

# Grupo 01 – Reto 1 - Análisis de resultados

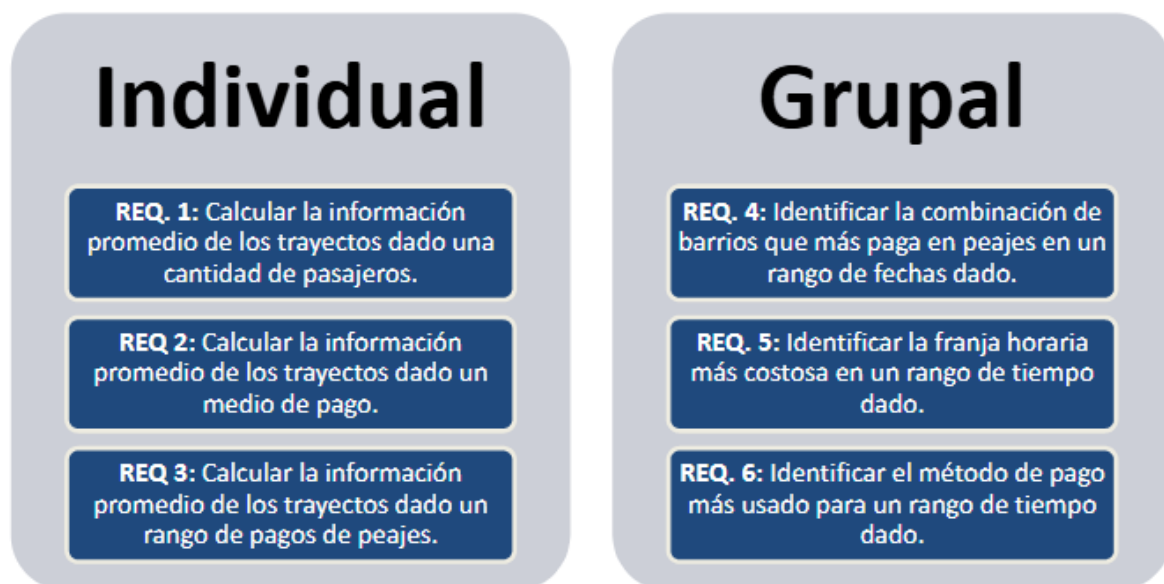
## Integrantes:

Espitia Reyes, Juan Esteban - [je.espitia@uniandes.edu.co](mailto:je.espitia@uniandes.edu.co) - 202516958

Salazar Ochoa, Fabio Andrés - [fa.salazar@uniandes.edu.co](mailto:fa.salazar@uniandes.edu.co) - 202511238

Tovar Palomo, Samuel Felipe - [sf.tovarp1@uniandes.edu.co](mailto:sf.tovarp1@uniandes.edu.co) - 202211625

## Análisis de complejidad en los requerimientos:



**Req-1:**

El responsable del requerimiento uno fue *Samuel Felipe*, en el análisis de la complejidad temporal se denota que no es un procedimiento tan pesado, al ser la mayoría de los algoritmos de asignación y no de bucles para recorrer la lista.

```
146 def req_1(catalog, pasajeros):
147     inicio = get_time()
148
149     n = len(catalog["taxi"])
150
151     count = 0
152     tiempo_prom = 0.0
153     costo_total = 0.0
154     dist_prom = 0.0
155     tarifa_prom = 0.0
156     tip_prom = 0.0
157
158     pago = {} # tipo de pago y cantidad
159     fecha = {}
160
161     for i in range(n):
162         e = catalog["taxi"][i]
163         if e["passenger_count"] == pasajeros:
164             count += 1
165
166             # duración en minutos
167             dur_min = (e["dropoff_datetime"] - e["pickup_datetime"]).total_seconds() / 60.0
168             tiempo_prom += dur_min
169
170             # promedios solicitados
171             costo_total += e["total_amount"]
172             dist_prom += e["trip_distance"]
173             tarifa_prom += e["total_amount"]
174             tip_prom += e["tip_amount"]
175
176             # pago más usado
177             p = e["payment_type"]
178             pago[p] = pago.get(p, 0) + 1
179
180             # fecha de inicio (solo AAAA-MM-DD, sin horas)
181             f = e["pickup_datetime"].strftime("%Y-%m-%d")
182             fecha[f] = fecha.get(f, 0) + 1
183
184     if count == 0:
185         end = get_time()
186         return {
187             "tiempo_ms": delta_time(inicio, end),
188             "total_trayectos": 0,
189             "prom_duracion_min": 0.0,
190             "prom_costo_total": 0.0,
191             "prom_dist_millas": 0.0,
192             "prom_tarifa": 0.0,
193             "pago_mas_usado": "N/A",
194             "promina_promedio": 0.0,
195             "fecha_mas_frecuente": "N/A"
196         }
197
198     # cálculo de máximos (pago y fecha más frecuente)
199     pago_top, pago_top_count = None, -1
200     for k, v in pago.items():
201         if v > pago_top_count:
202             pago_top, pago_top_count = k, v
203
204     fecha_top, fecha_top_count = None, -1
205     for k, v in fecha.items():
206         if v > fecha_top_count:
207             fecha_top, fecha_top_count = k, v
208
209     final = get_time()
210     return {
211         "tiempo_ms": round(delta_time(inicio, final), 0),
212         "total_trayectos": count,
213         "prom_duracion_min": round(tiempo_prom / count, 0),
214         "prom_costo_total": round(costo_total / count, 0),
215         "prom_dist_millas": round(dist_prom / count, 0),
216         "prom_tarifa": round(tarifa_prom / count, 0),
217         "pago_mas_usado": f"{pago_top} - {pago_top_count}",
218         "promina_promedio": round(tip_prom / count, 0),
219         "fecha_mas_frecuente": fecha_top
220     }
```

