ANÁLISIS DEL RETO

Bastien Quentin Clement Thirion, 202525085, b.thirion@uniandes.edu.co

Jonathan David Galeano Sosa, 202226332, j.galeanos@uniandes.edu.co

Requerimiento 1

```python
def req_1(catalog,num_pasajeros):
    inicio = time.time()

    trayectos_filtrados = [
        t for t in catalog.get("taxis", [])
        if int(t["passenger_count"]) == num_pasajeros
    ]

    inicio = time.time()

    suma_duracion = suma_costo = suma_distancia = suma_peajes = suma_propina = 0
    total_trayectos = len(trayectos_filtrados)

    freq_pagos = {}
    freq_fechas = {}
    max_pago = None
    max_pago_count = 0
    max_fecha = None
    max_fecha_count = 0

    for t in trayectos_filtrados:
        inicio_tray = datetime.strptime(t["pickup_datetime"], "%Y-%m-%d %H:%M:%S")
        fin_tray = datetime.strptime(t["dropoff_datetime"], "%Y-%m-%d %H:%M:%S")
        duracion_min = (fin_tray - inicio_tray).total_seconds() / 60

        suma_duracion += duracion_min
        suma_costo += float(t["total_amount"])
        suma_distancia += float(t["trip_distance"])
        suma_peajes += float(t["tolls_amount"])
        suma_propina += float(t["tip_amount"])

        # Contar fecha
        fecha = inicio_tray.strftime("%Y-%m-%d")
        if fecha not in freq_fechas:
            freq_fechas[fecha] = 0
        freq_fechas[fecha] += 1
        if freq_fechas[fecha] > max_fecha_count:
            max_fecha = fecha
            max_fecha_count = freq_fechas[fecha]

        # Contar tipo de pago
        pago = t["payment_type"]
        if pago not in freq_pagos:
            freq_pagos[pago] = 0
        freq_pagos[pago] += 1
        if freq_pagos[pago] > max_pago_count:
            max_pago = pago
            max_pago_count = freq_pagos[pago]

    tiempo_promedio = suma_duracion / total_trayectos
    costo_promedio = suma_costo / total_trayectos
    distancia_promedio = suma_distancia / total_trayectos
    peaje_promedio = suma_peajes / total_trayectos
    propina_promedio = suma_propina / total_trayectos

    fin = time.time()
    tiempo_ejecucion_ms = (fin - inicio) * 1000

    return {
        "tiempo_ejecucion_ms": tiempo_ejecucion_ms,
        "total_trayectos": total_trayectos,
        "tiempo_promedio_min": tiempo_promedio,
        "costo_total_promedio": costo_promedio,
        "distancia_promedio_millas": distancia_promedio,
        "peaje_promedio": peaje_promedio,
```

Descripción

Este requerimiento se encarga de filtrar los trayectos del catálogo según el número de pasajeros. Una vez filtrados, calcula estadísticas agregadas: duración promedio, costo promedio, distancia promedio, peajes y propinas promedio. Además, identifica el tipo de pago más usado y la fecha de inicio más frecuente de los trayectos.

Entrada	Catalog : Catalogo con los trayetos cargados num_pasajeros : numero de pasajeros a filtrar
Salidas	Tiempo de ejecucion Total de trayectos con ese número de pasajeros. Promedio de duración, costo, distancia, peajes y propinas. Nombre y cantidad del tipo de pago más usado. Fecha más frecuente de inicio de trayectos.
Implementado (Sí/No)	Si, Bastien Thirion

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Filtrar trayectos por número de pasajeros	O(n)
Recorrer todos los trayectos filtrados	O(m) con m<n
Para cada trayecto: convertir fechas, calcular duración y acumular costos/distancias/peajes/propinas, frecuencias de tipo de pago y de fechas	O(1)
Calcular promedias y retornar resultados	O(1)
TOTAL	***O(n)***

Análisis

La implementación itera una sola vez sobre los trayectos (n elementos) y una sola vez sobre los trayectos filtrados, y en cada iteración todas las operaciones realizadas (suma de valores, conversión de fechas, actualización de diccionarios) tienen un costo constante **O(1)**.

Por lo tanto, el algoritmo tiene un comportamiento lineal **O(n)**, que se confirma experimentalmente: con catálogos grandes (500 000 trayectos), los tiempos de ejecución están en el orden de segundos, lo cual corresponde a un crecimiento lineal respecto al tamaño de entrada.

