ANÁLISIS DEL RETO

Estudiante 1, código 1, email 1

Santiago Garzon Garcia, 202512373, S.garzong2@uniandes.edu.co

Santiago Ismael Escobar Maidana, 202516956, si.escobar@uniandes.edu.co

Requerimiento <<n>>

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

Descripción

Breve descripción de como abordaron la implementación del requerimiento

Entrada	Parámetros necesarios para resolver el requerimiento.
Salidas	Respuesta esperada del algoritmo.
Implementado (Sí/No)	Si se implementó y quien lo hizo.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Paso 1	O(...)
Paso 2	O(...)
Paso ….	O(...)
TOTAL	***O(...)***

Análisis

Análisis de resultados de la implementación, tener cuenta las pruebas realizadas y el analisis de complejidad.

Requerimiento Ejemplo

Descripción

```python
def get_data(data_structs, id):
    """
    Retorna un dato a partir de su ID
    """
    pos_data = lt.isPresent(data_structs["data"], id)
    if pos_data > 0:
        data = lt.getElement(data_structs["data"], pos_data)
        return data
    return None
```

Este requerimiento se encarga de retornar un dato de una lista dado su ID. Lo primero que hace es verificar si el elemento existe. Dado el caso que exista, retorna su posición, lo busca en la lista y lo retorna. De lo contrario, retorna None.

Entrada	Estructuras de datos del modelo, ID.
Salidas	El elemento con el ID dado, si no existe se retorna None
Implementado (Sí/No)	Si. Implementado por Juan Andrés Ariza

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Buscar si el elemento existe (isPresent)	O(n)
Obtener el elemento (getElement)	O(1)
TOTAL	***O(n)***

Análisis

A pesar de que obtener un elemento en un *ArrayList,* dada su posición, tiene complejidad constante, la implementación de este requerimiento tiene un orden lineal O(n). Esto debido a que, lo primero que se hace es verificar si el elemento hace parte de la lista. Específicamente, a la hora de buscar un elemento en una lista, en el peor de los casos es necesario recorrer toda la lista, es decir, complejidad lineal.

Requerimiento <<1>>

Descripción

Lo que se hace es que se recorre la lista de trayectos del catálogo y se filtran únicamente aquellos cuya hora de recogida (pickup_ts) esté dentro del rango de tiempo especificado por los parámetros start_dt_str y end_dt_str. Una vez filtrados, los trayectos se ordenan de forma ascendente según el timestamp de recogida, de manera que los más antiguos aparezcan primero. Finalmente, se toman los primeros N y los últimos N trayectos del conjunto filtrado; si la cantidad total de resultados es menor o igual a 2N, se retornan todos una sola vez.

Entrada	Los parámetros de entrada son: Catalog: Catálogo con la lista de trayectos catalog["trips"] start_dt_str: timestamp inicial del rango (formato "YYYY-MM-DD HH:MM:SS") end_dt_str: timestamp final del rango sample_n: cantidad de elementos a mostrar al inicio y final del conjunto filtrado
Salidas	La función retorna un diccionario con los siguientes elementos: tiempo_ms: tiempo total de ejecución del requerimiento total_filtrados: Cantidad total de trayectos que cumplen con el filtro de tiempo primeros: Lista con los primeros N trayectos tras ordenar ultimos : lista con los últimos N trayectos tras ordenar (o igual a primeros si total_filtrados ≤ 2N) first_n: Lista auxiliar con los primeros N elementos en formato Python nativo last_n: Lista auxiliar con los últimos N elementos en formato Python nativo total_trips: Alias de total_filtrados, incluido para compatibilidad elapsed_ms; Alias del tiempo de ejecución
Implementado (Sí/No)	Si, lo implemento Mateo Sánchez

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Paso 1	Conversión de fechas a timestamp (start_ts, end_ts) O(1)
Paso 2	Recorrido de la lista de viajes para filtrar por rango de tiempo (N viajes) O(N)
Paso3	Ordenamiento del conjunto filtrado con merge_sort (M elementos filtrados) O(M log M)
Paso 4	Construcción de la lista de los primeros N elementos O(N)
Paso 5	Construcción de la lista de los últimos N elementos O(N)
Paso 6	Conversión a listas Python (first_py, last_py) O(N)
Paso 7	Construcción del retorno final O(1)
TOTAL	***Complejidad temporal dominante:*** ***O(M log M)*** ***Complejidad espacial dominante:*** ***O(M)***

Análisis

El rendimiento del requerimiento 1 está dominado por el filtrado lineal sobre todos los viajes en O(N) y por el ordenamiento del subconjunto filtrado en O(M log M). Si el rango de tiempo es amplio y M se aproxima a N, la complejidad total se acerca a O(N log N), mientras que si el rango es más restrictivo y M ≪ N, el tiempo real percibido se aproxima a O(N). El uso de merge_sort garantiza estabilidad en el ordenamiento y un consumo de memoria proporcional a O(M), lo cual es aceptable dado el volumen de datos manejado.

