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

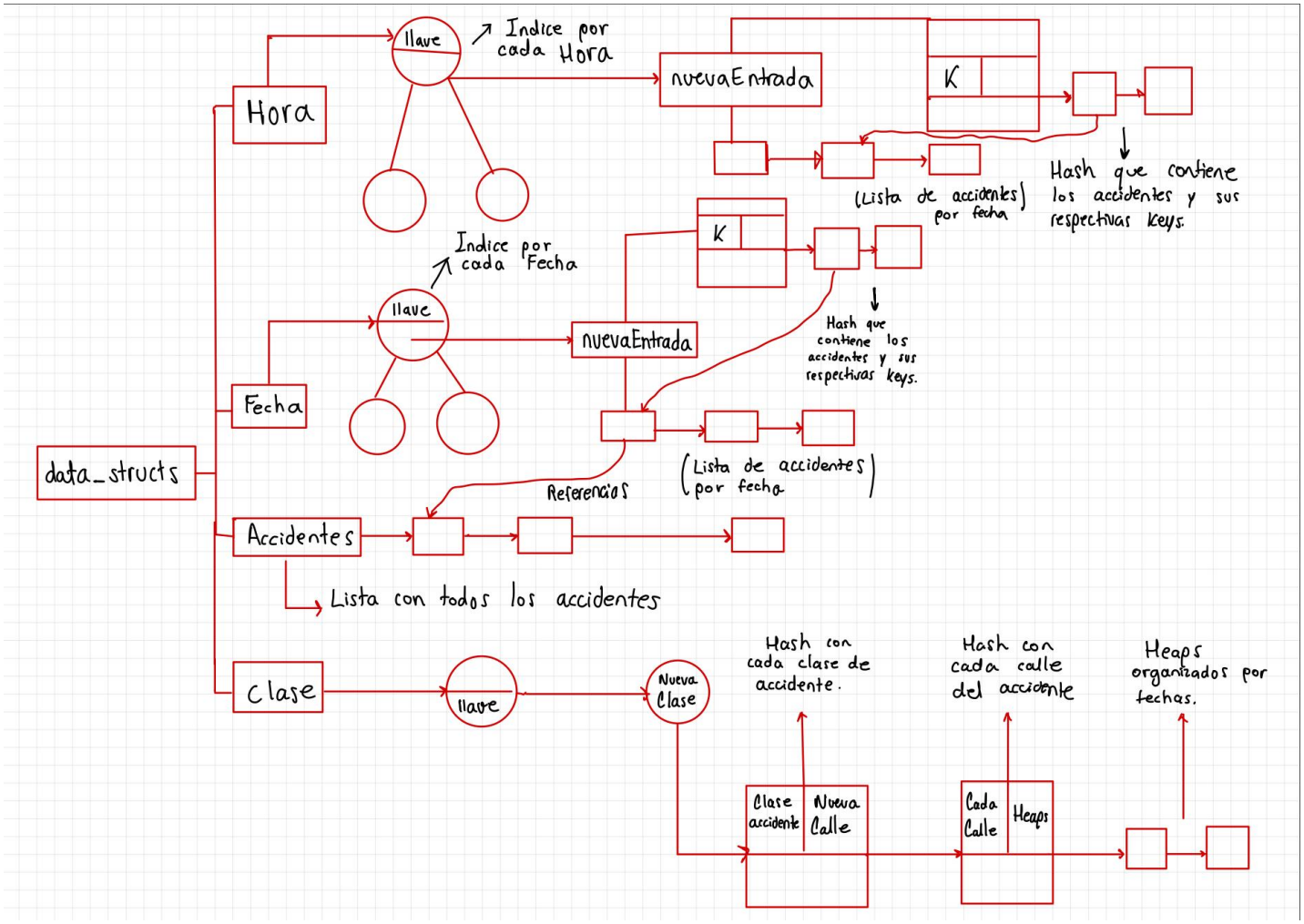
David Fernando Caro Ladino, d.carol@uniandes.edu.co, 202222073

David Alejandro Abril Medina, d.abrilm@uniandes.edu.co, 202224328

Gabriel Leonardo Martínez Martínez, gl.martinez@uniandes.edu.co, 202214559

Carga de datos

Plantilla para documentar y analizar las diferentes cargas de datos implementadas.

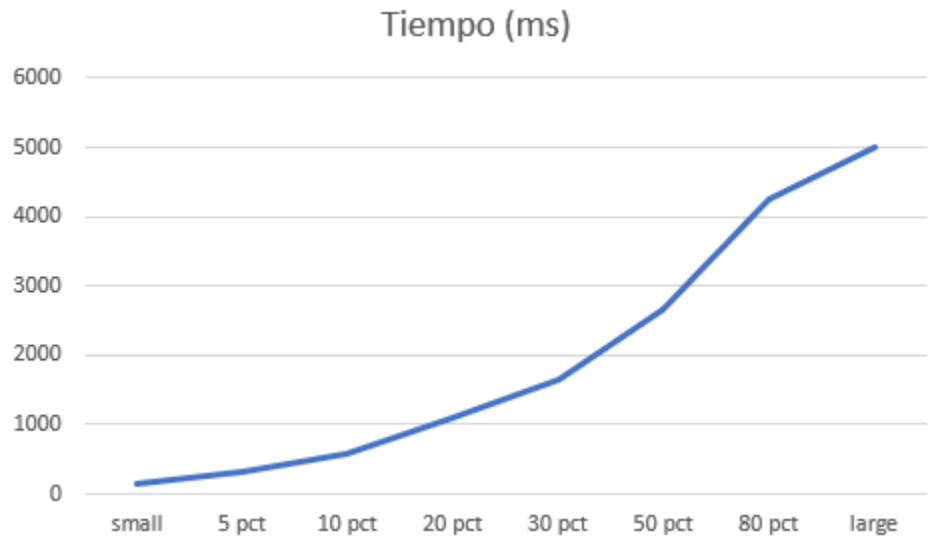


Entrada	Memoria(kB)	Tiempo (ms)

small	6713.604	148.576
5 pct	23091.650	316.613
10 pct	40600.124	580.303
20 pct	76009.632	1087.796
30 pct	108940.661	1642.237
50 pct	174513.761	2657.987
80 pct	275812.895	4252.673
large	340552.496	5006.978

Gráficas

Las gráficas con la representación de las pruebas realizadas.



Requerimiento 1

Plantilla para documentar y analizar cada uno de los requerimientos.

