ANÁLISIS DEL RETO 3

Nicolás Arango, 202220342, n.arangor

Miguel Castillo, 201633992, ms.castillo

Sergio Cuevas, 202020957, s.cuevas

Carga de datos

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

Descripción

Breve descripción de como abordaron la implementación del requerimiento

Entrada	El tamaño del archivo
Salidas	La carga del porcentaje de los datos
Implementado (Sí/No)	Sí se implementó

Análisis de complejidad

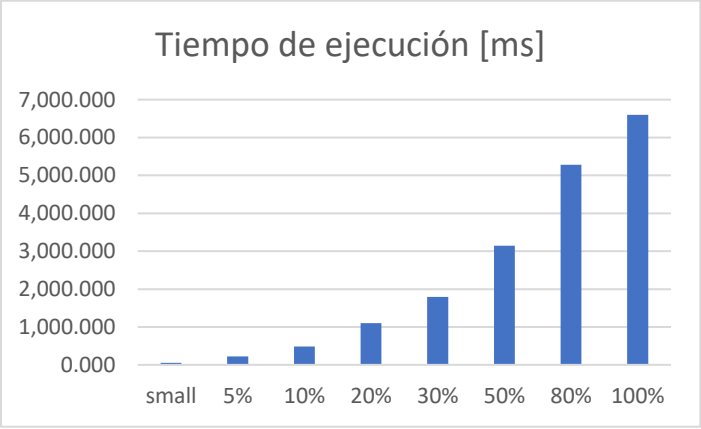
Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
TOTAL	***O(NlogN)***

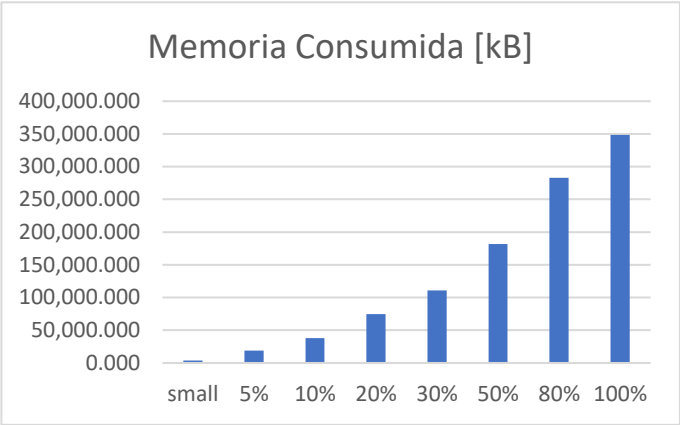
Pruebas Realizadas

Las pruebas se ejuctaron en un computador

Tamaño de Datos	Tiempo de ejecución [ms]
small	52,157
5%	225,718
10%	484,780
20%	1.099,216
30%	1.796,110
50%	3.141,143
80%	5.277,570
100%	6.600,172



Tamaño de Datos	Memoria Consumida [kB]
small	3.831,536
5%	19.002,086
10%	37.783,043
20%	74.710,297
30%	111.020,518
50%	181.585,033
80%	283.105,676
100%	348.504,109



Análisis

Para la carga de datos decidimos ser lo mas eficientes posibles consumiendo el menor espacio en memoria. Por esta razón solo cargamos ocho columnas en las cuales una de ellas (FECHA_HORA_ACC) nos permitía conseguir los datos que se daban en tres columnas diferentes. Al utilizar un árbol rojo-negro aseguramos la eficiencia al trabajar con gran cantidad de datos consiguiendo una complejidad de NlogN.

Requerimiento 1

Descripción

```python
def req_1(data_structs, fechaInicia, fechaFinal ):
    """
    Función que soluciona el requerimiento 1
    """
    # TODO: Realizar el requerimiento 1
    fechaInicia = fechaInicia +str(" 00:00:00+00")
    fechaFinal = fechaFinal + str(" 23:59:59+00")
    llavesImportantes = om.keys(data_structs ["fecha"], fechaFinal, fechaInicia)
    lista = lt.newList(datastructure="ARRAY_LIST")
    for elemnt in lt.iterator(llavesImportantes):
        datos = dict(me.getValue(om.get(data_structs ["fecha"], elemnt)))
        datos.pop("HORA_OCURRENCIA_ACC")
        lt.addLast(lista, datos)
    return lista["elements"]
```

En este requerimiento se usa un árbol rojo negro el cual esta ordenado por fecha y hora a partir de esto se saca una lista de llaves las cuales van desde la fecha inicial hasta la fecha final que entran por parámetros. Con estas llaves se hace un bucle para pasar todos los valores ha una lista tipo ARRAY y después sacar los elementos para imprimirlos

Entrada	Fecha Inicial, fecha Final
Salidas	Los accidente concurridos en un rango de fechas
Implementado (Sí/No)	Si. Implementado por Miguel Castillo

