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

# ANÁLISIS DEL RETO 3

*Pedro Pablo Sanín, 202221527, p.sanin@uniandes.edu.co*
*Juan Esteban Rojas Olave, 202211481, je.rojaso1@uniandes.edu.co*
*Laura Andrea Hurtado, 202213259, la.hurtadoa1@uniandes.edu.co*

# Carga de datos

```
def load_data(control, filename, memflag):
    """
    Carga los datos del reto
    """
    # TODO: Realizar la carga de datos
    if memflag:
        tracemalloc.start()
        start_memory = get_memory()

    start_time = get_time()
    input_file = csv.DictReader(open(filename, encoding='utf-8'))
    idx = 0

    for accidente in input_file:
        idx += 1
        data = model.new_data(idx, accidente)
        model.add_data(control['model'], data)

    end_time = get_time()
    delta_time = float(end_time-start_time)
    acc_todos = control['model']['acc_fecha']
    lista_todos = control['model']['lista_todos']

    if memflag:
        end_memory = get_memory()
        tracemalloc.stop()
        delta_mem = delta_memory(end_memory, start_memory)
        return delta_time, model.tree_size(acc_todos), model.first_last_n_elems_list(lista_todos,3), delta_mem

    return delta_time, model.tree_size(acc_todos), model.first_last_n_elems_list(lista_todos,3)
```

# Descripción

Este proceso dividido entre el controlador y el modelo lee un CSV y guarda sus datos en distintas estructuras

| | |
|---|---|
| **Entrada** | Controlador con estructuras iniciales, nombre del archivo CSV, marcador de memoria |
| **Salidas** | El controlador con las distintas estructuras de datos lleno. La función en el controlador retorna también los primeros y últimos tres elementos la lista con todos los accidentes y el tamaño del árbol con todos los accidentes ordenados por parámetro combinado fecha_hora_acc. |
| **Implementado (Sí/No)** | Si |

## Estructura del controlador

Todas las estructuras de datos están guardadas en un diccionario de Python, las parejas llave valor de este diccionario son:

- 'acc_fecha': Un árbol RBT con función de comparación compare_date_time_tuples que guarda y organiza todas las entradas del CSV de acuerdo a la fecha y hora del accidente. Las llaves son una fecha/hora y los valores son arrays con todos los accidentes ocurridos en esa fecha/hora.
- 'acc_clase': Una tabla de hash de tamaño 8 con mecanismo de colisión linear probing y factor de carga 0.5. Las llaves de esta tabla serán las distintas clases de accidente, y los valores serán una lista con todos los accidentes de dicha clase.
- 'acc_gravedad': Una tabla de hash de tamaño 8 con mecanismo de colisión linear probing y factor de carga 0.5. Las llaves de esta tabla serán las distintas gravedadeas de accidente, y los valores serán una lista con todos los accidentes de dicha gravedad.
- 'acc_localidad': Una tabla de hash de tamaño 60 con mecanismo de colisión linear probing y factor de carga 0.5. Las llaves de esta tabla serán las distintas localidades de Bogotá, y los valores serán una lista con todos los accidentes en dicha localidad.
- 'acc_anio_mes': Una tabla de hash de tamaño 100 con mecanismo de colisión linear probing y factor de carga 0.5. Las llaves de esta tabla serán tuplas (año, mes), y los valores serán una lista con todos los accidentes en ocurridos en dicho año y mes.
- 'lista_todos': Un array que contendrá todos los elementos cargados. (Implementada principalmente para el view de la carga de datos)

## Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

| **Pasos** | **Complejidad** |
|---|---|
| Paso 1: Abrir el documento CSV a leer e inicializar un contador para los id | O(1) |
| Paso 2 : Recorrer la lista, crear el dato a insertar utilizando la función new_data() y luego, insertarlo a la estructura de datos utilizando la función add_data() | O(Nlog(N)) |
| Paso 3: Retornar el tamaño del árbol 'acc_fecha' y los primeros y últimos tres elementos de la lista 'lista_todos' | O(1) |
| ***TOTAL*** | ***O(Nlog(N))*** |

### Complejidad de la función new_data(id, info)

```
def new_data(id, info):
    """
    Crea una nueva estructura para modelar los datos
    """
    #TODO: Crear la función para estructurar los datos
```

```
    data = {'idx': id}
    for column in info.keys():
        if info[column].isnumeric():
            data[column] = int(info[column])
        else:
            data[column] = str(info[column])
    fecha = info['FECHA_OCURRENCIA_ACC'].split('/')
    hora = info['HORA_OCURRENCIA_ACC'].split(':')
    data['FECHA_HORA_ACC'] = datetime.datetime(year=int(fecha[0]), month=int(fecha[1]), day=int(fecha[2]), hour=
int(hora[0]), minute=int(hora[1]), second = int(hora[2]))
    lista_calles = data['DIRECCION'].split('-')
    if len(lista_calles) >= 2:
        data['CALLE_ACC'] = lista_calles[0].strip()
        data['CARRERA_ACC'] = lista_calles[1].strip()
    else:
        data['CALLE_ACC'] = lista_calles[0].strip()
        data['CARRERA_ACC'] = '00'
    data['LATITUD'] = float(data['LATITUD'])
    data['LONGITUD'] = float(data['LONGITUD'])
    return data
```

| Pasos | Complejidad |
|---|---|
| Paso 1: Crea una nueva columna con el id pasado por parámetro. | O(1) |
| Paso 2: Recorre las llaves de info, si son numéricas, los convierte en int, si no, quedan string. | O(1) |
| Paso 3: A partir de las columnas 'FECHA_OCURRENCIA_ACC' y 'HORA_OCURRENCIA_ACC', crea una nueva columna de tipo datetime, 'FECHA_HORA_ACC' | O(1) |
| Paso 4: Parte la dirección utilizando split y crea dos nuevas columnas, 'CALLE_ACC', 'CARRERA_ACC' | O(1) |
| Paso 5: Formatea la latitud y longitud como floats | O(1) |

## Complejidad función add_data(data_structs, data)

```
def add_data(data_structs, data):
    """
    Función para agregar nuevos elementos a la lista
    """
    #TODO: Crear la función para agregar elementos a una lista
    data_date_time = data['FECHA_HORA_ACC']
    data_gravedad = data['GRAVEDAD']
    data_clase = data['CLASE_ACC']
    data_localidad = data['LOCALIDAD']
    data_anio = data['ANO_OCURRENCIA_ACC']
    data_mes = data['MES_OCURRENCIA_ACC']

    datastructs_anio_mes = data_structs['acc_anio_mes']
```

```
datastructs_gravedad = data_structs['acc_gravedad']
datastructs_clase = data_structs['acc_clase']
datastructs_localidad = data_structs['acc_localidad']
datastructs_lista = data_structs['lista_todos']

tupla_anio_mes = (data_anio, data_mes)

lt.addLast(datastructs_lista, data)

datastructs_fecha = data_structs['acc_fecha']
data_structs['acc_fecha'] = insert_to_tree(datastructs_fecha, data_date_time, data)


agregar_a_filtro(datastructs_gravedad, data_gravedad, data)
agregar_a_filtro(datastructs_clase, data_clase, data)
agregar_a_filtro(datastructs_localidad, data_localidad, data)
agregar_a_filtro(datastructs_anio_mes, tupla_anio_mes, data)
```

| **Pasos** | **Complejidad** |
|---|---|
| Paso 1: Para el dato, recoge los datos que serán las llaves de las estructuras, es decir, almacena la fecha y hora, la clase, la gravedad, la localidad, y además crea una tupla año mes. | O(1) |
| Paso 2: Utilizando la función insert_to_tree(), revisa si la fecha/hora, está en las llaves, si lo está, lo agrega al array correspondiente, y si no, crea e inserta la pareja llave valor con la llave siendo la fecha/hora, y el valor una lista que contiene el elemento. | O(log(N)) |
| Paso 3: Utilizando la función agregar_a_filtro(), agregamos cada elemento a la tabla de hash de clase, localidad, gravedad, año/mes. La función agregar_a_filtro() es equivalente a la función insert_to_tree(), esta revisa si la clase ya está, si no lo está, crea la lista con el elemento y lo inserta, y si lo está, simplemente agrega el elemento a la lista existente. | O(1) |
| Paso 4: Se agrega el dato al final de la lista con todos los elementos. | O(1) |
| **Total** | **O(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).

| **Especificaciones** | 1,4 GHz Intel Core i5 de cuatro núcleos, 8 GB 2133 MHz LPDDR3, macOS 13.3.1 |
|---|---|

| **Entrada** | **Tiempo (ms)** |
|---|---|
| 1% (2352 accidentes) | 158.62 |
| 5% (11692 accidentes) | 785.22 |

