ANÁLISIS DEL RETO 3

1. David Elias Forero Cobos, de.foreroc1@uniandes.edu.co, 202310499.

2. Alejandro Garcia Rojas, a.garciar23@uniandes.edu.co, 202122516.

3. Pablo Andrés Sebastián Parra Céspedes, p.parrac@uniandes.edu.co, 202310768.

Requerimiento No.1

Descripción

```python
def req_1(dataStructs, date1, date2):
    events = lt.newList()
    filtered = lt.newList('ARRAY_LIST')
    dateTree = dataStructs["seismicEventsByDate"]
    initialDate = datetime.strptime(date1,"%Y-%m-%dT%H:%M")
    finalDate = datetime.strptime(date2,"%Y-%m-%dT%H:%M")
    dates = om.keys(dateTree, initialDate, finalDate)
    metaData = {"totalDates": lt.size(dates), "totalSeismicEvents": 0}
    for date in lt.iterator(dates):
        seismicEvents = me.getValue(om.get(dateTree, date))
        for seismicEvent in lt.iterator(seismicEvents):
            lt.addLast(events, seismicEvent)
        date = datetime.strftime(date, "%Y-%m-%dT%H:%M:%S.%fZ")
        datum = {'time': date, 'events': lt.size(seismicEvents), 'details': sortData(seismicEvents, compareByDateDescending)}
        lt.addLast(filtered, datum)
        metaData['totalSeismicEvents'] += lt.size(seismicEvents)
    sortData(filtered, compareByDateDescending)
    metaData["map"] = lt.size(events) > 0
    metaData["path"] = "req_1"
    req_8(events, "req_1")
    return filtered, metaData
```

La función `req_1` procesa y filtra eventos sísmicos en un rango de fechas dado, organiza los eventos por fecha en orden descendente y recopila metadatos relacionados con el número total de eventos y fechas. La salida consiste en dos conjuntos de datos: una lista de eventos sísmicos con detalles específicos y un conjunto de metadatos que incluye estadísticas generales y un indicador de si hay datos para visualizar en un mapa. Además, la función realiza una operación de ordenamiento final y ejecuta una acción adicional con la función `req_8`.

Entrada	El requerimiento 1 pide como parámetros de entradas los siguientes dos valores: - La estructura de datos con los eventos a consultar. - La fecha inicial del intervalo (en formato "%Y-%m-%dT%H:%M"). - La fecha final del intervalo (en formato "%Y-%m-%dT%H:%M").
Salida	El requerimiento 1 retorna como salida:

	- El número total de fechas encontradas. - El número total de eventos encontrados en esas fechas. - Una tabla con los primeros tres y los últimos tres eventos encontrados durante el rango de fechas ingresadas como parámetros.
Implementado	Sí, el requerimiento fue implementado, desarrollado y completado por el grupo No.1 de la sección No.3 del curso de Estructuras de Datos y Algoritmos.

Análisis de complejidad

Pasos	**Complejidad**
def req_1(dataStructs, date1, date2):	O(1)
events = lt.newList()	O(1)
filtered = lt.newList('ARRAY_LIST')	O(1)
dateTree = dataStructs["seismicEventsByDate"]	O(1)
initialDate = datetime.strptime(date1,"%Y-%m-%dT%H:%M")	O(1)
finalDate = datetime.strptime(date2,"%Y-%m-%dT%H:%M")	O(1)
dates = om.keys(dateTree, initialDate, finalDate)	O(logN)
metaData = {"totalDates": lt.size(dates), "totalSeismicEvents": 0}	O(1)
for date in lt.iterator(dates):	O(M) (donde M<=N)
seismicEvents = me.getValue(om.get(dateTree, date))	O(logN)
for seismicEvent in lt.iterator(seismicEvents):	O(L) (donde L<=M)
lt.addLast(events, seismicEvent)	O(1)
date = datetime.strftime(date, "%Y-%m-%dT%H:%M:%S.%fZ")	O(1)

datum = {'time': date, 'events': lt.size(seismicEvents), 'details': sortData(seismicEvents, compareByDateDescending)}	O(M^(3/2)) (donde M<=N)
lt.addLast(filtered, datum)	O(1)
metaData['totalSeismicEvents'] += lt.size(seismicEvents)	O(1)
sortData(filtered, compareByDateDescending)	O(L^(3/2)) (donde L<=M)
metaData["map"] = lt.size(events) > 0	O(1)
metaData["path"] = "req_1"	O(1)
req_8(events, "req_1")	O(M*L)
return filtered, metaData	O(1)
Total	O(N^2)

Pruebas Realizadas

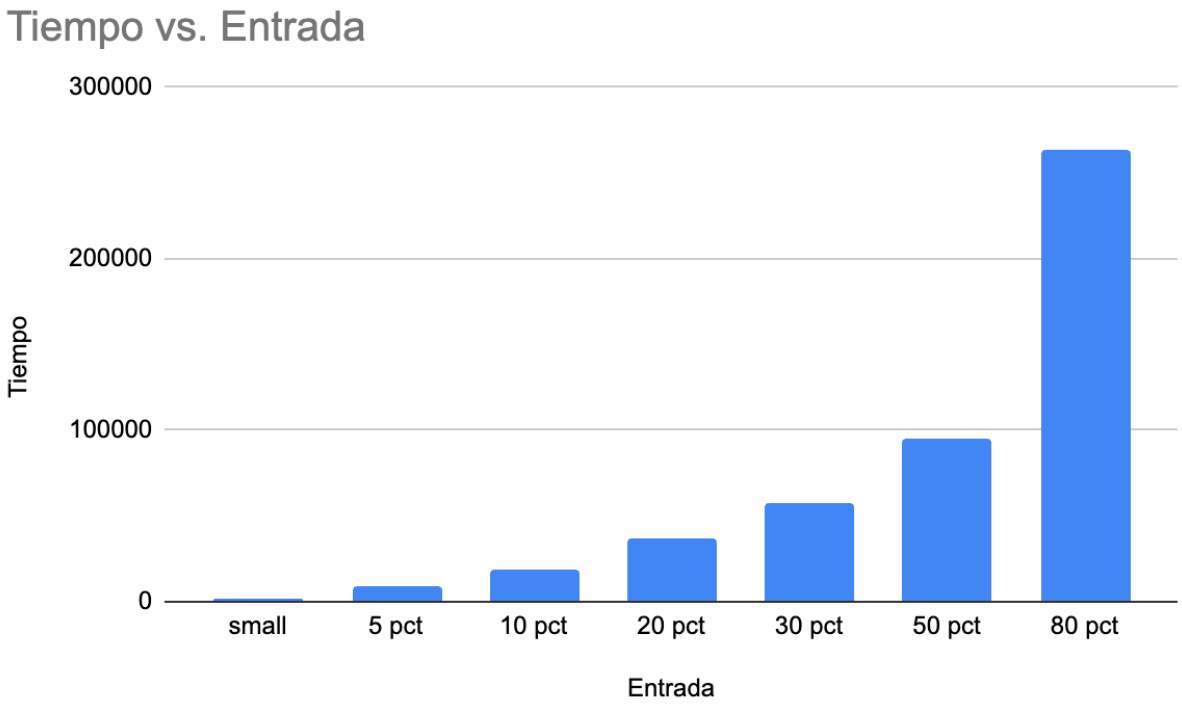
Las pruebas se realizaron utilizando un ordenador portátil con un nivel de batería del 50%, con la única aplicación en funcionamiento en ese momento siendo VSCode. Los datos se adquirieron de manera externa para evitar la necesidad de abrir otras aplicaciones durante la ejecución del programa.