Análisis de complejidad

Pasos	**Complejidad**
Sacar las llaves entre las dos fechas	O(N)
Crear una nueva lista vacía	O(1)
Iterar por cada llave de la lista	O(N)
Acceder al valor a partir de todas las llaves de la lista	O(N)
Añadir el valor a una lista	O(N)
TOTAL	***O(N)***

Pruebas Realizadas

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

Procesadores	**Intel Core i7 11th Gen**
Memoria RAM	16 GB
Sistema Operativo	Windows 11 Pro

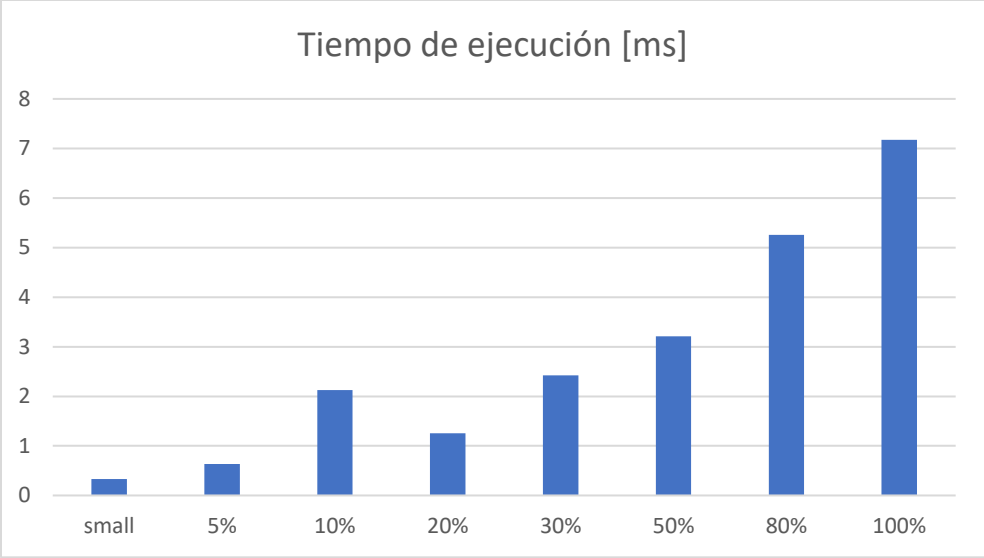
Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

Tamaño de Datos	Tiempo de ejecución [ms]
small	0,334
5%	0,635
10%	2,123
20%	1,253
30%	2,422
50%	3,209
80%	5,256
100%	7,172

Graficas

Las gráficas con la representación de las pruebas realizadas.



Análisis

A partir del árbol Rojo Negro que tiene como llaves la fecha y hora de los accidentes se saca una lisa de las llaves que están en el rango de horas que el usuario ingresa. Se usa una lista de tipo array para que a partir de estas se saquen los valores del árbol y se metan en esta lista, se usó una ARRY lista porque es más fácil acceder a los elementos para después imprimirlos. La complejidad es O(N) porque en el peor de los casos tiene que recorrer todos los elementos, aunque rara vez va a pasar.

Requerimiento 2

```python
def req_2(data_structs, mes, año, horaInicio, HoraFinal):
    """
    Función que soluciona el requerimiento 2
    """
    # TODO: Realizar el requerimiento 2
    rango1 = año + "/" + str(mes) + "/01" + " " + horaInicio + ":00+00"
    rango2 = año + "/" + str(mes) + "/31" + " " + HoraFinal + ":00+00"
    llavesImportantes = om.values(data_structs ["fecha"], rango2,  rango1)
    lista = lt.newList(datastructure="ARRAY_LIST")
    for elemnt in lt.iterator(llavesImportantes):
        elmetoHora =elemnt["HORA_OCURRENCIA_ACC"]
        if len(elemnt["HORA_OCURRENCIA_ACC"]) < 8:
            elmetoHora= "0"+elemnt["HORA_OCURRENCIA_ACC"]
        if (horaInicio) <= elmetoHora <= (HoraFinal):
            datos = dict(elemnt)
            datos.pop("HORA_OCURRENCIA_ACC")
            lt.addLast(lista, datos)
    return lista["elements"]
```

Descripción

En este requerimiento se usa un árbol rojo negro el cual esta ordenado por fecha y hora a partir de esto se saca una lista de valores los cuales van desde el primer día del mes hasta el último día para el mes y año que entran por parámetros. A partir de esta lista se hace un bucle para todos los elementos de la lista en el cual compara la hora de cada elemento con el rango de horas que entran por parámetros. Si el elementó está dentro del rango lo añade a una lista de tipo ARRAY y devuelve los elementos de esta lista.