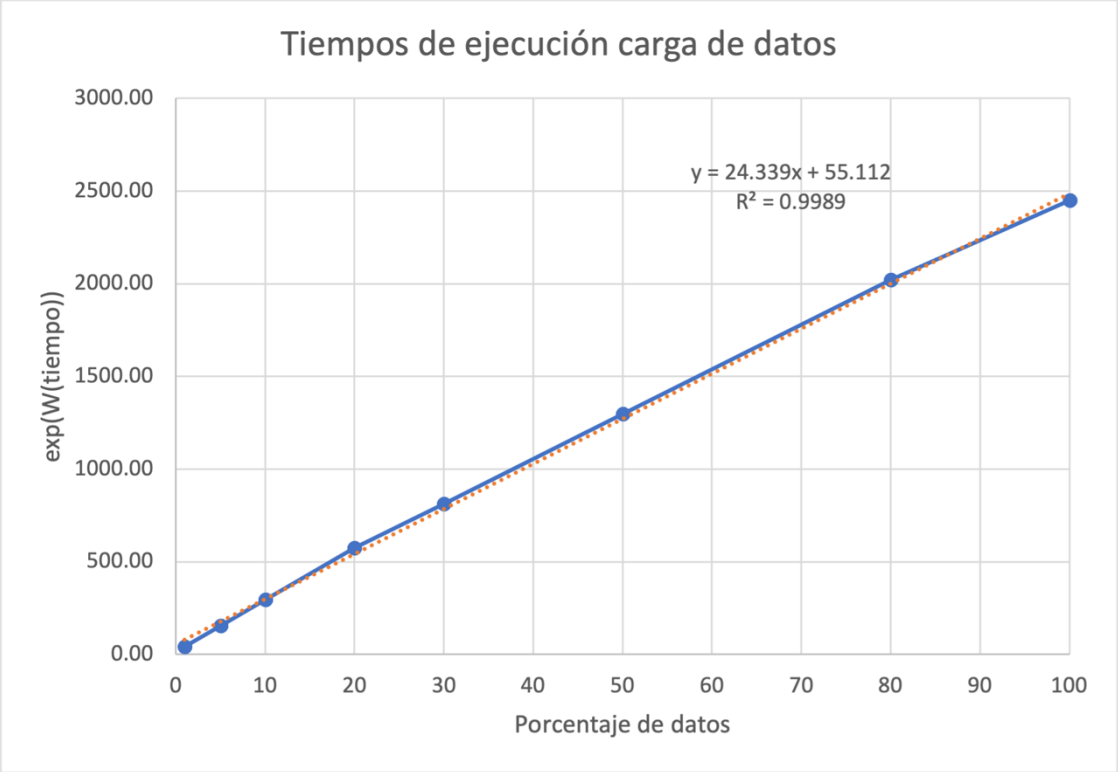
| | |
|---|---|
| 10% (23124 accidentes) | 1679.51 |
| 20% (45230 accidentes) | 3644.66 |
| 30% (66532 accidentes) | 5437.07 |
| 50% (106724 accidentes) | 9291.29 |
| 80% (161640 accidentes) | 15383.23 |
| 100% (195414 accidentes) | 19113.09 |

## Gráficas

Como sospechamos que la complejidad de nuestro algoritmo es Nlog(N), y queremos hacer una regresión de los datos, aplicaremos la función exp(W(tiempo)) (donde W es la función W de Lambert, inversa a N*exp(N)) al conjunto de datos. Se puede verificar que esta función es la función inversa a Nln(N) y por ende, si se cumple con el análisis teórico, la regresión será lineal.

## Análisis

Como era de esperar, al ver la regresión lineal, vemos que la carga de datos de nuestro requerimiento tiene una complejidad de O(Nlog(N)). Esto corresponde con el análisis de complejidad temporal que realizamos. Se sabe que la complejidad viene principalmente de la inserción de elementos al árbol RBT, como se insertan N elementos y la inserción de cada uno tiene complejidad Nlog(N), la complejidad general de la carga es simplemente O(Nlog(N)).



# Requerimiento 1

```
def req_1(data_structs, fecha_final, fecha_inicial):
    """
    Función que soluciona el requerimiento 1
    """
    # TODO: Realizar el requerimiento 1
    data_structs_fecha = data_structs['acc_fecha']
    accidentes_rango = om.values(data_structs_fecha, fecha_inicial, fecha_final)
    lista_result = elems_rango_a_lista(accidentes_rango)
    size_accidentes = lt.size(lista_result)
    return size_accidentes, lista_result
```

## Descripción

Este requerimiento recibe la estructura de datos completa,y un rango de fechas, y retorna todas los accidentes ocurridos en ese rango de fechas, ordenados del más reciente al menos reciente y el número de accidentes ocurridos en dicho intervalo.

| | |
|---|---|
| **Entrada** | data_structs, fecha_inicial, fecha_final |
| **Salidas** | size_accidentes, lista_result |
| **Implementado (Sí/No)** | Si |

## Análisis de complejidad

| **Pasos** | **Complejidad** |
|---|---|
| Paso 1: Guarda el árbol con todos los accidentes data_structs[‘acc_todos’] en la variable data_structs_fecha. | O(1) |
| Paso 2: Utilizando la función om.values(data_structs_fecha, fecha_incial, fecha_final), sacamos la lista de todos los elementos en el intervalo entre fecha_inicial y fecha_final. Guardamos esta lista en la variable accidentes_rango. | O(log(N) + k), donde k es el tamaño del intervalo de fechas. |
| Paso 3: Como la accidentes_rango es una lista de lista, utilizamos la función elems_rango_a_lista(accidentes_rango) para eliminar todas las listas intermedias. Esta función recorre todas las llaves en accidentes_rango. Guardamos esta nueva lista en la variable lista_result. | O(N) |
| Paso 4: Guardamos el tamaño de lista_result en una variable, size_accidentes. | O(1) |
| **Total:** | **O(N)** |

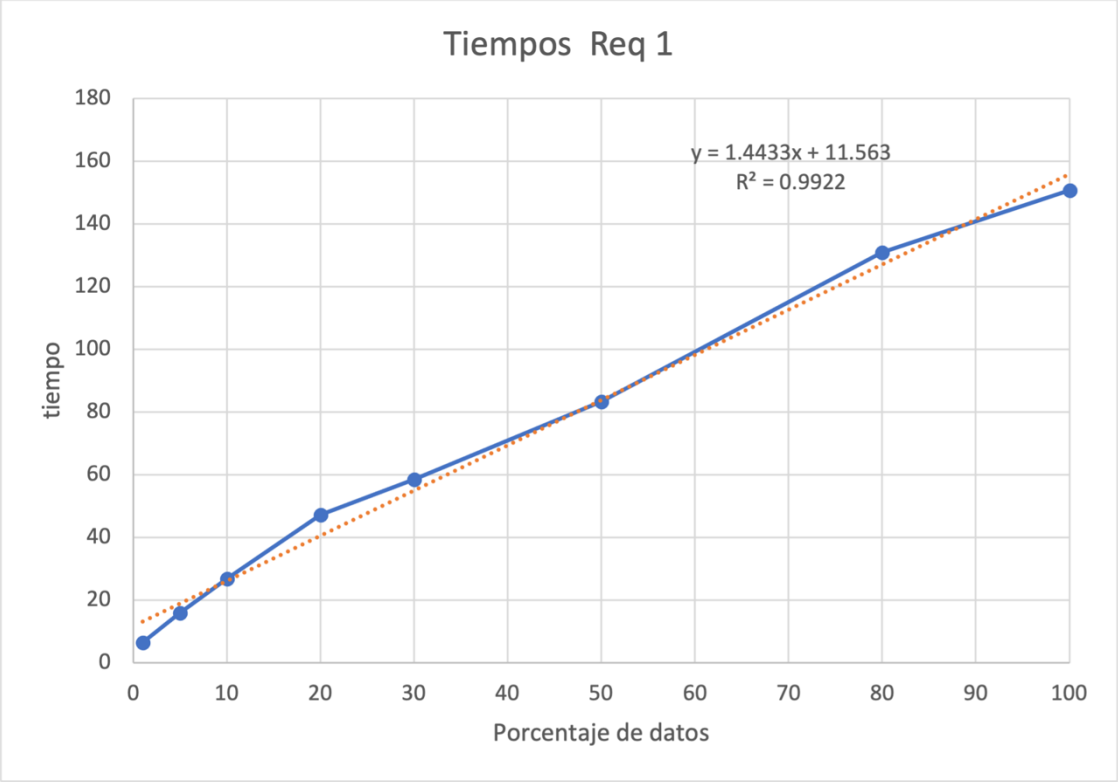
## Pruebas realizadas

| | |
|---|---|
| **Especificaciones** | 1,4 GHz Intel Core i5 de cuatro núcleos, 8 GB 2133 MHz LPDDR3, macOS 13.3.1 |
| **Fecha inicial** | 01/11/2016 |

| **Fecha final** | 20/06/2018 |
|---|---|

| **Entrada** | **Tiempo (ms)** |
|---|---|
| 1% (2352 accidentes) | 6.43 |
| 5% (11692 accidentes) | 15.91 |
| 10% (23124 accidentes) | 26.78 |
| 20% (45230 accidentes) | 47.23 |
| 30% (66532 accidentes) | 58.43 |
| 50% (106724 accidentes) | 83.35 |
| 80% (161640 accidentes) | 130.92 |
| 100% (195414 accidentes) | 150.66 |

## Gráficas




## Análisis de resultados

En este caso vemos que una vez más, el tiempo de ejecución experimental corresponde con los datos experimentales. Desafortunadamente, al existir la posibilidad de tener dos accidentes en el mismo momento, tuvimos que recorrer la lista con todos los accidentes para poder presentar el resultado. No obstante, la complejidad sin tener que modificar la lista es O(log(N)). Esto podría solucionarse al modificar la lista en el view y después calcular el tiempo. No obstante, esto no respetaría las instrucciones y el patrón MVC. Si bien el orden de crecimiento es lineal,

vemos que los tiempos de ejecución son muy reducidos, y por ende consideramos que la implementación del requerimiento fue exitosa.

