ANÁLISIS DEL RETO 3

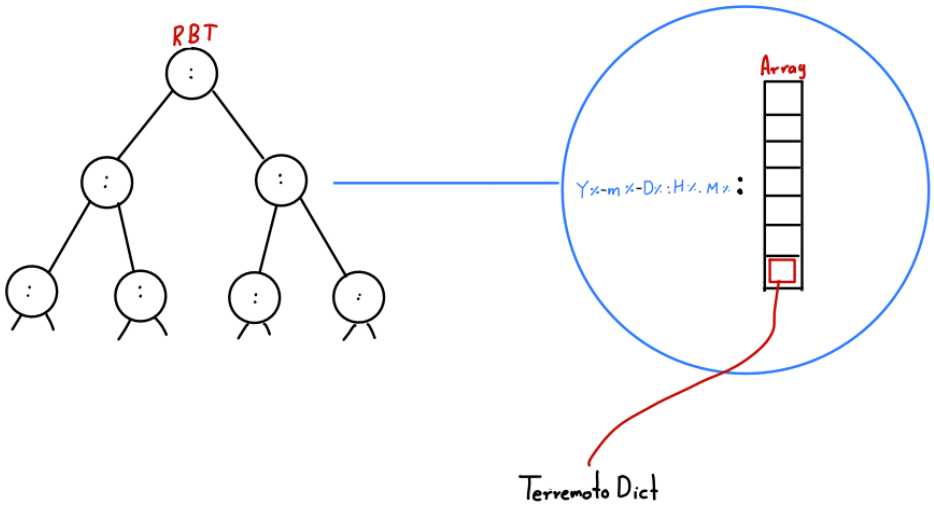
Andres Chaparro Diaz, 202111146, a.chaparrod

Edward Camilo Sánchez Novoa, 202113020, e.sanchezn

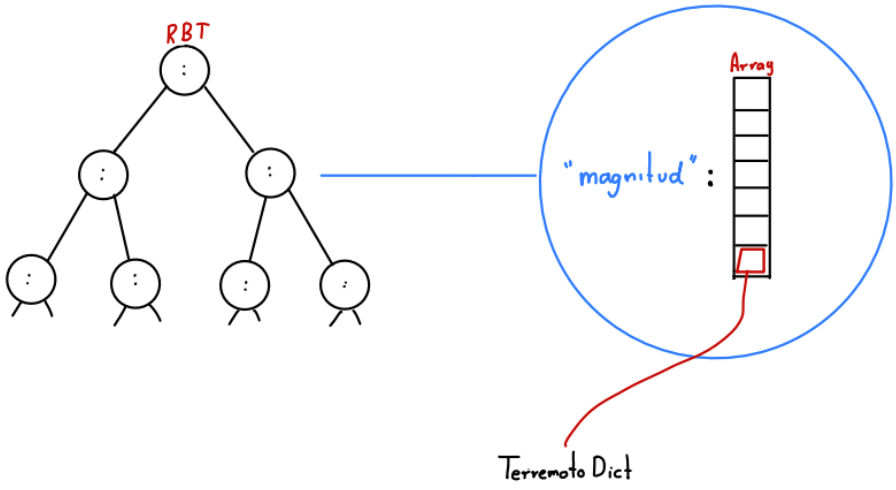
Juan Esteban Rojas, 202124797, je.rojasc1

