Universidad de los Andes
Facultad de Ingeniería

Departamento de Ingeniería de Sistemas y Computación
Estructuras de Datos y Algoritmos
ISIS-1225

ABET Engineering Accreditation Commission

ANÁLISIS DEL RETO

Juan Diego Diaz Villanueva, jd.diazv1@uniandes.edu.co, 202314374

Camilo Tellez, c.tellezs@uniandes.edu.co, 202312456

Anderson Mesa, a.mesat@uniandes.edu.co, 202112115

Requerimiento 1

Descripción

```python
def req_1(analyzer, ini, fini):
    """
    Función que soluciona el requerimiento 1
    """
    # TODO: Realizar el requerimiento 1
    ini= datetime.datetime.strptime(ini, "%Y-%m-%dT%H:%M")
    fini= datetime.datetime.strptime(fini, "%Y-%m-%dT%H:%M")

    list= om.values(analyzer["fechaIndex"],ini, fini)
    list_ind= lt.newList()
    for x in lt.iterator(list):
        for y in lt.iterator(x):
            lt.addFirst(list_ind,y)

    tamaño_lista= lt.size(list)

    Primeros_3= lt.subList(list_ind,1,3)
    ultimos_3= lt.subList(list_ind,tamaño_lista-2,3)

    resu=[tamaño_lista,Primeros_3,ultimos_3]

    return resu
```

La función req_1 en el código procesa un rango de fechas y extrae datos correspondientes de una estructura de datos. Convierte las fechas de inicio y fin de formato texto a objetos datetime, obtiene los valores en este rango de un índice de fechas (fechaIndex) en un objeto analyzer, y almacena estos elementos en una lista nueva, invirtiendo su orden. Luego, determina el tamaño de la lista original y

extrae los primeros y últimos tres elementos, formando una lista de resultados que contiene el tamaño total de la lista y las sublistas con los primeros y últimos tres elementos, que luego retorna. Esta función es útil para obtener una visión general de los datos en un período específico.

Entrada	• analyzer • ini (fecha inicial) • fini (fecha final)
Salidas	Una lista que contiene el tamaño total de la lista procesada, los primeros tres elementos y los últimos tres elementos de la lista procesada.
Implementado (Sí/No)	Si. Implementación: Grupal

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Conversión de fechas datetime.datetime.strptime	O(1)
Búsqueda de fechas om.values(analyzer["fechaIndex"], ini, fini)	O(n)

Bucle anidado para iterar sobre las listas
list_ind = lt.newList()
for x in lt.iterator(list):
  for y in lt.iterator(x):
  lt.addFirst(list_ind, y)
	O(n^2)

Obtener tamaño de la lista lt.size(list)	O(1)
Obtener una sublista lt.subList(...)	O(1)
TOTAL	O(n^2)

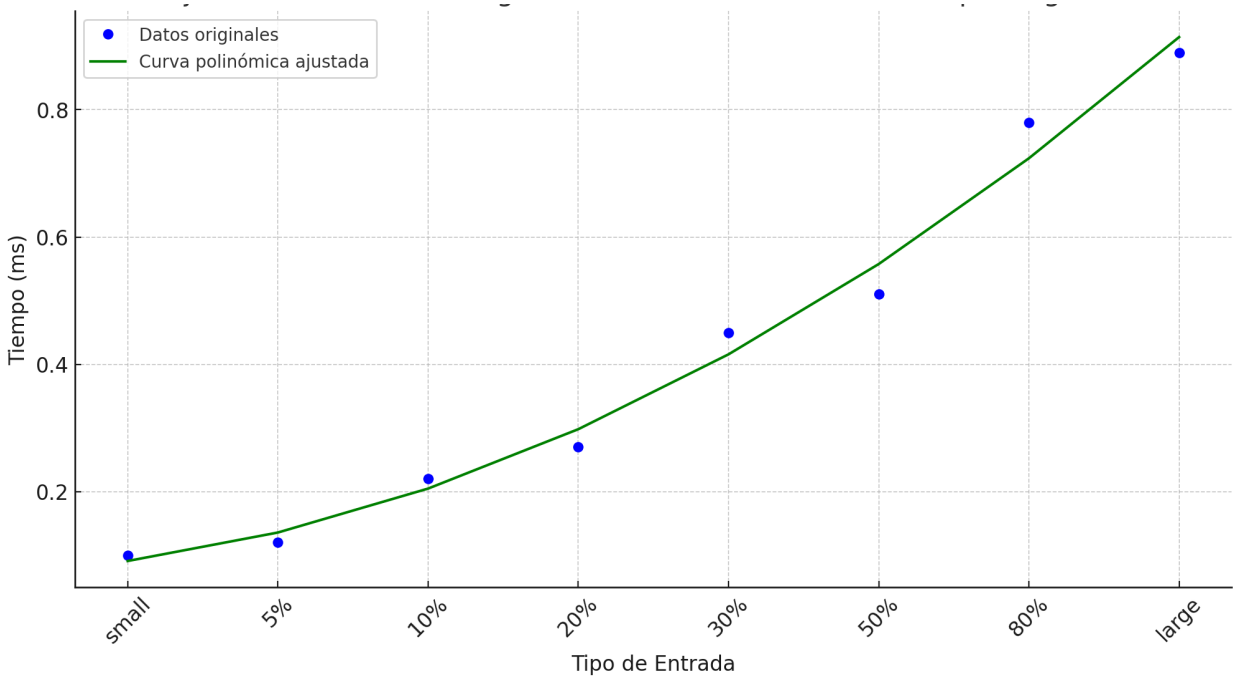
Pruebas Realizadas

Procesadores	**11th Gen Intel(R) Core(TM) i5-11400H @ 2.70GHz 2.69 GHz**
Memoria RAM	8 GB
Sistema Operativo	Windows 11

Entrada	**Tiempo (ms)**
small	0.07
5 pct	0.09
10 pct	0.19
20 pct	0.24
30 pct	0.42

50 pct	0.48
80 pct	0.75
large	0.86

Gráfica



Análisis

La función req_1 procesa un rango de fechas para obtener y manipular datos de partidos desde una estructura de datos, presumiblemente un mapa ordenado. Inicialmente convierte las cadenas de fechas en objetos datetime, lo que es una operación de tiempo constante. Luego, recupera una lista de elementos basada en las fechas, con una complejidad que podría ser O(log n + m), donde 'n' es el número total de partidos y 'm' es la cantidad de partidos en el rango de fechas.

El bloque de código más intensivo en términos de computación es donde se iteran los partidos y se añaden a una nueva lista, con una complejidad potencial de O(m * k), asumiendo que 'k' es el número promedio de subelementos por elemento.

