iterator(llaves):
                value=sacar_informacion(data_structs,llave)
                retorno=dicinformacion(value,llavesImp)
                tabla=tabulate(retorno, headers="keys", tablefmt="fancy_grid",maxcolwidths=[None,None,5, None,9,7,13,6,5,None,6,7,17], showindex=False)
                fi={"mag":llave, "events":len(retorno),"details":retorno}
                tabulat.append(fi)
                event+=len(retorno)

        for event_list in lt.iterator(lst):
            for event in lt.iterator(event_list):
                coords = [event['lat'], event['long']]
                info = str(event)

                folium.Marker(location=coords, popup=info).add_to(marker_cluster)

        map_file = cf.maps_path + "/req2.html"
        map.save(map_file)

    else:
        return None
    return tabulat,event
```

Descripción

Entrada	Data_struct, magnitud min y magnitud máxima
Salidas	Lista con diccionarios
Implementado (Sí/No)	Si, implementado por todos.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Sacar la lista con las llaves	O(1)
Sacar menor a 6	O(1)
Sacar informacion	O(log(n))
TOTAL	***O(log(n))***

Pruebas Realizadas

Descripción de las pruebas de tiempos de ejecución y memoria utilizada. Incluir descripción del procedimiento, las condiciones, las herramientas y recursos utilizados (librerías, computadores donde se ejecutan las pruebas, entre otros).

Entrada	**Tiempo (s)**

Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

Graficas

Las gráficas con la representación de las pruebas realizadas.

Análisis

Análisis de resultados de la implementación, tener cuenta las pruebas realizadas y el analisis de complejidad.

Requerimiento 3

```
def req_3(data_structs, mag, prof):

    """
    Función que soluciona el requerimiento 3
    magnitud minima osea revisar los mayores
    profundidad maxima no superable
    """
    llavesImp=["mag","lat","long","depth","sig","gap","nst","title", "cdi","mmi","magType","type","code"]
    profundidad=data_structs["depth"]
    magnitud=data_structs["mag"]
    maxMag=bs.maxKey(magnitud)
    minpro=bs.minKey(profundidad)
    listmag=bs.keys(magnitud,mag, maxMag)
    listapro=bs.keys(profundidad, minpro, prof)
    lst = om.values(data_structs["depth"], float(mag)+0.1, maxMag)
    map = folium.Map(location=[0, 0], zoom_start=2)
    marker_cluster = MarkerCluster().add_to(map)
    mapdata = lt.newList('ARRAY_LIST')
    for lstdepth in lt.iterator(lst):
        for item in lt.iterator(lstdepth):
            if item["depth"] == "" or item["depth"] == None:
                continue
            elif float(item["depth"]) <= float(prof)-0.1:
                lt.addLast(mapdata, item)

    if lt.size(listmag) != 0 and lt.size(listapro) !=0:
        tabulat=[]
        iguales, magi, profe=encontrar_semejantes(listmag,listapro, magnitud)
        if lt.size(iguales)>10:
            lastMatches = newSublist(iguales, int(lt.size(iguales)) - 10, 10)
            for fecha in lt.iterator(lastMatches):
                val=sacar_informacionre3(data_structs["bydate"], fecha)
                retorno=dicinformacion_req3(val,llavesImp,magi, profe)
                tabla=tabulate(retorno, headers="keys", tablefmt="fancy_grid",maxcolwidths=[None,None,5, None,9,7,13,6,5,None,6,7,17], showindex=False)
                fi={"Time":fecha, "events":len(retorno),"details":tabla}
                tabulat.append(fi)
        else:
            for fecha in lt.iterator(iguales):
                val=sacar_informacionre3(data_structs["bydate1"], fecha)
                retorno=dicinformacion_req3(val,llavesImp,magi, profe)
                tabla=tabulate(retorno, headers="keys", tablefmt="fancy_grid",maxcolwidths=[None,None,5, None,9,7,13,6,5,None,6,7,17], showindex=False)
                fi={"Time":fecha, "events":len(retorno),"details":retorno}
                tabulat.append(fi)