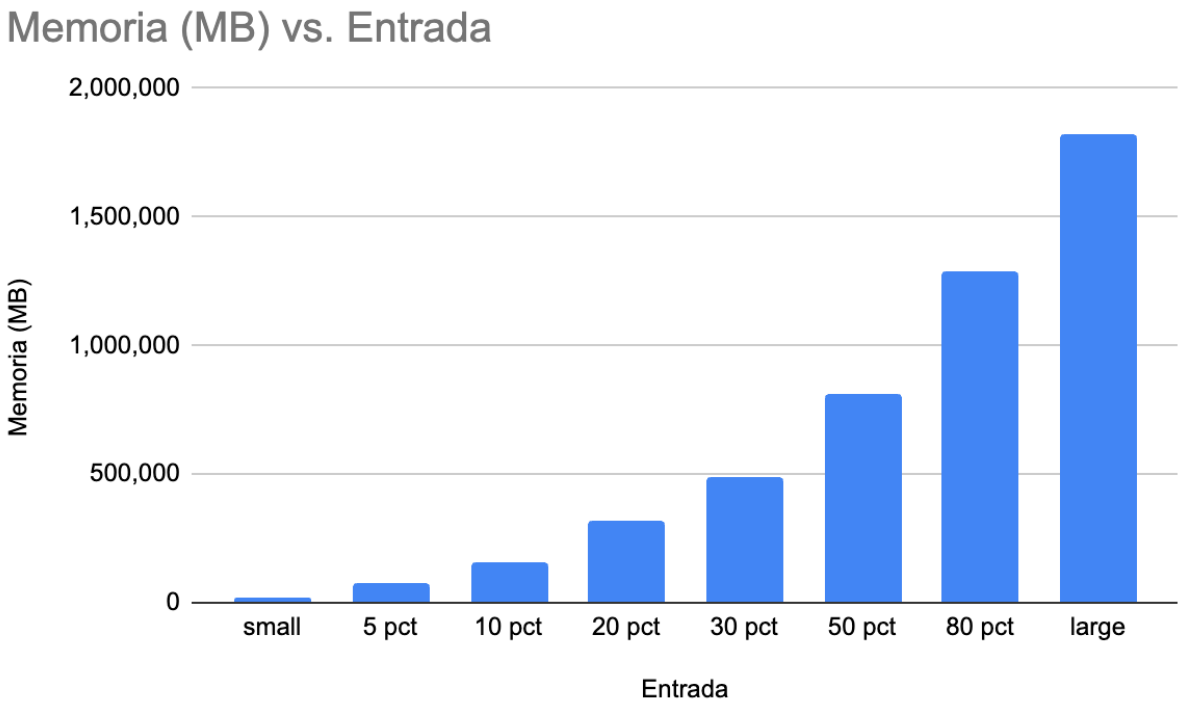
Pruebas realizadas con el computador de David Elias Forero Cobos:

Procesadores	**Apple M1**
Memoria RAM	8 GB
Sistema Operativo	macOS Ventura Versión 13.5.2

Entrada	**Tiempo (s)**	**Memoria (MB)**
small	2,554	16,471
5 pct	14,327	79,544
10 pct	30,645	160,491
20 pct	64,768	321,852
30 pct	99,992	487,019
50 pct	170,629	810,429
80 pct	298,359	1289,397
large	366,366	1615,987

Graficas





Análisis

La implementación correspondiente al requerimiento 1, en su caso más desfavorable, ostenta una complejidad algorítmica de O(N^2). La representación gráfica del rendimiento en términos de tiempo y memoria, al escalar la cantidad de datos de entrada, ha evidenciado una tendencia de crecimiento exponencial en ambas métricas. Esta constatación permite corroborar que la complejidad proyectada del código se refleja fielmente en la dinámica de respuesta del requerimiento frente al incremento sucesivo de la carga de datos procesados.

Requerimiento 2

Descripción

```
def req_2(dataStructs, inferiorMagLimit, superiorMagLimit):
    """
    Función que soluciona el requerimiento 2
    """
    # TODO: Realizar el requerimiento 2
    events = lt.newList()
    filtered = lt.newList("ARRAY_LIST")
    magTree = dataStructs["seismicEventsByMag"]
    magKeys = om.keys(magTree, inferiorMagLimit, superiorMagLimit)
    metaData = {"totalMagnitudes": lt.size(magKeys) ,"totalSeismicEvents": 0}
    for magKey in lt.iterator(magKeys):
        seismicEvents = me.getValue(om.get(magTree, magKey))["seismicEvents"]
        for seismicEvent in lt.iterator(seismicEvents):
            lt.addLast(events, seismicEvent)
        datum = {"mag": magKey, "events": lt.size(seismicEvents), "details": sortData(seismicEvents, compareByDateDescending)}
        lt.addLast(filtered, datum)
        metaData["totalSeismicEvents"] += lt.size(seismicEvents)
    sortData(filtered, compareByMag)
    metaData["map"] = lt.size(events) > 0
    metaData["path"] = "req_2"
    req_8(events, "req_2")
    return filtered, metaData
```

La función `req_2` filtra eventos sísmicos dentro de un rango de magnitudes, los agrupa y ordena por magnitud y fecha, y recopila metadatos que incluyen el conteo de eventos y magnitudes. Finalmente, realiza un procesamiento adicional con la función `req_8` y devuelve una lista organizada de eventos sísmicos junto con un resumen de metadatos, los cuales indican si los datos son suficientes para una representación cartográfica y la identificación del proceso dentro del sistema.

Entrada	El requerimiento 2 pide como parámetros de entradas los siguientes dos valores: - La estructura de datos con los eventos a consultar. - El límite inferior de la magnitud del intervalo (en formato decimal). - El límite superior de la magnitud del intervalo (en formato decimal).
Salida	El requerimiento 2 retorna como salida: - El número total de eventos sísmicos encontrados entre las magnitudes indicadas - El total de días diferentes - Una tabla con los primeros tres y los últimos tres eventos encontrados durante el rango de magnitudes ingresado como parámetros.

Implementado	Sí, el requerimiento fue implementado, desarrollado y completado por el grupo No.1 de la sección No.3 del curso de Estructuras de Datos y Algoritmos.

Análisis de complejidad

Pasos	**Complejidad**
def req_2(dataStructs, inferiorMagLimit, superiorMagLimit):	O(1)
events = lt.newList()	O(1)
filtered = lt.newList("ARRAY_LIST")	O(1)
magTree = dataStructs["seismicEventsByMag"]	O(1)
magKeys = om.keys(magTree, inferiorMagLimit, superiorMagLimit)	O(Log N+M)(M<=N)
metaData = {"totalMagnitudes": lt.size(magKeys) ,"totalSeismicEvents": 0}	O(1)
for magKey in lt.iterator(magKeys):	O(M)
seismicEvents = me.getValue(om.get(magTree, magKey))["seismicEvents"]	O(1)
for seismicEvent in lt.iterator(seismicEvents):	O(L) (L<=M
lt.addLast(events, seismicEvent)	O(1)
datum = {"mag": magKey, "events": lt.size(seismicEvents), "details": sortData(seismicEvents, compareByDateDescending)}	O(L^(3/2)) (donde L<=M)
lt.addLast(filtered, datum)	O(1)
metaData["totalSeismicEvents"] += lt.size(seismicEvents)	O(1)
sortData(filtered, compareByMag)	O((L^(3/2))
metaData["map"] = lt.size(events) > 0	O(1)
metaData["path"] = "req_2"	O(1)

req_8(events, "req_2")	O(M*L)
return filtered, metaData	O(1)
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).

Procesadores	**Apple M1**
Memoria RAM	8 GB
Sistema Operativo	macOS Ventura Versión 13.5.2

Entrada	**Tiempo (s)**	**Memoria**
small	7807	88789
5 pct	43667	432753
10 pct		
20 pct		
30 pct		
50 pct		
80 pct		
large		

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

Descripción

```python
def req_3(dataStructs, minMag, maxDepth):
    """
    Función que soluciona el requerimiento 3
    """
    # TODO: Realizar el requerimiento 3
    events = lt.newList()
    filtered = lt.newList()
    magTree = dataStructs["seismicEventsByMag"]
    magNodes = om.values(magTree, minMag, om.maxKey(magTree))
    dateMap = mp.newMap()
    metaData = {"totalEventsBetweenDates": 0}
    for magNode in lt.iterator(magNodes):
        depthTree = magNode["seismicEventsByDepth"]
        depthLists = om.values(depthTree, om.minKey(depthTree), maxDepth)
        for depthList in lt.iterator(depthLists):
            for seismicEvent in lt.iterator(depthList):
                addEventToSimpleMap(dateMap, seismicEvent, "time")
                lt.addLast(events, seismicEvent)
    for dateKey in lt.iterator(mp.keySet(dateMap)):
        dateList = me.getValue(mp.get(dateMap, dateKey))
        metaData["totalEventsBetweenDates"] += lt.size(dateList)
        datum = {"time": dateKey, "events": lt.size(dateList), "details": dateList}
        lt.addLast(filtered, datum)
    metaData["totalDifferentDates"] = mp.size(dateMap)
    sortData(filtered, compareByNumber)
    metaData["map"] = lt.size(events) > 0
    metaData["path"] = "req_3"
    req_8(events, "req_3")
    return getNData(filtered, 10), metaData
```

La función `req_3` identifica y organiza eventos sísmicos que superan una magnitud mínima y se encuentran a una profundidad no mayor que un límite máximo, agrupándolos por fecha. Genera una lista de los eventos y metadatos que incluyen conteos totales y distintos, y ordena los eventos para un procesamiento adicional mediante `req_8`. La función devuelve los diez eventos más relevantes y metadatos que indican la disponibilidad de datos para mapeo y la ruta del proceso dentro del sistema.