Descripción

```python
def req_1(data_structs, fechaInicial, fechaFinal):
    """
    Función que soluciona el requerimiento 1
    """
    # TODO: Realizar el requerimiento 1
    finalList = lt.newList("ARRAY_LIST")
    mapa = data_structs["Fecha"]
    rango = om.values(mapa, fechaInicial, fechaFinal)
    for fechas in lt.iterator(rango):
        for i in lt.iterator(fechas["lstaccidentes"]):
            lt.addLast(finalList, i)
    merg.sort(finalList, cmpreq1)
    return finalList
```

Se sacan los valores del data structs que corresponden con el rango ingresado por parámetro, posteriormente se ingresan los valores en una lista y finalmente se ordenan del más reciente al más antiguo

Entrada	El datastructs con los datos, una fecha inicial y una fecha final
Salidas	Los accidentes que ocurrieron dentro del rango ingresado ordenados de más reciente a más antiguo
Implementado (Sí/No)	Si, se implementó por los tres integrantes

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Paso 1: crear una lista final donde se almacenará la respuesta	O(1)
Paso 2: con .values() obtener los accidentes dentro del rango requerido	O(log(N))
Paso 3: se agregan los valores a la lista final	O(log(N))
Paso 4: se ordenan los valores desde el más reciente en adelante	O(Nlog(N))
TOTAL	***O(Nlog(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).

Procesadores	**12th Gen Intel(R) Core(TM) i7-12700 2.10 GHz**
Memoria RAM	32 GB

Sistema Operativo	Windows 10

Entrada	Tiempo (ms)
small	1.041
5 pct	4.759
10 pct	6.136
20 pct	17.043
30 pct	27.454
50 pct	47.506
80 pct	78.656
large	95.837

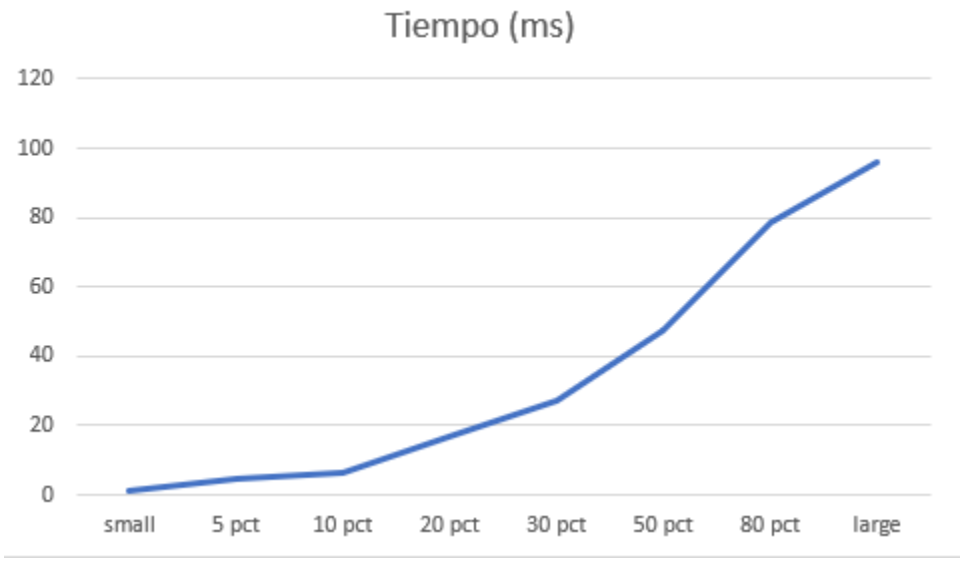
Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

Muestra	Salida	Tiempo (ms)
small	Dato1	1.041
5 pct	Dato2	4.759
10 pct	Dato 3	6.136
20 pct	Dato4	17.043
30 pct	Dato 5	27.454
50 pct	Dato 6	47.506
80 pct	Dato7	78.656
large	Dato 8	95.837

Gráficas

Las gráficas con la representación de las pruebas realizadas.



Análisis

Se puede considerar O(Nlog(N)) como una complejidad aceptable, ya que aun se considera óptima. Respecto a los tiempos tomados se puede ver un crecimiento constante. Es bueno aclarar que se priorizo velocidad de respuesta sobre uso de memoria, por lo que la complejidad podría ser de O(N) si se implementara otro método de ordenamiento.

Requerimiento 2

Plantilla para documentar y analizar cada uno de los requerimientos.

Descripción

```python
def req_2(data_structs, horaInicial, horaFinal, mes, año):
    """
    Función que soluciona el requerimiento 2
    """
    # TODO: Realizar el requerimiento 2
    finalList = lt.newList("ARRAY_LIST")
    mapa = data_structs["Hora"]
    rango = om.values(mapa, horaInicial, horaFinal)
    for horas in lt.iterator(rango):
        for i in lt.iterator(horas["lstaccidentes"]):
            if str(i["MES_OCURRENCIA_ACC"]) == mes:
                if str(i["ANO_OCURRENCIA_ACC"]) == año:
                    respuesta = i
                    lt.addLast(finalList, respuesta)
    merg.sort(finalList, cmpreq2)
    return finalList
```

Se sacan los valores del data structs creado con las fechas, se sacan los valores que están dentro del rango de horas y minutos ingresados por parámetro, después se recorre el rango y se filtran los datos con el mes y el año ingresados por parámetro, para finalmente ser agregados a una lista y ser retornados al usuario

Entrada	El datastructs con los datos, una hora inicial, una hora final, un mes y un año
Salidas	Los accidentes que ocurrieron dentro del rango de horas y minutos, el mes y el año ingresado
Implementado (Sí/No)	Si, se implementa por los tres integrantes

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Paso 1: crear una lista final donde se almacenará la respuesta	O(1)
Paso 2: con .values() obtener los accidentes dentro del rango requerido	O(log(N))
Paso 3: buscar cuales de estos valores corresponden con el año y el mes ingresados	O(N)
Paso 4: se agregan los valores a la lista final retornada al usuario	O(log(N))
Paso 5: se ordenan los valores de la lista previamente ordenada con un cmp que compara fecha y hora.	O(N(log(N)))
TOTAL	***O(N(log(N)))***

Pruebas Realizadas

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

Procesadores	12th Gen Intel(R) Core(TM) i7-12700 2.10 GHz
Memoria RAM	32 GB
Sistema Operativo	Windows 10

Entrada	Tiempo (ms)
small	0.558
5 pct	2.800
10 pct	2.864
20 pct	4.742
30 pct	10.029
50 pct	15.446
80 pct	20.781
large	25.933