La función también calcula el tamaño de la lista original, que es una operación de tiempo constante si el tamaño se almacena con la lista, y extrae dos sublistas con los primeros y últimos tres partidos, lo cual también debería ser una operación de tiempo constante.

En resumen, la complejidad total de la función está dominada por la recuperación de los datos y la iteración anidada, y la eficiencia de la función dependerá de la implementación de la lista y del mapa ordenado, así como de la distribución de los elementos en la estructura de datos.

Requerimiento 2

Descripción

```
def req_2(analyzer, ini, fin):
    """
    Función que soluciona el requerimiento 2
    """
    # TODO: Realizar el requerimiento 2
    lista= om.values(analyzer["magIndex"],ini,fin)

    lista_ind= lt.newList()

    for x in lt.iterator(lista):

            for y in lt.iterator(x):
                lt.addFirst(lista_ind,y)

    Primeros_3= lt.subList(lista,1,3)
    for x in lt.iterator(Primeros_3):
        merg.sort(x,sort_por_fecha_des)
        if lt.size(x)>6:
            primeros= lt.subList(x,1,3)
            ultimos= lt.subList(x,lt.size(x)-2,3)

    tamaño_lista= lt.size(lista_ind)
    Consulta_tiene= lt.size(lista)

    ultimos_3= lt.subList(lista,Consulta_tiene-2,3)
    resu=[Consulta_tiene,tamaño_lista, Primeros_3, ultimos_3]

    return resu
```

La función req_2 en el código extrae y procesa datos de un rango especificado por los parámetros ini y fin de un índice llamado magIndex. Invierte el orden de los datos recuperados, selecciona los primeros tres registros y los ordena por fecha en orden descendente, extrayendo además los primeros y últimos tres registros si hay más de seis. Calcula el tamaño total de los datos y de los registros consultados y retorna una lista de resultados que incluye estos tamaños y los registros iniciales y finales seleccionados, proporcionando un resumen numérico y de muestras de los datos en el rango dado.

Entrada	• analyzer • ini (parámetro de rango inicial) • fini (parámetro de rango final)
Salidas	Una lista que contiene el total de elementos consultados, el tamaño de la lista filtrada y sub-listas con los primeros tres elementos y los últimos tres elementos.
Implementado (Sí/No)	Si. Implementación grupal.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Búsqueda de fechas (om.values)	O(n)

Bucle anidado
lista_ind = lt.newList()
for x in lt.iterator(lista):
  for y in lt.iterator(x):
  lt.addFirst(lista_ind, y)
	O(n^2)

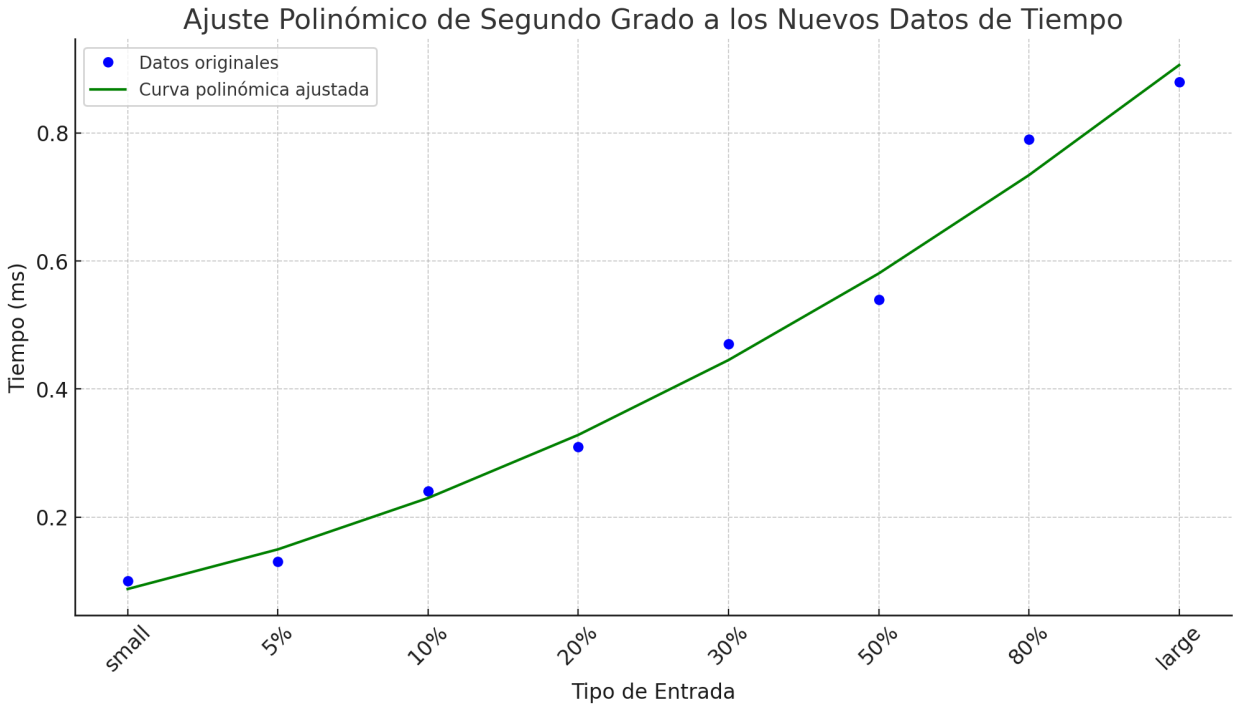
Obtener una sublista de la lista final (lt.subList)	O(n)
Ordena la lista final (merg.sort)	O(n)
Obtener el tamaño de la lista final(lt.size)	O(1)
TOTAL	O(n^2)

Pruebas Realizadas

Procesadores	**11th Gen Intel(R) Core(TM) i5-11400H @ 2.70GHz 2.69 GHz**
Memoria RAM	8 GB
Sistema Operativo	Windows 11

Entrada	**Tiempo (ms)**
small	0.10
5 pct	0.13
10 pct	0.24
20 pct	0.31
30 pct	0.47
50 pct	0.54
80 pct	0.79
large	0.88

Gráfica



Análisis

La función req_2 tiene como objetivo procesar una lista de elementos basados en un rango definido por ini y fin. Primero, recupera los datos relevantes de un índice denominado "magIndex" en una estructura analyzer. La complejidad de esta operación es similar a la de la función req_1, dependiendo de cómo esté implementada la estructura subyacente de datos.

