Isai Daniel Chacón Silva – 201912015 - Correo: id.chacon@uniandes.edu.co

Nicolás Aparicio Claros – 201911357 – Correo: n.aparicioc@uniandes.edu.co

Estructuras de datos y algoritmos

Reto 3

En este documento se realiza el análisis de complejidad para cada uno de los algoritmos implementados para resolver cada requerimiento del reto 3 de la materia Estructuras de datos y algoritmos. En primer lugar, vale recalcar que el análisis teórico se realizó con la notación Big O, el cual es el peor caso posible para cada uno de los algoritmos realizados, sin embargo, este no ocurre siempre en la práctica y varía dependiendo en la distribución de las muestras de datos.

La Tabla 1 muestra las especificaciones de hardware y software de la máquina en la cual se corrieron las pruebas de rendimiento respectivas, aquí presentadas.

Tabla 1. Especificaciones de la máquina para ejecutar las pruebas de rendimiento.

Características	**Máquina de Prueba 1**
Procesadores	Intel ® core TM i5-8250U CPU @1,60GHz
Memoria RAM /GB)	8 GB
Sistema Operativo	Microsoft Windows 10 Pro basado en x64

Análisis Teórico

Requerimiento 1

La complejidad de este algoritmo viene dada por 4 funciones que se llevan a cabo para cumplir con toda la información que se busca imprimir. La primera de estas es mostrar el tamaño del árbol almacenado en el catálogo de ciudades, cuya complejidad es O(1) ya que se conoce a priori el tamaño de este.

Para encontrar la ciudad con más avistamientos de ovnis, la función *largestCity* recorre todo el catálogo de ciudades y obtiene sus avistamientos, actualizando así el máximo. Dado que no fue posible establecer una noción de orden ya que la llave era el nombre de la ciudad, de modo que se utilizaba el *Unicode code,* el cual no permitía indexar solo una sección del árbol en relación con su total de avistamientos. Por lo tanto, esta función es $O(n)$, para n el tamaño total de avistamientos.

Finalmente, para contar el número de avistamientos para una ciudad dada por el usuario, se utiliza la función *cities,* la cual obtiene directamente el valor del mapa de ciudades con la llave directamente. Obtener esta lista de avistamientos es $O(1)$ gracias a las propiedades de los mapas. Con el fin de imprimirlas en orden cronológico, se utiliza el algoritmo *merge.sort*

sobre la sublista encontrada. En teoría, el peor caso sería si todos los datos provenientes del *.csv* provinieran de una misma ciudad (que no es el caso en la práctica), ya que esto incurriría en O(nlog_n). Una vez ordenado, se obtienen las sublistas correspondientes a las 3 primeras, y 3 últimas posiciones y se imprime su información al usuario O(1).

De modo que este requerimiento, en total, establecería una complejidad de O(nlog_n) aproximadamente por ser el término más significativo de todas las operaciones descritas.

Tabla 2. Resultado del tiempo de ejecución para el requerimiento 1, cambiando el tamaño del conjunto de datos.

Porcentaje de la muestra [pct]	Tamaño de la muestra (ARRAYLIST)	Merge Sort [ms]	
10,00%	8033,00	47,00	**REQUERIMIENTO 1**
20,00%	16066,00	50,00	
30,00%	23999,00	48,00	
50,00%	40166,00	48,00	
80,00%	64265,00	83,00	
100,00%	80332,00	89,00	

Requerimiento 2 – implementado por Nicolás Aparicio

Para este requerimiento, se obtiene el tamaño del árbol mediante la función *omap.size*¸que es O(1). Por su parte, el contador de diferentes elementos de duración llama a la función *longestDurationSeconds*, el cual hace uso de la función *maxKey* sobre el mapa de catálogo de duraciones en segundos. Al tener un árbol ordenado y balanceado, esta función es O(nlog_n).

A continuación, se corre la función *seconds_range* que saca los valores de las sublistas del mapa en el rango ingresado por el usuario, Posteriormente, se recorren los valores de las sublistas, aplicando un algoritmo de ordenamiento *merge.sort* que es O(nlog_n), en el peor caso teórico para el cual se cargan todos los datos, es decir, la duración ingresada por parámetros sería (0, max(duration)) de los datos de los avistamientos de ovnis. Finalmente, se imprimen los valores requeridos por el usuario en consola.

En síntesis, el orden de este requerimiento es O(nlog_n) dadas las tareas previamente descritas.