Carga de Datos

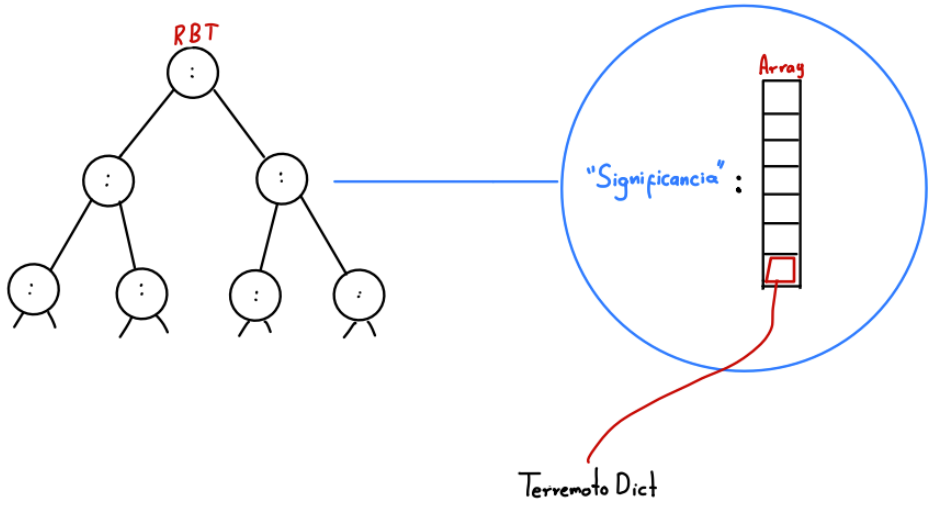
- **ArbolFechas**



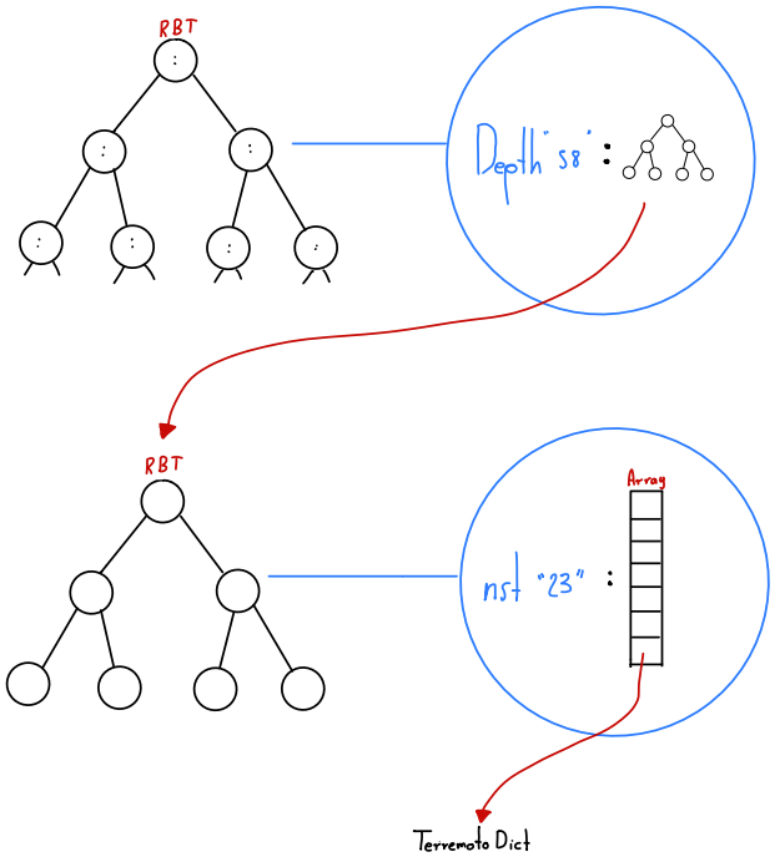
- **ArbolMagnitudes**



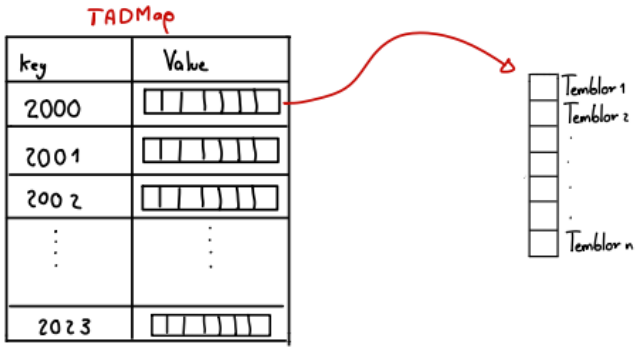
- **ArbolSignificancia**



- **DepthTree**



- **MapYears**



Requerimiento 1

Descripción

```
def req_1(control, fecha_inicial, fecha_final):
    """
    Función que soluciona el requerimiento 1
    """
    fecha_inicial = datetime.strptime(fecha_inicial, '%Y-%m-%dT%H:%M')
    fecha_final = datetime.strptime(fecha_final, '%Y-%m-%dT%H:%M')
    results = om.values(control['ArbolFechas'], fecha_inicial, fecha_final)
    return results
```

Para el desarrollo del requerimiento 1, se utilizó el mapa binario ordenado de las magnitudes. El mapa ‘ArbolFechas contiene llaves del tipo:

{fecha1: [temblor 1] , fecha2 : [temblor2] …, fechan :[temblor n]}

Para este requerimento se uso la funcion de values que nos retorna todos los valores que se encuentren entre la fecha inicial y la fecha final.

Entrada	control, fecha inicial, fecha final
Salidas	La lista con los valores entre estas fechas
Implementado (Sí/No)	SI

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

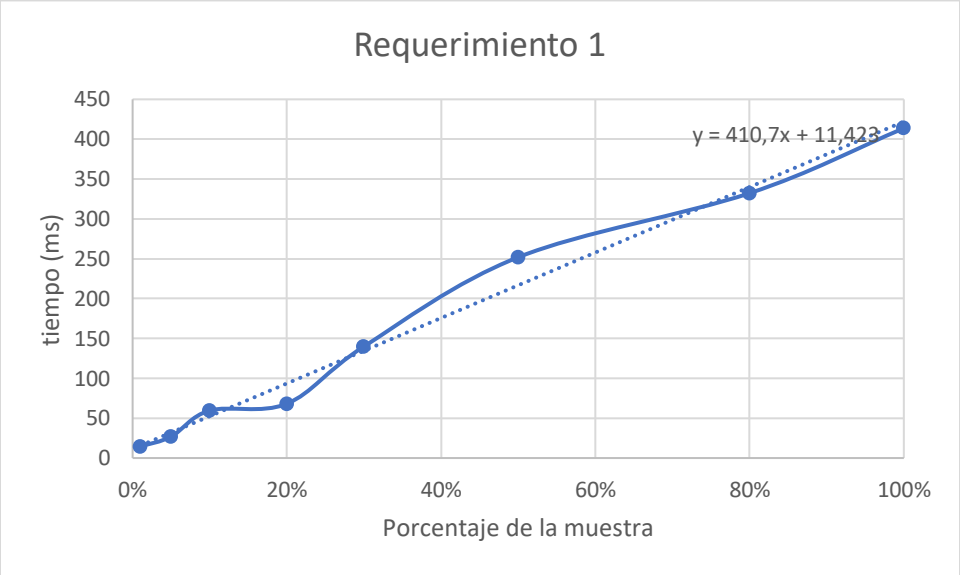
Pasos	**Complejidad**
Iteración por cada temblor entre estas fechas	O(log n)
TOTAL	***O (log n)***

Pruebas Realizadas

Procesadores	intel(R) Core(TM) i3-1005G1 CPU @ 1.20GHz, 1201
Memoria RAM	12 GB

REQ 1		
1%	small	14,4
5%	5 pct	26,95
10%	10 pct	59,64
20%	20 pct	68,14
30%	30 pct	139,97
50%	50 pct	251,94
80%	80 pct	332,27
100%	large	413,73

Graficas



Análisis

En el requerimiento 1 se encontró una complejidad O (log n) ya que para buscar los temblores entre la fecha inicial y final debido a su estructura de mapa ordenado lo hace mediante una búsqueda binaria que es una complejidad logarítmica. Por lo tanto esta estructura es adecuada para buscar rangos.

Requerimiento 2

Descripción

```python
def req_2(data_structs, min_mag, max_mag):
    """
    Función que soluciona el requerimiento 2
    """
    min_mag = str(float(min_mag))
    max_mag = str(float(max_mag))
    lista = om.values(data_structs['ArbolMagnitudes'], min_mag, max_mag)
    diferent_mag = lt.size(lista)
    total_events = 0
    for lista_i in lt.iterator(lista):
        total_events += lt.size(lista_i)
    primeros = lt.subList(lista, 1, 3)
    ultimos = lt.subList(lista, lt.size(lista)-2, 3)
    lista_grande = lt.newList(datastructure='ARRAY_LIST')
    for i in range(1,4):
        lt.addLast(lista_grande, [lt.getElement(lt.getElement(primeros,i), 1)['mag'], lt.size(lt.getElement(primeros,i)), ''])
    for i in range(1,4):
        lt.addLast(lista_grande, [lt.getElement(lt.getElement(ultimos,i), 1)['mag'], lt.size(lt.getElement(ultimos,i)), ''])
    sub_lista = lt.newList(datastructure='ARRAY_LIST')
    for i in range(1,4):
        primeros_i = quk.sort(lt.getElement(primeros,i),sort_criteria)
        lt.addLast(sub_lista, primeros_i)
    for i in range(1,4):
        ultimos_i = quk.sort(lt.getElement(ultimos,i),sort_criteria)
        lt.addLast(sub_lista, ultimos_i)
    return lista_grande, sub_lista, diferent_mag, total_events
```

Para el desarrollo del requerimiento 2, se utilizó el mapa binario ordenado de las magnitudes. El mapa ‘ArbolMagnitudes’ contiene llaves del tipo:

{magnitud: [temblor 1] , [temblor2] … [temblor n]}

Para el desarrollo del requerimiento se itero sobre la lista que contiene los temblores de las magnitudes entre el rango de búsqueda que se obtuvieron usando la función de values. Finalmente se ordenó las listas que contienen las magnitudes necesarias para el requerimiento, pero solamente las 3 primeras y tres últimas.

