Camilo Ortiz Cruz 201821615 c.ortizc@uniandes.edu.co

Kevin Fernando Gómez Camargo 202015120 k.gomezc@uniandes.edu.co

Maquinas de Pruebas

	Maquina
Procesador	11th Gen Intel® Core™ i7-1165G7 @ 2.80Ghz
Memoria RAM	16 GB
Sistema Operativo	Windows 10 Home 64-bits

Pruebas de Tiempo:

Los gráficos de las pruebas están al final del documento

	NumDatos	pReq1	pReq2	pReq4	pReq5	pReq6	Req.1	Req.2	Req.4	Req.5	Req.6
5pct	4016.0	2685.107496	19.0	216.0	43.0	43.0	3.125	0.0	0.000	3.125	18.750
10pct	8033.0	4556.130687	27.0	396.0	83.0	83.0	6.250	0.0	0.000	3.125	18.750
30pct	24099.0	9784.306305	52.0	1007.0	267.0	267.0	15.625	0.0	3.125	0.000	18.750
50pct	40166.0	14085.072168	67.0	1468.0	474.0	474.0	21.875	0.0	3.125	0.000	18.750
80pct	64265.0	19684.328288	80.0	2036.0	737.0	737.0	31.250	0.0	6.250	3.125	25.000
large	80332.0	23036.670800	86.0	2389.0	928.0	928.0	34.375	0.0	3.125	3.125	21.875

Nota:

El tiempo total de cargar todos los archivos y correr todas las funciones 5 veces fue en promedio de 17s.

pReqx hace referencia a los datos que utiliza la función.

Requerimiento 1:

Para poder realizar el requerimiento 1 se creó un RBT en el catálogo que tuviera como llave la ciudad y como valor una lista con todos los avistamientos en esa ciudad.

1. Se calcula el size del arbol rbt que tiene como llaves las ciudades donde sucedieron avistamiento $O(1)$
2. Se recorren todas las llaves del arbol y se agrega a una lista la llave y el valor $O(n)$
3. Se busca el maximo de la lista creada en el paso 2 $O(n)$
4. Se revisa si el arbol contiene la ciudad pasa por parametro $O(log(n))$

Si la contiene:

5. Se hace get de la ciudad en el arbol $O(log(n))$
6. Se calcula el size de los values de esa ciudad $O(1)$
7. Se ordena los values de esa ciudad $O(k \cdot log(k))$

Si la lista es menor que 6 elementos:

8. Se retorna los valores pedidos y los values ordenados ya que no se puede hacer los primero 3 y ultimos 3

Si la lsita tiene 6 o mas elementos:

9. Se hace sublist de los primer 3 elementos y ultimos 3 $O(1)$
10. Se retorna todo lo pedido y los primero 3 y ultimos 3

Si no la contiene:

11. Se retorna False

Total = $O(n + k \cdot log(k))$, depende de los datos

A partir del análisis realizado se llega a que la complejidad es O(n), consideramos que es una complejidad aceptable ya que toca revisar todas las ciudades para encontrar cual es la que más tiene por lo cual lo mínimo que se puede hacer es revisar todas las ciudades lo cual es n. Cabe resaltar que la complejidad del algoritmo será $O(k \cdot \log(k))$ cuando el número de avistamientos en una ciudad se mayor que el numero de ciudades en el árbol. Por tanto, se deja como $O(n + k \cdot \log(k))$ pues depende mucho de los datos que tengamos, pero en la mayoría de caso de este reto será O(n) puesto que en general hay más llaves que valores en una llave.

Al revisar las pruebas de tiempo se puede ver un crecimiento cuasi lineal pero ya que los tiempos son tan pequeños, el estado del computador puede afectar los tiempos.