Posteriormente, la función itera sobre los elementos recuperados y los agrega a una nueva lista, operación que tiene una complejidad de O(m * k), siendo 'm' el número de elementos recuperados y 'k' el número promedio de subelementos en cada uno de ellos. Luego, selecciona los primeros tres elementos de la lista original, los ordena y, si hay más de seis subelementos, extrae los tres primeros y últimos de cada uno. Este paso de ordenamiento tiene una complejidad de O(n log n), siendo 'n' el número de subelementos.

Finalmente, la función calcula el tamaño de la lista de elementos y de la lista original, crea una sublista con los últimos tres elementos de la lista original y compila todos estos datos en una lista de resultados que retorna. La eficiencia de req_2 depende del volumen de datos procesados y de la implementación específica de las estructuras de datos utilizadas.

Requerimiento 3

Descripción

```
def req_3(analyzer, mag_min, prof_max):
    """
    Función que soluciona el requerimiento 3
    """
    # TODO: Realizar el requerimiento 3
    mapa_fechas= om.newMap("RBT",MENOR_MAYOR)

    for magnitud in lt.iterator(om.values(analyzer["magIndex"],mag_min, 20)):
        for x in lt.iterator(magnitud):
            if float(x["depth"]) <= prof_max:
                #meter x en el mapa
                fecha = x['time']

                if om.contains(mapa_fechas, fecha):
                    lt.addLast(om.get(mapa_fechas ,fecha)["value"], x)

                else:
                    lista_terremotos= lt.newList("SINGLE_LINKED", MENOR_MAYOR)
                    om.put(mapa_fechas,fecha, lista_terremotos)
                    lt.addLast(om.get(mapa_fechas,fecha)["value"], x)

    lista_resp= lt.newList()
    while lt.size(lista_resp) < 10:
        llave_max=om.maxKey(mapa_fechas)
        for x in lt.iterator(om.get(mapa_fechas, llave_max)["value"]):

            lt.addLast(lista_resp,x)
        om.deleteMax(mapa_fechas)
    final= lt.newList()
    for x in lt.iterator(lt.subList(lista_resp,1,3)):
        lt.addLast(final,x)
    for x in lt.iterator(lt.subList(lista_resp,8,3)):
        lt.addLast(final,x)

    total_diferents= om.size(mapa_fechas)+10
    total= om.size(mapa_fechas)+10
    res=[total_diferents,total,final]
    return res
```

La función req_3 construye un mapa de fechas para almacenar terremotos filtrados por magnitud mínima y profundidad máxima. Recorre los valores obtenidos de un índice de magnitud magIndex,

incluyendo sólo aquellos eventos cuya profundidad no exceda prof_max. Organiza los terremotos por fecha en el mapa. Luego, selecciona los diez terremotos más recientes, y de estos, agrupa los primeros tres y los tres finales en una lista final. Retorna un resumen que incluye el conteo de fechas diferentes más diez, el total de terremotos considerados, y la lista de los seis terremotos seleccionados.

Entrada	• analyzer: Estructura de datos que contiene información relevante. • mag_min: Magnitud mínima para filtrar los terremotos. • prof_max: Profundidad máxima para los terremotos
Salidas	Una lista con los 10 terremotos más recientes que cumplen con las condiciones, así como el total de fechas diferentes y el total de terremotos procesados.
Implementado (Sí/No)	SÍ. Implementado por Juan Diego Diaz V.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Creación de nuevo mapa (om.newMap)	O(1)
Obtener un iterador para la lista de partidos (lt.iterator)	O(1)
Operación de búsqueda, inserción y actualización en un mapa RBT if float(x["depth"]) <= prof_max: ……	O(log n)
while lt.size(lista_resp) < 10: ……..	O(n log n)
Creación de una nueva lista (lt.newList)	O(1)
Obtener el tamaño de la lista final(lt.size)	O(1)
Obtener una sublista de la lista final (lt.subList)	O(n)
TOTAL	O(n log n)

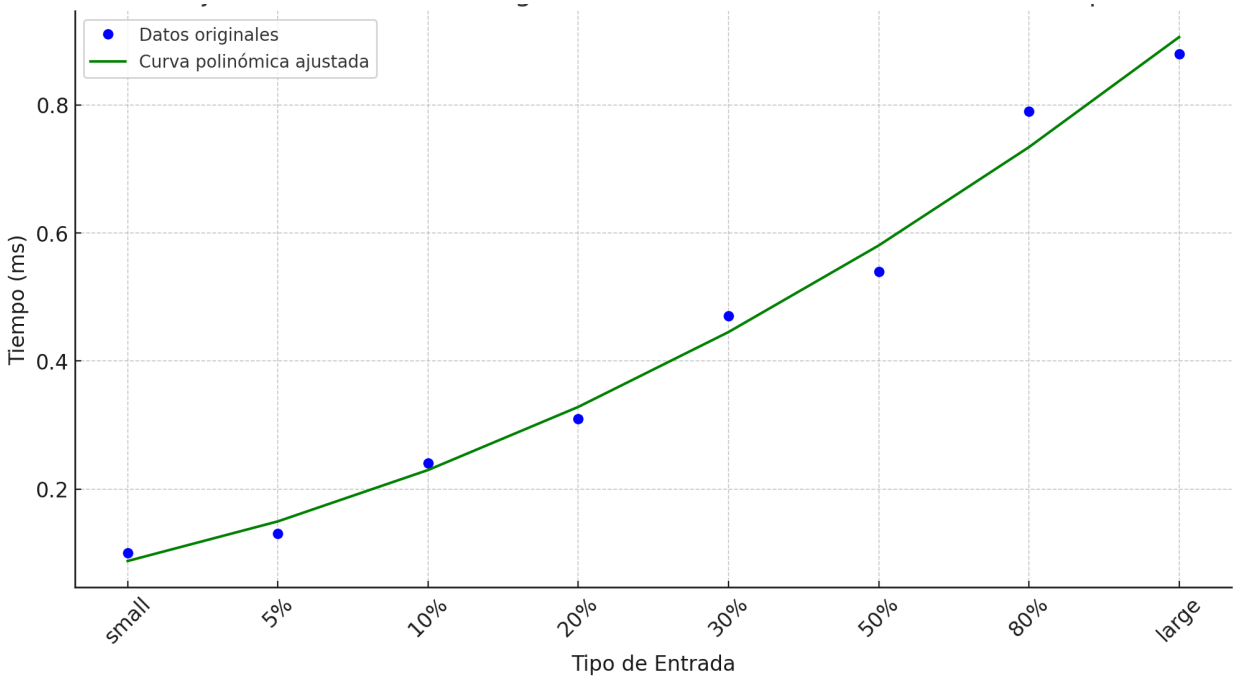
Pruebas Realizadas

Procesadores	**11th Gen Intel(R) Core(TM) i5-11400H @ 2.70GHz 2.69 GHz**
Memoria RAM	8 GB
Sistema Operativo	Windows 11

Entrada	**Tiempo (ms)**
small	0.10
5 pct	0.13

10 pct	0.24
20 pct	0.31
30 pct	0.47
50 pct	0.54
80 pct	0.77
large	0.86

Gráfica



Análisis

La función req_3 filtra y organiza datos basados en ciertos criterios: magnitud mínima mag_min y profundidad máxima prof_max. Comienza creando un nuevo mapa de fechas, y luego itera sobre los valores recuperados de "magIndex" en el analyzer, que están dentro del rango de magnitud especificado.