Entrada	data_structs, min_mag, max_mab
Salidas	Lista_grande, sub_lista, diferents_mag, total_events
Implementado (Sí/No)	SI

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**

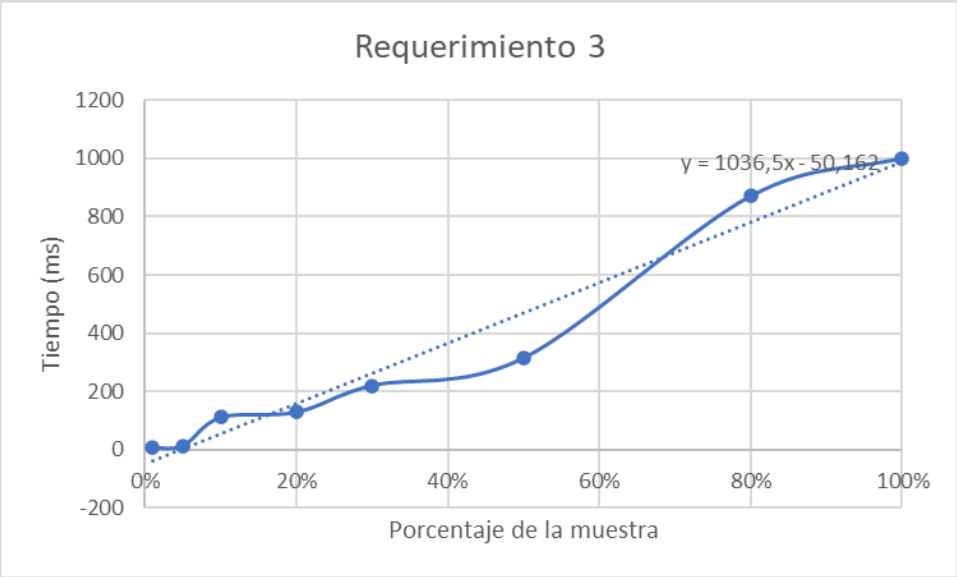
Obtener los temblores dentro de este rango	O(logn)
Ordenar los tres primeros y tres ultimos	O (nlogn)
TOTAL	***O (logn)***

Pruebas Realizadas

Procesadores	intel(R) Core(TM) i3-1005G1 CPU @ 1.20GHz, 1201
Memoria RAM	12 GB

REQ 3		
1%	small	5,82
5%	5 pct	14,22
10%	10 pct	111,21
20%	20 pct	128,98
30%	30 pct	220,23
50%	50 pct	315,21
80%	80 pct	870,43
100%	large	1000,52

Graficas



Análisis

En términos de complejidad este requerimiento es O(logn) debido a la función e obtener los valores dentro de un rango tienen esta complejidad y aunque se ordenan los tres primeros y tres últimos de este rango, no termina fluyendo en la complejidad cuando se trata de muchos datos porque es constante.

Requerimiento 3

Descripción

```python
def req_3(data_structs, mag_min, prof_max):
    """
    Función que soluciona el requerimiento 3
    """
    mag_min = mag_min[:3]
    mag_max = om.maxKey(data_structs['ArbolMagnitudes'])
    lista = om.values(data_structs['ArbolMagnitudes'], mag_min, mag_max)
    prof_max = int(float(prof_max))
    temblores = lt.newList(datastructure='ARRAY_LIST')
    for magnitud in lt.iterator(lista):
        for temblor in lt.iterator(magnitud):
            prof_i = int(float(temblor['depth']))
            if prof_i <= prof_max:
                lt.addLast(temblores, temblor)
    temblores_ordenado = quk.sort(temblores, sort_criteria)
    return temblores_ordenado
```

Para el desarrollo del requerimiento 3, se utilizó el mapa binario ordenado de las magnitudes. El mapa 'ArbolMagnitudes' contiene llaves del tipo:

{magnitud: [temblor 1] , [temblor2] … [temblor n]}

Para el desarrollo del requerimiento se itero sobre la lista que contiene los temblores de las magnitudes mayores o iguales a la mínima. Por cada temblor que se iteraba se revisaba que no superaran a la profundidad máxima y se agregaban a una lista. Finalmente se ordenó esta lista que contiene todos los temblores que cumplen con los requerimientos por la fecha.

Entrada	control, magnitude minima, profundidad maxima
Salidas	Temblores ordenados
Implementado (Sí/No)	SI – Andres Felipe Chaparro

Análisis de complejidad

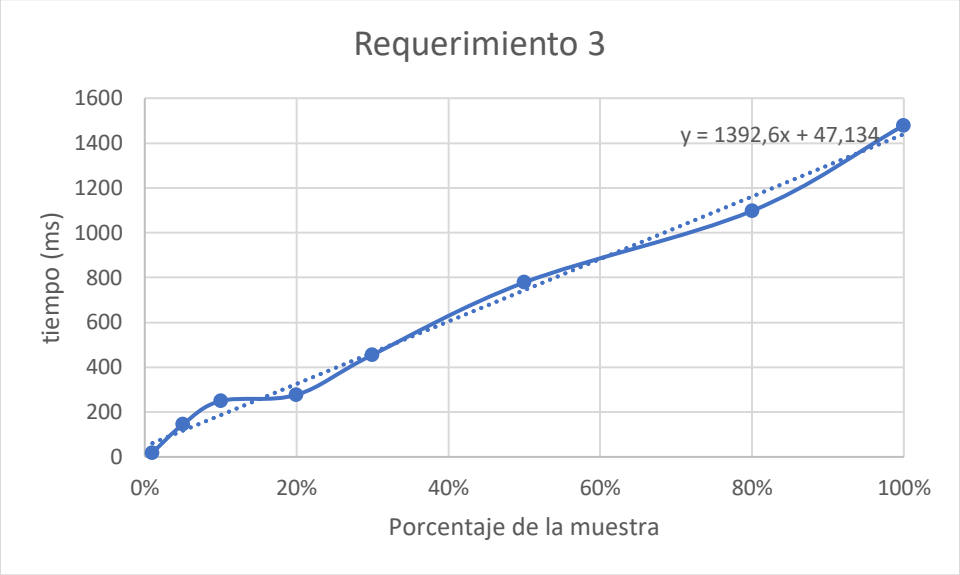
Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Iteración por temblor que sea mayor a la magnitud minima	O(logn)
TOTAL	***O(logn)***

Pruebas Realizadas

Procesadores	intel(R) Core(TM) i3-1005G1 CPU @ 1.20GHz, 1201
Memoria RAM	12 GB

REQ 3		
1%	small	18,81
5%	5 pct	144,25
10%	10 pct	249,42
20%	20 pct	277,06
30%	30 pct	455,34
50%	50 pct	778,12
80%	80 pct	1097,19
100%	large	1478,88