En la foto y en el código en sí, podemos denotar en el área que resalte como “O(1)” como las asignaciones iniciales de valores tales como el size, módulo de nuestro documento “array\_list.py” con complejidad “O(1)” y valores usados para guardar los valores del filtro en si de los datos solicitados en el requerimiento.

```

146 def req_1(catalog, pasajeros):
164     inicio = get_time()
165
166     n = al.size(catalog["taxis"])
167
168     conteo = 0
169     tiempo_prom = 0.0
170     costo_total = 0.0
171     dist_prom = 0.0
172     tolls_prom = 0.0
173     tip_prom = 0.0
174
175     pagos = {} # tipo de pago y cantidad
176     fechas = {}

```

O(1)

En el apartado resaltado como “O(n)” se puede ver un bucle iterativo que recorre la lista, extrayendo los elementos de la llave “taxis” del diccionario “catalog”.

```

178     for i in range(n):
179         e = al.get_element(catalog["taxis"], i)
180         if e["passenger_count"] == pasajeros:
181             conteo += 1
182
183             # duración en minutos
184             dur_min = (e["dropoff_datetime"] - e["pickup_datetime"]).total_seconds() / 60.0
185             tiempo_prom += dur_min
186
187             # promedios solicitados
188             costo_total += e["total_amount"]
189             dist_prom += e["trip_distance"]
190             tolls_prom += e["tolls_amount"]
191             tip_prom += e["tip_amount"]
192
193             # pago nds usado
194             p = e["payment_type"]
195             pagos[p] = pagos.get(p, 0) + 1
196
197             # fecha de inicio (solo AAAA-MM-DD, sin horas)
198             f = e["pickup_datetime"].strftime("%Y-%m-%d")
199             fechas[f] = fechas.get(f, 0) + 1
200

```

O(n)

En el código, también se puede ver comparaciones lógicas, asignaciones, sumas y restas para al final hacer un print de todos los valores ya recolectados. Así haciendo un print del resultado deseado del requerimiento.

```

201     if conteo == 0:
202         end = get_time()
203         return {
204             "tiempo_ms": delta_time(inicio, end),
205             "total_trayectos": 0,
206             "prom_duracion_min": 0.0,
207             "prom_costo_total": 0.0,
208             "prom_dist_millas": 0.0,
209             "prom_peajes": 0.0,
210             "pago_mas_usado": "N/A - 0",
211             "propina_promedio": 0.0,
212             "fecha_mas_frecuente": "N/A"
213         }
214
215     # cálculo de máximos (pago y fecha más frecuente)
216     pago_top, pago_top_conteo = None, -1
217     for k, v in pagos.items():
218         if v > pago_top_conteo:
219             pago_top, pago_top_conteo = k, v
220
221     fecha_top, fecha_top_cnt = None, -1
222     for k, v in fechas.items():
223         if v > fecha_top_cnt:
224             fecha_top, fecha_top_cnt = k, v
225
226     final = get_time()
227     return {
228         "tiempo_ms": round(delta_time(inicio, final), 3),
229         "total_trayectos": conteo,
230         "prom_duracion_min": round(tiempo_prom / conteo, 3),
231         "prom_costo_total": round(coste_total / conteo, 3),
232         "prom_dist_millas": round(dist_prom / conteo, 3),
233         "prom_peajes": round(tolls_prom / conteo, 3),
234         "pago_mas_usado": f"{pago_top} - {pago_top_conteo}",
235         "propina_promedio": round(tip_prom / conteo, 3),
236         "fecha_mas_frecuente": fecha_top
237     }
238