# Requerimiento 2

```
def req_2(data_structs, mes, anio, hora_minutos_iniciales, hora_minutos_finales):
    """
    Función que soluciona el requerimiento 2
    """
    # TODO: Realizar el requerimiento 2
    tupla_anio_mes = (anio,mes)
    datastructs_anio_mes = data_structs['acc_anio_mes']
    data_anio_mes_k_v = mp.get(datastructs_anio_mes, tupla_anio_mes)
    data_anio_mes = me.getValue(data_anio_mes_k_v)

    tree_anio_mes = om.newMap(omaptype='RBT')

    for elem in lt.iterator(data_anio_mes):
        tree_anio_mes_aux = tree_anio_mes
        data_hora_str = elem['HORA_OCURRENCIA_ACC']
        data_hora_list = data_hora_str.split(':')
        data_hora = datetime.time(int(data_hora_list[0]), int(data_hora_list[1]), int(data_hora_list[2]))
        tree_anio_mes = insert_to_tree(tree_anio_mes_aux, data_hora, elem)

    data_entre_horas_aux = om.values(tree_anio_mes, hora_minutos_iniciales, hora_minutos_finales)
    data_entre_horas = elems_rango_a_lista(data_entre_horas_aux)


    size_accidentes = lt.size(data_entre_horas)

    return size_accidentes, data_entre_horas
```

## Descripción

En este requerimiento se buscan conocer todos los accidentes ocurridos en un intervalo de horas del día para un mes y año dados.

| | |
|---|---|
| **Entrada** | data_structs, mes, anio, hora_minutos_iniciales, hora_minutos_finales |
| **Salidas** | size_accidentes, data_entre_horas |
| **Implementado (Sí/No)** | Si |

## Análisis de complejidad

| **Pasos** | **Complejidad** |
|---|---|
| Paso 1: Se construye la tupla año,mes y se busca su valor en tabla de hash data_structs[‘acc_anio_mes’], este valor, | O(1) |

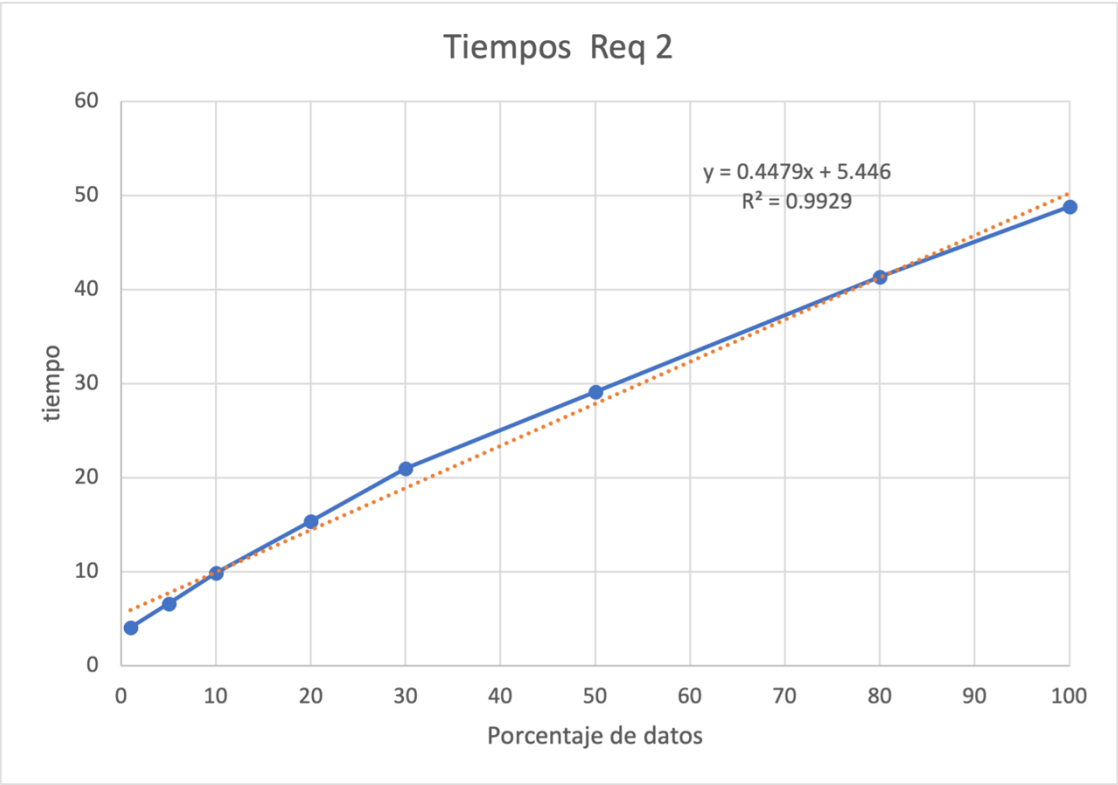
| | |
|---|---|
| data_anio_mes, es una lista con todos los accidentes ocurridos en ese año/mes. | |
| Paso 2: Se crea un árbol RBT vacío, tree_anio_mes. | O(1) |
| Paso 3: Se recorre data_anio_mes, y se va llenando tree_anio_mes utilizando la función insert_to_tree. Las llaves de tree_anio_mes, son las horas del día y los valores, una vez más, son arrays con todos los accidentes ocurridos en esa hora. | O(N) (Si bien añadir un elemento a un arbol tiene complejidad O(log(N)), las horas del día no aumentan con los datos. |
| Paso 3: Utilizando la función om.values(tree_anio_mes, hora_minutos_iniciales, hora_minutos_finales), sacamos todos los valores entre las dos horas. Una vez más, como esto es una lista de listas, debe simplificarse utilizando la función elems_rango_a_lista(). Se incializa la variable data_entre_horas para la lista final. | O(N) |
| Paso 4: Se calcula el tamaño de esta lista y se retorna el tamaño size_accidentes, y data_entre_horas. | O(1) |
| **Total** | O(N) |

## Pruebas

| | |
|---|---|
| **Especificaciones** | 1,4 GHz Intel Core i5 de cuatro núcleos, 8 GB 2133 MHz LPDDR3, macOS 13.3.1 |
| **Mes** | Marzo |
| **Año** | 2019 |
| **Hora inicial** | 07:00 |
| **Hora final** | 18:00 |

| **Entrada** | **Tiempo (ms)** |
|---|---|
| 1% (2352 accidentes) | 4.07 |
| 5% (11692 accidentes) | 6.6 |
| 10% (23124 accidentes) | 9.88 |
| 20% (45230 accidentes) | 15.36 |
| 30% (66532 accidentes) | 20.98 |
| 50% (106724 accidentes) | 29.11 |
| 80% (161640 accidentes) | 41.35 |
| 100% (195414 accidentes) | 48.81 |

## Gráficas




## Análisis de resultados

Una vez más, vemos que los resultados experimentales corresponden con los resultados teóricos. Decidimos no crear el árbol tree_anio_mes desde el comienzo pues si lo hicieramos tendríamos que crear un árbol para cada combinación año mes y eso tomaría muchísima memoria y tiempo. En el paso 2, la complejidad temporal es únicamente O(N) pues el árbol tiene un número de llaves constante independientemente de las pruebas.