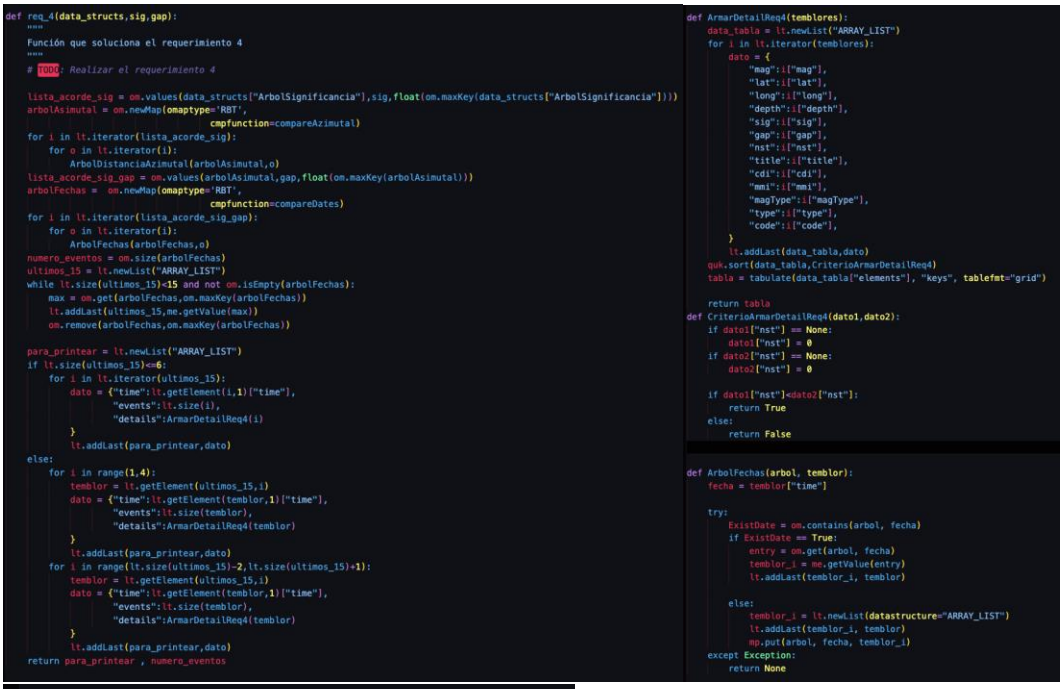


Análisis

Teniendo en cuenta las pruebas realizadas y el análisis de complejidad, podemos concluir que el uso de estructuras de datos como mapas binarios ordenados mejoro significativamente la eficiencia de la función req_3. Por otro lado, el requerimiento tiene una complejidad logarítmica ya que al buscar los datos lo hace como si fuera una búsqueda binaria debido a su estructura y al recorrer estos datos no tiene que iterar todos para encontrarlos.

Requerimiento 4

Descripción



```
def req_4(data_structs,sig,gap):
    """
    Función que soluciona el requerimiento 4
    """
    # TODO: Realizar el requerimiento 4

    lista_acorde_sig = om.values(data_structs["ArbolSignificancia"],sig,float(om.maxKey(data_structs["ArbolSignificancia"])))
    arbolAsimutal = om.newMap(omaptype='RBT',
                              cmpfunction=compareAzimutal)
    for i in lt.iterator(lista_acorde_sig):
        for o in lt.iterator(i):
            ArbolDistanciaAzimutal(arbolAsimutal,o)
    lista_acorde_sig_gap = om.values(arbolAsimutal,gap,float(om.maxKey(arbolAsimutal)))
    arbolFechas =  om.newMap(omaptype='RBT',
                              cmpfunction=compareDates)
    for i in lt.iterator(lista_acorde_sig_gap):
        for o in lt.iterator(i):
            ArbolFechas(arbolFechas,o)
    numero_eventos = om.size(arbolFechas)
    ultimos_15 = lt.newList("ARRAY_LIST")
    while lt.size(ultimos_15)<15 and not om.isEmpty(arbolFechas):
        max = om.get(arbolFechas,om.maxKey(arbolFechas))
        lt.addLast(ultimos_15,me.getValue(max))
        om.remove(arbolFechas,om.maxKey(arbolFechas))

    para_printear = lt.newList("ARRAY_LIST")
    if lt.size(ultimos_15)<=6:
        for i in lt.iterator(ultimos_15):
            dato = {"time":lt.getElement(i,1)["time"],
                    "events":lt.size(i),
                    "details":ArmarDetailReq4(i)
                }
            lt.addLast(para_printear,dato)
    else:
        for i in range(1,4):
            temblor = lt.getElement(ultimos_15,i)
            dato = {"time":lt.getElement(temblor,1)["time"],
                    "events":lt.size(temblor),
                    "details":ArmarDetailReq4(temblor)
                }
            lt.addLast(para_printear,dato)
        for i in range(lt.size(ultimos_15)-2,lt.size(ultimos_15)+1):
            temblor = lt.getElement(ultimos_15,i)
            dato = {"time":lt.getElement(temblor,1)["time"],
                    "events":lt.size(temblor),
                    "details":ArmarDetailReq4(temblor)
                }
            lt.addLast(para_printear,dato)
    return para_printear , numero_eventos
```

```
def ArmarDetailReq4(temblores):
    data_tabla = lt.newList("ARRAY_LIST")
    for i in lt.iterator(temblores):
        dato = {
            "mag":i["mag"],
            "lat":i["lat"],
            "long":i["long"],
            "depth":i["depth"],
            "sig":i["sig"],
            "gap":i["gap"],
            "nst":i["nst"],
            "title":i["title"],
            "cdi":i["cdi"],
            "mmi":i["mmi"],
            "magType":i["magType"],
            "type":i["type"],
            "code":i["code"],
        }
        lt.addLast(data_tabla,dato)
    quk.sort(data_tabla,CriterioArmarDetailReq4)
    tabla = tabulate(data_tabla["elements"], "keys", tablefmt="grid")

    return tabla
def CriterioArmarDetailReq4(dato1,dato2):
    if dato1["nst"] == None:
        dato1["nst"] = 0
    if dato2["nst"] == None:
        dato2["nst"] = 0

    if dato1["nst"]<dato2["nst"]:
        return True
    else:
        return False
```

```
def ArbolFechas(arbol, temblor):
    fecha = temblor["time"]

    try:
        ExistDate = om.contains(arbol, fecha)
        if ExistDate == True:
            entry = om.get(arbol, fecha)
            temblor_i = me.getValue(entry)
            lt.addLast(temblor_i, temblor)

        else:
            temblor_i = lt.newList(datastructure="ARRAY_LIST")
            lt.addLast(temblor_i, temblor)
            mp.put(arbol, fecha, temblor_i)
    except Exception:
        return None
```

```
def ArbolDistanciaAzimutal(arbol, temblor):
    if temblor["gap"] ==None:
        temblor["gap"] = 0
    DistanciaAzimutal = float(temblor["gap"])

    try:
        ExistAz = om.contains(arbol, DistanciaAzimutal)
        if ExistAz == True:
            entry = om.get(arbol, DistanciaAzimutal)
            temblor_i = me.getValue(entry)
        else:
            temblor_i = lt.newList(datastructure="ARRAY_LIST")
            mp.put(arbol, DistanciaAzimutal, temblor_i)
        lt.addLast(temblor_i, temblor)
    except Exception:
        return None
```