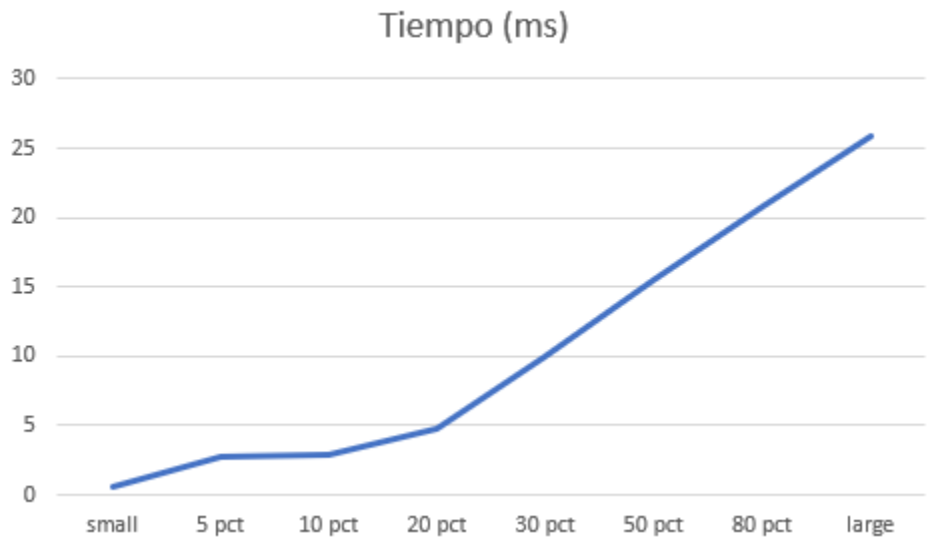
Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

Muestra	Salida	Tiempo (ms)
small	Dato1	0.558
5 pct	Dato2	2.800
10 pct	Dato3	2.864
20 pct	Dato4	4.742
30 pct	Dato5	10.029
50 pct	Dato6	15.446
80 pct	Dato7	20.781
large	Dato8	25.933

Gráficas

Las gráficas con la representación de las pruebas realizadas.



Análisis

Se puede considerar O(Nlog(N)) como una complejidad aceptable, ya que aún se considera óptima. Respecto a los tiempos se evidencia que es de crecimiento constante, lo cual indica un resultado favorable para el proceso del requerimiento

Requerimiento 3

Plantilla para documentar y analizar cada uno de los requerimientos.

Descripción

```
def req_3(data_structs, clase, calle):
    mapa = data_structs["Clase del accidente"]
    hash = me.getValue(mp.get(mapa, clase))
    lista = me.getValue(mp.get(hash["data"], calle))
    lista = lista["data"]
    lista = merg.sort(lista, cmpreq1)
    size = lt.size(lista)
    if size > 3:
        finalList = primerosTres(lista)
    else:
        finalList
    return size, finalList
```

Se sacan los valores del datastructs creado con el tipo de accidente, se sacan los accidentes de esa clase y esa calle, se ordena la lista por fecha y hora, finalmente se retornan los 3 más recientes o en su defecto si hay menos retornar esa cantidad.

Entrada	El datastructs con los datos, la clase de accidente y una calle
Salidas	Los 3 accidentes más recientes que ocurrieron en una vía de una clase específica
Implementado (Sí/No)	Si se implementa por Alejandro Abril

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Paso 1: sacar los accidentes con la misma clase del parámetro	O(N)
Paso 2: sacar los accidentes con la misma calle del parámetro	O(N)
Paso 3: Se ordena la lista.	O(Mlog(M))
TOTAL	***O(Mlog(M))***

Siendo m una lista filtrada más pequeña que n por lo que m(log(m)) es mucho más pequeño que 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).

Procesadores	**12th Gen Intel(R) Core(TM) i7-12700 2.10 GHz**
Memoria RAM	32 GB
Sistema Operativo	Windows 10

Entrada	**Tiempo (ms)**
small	1.516
5 pct	9.778
10 pct	24.098
20 pct	52.802
30 pct	84.815
50 pct	110.217
80 pct	142.235
large	170.259

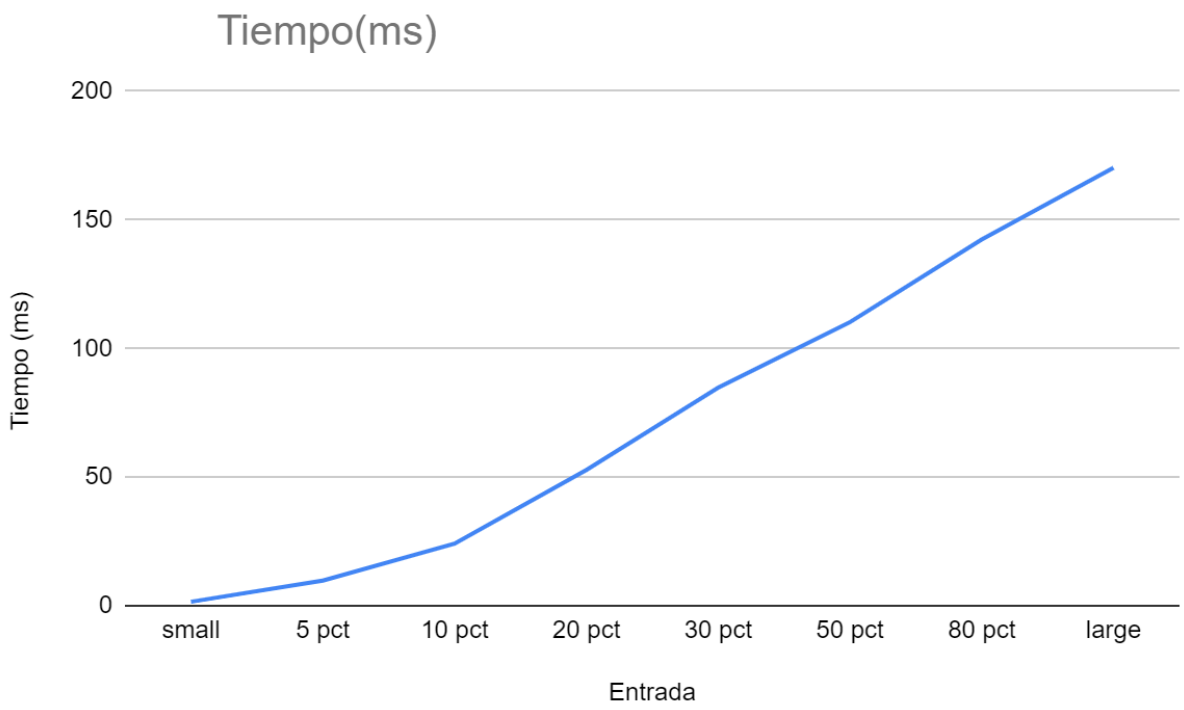
Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

Muestra	Salida	Tiempo (ms)
small	Dato1	1.516
5 pct	Dato2	9.778
10 pct	Dato3	24.098
20 pct	Dato4	52.802
30 pct	Dato5	84.815
50 pct	Dato6	110.217
80 pct	Dato7	142.235
large	Dato8	170.259

Gráficas

Las gráficas con la representación de las pruebas realizadas.