# Requerimiento 3

```
def req_3(data_structs, clase, via):
    """
    Función que soluciona el requerimiento 3
    """
    # TODO: Realizar el requerimiento 3
    datastructs_clase = data_structs['acc_clase']
    data_clase_k_v = mp.get(datastructs_clase, clase)
    data_clase = me.getValue(data_clase_k_v)

    rbt_via = om.newMap('RBT')


    for elem in lt.iterator(data_clase):
        calle = elem['CALLE_ACC']
        carrera =  elem['CARRERA_ACC']
        if via in [calle,carrera]:
            rbt_via_aux = rbt_via
            rbt_via = om.put(rbt_via_aux, elem['FECHA_HORA_ACC'], elem)


    result = lt.newList('ARRAY_LIST')

    if om.isEmpty(rbt_via):
        return result

    for i in range(3):
        rbt_max_key = om.maxKey(rbt_via)
        rbt_max = me.getValue(om.get(rbt_via, rbt_max_key))
        rbt_via_aux = rbt_via
        rbt_via = om.deleteMax(rbt_via_aux)
        lt.addLast(result, rbt_max)
    return lt.size(data_clase), result
```

## Descripción

En este requerimiento se busca reportar los tres accidentes mas recientes de cierta clase ocurridos en una via.

| | |
|---|---|
| **Entrada** | data_structs, clase, via |
| **Salidas** | size(data_clase), result |
| **Implementado (Sí/No)** | Si |

## Análisis de complejidad

| Pasos | Complejidad |
|---|---|
| Paso 1: Utilizando la estructura de datos data_structs[‘acc_clase’], se sacan todos los accidentes de cierta clase y se guardan en la variable data_clase. | O(1) |
| Paso 2: Se crea un rbt para guardar los datos, rbt_via. | O(1) |
| Paso 3: Se recorre la lista data_clase. Para cada iteración se revisa si el elemento ocurrió en la vía especificada y si sí, se inserta en rbt_via con llave su fecha y hora. | O(Nlog(N)) |
| Paso 4: Se revisa si el rbt_via está vacío, si lo está se retorna una lista vacía. | O(1) |
| Paso 5: Se crea un array list para retornar el resultado. | O(1) |
| Paso 5: Se saca la máxima llave de rbt_via, se busca esa llave en rbt_via, y se añade su valor a result. Luego, se elimina el elemento con la máxima llave. Se repite esto tres veces para sacar los tres accidentes más recientes. | O(log(N)) |
| Paso 6: Se retorna result y el tamaño de data_clase. | O(1) |
| **Total** | O(Nlog(N)) |

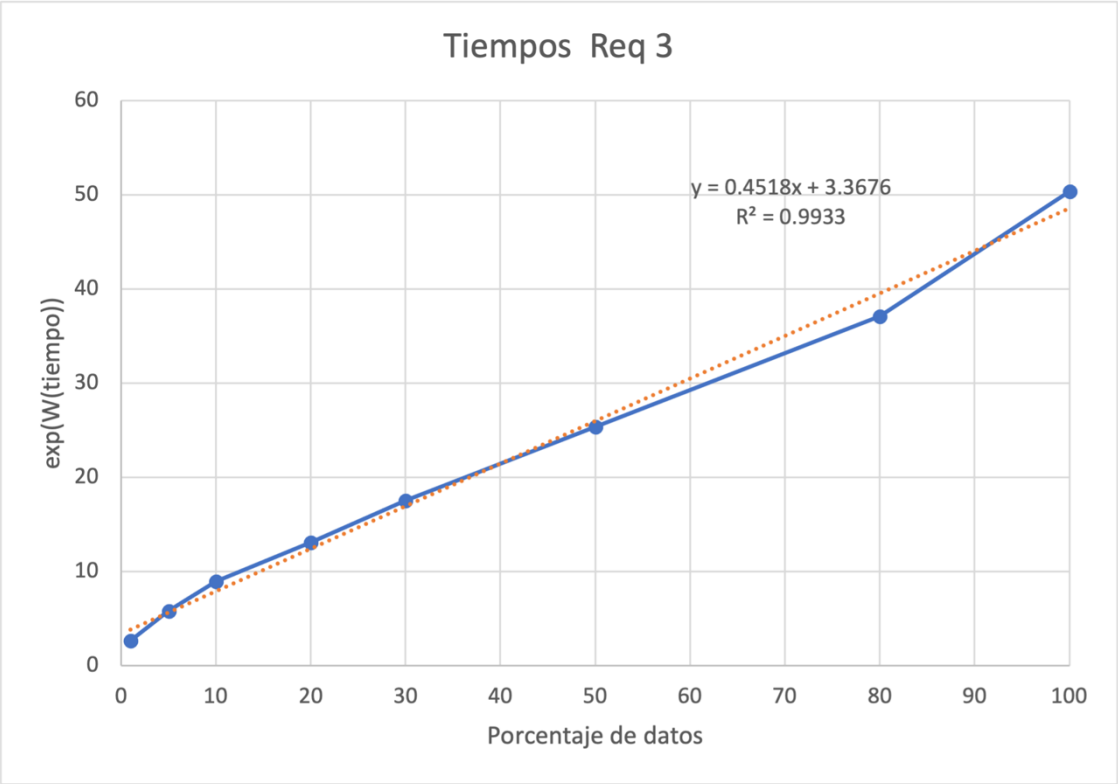
## Pruebas

| | |
|---|---|
| **Especificaciones** | 1,4 GHz Intel Core i5 de cuatro núcleos, 8 GB 2133 MHz LPDDR3, macOS 13.3.1 |
| **Clase** | CHOQUE |
| **Via** | AV AVENIDA DE LAS AMERICAS |

| Entrada | Tiempo (ms) |
|---|---|
| 1% (2352 accidentes) | 2.51 |
| 5% (11692 accidentes) | 10.17 |
| 10% (23124 accidentes) | 19.49 |
| 20% (45230 accidentes) | 33.49 |
| 30% (66532 accidentes) | 50.06 |
| 50% (106724 accidentes) | 81.98 |
| 80% (161640 accidentes) | 134.03 |
| 100% (195414 accidentes) | 197.52 |

## Gráficas

Una vez más, como sospechamos que nuestros datos van a seguir un orden de crecimiento O(Nlog(N)), en vez de graficar el tiempo, graficaremos exp(W(tiempo)). Si obtenemos una función lineal, sabremos que el orden de crecimiento es O(Nlog(N)).




## Análisis de resultados

Una vez más vemos que los resultados experimentales coinciden con los resultados teóricos, pues el crecimiento lineal de exp(W(tiempo)) implica que tiempo crece como O(Nlog(N)). La complejidad del requerimiento radica principalmente en que añadir un elemento a un RBT tiene complejidad O(log(N)) y esto se hace N veces pues la lista de elementos dada una clase crece con el conjunto de datos.

## Requerimiento 4

```python
def req_4(data_structs,fecha_inicial,fecha_final,gravedad):
    """
    Función que soluciona el requerimiento 4
    """
    # TODO: Realizar el requerimiento 4
    data_structs_gravedad=data_structs["acc_gravedad"]
    data_gravedad_k_v=mp.get(data_structs_gravedad,gravedad)
    data_gravedad=me.getValue(data_gravedad_k_v)
    map_gravedad=om.newMap("RBT")

    for elem in lt.iterator(data_gravedad):
        data_fecha_str=elem["FECHA_OCURRENCIA_ACC"]
        data_fecha_list=data_fecha_str.split("/")
        fecha_elem= datetime.date(day=int(data_fecha_list[2]), month=int(data_fecha_list[1]),year=int(data_fecha_list[0]))
        if fecha_elem>=fecha_inicial and fecha_elem<=fecha_final:
            if not (om.contains(map_gravedad,data_fecha_str)):
                om.put(map_gravedad,data_fecha_str,lt.newList("ARRAY_LIST"))
                gravedad_fecha_list=om.get(map_gravedad,data_fecha_str)["value"]
                lt.addLast(gravedad_fecha_list,elem)
            else:
                gravedad_fecha_list=om.get(map_gravedad,data_fecha_str)["value"]
                lt.addLast(gravedad_fecha_list,elem)

    result_list=merg.sort(gravedad_fecha_list,sort_criteria_fecha)
    return result_list
```

## Descripción

Este requerimiento permite identificar los 5 accidentes más recientes para un intervalo de fechas y una gravedad dadas.

| | |
|---|---|
| **Entrada** | Data structs, fecha inicial, fecha final,gravedad |
| **Salidas** | Result_list |
| **Implementado (Si/No)** | Sí |

## Análisis de complejidad

| **Pasos** | **Complejidad** |
|---|---|
| 1) Mediante la data structs [“acc_gravedad”] se filtran los accidentes por la gravedad ingresada por el usuario | O(1) |
| 2) Se crea un RBT para almacenar los datos | O(1) |
| 3) Mediante un For, se recorre la lista data_gravedad | O(N) |
| 4) Se compara si la fecha del elemento se encuentra dentro del rango fecha inicial y final ingresados por | O(Log N) para om.contains<br>O(1) el recorrido que se hace para ir añadiendo a la lista |