Para el desarrollo del requerimiento 4, se utilizó un árbol y se crearon diferentes mapas auxiliares. Primero, el árbol ‘ArbolSignificancia’ (del tipo RBT) que contiene valores acordes a la significancia de cada dato, esto ordenado de menor a mayor siendo la llave mayor la raíz del árbol:

Para el desarrollo del requerimiento se extrajeron los valores con significancia mayor o igual a la ingresada del primer árbol, posteriormente estas se ordenaron en un árbol RBT ordenado hacia menor acorde al valor de la distancia azimutal, de esta se extrajeron los valores con distancia azimutal menor o igual a la ingresada. Por último, estos datos se ingresaron a un árbol RBT ordenado acorde a las fechas, siendo la más reciente la raíz, de esta se extrae y se elimina la raíz 15 veces, con esto tenemos los 15 eventos más recientes para ordenarlos posteriormente en una tabla para imprimir.

Entrada	control, significancia, distancia azimutal
Salidas	para_printear , numero de eventos
Implementado (Sí/No)	SI – Esteban

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Filtrar árbol significancia	O(logn)
Filtrar árbol distancia azimutal	O(logn)
Obtener los 15 datos árbol fechas	O(logn)
TOTAL	***O(logn)***

Pruebas Realizadas

Procesadores	**M2**
Memoria RAM	8 GB

Entrada	**Tiempo (ms)**
small	67,94
5 pct	185,06
10 pct	330,84
20 pct	1225,35
30 pct	1267,43
50 pct	1778,91
80 pct	3298,89
large	4878,42

Análisis

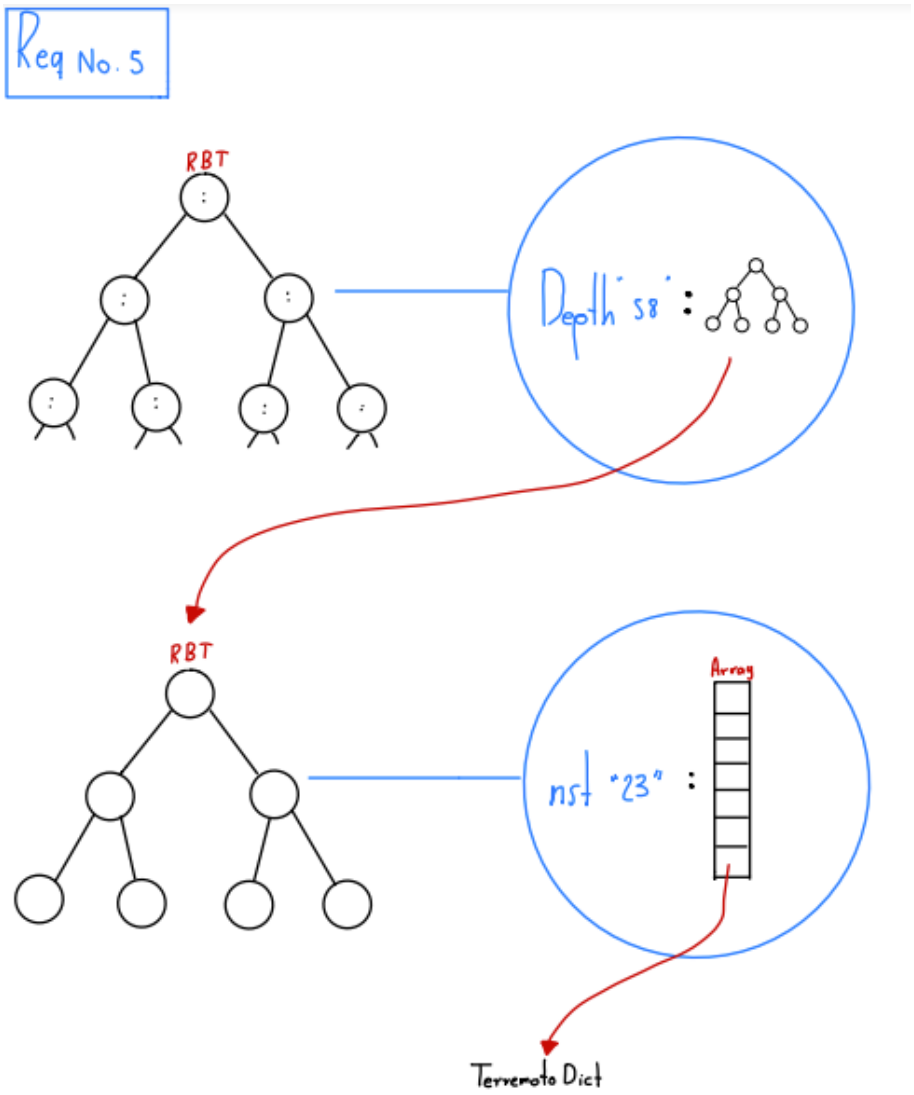
Teniendo en cuenta las pruebas realizadas y el análisis de complejidad, podemos concluir que el uso de estructuras de datos como arboles mejoro significativamente la eficiencia de la función req_4. Por otro lado, el requerimiento tiene una complejidad log n sin embargo no se puede apreciar muy bien en los experimentos realizados.