Requerimiento 2 Camilo:

Para realizar el requerimiento 2 se creó un RBT que tenía como llaves la duración de un avistamiento y como valor una lista con los avistamientos:

1. Encontrar la llave maxima del arbol de duración $O(log(n))$
2. Encontrar el numero de elementos en la llave maxima $O(log(n))$
3. Sacar una lista con los valores en el rango dado por parametro $O(log(n) + k)$
4. Calcular el numero de elementos en la lista de los valores del paso 3 $O(k)$
5. Sacar los primero 3 valores $O(1)$
6. Sacar los ultimos 3 valores $O(1)$

Total = $O(k)$, depende de cuantas llaves hay en el rango y el numero de elementos del arbol

La complejidad de este requerimiento consideramos que es muy buena ya que para solucionarlo es necesario encontrar los valores que están en el rango dado, esto como mínimo será el número de elementos que hay en el rango (k) por ende el algoritmo se acerca casi a lo mínimo posible para solucionar el requerimiento, el algoritmo puede tener una complejidad $O(\mathrm{k} + \log(n))$ dependiente de si hay más llaves en el rango que numero de elementos en el rango, sin embargo, dado que k puede ser todas las llaves en el rango de acuerdo a los parámetros pasados esto sería como tener $\mathrm{O}(\mathrm{n} + \log(n))$ lo cual termina siendo $O(n)$, por eso se deja como $O(k)$, Si vemos las pruebas de tiempo notaremos que es extremadamente rápido, básicamente toma 0 ms para cualquier tamaño de archivo.

Requerimiento 3 Kevin Fernando:

Para el desarrollo de este requerimiento se implementó un árbol RBT con la hora de cada avistamiento como llave y un arreglo como valor en el que se van guardando los avistamientos con igual hora. También se cambió la función values de la carpeta DataStructures en el archivo rbt.py para que retornara un arreglo y mejorar los tiempos de ordenamiento.

Paso a paso:

1. Se halla la llave mayor que corresponde a la hora más tardía en la que se presentan avistamientos
2. Se busca el valor de esa llave
3. Se sacan los valores en el rango especificado (lista de listas)
4. Se convierte el arreglo de arreglos en un solo arreglo
5. Se hace un MergeSort del arreglo unificado
6. Se creand dos sublistas

Complejidad:

$$P_{1,2} = O(\log n)\ n\text{: numero de llaves del árbol}$$

$$P_3 = O(\log n + k)\ \ k\text{: llaves en el rango}$$

$$P_4 = O(k \times j)\ \ j\text{: numero de elementos (avistamientos) de una sola llave}$$

$$P_5 = O(m \log m)\ m\text{: elementos en todo el rango}$$

$$P_6 = O(1)$$

Complejidad total:

$$O(\log n) + O(\log n + k) + O(k \times j) + O(m \log m) + O(1)$$

$$O(\log n) + O(\log n + k) + O(m) + O(m \log m) + O(1)$$

$$O(m \log m)$$

El log de un número grande generalmente es pequeño, por esa razón y porque pueden haber muchas horas repetidas se omiten los dos primeros términos, el producto de las llaves en el rango por el número de elementos de cada llave (k x j) es igual al total de elementos (avistamientos) en el rango (m), por ende, siguiendo la notación Big O la complejidad total es O(mlog m) para m avistamientos en el rango. Es un orden linearitmico que es aceptable porque toca hacer un MergeSort en un arreglo para ordenar según el requerimiento y depende de los parámetros que se le pasen a la función así como de la distribución de los datos.

Requerimiento 4:

Para realizar el requerimiento 4 se creó un RBT que tuviera como llave la fecha de un avistamiento y como valor una lista con los avistamientos.

1. Calcular cuantas fechas diferentes tienen avistamientos $O(1)$
2. Encontrar cual es la fecha mas antigua en la que se ha reportado un avistamiento $O(log(n))$
3. Encontrar el numero de elementos de la fecha mas antigua $O(log(n))$
4. Sacar una lista con los valores en el rango dado por parametro $O(log(n) + k)$
5. Sacar cuantos elementos hay en el rango $O(k)$
6. Encontrar los primero 3 y ultimos 3 elementos $O(1)$