        for event in lt.iterator(mapdata):
            coords = [event['lat'], event['long']]
            info = str(event)

            folium.Marker(location=coords, popup=info).add_to(marker_cluster)

        map_file = cf.maps_path + "/req3.html"
        map.save(map_file)
    else:
        return None
    return tabulat
```

Descripción

La función recibe como parámetros el "data_struct", una magnitud que indica la búsqueda de valores mayores entre dos puntos específicos, y una profundidad que tiene un límite fijo. En primer lugar, se determina la magnitud máxima en el árbol de magnitudes y la profundidad mínima en el árbol de profundidades. A continuación, se identifican todas las magnitudes entre la magnitud del parámetro y la máxima del árbol, para luego compararlas con la lista de profundidades menores a la proporcionada como parámetro. Se extrae la información necesaria para comprender cómo ocurrieron los eventos. Finalmente, se organiza la información por fechas, se tabulan los resultados y se retorna un diccionario que incluye la fecha, la cantidad de eventos en esa fecha y la información correspondiente a dicho “Time”.

Entrada	Data_struct,magnitud y profundidad
Salidas	Lista con diccionarios
Implementado (Sí/No)	Si, implementado por julian.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	Complejidad
Sacar la lista con las llaves	O(1)
Sacar menor a 6	O(1)
Sacar informacion	O(log(n))
TOTAL	***O(log(n))***

Pruebas Realizadas

Descripción de las pruebas de tiempos de ejecución y memoria utilizada. Incluir descripción del procedimiento, las condiciones, las herramientas y recursos utilizados (librerías, computadores donde se ejecutan las pruebas, entre otros).

Entrada	Tiempo (s)

Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

Graficas

Las gráficas con la representación de las pruebas realizadas.

Análisis

Análisis de resultados de la implementación, tener cuenta las pruebas realizadas y el analisis de complejidad.

Requerimiento 4

Plantilla para el documentar y analizar cada uno de los requerimientos.

Descripción

Este requerimiento se encarga de retornar el numero total de eventos sismicos con mayor significancia y menor distancia azimutal que se indica y, además los eventos que cumplan los parámetros establecidos. Lo primero que hace es crear un arbol rojo negro donde se almacena los eventos filtrados

```
seismic_events = data_structs['earthquakes']
filtered_events = om.newMap(omaptype='RBT',
                            comparefunction= comparedates)
```

, después se itera en la lista principal donde podamos encontrar todo para hacer las comparaciones dado los parámetros de sig. y gap y va añadiendo dependiendo de la fecha

```
for event in lt.iterator(seismic_events):
    if event['sig'] > sig and event['gap'] < gap:
        date = event['time'].split("T")[0]
        if om.contains(filtered_events, date):
            date_events = om.get(filtered_events, date)['value']
        else:
            date_events = lt.newList('ARRAY_LIST')
            om.put(filtered_events, date, date_events)
        lt.addLast(date_events, event)

sorted_dates = merg.sort(om.keySet(filtered_events), comparedates)
recent_dates = lt.subList(sorted_dates, lt.size(sorted_dates) - 14, 15)
```

, posteriormente, hace un merge sort de las fechas y una sublista con estas, para finalmente crear una lista para añadir los eventos.

```
recent_events = lt.newList('ARRAY_LIST')
for date in lt.iterator(recent_dates):
    date_events = om.get(filtered_events, date)['value']
    for event in lt.iterator(date_events):
        lt.addLast(recent_events, event)