Entrada	mes, año, hora Inicio, Hora Final
Salidas	Accidentes de cada día del mes para el rango de horas
Implementado (Sí/No)	Si se implementó por Nicolas

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Sacar las llaves del mes y año especifico	O(N)
Crear una nueva lista vacía	O(1)
Iterar por cada llave de la lista	O(N)
Comparar si el elemento está dentro de las horas limites	O(N)
Añadir los valores a una lista si esto es verdad	O(N)
TOTAL	O(N)

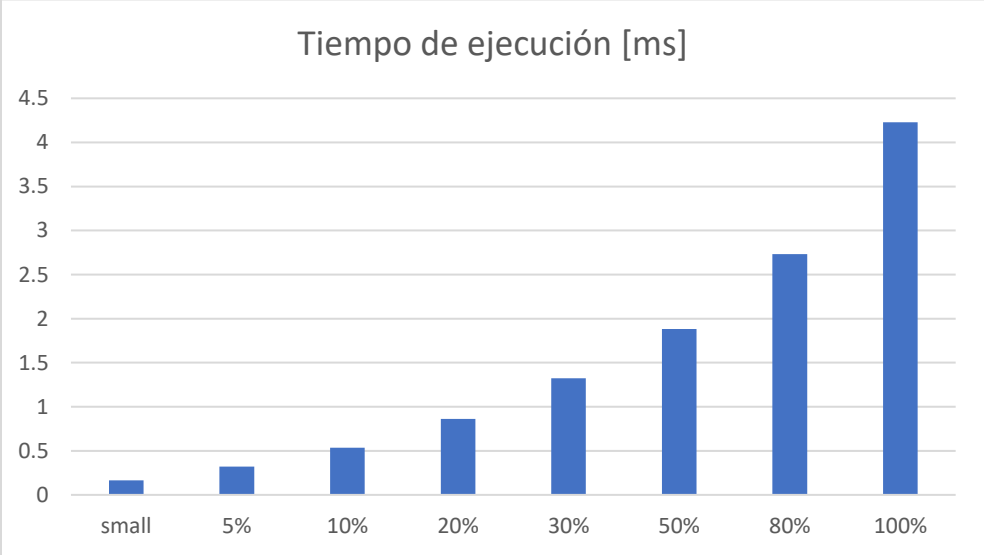
Pruebas Realizadas

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

Tamaño de Datos	Tiempo de ejecución [ms]
small	0,166
5%	0,321
10%	0,534
20%	0,863
30%	1,322
50%	1,882
80%	2,734
100%	4,229

Graficas

Las gráficas con la representación de las pruebas realizadas.



Análisis

A partir del árbol Rojo Negro que tiene como llaves la fecha y hora de los accidentes se saca una lista de los valores de todo el mes. Después compara si el accidente está en el rango de horas para cada día y sí si se añade a una lista de tipo array. Se metan en esta lista, se usó una ARRY lista porque es más fácil acceder a los elementos para después imprimirlos. La complejidad es O(N) porque en el peor de los casos tiene que recorrer todos los elementos, aunque rara vez va a pasar.

Requerimiento 3

```python
def req_3(data_structs, direccion, accidente):
    """
    Función que soluciona el requerimiento 3
    """
    # TODO: Realizar el requerimiento 3
    llaves = om.valueSet(data_structs['fecha'])
    lst = lt.newList(datastructure='ARRAY_LIST')
    for elmt in lt.iterator(llaves):
        if accidente.lower() == elmt['CLASE_ACC'].lower() and direccion.lower() in elmt['DIRECCION'].lower():
            data = dict(elmt)
            data.pop('CLASE_ACC')
            data.pop('HORA_OCURRENCIA_ACC')
            lt.addLast(lst, data)
    merg.sort(lst, cmpFecha)
    return lst
```

Descripción

Breve descripción de como abordaron la implementación del requerimiento

Entrada	La clase de accidente y la dirección donde se reportó
Salidas	Tabla con los accidentes de la clase y la dirección dada
Implementado (Sí/No)	Sí se implementó por Nicolás Arango

Análisis de complejidad

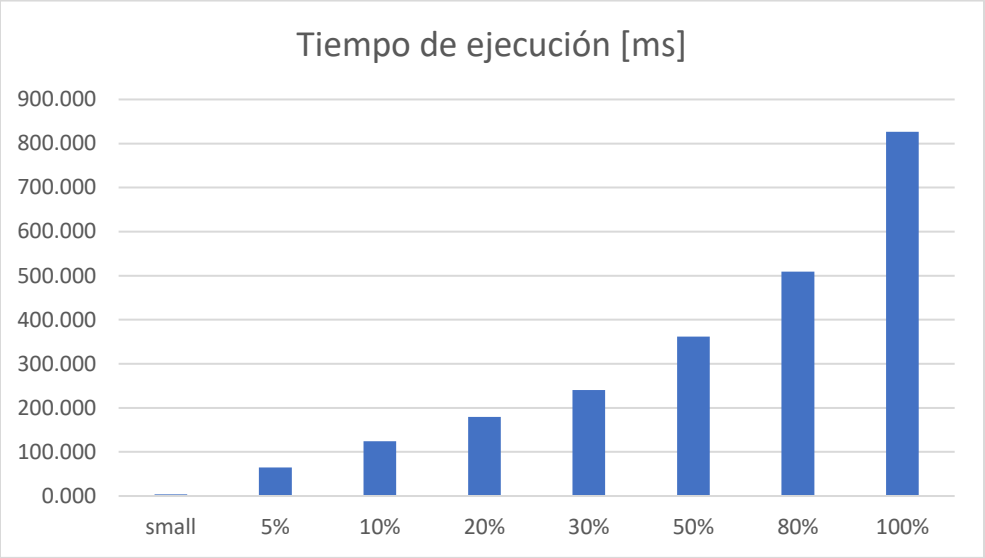
Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Sacar las llaves de todo el dict	O(N)
Crear una nueva lista vacía	O(1)
Iterar por cada llave de la lista	O(N)
Comparar si el elemento está dentro de la clase y dirección dada	O(N)
Añadir los valores a una lista si esto es verdad	O(N)
Organizar los accidentes por más reciente	O(NlogN)
TOTAL	O(Nlog(N))