Entrada	El requerimiento 3 pide como parámetros de entradas los siguientes dos valores: - La estructura de datos con los eventos a consultar. - La magnitud mínima a consultar. - La profundidad màxima a consultar.
Salida	El requerimiento 3 produce un conjunto de metadatos que incluye el recuento total de las distintas fechas identificadas y la cantidad de eventos sísmicos registrados en esas fechas. Adicionalmente, proporciona una tabla

	que destaca las tres primeras y tres últimas fechas en orden cronológico que satisfacen ciertos criterios preestablecidos de magnitud mínima y profundidad máxima.
Implementado	Sí, el requerimiento fue implementado, desarrollado y completado por David Elias Forero Cobos, código 202310499, del grupo No.1 de la sección No.3 del curso de Estructuras de Datos y Algoritmos.

Análisis de complejidad

Pasos	**Complejidad**
def req_3(dataStructs, minMag, maxDepth):	O(1)
events = lt.newList()	O(1)
filtered = lt.newList()	O(1)
magTree = dataStructs["seismicEventsByMag"]	O(1)
magNodes = om.values(magTree, minMag, om.maxKey(magTree))	O(log(N+M)) (donde M<=N)
dateMap = mp.newMap()	O(1)
metaData = {"totalEventsBetweenDates": 0}	O(1)
for magNode in lt.iterator(magNodes):	O(M) (donde M<=N)
depthTree = magNode["seismicEventsByDepth"]	O(1)
depthLists = om.values(depthTree, om.minKey(depthTree), maxDepth)	O(log(M+L)) (donde L<=M)
for depthList in lt.iterator(depthLists):	O(L) (donde L<=M)
for seismicEvent in lt.iterator(depthList):	O(K) (donde K<=L)
addEventToSimpleMap(dateMap, seismicEvent, "time")	O(1)
lt.addLast(events, seismicEvent)	O(1)
for dateKey in lt.iterator(mp.keySet(dateMap)):	O(K) (donde K<=L)
dateList = me.getValue(mp.get(dateMap, dateKey))	O(1)

metaData["totalEventsBetweenDates"] += lt.size(dateList)	O(1)
datum = {"time": dateKey, "events": lt.size(dateList), "details": dateList}	O(1)
lt.addLast(filtered, datum)	O(1)
metaData["totalDifferentDates"] = mp.size(dateMap)	O(1)
sortData(filtered, compareByNumber)	O(K^(3/2)) (donde K<=L)
metaData["map"] = lt.size(events) > 0	O(1)
metaData["path"] = "req_3"	O(1)
req_8(events, "req_3")	O(K) (donde K<=L)
return getNData(filtered, 10), metaData	O(K)
Total	O(M*L*K)

Pruebas Realizadas

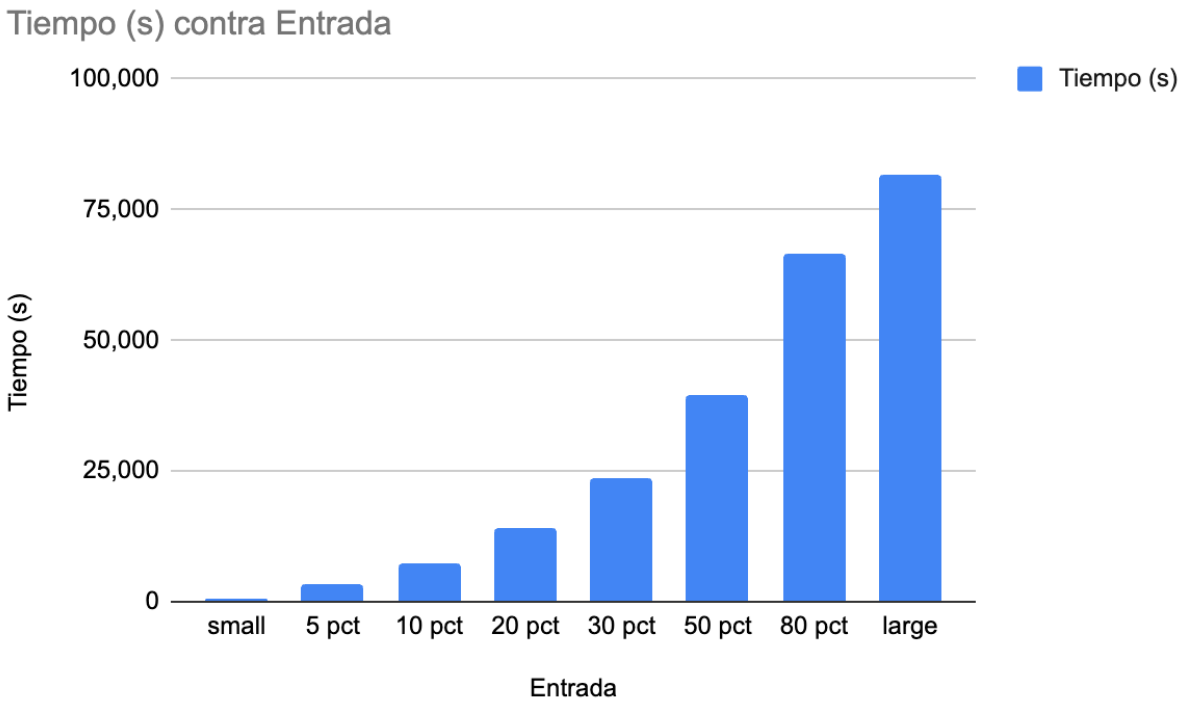
Las pruebas se realizaron utilizando un ordenador portátil con un nivel de batería del 50%, con la única aplicación en funcionamiento en ese momento siendo VSCode. Los datos se adquirieron de manera externa para evitar la necesidad de abrir otras aplicaciones durante la ejecución del programa.

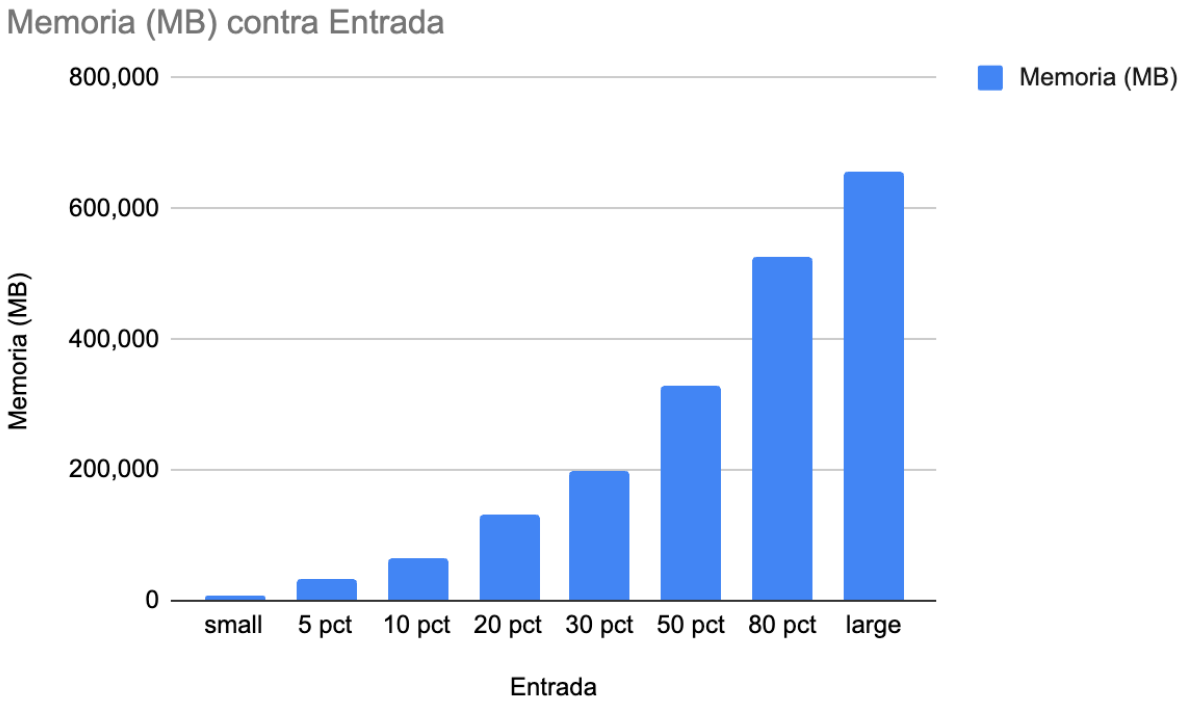
Pruebas realizadas con el computador de David Elias Forero Cobos:

Procesadores	**Apple M1**
Memoria RAM	8 GB
Sistema Operativo	macOS Ventura Versión 13.5.2

Entrada	**Tiempo (s)**	**Memoria (MB)**
small	0,689	6,711
5 pct	3,490	32,909
10 pct	7,511	64,578
20 pct	14,242	131,920
30 pct	23,538	197,742
50 pct	39,702	328,326
80 pct	66,681	525,771
large	81,704	656,618

Graficas





Análisis

La complejidad computacional de la implementación del requerimiento 3 se ha determinado en O(M*L*K), donde K es inferior a L, L es a su vez inferior a M, y M es menor que N. Esto implica que, en el escenario más exigente, K, L y M alcanzarían el valor de N, conduciendo a una complejidad teórica de O(N^3). Los gráficos derivados de nuestras pruebas experimentales corroboran esta evaluación, evidenciando un crecimiento proporcional al aumento del volumen de datos, lo cual es consistente con una progresión exponencial predicha por la complejidad O(N^3).

Requerimiento 4

Descripción

```python
def req_4(dataStructs, minSig, maxGap):
    """
    Función que soluciona el requerimiento 4
    """
    # TODO: Realizar el requerimiento 4
    events = lt.newList()
    filtered = lt.newList()
    sigTree = dataStructs["seismicEventsBySig"]
    sigNodes = om.values(sigTree, minSig, om.maxKey(sigTree))
    dateMap = mp.newMap()
    metaData = {"totalEventsBetweenDates": 0}
    for sigNode in lt.iterator(sigNodes):
        gapTree = sigNode["seismicEventsByGap"]
        gapLists = om.values(gapTree, om.minKey(gapTree), maxGap)
        for gapList in lt.iterator(gapLists):
            for seismicEvent in lt.iterator(gapList):
                addEventToSimpleMap(dateMap, seismicEvent, "time")
                lt.addLast(events, seismicEvent)
    for dateKey in lt.iterator(mp.keySet(dateMap)):
        dateList = me.getValue(mp.get(dateMap, dateKey))
        metaData["totalEventsBetweenDates"] += lt.size(dateList)
        datum = {"time": dateKey, "events": lt.size(dateList), "details": dateList}
        lt.addLast(filtered, datum)
    metaData["totalDifferentDates"] = mp.size(dateMap)
    metaData["map"] = lt.size(events) > 0
    metaData["path"] = "req_4"
    sortData(filtered, compareByNumber)
    req_8(events, "req_4")
    return getNData(filtered, 15), metaData
```

La función `req_4` busca eventos sísmicos con ciertos niveles de importancia y brechas angulares, los organiza por fecha y recopila metadatos sobre la cantidad y distribución de estos eventos. Después de un proceso de ordenación y la llamada a `req_8` para procesos adicionales, la función devuelve una lista con los 15 eventos más notables y metadatos que reflejan la capacidad de mapeo de los datos y la identificación del proceso en el sistema.

Entrada	
Salida	
Implementado	Sí, el requerimiento fue implementado, desarrollado y completado por Alejandro García Rojas, código 202122516, del grupo No.1 de la sección No.3 del curso de Estructuras de Datos y Algoritmos.

Análisis de complejidad

Pasos	**Complejidad**
def req_4(dataStructs, minSig, maxGap):	O(1)
events = lt.newList()	O(1)
filtered = lt.newList()	O(1)
sigTree = dataStructs["seismicEventsBySig"]	O(1)
sigNodes = om.values(sigTree, minSig, om.maxKey(sigTree))	O(Log N + M) (M<=N)
dateMap = mp.newMap()	O(1)
metaData = {"totalEventsBetweenDates": 0}	O(1)
for sigNode in lt.iterator(sigNodes):	O(M)
gapTree = sigNode["seismicEventsByGap"]	O(1)
gapLists = om.values(gapTree, om.minKey(gapTree), maxGap)	O(Log M + P) (P<=M)
for gapList in lt.iterator(gapLists):	O(P)
for seismicEvent in lt.iterator(gapList):	O(K) (K<=P)
addEventToSimpleMap(dateMap, seismicEvent, "time")	O(1)
lt.addLast(events, seismicEvent)	O(1)
for dateKey in lt.iterator(mp.keySet(dateMap)):	O(K)
dateList = me.getValue(mp.get(dateMap, dateKey))	O(1)

metaData["totalEventsBetweenDates"] += lt.size(dateList)	O(1)
datum = {"time": dateKey, "events": lt.size(dateList), "details": dateList}	O(1)
lt.addLast(filtered, datum)	O(1)
metaData["totalDifferentDates"] = mp.size(dateMap)	O(1)
metaData["map"] = lt.size(events) > 0	O(1)
metaData["path"] = "req_4"	O(1)
sortData(filtered, compareByNumber)	O(K^(3/2))
req_8(events, "req_4")	O(K)
return getNData(filtered, 15), metaData	O(K)
Total	O(M*P*K)