return recent_events
```

Entrada	Estructura de datos, significancia mínima, (sig), a distancia azimutal máxima (gap)
Salidas	El número total de eventos sísmicos registrados mayores a la significancia y menores a la distancia azimutal indicada y los 15 eventos que cumpla ciertos parametros
Implementado (Sí/No)	Si, implementado por Sebastián Rojas

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**

Iterar sobre earthquakes	O(n)
Crear arbol rojo	O(log n)
Sortear fechas	O(n log n)
TOTAL	***O(n log n)***

Pruebas Realizadas

Descripción de las pruebas de tiempos de ejecución y memoria utilizada. Incluir descripción del procedimiento, las condiciones, las herramientas y recursos utilizados (librerías, computadores donde se ejecutan las pruebas, entre otros).

Entrada	**Tiempo (s)**
small	
30 pct	
large	

Tablas de datos

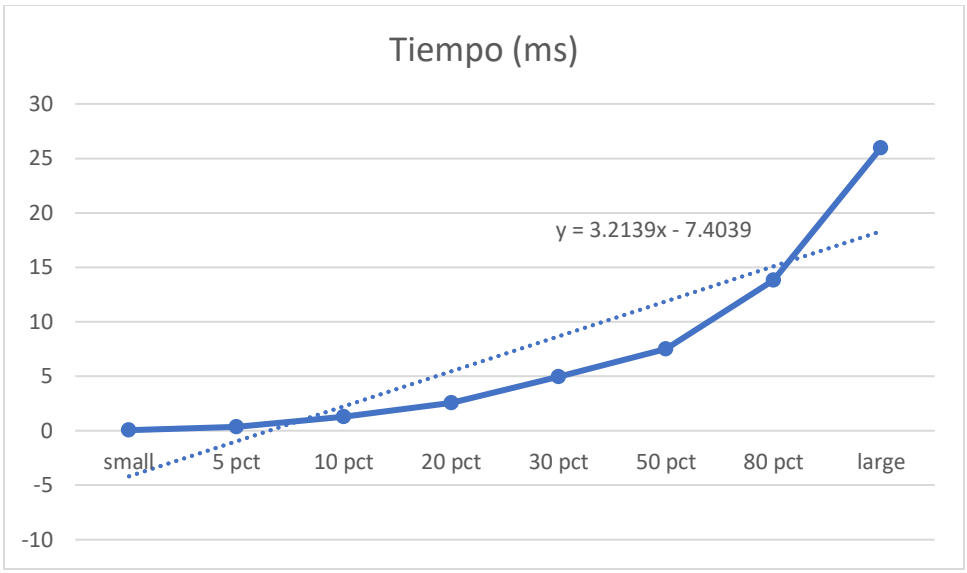
Las tablas con la recopilación de datos de las pruebas.

Graficas

Las gráficas con la representación de las pruebas realizadas.

Análisis

Análisis de resultados de la implementación, tener cuenta las pruebas realizadas y el analisis de complejidad.



Análisis

A pesar de que obtener un elemento en un *ArrayList,* dada su posición, tiene complejidad constante, la implementación de este requerimiento tiene un orden lineal O(n). Esto debido a que, lo primero que se hace es verificar si el elemento hace parte de la lista. Específicamente, a la hora de buscar un elemento en una lista, en el peor de los casos es necesario recorrer toda la lista, es decir, complejidad lineal.

Este comportamiento se puede evidenciar experimentalmente en la gráfica. Ya que, gracias a que los datos no se encuentran tan dispersos con respecto a la línea de tendencia, la curva coincide con el comportamiento lineal esperado.

Requerimiento 5

```python
def req_5(data_structs, depth, nst, consultnum):
    """
    Función que soluciona el requerimiento 5
    """
    data = lt.newList("ARRAY_LIST")
    maxdepth = om.maxKey(data_structs["depth"])
    lst = om.values(data_structs["depth"], depth, maxdepth)
    totalevents = 0
    map = folium.Map(location=[0, 0], zoom_start=2)
    marker_cluster = MarkerCluster().add_to(map)
    full = lt.newList("ARRAY_LIST")