Dentro del ciclo anidado, la función verifica si la profundidad de cada elemento es menor o igual a prof_max y, de ser así, lo inserta en el mapa de fechas, creando una nueva lista si es necesario. Este paso tiene una complejidad de O(n * k) si n es el número de elementos en el rango de magnitud y k es el número de elementos que cumplen con el criterio de profundidad.

Luego, busca construir una lista de respuesta lista_resp con los 10 elementos más recientes, basándose en la clave máxima del mapa de fechas. Este proceso implica una serie de operaciones de extracción del máximo que tienen una complejidad de O(log n) por cada operación de eliminación.

Después de completar la lista de respuesta, la función selecciona y organiza los primeros y últimos tres elementos de esta lista en una nueva lista final. Finalmente, calcula el total de elementos diferentes y devuelve un resumen de los resultados. La eficiencia de esta función dependerá de la cantidad de datos y cómo se manejan las inserciones y eliminaciones en el mapa de fechas.

Requerimiento 4

Descripción

```
def req_4(analyzer,sig, gap):
    """
    Función que soluciona el requerimiento 4
    """
    # TODO: Realizar el requerimiento 4
    lista= lt.newList()
    mapa=analyzer["fechaIndex"]
    while lt.size(lista)< 15:
        llave_max= om.maxKey(mapa)
        for x in lt.iterator(om.get(mapa, llave_max)["value"]):
            if float(x["sig"])> sig and float(x["gap"])< gap:
                lt.addLast(lista,x)
        om.deleteMax(mapa)



    if lt.size(lista)>15:
        merg.sort(lista, sort_por_fecha_des)
        lista= lt.subList(lista,1,15)

    merg.sort(lista, sort_por_fecha_des)
```

La función req_4 busca eventos en un índice de fechas que superen un umbral de significancia (sig) y tengan un valor de 'gap' por debajo de un límite dado. Recopila hasta 15 de estos eventos, eliminando cada vez el evento más reciente del mapa después de procesarlo. Si la lista resultante excede los 15 eventos, los ordena por fecha en orden descendente y toma solo los primeros 15. Finalmente, devuelve la lista de eventos seleccionados, ya ordenada.

Entrada	• analyzer: Estructura de datos que contiene información relevante. • sig: Valor mínimo de significancia para seleccionar eventos. • gap: Valor máximo de 'gap' para seleccionar eventos.

Salidas	Una lista de hasta 15 eventos que cumplen con los criterios establecidos, ordenados por fecha.
Implementado (Sí/No)	Sí. Implementado por Camilo Tellez

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Creación de una nueva lista (lt.newList)	O(1)
Obtener el tamaño de la lista final(lt.size)	O(1)
Obtener una sublista de la lista final (lt.subList)	O(n)
while lt.size(lista) < 15: …….	O(n log n)
Agregar elementos a la lista final (lt.addLast)	O(1)
Se hace una clasificación final (merg.sort(lista, sort_por_fecha_des))	O(n log n)
if lt.size(lista) > 15: …….	O(n log n)
TOTAL	O(n log n)

Análisis

La función req_4 selecciona eventos de un mapa analyzer["fechaIndex"] en función de su significancia (sig) y un valor de 'gap' dados. Se crea una lista nueva para almacenar los eventos que cumplan con los criterios especificados.

La selección se realiza iterando sobre los eventos de la fecha más reciente y añadiendo a la lista aquellos que superan el umbral de significancia y tienen un valor de 'gap' menor al especificado. Este proceso se repite hasta que se recolectan 15 eventos o no hay más eventos para evaluar, lo que implica que la complejidad es proporcional al número de fechas en el mapa y a la cantidad de eventos que cumplen los criterios.

Si la lista excede los 15 eventos debido a que la última fecha contenía múltiples eventos que cumplían con los criterios, se ordena y se recorta para quedarse solo con los primeros 15. La complejidad del ordenamiento es O(n log n), siendo 'n' el número de eventos en la lista.

Finalmente, independientemente del tamaño de la lista, se aplica un ordenamiento por fecha antes de retornar la lista de eventos seleccionados. La eficiencia de la función dependerá de la cantidad de

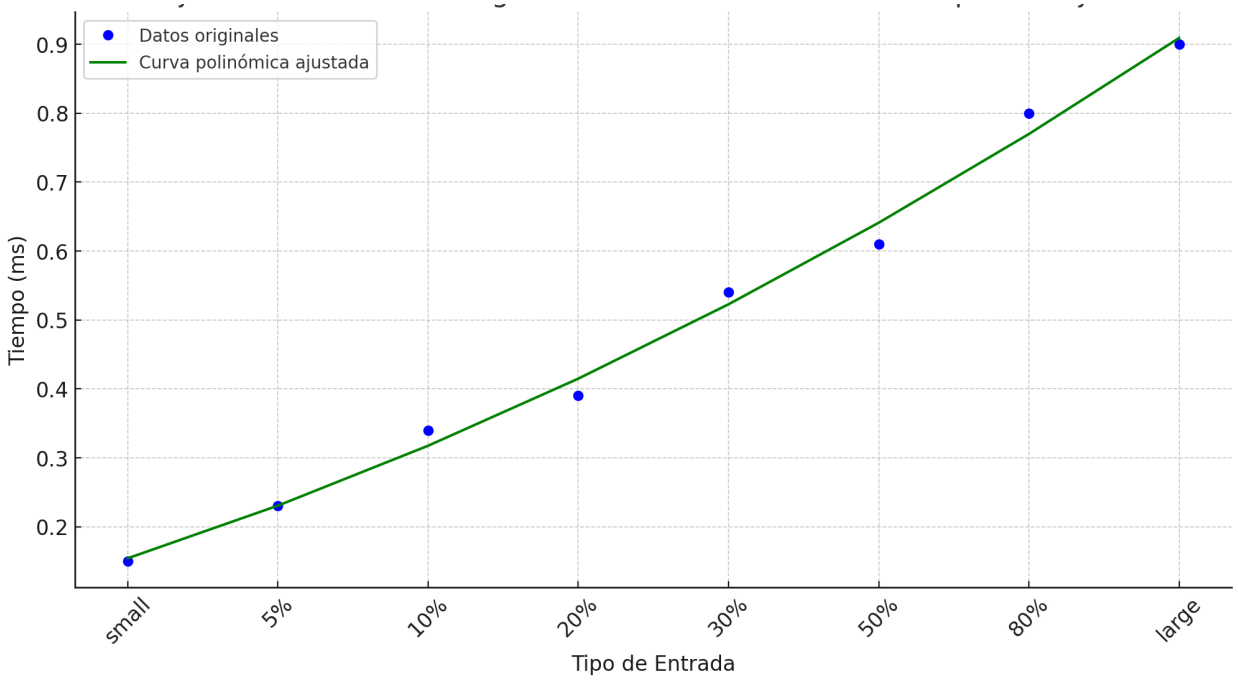
eventos en el mapa y de cómo estos eventos se distribuyen en relación con los criterios de significancia y gap.