Tabla 3. Resultado del tiempo de ejecución para el requerimiento 2, cambiando el tamaño del conjunto de datos.

Porcentaje de la muestra [pct]	Tamaño de la muestra (ARRAYLIST)	Merge Sort [ms]	
10,00%	8033,00	50,00	**REQUERIMIENTO 2**
20,00%	16066,00	52,00	
30,00%	23999,00	48,00	

50,00%	40166,00	63,00	
80,00%	64265,00	78,00	
100,00%	80332,00	98,00	

Requerimiento 3 – implementado por Daniel Chacón

De manera análoga a los 2 requerimientos anteriores, encontrar el tamaño del árbol del catálogo de fechas por hora, toma un tiempo constante O(1). Dado que se tiene una noción de orden, dada por la fecha del avistamiento en formato datetime, es posible encontrar la fecha más antigua así como el tamaño de la lista en la fecha dada por parámetro en O(log_n) mediante la función *maxKey* y *lt.size*, teniendo en cuenta que el árbol se encuentra tanto ordenado, como balanceado.

Ahora bien, para contar el número de avistamientos en el rango dado, e imprimir la tabla de la información al usuario se utiliza la función *dates_rangeByHour*, la cual saca la lista de valores en el rango dado mediante *omap.values*. Esta lista de sublistas es recorrida y ordenada mediante *merge.sort*. Así como se analizó previamente, el peor caso sería en teoría si el rango de horario ingresado por el usuario fuera de 0:00 a 23:59, ya que se deberían incluir todos los elementos del árbol en las sublistas. Esto incurriría en O(nlog_n) con n el número total de avistamientos. Una vez ordenada cada sublista, se obtienen las sublistas correspondientes a las 3 primeras, y 3 últimas posiciones y se imprime su información al usuario O(1). Para poder recorrer estas sublistas de forma más eficiente, se realizaron 2 funciones aparte *getElementsk*, que recibe una bandera para indicar si se buscan los primeros 3 o últimos 3 elementos, para k=3.

De modo que, en total, el orden de este algoritmo sería de O(nlog_n) para todos los procedimientos llevados a cabo.

Tabla 4. Resultado del tiempo de ejecución para el requerimiento 3, cambiando el tamaño del conjunto de datos.

Porcentaje de la muestra [pct]	Tamaño de la muestra (ARRAYLIST)	Merge Sort [ms]	
10,00%	8033,00	96,00	
20,00%	16066,00	1097,00	
30,00%	23999,00	1035,00	**REQUERIMIENTO 3**
50,00%	40166,00	5010,00	
80,00%	64265,00	9015,00	
100,00%	80332,00	13630,00	

Requerimiento 4

El requerimiento 4 es análogo al 3. En primer lugar se obtiene el tamaño del árbol del catálogo de fechas, lo cual toma un tiempo constante O(1). A su vez, ya que se tiene una noción de

orden, dada por la fecha del avistamiento en formato datetime, es posible encontrar la fecha más antigua así como el tamaño de la lista en la fecha dada por parámetro en O(log_n) mediante la función *minKey* y *lt.size*, teniendo en cuenta que el árbol se encuentra tanto ordenado, como balanceado.

Luego, para contar el número de avistamientos en el rango dado, e imprimir la tabla de la información al usuario se utiliza la función *dates_range*, la cual saca la lista de valores en el rango dado mediante *omap.values*. Esta lista de sublistas ya se encuentran ordenadas debido a que se utiliza un mapa ordenado y balanceado mediante el mismo parámetro ingresado por el usuario. Así, se obtienen las sublistas correspondientes a las 3 primeras, y 3 últimas posiciones y se imprime su información al usuario O(1). Para poder recorrer estas sublistas de forma más eficiente, se realizaron 2 funciones aparte *getElementsk*.

De modo que, en conclusión, se tiene una complejidad total de O(log_n) para el requerimiento 4.

Tabla 5. Resultado del tiempo de ejecución para el requerimiento 4, cambiando el tamaño del conjunto de datos.

Porcentaje de la muestra [pct]	Tamaño de la muestra (ARRAYLIST)	Merge Sort [ms]	
10,00%	8033,00	25,00	**REQUERIMIENTO 4**
20,00%	16066,00	35,00	
30,00%	23999,00	38,00	
50,00%	40166,00	45,00	
80,00%	64265,00	43,00	
100,00%	80332,00	45,00	

Requerimiento 5

Para el último requerimiento se utilizó una estructura de datos tipo mapa ordenado para representar los avistamientos ordenados por la coordenada longitud (aproximada a dos cifras decimales). Al interior de cada coordenada longitud se representaron los avistamientos que tengan dicha longitud, mediante otro mapo, ordenado y balanceado, por las coordenadas de la latitud del avistamiento.