Requerimiento <<2>>

Descripción

Lo que se hace es que se recorre el catálogo de trips para filtrar lo que estén dentro del rango de latitudes que se da como parámetro. Después, los filtrados se ordenan de mayor a menor latitud y en caso de que haya un empate se ordenan por longitud de recogida. Por último, se toman los primeros N y los últimos N de los filtrados, o en el caso de que sean menos de 2N se toman todos.

Entrada	Los parámetros de entrada son: ● catalog: Catálogo con la lista de trayectos catalog[“trips”] ● lat_min: La latitud mínima del rango. ● lat_max: La latitud máxima del rango ● N: cantidad de elementos a mostrar al inicio y al final.
Salidas	El retorno de la función es un diccionario con los siguientes elementos: ● tiempo_ms: Tiempo de ejecución del requerimiento ● total_filtrados: Cantidad de trayectos que cumplen con el filtro de latitudes. ● primeros: los primeros N trayectos en el filtro ● ultimos: los ultimos N trayectos en el filtro (si total_filtrados es menor a 2N entonces es igual a primeros)
Implementado (Sí/No)	Si, Santiago Escobar

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Paso 1	Inicialización de variables y toma de tiempo. Complejidad: O(1)
Paso 2	Recorrido de la lista de viajes para filtrar. Complejidad O(n)

Paso 3	Ordenamiento de la lista filtrada con merge_sort. m es el numero de los elementos filtrados (menor a n). Complejidad O(m logm)
Paso 4	Construcción de lista de primeros N elementos. O(N)
Paso 5	Construcción de lista de últimos N elementos. O(N)
Paso 6	Construcción de la respuesta final. O(1)
TOTAL	***El total sería O(n)+O(mlogm)+O(N)*** ***Como N < m y m <= n*** ***La complejidad temporal dominante es:*** ***O(m logm)*** ***La complejidad espacial dominante (como se usa merge_sort y como filtrados puede llegar a ser n) es:*** ***O(n)***

Análisis

El rendimiento del requerimiento 2 está dominado por el filtrado lineal de todos los viajes en O(n) y el ordenamiento de los elementos filtrados en O(mlogm) .Si m tiende a n, la complejidad se aproxima a O(nlog n); si m < n, predomina el filtrado y el tiempo real se acerca a O(n). El uso de Merge Sort asegura eficiencia estable y un consumo de memoria adicional de O(m).

Requerimiento <<3>>

Descripción

Filtrar los viajes (trayectos) cuya distancia de viaje (trip_distance) se encuentra dentro de un rango específico [distancia_min, distancia_max], ordenar los resultados en orden descendente por trip_distance y, en caso de empate, por total_amount también en orden descendente. Finalmente, retornar información resumida del tiempo de ejecución, el total de viajes filtrados, y una muestra de los N primeros y N últimos viajes (si el total es menor o igual a 2N, se retornan todos).

Entrada	catalog: Estructura de datos que contiene todos los viajes distancia_min: Límite inferior (incluido) para la distancia del viaje (trip_distance). distancia_max: Límite superior (incluido) para la distancia del viaje (trip_distance).

	n_muestra: Número de elementos a mostrar al principio y al final de la lista de resultados
Salidas	tiempo_ms: Tiempo total de ejecución del algoritmo en milisegundos. total: Número total de viajes que cumplieron con el criterio de filtrado primeros: Lista con los viajes con mayor distancia (y mayor costo en caso de empate). ultimos: Lista con los viajes con menor distancia (y menor costo en caso de empate) de la lista filtrada
Implementado (Sí/No)	Si se implementó , implementado por Santiago Garzon

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Paso 1	Inicialización O(1)
Paso 2	Filtración de los viajes , iterando sobre todos los viajes O(N)
Paso 3	Ordenamiento de los viajes usando merge sort O(M log M)
paso 4	Cálculo del total O(1)
paso 5	Armar diccionario de salida O(M)
paso 6	Retornar resultados O(1)
TOTAL	***O(N + M log M)***

Análisis

El cuello de botella del algoritmo es la fase de ordenamiento O(M log M) de los viajes filtrados. La complejidad total O(N + M log M) está dominada por el filtrado inicial O(N) y el Merge Sort. Si el filtro es poco restrictivo (M es cercano a N), la complejidad se aproxima a O(N log N). El uso de Merge Sort asegura una complejidad O(M log M) en el peor de los casos para el ordenamiento.