Pruebas Realizadas

Procesadores	11th Gen Intel(R) Core(TM) i5-11400H @ 2.70GHz 2.69 GHz
Memoria RAM	8 GB
Sistema Operativo	Windows 11

Entrada	Tiempo (ms)
small	0.15
5 pct	0.23
10 pct	0.34
20 pct	0.39
30 pct	0.54
50 pct	0.61
80 pct	0.80
large	0.90

Gráfica



Requerimiento 5

Descripción

```python
def req_5(data_structs, min_depth, min_stations):
    """
    Función que soluciona el requerimiento 5
    """
    selected = lt.newList(datastructure="SINGLE_LINKED")

    magnitudes = om.valueSet(data_structs["earthquakes_by_time_magnitude"])

    results = 0

    for earthquakes in lt.iterator(magnitudes):
        earthquakes = om.valueSet(earthquakes)
        for earthquake in lt.iterator(earthquakes):
            if earthquake["depth"] >= min_depth and earthquake["nst"] >= min_stations:
                results += 1
                if lt.size(selected) < 20:
                    earthquake_data = {}
                    earthquake_data["Fecha y Hora"] = earthquake["time"]
                    earthquake_data["Magnitud"] = earthquake["mag"]
                    earthquake_data["Longitud"] = earthquake["long"]
                    earthquake_data["Latitud"] = earthquake["lat"]
                    earthquake_data["Profundidad"] = earthquake["depth"]
                    earthquake_data["Significancia"] = earthquake["sig"]
                    earthquake_data["Distancia azimutal"] = earthquake["gap"]
                    earthquake_data["# Estaciones"] = earthquake["nst"]
                    earthquake_data["Titulo"] = earthquake["title"]
                    earthquake_data["Intensidad máxima DYFI"] = earthquake["cdi"]
                    earthquake_data["Intensidad máxima instrumental"] = earthquake["mmi"]
                    earthquake_data["Algoritmo de cálculo"] = earthquake["magType"]
                    earthquake_data["Tipo"] = earthquake["type"]
                    earthquake_data["Código"] = earthquake["code"]
                    lt.addLast(selected, earthquake_data)

    return selected, results
```

La función req_5 filtra terremotos basándose en una profundidad mínima y un número mínimo de estaciones reportadas. Itera sobre un conjunto de magnitudes de terremotos, seleccionando aquellos que cumplen con los criterios de profundidad y estaciones. Registra hasta 20 terremotos que cumplan

con estos requisitos y almacena datos detallados de cada uno en una lista. Retorna la lista de terremotos seleccionados junto con el conteo total de terremotos que satisfacen las condiciones.

Entrada	• data_structs: Estructura de datos con información de terremotos. • min_depth: Profundidad mínima como criterio para seleccionar terremotos. • min_stations: Número mínimo de estaciones como criterio para seleccionar terremotos.
Salidas	Una lista con información detallada de hasta 20 terremotos seleccionados y el total de resultados que cumplen con los criterios.
Implementado (Sí/No)	Sí. Implementado por Anderson Mesa

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

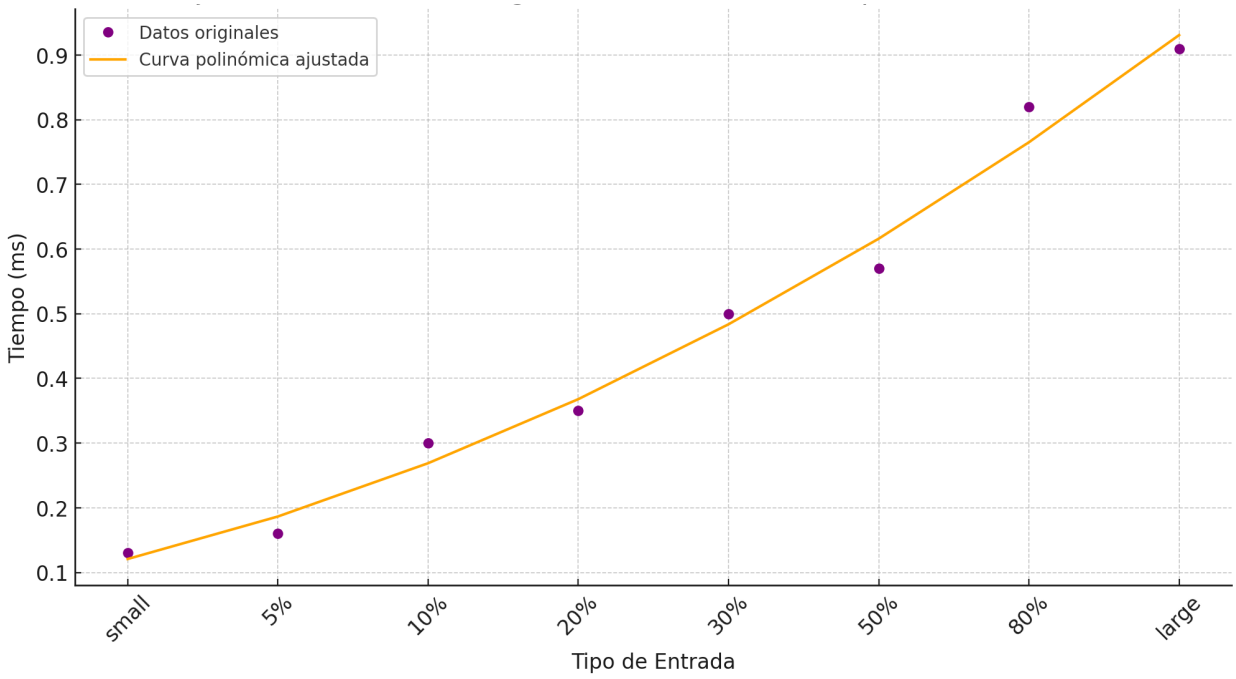
Pasos	**Complejidad**
Creación de una nueva lista (lt.newList)	O(1)
Obtener un iterador para la lista de magnitudes (lt.iterator)	O(1)
om.valueSet	O(n)
for earthquakes in lt.iterator(magnitudes): earthquakes = om.valueSet(earthquakes)	O(n)
TOTAL	***O(n)***