Pruebas Realizadas

Tamaño de Datos	Tiempo de ejecución [ms]
small	3,723
5%	64,465
10%	124,184
20%	179,700
30%	240,533
50%	361,703
80%	509,150
100%	826,491

Graficas



Análisis

A partir del árbol Rojo Negro que tiene como llaves la dirección y clase de los accidentes, se saca una lista de los valores con las fechas. Después compara si la clase del accidente y la dirección está dentro de la dirección dada se añade a una lista de tipo arreglo (‘ARRAY_LIST’). Se usó un arreglo porque es menos complejo acceder a los elementos para después imprimirlos. La complejidad es O(N) porque en el peor de los casos tiene que recorrer todos los elementos, aunque rara vez va a pasar.

Requerimiento 4

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

```python
def req_4(data_structs, fechaInicia, fechaFinal, gravedad):
    """
    Función que soluciona el requerimiento 4
    """
    # TODO: Realizar el requerimiento 4

    fechaInicia = fechaInicia +str(" 00:00:00+00")
    fechaFinal = fechaFinal + str(" 23:59:59+00")
    llavesImportantes = om.values(data_structs ["fecha"],   fechaFinal, fechaInicia)
    ListaGraveda = lt.newList(datastructure="ARRAY_LIST")

    for item in lt.iterator(llavesImportantes ):
        if item["GRAVEDAD"].lower() == gravedad.lower():
            datos = dict(item)
            datos.pop("GRAVEDAD")
            lt.addLast(ListaGraveda, datos)

    return ListaGraveda["elements"]
```

Descripción

Saca una lista de valores la cual las llaves están entre el rango de fechas que entran por parámetros. A partir de esta lista compara la gravedad todos los accidentes con la gravedad que entra por parámetros y si son iguales los mete en una lista de tipo ARRAY

Entrada	Fecha Inicia, fecha Final, gravedad
Salidas	Los accidentes dentro rango de fechas dado y tienen la gravedad
Implementado (Sí/No)	Si, se implementó por Miguel Castillo

Análisis de complejidad

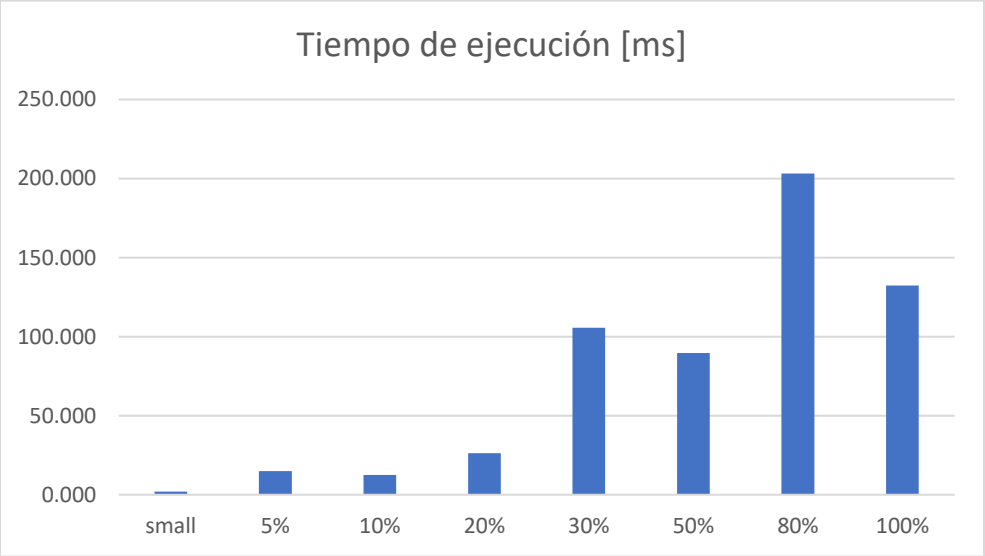
Pasos	**Complejidad**
Sacar las llaves del mes y año especifico	O(N)
Crear una nueva lista vacía	O(1)
Iterar por cada valor de la lista	O(N)
Comparar si el elemento tienen el mismo nivel de gravedad que el del parámetro	O(N)
Añadir los valor a una lista si esto es verdad	O(N)
TOTAL	O(N)