Análisis

En cuanto a complejidad, este es Mlog(M) debido a que está organizando una lista filtrada más pequeña que n , en busca de priorizar el menor tiempo posible, tal como se muestra en las gráficas de tiempo, la cual crece a medida que lo hace el tamaño de la muestra de manera no muy drástica con una tendencia

lineal. Por lo anterior se considera que se cumplió con el requerimiento y se da la respuesta en el mejor tiempo posible.

Requerimiento 4

Plantilla para documentar y analizar cada uno de los requerimientos.

Descripción

```
def req_4(data_structs, fechaInicial, fechaFinal, gravedad):
    """
    Función que soluciona el requerimiento 4
    """
    # TODO: Realizar el requerimiento 4
    finalList = lt.newList("ARRAY_LIST")
    mapa = data_structs["Fecha"]
    rango = om.values(mapa, fechaInicial, fechaFinal)
    for fechas in lt.iterator(rango):
        for i in lt.iterator(fechas["lstaccidentes"]):
            if str(i["GRAVEDAD"]) == gravedad:
                respuesta = i
                lt.addLast(finalList, respuesta)
    merg.sort(finalList, cmpreq1)
    size =lt.size(finalList)
    if size > 5:
        subList = primerosCinco(finalList)
    else:
        subList = finalList
    return subList, finalList
```

Se sacan los valores del datastructs que corresponden con el rango ingresado por parámetro, posteriormente se ingresan los valores en una lista y finalmente se ordenan del más reciente al más antiguo

Entrada	El datastructs con los datos, una fecha inicial, una fecha final y la gravedad del accidente
Salidas	Los 5 accidentes más recientes que ocurrieron dentro del rango ingresado y para una gravedad específica

Implementado (Sí/No)	Si se implementó por Gabriel Martinez

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	Complejidad
Paso 1: crear una lista final donde se almacenará la respuesta	O(1)
Paso 2: con .values() obtener los accidentes dentro del rango requerido	O(N)
Paso 3: se filtran los valores del rango que poseen la misma gravedad que la ingresada	O(log(N))
Paso 4: se ordenan los valores por hora y fecha	O(Nlog(N))
TOTAL	***O(Nlog(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).

Procesadores	**12th Gen Intel(R) Core(TM) i7-12700 2.10 GHz**
Memoria RAM	32 GB
Sistema Operativo	Windows 10

Entrada	Tiempo (ms)
small	1.799
5 pct	5.470
10 pct	11.201
20 pct	20.642
30 pct	31.096
50 pct	54.063
80 pct	90.674
large	115.569

Tablas de datos

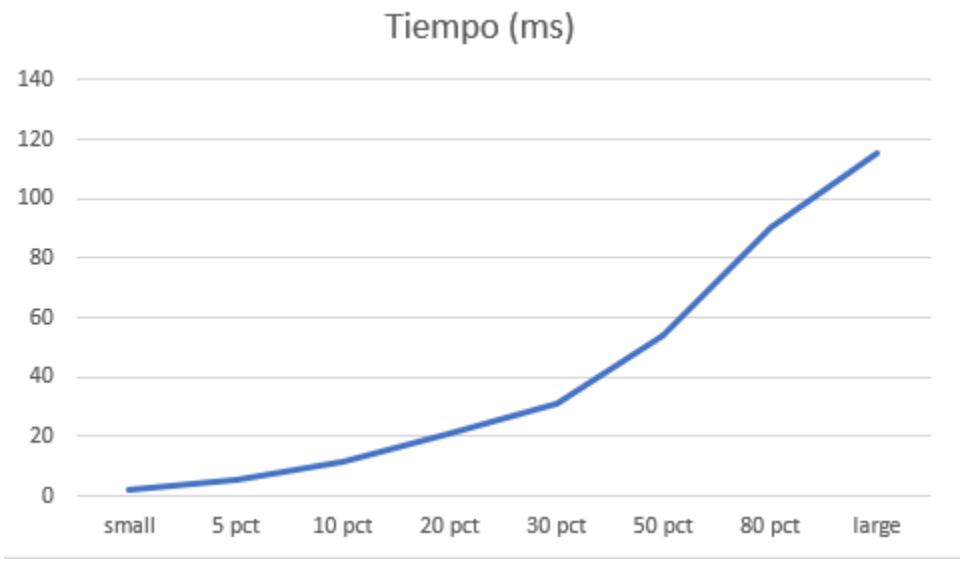
Las tablas con la recopilación de datos de las pruebas.

Muestra	Salida	Tiempo (ms)
small	Dato1	1.799
5 pct	Dato2	5.470

10 pct	Dato3	11.201
20 pct	Dato4	20.642
30 pct	Dato5	31.096
50 pct	Dato6	54.063
80 pct	Dato7	90.674
large	Dato8	115.569

Gráficas

Las gráficas con la representación de las pruebas realizadas.



Análisis

Respecto al tiempo se puede decir que se cumplió con el objetivo del requerimiento, ya que este crece a medida que lo hace la cantidad de datos. La complejidad al igual que en requerimientos anteriores es O(Nlog(N)) por el ordenamiento de la lista, e igualmente podría ser de menor complejidad si se implementara otro tipo de sort().

Requerimiento 5

Plantilla para documentar y analizar cada uno de los requerimientos.

Descripción

```python
def req_5(data_structs, localidad, mes, año):
    """
    Función que soluciona el requerimiento 5
    """
    # TODO: Realizar el requerimiento 5
    mapa = data_structs["Fecha"]
    min_año = datetime.datetime(2015, 1, 1)
    max_año = datetime.datetime(2022, 12, 31)
    rango = om.values(mapa, min_año.date(), max_año.date())
    finalList = lt.newList("ARRAY_LIST")
    for fechas in lt.iterator(rango):
        for i in lt.iterator(fechas["lstaccidentes"]):
            if str(i["LOCALIDAD"]) == localidad:
                if str(i["MES_OCURRENCIA_ACC"]) == mes:
                    if str(i["ANO_OCURRENCIA_ACC"]) == año:
                        respuesta = i
                        lt.addLast(finalList, respuesta)
    merg.sort(finalList, cmpreq1)
    size = lt.size(finalList)
    if size > 10:
        subList = ultimosDiez(finalList)
    else:
        subList = finalList
    return finalList, subList
```

Se reemplazan los espacios y se pone el texto en mayúsculas, se toma el datastructs ordenado por accidente y se itera para filtrar los nodos que contienen la localidad, el mes y el año ingresado por parámetro, finalmente se ordena y se retorna la información.