Requerimiento <<4>>

Descripción

Filtrar los viajes que terminaron en una fecha específica (fecha_yyyy_mm_dd) y cuya hora de terminación (dropoff_time) cumpla con el criterio de momento_interes ("ANTES" o "DESPUES") de una hora de referencia. El resultado final debe estar ordenado por dropoff_datetime en orden descendente

(más reciente primero). Se retorna información resumida del tiempo de ejecución, el total de viajes filtrados, y una muestra de los N primeros y N últimos viajes.

Entrada	catalog: Estructura de datos de los viajes. fecha_yyyy_mm_dd: Fecha de terminación (YYYY-MM-DD) para filtrar. momento_interes: Criterio de filtrado por hora, puede ser "ANTES" o "DESPUES". hora_referencia_hms: Hora de referencia (HH:MM:SS) para el filtro de tiempo. n_muestra: Número de elementos a mostrar en la muestra
Salidas	tiempo_ms: Tiempo total de ejecución del algoritmo en milisegundos. total: Número total de viajes que cumplieron con los criterios de filtrado primeros: Lista con los viajes más recientes. ultimos: Lista con los viajes menos recientes.
Implementado (Sí/No)	Si se implementó , implementado por todos

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Paso 1	Inicialización O(1)
Paso 2	iteración sobre los N viajes O(N)
Paso 3	Búsqueda rapida O(1)
Paso 4	Filtración por hora O(D)
Paso 5	Ordenamiento de los viajes usando merge sort O(M log M)
Paso 6	Armar diccionario de salida O(M)
Paso 7	Retornar resultados O(1)
TOTAL	***O(N + M log M)***

Análisis

La fase de indexación Hash tiene una complejidad O(N) promedio, que domina el inicio de la ejecución. La consulta específica (Paso 2) se beneficia de la estructura Hash, costando solo O(D) para iterar sobre los viajes D del día de interés. El cuello de botella posterior es el ordenamiento O(M log M) de los M candidatos finales. La eficiencia del algoritmo para búsquedas repetidas se incrementa, ya que, si el índice se construye una sola vez, las consultas subsecuentes solo costarían O(D + M log M), lo que es mucho más rápido que O(N log N) si el catálogo es grande (N) y el número de viajes por día (D) y los filtrados (M) son relativamente pequeños.

Requerimiento <<5>>

Descripción

Lo que se hace es que primero se construye un índice hash donde los viajes del catálogo se agrupan según la hora de finalización del servicio. Luego, se busca directamente el bucket correspondiente a la hora dada como parámetro. Si no existen viajes en ese rango horario, se retorna una respuesta vacía. En caso de que sí existan, esos viajes se ordenan de forma descendente por el timestamp de finalización, de manera que los más recientes queden primero en la lista. Finalmente, se toman los primeros N y los últimos N viajes del bucket, o si hay menos de 2N elementos se retornan todos una sola vez.

Entrada	Los parámetros de entrada son: Catalog: Catálogo con la lista de trayectos y la estructura de índices term_dt_hour_str: Hora de terminación del servicio que se desea consultar (en formato de hora) sample_n: Cantidad de elementos a mostrar al inicio y al final del bucket
Salidas	El retorno de la función es un diccionario con los siguientes elementos: tiempo_ms: Tiempo total de ejecución del requerimiento total_filtrados: Cantidad de trayectos que corresponden a la hora de terminación consultada primeros: Lista con los primeros N trayectos ordenados por hora de finalización descendente ultimos: Lista con los últimos N trayectos del mismo conjunto (si total_filtrados ≤ 2N, entonces ultimos es igual a primeros) first_n: Lista auxiliar de los primeros N elementos en formato Python nativo last_n: Lista auxiliar de los últimos N elementos en formato Python nativo total_trips: Número total de viajes dentro del bucket para la hora consultada elapsed_ms: Alias del tiempo de ejecución, duplicado en la estructura por compatibilidad
Implementado (Sí/No)	Si , fue implementada por todos.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Paso 1	Inicialización de variables y toma de tiempo O(1)