Pruebas Realizadas

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

Tamaño de Datos	Tiempo de ejecución [ms]
small	1,980
5%	14,946
10%	12,560
20%	26,200
30%	105,691
50%	89,719
80%	203,088
100%	132,318

Graficas



Análisis

A partir del árbol Rojo Negro que tiene como llaves la fecha y hora de los accidentes se saca una lista de los valores de las rengo de fecha. Después compara si la gravedad del accidente es igual a la del parámetro añade a una lista de tipo array. Se metan en esta lista, se usó una ARRY lista porque es más fácil acceder a los elementos para después imprimirlos. La complejidad es O(N) porque en el peor de los casos tiene que recorrer todos los elementos, aunque rara vez va a pasar.

Requerimiento <<5>>

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

Descripción

Este código tiene por objetivo reportar los 10 accidentes menos recientes ocurridos en un mes y año para una localidad de la ciudad. Inicialmente recibe todos los datos, y los comienza a filtrar por la fecha en conjunción de mes y año de ahí él restringe por la localidad, ya con esto él nos retorna los 10 menos recientes.

Entrada	Un diccionario con diccionarios en donde están todos los registros de los accidentes.
Salidas	Retorna un diccionario en donde el valor de las llaves son los datos solicitados.
Implementado (Sí/No)	Si se implementó. Implementado por Sergio Cuevas Clavijo

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
La función tiene un ciclo "for" que recorre todos los accidentes del mes y la localidad especificados, y detiene el ciclo después de encontrar los primeros 10 accidentes	O(n)
TOTAL	***O(n)***

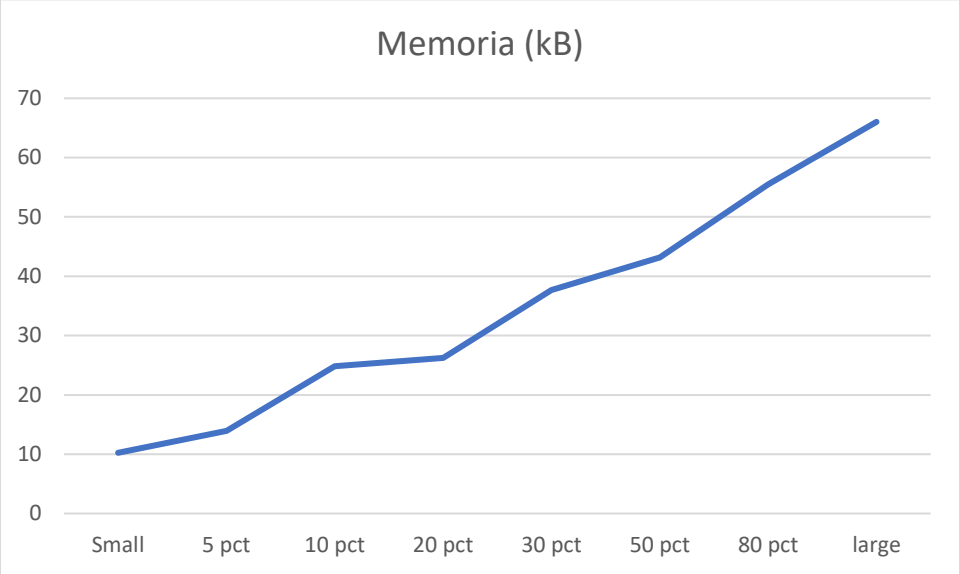
Pruebas Realizadas

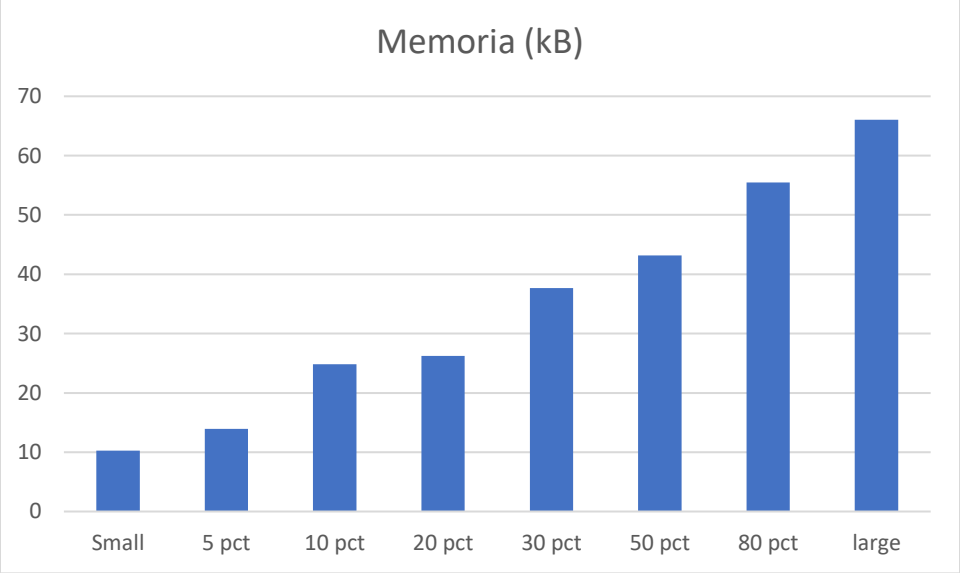
Las pruebas realizadas fueron realizadas en una maquina con las siguientes especificaciones. Los datos de entrada fueron el ID 1.