En este orden de ideas, la función *coordinates* se encarga de obtener una lista con los *values* en el rango de longitud dada (*omap.values)*. Estas sublistas de mapas son recorridas mediante nuevamente una sublista de coordenadas *omap.values* por cada mapa, pero esta vez, sobre el rango de latitud esperado. De modo que, teóricamente, el peor caso vendría dado por el mayor rango de longitudes y latitudes posible, i.e., se recorrerá toda la lista de entradas de avistamientos, O(n), de modo que incremente el contador de avistamientos en el rango conjunto y se añadan a una nueva lista, sobre la cual se obtienen los primeros 5 y últimos 5 elementos para imprimir la información al usuario en view. En conclusión, este algoritmo presenta un orden de O(n).

Tabla 6. Resultado del tiempo de ejecución para el requerimiento 5, cambiando el tamaño del conjunto de datos.

Porcentaje de la muestra [pct]	**T**amaño de la muestra (ARRAYLIST)	**Merge Sort [ms]**	
10,00%	8033,00	40,00	**REQUERIMIENTO 5**
20,00%	16066,00	40,00	
30,00%	23999,00	38,00	
50,00%	40166,00	41,00	
80,00%	64265,00	43,00	
100,00%	80332,00	41,00	

Conclusiones

La conclusión más importante a la que se llegó es que el algoritmo más eficiente fue aquel implementado mediante un mapa, cuyos *values* fueran mapas a su vez (requerimiento 5), ver Figura. 1. A pesar de haber concluido que sería O(n), en la práctica, así como se observó en los experimentos llevados a cabo, su tiempo varió mínimamente, ante cambios drásticos en la cantidad de datos que se cargaban, acercándose incluso a tiempos constantes.

Por su parte, los demás algoritmos, presentaron también aumentos en el tiempo un poco mayores en comparación con el requerimiento 5. Esto va de acuerdo parcialmente con la teoría presentada, donde se mostró que su crecimiento sería linearítmico, debido a los ordenamientos realizados sobre las sublistas. Sin embargo, estos crecimientos en complejidad fueron mínimos en la práctica, lo que realza la importancia de usar mapas ordenados y balanceados. Vale aclarar también, que la carga de datos fue el procedimiento en el que más tiempo se gastó, debido a que, antes grandes volúmenes de datos, el árbol debe estar balanceándose todo el tiempo, gracias a la implementación de funciones como rotar a la izquierda o a la derecha. Dado que sus entradas se podrían considerar aleatorias, las búsquedas de elementos llegan a ser 1.39 log_n, lo que concuerda más con lo encontrado en la práctica, que en comparación con el peor caso linearítmico (Figura. 2) de nlog_n.