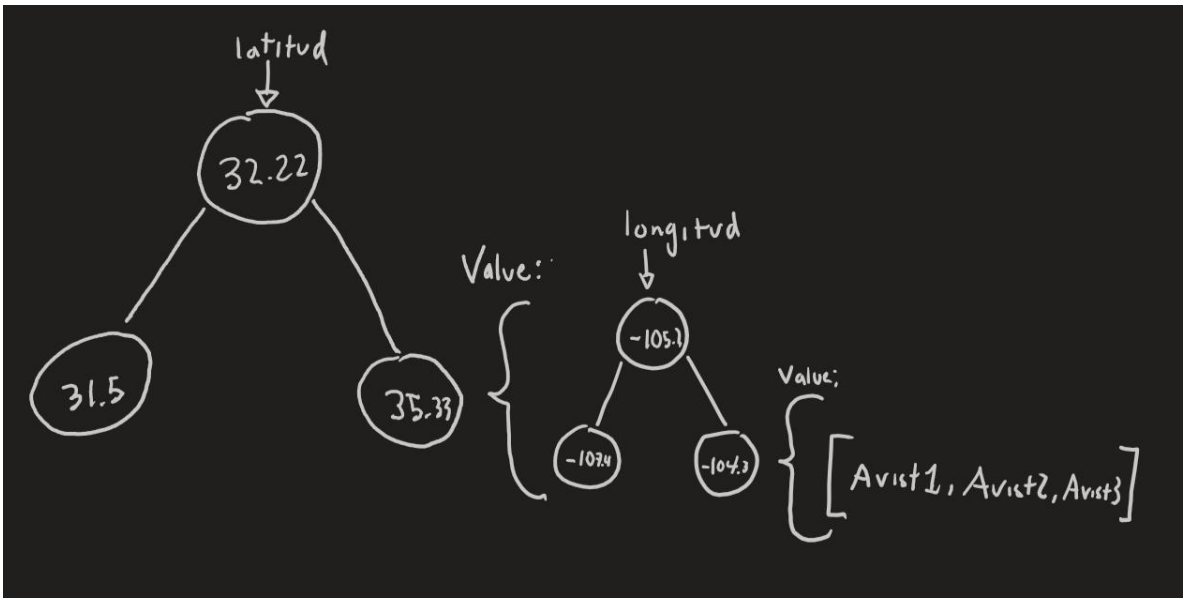
Total = $O(k)$, Depende de cuantas llaves tenga el arbol y cuantas llaves hay en el rango

Este algoritmo fue muy similar al requerimiento 2 pues los pasos para solucionarlos son casi idénticos cambiando solamente la llave sobre la que se quiere hacer el rango, consideramos que la complejidad es muy buena puesto que al igual que en el req. 2 se acerca casi a lo mínimo que se puede hacer para solucionarlo, al igual que en el 2 las pruebas de tiempo fueron de prácticamente 0 ms para todos los archivos, lo cual es muy bueno.

Requerimiento 5:

Para solucionar el requerimiento 5 se creó un RBT que tuviera como llave la latitud redondeada a 2 decimales y como valor tuviera un RBT con llave longitud y de valor una lista de los avistamiento que tuvieran esa latitud y longitud redondeada.

Diagrama estructura árbol:



Si utilizo latitud primero para que estuviera ordenado como se pide en el enunciado, primero por latitud y después por longitud.

1. Sacar una lista de los valores que tienen la llave latitud en el rango pasado por parametro $O(log(n) + k)$
2. Iterar sobre las los valores retornados en el paso 1 $O(k)$
3. Por cada valor de la iteración sacar la lista de los valores del valor (arbol) que esten en el rango de longitud $O(log(c) + e)$
4. Si el tamaño de la lista retornada es mayor a 0 iterar la lista de los valores del paso 3 y pasar todos los elementos a una lista $O(e \cdot p)$
5. Si hay mas de 10 elementos en el resultado hacer sublist de los primero 5 y ultimos 5 y retornar los valores $O(1)$
6. De lo contrario retornar la lista con todos los valores

Total = $O(k \cdot (log(c) + e \cdot p))$

La complejidad temporal de este algoritmo es más difícil de calcular ya que depende de diferentes elementos y estos son dependientes entre ellos, ya que es difícil saber si log(c) el cual es el número de elementos en el árbol de longitud es menor o mayor al número de avistamiento en el rango de longitud dado (e*p). Adicionalmente los valores de c y e*p no son los mismos pues depende en que nodo de latitud que estemos, no obstante, a partir de las pruebas y análisis de los datos se puede asumir que los valores de log(c) y e*p son pequeños, en base al archivo large los valores son:

```
El maximo numero de elementos en una lista 584 y el minimo 2
Elementos en promedio 7.62916575909388
El numero maximo de nodos en un arbols de longitud es 666 y el minimo 1
En promedio el numero de nodos de los arboles de longitud es 20.90890161374284
```