Entrada	El datastructs con los datos, una localidad, un mes y un año
Salidas	Los 10 accidentes menos recientes para un mes y un año en una localidad de la ciudad
Implementado (Sí/No)	Si se implementó por David Caro

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Paso 1: crear una lista final donde se almacenará la respuesta	O(1)
Paso 2: con .values() obtener los accidentes dentro del rango requerido	O(N)

Paso 2: se comprueba si los datos coinciden con los 3 parámetros ingresados y se añaden a la lista final	O(N)
Paso 3: se ordenan desde el más antiguo hasta el más reciente	O(Nlog(N))
Paso 4: se retornan los 10 primeros valores o en su defecto los que haya al ser menos de 10	O(1)
TOTAL	***O(Nlog(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).

Procesadores	**12th Gen Intel(R) Core(TM) i7-12700 2.10 GHz**
Memoria RAM	32 GB
Sistema Operativo	Windows 10

Entrada	Tiempo (ms)
small	8.141
5 pct	8.440
10 pct	11.501
20 pct	20.046
30 pct	28.445
50 pct	44.759
80 pct	71.851
large	91.906

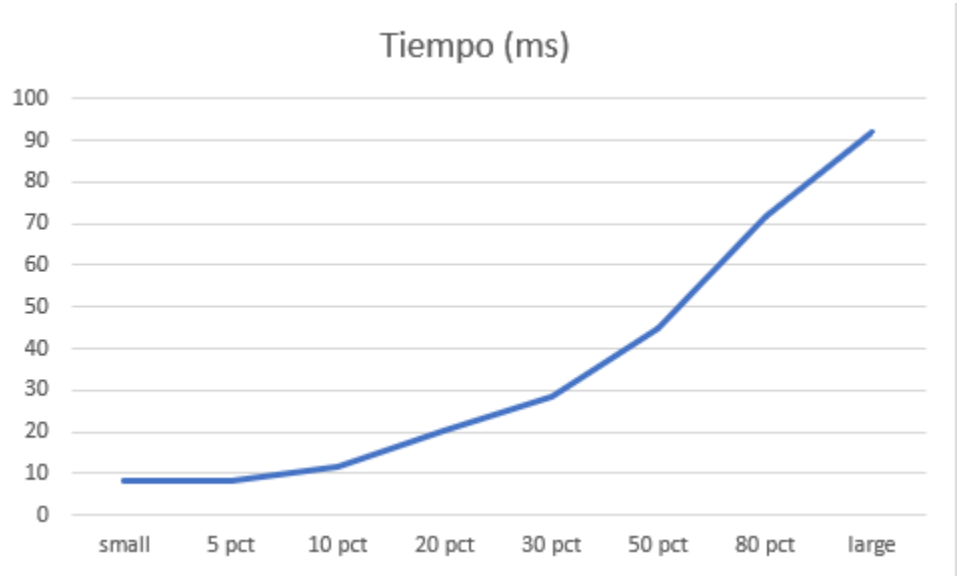
Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

Muestra	Salida	Tiempo (ms)
small	Dato1	8.141
5 pct	Dato2	8.440
10 pct	Dato3	11.501
20 pct	Dato4	20.046
30 pct	Dato5	28.445
50 pct	Dato6	44.759
80 pct	Dato7	71.851
large	Dato8	91.906

Gráficas

Las gráficas con la representación de las pruebas realizadas.



Análisis

Respecto al tiempo se puede decir que se cumplió con el objetivo del requerimiento, ya que este crece a medida que lo hace la cantidad de datos. La complejidad al igual que en requerimientos anteriores es O(Nlog(N)) por el ordenamiento de la lista, e igualmente podría ser de menor complejidad si se implementara otro tipo de sort() o si se accediera directamente a la lista de los accidentes en lugar del rango del árbol binario de las fechas. .

Requerimiento 6

Plantilla para documentar y analizar cada uno de los requerimientos.

Descripción

```
def req_6(data_structs, mes, año, latitud1, longitud1, radio, num_acc):
    mapa = data_structs["Fecha"]
    min_año = datetime.datetime(2015,1,1)
    max_año = datetime.datetime(2022,12,31)
    rango = om.values(mapa, min_año.date(), max_año.date())
    finalList = lt.newList("ARRAY_LIST")
    for fechas in lt.iterator(rango):
        for i in lt.iterator(fechas["lstaccidentes"]):
            if str(i["ANO_OCURRENCIA_ACC"]) == año and str(i["MES_OCURRENCIA_ACC"]) == mes:
                latitud = float(i["LATITUD"])
                longitud = float(i["LONGITUD"])
                distancia = haversine(longitud, latitud, longitud1, latitud1)
                if distancia <= radio:
                    i["DISTANCIA"] = distancia
                    lt.addLast(finalList, i)
    size = lt.size(finalList)
    finalList = merg.sort(finalList, compararDistancia)
    if size > num_acc:
        finalList = lt.subList(finalList, 1, num_acc)
    else:
        finalList = finalList
    return finalList



def haversine(longitud1, latitud1, longitud2, latitud2):
    longitud1 = radians(longitud1)
    longitud2 = radians(longitud2)
    latitud1 = radians(latitud1)
    latitud2 = radians(latitud2)
    dlon = longitud2 - longitud1
    dlat = latitud2 - latitud1
    a = sin(dlat/2)**2 + cos(latitud1) * cos(latitud2) * sin((dlon/2)**2)
    c = 2 * sin(sqrt(sin((dlat/2)**2) + cos(latitud1) * cos(latitud2) * sin((dlon/2)**2)))
    r = 6371
    return c * r
```

Se toma el datastructs ordenado por fechas y se sacan los valores en el rango de los datos, luego se filtran los datos por el mes y año ingresados por el usuario, luego se asignan a las variables latitud y longitud los valores de la iteración actual y en la variable distancia se utiliza una función externa que realiza distintos cálculos para hallar, finalmente se comprueba que esta no se salga del radio de búsqueda y se añade a la lista final.

