

Análisis de Resultados-Reto3-EDA

David Burgos – 201818326-REQ2

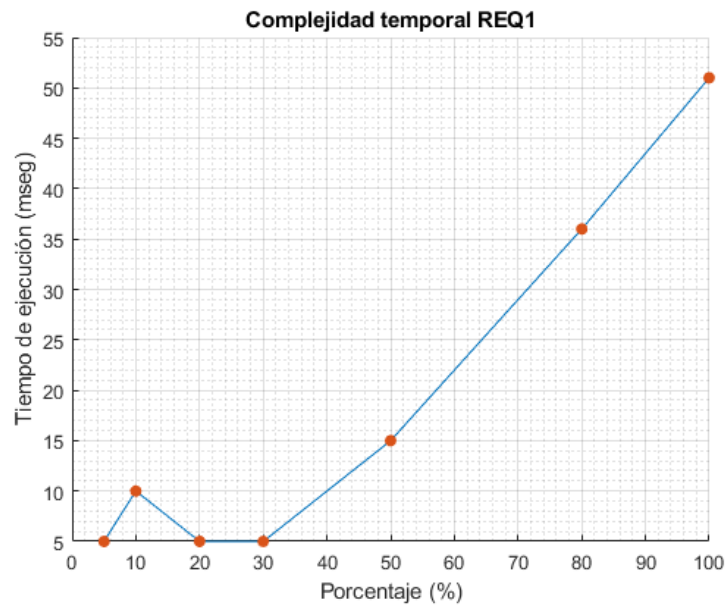
Andrés Mugnier – 201729994-REQ3

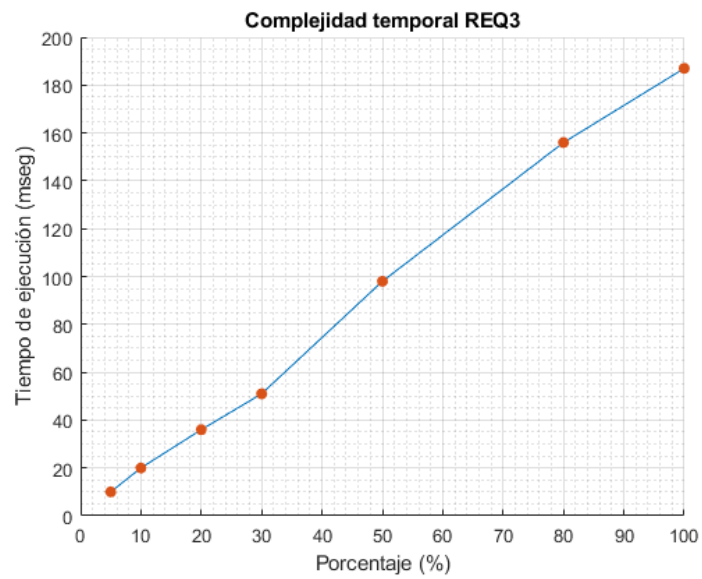
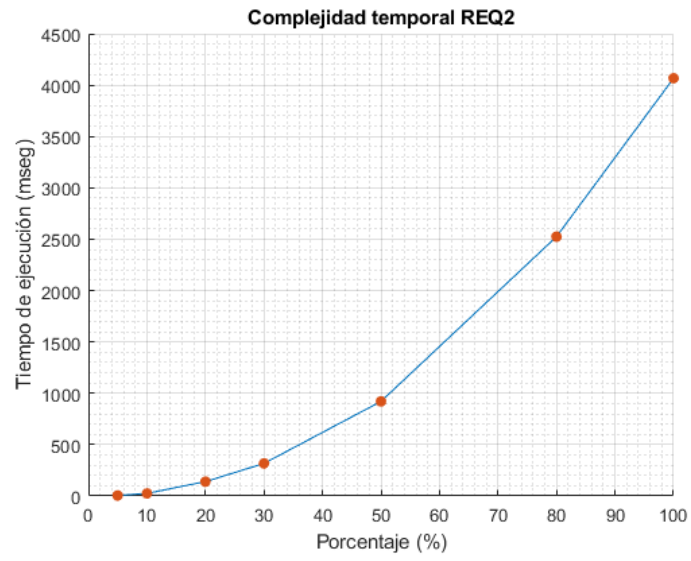
Análisis temporal experimental de los requerimientos:

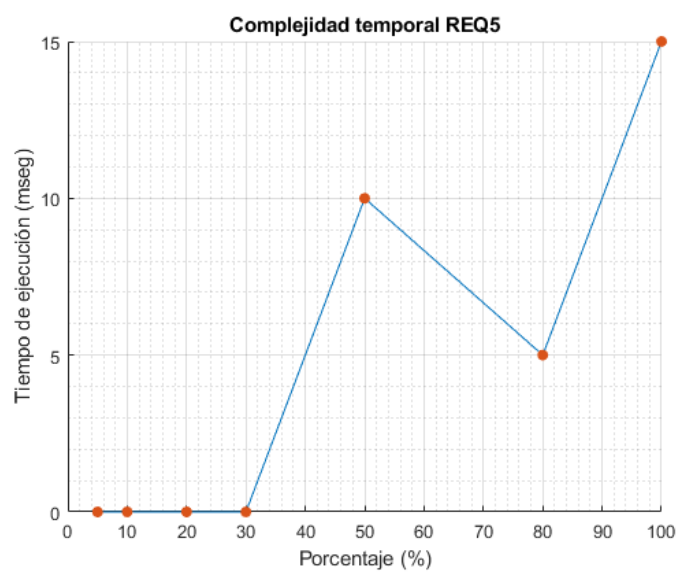
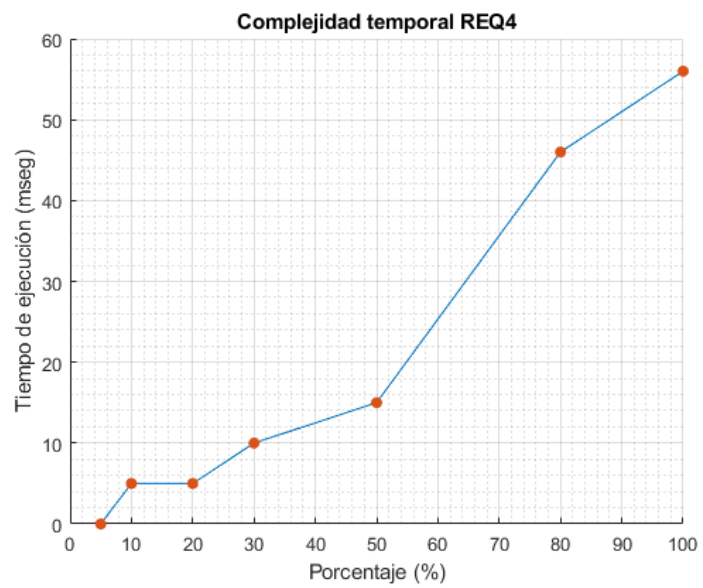
Se realizaron las pruebas de complejidad temporal para cada requerimiento utilizando los siguientes inputs:

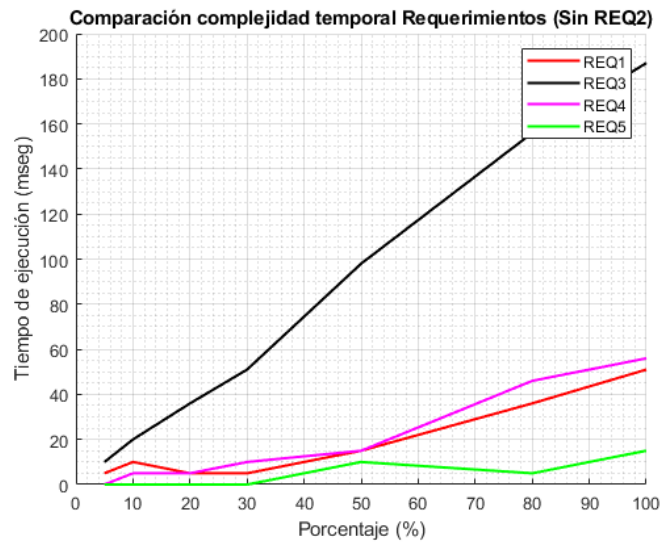
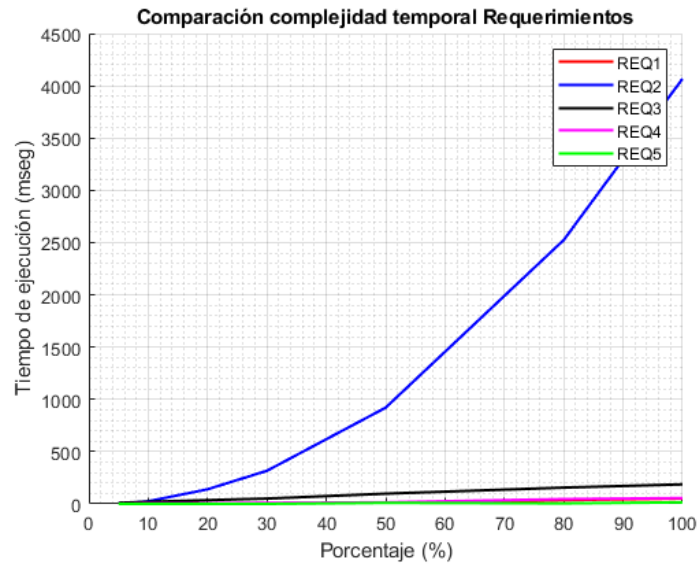
- REQ1: las vegas
- REQ2: 30-150 seg
- REQ3: 20:45-23:15
- REQ4: 1945-08-06, 1984-11-15
- REQ5: longitud -103.00 a -109.05 y una latitud 31.33 a 37.00.