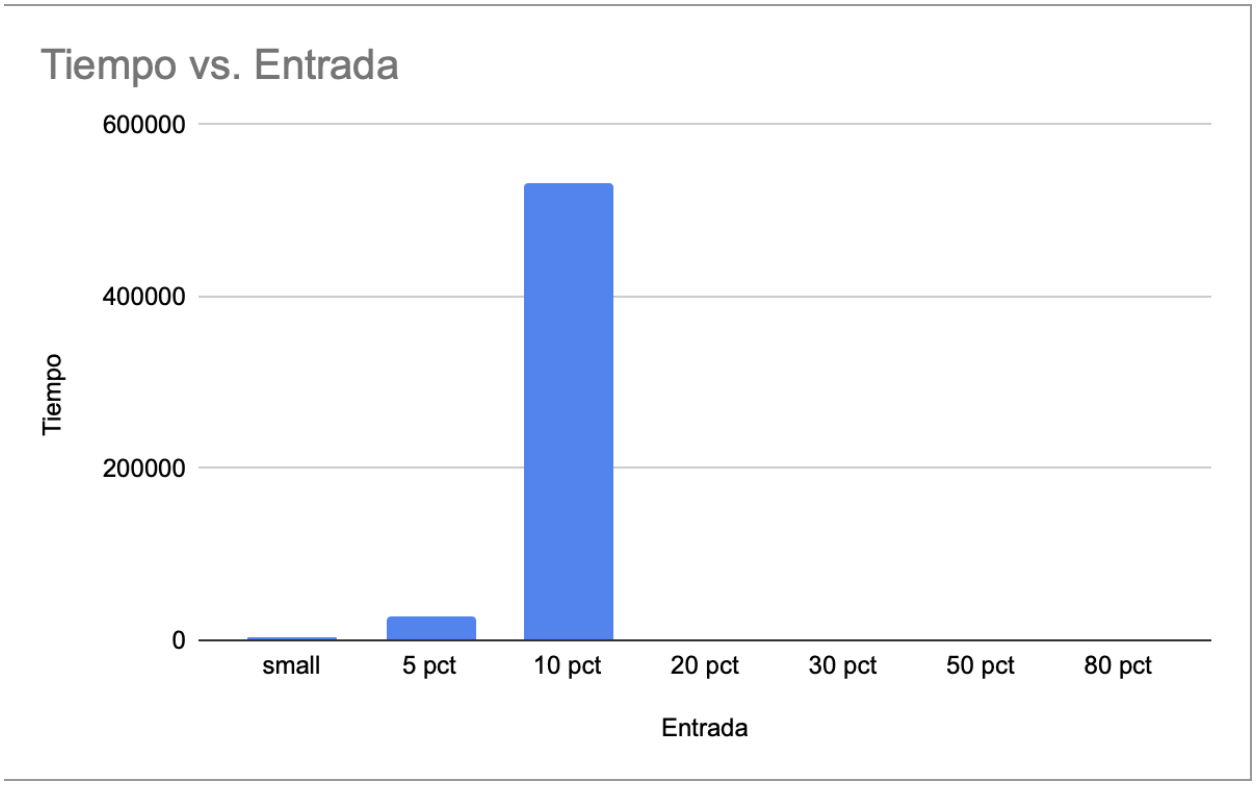
Pruebas Realizadas

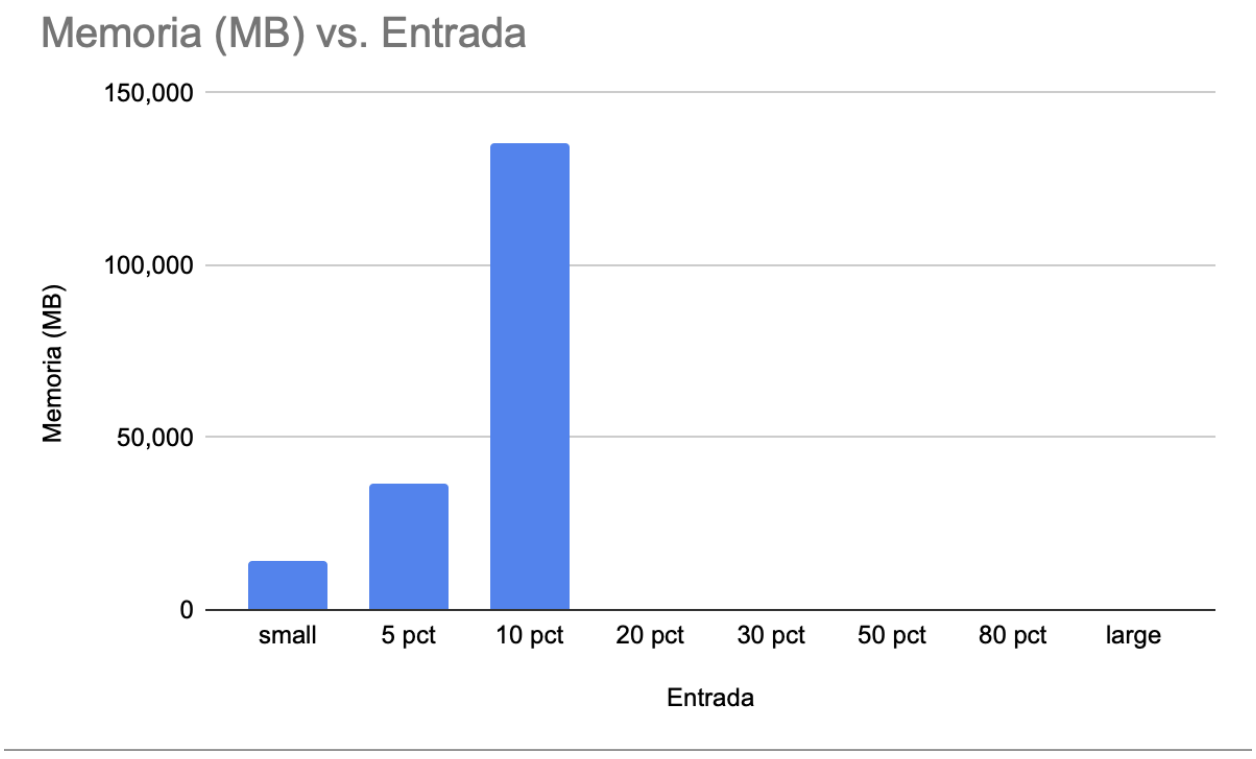
Descripción de las pruebas de tiempos de ejecución y memoria utilizada. Incluir descripción del procedimiento, las condiciones, las herramientas y recursos utilizados (librerías, computadores donde se ejecutan las pruebas, entre otros).

Entrada	Tiempo (s)	Memoria
small	2766	13979
5 pct	26897	36361
10 pct	530957	135433
20 pct		
30 pct		
50 pct		
80 pct		
large		

Graficas

Las gráficas con la representación de las pruebas realizadas.


Tiempo vs. Entrada
600000
400000
200000
0
Tiempo
small
5 pct
10 pct
20 pct
30 pct
50 pct
80 pct
Entrada


Memoria (MB) vs. Entrada
150,000
100,000
50,000
0
Memoria (MB)
small
5 pct
10 pct
20 pct
30 pct
50 pct
80 pct
large
Entrada

Análisis

La implementación correspondiente al requerimiento 4, en su caso más desfavorable, ostenta una complejidad algorítmica de O(N^3), asumiendo que los parámetros suministrados por el usuario abarquen la totalidad de los datos. La representación gráfica del rendimiento en términos de tiempo y memoria solo se tomaron hasta el 10 pct, debido a la demora en el tiempo de respuesta, al escalar la cantidad de datos de entrada, ha evidenciado una tendencia de crecimiento exponencial en ambas métricas. Esta constatación permite corroborar que la complejidad proyectada del código se refleja fielmente en la dinámica de respuesta del requerimiento frente al incremento sucesivo de la carga de datos procesados.

Requerimiento 5

Descripción

```python
def req_5(dataStructs, minDepth, minNst):
    """
    Función que soluciona el requerimiento 5
    """
    # TODO: Realizar el requerimiento 5
    events = lt.newList()
    filtered = lt.newList()
    depthTree = dataStructs["seismicEventsByDepth"]
    depthNodes = om.values(depthTree, minDepth, om.maxKey(depthTree))
    dateMap = om.newMap()
    metaData = {"totalDifferentDates": 0, "totalEventsBetweenDates": 0}
    for depthNode in lt.iterator(depthNodes):
        nstTree = depthNode["seismicEventsByNst"]
        nstLists = om.values(nstTree, minNst, om.maxKey(nstTree))
        for nstList in lt.iterator(nstLists):
            for seismicEvent in lt.iterator(nstList):
                addEventToSimpleMap(dateMap, seismicEvent, "time")
                lt.addLast(events, seismicEvent)
    for dateKey in lt.iterator(mp.keySet(dateMap)):
        dateList = me.getValue(mp.get(dateMap, dateKey))
        metaData["totalEventsBetweenDates"] += lt.size(dateList)
        datum = {"time": dateKey, "events": lt.size(dateList), "details": dateList}
        lt.addLast(filtered, datum)
    metaData["totalDifferentDates"] = mp.size(dateMap)
    sortData(filtered, compareByNumber)
    metaData["map"] = lt.size(events) > 0
    metaData["path"] = "req_5"
    req_8(events, "req_5")
    return getNData(filtered, 20), metaData
```

`req_5` es una función que filtra eventos sísmicos basándose en una profundidad mínima y un número mínimo de estaciones reportantes. Agrupa los eventos por fecha y compila metadatos sobre el total de eventos y la diversidad de fechas. Tras ordenarlos y procesarlos

adicionalmente con `req_8`, devuelve una selección de los 20 eventos principales y metadatos que indican la posibilidad de mapeo y la ruta de proceso.

Entrada	
Salida	
Implementado	Sí, el requerimiento fue implementado, desarrollado y completado por Pablo Andrés Sebastián Parra Céspedes, código 202310768, del grupo No.1 de la sección No.3 del curso de Estructuras de Datos y Algoritmos.