Requerimiento 2

Descripción

Analiza los trayectos filtrados por un método de pago específico, calculando promedios de duración, costo, distancia, peajes y propinas, además de identificar el número de pasajeros más frecuente y la fecha de finalización más común.

Entrada	My_list : lista con los trayectos Metodo_pago : Nombre del método de pago a filtrar
Salidas	tiempo_ms: Tiempo de ejecución en milisegundos total_viajes: Cantidad de viajes con ese método de pago duracion_promedio_min: Tiempo promedio en minutos costo_promedio: Promedio del costo total distancia_promedio: Promedio de distancia en millas peajes_promedio: Promedio de peajes pagados propinas_promedio: Promedio de propinas pasajero_mas_frecuente: Número de pasajeros más frecuente con su frecuencia fecha_finalizacion_mas_frecuente: Fecha de finalización más común
Implementado (Sí/No)	Si, Jonathan

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Recorre toda la lista	O(n)
Recorre viajes en tamaño de x, siendo v <= n	O(x)
Calcular la moda de los datos	O(x)
Calculo de promedio	O(1)
TOTAL	***O(n)***

Análisis

La función trabaja efectivamente con datos de menor y mediano tamaño, esto gracias a que no son demasiado extensos, el cálculo de los promedios se hace más cómodamente al no tener que tener en cuenta muchos datos. Su complejidad de O(n) hace que cuando se manejan una gran cantidad de datos, la función pueda arrojar un tiempo de ejecución más uniforme y pronosticado en perspectiva de su entrada.

Requerimiento 3 no implementado

Requerimiento 4

```python
def req_4(catalog, filtro, fecha_inicio_str, fecha_fin_str):
    inicio = time.time()
    fecha_inicio = datetime.strptime(fecha_inicio_str, "%Y-%m-%d").date()
    fecha_fin = datetime.strptime(fecha_fin_str, "%Y-%m-%d").date()

    combinaciones = defaultdict(list)
    centroides= cargar_centroides("Data/nyc-neighborhoods.csv")

    for t in catalog["taxis"]:
        try:
            fecha = datetime.strptime(t["pickup_datetime"], "%Y-%m-%d %H:%M:%S").date()
            if not (fecha_inicio <= fecha <= fecha_fin):
                continue

            ori = barrio_mas_cercano(float(t["pickup_latitude"]), float(t["pickup_longitude"]), centroides)
            dest = barrio_mas_cercano(float(t["dropoff_latitude"]), float(t["dropoff_longitude"]), centroides)
            dur = (datetime.strptime(t["dropoff_datetime"], "%Y-%m-%d %H:%M:%S") -
                   datetime.strptime(t["pickup_datetime"], "%Y-%m-%d %H:%M:%S")).total_seconds()/60
            combinaciones[(ori, dest)].append((float(t["trip_distance"]), dur, float(t["total_amount"])))
        except:
            continue

    promedios = {k: (sum(d[0] for d in v)/len(v),
                     sum(d[1] for d in v)/len(v),
                     sum(d[2] for d in v)/len(v))
                 for k, v in combinaciones.items()}

    if not promedios:
        return {"mensaje": "No hay trayectos en ese rango de fechas."}

    if filtro.upper() == "MAYOR":
        sel = max(promedios.items(), key=lambda x: x[1][2])
    elif filtro.upper() == "MENOR":
        sel = min(promedios.items(), key=lambda x: x[1][2])
    else:
        return {"mensaje": "Filtro inválido"}

    fin = time.time()
    tiempo_ejecucion_ms = (fin - inicio) * 1000
    (ori, dest), (dist_prom, tiempo_prom, costo_prom) = sel

    return {
        "tiempo_ejecucion_ms": tiempo_ejecucion_ms,
        "filtro_costo": filtro.upper(),
        "total_trayectos": sum(len(v) for v in combinaciones.values()),
        "origen": ori,
        "destino": dest,
        "distancia_promedio_millas": dist_prom,
        "tiempo_promedio_min": tiempo_prom,
        "costo_total_promedio": costo_prom
    }
```

Descripción

Este requerimiento recibe un rango de fechas y busca identificar la combinación de barrios origen–destino con mayor o menor costo promedio de viaje.
Para ello:

Se cargan los centroides de los barrios desde un archivo externo.

Se filtran los trayectos del catálogo que estén dentro del rango de fechas.