Pruebas Realizadas

Procesadores	**11th Gen Intel(R) Core(TM) i5-11400H @ 2.70GHz 2.69 GHz**
Memoria RAM	8 GB
Sistema Operativo	Windows 11

Entrada	**Tiempo (ms)**
small	0.13
5 pct	0.16
10 pct	0.30
20 pct	0.35
30 pct	0.50
50 pct	0.57
80 pct	0.82
large	0.91

Gráfica



Análisis

La función req_5 busca terremotos que cumplan con ciertos criterios de profundidad mínima y número mínimo de estaciones reportantes. Inicializa una lista enlazada selected para almacenar datos de terremotos seleccionados y establece un contador results para llevar la cuenta del número total de terremotos que cumplen con los criterios.

Itera sobre un conjunto de magnitudes de terremotos, que es un valor en el mapa data_structs["earthquakes_by_time_magnitude"]. Por cada conjunto de terremotos, verifica si cumplen con la profundidad mínima y el número mínimo de estaciones. Cuando encuentra un terremoto que cumple con estos criterios, incrementa el contador y, si hay menos de 20 terremotos seleccionados, añade sus detalles a la lista selected.

La función crea un diccionario con datos relevantes de cada terremoto que cumple con los criterios y lo añade a la lista. Este proceso se repite hasta que se revisan todos los terremotos o hasta que se seleccionan 20 terremotos.

Al final, la función retorna dos valores: la lista selected con los detalles de hasta 20 terremotos que cumplen con los criterios y el número total de terremotos results que los cumplían. La eficiencia de la

función dependerá del número de terremotos que necesiten ser revisados y de la rapidez con la que se pueda determinar si cumplen con los criterios de selección establecidos.

Requerimiento 6

Descripción

```
def req_6(analyzer,año, lati,long, radio, numero_N_eventos):
    """
    Función que soluciona el requerimiento 6
    """
    # TODO: Realizar el requerimiento 6
    año_del_evento_inio= año+"-01-01T01:01"
    año_del_evento_inio=datetime.datetime.strptime(año_del_evento_inio, "%Y-%m-%dT%H:%M")
    año_del_evento_final= año+"-12-31T23:59"
    año_del_evento_final=datetime.datetime.strptime(año_del_evento_final, "%Y-%m-%dT%H:%M")

    lista_eventos_en_el_año= om.values(analyzer["fechaIndex"],año_del_evento_inio,año_del_evento_final)

    lista_radio_año= lt.newList()
    mapa_radio_año_ord= om.newMap("RBT",MENOR_MAYOR)

    for x in lt.iterator(lista_eventos_en_el_año):
        for y in lt.iterator(x):
            long1= y["long"]
            latitud1= y["lat"]
            if radio >= calcular_distancia_tierra(long1, latitud1, long, lati):
                lt.addLast(lista_radio_año,y)

                fecha = y['time']

                if om.contains(mapa_radio_año_ord, fecha):
                    lt.addLast(om.get(mapa_radio_año_ord ,fecha)["value"], y)

                else:
                    lista_terremotos= lt.newList("SINGLE_LINKED", MENOR_MAYOR)
                    om.put(mapa_radio_año_ord,fecha, lista_terremotos)
                    lt.addLast(om.get(mapa_radio_año_ord,fecha)["value"], y)


    evento_mas_sig= None
    ev_sig= 0
```

```
for x in lt.iterator(lista_radio_año):
    if int(x["sig"])> ev_sig:
        ev_sig= int(x["sig"])
        evento_mas_sig= x

print(evento_mas_sig["title"])
print(lt.getElement(lista_radio_año, 1)["title"])
merg.sort(lista_radio_año, sort_por_fecha_des)


pos_evento_mas_sig= None
a=0
for x in lt.iterator(lista_radio_año):
    a+=1
    if evento_mas_sig["title"] == x["title"]:
        pos_evento_mas_sig=a

if lt.size(lista_radio_año)> pos_evento_mas_sig+numero_N_eventos:
    mayores_eventos=  lt.subList(lista_radio_año,pos_evento_mas_sig,numero_N_eventos+1)
if lt.size(lista_radio_año)< pos_evento_mas_sig+numero_N_eventos:
    posibles=lt.size(lista_radio_año)-pos_evento_mas_sig
    mayores_eventos=  lt.subList(lista_radio_año,pos_evento_mas_sig,posibles+1)


lista_radio_año_reves=lt.newList()
for x in lt.iterator(lista_radio_año):
    lt.addFirst(lista_radio_año_reves,x)
# posicion al reves
pos_evento_mas_sig2= None
a2=0
for x in lt.iterator(lista_radio_año_reves):
    a+=1
    if evento_mas_sig["title"] == x["title"]:
        pos_evento_mas_sig2=a2

# cambiar lista radio año    YA
# cambiar posicion          YA
# cambiar menores eventos   YA
if lt.size(lista_radio_año_reves)> pos_evento_mas_sig2+numero_N_eventos:
    menores_eventos=  lt.subList(lista_radio_año_reves,pos_evento_mas_sig2,numero_N_eventos)
if lt.size(lista_radio_año_reves)< pos_evento_mas_sig2+numero_N_eventos:
    posibles=lt.size(lista_radio_año_reves)-pos_evento_mas_sig2
    menores_eventos=  lt.subList(lista_radio_año_reves,pos_evento_mas_sig2,numero_N_eventos)
```

```
resp=lt.newList()
for x in lt.iterator(menores_eventos):
    lt.addLast(resp,x)
for x in lt.iterator(mayores_eventos):
    lt.addLast(resp,x)

merg.sort(resp, sort_por_fecha_des)
mas=evento_mas_sig
ya=[mas,resp]
return ya
```

La función req_6 procesa eventos de un año específico, filtrando aquellos dentro de un radio determinado alrededor de un punto geográfico dado y ordenándolos por fecha y significancia. Determina el evento más significativo dentro de este conjunto y selecciona un número específico de eventos antes y después de este, proporcionando un análisis focalizado en torno al evento clave. Retorna una lista con el evento más significativo y una lista de eventos circundantes organizados por fecha en orden descendente.