    if lst == None:
        return None

    for lstdepth in lt.iterator(lst):
        for item in lt.iterator(lstdepth):
            dic = {"time": None, "events": None, "details": []}
            innerdic = {}
            if item["nst"] == "" or item["nst"] == None:
                item["nst"] = 1
            if float(item["nst"]) >= nst:
                lt.addLast(full, item)
                if dic["time"] == None:
                    dic["time"] = item["time"][:16]
                    dic["events"] = 1
                    totalevents += 1
                    innerdic = {"mag": item["mag"], "lat": item["lat"], "long": item["long"], "depth": item["depth"], "sig": item["sig"],
                                "gap": item["gap"], "nst": item["nst"], "title": item["title"], "cdi": item["cdi"], "mmi": item["mmi"],
                                "magType": item["magType"], "type": item["type"], "code": item["code"]}
                    dic["details"].append(innerdic)
                    lt.addLast(data, dic)
                else:
                    dic["events"] += 1
                    totalevents += 1
                    innerdic = {"mag": item["mag"], "lat": item["lat"], "long": item["long"], "depth": item["depth"], "sig": item["sig"],
                                "gap": item["gap"], "nst": item["nst"], "title": item["title"], "cdi": item["cdi"], "mmi": item["mmi"],
                                "magType": item["magType"], "type": item["type"], "code": item["code"]}
                    dic["details"].append(innerdic)
                    lt.addLast(data, dic)

    sort_time(data)
```

```python
    # Se mira si el tamaño de el array es mayor al solicitado, si lo es se toman los ultimos x datos pedidos
    if int(lt.size(data)) > consultnum:
        selectedData = newSublist(data, int(lt.size(data)) - consultnum + 1, consultnum)
        mapdata = newSublist(full, int(lt.size(data)) - consultnum + 1, consultnum)
    else:
        selectedData = data
        mapdata = full

    for event in lt.iterator(mapdata):
        coords = [event['lat'], event['long']]
        info = str(event)

        folium.Marker(location=coords, popup=info).add_to(marker_cluster)

    map_file = cf.maps_path + "/req5.html"
    map.save(map_file)

    # Conversión a lista de python para tabulate
    finaldata = []
    if lt.size(selectedData) > 6:
        first3 = newSublist(selectedData, 1, 3)
        last3 = newSublist(selectedData, int(lt.size(selectedData)) - 2, 3)
        sampleData = mergelists(first3, last3)
        for item in lt.iterator(sampleData):
            newdetails = tabulate(item["details"], headers="keys", tablefmt="fancy_grid")
            item["details"] = newdetails
            finaldata.append(item)
    else:
        for item in lt.iterator(data):
            newdetails = tabulate(item["details"], headers="keys", tablefmt="fancy_grid")
            item["details"] = newdetails
            finaldata.append(item)