Entrada	El datastructs con los datos, un mes, un año, una latitud, una longitud, un radio y una cantidad de accidentes que el usuario quiera mostrar

Salidas	Los N accidentes que ocurrieron dentro de una zon para un mes y un añO
Implementado (Sí/No)	Si se implementó por los tres integrantes

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Paso 1: crear un rango con todos los valores del mapa	O(N)
Paso 2: se itera el rango y la lista de accidentes para filtrar la información requerida	O(N^2)
Paso 3: se ordena la lista con merge	O(Nlog(N))
Paso 4: se comprueba la cantidad de datos para saber qué información retornar	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	**12th Gen Intel(R) Core(TM) i7-12700 2.10 GHz**
Memoria RAM	32 GB
Sistema Operativo	Windows 10

Entrada	**Tiempo (ms)**
small	3.349
5 pct	7.193
10 pct	11.116
20 pct	20.428
30 pct	24.347
50 pct	37.400
80 pct	67.700
large	74.937

Tablas de datos

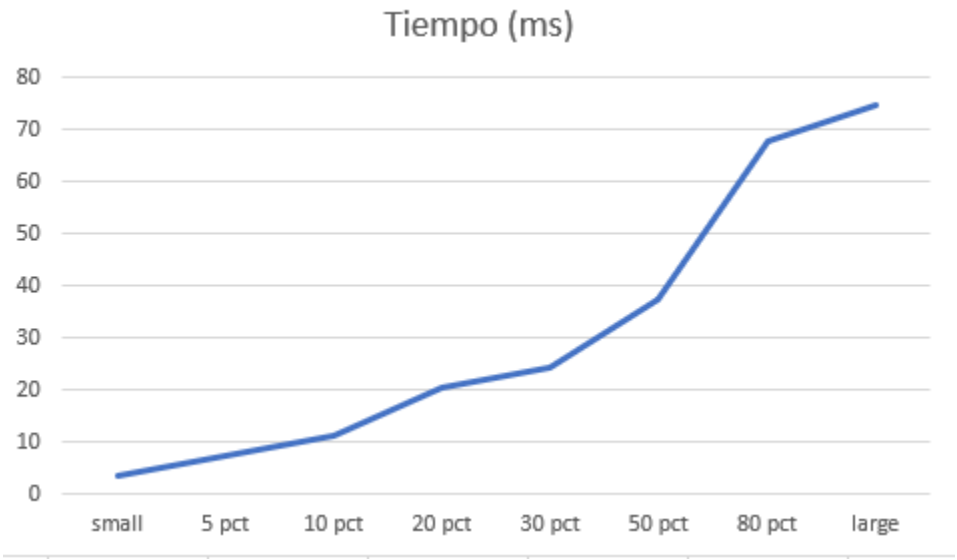
Las tablas con la recopilación de datos de las pruebas.

Muestra	**Salida**	**Tiempo (ms)**
small	Dato1	3.349

5 pct	Dato2	7.193
10 pct	Dato3	11.116
20 pct	Dato4	20.428
30 pct	Dato5	24.347
50 pct	Dato6	37.400
80 pct	Dato7	67.700
large	Dato8	74.937

Gráficas

Las gráficas con la representación de las pruebas realizadas.



Análisis

A pesar de los tiempos favorables, si cabe resaltar que la complejidad es algo elevada, esto podría ser mejorado haciendo que el rango filtra de una vez por los parámetros pasados por el usuario, sin embargo, no hubo tiempo necesario para cambiarlos. El resto del requerimiento se puede considerar exitoso.

Requerimiento 7

Plantilla para documentar y analizar cada uno de los requerimientos.

Descripción

```python
def req_7(data_structs, mes, año):
    """
    Función que soluciona el requerimiento 7
    """
    # TODO: Realizar el requerimiento 7
    finalList = lt.newList("ARRAY_LIST")
    mapa = data_structs["Fecha"]
    min_año = datetime.datetime(2015,1,1)
    max_año = datetime.datetime(2022,12,31)
    rango = om.values(mapa, min_año.date(), max_año.date())
    finalMap = mp.newMap(31, maptype="CHAINING",
                         loadfactor=0.5,
                         cmpfunction=None)
    for fechas in lt.iterator(rango):
        for i in lt.iterator(fechas["lstaccidentes"]):
            if str(i["MES_OCURRENCIA_ACC"]) == mes and str(i["ANO_OCURRENCIA_ACC"]) == año:
                respuesta = i
                lt.addLast(finalList, respuesta)
    merg.sort(finalList, cmpreq7)
    subList = lt.newList("ARRAY_LIST")
    fecha = lt.getElement(finalList, 1)

    fecha = fecha["FECHA_OCURRENCIA_ACC"]
    fecha1 = datetime.datetime.strptime(fecha, "%Y/%m/%d")
    fecha = int(fecha[8:10])
    fechaNumero = datetime.timedelta(days = 1)
    for x in range(1, 32):
        subList = lt.newList("ARRAY_LIST")
        for i in lt.iterator(finalList):
            if datetime.datetime.strptime(i["FECHA_OCURRENCIA_ACC"] , "%Y/%m/%d") == fecha1:
                lt.addLast(subList, i)
        if lt.isEmpty(subList) == False:
            mp.put(finalMap, fecha, subList)
        fecha1 += fechaNumero
        fecha += 1
```

```python
    diccionario_grafica = {"00:00:00": 0,"01:00:00": 0,
                           "02:00:00": 0,"03:00:00": 0,
                           "04:00:00": 0,"05:00:00": 0,
                           "06:00:00": 0,"07:00:00": 0,
                           "08:00:00": 0,"09:00:00": 0,
                           "10:00:00": 0,"11:00:00": 0,
                           "12:00:00": 0,"13:00:00": 0,
                           "14:00:00": 0,"15:00:00": 0,
                           "16:00:00": 0,"17:00:00": 0,
                           "18:00:00": 0,"19:00:00": 0,
                           "20:00:00": 0,"21:00:00": 0,
                           "22:00:00": 0,"23:00:00": 0}
    for i in lt.iterator(finalList):
        hora = i["HORA_OCURRENCIA_ACC"]

        if len(hora) == 7:
            hora = "0" + hora
        hora = hora[0:2]
        hora = hora + ":00:00"
        diccionario_grafica[hora] += 1
    size = lt.size(finalList)
    return finalMap, diccionario_grafica, size
```

Se sacan los valores del datastructs que corresponden con el rango ingresado por parámetro, posteriormente se ingresan los valores en una lista y finalmente se ordenan del más reciente al más antiguo

Entrada	El datastructs con los datos, una fecha inicial y una fecha final
Salidas	Los accidentes que ocurrieron dentro del rango ingresado ordenados de más reciente a más antiguo
Implementado (Sí/No)	Si se implementa por los tres integrantes

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Paso 1: crear una lista final donde se almacenará la respuesta	O(1)
Paso 2: con .values() obtener los accidentes dentro del rango requerido	O(log(N))
Paso 3: se agregan los valores a la lista final	O(log(N))
Paso 4: se ordenan los valores desde el más reciente en adelante	O(Nlog(N))
Paso 5: se hace un nuevo mapa al cual se le agregan los valores de la lista filtrada	O(M)
Paso 6: se itera la lista filtrada para sacar el diccionario para poder graficar	O(M)
TOTAL	***O(Nlog(N))***

Siendo M una lista filtrada más pequeña que n por lo que M es mucho más pequeño que 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).