Requerimiento 5

Descripción

```python
def req_5(data_structs, depthMin, nstMin):
    """
    Función que soluciona el requerimiento 5
    """
    # TODO: Realizar el requerimiento 5

    map = data_structs['depthTree']
    solucion = lt.newList('ARRAY_LIST')


    lstDepths = om.values(map, depthMin, om.maxKey(map) )

    for entry in lt.iterator(lstDepths):
        nstIndex = entry['nstIndex']
        Lstnst = om.values(nstIndex, nstMin, om.maxKey(nstIndex))
        for entrynst in lt.iterator(Lstnst):
            listanst = entrynst['lstearthquakes']
            for terremoto in lt.iterator(listanst):
                lt.addLast(solucion, terremoto)

    sortedsolution =   quk.sort(solucion, sort_criteria_REQ5)

    if lt.size(sortedsolution) > 20:
        FinalSolution = lt.subList(sortedsolution, 1,20)
    else:
        FinalSolution = sortedsolution

    #Creación de la tabla de respuetsas
    if lt.size(FinalSolution)> 6:
        FinalSolution= FirstandALst(FinalSolution)


    SolutionTable = lt.newList('ARRAY_LIST')

    for earthquake in lt.iterator(FinalSolution):
        infoearthquake = {
            'time': earthquake['time'],
            'events': 1
        }

        detailsearthquake = {
            'mag': round(float(earthquake['mag']), 3),
            'lat': round(float(earthquake['lat']), 3),
            'long': round(float(earthquake['long']), 3),
            'depth': round(float(earthquake['depth']), 3),
            'sig': round(float(earthquake['sig']), 3),
            'gap': round(float(earthquake['gap']), 3) if earthquake['gap'] != 'unknown' else earthquake['gap'],
            'nst': round(float(earthquake['nst']), 3),
            'title': earthquake['title'],
            'cdi': round(float(earthquake['cdi']), 3) if earthquake['cdi'] != 'unknown' else earthquake['cdi'],
            'mmi': round(float(earthquake['mmi']), 3) if earthquake['mmi'] != 'unknown' else earthquake['mmi'],
            'magType': earthquake['magType'],
            'type': earthquake['type'],
            'code': earthquake['code']
        }

        infoearthquake['details'] = tabulate([detailsearthquake], headers='keys', tablefmt="grid")
        lt.addLast(SolutionTable, infoearthquake)

    solutionElements = SolutionTable['elements']
    SolutionTableF = tabulate(solutionElements, headers='keys', tablefmt="grid")

    return SolutionTableF, lt.size(solucion)
```

Para el desarrollo del requerimiento 5, se utilizó una estructura de datos de la siguiente forma:



Para obtener la información solicitada, se hicieron varios filtros de información. Primero, con la ayuda de la función values() se obtuvo una lista de mapas que cumplieran con la profundidad solicitada por el usuario. Luego, se volvió a usar la función values() y se obtuvieron las listas que cumpliera tanto con el requisito de profundidad como el de nst. Por último, se iteraron estas listas para añadir cada terremoto a una lista de solución.

Entrada	control, player, Inicio, Final
Salidas	SolutionTableF, lt.size(solucion)
Implementado (Sí/No)	SI – Camilo Sánchez Novoa

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

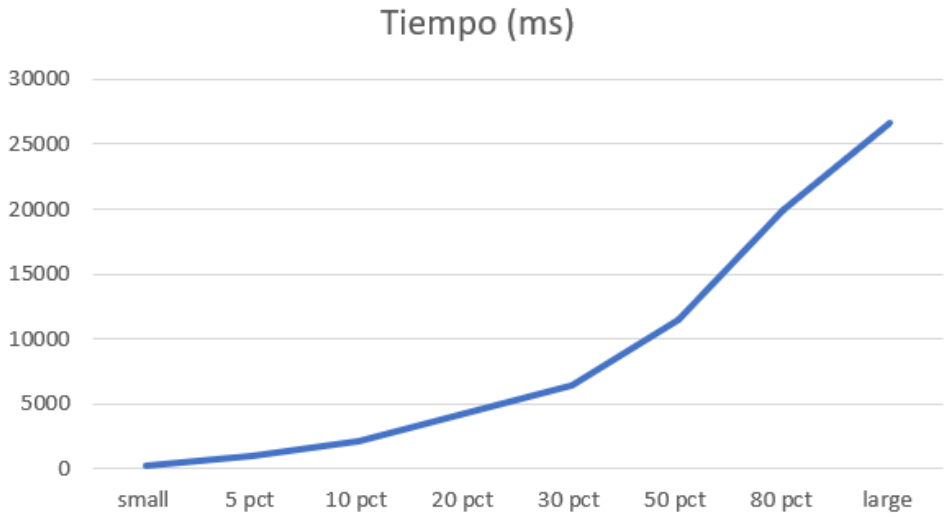
Pasos	Complejidad
Values() depthTree	O(log n)
Iteración: Values() depthTree	O(log n)
Detro de la iteracion: Values() nstIndex	O(log n)
Quick sort	n log n
TOTAL	***O (log n)²***

Pruebas Realizadas

Procesadores	**Intel(R) Core(TM) i7-10750H CPU**
Memoria RAM	32 GB

Entrada	Tiempo (ms)
small	233.3748999999989
5 pct	1034.939099999996
10 pct	2088.0449999999983
20 pct	4252.221300000005
30 pct	6450.821800000005
50 pct	11475.6829
80 pct	19924.2745
large	26662.799600000028

Graficas



Análisis

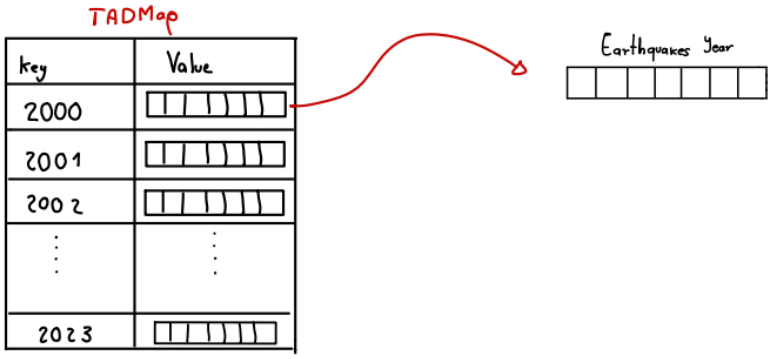
En Cuanto al análisis de este requerimiento, concluimos que, a pesar de no tener una complejidad algorítmica muy eficiente, esta función es efectiva debido a la óptima organización de los datos en los árboles ‘depthTree’ y en ‘nstIndex’. Esto permite acceder a las listas de los datos filtrados con mayor facilidad. Cabe resaltar que, debido a la organización de los datos, cada lista de datos ‘lstearthquake’ no tiene un tamaño significativamente grande.

Requerimiento 6

Descripción

```python
def req_6(data_structs, year, latRef, longRef, radio, n):
    """
    Función que soluciona el requerimiento 6
    """
    # TODO: Realizar el requerimiento 6
    mapYears = data_structs['mapYears']

    entry = mp.get(mapYears, year )
    earthquakesYear = me.getValue(entry)['datos']
    tembloresArea = lt.newList('ARRAY_LIST')

    for temblor in lt.iterator(earthquakesYear):
        if AreaPresent(latRef, longRef, float(temblor['lat']),   float(temblor['long']), radio ):
            temblor['distance'] = haversine(latRef, longRef, float(temblor['lat']),   float(temblor['long']))
            lt.addLast(tembloresArea, temblor)

    sortedAreas = quk.sort(tembloresArea, sort_criteria_REQ6)
    maxArea, posicion= encontrar_mayor_magnitud(sortedAreas)

    Sublista = obtener_sublista(sortedAreas, posicion, n)


    if lt.size(Sublista) <= 6:
        TablaMaxArea, TablaTopN = CreateTables(maxArea, Sublista)
    else:
        List6 = FirstandALst(Sublista)
        TablaMaxArea, TablaTopN = CreateTables(maxArea, List6)


    return TablaMaxArea, TablaTopN, lt.size(tembloresArea), lt.size(Sublista), maxArea
```

Para el requerimiento 6 se utilizó como estructura de datos principal un mapa que tiene como llave un año y como valor una lista con los sismos que ocurrieron ese año, El primer paso consiste en obtener la lista asociada al año que el usuario requiere.