    return lt.size(data), totalevents, finaldata
```

Descripción

La función recibe como parámetros "data_structs", una profundidad mínima, un número de estaciones mínimo y el número de datos recientes a tomar. Su propósito es explorar el rango de profundidades que cumplan la condición de ser mayores la anteriormente dada mediante la función "values" de la

estructura de datos, generando así una lista que incluye la información de todos los datos que cumplen con el requisito, para poder iterarlo y revisar si cumplen con el número de estaciones mínimo, siendo que si el dato este vacío se establece como uno (al menos 1 estación debió tomar los datos del terremoto). Al verificarlo crea un diccionario que dentro contiene el tabulate de sus detalles. La función selecciona el número de datos más reciente cronológicamente que se haya solicitado, con estos datos crea el mapa respectivo, y en caso de que esta lista tenga más de 6 elementos, la función selecciona las primeros y últimos 3 elementos. Posteriormente, extrae la información asociada a cada fecha, presentando los resultados en forma tabulada. Finalmente, retorna el tamaño de la consulta, la cantidad de eventos y una lista con los datos para el tabulate de los mismos.

Entrada	Data_struct, profundidad mínima, # de estaciones mínimo, # de consultas deseado
Salidas	Tamaño de la consulta, la cantidad de eventos y la lista para tabulate
Implementado (Sí/No)	Si se implementó, la implemento Sergio.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Sacar la lista con los valores, y crear las variables que se usaran	O(log(N))
Iterar los datos y tomar los datos (Hay 2 for debido a que hay arrays dentro de cada key para cuando la key se repite)	$O(N^2)$
Sortear los datos (quicksort)	O(log(N))
Tomar los últimos datos cronológicamente de acuerdo a la cantidad solicitada	O(1)
Crear el mapa	O(N)
Crear la lista del tabulate	O(N)
TOTAL	***$O(N^2)$***

Pruebas Realizadas

Descripción de las pruebas de tiempos de ejecución y memoria utilizada. Incluir descripción del procedimiento, las condiciones, las herramientas y recursos utilizados (librerías, computadores donde se ejecutan las pruebas, entre otros).

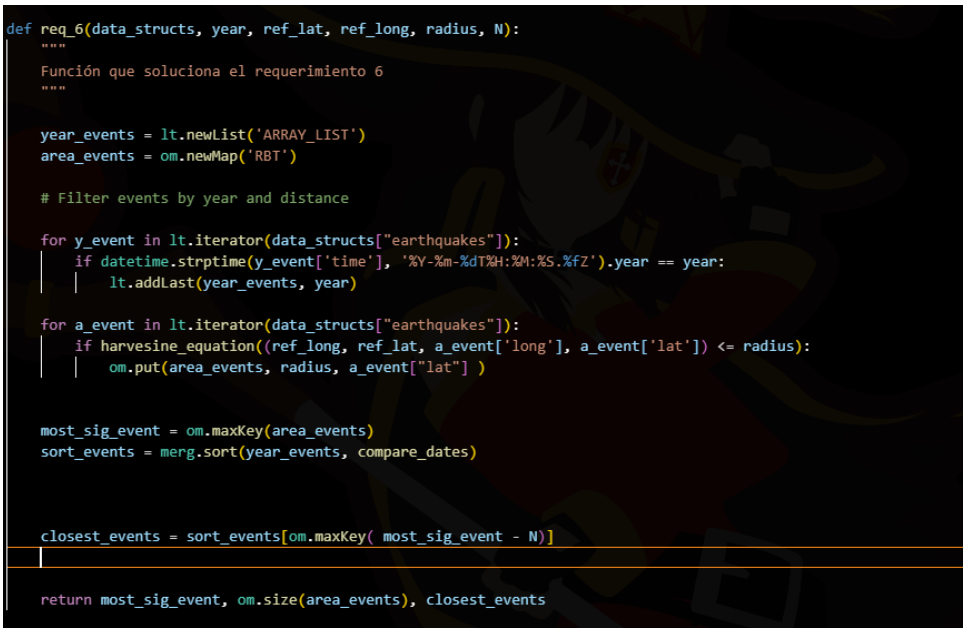
Entrada	**Tiempo (s)**

Tablas de datos
Las tablas con la recopilación de datos de las pruebas.

Graficas
Las gráficas con la representación de las pruebas realizadas.

Análisis
Análisis de resultados de la implementación, tener cuenta las pruebas realizadas y el analisis de complejidad.

Requerimiento 6



```
def req_6(data_structs, year, ref_lat, ref_long, radius, N):
    """
    Función que soluciona el requerimiento 6
    """

    year_events = lt.newList('ARRAY_LIST')
    area_events = om.newMap('RBT')

    # Filter events by year and distance

    for y_event in lt.iterator(data_structs["earthquakes"]):
        if datetime.strptime(y_event['time'], '%Y-%m-%dT%H:%M:%S.%fZ').year == year:
            lt.addLast(year_events, year)

    for a_event in lt.iterator(data_structs["earthquakes"]):
        if harvesine_equation((ref_long, ref_lat, a_event['long'], a_event['lat']) <= radius):
            om.put(area_events, radius, a_event["lat"] )


    most_sig_event = om.maxKey(area_events)
    sort_events = merg.sort(year_events, compare_dates)



    closest_events = sort_events[om.maxKey( most_sig_event - N)]