Paso 2	Construcción del índice agrupando los viajes por hora de finalización O(N)
Paso 3	Búsqueda del bucket en el mapa hash. O(1) promedio
Paso 4	Ordenamiento del bucket filtrado con merge_sort (M elementos) O(M log M)
Paso 5	Construcción de la lista de los primeros N elementos O(N)
Paso 6	Construcción de la lista de los últimos N elementos O(N)
Paso 7	Conversión a listas Python (first_py, last_py) O(N)
Paso 8	Construcción del retorno final O(1)
TOTAL	***O(N) + O(M log M)*** ***Complejidad Temporal Dominante:*** ***O(M log M)*** ***Complejidad Espacial Dominante:*** ***O(M)***

Análisis

El rendimiento del requerimiento 5 está determinado por la construcción del índice hash en O(N) y el ordenamiento del bucket filtrado en O(M log M). Si la hora especificada contiene muchos viajes (M grande), el ordenamiento se convierte en el paso dominante; en cambio, si el bucket tiene pocos elementos, el costo total se aproxima a O(N) debido a la etapa de indexación inicial. La estructura hash permite acceso directo al bucket en tiempo constante promedio, optimizando la búsqueda. En memoria, el índice y el bucket implican O(N) espacio, mientras que el ordenamiento introduce un uso adicional de O(M) debido a las estructuras auxiliares de merge_sort.

Requerimiento <<6>>

Descripción

Se construye un índice hash para agrupar los viajes por barrio. Para cada trayecto, se calcula la distancia Haversine entre su punto de recogida y cada uno de los barrios en el catálogo para asignarlo al barrio más cercano. Luego, se toma el bucket correspondiente al barrio buscado y se filtran únicamente los viajes cuya hora de recogida esté dentro del rango dado, considerando el caso especial donde el rango cruza la medianoche. Finalmente, se ordenan los viajes filtrados por tiempo de recogida ascendente (más antiguo primero) y se retornan los primeros N y últimos N, o todos si son menos de 2N.

Entrada	Los parámetros de entrada de la función son: ● catalog: Catálogo con la lista de trayectos y la lista de barrios. ● barrio_buscar: Nombre del barrio objetivo.

	● hora_inicio_str: Hora inicial del rango. ● hora_fin_str: Hora final del rango. ● n_muestra: Cantidad de trayectos a mostrar al inicio y final.
Salidas	La función retorna un diccionario con: ● tiempo_ms: Tiempo total de ejecución. ● total: Número de trayectos que cumplen con el filtro. ● primeros: Lista con los primeros N trayectos ordenados por tiempo. ● ultimos: Lista con los últimos N trayectos ordenados por tiempo (o igual a primeros si total ≤ 2N).
Implementado (Sí/No)	Si, lo implementaron Mateo Sanchez, Santiago Escobar y Santiago Garzón.

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Paso 1	Normalización de parámetros y creación del índice hash de barrios. O(b)
Paso 2	Asignación de cada viaje al barrio más cercano usando Haversine (b checks por viaje). O(b*t)
Paso 3	Filtrado por rango de horas sobre el bucket del barrio. O(m)
Paso 4	Ordenamiento con merge_sort de los candidatos. O(mlogm)
Paso 5	Construcción de listas de salida primeros y ultimos. O(n_muestra)
Paso 6	Construcción del retorno final. O(1)
TOTAL	***O(t * b) + O(mlogm)*** ***Complejidad temporal en peor caso: O(mlogm)*** ***Complejidad espacial en peor caso: Total espacial dominante: O(t)***

Análisis

El tamaño del índice hash de barrios se representa como b.

El tamaño de los buckets con trayectos agrupados se representan como t.

El tamaño de la lista de candidatos filtrados se representa como m.

El algoritmo está dominado por la fase de asignación de viajes a barrios, con complejidad O(t*b) debido al cálculo repetido de Haversine para cada viaje contra cada barrio. Si el número de barrios es reducido y

constante, este costo temporal se aproxima a O(t). Una vez agrupados los viajes, el ordenamiento de los candidatos en el barrio específico tiene costo O(mlogm), que se vuelve dominante temporalmente solo si el bucket de ese barrio contiene una gran cantidad de trayectos. El uso de estructuras auxiliares en merge_sort introduce un consumo de memoria proporcional a O(m), pero el almacenamiento de los buckets y el índice hash hace que el uso espacial total esté dominado por O(t).