En base a esto podemos decir que la complejidad final es de $O(k \cdot e \cdot p)$ lo cual para simplificar se dejara como $O(k \cdot m) \mid m = e \cdot p$, siendo m el numero de elementos en el rango final, sin embargo, en las pruebas de tiempo se puede ver que incluso en el archivo large los tiempos de ejecución fueron también de casi 0

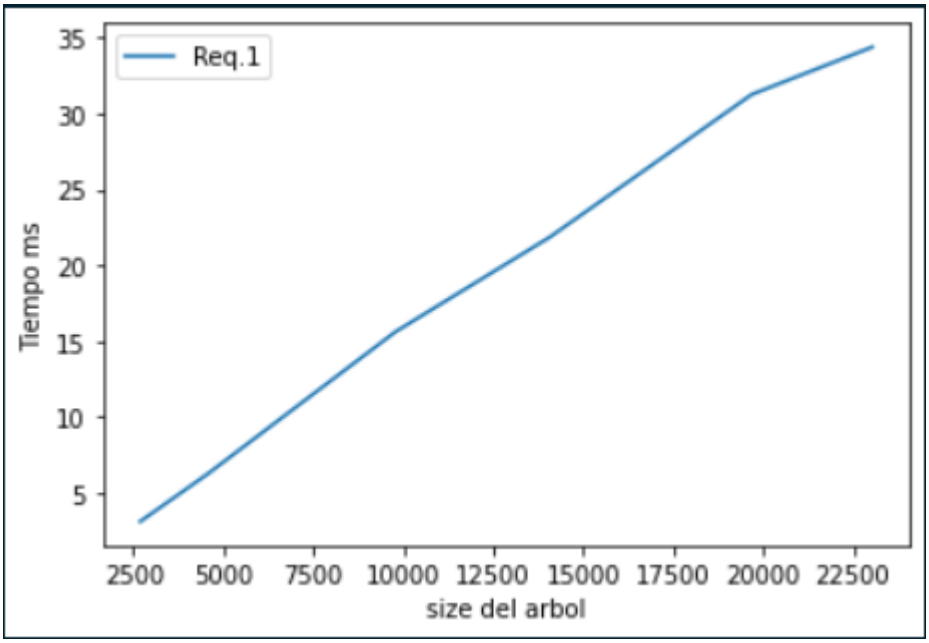
Para el análisis de los parámetros se usó el siguiente código:

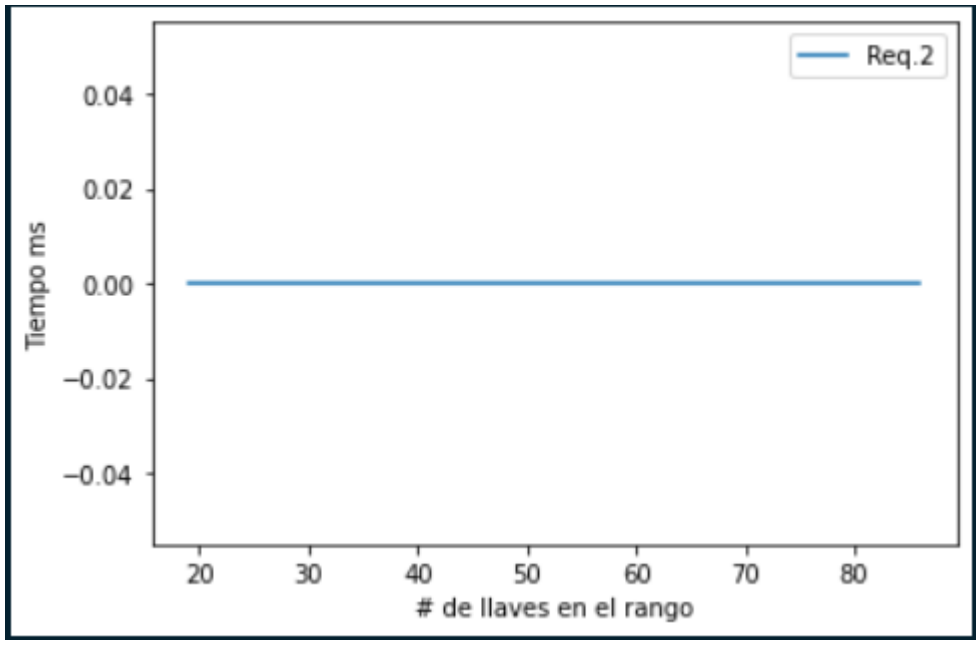
```
df = pd.read_csv(r"C:\Users\camil\OneDrive\Desktop\Los Andes\5to Semestre\EDA\Retos\Reto3-G08\Data\UFOS\UFOS-utf8-large.csv")
df["coord"] = df.apply(lambda x: (round(x.latitude,2), round(x.longitude,2)), axis=1)
df["latround"] = df.latitude.apply(lambda x: round(x,2))
df["longround"] = df.longitude.apply(lambda x: round(x,2))
areq5 = df[df.coord.duplicated(keep=False)].sort_values(by="coord").iloc[:,[-3,-2,-1]]
elemlist = areq5.groupby("coord").coord.count().sort_values()
print(f"El maximo numero de elementos en una lista {elemlist.max()} y el minimo {elemlist.min()}")
print(f"Elementos en promedio {elemlist.mean()}")

nodsArbolLong = df.groupby(["latround"]).coord.count()
print(f"El numero maximo de nodos en un arbols de longitud es {nodsArbolLong.max()} y el minimo {nodsArbolLong.min()}")
print(f"En promedio el numero de nodos de los arboles de longitud es {nodsArbolLong.mean()}")
```