Entrada	**Tiempo (ms)**
Small	1.331
5 pct	3.553
10 pct	4.697
20 pct	6.082
30 pct	7.788
50 pct	10.846
80 pct	15.894
large	18.355

Tablas de datos

Graficas





Análisis

En este requerimiento se ve un comportamiento bastante “genérico” frente los datos y el tiempo. Se debe tomar en cuenta que todos los procesos dentro de este requerimiento son O(n) y la complejidad del ordenamiento para retornar el resultado es de O(n). En este caso la gráfica muestra fielmente un comportamiento O(n) cosa que en el mismo requerimiento pero del reto anterior no ocurría del todo.

Requerimiento 6

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

```python
def req_6(data_structs, año, mes, coordenadas, Radio, top):
    """
    Función que soluciona el requerimiento 6
    """
    # TODO: Realizar el requerimiento 6
    latitud1, longitud1=  coordenadas
    rango1 = año + "/" + str(mes) + "/01" + " 00:00:00+00"
    rango2 = año + "/" + str(mes) + "/31" + " 23:59:59+00"
    llavesImportantes = om.values(data_structs ["fecha"], rango2,  rango1)
    latitud1 = ma.radians(float(latitud1))
    longitud1 = ma.radians(float(longitud1))
    mapa = om.newMap(omaptype='RBT', comparefunction= ordenar)
    for elemnt in lt.iterator(llavesImportantes):
        latitud2 = ma.radians(float(elemnt["LATITUD"]))
        longitud2 = ma.radians(float(elemnt["LONGITUD"]))
        D = 2*(ma.asin(ma.sqrt((ma.sin((latitud2-latitud1)/2)**2)+ma.cos(latitud1)*ma.cos(latitu
        if D <= float(Radio):
            datos = dict(elemnt)
            datos.pop("HORA_OCURRENCIA_ACC")
            om.put(mapa, D,datos)

    llaves = om.valueSet(mapa)
    if top >= lt.size(llaves):
        lista = llaves
    else:
        lista = lt.subList(llaves, 1, int(top))
    listaImprimir= []
    for elemnto in lt.iterator(lista):
        listaImprimir.append(elemnto)
    return (listaImprimir)
```

Descripción

Apartir del árbol rojo negro que tiene como llaves las fecha y hora de cada accidente se saca una lista de elementos que van desde el primer hasta el último día del mes y año que entran por parámetros. A partir de esta lista se hace un ciclo en el cual se calcula que tan lejos fue el accidente del punto deseado y después se compara con el rango que entra por parámetros. Si el rango es mayor que la distancia del accidente este accidente entra a un RBT el cual la llave es la distancia desde el accidente hasta las coordenadas y el valor es el accidente. Después del ciclo se hace una lista con todos los valores del RBT y se saca una sublista de la cantidad de elementos que entra por parámetro.

Entrada	año, mes, coordenadas, Radio, top
Salidas	Una lista con los accidentes más cercanos
Implementado (Sí/No)	Si se implementó por Miguel Castillo

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

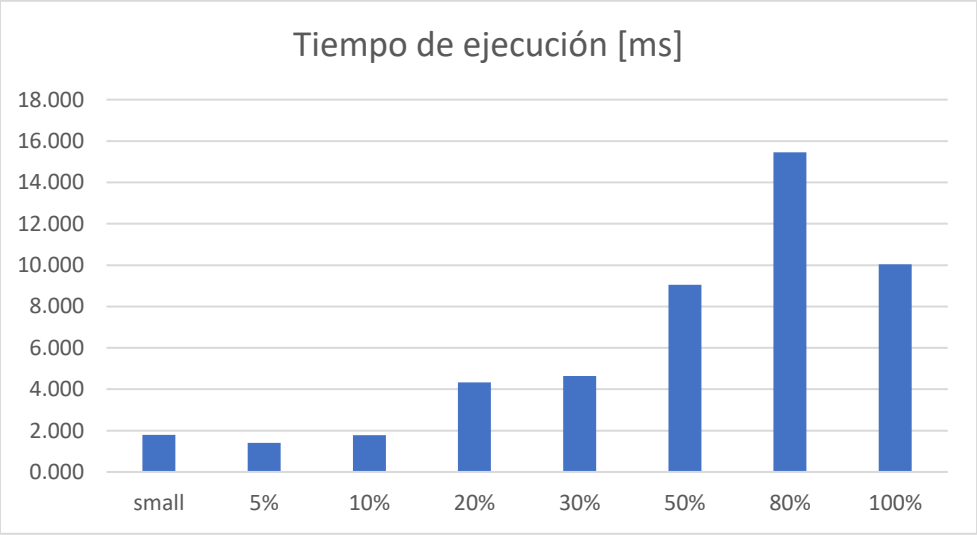
Pasos	**Complejidad**
Sacar las llaves del mes y año especifico	O(N)
Crear una nueva lista vacía	O(1)