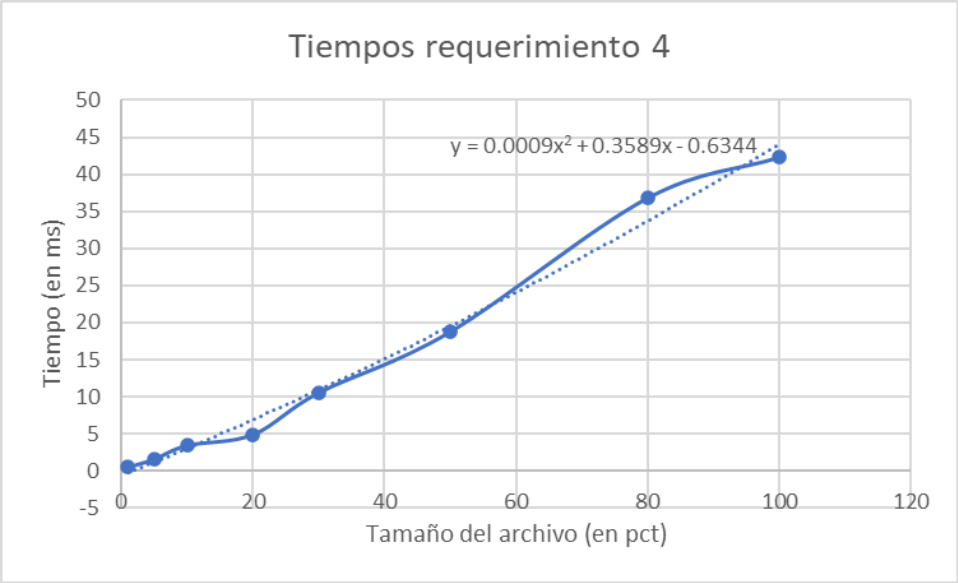
| | |
|---|---|
| parámetros. Si la gravedad no está en el árbol creado, se crea una nueva lista para que contenga la gravedad y sus valores. Si ya está, simplemente se actualizan los datos. | |
| 5) Se crea “result_list” que es la variable donde se van a almacenar los resultados. | O(log N) |
| 6) Mediante el algoritmo de ordenamiento MergeSort, organizamos los datos de “result_list” | O(Nlog(N)) * [O (1)] |

## Pruebas

| | |
|---|---|
| **Especificaciones** | Ryzen 5 5500U, 20 GB RAM DDR4, Windows 11 |
| **Fecha inicial- fecha final** | 01/10/2016-01/10/2018 |
| **Gravedad** | CON MUERTOS |

| **Entrada** | **Tiempo (ms)** |
|---|---|
| 1% (2352 accidentes) | 0.49 |
| 5% (11692 accidentes) | 1.62 |
| 10% (23124 accidentes) | 3.46 |
| 20% (45230 accidentes) | 4.93 |
| 30% (66532 accidentes) | 10.58 |
| 50% (106724 accidentes) | 18.79 |
| 80% (161640 accidentes) | 36.85 |
| 100% (195414 accidentes) | 38.34 |

## Gráficas




## Análisis de resultados

Teóricamente la complejidad es N log N, sin embargo, se debe tener en cuenta que al filtrar los datos por una gravedad específica y por un rango de fechas, esta “N” es mucho más pequeña ya que hay menor cantidad de datos por recorrer. Los datos no sufren mucha diferencia temporal entre archivo y archivo, por lo que se denota la eficiencia de las estructuras de datos.

# Requerimiento 5

## Descripción

```python
def req_5(data_structs, localidad_s, mes, anio):
    """
    Función que soluciona el requerimiento 5
    """
    # TODO: Realizar el requerimiento 5
    tupla_anio_mes = (int(anio),mes)
    datastructs_anio_mes = data_structs['acc_anio_mes']
    data_anio_mes_k_v = mp.get(datastructs_anio_mes, tupla_anio_mes)
    data_anio_mes = me.getValue(data_anio_mes_k_v)

    map_localidad=om.newMap ("RBT")

    for elem in lt.iterator(data_anio_mes):
        localidad = elem['LOCALIDAD']
        if not (om.contains(map_localidad,localidad)):
            om.put(map_localidad,localidad,lt.newList("ARRAY_LIST"))
            localidad_list=om.get(map_localidad,localidad)["value"]
            lt.addLast(localidad_list,elem)
        else:
            localidad_list=om.get(map_localidad,localidad)["value"]
            lt.addLast(localidad_list,elem)

    result_list=om.get(map_localidad,localidad_s)['value']
    result_list=merg.sort(result_list,sort_criteria_fecha_inversa)
    return result_list

```

| | |
|---|---|
| **Entrada** | Localidad, mes y año entre 2015 y 2022 |
| **Salidas** | Los 10 accidentes menos recientes ocurridos en un mes y año para la localidad dada, organizados del más reciente al más antiguo |
| **Implementado (Sí/No)** | Si se implementó lo realizó Laura Hurtado |

## Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

| **Pasos** | **Complejidad** |
|---|---|
| Línea 239: Se crea una tupla con los valores del mes y el año, los cuales están organizados con la variable “acc_anio_mes” como un mapa. Se | O (1) |

| | |
|---|---|
| accede a la estructura de datos y se retorna la lista con los datos del año y mes a consultar (data_anio_mes) | |
| Línea 244: Se crea una tabla de símbolos RBT donde se van a encontrar las localidades | O (1) |
| Línea 246: Se recorre el data del año y el mes ingresados para buscar si esta la localidad. | O (N) siendo N el número de registros |
| Línea 248: Si la localidad no está en el árbol creado, se crea una lista ingresando la localidad con sus valores, añadiendo en la última posición, si esta la actualiza. | O(logN) om.contains<br>O (1) el recorrido agregando a la lista |
| Línea 256: se crea una variable (result_list) la cual va a retornar la localidad que entro como parámetro ya filtrado con sus valores correspondientes. | O(logN) |
| Línea 257: Organizamos el resultado por medio del merg sort de forma descendente. | O(Nlog(N)) * [O (1)] |
| Línea 258: Retorna la lista de resultados ordenada | O (1) |
| ***TOTAL*** | ***O(NlogN)*** |

## 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** | Intel(R) Core (TM) i7-6500U CPU @ 2.50GHz 2.59 GHz |
| **Memoria RAM** | 12,0 GB (11,9 GB utilizable) |
| **Sistema Operativo** | Windows 10 Sistema operativo de 64 bits, procesador x64 |
| **Entrada: Fontibón, noviembre,2016** | |
| **Entrada** | **Tiempo (s)** |
| small | 1.96 |
| 5 pct | 4.97 |
| 10 pct | 5.52 |
| 20 pct | 16.67 |
| 30 pct | 32.58 |
| 50 pct | 42.64 |
| 80 pct | 64.17 |
| large | 98.98 |

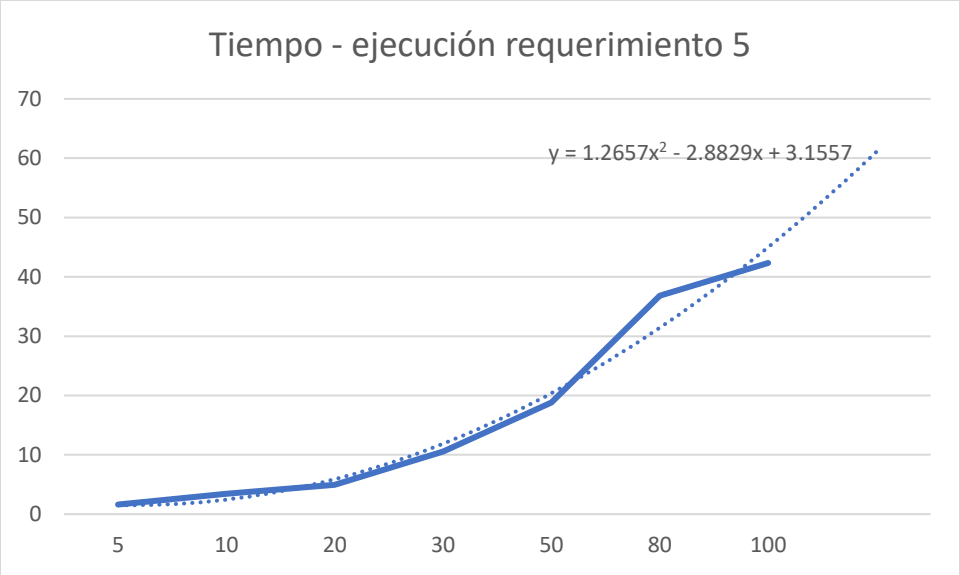
## Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

| Muestra | Salida | Tiempo (ms) |
|---|---|---|
| small | Los 10 accidentes menos recientes ocurridos en un mes y año dada la localidad organizados del más reciente al más antiguo. | 1.96 |
| 5 pct | Los 10 accidentes menos recientes ocurridos en un mes y año dada la localidad organizados del más reciente al más antiguo. | 4.97 |
| 10 pct | Los 10 accidentes menos recientes ocurridos en un mes y año dada la localidad organizados del más reciente al más antiguo. | 5.52 |
| 20 pct | Los 10 accidentes menos recientes ocurridos en un mes y año dada la localidad organizados del más reciente al más antiguo. | 16.67 |
| 30 pct | Los 10 accidentes menos recientes ocurridos en un mes y año dada la localidad organizados del más reciente al más antiguo. | 32.58 |
| 50 pct | Los 10 accidentes menos recientes ocurridos en un mes y año dada la localidad organizados del más reciente al más antiguo. | 42.64 |
| 80 pct | Los 10 accidentes menos recientes ocurridos en un mes y año dada la localidad organizados del más reciente al más antiguo. | 64.17 |
| large | Los 10 accidentes menos recientes ocurridos en un mes y año dada la localidad organizados del más reciente al más antiguo. | 98.98 |

## Graficas

Las gráficas con la representación de las pruebas realizadas.