Para cada trayecto válido se determina el barrio de origen y destino más cercano (usando la fórmula de Haversine).

Se acumulan las métricas de distancia, duración y costo para cada par de barrios.

Finalmente, se calculan los promedios y se selecciona la combinación con mayor o menor costo promedio según el filtro dado.

Entrada	catalog : catalogo con los trayectos de taxi filtro : criterio de seleccion (“MAYOR” o “MENOR”) fecha_inicio_str : fecha inicial en formato “YYYY-MM-DD” fecha_fin_str : fecha final en formato “YYYY-MM-DD”
Salidas	Origen y destino seleccionados. Distancia promedio (millas). Tiempo promedio (minutos). Costo total promedio (USD). Número total de trayectos procesados. Tiempo de ejecución en milisegundos.
Implementado (Sí/No)	Si, Bastien

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Carga de centroides	O(m) con m el nombre de centroids
Recorrido de todos los trayectos del catalogo (n trayectos), operaciones clasicas y barrio_mas_cercano	O(n*m)
Cálculo de promedios por combinación origen–destino	O(k) con k el nombre de promedios
Selección de la combinación con mayor/menor costo y retorno de resultados	O(k)
TOTAL	***O(n*m)***

Análisis

El costo dominante está en la búsqueda del barrio más cercano para cada trayecto, que requiere recorrer todos los centroides (O(m)) dos veces (origen y destino).
Por lo tanto, el tiempo crece de forma proporcional a n·m.
Con catálogos grandes (ej. 500 000 trayectos) y ~30 barrios, el tiempo puede ser de varios segundos, lo cual se alinea con las pruebas experimentales (~35s en datasets grandes).

Requerimiento 5

```python
def req_5(catalog, filtro, fecha_inicio_str, fecha_fin_str):
    # Diccionario para acumular datos por franja horaria
    franjas = defaultdict(lambda: {
        "costos": [],
        "duraciones": [],
        "pasajeros": [],
        "trayectos": []
    })

    total_filtrados = 0

    for t in trayectos:
        try:
            pickup = datetime.strptime(t["pickup_datetime"], "%Y-%m-%d %H:%M:%S")
            dropoff = datetime.strptime(t["dropoff_datetime"], "%Y-%m-%d %H:%M:%S")
        except Exception:
            continue

        # Filtrar por rango de fechas (solo la fecha de inicio)
        if not (fecha_inicio <= pickup.date() <= fecha_fin):
            continue

        total_filtrados += 1
        franja = pickup.hour  # ej. 13 → franja [13 - 14)

        duracion_min = (dropoff - pickup).total_seconds() / 60
        costo = float(t["total_amount"])
        pasajeros = int(t["passenger_count"])

        franjas[franja]["costos"].append(costo)
        franjas[franja]["duraciones"].append(duracion_min)
        franjas[franja]["pasajeros"].append(pasajeros)
        franjas[franja]["trayectos"].append({
            "costo": costo,
            "dropoff": dropoff
        })

    if total_filtrados == 0:
        return {"mensaje": "No hay trayectos en ese rango de fechas."}

    # Calcular estadísticas por franja
    stats_franjas = []
    for franja, datos in franjas.items():
        if not datos["costos"]:
            continue

        costo_prom = sum(datos["costos"]) / len(datos["costos"])
        duracion_prom = sum(datos["duraciones"]) / len(datos["duraciones"])
        pasajeros_prom = sum(datos["pasajeros"]) / len(datos["pasajeros"])

        # Mayor y menor costo total (con desempate por fecha más reciente)
        mayor_tray = max(datos["trayectos"], key=lambda x: (x["costo"], x["dropoff"]))
        menor_tray = min(datos["trayectos"], key=lambda x: (x["costo"], -x["dropoff"].timestamp()))

        stats_franjas.append({
            "franja": f"[{franja} - {franja+1})",
            "costo_prom": costo_prom,
            "num_trayectos": len(datos["costos"]),
            "duracion_prom": duracion_prom,
            "pasajeros_prom": pasajeros_prom,
            "costo_max": mayor_tray["costo"],
            "costo_min": menor_tray["costo"]
        })

    # Seleccionar franja segun filtro
    if filtro == "MAYOR":
        mejor = max(stats_franjas, key=lambda x: x["costo_prom"])
    elif filtro == "MENOR":
        mejor = min(stats_franjas, key=lambda x: x["costo_prom"])
    else:
        return {"error": "Filtro inválido, use 'MAYOR' o 'MENOR'"}

    fin = time.time()
    tiempo_ms = (fin - inicio) * 1000

    return {
        "tiempo_ejecucion_ms": tiempo_ms,
        "filtro": filtro,
        "total_trayectos": total_filtrados,
        "resultado": mejor
```