Luego, iterando la lista ‘earthquakesYear’ y aplicando la función ‘haversine’ a cada temblor se obtiene una lista con los terremotos dentro del área especificada por el usuario. Ademas, de esa lista se obtiene el evento más significativo utilizando la función ‘encontrar_mayor magnitud’


```python
def haversine(lat1, lon1, lat2, lon2):
    # Radio de la Tierra en kilómetros
    R = 6371.0

    # Convierte las coordenadas de grados a radianes
    lat1, lon1, lat2, lon2 = map(radians, [lat1, lon1, lat2, lon2])

    # Diferencias de coordenadas
    dlat = lat2 - lat1
    dlon = lon2 - lon1

    # Fórmula de Haversine
    a = sin(dlat / 2)**2 + cos(lat1) * cos(lat2) * sin(dlon / 2)**2
    c = 2 * atan2(sqrt(a), sqrt(1 - a))

    # Distancia en kilómetros
    distance = R * c

    return distance
```

Por último, utilizando la funcion obtener-sublista se obtiene la lista que contiene los elementos anteriores y posteriores al evento máximo.

```python
def obtener_sublista(lista, posicion, n):
    """
    Retorna una nueva sublista con n elementos arriba de la posición,
    el elemento en la posición y n elementos abajo de la posición.
    Si no hay suficientes elementos, se agregan los que haya.
    """

    lista = lista['elements']

    sublista = lt.newList('ARRAY_LIST')

    # Agrega los elementos hacia arriba de la posición
    for i in range(max(0, posicion - n), posicion):
        lt.addLast(sublista, lista[i] )

    # Agrega el elemento en la posición
    lt.addLast(sublista, lista[posicion])

    # Agrega los elementos hacia abajo de la posición
    for i in range(posicion + 1, min(posicion + n + 1, len(lista))):
        lt.addLast(sublista, lista[i])

    return sublista
```

Entrada	data_structs, year, latRef, longRef, radio, n
Salidas	TablaMaxArea, TablaTopN, lt.size(temblоresArea), lt.size(Sublista), maxArea
Implementado (Sí/No)	SI

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

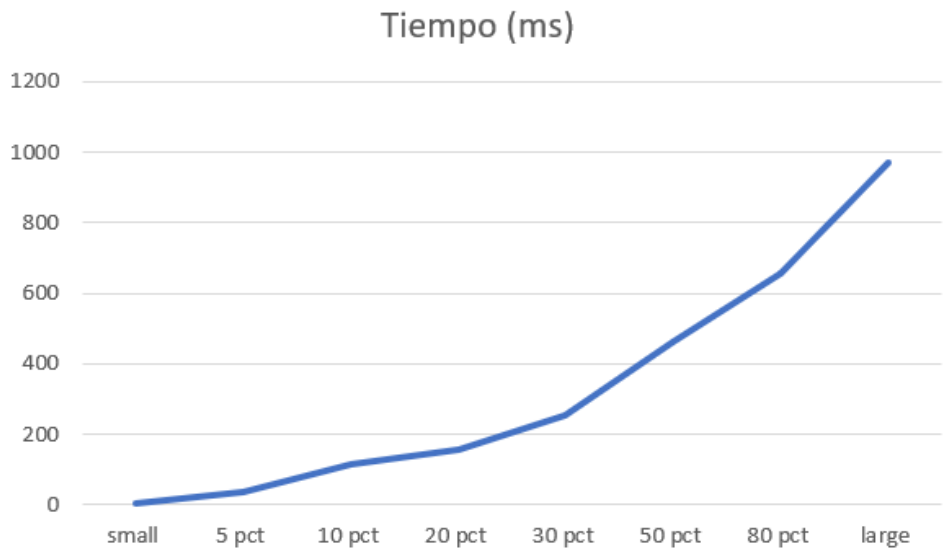
Pasos	**Complejidad**
Función getValue(entry)	O (1)
iterator(earthquakesYear)	O(n)
AreaPresent / Haversine	O (1)
AddLast	O (1)
QuickSort	O (n log n)
Encontrar mayor magnitud	O(n)
TOTAL	***O (n log n)***

Pruebas Realizadas

Procesadores	**Intel(R) Core(TM) i7-10750H CPU**
Memoria RAM	32 GB

Entrada	**Tiempo (ms)**
small	6.130299999997078
5 pct	38.428500000001804
10 pct	112.72960000000239
20 pct	155.05509999999776
30 pct	254.2441000000108
50 pct	461.0024999999732
80 pct	656.9710999999952
large	968.9361000000499

Graficas



Análisis

Para el requerimiento 6 no se utilizar arboles debido a que después de hacer la planeación de ejecución, se encontró que sería más eficiente en términos de complejidad buscar una llave única en un mapa que en un árbol. Esto ya que nuestro mapa tiene años como llaves y el usuario ingresa un año como parámetro. Por otro lado, se obtuvo una función eficiente debido al correcto uso de funciones auxiliares como: Haversine, obtener_mayor_magnitud y obtener_sublista.

Requerimiento 7

Descripción

```python
def req_7(data_structs, año, title, prop, bins):
    """
    Función que soluciona el requerimiento 7
    """
    bins_considerar = bins
    fecha_inicial = datetime.strptime(año, '%Y')
    fecha_final= datetime.strptime(str(int(año)+1), '%Y')
    fecha_inicial = fecha_inicial.replace(month=1, day=1, hour=0, minute=0)
    fecha_inicial = om.ceiling(data_structs['ArbolFechas'], fecha_inicial)
    fecha_final = fecha_final.replace(month=1, day=1, hour=0, minute=0)
    fecha_final = om.floor(data_structs['ArbolFechas'], fecha_final)
    lista_temblores = om.values(data_structs['ArbolFechas'], fecha_inicial, fecha_final)
    lista_filtrada = lt.newList(datastructure='ARRAY_LIST')
    max_prop = -1000
    min_prop = 1000
    for temblor_i in lt.iterator(lista_temblores):
        size = lt.size(temblor_i)
        if size > 1:
            for temblor_e in lt.iterator(temblor_i):
                title_e = temblor_e['title']
                match = re.search(r', (.+)$', title_e)
                if match:
                    region = match.group(1)
                prop_e = temblor_e[prop]
                if region == title and prop_e != "":
                    lt.addLast(lista_filtrada, temblor_e)
                    prop_e = float(prop_e)
                    if prop_e > max_prop:
                        max_prop = prop_e
                    if prop_e < min_prop:
                        min_prop = prop_e
        else:
            title_i = temblor_i['elements'][0]['title']
            match = re.search(r', (.+)$', title_i)
            region = ''
            if match:
                region = match.group(1)
            prop_i = temblor_i['elements'][0][prop]
            if region == title and prop_i != "":
                lt.addLast(lista_filtrada, temblor_i['elements'][0])
                prop_i = float(prop_i)
                if prop_i > max_prop:
                    max_prop = prop_i
                if prop_i < min_prop:
                    min_prop = prop_i
    tamaño = lt.size(lista_filtrada)
    dif = max_prop - min_prop
    rango_bins = round(dif/int(bins),2)
    if prop == 'mag':
        quk.sort(lista_filtrada, sort_criteria_mag)
    elif prop == 'sig':
        quk.sort(lista_filtrada, sort_criteria_sig)
    elif prop =='depth':
        quk.sort(lista_filtrada, sort_criteria_depht)