## Análisis

Como se puede observar teóricamente la complejidad resultante es nlogn donde el N es mucho menor al N original ya que los datos al ser filtrados por mes y año y localidad van a ser menores, el tiempo de ejecución n log n es simplemente el resultado de realizar una operación Θ (log n) n veces en el ordenamiento del árbol binario creado. En la gráfica resultante al ejecutar el requerimiento 5 tiene un comportamiento exponencial como se puede visualizar la complejidad temporal NlogN donde a mayores datos el tiempo va a aumentar sin superar los 100ms.

# Requerimiento 6

```
def req_6(data_structs, mes, anio, radio, latitud, longitud, numero_acc):
    """
    Función que soluciona el requerimiento 6
    """
    # TODO: Realizar el requerimiento 6
    tupla_anio_mes = (anio,mes)
    datastructs_anio_mes = data_structs['acc_anio_mes']
    data_anio_mes_k_v = mp.get(datastructs_anio_mes, tupla_anio_mes)
    data_anio_mes = me.getValue(data_anio_mes_k_v)

    latitud_rad = float(latitud) * (m.pi/180)
    longitud_rad = float(longitud) * (m.pi/180)

    RBT_distancia = om.newMap('RBT')

    for elem in lt.iterator(data_anio_mes):
        elem_latitud_deg = float(elem['LATITUD'])
```

```
    elem_longitud_deg = float(elem['LONGITUD'])
    elem_latitud_rad = elem_latitud_deg * (m.pi/180)
    elem_longitud_rad = elem_longitud_deg * (m.pi/180)
    distancia = calcular_distacia_rad(elem_latitud_rad, elem_longitud_rad, latitud_rad, longitud_rad)
    RBT_distancia_aux = RBT_distancia
    if distancia <= radio:
        RBT_distancia = om.put(RBT_distancia_aux, distancia, elem)

list_result = lt.newList('ARRAY_LIST', cmpfunction= compare_id)

if om.isEmpty(RBT_distancia):
    return list_result

for i in range(numero_acc):
    tupla_elem_min_key = om.minKey(RBT_distancia)
    elem_min = me.getValue(om.get(RBT_distancia, tupla_elem_min_key))
    if not lt.isPresent(list_result, elem_min):
        distancia = tupla_elem_min_key
        elem_min['DISTANCIA'] = distancia
    lt.addLast(list_result, elem_min)
    RBT_distancia_aux = RBT_distancia
    if om.size(RBT_distancia) > 1:
        RBT_distancia = om.deleteMin(RBT_distancia_aux)

return list_result
```

## Descripción

En este requerimiento, se buscan los N accidentes ocurridos en un año/mes dentro de una zona circular de cierto radio centrada en una latitud/longitud definida por parámetro.

| | |
|---|---|
| **Entrada** | data_structs, año, mes, radio, latitud, longitud, numero_acc |
| **Salidas** | List_result |
| **Implementado (Sí/No)** | Si |

## Análisis de complejidad

| **Pasos** | **Complejidad** |
|---|---|
| Paso 1: Utilizando la estructura de datos data_structs[‘acc_clase’], se sacan todos los accidentes de en cierto año/mes y se guardan en la variable data_anio_mes. | O(1) |
| Paso 2: Se convierten la latitud y longitud pasadas por parámetro a radianes. | O(1) |
| Paso 3: Se crea un nuevo RBT para organizar los datos por distancia, RBT_distancia | O(1) |
| Paso 4: Se recorre data_anio_mes. Para cada elemento, se convierten su latitud y longitud a radianes y se calcula su distancia de la latitud y longitud pasadas por parámetro utilizando la función calcular_distancia_rad(), que | O(Nlog(N)) |

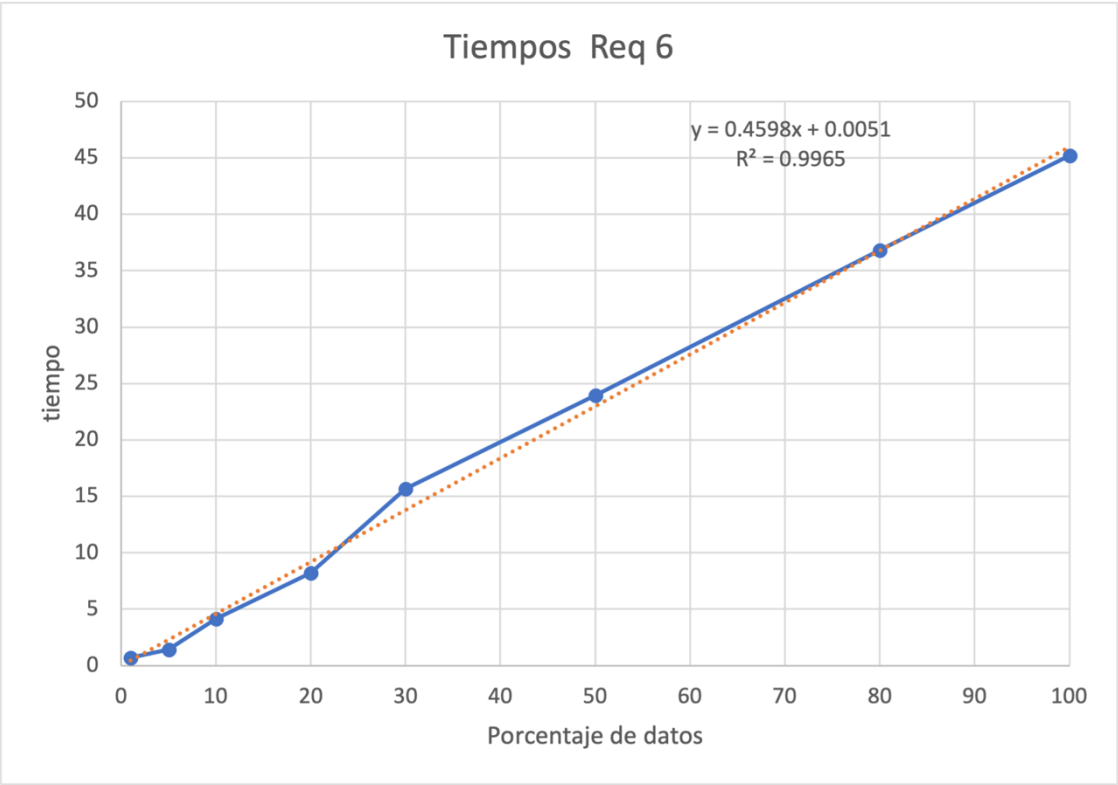
| | |
|---|---|
| simplemente recibe las latitudes y longitudes pasadas por parámetro y utilizando la fórmula de Haversine, retorna su distancia. Si la distancia es menor o igual al radio pasado por parámetro, se añade el elemento a RBT_distancia, con llave igual a la distancia entre el accidente y las coordenadas pasadas por parámetro y valor el elemento. | |
| Paso 5: Se crea una lista para mostrar el resultado, result. | O(1) |
| Paso 6: Se obtiene el mínimo elemento de RBT_distancia y si no está presente en result, se le añade. Luego, se elimina el elemento mínimo de RBT_distancia. Se repite ese proceso el numero_acc veces. | O(log(N)) |
| **Total** | **O(Nlog(N))** |

## Pruebas

| | |
|---|---|
| **Especificaciones** | 1,4 GHz Intel Core i5 de cuatro núcleos, 8 GB 2133 MHz LPDDR3, macOS 13.3.1 |
| **Año** | 2022 |
| **Mes** | Enero |
| **Radio** | 15 |
| **Latitud** | 4.674 |
| **Longitud** | -74.068 |
| **Num act** | 4 |

| **Entrada** | **Tiempo (ms)** |
|---|---|
| 1% (2352 accidentes) | 0.69 |
| 5% (11692 accidentes) | 1.44 |
| 10% (23124 accidentes) | 4.17 |
| 20% (45230 accidentes) | 8.19 |
| 30% (66532 accidentes) | 15.67 |
| 50% (106724 accidentes) | 23.97 |
| 80% (161640 accidentes) | 36.80 |
| 100% (195414 accidentes) | 45.22 |

## Gráficas




## Análisis requerimiento

En este caso no hubo necesidad de incorporar ninguna función auxiliar pues los datos siguen un crecimiento lineal. Por ende, es posible que nos hayamos equivocado en el analisis de complejidad teórico. Posiblemente, el número de accidentes dentro de un radio no crece como el conjunto de datos, y el árbol RBT_distancias, tiene un tamaño relativamente constante a medida que aumenta el tamaño de los datos. En ese caso, el paso 4 tendría complejidad O(N). Si este fuera el caso, podríamos explicar el cambio de pendiente entre el 20% y 30% de los datos, pues posiblemente, el archivo de 30% tenga más elementos a una distancia menor a 15km del punto (4.674,-74.068). En ese caso RBT_distancias sería mayor y la complejidad del paso 4 incrementaría también. Posteriormente, posiblemente, no se agreguen nuevos datos que cumplan las condiciones al archivo y por ende la pendiente se mantiene constante entre 30% y 100%.