Análisis de complejidad

Pasos	**Complejidad**
def req_5(dataStructs, minDepth, minNst):	O(1)
events = lt.newList()	O(1)
filtered = lt.newList()	O(1)
depthTree = dataStructs["seismicEventsByDepth"]	O(1)
depthNodes = om.values(depthTree, minDepth, om.maxKey(depthTree))	O(Log (N+M))(M<=N)
dateMap = om.newMap()	O(1)
metaData = {"totalDifferentDates": 0, "totalEventsBetweenDates": 0}	O(1)
for depthNode in lt.iterator(depthNodes):	O(M)
nstTree = depthNode["seismicEventsByNst"]	O(1)
nstLists = om.values(nstTree, minNst, om.maxKey(nstTree))	O(Log(M+L) (L<=M)
for nstList in lt.iterator(nstLists):	O(L)
for seismicEvent in lt.iterator(nstList):	O(K) (K<=L)
addEventToSimpleMap(dateMap, seismicEvent, "time")	O(1)
lt.addLast(events, seismicEvent)	O(1)
for dateKey in	O(1)

lt.iterator(mp.keySet(dateMap)):	
dateList = me.getValue(mp.get(dateMap, dateKey))	O(K)
metaData["totalEventsBetweenDates"] += lt.size(dateList)	O(1)
datum = {"time": dateKey, "events": lt.size(dateList), "details": dateList}	O(1)
lt.addLast(filtered, datum)	O(1)
metaData["totalDifferentDates"] = mp.size(dateMap)	O(1)
sortData(filtered, compareByNumber)	O(K^(3/2)
metaData["map"] = lt.size(events) > 0	O(1)
metaData["path"] = "req_5"	O(1)
req_8(events, "req_5")	O(K)
return getNData(filtered, 20), metaData	O(1)
Total	O(M*L*K)

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)	Memoria
small	961	9020
5 pct	21625	44316
10 pct		
20 pct		
30 pct		
50 pct		
80 pct		
large		

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

Descripción

```python
def req_6(dataStructs, year, lat, lon, r, N):
    """
    Función que soluciona el requerimiento 6
    """
    # TODO: Realizar el requerimiento 6
    filtered = lt.newList()
    metaData = {"highestEvent": lt.newList(), "maxNPossibleEvents": N*2, "totalEvents": 0}
    eventsInRange = lt.newList("ARRAY_LIST")
    center = (float(lat), float(lon))
    highestSigEvent = {"sig": 0}
    yearMap = dataStructs["seismicEventsByYear"]
    entry = mp.get(yearMap, year)
    if entry:
        seismicEvents = me.getValue(entry)["seismicEvents"]
        for seismicEvent in lt.iterator(seismicEvents):
            distance = haversine(center, (float(seismicEvent["lat"]), float(seismicEvent["long"])))
            if distance <= float(r):
                lt.addLast(eventsInRange, seismicEvent)
                if float(seismicEvent["sig"]) > float(highestSigEvent["sig"]):
                    highestSigEvent = seismicEvent
    sortData(eventsInRange, compareByDateAscending)
    dateMap = mp.newMap()
    for seismicEvent in lt.iterator(getPreAndPosN(eventsInRange, getIndex(eventsInRange, highestSigEvent), N)):
        addEventToSimpleMap(dateMap,seismicEvent, "time")
    for dateKey in lt.iterator(mp.keySet(dateMap)):
        dateList = me.getValue(mp.get(dateMap, dateKey))
        metaData["totalEvents"] += lt.size(dateList)
        datum = {"time": dateKey, "events": lt.size(dateList), "details": dateList}
        lt.addLast(filtered, datum)
    lt.addLast(metaData["highestEvent"], highestSigEvent)
    metaData["differentDates"] = mp.size(dateMap)
    metaData["eventsInRange"] = lt.size(eventsInRange)
    metaData["map"] = lt.size(eventsInRange) > 0
    metaData["path"] = "req_6"
    req_8(eventsInRange, "req_6", r, (lat, lon))
    return sortData(filtered, compareByDateAscending), metaData
```

La función `req_6` busca y clasifica eventos sísmicos basándose en la proximidad a un punto dado y su significancia dentro de un año específico. Encuentra el evento más significativo dentro de un radio definido, organiza los eventos por fecha y selecciona un subconjunto alrededor del evento más significativo. Compila metadatos sobre los eventos y fechas, y determina la viabilidad de mapeo. Finalmente, procesa los datos con `req_8` y devuelve los eventos ordenados junto con un resumen de metadatos que incluyen el evento más significativo y el conteo total de eventos.

Entrada	

Salida	
Implementado	Sí, el requerimiento fue implementado, desarrollado y completado por el grupo No.1 de la sección No.3 del curso de Estructuras de Datos y Algoritmos.

Análisis de complejidad

Pasos	**Complejidad**
def req_6(dataStructs, year, lat, lon, r, N):	O(1)
filtered = lt.newList()	O(1)
metaData = {"highestEvent": lt.newList(), "maxNPossibleEvents": N*2, "totalEvents": 0}	O(1)
eventsInRange = lt.newList("ARRAY_LIST")	O(1)
center = (float(lat), float(lon))	O(1)
highestSigEvent = {"sig": 0}	O(1)
yearMap = dataStructs["seismicEventsByYear"]	O(1)
entry = mp.get(yearMap, year)	O(1)
if entry:	O(1)
seismicEvents = me.getValue(entry)["seismicEvents"]	O(1)
for seismicEvent in lt.iterator(seismicEvents):	O(N)
distance = haversine(center, (float(seismicEvent["lat"]), float(seismicEvent["long"])))	O(1)
if distance <= float(r):	O(1)
lt.addLast(eventsInRange, seismicEvent)	O(1)
if float(seismicEvent["sig"]) > float(highestSigEvent["sig"]):	O(1)
highestSigEvent = seismicEvent	O(1)
sortData(eventsInRange,	O(K^(3/2))

compareByDateAscending)	
dateMap = mp.newMap()	O(1)
for seismicEvent in lt.iterator(getPreAndPosN(eventsInRange, getIndex(eventsInRange, highestSigEvent), N)):	O(N)
addEventToSimpleMap(dateMap,seismicEvent, "time")	O(1)
for dateKey in lt.iterator(mp.keySet(dateMap)):	O(M)
dateList = me.getValue(mp.get(dateMap, dateKey))	O(1)
metaData["totalEvents"] += lt.size(dateList)	O(1)
datum = {"time": dateKey, "events": lt.size(dateList), "details": dateList}	O()
lt.addLast(filtered, datum)	O(1)
lt.addLast(metaData["highestEvent"], highestSigEvent)	O(1)
metaData["differentDates"] = mp.size(dateMap)	O(1)
metaData["eventsInRange"] = lt.size(eventsInRange)	O(1)
metaData["map"] = lt.size(eventsInRange) > 0	O(1)
metaData["path"] = "req_6"	O(1)
req_8(eventsInRange, "req_6", r, (lat, lon))	O(N)
return sortData(filtered, compareByDateAscending), metaData	O(P)
Total	O(N)