```

```
    contador_rango = min_prop
    datos = lt.newList("ARRAY_LIST")
    for i in lt.iterator(lista_filtrada):
        rango = str(round(contador_rango,2))+"-"+str(round(contador_rango+rango_bins,2))
        if float(i[prop]) >=round(contador_rango,2) and float(i[prop]) <=round(contador_rango+rango_bins,2):
            lt.addLast(datos,rango)
        else:
            contador_rango+=rango_bins
            rango = str(round(contador_rango,2))+"-"+str(round(contador_rango+rango_bins,2))
            lt.addLast(datos,rango)

    datos_tabla = datos["elements"]

    quk.sort(lista_filtrada,sort_criteria)
    tabla_detalles = lt.newList("ARRAY_LIST")
    if lt.size(lista_filtrada)>6:
        for i in range(1,4):
            detalles = lt.getElement(lista_filtrada,i)
            dato = {
                "time":detalles["time"],
                "lat":detalles["lat"],
                "long":detalles["long"],
                "title":detalles["title"],
                "code":detalles["code"],
                prop:detalles[prop],
            }
            lt.addLast(tabla_detalles,dato)
        for i in range(lt.size(lista_filtrada)-2,lt.size(lista_filtrada)+1):
            detalles = lt.getElement(lista_filtrada,i)
            dato = {
                "time":detalles["time"],
                "lat":detalles["lat"],
                "long":detalles["long"],
                "title":detalles["title"],
                "code":detalles["code"],
                prop:detalles[prop],
            }
            lt.addLast(tabla_detalles,dato)
    else:
        for detalles in lt.iterator(lista_filtrada):
            dato = {
                "time":detalles["time"],
                "lat":detalles["lat"],
                "long":detalles["long"],
                "title":detalles["title"],
                "code":detalles["code"],
                prop:detalles[prop],
            }
            lt.addLast(tabla_detalles,dato)
    histograma = plt.hist(datos_tabla, bins = int(bins_considerar), color = "darkseagreen",edgecolor = "black",linewidth = 2, rwidth=0.8)



    tabla_texto = tabulate(tabla_detalles["elements"], headers='keys', tablefmt='fancy_grid')



    return tamaño , histograma , tabla_texto
```

Como primer paso se filtran los temblores que cumplen con el requisito del año, con el requisito de región y con la propiedad de conteo (magnitud, profundidad, significancia), se ordenan en base a esto último. Posteriormente esta lista es pasada por una función que define entre que rango de valores se encuentra (la longitud de estos rangos depende de los bins que elija el usuario), esto se añade a una lista que a continuación se usa para generar el histograma solicitado. A continuación, se pasa la misma lista por un iterador que toma los primeros 3 y los últimos 3 datos de interés, en caso de ser de un tamaño menor a 6 se usan todos los datos.

Entrada	Control, año, title, prop, bins
Salidas	Histograma, tabla_texto, tamaño
Implementado (Sí/No)	SI

Análisis de complejidad

Pasos	**Complejidad**
Filtrado lista inicial	O (log n)
Toma de Datos para histograma	O (n)
Toma de Datos para tabla	O (6)

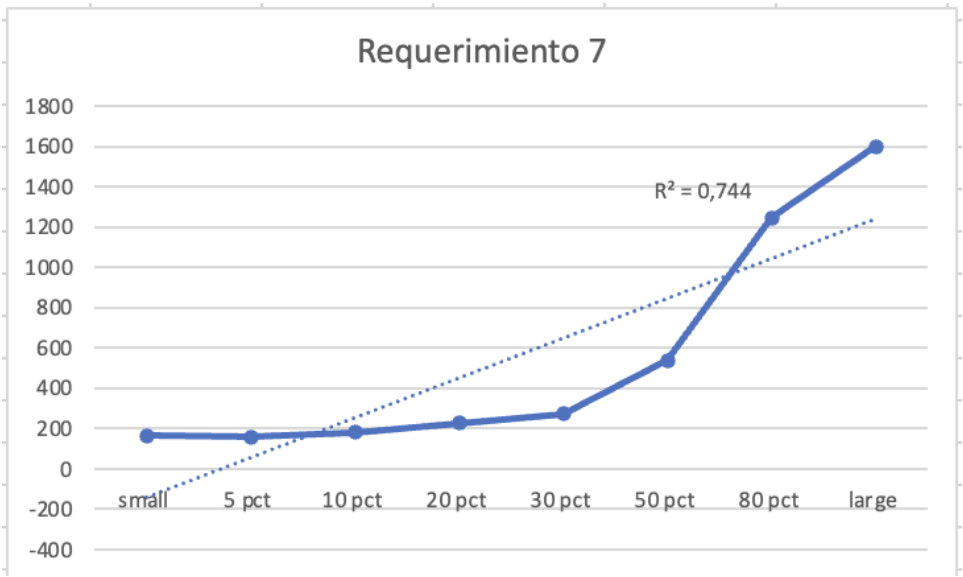
TOTAL	***O (n)***

Pruebas Realizadas

Procesadores	**M2**
Memoria RAM	8 GB

Entrada	**Tiempo (ms)**
small	165.41
5 pct	158.54
10 pct	182.03
20 pct	228.83
30 pct	273.43
50 pct	539.03
80 pct	1246.92
large	1600.37

Análisis



El ciclo tiene una complejidad lineal, en el algoritmo lo más tardado fue el segundo recorrido que busca en toda la lista filtrada, sin embargo, esto no se ve en la gráfica y puede deberse un aumento de datos que no es constante, por ejemplo, si se usaron los mismos ingresos para todos los tamaños de archivo, pero las coincidencias aumentaron considerablemente después del 50% es normal que la gráfica presente irregularidades.