    return most_sig_event, om.size(area_events), closest_events
```

Descripción

El requerimiento retorna el evento más significativo de ese año, el numero total de eventos sismicos respecto al area con la ecuacion de harvesine y los n eventos más cercanos a lo primero. Primero creamos un arbol y una lista en donde se almacena por año y por area, despues hacemos comparaciones basados en la ecuacion de harvesine y en el año dado, organizamos y buscamos las llaves mayores y finalmente retornamos los resultados.

Entrada	El año, latitud, longitud, radio, numero N eventos a mostrar
Salidas	El requerimiento retorna el evento más significativo de ese año, el numero total de eventos sismicos respecto al area con la ecuacion de harvesine y los n eventos más cercanos a lo primero.
Implementado (Sí/No)	Si, implementado por el grupo

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Iterar sobre earthquakes	O(n)
Om. Put addlast	O(1)
MaxKey	O(log n)
TOTAL	***O(n log n)***

Pruebas Realizadas

Descripción de las pruebas de tiempos de ejecución y memoria utilizada. Incluir descripción del procedimiento, las condiciones, las herramientas y recursos utilizados (librerías, computadores donde se ejecutan las pruebas, entre otros).

Entrada	**Tiempo (s)**
small	
30 pct	
large	

Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

Graficas

Las gráficas con la representación de las pruebas realizadas.

Análisis

Análisis de resultados de la implementación, tener cuenta las pruebas realizadas y el analisis de complejidad.

Requerimiento 7

```
def req_3(data_structs, mag, prof):

    """
    Función que soluciona el requerimiento 3
    magnitud minima osea revisar los mayores
    profundidad maxima no superable
    """
    llavesImp=["mag","lat","long","depth","sig","gap","nst","title", "cdi","mmi","magType","type","code"]
    profundidad=data_structs["depth"]
    magnitud=data_structs["mag"]
    maxMag=bs.maxKey(magnitud)
    minpro=bs.minKey(profundidad)
    listmag=bs.keys(magnitud,mag, maxMag)
    listapro=bs.keys(profundidad, minpro, prof)
    lst = om.values(data_structs["depth"], float(mag)+0.1, maxMag)
    map = folium.Map(location=[0, 0], zoom_start=2)
    marker_cluster = MarkerCluster().add_to(map)
    mapdata = lt.newList('ARRAY_LIST')
    for lstdepth in lt.iterator(lst):
        for item in lt.iterator(lstdepth):
            if item["depth"] == "" or item["depth"] == None:
                continue
            elif float(item["depth"]) <= float(prof)-0.1:
                lt.addLast(mapdata, item)

    if lt.size(listmag) != 0 and lt.size(listapro) !=0:
        tabulat=[]
        iguales, magi, profe=encontrar_semejantes(listmag,listapro, magnitud)
        if lt.size(iguales)>10:
            lastMatches = newSublist(iguales, int(lt.size(iguales)) - 10, 10)
            for fecha in lt.iterator(lastMatches):
                val=sacar_informacionre3(data_structs["bydate"], fecha)
                retorno=dicinformacion_req3(val,llavesImp,magi, profe)
                tabla=tabulate(retorno, headers="keys", tablefmt="fancy_grid",maxcolwidths=[None,None,5, None,9,7,13,6,5,None,6,7,17], showindex=False)
                fi={"Time":fecha, "events":len(retorno),"details":tabla}
                tabulat.append(fi)
        else:
            for fecha in lt.iterator(iguales):
                val=sacar_informacionre3(data_structs["bydate1"], fecha)
                retorno=dicinformacion_req3(val,llavesImp,magi, profe)
                tabla=tabulate(retorno, headers="keys", tablefmt="fancy_grid",maxcolwidths=[None,None,5, None,9,7,13,6,5,None,6,7,17], showindex=False)
                fi={"Time":fecha, "events":len(retorno),"details":retorno}
                tabulat.append(fi)