Entrada	analyzer: Estructura de datos que contiene información relevante. año: Año específico para filtrar eventos. lati: Latitud para calcular la distancia. long: Longitud para calcular la distancia. radio: Radio de distancia para incluir eventos. numero_N_eventos: Número de eventos a incluir después del evento más significativo.
Salidas	Una lista que contiene el evento más significativo y una lista combinada de eventos antes y después del evento más significativo, todos dentro del año y radio especificados.
Implementado (Sí/No)	Sí. Implementación grupal.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Conversión de fechas	O(1)
Creación de una nueva lista (lt.newList)	O(1)
Obtener el tamaño de la lista final(lt.size)	O(1)
(lt.iterator)	O(1)
Obtener una sublista de la lista final (lt.subList)	O(n)
Ordena la lista final (merg.sort)	O(n log n)
Obtener una lista de resultados (mp.get)	O(1)
for x in lt.iterator(menores_eventos): lt.addLast(resp, x) ……..	O(n log n)

for x in lt.iterator(lista_radio_año):
if int(x["sig"]) > ev_sig:
….
	O(n log n)

om.Values	O(n)
TOTAL	O(n log n)

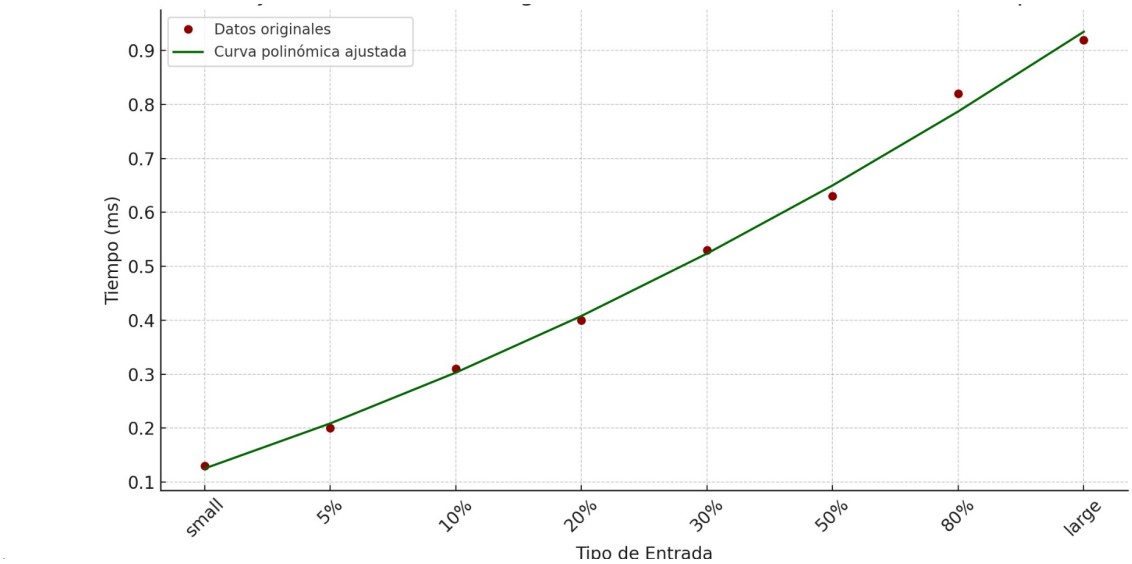
Pruebas Realizadas

Procesadores	**11th Gen Intel(R) Core(TM) i5-11400H @ 2.70GHz 2.69 GHz**

Memoria RAM	8 GB
Sistema Operativo	Windows 11

Entrada	Tiempo (ms)
small	0.13
5 pct	0.20
10 pct	0.31
20 pct	0.40
30 pct	0.53
50 pct	0.63
80 pct	0.82
large	0.92

Gráfica



Análisis

La función req_6 procesa eventos basándose en un año, ubicación geográfica y un radio de búsqueda específicos para encontrar eventos significativos dentro de ese rango. Convierte dos cadenas de fecha en objetos datetime para delimitar el año de interés y luego recupera todos los eventos del año de un índice llamado "fechaIndex".

Después, filtra los eventos que están dentro del radio especificado usando una función calcular_distancia_tierra y los almacena tanto en una lista como en un mapa ordenado por fecha. La complejidad de filtrar los eventos es proporcional al número total de eventos y a la complejidad de la función de cálculo de distancia.

Identifica el evento más significativo basado en una propiedad "sig" y ordena los eventos por fecha. Luego, busca los eventos que siguen al más significativo hasta llegar a un número deseado numero_N_eventos. Si el número de eventos es menor que el deseado, ajusta el rango de búsqueda.

Finalmente, invierte la lista de eventos, repite el proceso de búsqueda para eventos anteriores al más significativo y combina ambos conjuntos de eventos en una lista final, que se vuelve a ordenar por fecha. Retorna el evento más significativo junto con la lista de eventos anteriores y posteriores a este. La eficiencia de la función está ligada a las operaciones de filtrado, ordenamiento y la gestión de listas y mapas.

Requerimiento 7

```python
def req_7(data_structs, year, region, property, bins):
    earthquakes_in_zone = mp.get(data_structs["earthquakes_by_zone_year"], region)

    if earthquakes_in_zone is None:
        return None

    earthquakes_in_year = mp.get(me.getValue(earthquakes_in_zone), year)

    if earthquakes_in_year is None:
        return None

    earthquakes_in_year = me.getValue(earthquakes_in_year)

    earthquakes_by_property = om.newMap(omaptype="RBT")

    for earthquake in lt.iterator(earthquakes_in_year):
        exists = om.get(earthquakes_by_property, earthquake[property])
        if exists is not None:
            lst = me.getValue(exists)
            lt.addLast(lst, earthquake)
        else:
            lst = lt.newList(datastructure="SINGLE_LINKED")
            lt.addLast(lst, earthquake)
            om.put(earthquakes_by_property, earthquake[property], lst)

    min_property = om.minKey(earthquakes_by_property)
    max_property = om.maxKey(earthquakes_by_property)

    gap = round((max_property - min_property) / bins, 3)

    categories = []
    values = []

    current_min = min_property

    for i in range(bins):
        next = round(current_min + gap, 3)
        if i == bins - 1:
            next = max_property
        categories.append(f"{current_min}-{next}")

        amount = 0

        for lst in lt.iterator(om.values(earthquakes_by_property, current_min, next)):
            amount += lt.size(lst)

        values.append(amount)

        current_min = next

    merg.sort(earthquakes_in_year, sort_time)

    return earthquakes_in_year, categories, values
```

Descripción

La función req_7 clasifica los terremotos por una propiedad específica dentro de una región y año dados, dividiéndolos en una cantidad determinada de intervalos (bins). Recupera los terremotos de la región y el año especificados, luego crea un mapa ordenado para agrupar los terremotos según el valor de la propiedad elegida. Calcula el rango de valores (gap) para los intervalos y cuenta cuántos terremotos caen en cada uno. Retorna los terremotos ordenados por tiempo, junto con una lista de categorías que representan los intervalos de la propiedad y una lista de valores que indica la cantidad de terremotos en cada intervalo.