Cada prueba se realizó 3 veces para cada porcentaje de los datos propuestos. Los resultados de estas pruebas se pueden observar en las siguientes imágenes:









Como se puede ver en las figuras anteriores, en todos los requerimientos se obtuvieron tiempos muy buenos, siendo el tiempo de respuesta más lento (REQ2, 100%) de 0.67 minutos. Esto se debe, en su mayoría a las estructuras de datos utilizadas durante la carga de datos, pues todos los requerimientos tenían prácticamente la misma estructura: busque los elementos en un rango $[liminf, limsup]$ para algún criterio y luego organice estos elementos por otro criterio. Por lo que se creó un RBT para cada requerimiento con las llaves del primer criterio y luego se organizaban con respecto al segundo criterio.

Como todos los arboles utilizados fueron RBT's, el tiempo más largo fue de hecho en la carga de datos, pues los RBT's hicieron muy eficientes las búsquedas ($O(\log(N))$) en todos los requerimientos, pues como ya sabemos los RBT's siempre están balanceados.

Análisis temporal teórico de los requerimientos:

- REQ1:

```
def AvistamienCiudad(catalog, ciudad):  
    ...  
    Devuelve todos los avistamientos de una ciudad y los organiza por fecha  
    ...  
  
    #Map ordenado por ciudades  
    principal = catalog['UFOSByCity'] → t  
    #Extrae la llave de la ciudad que se está buscando  
    especifico = om.get(principal, ciudad)["value"] → log(t)  
    #Ordena los avistamientos por fecha  
    mrgsort.sort(especifico, compByDateFormat) → t log(t)  
  
    return especifico
```

Para este REQ tenemos un `om.get()` de un RBT que tiene complejidad $\log(N)$ y luego un mergesort que tiene complejidad $N\log(N)$. Por lo tanto, la complejidad del REQ1 es $O(N\log(N))$.

- REQ2:

```
def AvistamienDireccion(catalog, limInf, limSup):
    """
    Retorna los avistamientos por duración [liminf,limsup] y los ordena por esa duración.
    """
    #Toma el map ordenado por duración
    principal = catalog['UFOSBySeconds']
    retorno = lt.newList()

    listaVal = [limInf]
    val1 = limInf
    val2 = limSup

    #Toma la llave máxima del map y su value y cuantos avistamientos tiene
    maximoLL = om.maxKey(principal)
    maximoComp = om.get(principal, maximoLL)["value"]
    maximoCant = lt.size(maximoComp)

    while val1 < val2:
        val1 += 1
        listaVal.append(val1)

    #Recorre las duraciones entre limInf y limSup y toma sus } M log(t)
    # avistamientos para añadirlos a retorno (Ya quedan ordenados)
    for x in listaVal:

        if om.contains(principal, float(x)):
            espezifico = om.get(principal, x)["value"]

            for x in range(lt.size(espezifico)):
                elemento = lt.getElement(espezifico, x+1)
                lt.addLast(retorno, elemento)

    return [retorno, [maximoLL, maximoCant]]
```

Para este REQ tenemos que si $M = \text{limsup} - \text{liminf}$ entonces al recorrer todas las posibles duraciones entre liminf y limsup tendríamos un ciclo de $M \cdot N$.

Si $N = \text{avistamientos en el rango } \text{limsup} - \text{liminf}$ entonces el segundo for no es cuadrático pues recorre N elementos (el árbol es un árbol de listas entonces para cada nodo tenemos que recorrer una lista, esto parece cuadrático pero no lo es).