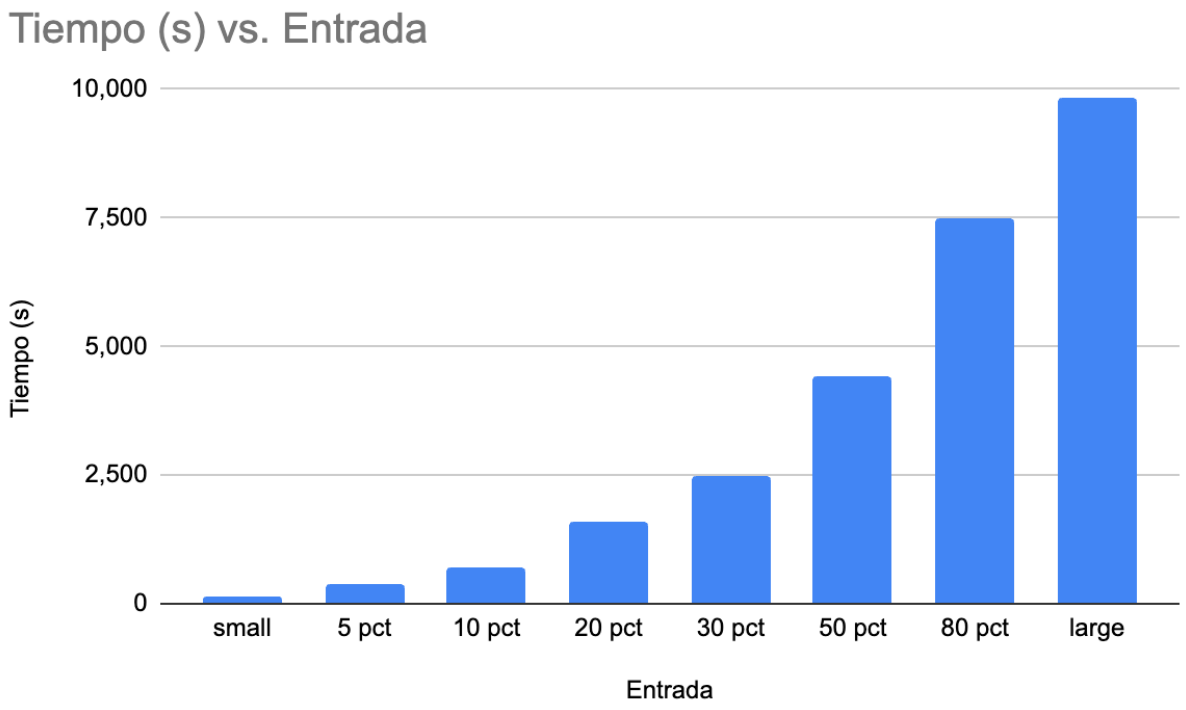
Pruebas Realizadas

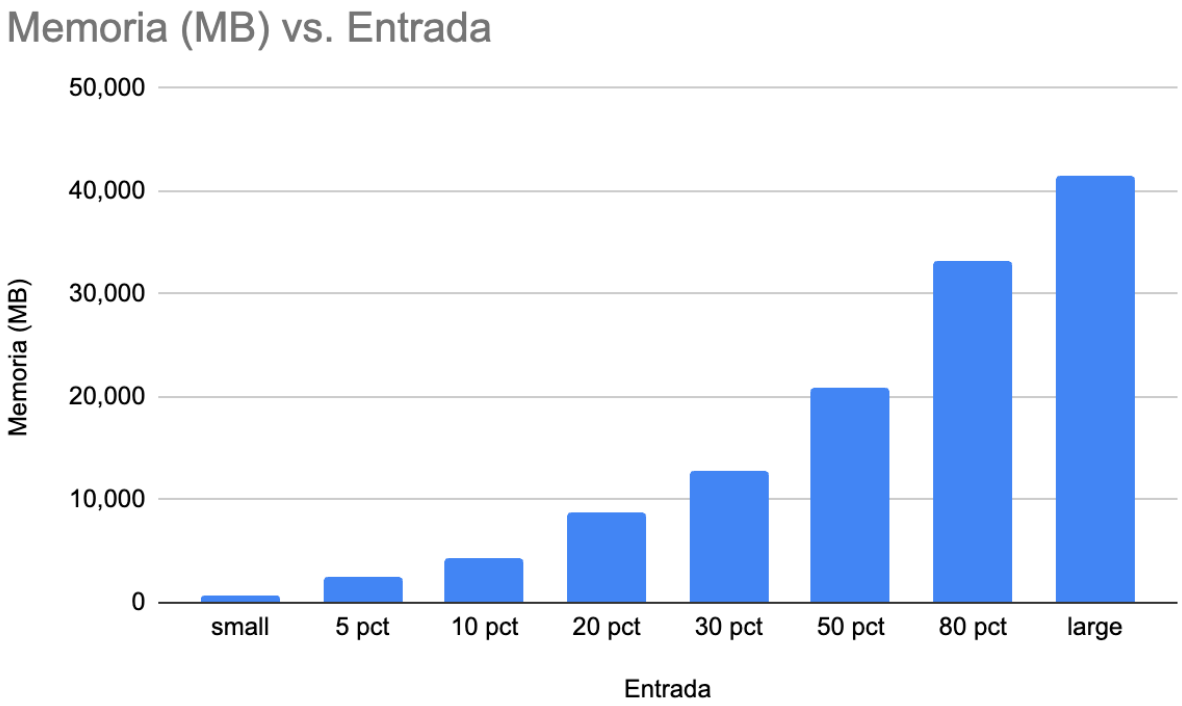
Las pruebas se realizaron utilizando un ordenador portátil con un nivel de batería del 50%, con la única aplicación en funcionamiento en ese momento siendo VSCode. Los datos se adquirieron de manera externa para evitar la necesidad de abrir otras aplicaciones durante la ejecución del programa.

Procesador	Apple M1
Memoria RAM	8 GB
Sistema operativo	macOS Ventura Versión 13.5.2

Entrada	Tiempo (s)	Memoria (MB)
small	0,122	0,630
5 pct	0,366	2,462
10 pct	0,723	4,407
20 pct	1,585	8,856
30 pct	2,474	12,849
50 pct	4,407	20,905
80 pct	7,491	33,117
large	9,855	41,558

Graficas





Análisis

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

Requerimiento 7

Descripción

```python
def req_7(dataStructs, year, place, feature, N):
    """
    Función que soluciona el requerimiento 7
    """
    # TODO: Realizar el requerimiento 7
    filtered = lt.newList()
    histogram = None
    yearMap = dataStructs["seismicEventsByYear"]
    entry = mp.get(yearMap, year)
    metaData = {}
    if entry:
        placeMap = me.getValue(entry)["seismicEventsByPlace"]
        entry = mp.get(placeMap, place)
        if entry:
            featureTree = me.getValue(entry)[feature]
            filtered = me.getValue(entry)["seismicEvents"]
            intervalList = getIntervals(featureTree, N)
            featureCounters = []
            for interval in intervalList:
                featureLists = om.values(featureTree, interval[0], interval[1])
                eventsCounter = 0
                for featureList in lt.iterator(featureLists):
                    eventsCounter += lt.size(featureList)
                featureCounters.append(eventsCounter)
            histogram = {"x": [str(interval) for interval in intervalList],
                         "y": featureCounters,
                         "title": f'Histogram of {feature} in {place} in {year}',
                         "tableTitle": f'Event details in {place} in {year}',
                         "yLabel": 'Numero de eventos',
                         "xLabel": feature}
    metaData["map"] = lt.size(filtered) > 0
    metaData["path"] = "req_7"
    req_8(filtered, "req_7")
    return sortData(filtered, compareByDateAscending), histogram, metaData
```

`req_7` genera un histograma de una característica específica de eventos sísmicos seleccionados por año y lugar, y organiza los eventos en intervalos para el análisis. Crea y cuenta eventos en cada intervalo, proporcionando una visualización estadística y verifica la posibilidad de mapear los datos. Además, procesa los eventos con `req_8` y devuelve eventos ordenados y metadatos que incluyen la viabilidad de visualización en un mapa.

Entrada	
Salida	

Implementado	Sí, el requerimiento fue implementado, desarrollado y completado por el grupo No.1 de la sección No.3 del curso de Estructuras de Datos y Algoritmos.

Análisis de complejidad

Pasos	**Complejidad**
def req_7(dataStructs, year, place, feature, N):	O(1)
filtered = lt.newList()	O(1)
histogram = None	O(1)
yearMap = dataStructs["seismicEventsByYear"]	O(1)
entry = mp.get(yearMap, year)	O(1)
metaData = {}	O(1)
if entry:	O(1)
placeMap = me.getValue(entry)["seismicEventsByPlace"]	O(1)
entry = mp.get(placeMap, place)	O(1)
if entry:	O(1)
featureTree = me.getValue(entry)[feature]	O(1)
filtered = me.getValue(entry)["seismicEvents"]	O()
intervalList = getIntervals(featureTree, N)	O()
featureCounters = []	O(1)
for interval in intervalList:	O()
featureLists = om.values(featureTree, interval[0], interval[1])	O()
eventsCounter = 0	O(1)
for featureList in lt.iterator(featureLists):	O(1)
eventsCounter += lt.size(featureList)	O(1)

featureCounters.append(eventsCounter)	O(1)
histogram = {"x": [str(interval) for interval in intervalList], "y": featureCounters, "title": f'Histogram of {feature} in {place} in {year}', "tableTitle": f'Event details in {place} in {year}', "yLabel": 'Numero de eventos', "xLabel": feature}	O(1)
metaData["map"] = lt.size(filtered) > 0	O(1)
metaData["path"] = "req_7"	O(1)
req_8(filtered, "req_7")	O(N)
return sortData(filtered, compareByDateAscending), histogram, metaData	O(1)
Total	O(N)