Procesadores	**12th Gen Intel(R) Core(TM) i7-12700 2.10 GHz**
Memoria RAM	32 GB
Sistema Operativo	Windows 10

Entrada	**Tiempo (ms)**
small	7.138
5 pct	33.296
10 pct	66.628
20 pct	132.382
30 pct	241.090
50 pct	359.907

80 pct	589.305
large	753.429

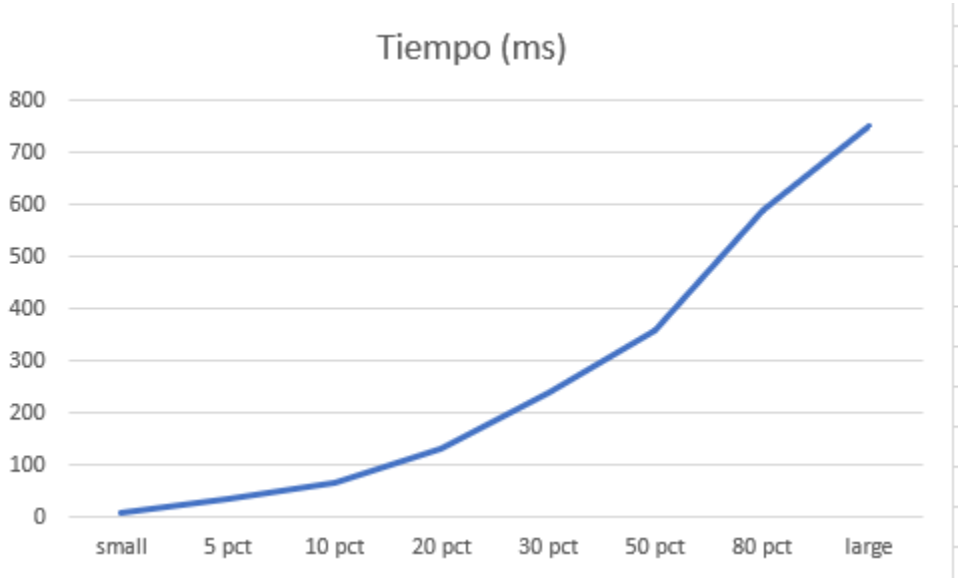
Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

Muestra	Salida	Tiempo (ms)
small	Dato1	7.138
5 pct	Dato2	33.296
10 pct	Dato3	66.628
20 pct	Dato4	132.382
30 pct	Dato5	241.090
50 pct	Dato6	359.907
80 pct	Dato7	589.305
large	Dato8	753.429

Gráficas

Las gráficas con la representación de las pruebas realizadas.



Análisis

La complejidad temporal del requerimiento fue aceptable debido al uso del mergesort, el cual es necesario para poder sacar de menos reciente a más reciente, este se mejoraría haciendo el uso de minpq, por otro lado se hacen más iteraciones que en otros requerimientos debido a que en este toca graficar en la librería matplotlib que no acepta nuestra librería DISClib por lo que toca convertir el data structure a un tipo de dato que matplotlib entienda.

Requerimiento 8

Plantilla para documentar y analizar cada uno de los requerimientos.

Descripción

```python
def req_8(data_structs, fechaInicial, fechaFinal, clase):
    """
    Función que soluciona el requerimiento 8
    """
    # TODO: Realizar el requerimiento 8
    finalList = lt.newList("ARRAY_LIST")
    mapa = data_structs["Fecha"]
    rango = om.values(mapa, fechaInicial, fechaFinal)
    total_accidentes = 0
    for fechas in lt.iterator(rango):
        for i in lt.iterator(fechas["lstaccidentes"]):
            if str(i["CLASE_ACC"]) == clase:
                respuesta = i
                lt.addLast(finalList, respuesta)
                size = lt.size(finalList)
    total_accidentes += size
    tiles = 'openstreetmap'
    m = folium.Map(location=[4.64230635, -74.06471382], zoom_start=10, tiles=tiles)
    marker_cluster = MarkerCluster()
    for i in lt.iterator(finalList):
        if i["GRAVEDAD"] == "CON MUERTOS":
            color = "red"
            popup = 'CON MUERTOS'
        elif i["GRAVEDAD"] == "CON HERIDOS":
            color = "orange"
            popup = 'CON HERIDOS'
        elif i["GRAVEDAD"] == "SOLO DANOS":
            color = "lightgreen"
            popup = 'SOLO DAÑOS'
        marker = folium.Marker(location=[i["LATITUD"], i["LONGITUD"]], popup=popup, icon=folium.Icon(color=color))
        marker.add_to(marker_cluster)
        marker_cluster.add_to(m)
    m.save("mymap_output.html")
    return total_accidentes, m
```

Se sacan los valores del datastructs que corresponden con el rango ingresado por parámetro, posteriormente se ingresan los valores en una lista y finalmente se ordenan del más reciente al más antiguo

Entrada	El datastructs con los datos, una fecha inicial, una fecha final y una clase
Salidas	Los accidentes que ocurrieron dentro del rango ingresado ordenados de más reciente a más antiguo
Implementado (Sí/No)	Si se implementó por los tres integrantes

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	Complejidad
Paso 1: crear una lista final donde se almacenará la respuesta	O(1)

Paso 2: con .values() obtener los accidentes dentro del rango requerido	O(log(N))
Paso 3: se agregan los valores a la lista final	O(log(N))
Paso 4: Se itera la lista filtrada para poder hacer el mapa interactivo	O(M)
TOTAL	***O(Nlog(N))***

Siendo M una lista filtrada más pequeña que n por lo que M es mucho más pequeño que 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).