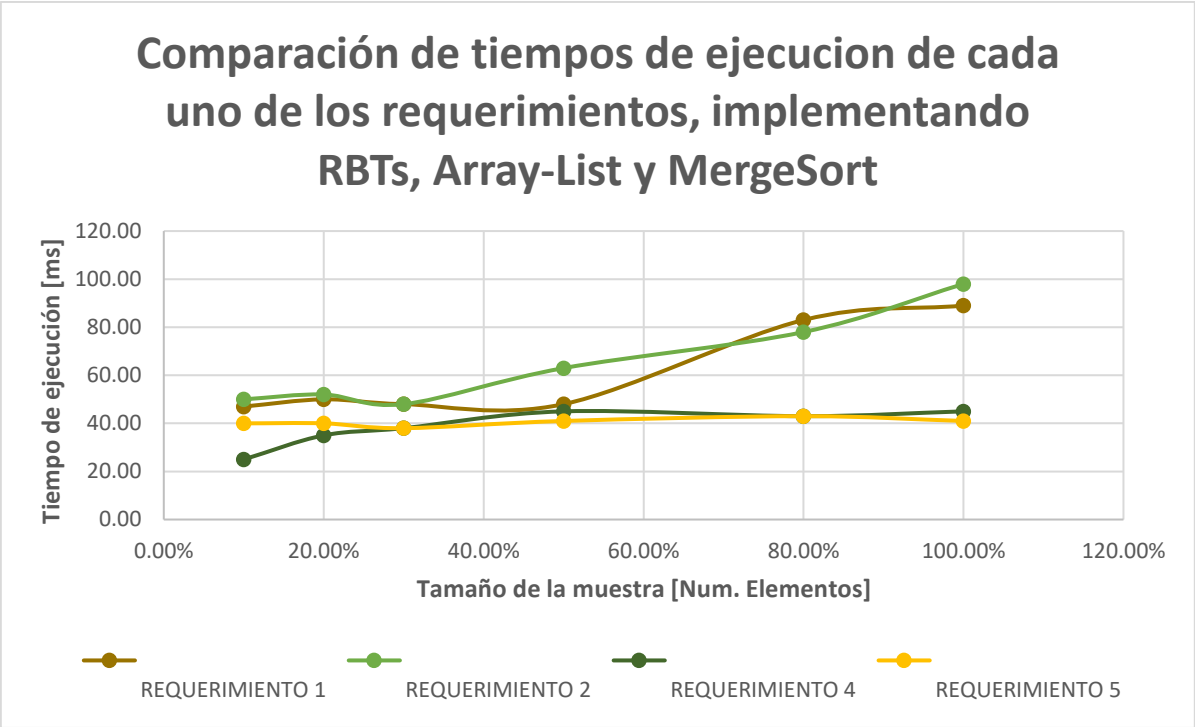

Figura 1. Grafica que muestra el comportamiento del tiempo de ejecución y el tamaño de la muestra para los requerimientos 1, 2, 4 y 5.

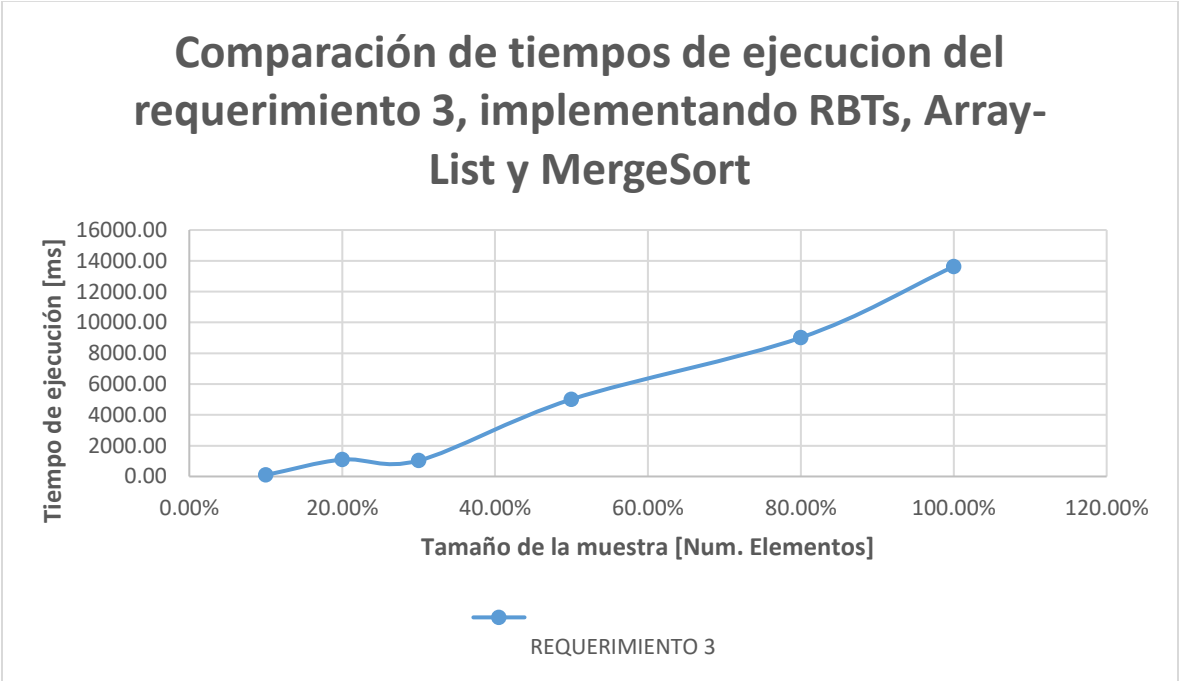

Figura 2. Grafica que muestra el comportamiento del tiempo de ejecución y el tamaño de la muestra para el requerimiento 3.