# Requerimiento 7

```
def req_7(data_structs, mes, anio):
    """
    Función que soluciona el requerimiento 7
    """
    # TODO: Realizar el requerimiento 7

    mapa_horas = om.newMap('BST')

    tupla_anio_mes = (anio,mes)
    datastructs_anio_mes = data_structs['acc_anio_mes']
    data_anio_mes_k_v = mp.get(datastructs_anio_mes, tupla_anio_mes)
    data_anio_mes = me.getValue(data_anio_mes_k_v)

    rbt_fechas = om.newMap(omaptype='RBT')

    for elem in lt.iterator(data_anio_mes):

        elem_hora_str = elem['HORA_OCURRENCIA_ACC']
        elem_hora_list = elem_hora_str.split(':')
        elem_hora = datetime.time(hour = int(elem_hora_list[0]), minute=int(elem_hora_list[1]), second =
int(elem_hora_list[2]))

        elem_hora_interval = datetime.time(hour=int(elem_hora_list[0]), minute=0, second=0)

        if not om.contains(mapa_horas, str(elem_hora_interval)):
            mapa_horas_aux = mapa_horas
            mapa_horas = om.put(mapa_horas_aux, str(elem_hora_interval), 1)
        else:
            num_elems = me.getValue(om.get(mapa_horas, str(elem_hora_interval)))
            num_elems += 1
            mapa_horas_aux = mapa_horas
            mapa_horas = om.put(mapa_horas_aux, str(elem_hora_interval), num_elems)

        elem['HORA_ACC'] = elem_hora

        elem_fecha_str = elem['FECHA_OCURRENCIA_ACC']
        elem_fecha_list = elem_fecha_str.split('/')
        elem_fecha = datetime.date(year = int(elem_fecha_list[0]), month = int(elem_fecha_list[1]), day =
int(elem_fecha_list[2]))
        elem['FECHA_ACC'] = elem_fecha

        if not om.contains(rbt_fechas, elem_fecha):
            min_heap_fecha_ = mpq.newMinPQ(compare_time_minpq)
            max_heap_fecha_ = mpq.newMinPQ(compare_time_maxpq)

            min_heap_fecha = mpq.insert(min_heap_fecha_,elem)
```

```
            max_heap_fecha = mpq.insert(max_heap_fecha_,elem)

            tupla_min_max_heap = (min_heap_fecha, max_heap_fecha)
            rbt_fechas_aux = rbt_fechas
            rbt_fechas = om.put(rbt_fechas_aux, elem_fecha, tupla_min_max_heap)
        else:
            key_value_fecha = om.get(rbt_fechas, elem_fecha)
            tupla_min_max_aux = me.getValue(key_value_fecha)
            min_heap = tupla_min_max_aux[0]
            max_heap = tupla_min_max_aux[1]

            min_heap_aux = min_heap
            max_heap_aux = max_heap

            min_heap = mpq.insert(min_heap_aux, elem)
            max_heap = mpq.insert(max_heap_aux,elem)

            tupla_min_max = (min_heap, max_heap)
            rbt_fechas_aux = rbt_fechas
            rbt_fechas = om.put(rbt_fechas_aux, elem_fecha, tupla_min_max)

    for h in range(0,24):
        time_interval = str(datetime.time(h, 0, 0))
        if not om.contains(mapa_horas, time_interval):
            mapa_horas_aux = mapa_horas
            mapa_horas = om.put(mapa_horas_aux, time_interval, 0)

    lista_result = lt.newList('ARRAY_LIST')

    for key in lt.iterator(om.keySet(rbt_fechas)):
        lista_fecha = lt.newList('ARRAY_LIST')
        min_max_heap_max_elem = me.getValue(mp.get(rbt_fechas, key))

        min_heap_max_elem = min_max_heap_max_elem[0]
        max_heap_max_elem = min_max_heap_max_elem[1]

        min_elem_heap = mpq.delMin(min_heap_max_elem)
        max_elem_heap = mpq.delMin(max_heap_max_elem)

        lt.addLast(lista_fecha, max_elem_heap)
        lt.addLast(lista_fecha, min_elem_heap)

        lt.addLast(lista_result, lista_fecha)
    return mapa_horas, lista_result
```

## Descripción del requerimiento

Se quiere conocer el accidente más temprano y más tarde para cada día de un mes/año dados, además, se quiere hacer un histograma que muestre el número total de accidentes ocurridos en cada hora del día para ese mes/año.

| | |
|---|---|
| **Entrada** | data_structs, año, mes |
| **Salidas** | mapa_horas, lista_result |
| **Implementado (Sí/No)** | Si |

## Análisis de complejidad

| **Pasos** | **Complejidad** |
|---|---|
| Paso 1: Se crea una tabla de hash para contar el número de accidentes en cada hora del día, mapa_horas | O(1) |
| Paso 2: Utilizando la estructura de datos data_structs['acc_clase'], se sacan todos los accidentes de en cierto año/mes y se guardan en la variable data_anio_mes. | O(1) |
| Paso 3: Se crea un RBT para guardar todos los días del año, rbt_fechas (el único propósito de este RBT es mostrar y organizar los días del mes) | O(1) |
| Paso 4: Se recorre data_anio_mes | O(N*N) |
| Paso 4.1: Se saca la hora específica del accidente y el intervalo horario en el que ocurrió. | O(1) |
| Paso 4.2: Se revisa si el intervalo horario es una llave de mapa_horas, si no lo es, se inserta como llave su valor correspondiente es 1, si lo está, simplemente se suma 1 al valor del intervalo horario ya existente. | O(1) |
| Paso 4.3: Se revisa si la fecha del accidente está en rbt_fechas, si no lo está, se crea una tupla con un minPQ y un maxPQ ordenados por hora, estos guardarán el elemento más temprano y más tarde para cada fecha. Se inserta el accidente en el minPQ y el maxPQ y se inserta la pareja llave valor en rbt_fechas, siendo la fecha la llave y la tupla con el minPQ y maxPQ el valor. Si la fecha si está en rbt_fechas, simplemente se recupera y se inserta el accidente en el minPQ y maxPQ correspondientes a la fecha. Se vuelve a insertar una versión actualizada de la tupla al diccionario. | O(N) |
| Paso 5: Se revisa cada hora del día para ver si está en mapa_horas. Si no lo está, se añade y su valor es 0 (Esto sirve únicamente para también mostrar las horas sin accidentes en el histograma final) | O(1) |
| Paso 6: Se crea una lista para presentar el resultado. | O(1) |
| Paso 7: Se recorren las llaves de rbt_fechas, para cada llave, es decir cada día del mes, se crea una lista. Luego, se saca el primer elemento del minPQ y del maxPQ, estos son el accidente más temprano y más tarde respectivamente. Se añaden estos dos accidentes a la lista del día y se añade la lista del día al resultado. | O(1) |
| Paso 8: Se retorna mapa_horas y lista_result | O(1) |