Procesadores	**12th Gen Intel(R) Core(TM) i7-12700 2.10 GHz**
Memoria RAM	32 GB
Sistema Operativo	Windows 10

Entrada	**Tiempo (ms)**
small	310.093
5 pct	3028.582
10 pct	6042.183
20 pct	11863.272
30 pct	17837.901
50 pct	30735.911
80 pct	49349.001
large	59401.407

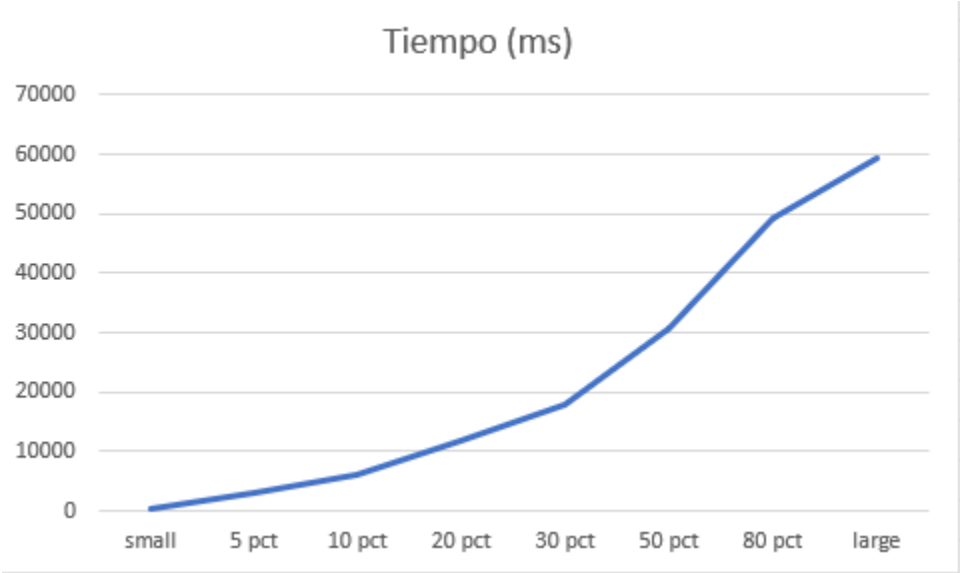
Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

Muestra	**Salida**	**Tiempo (ms)**
small	Dato1	310.093
5 pct	Dato2	3028.582
10 pct	Dato3	6042.183
20 pct	Dato4	11863.272
30 pct	Dato5	17837.901
50 pct	Dato6	30735.911
80 pct	Dato7	49349.001
large	Dato8	59401.407

Gráficas

Las gráficas con la representación de las pruebas realizadas.



Análisis

En este análisis se pueden observar tiempos bastante grandes comparados a los de otros requerimientos, pero esto es normal debido a que la librería folium gasta demasiado tiempo para poder hacer el mapa interactivo, de resto la complejidad temporal del algoritmo es bastante si no se tuviera en cuenta la creación del mapa interactivo.

Requerimiento Ejemplo

Descripción

```python
def get_data(data_structs, id):
    """
    Retorna un dato a partir de su ID
    """
    pos_data = lt.isPresent(data_structs["data"], id)
    if pos_data > 0:
        data = lt.getElement(data_structs["data"], pos_data)
        return data
    return None
```

Este requerimiento se encarga de retornar un dato de una lista dado su ID. Lo primero que hace es verificar si el elemento existe. Dado el caso que exista, retorna su posición, lo busca en la lista y lo retorna. De lo contrario, retorna None.

Entrada	Estructuras de datos del modelo, ID.
Salidas	El elemento con el ID dado, si no existe se retorna None
Implementado (Sí/No)	Si. Implementado por Juan Andrés Ariza

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	Complejidad
Buscar si el elemento existe (isPresent)	O(n)
Obtener el elemento (getElement)	O(1)
TOTAL	***O(n)***

Pruebas Realizadas

Las pruebas realizadas fueron realizadas en una máquina con las siguientes especificaciones. Los datos de entrada fueron el ID 1.

Procesadores	**AMD Ryzen 7 4800HS with Radeon Graphics**
Memoria RAM	8 GB
Sistema Operativo	Windows 10

Entrada	Tiempo (ms)
small	0.05
5 pct	0.33
10 pct	1.28
20 pct	2.54
30 pct	4.98
50 pct	7.51
80 pct	13.81
large	25.97

Tablas de datos

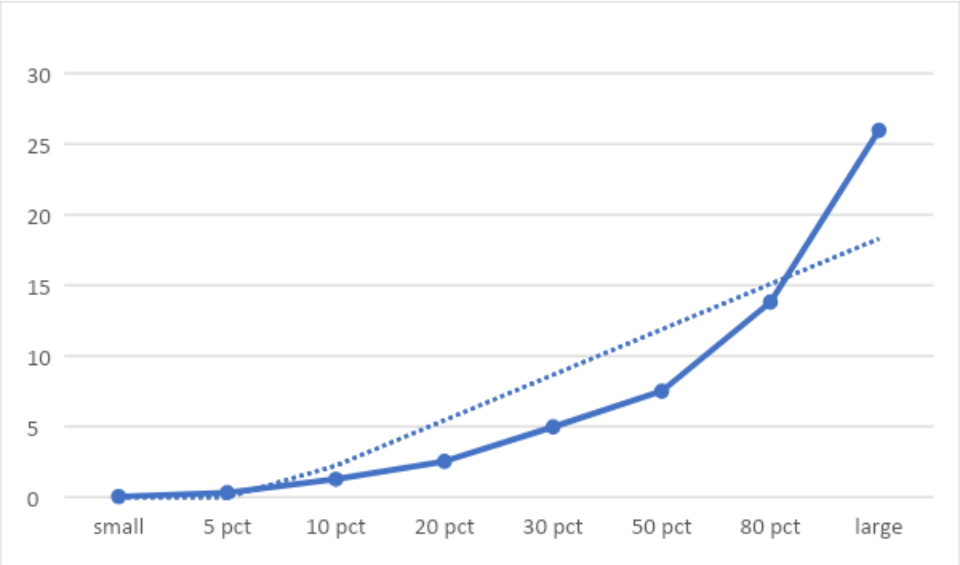
Las tablas con la recopilación de datos de las pruebas.

Muestra	Salida	Tiempo (ms)
small	Dato1	0.05
5 pct	Dato2	0.33
10 pct	Dato3	1.28
20 pct	Dato4	2.54
30 pct	Dato5	4.98
50 pct	Dato6	7.51
80 pct	Dato7	13.81

large	Dato8	25.97

Gráficas

Las gráficas con la representación de las pruebas realizadas.



Análisis

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

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