Crear un árbol rojo negro vacío	O(1)
Iterar por cada valor de la lista	O(N)
Calcular la distancia que cada lugar tiene con respecto al origen	O(N)
Comparar si el elemento esta adentro del radio o no	O(N)
Añadir los valor a una mapa si lo anterior es verdad	O(Nlog(N))
Sacar los valores del árbol en una lista	O(N)
Hacer una sublista	O(N)
Crear una lista nativa para imprimir los datos	O(N)
TOTAL	O(Nlog(N))

Pruebas Realizadas

Tamaño de Datos	Tiempo de ejecución [ms]
small	1,794
5%	1,405
10%	1,770
20%	4,337
30%	4,632
50%	9,054
80%	15,448
100%	10,045

Graficas



Análisis

A partir del árbol Rojo Negro que tiene como llaves la fecha y hora de los accidentes se saca una lista de los valores del mes. Se crea un RBT vacío, después se itera sobre la lista y se añaden los elementos al árbol que estén dentro del rango (complejidad O(N(log(N)). Se usa un RBT porque se debe tener los elementos ordenados de más cercano a más lejano y es la manera más eficaz para hacerlo. Se sacan los valores del

RBT a una lista y a esta lista se le saca una sublista O(N) estas listas son SINGLI-LINKED porque agregar elementos es más rápido. Finalmente se pasa a una lista nativa porque tabúlate solo lee listas nativas

Requerimiento 7

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

```
def req_7(data_structs, año, mes ):
    """
    Función que soluciona el requerimiento 7
    """    # TODO: Realizar el requerimiento 7
    rango1 = año + "/" + str(mes) + "/01" + " 00:00:00+00"
    rango2 = año + "/" + str(mes) + "/31" + " 23:59:59+00"
    llavesImportantes = om.values(data_structs ["fecha"], rango2,  rango1)
    #dia del mes
    tbaladiames = om.newMap(omaptype = "RBT", comparefunction=ordenar)
    for item in lt.iterator(llavesImportantes):
        datos = dict(item)
        datos.pop("HORA_OCURRENCIA_ACC")
        dia = datos["FECHA_HORA_ACC"][8:10]
        llave = om.contains(tbaladiames, dia)
        if llave:
            elementodiames = me.getValue(om.get(tbaladiames, dia) )
            lt.addLast(elementodiames, item)
        else:
            om.put(tbaladiames,dia, lt.newList(datastructure="ARRAY_LIST"))
            elementodiames = me.getValue(om.get(tbaladiames, dia))
            lt.addLast(elementodiames, item)
    tabalahora = om.newMap(omaptype = "RBT", comparefunction=ordenar)
    for item in lt.iterator(llavesImportantes):
        hora = int(str(item["HORA_OCURRENCIA_ACC"][0:2]).replace(":", ""))
        llave = om.contains(tabalahora, hora)
        if llave:
            elementohora = me.getValue(om.get(tabalahora, hora))+1
            om.put(tabalahora,hora, elementohora )
        else:
            om.put(tabalahora,hora, 1)
    return  tbaladiames, tabalahora
```

Saca una lista de valores la cual las llaves son del mes y año especifico que entra por parámetro. A partir de esta lista Separa todos los elementos por el día del mes en el cual se causo el accidente y los mete en un RBT el cual el cual la llave es el día en que ocurrieron. Además también cuenta la cantidad de veces que ocurren los accidentes esta cuenta lo hace a través de un lup y lo gurda en un RBT el cual la llave es la hora y el valor es la cantidad de accidentes que pasaron en es hora durante el mes.