Sin embargo, para cada uno de estos ciclos hacemos un `om.get()` y un `om.contains()`, por lo que la complejidad de este REQ termina siendo $O(N \cdot \log(N))$

- REQ3:

```

def AvistamientoHHMM(catalog,liminf,limsup):
    """
    Retorna la cantidad de avistamientos en el rango [liminf,limsup] y el top 3
    y más recientes y antiguos avistamientos en ese rango
    """
    #Toma el map ordenado por HHMM
    map=catalog['UFOSByHHMM'] → t
    #Extrae las llaves en el rango lim inf, lim sup
    KeysInRange=om.values(map,liminf,limsup) → log(t)

    #Iniciliza las fechas máximss y sus correspondientes ufos y las fechas mínimas con sus correspondientes ufos

    maxC1=datetime.datetime.strptime('1700-03-21', "%Y-%m-%d")
    maxC2=datetime.datetime.strptime('1700-03-21', "%Y-%m-%d")
    maxC3=datetime.datetime.strptime('1700-03-21', "%Y-%m-%d")
    maxC4=datetime.datetime.strptime('1700-03-21', "%Y-%m-%d")
    maxC5=datetime.datetime.strptime('1700-03-21', "%Y-%m-%d")

    eltoC1={'datetime':'','city':'','state':'','country':'','shape':'','durationS':'','durationHM':'',
    'comments':'','dateposted':'','latitude':'','longitude':''}

    eltoC2=eltoC3=eltoC4=eltoC5=eltoP1=eltoP2=eltoP3=eltoP4=eltoP5=eltoC1

    maxP1=datetime.datetime.strptime('2021-03-21', "%Y-%m-%d")
    maxP2=datetime.datetime.strptime('2021-03-21', "%Y-%m-%d")
    maxP3=datetime.datetime.strptime('2021-03-21', "%Y-%m-%d")
    maxP4=datetime.datetime.strptime('2021-03-21', "%Y-%m-%d")
    maxP5=datetime.datetime.strptime('2021-03-21', "%Y-%m-%d")

    i=0
    for Element in lt.iterator(KeysInRange):
        for Element2 in lt.iterator(Element): ... } N t

    #Se guardan en listas los UFOS con fecha máxima y los ufos con fecha mínima
    Pequenos=lt.newList()
    Grandes=lt.newList()
    lt.addLast(Pequenos,eltoC1)
    lt.addLast(Pequenos,eltoC2)
    lt.addLast(Pequenos,eltoC3)
    lt.addLast(Pequenos,eltoC4)
    lt.addLast(Pequenos,eltoC5)

    lt.addLast(Grandes,eltoP1)
    lt.addLast(Grandes,eltoP2)
    lt.addLast(Grandes,eltoP3)
    lt.addLast(Grandes,eltoP4)
    lt.addLast(Grandes,eltoP5)

    size=i

    return size,Pequenos,Grandes

```

En este requerimiento tenemos un `om.values()` para sacar los nodos en el rango requerido, por lo tanto tenemos una complejidad de $\log(N)$. Nuevamente, el doble for no es cuadrático pues M =número de avistamientos y N =número de nodos.

- REQ4: Este requerimiento es exactamente el mismo REQ3, por lo tanto tiene complejidad $O(\log(N))$ también.
- REQ5:

```

def AvistamienCordenadas(catalog, LonglimInf, LonglimSup, LatlimInf, LatlimSup):

    principal = catalog["UFOSByLONG"]
    retorno = lt.newList()

    KeysRangoLong = om.keys(principal, LonglimSup, LonglimInf)

    for x in range(lt.size(KeysRangoLong)):
        espezifico = om.get(principal, lt.getElement(KeysRangoLong, x+1))["value"]

        for y in range(lt.size(espezifico)):
            avis = lt.getElement(espezifico, y+1)

            if round(float(avis["latitude"]), 2) > LatlimInf and round(float(avis["latitude"]), 2) < LatlimSup:
                lt.addLast(retorno, avis)

    return retorno

```

Handwritten annotations in the image:

- Next to `principal = catalog["UFOSByLONG"]`: $\} t$
- Next to `KeysRangoLong = om.keys(...)`: $\rightarrow \log(t)$
- Next to `espezifico = om.get(...)`: $\rightarrow \log(t)$
- Next to the inner `for y in range(...)` loop: $\} t \log(t)$

Para este requerimiento nuevamente tenemos un `om.keys()` y un doble `for` que por las mismas razones anteriores no es cuadrático. Sin embargo, dentro de este `for` siempre se hace un `om.get()`, por lo que la complejidad de este req sería de $O(N \log(N))$.

- Resumen:

| Complejidades temporales | |
|--------------------------|----------------|
| Requisito | Complejidad |
| 1 | $O(N \log(N))$ |
| 2 | $O(N \log(N))$ |
| 3 | $O(\log(N))$ |
| 4 | $O(\log(N))$ |
| 5 | $O(N \log(N))$ |