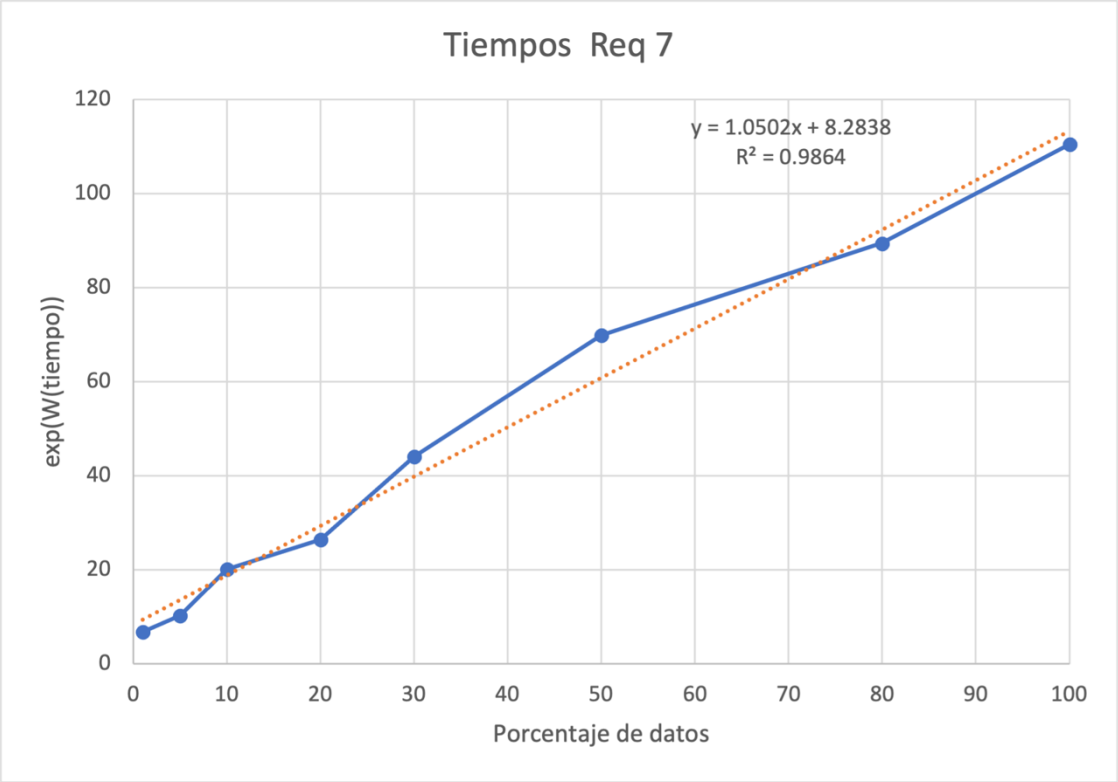
| **Total** | **O(N*N)** |
|---|---|

## Pruebas

| **Especificaciones** | 1,4 GHz Intel Core i5 de cuatro núcleos, 8 GB 2133 MHz LPDDR3, macOS 13.3.1 |
|---|---|
| **Año** | 2019 |
| **Mes** | Abril |

| **Entrada** | **Tiempo (ms)** |
|---|---|
| 1% (2352 accidentes) | 6.75 |
| 5% (11692 accidentes) | 10.19 |
| 10% (23124 accidentes) | 20.05 |
| 20% (45230 accidentes) | 26.41 |
| 30% (66532 accidentes) | 44.04 |
| 50% (106724 accidentes) | 69.79 |
| 80% (161640 accidentes) | 89.42 |
| 100% (195414 accidentes) | 110.47 |

## Gráficas




## Análisis del requerimiento

El requerimiento tuvo mejor desempeño del que esperábamos, pues en el análisis teórico este tuvo complejidad O(N*N) y mientras que experimentalmente se comportó como O(N).

Supusimos que al recorrer los N elementos de data_anio_mes, y en cada iteración del recorrido agregar un elemento a un minPQ y un maxPQ, dos operaciones con complejidad O(N), obtendríamos una complejidad de O(N*N). No obstante es posible que el minPQ y maxPQ de cada fecha tengan un numero de elementos que no cambie mucho con el tamaño de los datos y se comporten como O(1). En este requerimiento las complejidades del RBT se toman como constantes pues las llaves de este son simplemente todos los días en determinado mes y esto no cambia a medida que aumentamos el tamaño del archivo, por ende se comporta como O(1). Al hacer un recorrido inorder al sacar las llaves (función om.keySet()) pudimos presentar el primer y último accidente de cada día con los días del mes ordenados de menor a mayor. La lectura de la tabla de hash de las frecuencias se hace en el view.

# Requerimiento 8

```
def req_8(data_structs, fecha_inical, fecha_final, clase):
    """
    Función que soluciona el requerimiento 8
    """
    # TODO: Realizar el requerimiento 8

    hash_elems= mp.newMap(numelements=8, maptype='PROBING', loadfactor=0.5)

    contador = 0

    datastructs_clase = data_structs['acc_clase']
    data_clase_k_v = mp.get(datastructs_clase, clase)
    data_clase = me.getValue(data_clase_k_v)

    for elem in lt.iterator(data_clase):
        data_fecha_str = elem['FECHA_OCURRENCIA_ACC']
        data_fecha_list = data_fecha_str.split('/')
        fecha_elem = datetime.date(day = int(data_fecha_list[2]), month = int(data_fecha_list[1]), year =
int(data_fecha_list[0]))
        accidente_gravedad = elem['GRAVEDAD']
        if fecha_elem>=fecha_inical and fecha_elem <= fecha_final:
            agregar_a_filtro(hash_elems, accidente_gravedad, elem)
            contador += 1

    return contador, hash_elems
```

## Descripción del requerimiento

Se quiere crear un mapa que muestre todos los accidentes de una clase determinada en un rango de fechas, con cada gravedad asignada a un color de marcador.

| | |
|---|---|
| **Entrada** | data_structs, fecha_inicial, fecha_final, clase |
| **Salidas** | contador, hash_elems |
| **Implementado (Sí/No)** | Si |

## Análisis de complejidad

| **Pasos** | **Complejidad** |
|---|---|
| Paso 1: Utilizando la estructura de datos ‘acc_clase’ se accede a la lista con elementos de la clase pasada por parámetro, data_clase. | **O(1)** |
| Paso 2: Se inicializa un contador | **O(1)** |
| Paso 3: Se crea una tabla de hash para filtrar por gravedad, hash_elems | **O(1)** |
| Paso 3: Se recorre data_clase, para cada elemento, se revisa si ocurrió entre fecha_inicial y fecha_final, si sí, se añade 1 al contador y se agrega este elemento a hash_elems, utilizando la función agregar_a_filtro. | **O(N)** |

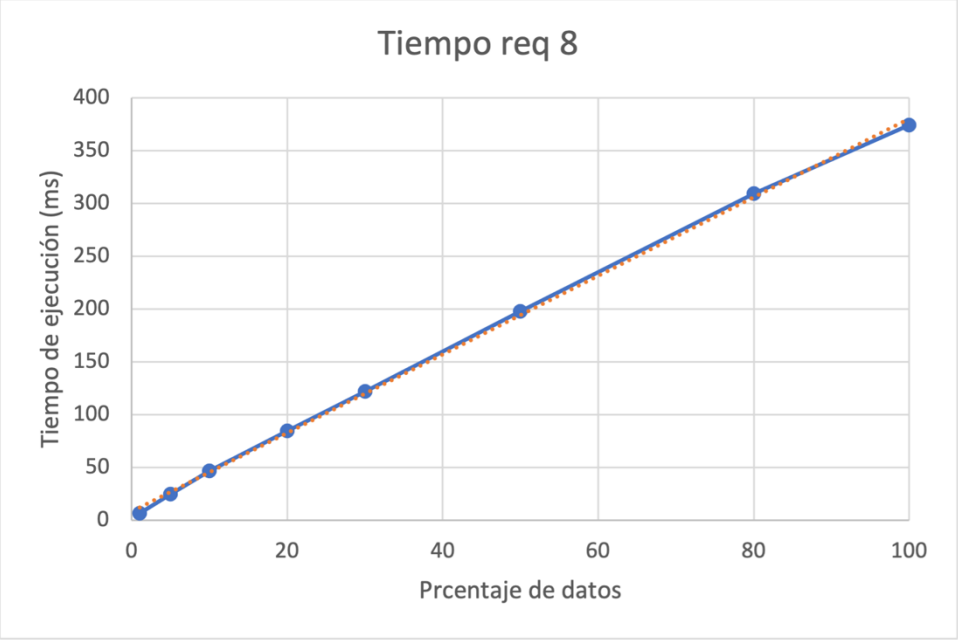
| | |
|---|---|
| Paso 4: Se retorna el contador y hash_elems, en el view se recorre hash_elems y se arma el mapa. | **O(1)** |
| **Total** | **O(N)** |

## Pruebas

| | |
|---|---|
| **Especificaciones** | 1,4 GHz Intel Core i5 de cuatro núcleos, 8 GB 2133 MHz LPDDR3, macOS 13.3.1 |
| **Año** | 2019 |
| **fecha_inic** | 01/01/2020 |
| **fecha_fin** | 01/02/2020 |
| **clase** | CHOQUE |

| **Entrada** | **Tiempo (ms)** |
|---|---|
| 1% (2352 accidentes) | 6.23 |
| 5% (11692 accidentes) | 24.74 |
| 10% (23124 accidentes) | 46.39 |
| 20% (45230 accidentes) | 84.69 |
| 30% (66532 accidentes) | 121.93 |
| 50% (106724 accidentes) | 197.78 |
| 80% (161640 accidentes) | 309.26 |
| 100% (195414 accidentes) | 373.94 |

## Gráficas




## Análisis

Los resultados experimentales concuerdan con los resultados teóricos.