Pruebas Realizadas

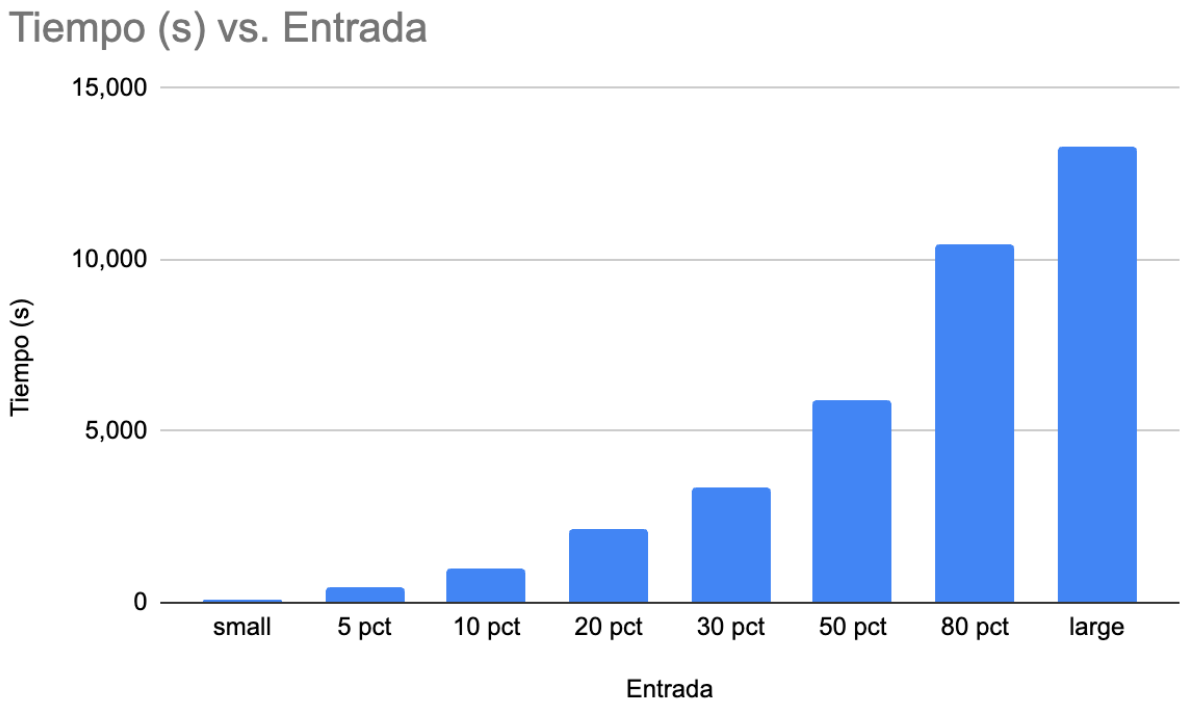
Las pruebas se realizaron utilizando un ordenador portátil con un nivel de batería del 50%, con la única aplicación en funcionamiento en ese momento siendo VSCode. Los datos se adquirieron de manera externa para evitar la necesidad de abrir otras aplicaciones durante la ejecución del programa.

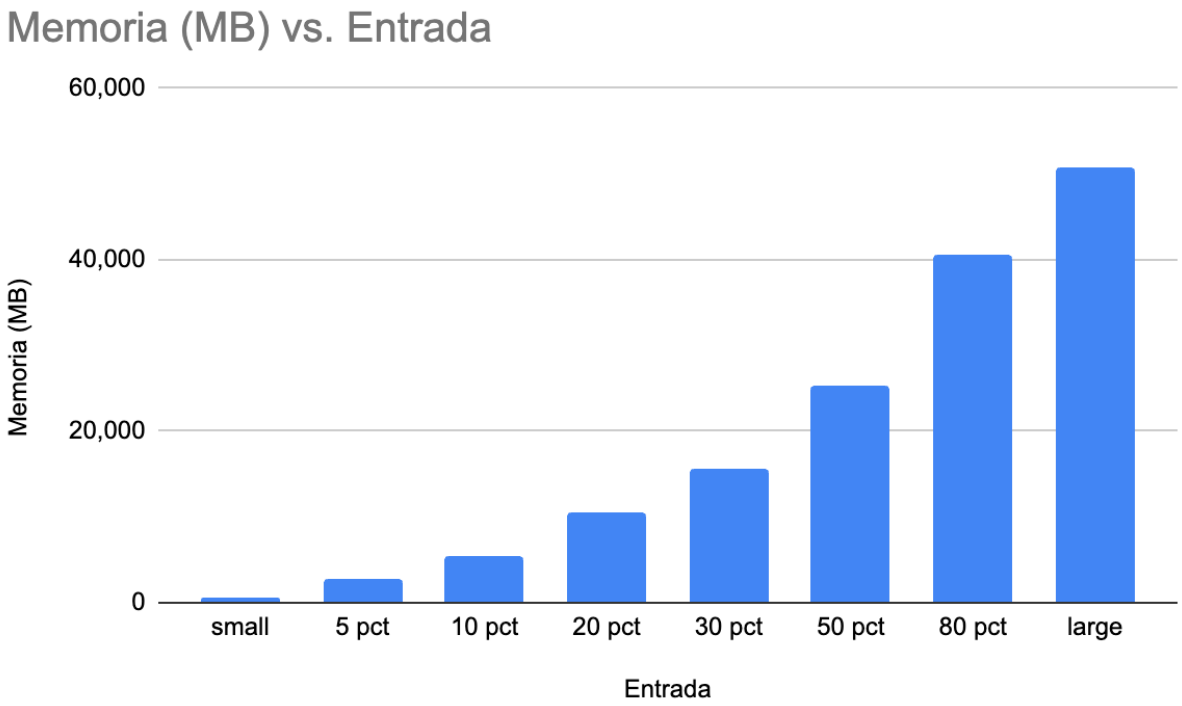
Procesador	Apple M1
Memoria RAM	8 GB
Sistema operativo	macOS Ventura Versión 13.5.2

Entrada	**Tiempo (s)**	**Memoria (MB)**
small	0,103	0,606
5 pct	0,457	2,844
10 pct	0,972	5,339
20 pct	2,172	10,637
30 pct	3,390	15,576
50 pct	5,920	25,433

80 pct	10,431	40,663
large	13,320	50,864

Graficas





Análisis

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

Requerimiento 8

Descripción

```python
def req_8(events, path, radius=False, center=None):
    """
    Función que soluciona el requerimiento 8
    """
    # TODO: Realizar el requerimiento 8
    eventsMap = folium.Map((0, 0), zoom_start= 2)
    mark = MarkerCluster()
    for event in lt.iterator(events):
        mark.add_child(folium.Marker((event["lat"], event["long"]), popup= createPopUp(event)))
    if radius and center:
        folium.Circle(center, radius*1000, color= "red").add_to(eventsMap)
    eventsMap.add_child(mark)
    eventsMap.save(f'{path}.html')
```

La función `req_8` utiliza Folium para mapear eventos sísmicos, marcando sus ubicaciones en un mapa interactivo y agrupándolos para mejorar la visualización. Si se especifican un radio y un centro, añade un círculo para destacar una región específica. El mapa completo se guarda como un archivo HTML, nombrado según el parámetro `path` proporcionado.

Entrada	
Salida	
Implementado	Sí, el requerimiento fue implementado, desarrollado y completado por el grupo No.1 de la sección No.3 del curso de Estructuras de Datos y Algoritmos.

Análisis de complejidad

Pasos	**Complejidad**
def req_8(events, path, radius=False, center=None):	O(1)
eventsMap = folium.Map((0, 0), zoom_start= 2)	O(1)
mark = MarkerCluster()	O(1)
for event in lt.iterator(events):	O(N)
mark.add_child(folium.Marker((event["lat"], event["long"]), popup= createPopUp(event)))	O(1)
if radius and center:	O(1)
folium.Circle(center, radius*1000, color= "red").add_to(eventsMap)	O(1)
eventsMap.add_child(mark)	O(1)
eventsMap.save(f'{path}.html')	O(1)
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)**

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.