Entrada	data_structs: Estructura de datos que contiene información de terremotos. year: Año específico para buscar terremotos. region: Región geográfica específica para buscar terremotos. property: Propiedad específica de los terremotos para clasificar. bins: Número de intervalos para dividir la propiedad seleccionada.
Salidas	La lista de terremotos del año y región especificados, las categorías de intervalos basadas en la propiedad seleccionada y la cantidad de terremotos en cada intervalo.
Implementado (Sí/No)	Sí. Implementación grupal.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

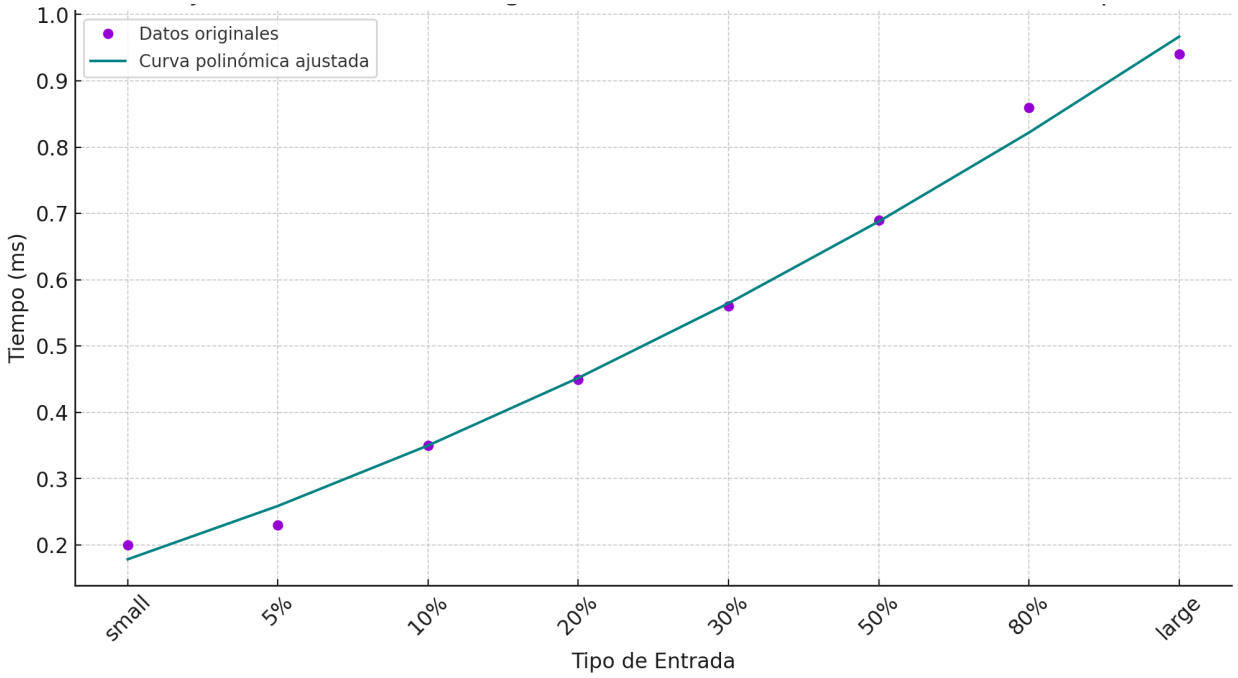
Pasos	**Complejidad**
Crear una nueva lista (lt.newList())	O(1)
Obtener una lista de resultados (mp.get)	O(1)
Obtener el tamaño de la lista final(lt.size)	O(1)
(lt.iterator)	O(1)
Obtener una sublista de la lista final (lt.subList)	O(n)
Ordena la lista final (merg.sort)	O(n log n)
om.newMap(omaptype="RBT")	O(1)
for earthquake in lt.iterator(earthquakes_in_year): exists = om.get(earthquakes_by_property, earthquake[property]) …..	O(n log n)
om.minKey(earthquakes_by_property) om.maxKey(earthquakes_by_property)	O(log n)
round((max_property - min_property) / bins, 3)	O(1)
me.getValue	O(1)
TOTAL	O(n log n)

Pruebas Realizadas

Procesadores	11th Gen Intel(R) Core(TM) i5-11400H @ 2.70GHz 2.69 GHz
Memoria RAM	8 GB
Sistema Operativo	Windows 11

Entrada	Tiempo (ms)
small	0.20
5 pct	0.23
10 pct	0.35
20 pct	0.45
30 pct	0.56
50 pct	0.69
80 pct	0.86
large	0.94

Gráfica



Análisis

La función req_7 organiza y clasifica terremotos de una región y año específicos, basándose en una propiedad particular, y los divide en una cantidad definida de intervalos o "bins". Inicialmente, intenta recuperar los terremotos de una zona y año específicos de una estructura de datos proporcionada. Si no encuentra datos para la región o el año dados, retorna None.

Si hay datos disponibles, la función procede a organizar los terremotos basándose en la propiedad de interés (por ejemplo, magnitud, profundidad, etc.) utilizando un mapa ordenado. Para cada terremoto, verifica si ya existe un grupo para el valor de la propiedad y, de ser así, añade el terremoto a ese grupo; si no, crea un nuevo grupo y lo añade al mapa.

Después, calcula el valor mínimo y máximo de la propiedad de los terremotos y establece el tamaño de cada intervalo o "bin". Con estos datos, crea categorías para los intervalos y cuenta cuántos terremotos caen en cada uno de ellos.

Finalmente, ordena la lista de terremotos por tiempo y devuelve tres resultados: la lista ordenada de terremotos, las categorías de intervalos y la cantidad de terremotos en cada intervalo. La eficiencia de la función está ligada a la eficacia de las operaciones del mapa ordenado y a la cantidad de terremotos a clasificar.