Descripción

El requerimiento busca identificar la franja horaria del día (0–23 horas) en la que los trayectos de taxi presentan el costo promedio más alto o más bajo dentro de un rango de fechas dado.
Para resolverlo, se recorren todos los viajes, se filtran por fecha de inicio y se agrupan por hora de recogida. Luego, se calculan estadísticas (promedio de costo, duración, pasajeros) y se selecciona la franja según el filtro indicado.

Entrada	catalog : catalogo con los trayectos de taxi filtro : criterio de seleccion (“MAYOR” o “MENOR”)

	fecha_inicio_str : fecha inicial en formato “YYYY-MM-DD” fecha_fin_str : fecha final en formato “YYYY-MM-DD”
Salidas	Tiempo de ejecución del requerimiento (ms). Filtro aplicado ("MAYOR" o "MENOR"). Número total de trayectos considerados en el rango de fechas. Para la franja horaria seleccionada: Intervalo horario ([h – h+1)), Costo promedio, Número de trayectos, Duración promedio, Pasajeros promedio, Costo máximo de un trayecto, Costo mínimo de un trayecto.
Implementado (Sí/No)	Si Bastien

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Recorrer todos los viajes del catálogo, Filtrar por rango de fechas , Determinar franja horaria y acumular datos en diccionario	O(n)
Calcular estadísticas por cada franja horaria (24)	O(1)
Seleccionar franja con mayor/menor costo promedio	O(1)
TOTAL	***O(n)***

Análisis

La implementación recorre los viajes una sola vez, filtrando por fechas y acumulando estadísticas en un diccionario de franjas (0–23). Como el número de franjas es fijo (24), los cálculos de promedios y la selección de la franja son operaciones en tiempo constante.

Esto garantiza que el algoritmo escala linealmente con el número de trayectos (**O(n)**), lo cual lo hace eficiente incluso con catálogos grandes (ej. cientos de miles o millones de trayectos).

En las pruebas realizadas, el tiempo de ejecución fue razonable (del orden de segundos para catálogos medianos). La salida cumple con todos los requisitos: tiempo, filtro aplicado, total de trayectos considerados y estadísticas completas de la franja horaria más/menos costosa.

Requerimiento 6

Descripción

Analiza trayectos que inician en un barrio dado dentro de un rango de fechas, calculando promedios de distancia y duración, el barrio destino más frecuente y las estadísticas de cada método de pago, destacando el más usado y el que más recaudó.

Entrada	My_list : lista con los trayectos neighborhoods: Lista de barrios con sus centroides barrio_inicio: Nombre del barrio de inicio fecha_inicial: Fecha inicial ("%Y-%m-%d") fecha_final: Fecha final ("%Y-%m-%d")
Salidas	total_viajes: Cantidad de viajes filtrados. distancia_promedio: Distancia promedio de los viajes. duracion_promedio: Duración promedio en minutos. barrio_destino_mas_visitado: Barrio más frecuente como destino. metodos_pago: Lista con estadísticas por método de pago: metodo: Nombre del método de pago. cantidad_trayectos: Número de trayectos con ese método. promedio_pago: Promedio de dinero pagado. es_mas_usado: True si fue el método con más viajes. es_mas_recaudo: True si fue el método que más dinero recaudó. duracion_promedio: Promedio de duración de trayectos con ese método.
Implementado (Sí/No)	Si, Jonathan

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Pasos	**Complejidad**
Pasar str a datetime	O(1)
Se recorre los viajes (n) a la vez que se revisan los barrios (m) y se recolecta los datos	O(n*m)
Se recorren los destinos	O(n)
Análisis de métodos de pago	O(n)
TOTAL	***O(n*m)***

Análisis

La función se comporta alrededor de O(n) aunque podríamos anotarlo como O(n*m) donde m es la cantidad de barrios, los cuales en su mayoría no son una estadística muy grande que comprometa mucho la complejidad de la función, aunque si hay que tenerlo en cuenta. Con todos los tamaños de cantidad de datos se comporta bien siendo más tardío cuando los registros son cada vez mas numerosos teniendo que recorrer más aumentando su ejecución.