Descripción

Breve descripción de como abordaron la implementación del requerimiento

Entrada	Año y mes
Salidas	Una gráfica con los crímenes del mes dividido en horas y el crimen mas tarde y temprano de cada día del mes
Implementado (Sí/No)	Si se implementó

Análisis de complejidad

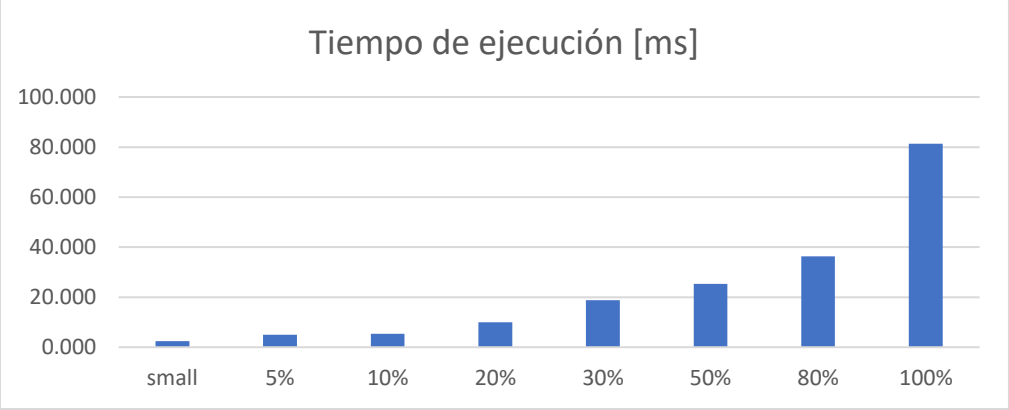
Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Sacar las llaves del mes y año especifico	O(N)
Crear un árbol rojo negro vacío	O(1)
Iterar por cada valor de la lista	O(N)
Añadir los elementos al árbol rojo negro dependiendo de su día en el mes	O(Nlog(n))
Crear un árbol rojo negro vacío	O(1)
Iterar por cada valor de la lista	O(N)
Añadir los elementos al árbol rojo negro dependiendo de su hora en el día	O(Nlog(n))
TOTAL	O(Nlog(N))

Pruebas Realizadas

Tamaño de Datos	Tiempo de ejecución [ms]
small	2,425
5%	4,951
10%	5,334
20%	10,002
30%	18,797
50%	25,362
80%	36,277
100%	81,344

Graficas



Análisis

A pesar de que se usan dos arboles RBT la complejidad total es de O(N) porque se tiene que iterar por todos los elementos para poder saber en qué llave meter los diferentes accidentes. Se usan los RBT porque se debe de mantenerse ordenado las horas y los días para poder mostrar correctamente las cosas. Finalmente se usan Array lista para poder guardar los crímenes que pasan el mismo día. Se usan array listo ya que es más rápido llegar a los diferentes elementos de la lista.

Requerimiento 8

```python
def req_8(data_structs, fechaInicia, fechaFinal, clase):
    """
    Función que soluciona el requerimiento 8
    """
    # TODO: Realizar el requerimiento 8
    fechaInicia = fechaInicia +str(" 00:00:00+00")
    fechaFinal = fechaFinal + str(" 23:59:59+00")
    llavesImportantes = om.keys(data_structs["fecha"], fechaFinal, fechaInicia)
    lista = lt.newList(datastructure='ARRAY_LIST')
    mapa = folium.Map(location=[4.658275, -74.051290])
    x = 0

    for elemnt in lt.iterator(llavesImportantes):
        datos = dict(me.getValue(om.get(data_structs ["fecha"], elemnt)))
        datos.pop("HORA_OCURRENCIA_ACC")
        lt.addLast(lista, datos)
    for accidente in lista['elements']:
        if accidente['CLASE_ACC'] == clase:
            x += 1
            latitud = accidente['LATITUD']
            longitud = accidente['LONGITUD']
            color = ''
            tooltip = ''
            if accidente['GRAVEDAD'] == 'SOLO DANOS':
                color = 'orange'
                tooltip = 'Solo Daños'
            elif accidente['GRAVEDAD'] == 'CON HERIDOS':
                color = 'blue'
                tooltip = 'Con heridos'
            elif accidente['GRAVEDAD'] == 'CON MUERTOS':
                color = 'red'
                tooltip = 'Con Muerto'
            folium.Marker([float(latitud), float(longitud)], icon=folium.Icon(color=color),tooltip=tooltip).add_to(mapa)
    folium.Map.show_in_browser(mapa)
    return(x, mapa)
```

Descripción

Breve descripción de como abordaron la implementación del requerimiento

Entrada	Fecha inicial, fecha final y clase de accidente
Salidas	Un mapa con los accidentes del rango de fechas de la clase dada
Implementado (Sí/No)	Si se implementó

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**

Sacar las llaves del rango de fechas	O(N)
Crear una lista vacía	O(1)
Iterar por cada valor de la lista	O(N)
Añadir los elementos al arreglo si conciden con los parámetros dados	O(Nlog(n))
Imprimir un mapa con los accidentes que coincidan	O(1)
TOTAL	O(Nlog(N))

Análisis

Dado que se hacen procesos externos, no es posible medir los tiempos o memoria consumida por el requerimiento. Lo que pretende este requerimiento es retornar un mapa el cual indique, a través de colores, la gravedad de los accidentes que coinciden con el rango de fechas y la clase dada.