        for event in lt.iterator(mapdata):
            coords = [event['lat'], event['long']]
            info = str(event)

            folium.Marker(location=coords, popup=info).add_to(marker_cluster)

        map_file = cf.maps_path + "/req3.html"
        map.save(map_file)
    else:
        return None
    return tabulat
```

Descripción

La función recibe como parámetros "data_structs", un año, un área y una propiedad. Su propósito es explorar el rango de profundidades que cumplan la condición de estar en el año dado mediante la función "values" de la estructura de datos, generando así una lista que incluye la información de todos los datos que cumplen con el requisito, para poder iterarlo y revisar si cumplen con estar en el área. Al verificarlo crea un arraylist para los datos que usaran en el mapa, una lista para tomar los primeros y últimos 3 datos para la tabla de mathplotlib y otra ultimo para los datos con los que se graficara el histograma. La función ordena los datos, con estos datos crea el mapa respectivo, y en caso de que esta lista tenga más de 6 elementos, la función selecciona las primeros y últimos 3 elementos. Posteriormente, extrae la información asociada a cada fecha, presentando los resultados en forma tabulada. Finalmente, retorna el tamaño de los eventos en el año, la cantidad de tomados para el histograma, el valor mínimo y máximo del mismo, los datos sorteados para graficar y los eventos para la tabla debajo de la gráfica.

Entrada	Data_struct, fecha de inicio, fecha final
Salidas	Tamaño de los eventos en el año, cantidad de tomados para el histograma, valor mínimo y máximo del mismo, los datos sorteados para graficar y los eventos para la tabla debajo de la grafica

Implementado (Sí/No)	Si se implementó y todos.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Sacar la lista con los valores, y crear las variables que se usaran	O(log(N))
Iterar los datos y tomar los datos (Hay 2 for debido a que hay arrays dentro de cada key para cuando la key se repite)	$O(N^2)$
Crear el mapa	O(N)
Sortear los datos	O(log(N))
Tomar los últimos datos cronológicamente de acuerdo a la cantidad solicitada	O(1)
Organizar los datos para la tabla	O(N)
TOTAL	***$O(N^2)$***

Pruebas Realizadas

Descripción de las pruebas de tiempos de ejecución y memoria utilizada. Incluir descripción del procedimiento, las condiciones, las herramientas y recursos utilizados (librerías, computadores donde se ejecutan las pruebas, entre otros).

Entrada	**Tiempo (s)**

Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

Graficas

Las gráficas con la representación de las pruebas realizadas.

Análisis

Análisis de resultados de la implementación, tener cuenta las pruebas realizadas y el analisis de complejidad.

Requerimiento 8

```python
def req_8():
    """
    Función que soluciona el requerimiento 8
    """
    map_files = ['req1.html', 'req2.html', 'req3.html', 'req4.html', 'req5.html', 'req6.html', 'req7.html']

    for map_file in map_files:

        full_path = os.path.join(cf.maps_path, map_file)

        if os.path.exists(full_path):
            webbrowser.open_new_tab(full_path)
        else:
            print(f"El archivo {full_path} no existe, se ha saltado.")
```

Descripción

La función abre en el navegador web los mapas, al iterar para revisar que estén en la carpeta "Maps", que se crearon a lo largo de la carga de los demás requerimientos.

Entrada	Nada
Salidas	Nada
Implementado (Sí/No)	Si se implementó y todos.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Iterar para revisar si existen los mapas html en la carpeta respectiva y abrir el navegador	O(N)
TOTAL	***O(N)***

Pruebas Realizadas

Descripción de las pruebas de tiempos de ejecución y memoria utilizada. Incluir descripción del procedimiento, las condiciones, las herramientas y recursos utilizados (librerías, computadores donde se ejecutan las pruebas, entre otros).

Entrada	**Tiempo (s)**

Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

Graficas

Las gráficas con la representación de las pruebas realizadas.

Análisis

Análisis de resultados de la implementación, tener cuenta las pruebas realizadas y el analisis de complejidad.