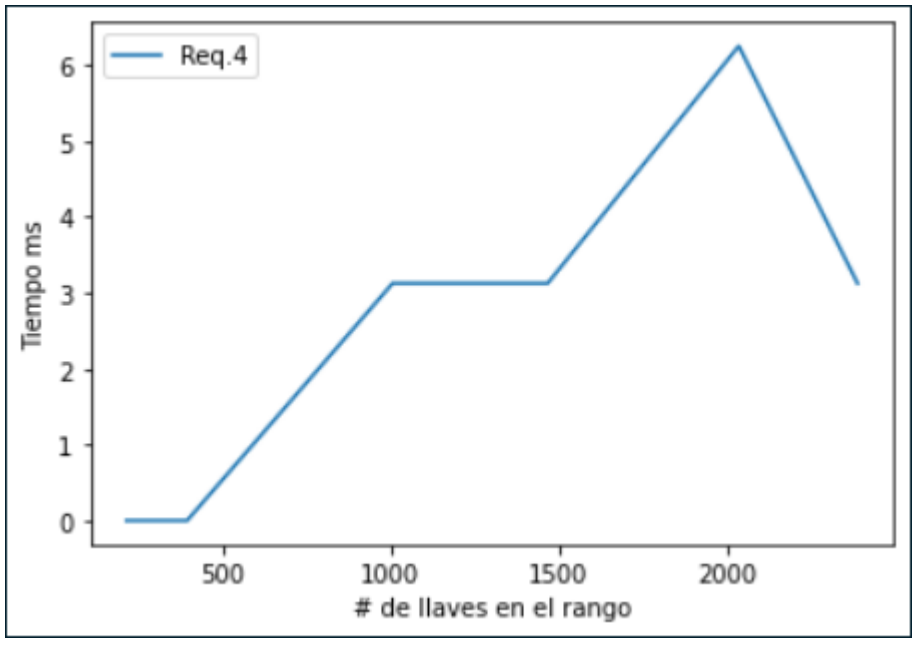
Requerimiento 6:

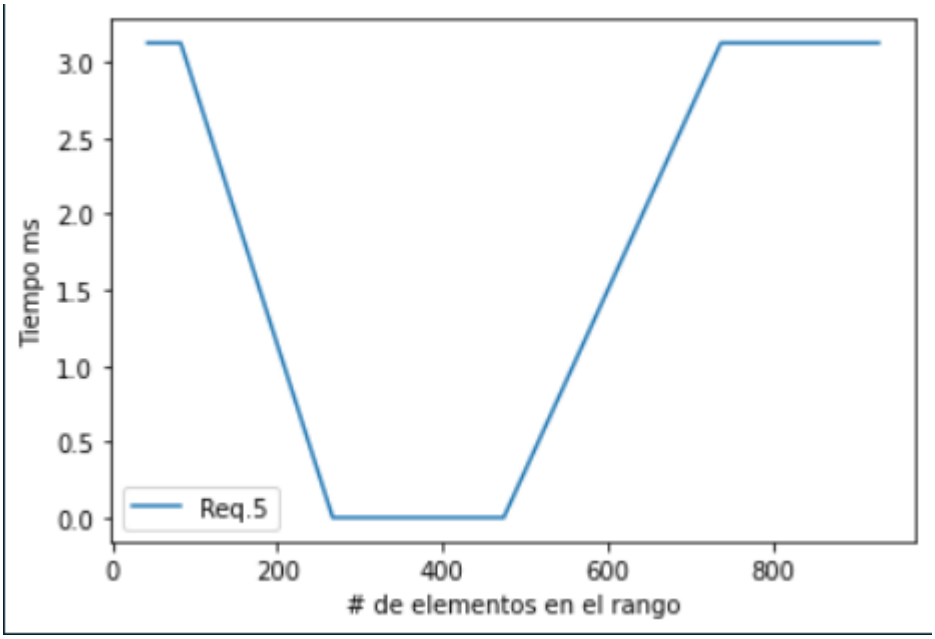
Para el requerimiento 6 se reutilizó el código del requerimiento 5 por lo cual la complejidad será de $O(k \cdot \log(c) + k \cdot e \cdot p)$ lo cual según el análisis será $O(k \cdot m)$, no obstante algo importante a resaltar es que debido a que se esta utilizando folium para crear el mapa, esto agregar una constante al tiempo de ejecución y dado que no se conoce la implementación no se puede decir cómo afecta pero basado en las pruebas de tiempo es un incremento significativo pues los tiempos pasaron de estar entre 0 y 3 ms a entre 15 y 25 ms, esto nos da a entender que plantear los puntos toma un tiempo significativo y como podemos ver es constante ya que solo se tienen que agregar a lo sumo 10 valores al mapa.

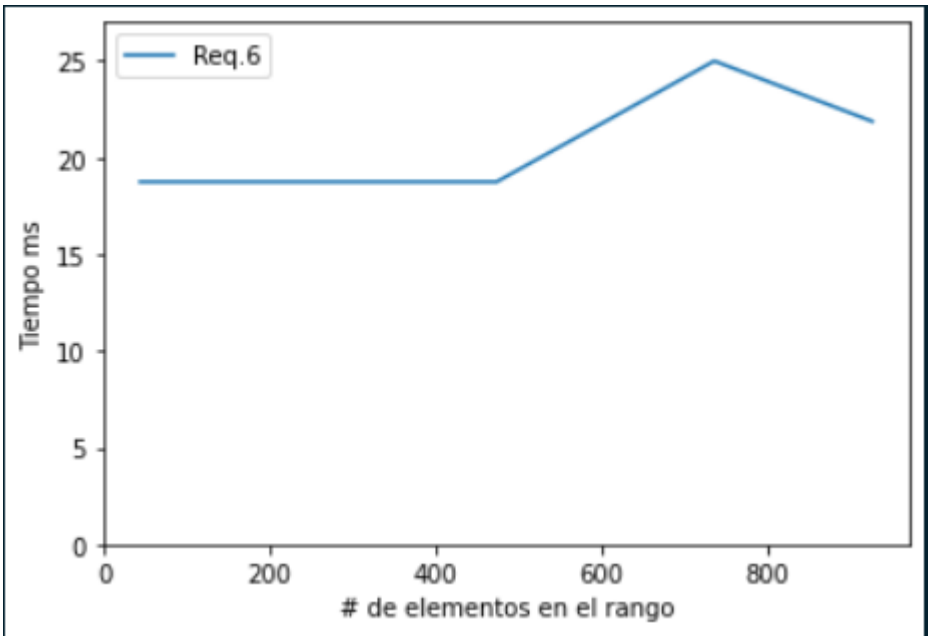
Gráficos:




Req.2
0.04
0.02
0.00
−0.02
−0.04
Tiempo ms
20
30
40
50
60
70
80
de llaves en el rango


Req.4
6
5
4
3
2
1
0
Tiempo ms
500
1000
1500
2000
de llaves en el rango





Nota: Dado que las funciones corren tan rápido en algunos casos el análisis grafico no es muy productivo puesto que son casi 0 ms o 0ms y los cambios que se dan (3ms) pueden deberse al estado del computador en el momento de las pruebas.