```

Al sumar las complejidades

$O(1) + O(1) + O(N)$ , en notación Big-O se refiere a que **la complejidad es  $O(N)$**

## Req-2:

El responsable de lograr este requerimiento fue el estudian Juan Esteban, en el código en el inicio se puede ver asignación de valores ( $O(1)$ ), después se ve un recorrido por las variables de la lista ya anteriormente usada para correr un filtro dependiente del método de pago, luego se asignan valores y un caso base de retorno

```

353 def req_2(catalog, n_pago):
354
355     inicio = get_time()
356
357     t_metodos = al.size(catalog["taxis"])
358     filtrados = 0
359     dur_prom = 0 # Minutos
360     c_prom = 0 # Dólares
361     dis_prom = 0 # Millas
362     c_pasajeros = 0
363     propina_prom = 0
364     frecuencia_pa = {}
365     frecuencia_fecha = {}
366
367     for i in range(t_metodos):
368         pago = al.get_element(catalog["taxis"], i)
369         if pago["payment_type"] == "n_pago":
370             filtrados += 1
371             dur_prom += (pago["dropoff_datetime"] - pago["pickup_datetime"]).total_seconds() / 60
372             c_prom += pago["total_amount"]
373             dis_prom += pago["trip_distance"]
374             c_pasajeros += pago["total_amount"]
375             propina_prom += pago["tip_amount"]
376             pasajero = pago["passenger_count"]
377             fecha = pago["dropoff_datetime"].strftime("%Y-%m-%d")
378
379             frecuencia_pa[pasajero] = frecuencia_pa.get(pasajero, 0) + 1
380             frecuencia_fecha[fecha] = frecuencia_fecha.get(pasajero, 0) + 1
381
382     if filtrados == 0: # hora de procesar toda esta vaina jaja
383         fin = get_time()
384         return_t = {
385             "tiempo de ejecución (ms)": delta_time(inicio, fin),
386             "trayectos totales": t_metodos,
387             "trayectos filtrados": 0,
388             "duración promedio p/trayecto (min)": 0.0,
389             "costo promedio (USD)": 0.0,
390             "distancia promedio (millas)": 0.0,
391             "costo de pasaje promedio": 0.0,
392             "propina promedio": 0.0,
393             "Pasajero mas frecuente": "N/A",
394             "frecuencia de fecha": "N/A"
395         }
396     elif filtrados > 0:
397         fin = get_time()
398
399         p_frecuente = None
400         frecuencia_p = 0
401         for i in frecuencia_pa:
402             if frecuencia_pa[i] > frecuencia_p:
403                 p_frecuente = i
404                 frecuencia_p = frecuencia_pa[i]
405
406         fechas_mas_frecuente = None
407         frecuencia_f = 0
408         for i in frecuencia_fecha:
409             if frecuencia_fecha[i] > frecuencia_f:
410                 fechas_mas_frecuente = i
411                 frecuencia_f = frecuencia_fecha[i]
412
413         return_t = {
414             "tiempo de ejecución (ms)": delta_time(inicio, fin),
415             "trayectos totales": t_metodos,
416             "trayectos filtrados": filtrados,
417             "duración promedio p/trayecto (min)": dur_prom/t_metodos,
418             "costo promedio (USD)": c_prom/t_metodos,
419             "distancia promedio (millas)": dis_prom/t_metodos,
420             "costo de pasaje promedio": c_pasajeros/t_metodos,
421             "propina promedio": propina_prom/t_metodos,
422             "Pasajero mas frecuente": f"[{p_frecuente}] ({frecuencia_p})",
423             "frecuencia de fecha": fechas_mas_frecuente
424         }
425     else:
426         raise Exception("Error: hay algo que no funciona en la carga de datos del req 2")
427     return return_t

```

Después de asignarse el caso base, se agrega dos bucles iterativos para conseguir el mayor número de frecuencia en tanto como fecha y cliente recurrente

```

311         for i in frecuencia_pa:
312             if frecuencia_pa[i] > frecuencia_p:
313                 p_frecuente = i
314                 frecuencia_p = frecuencia_pa[i]
315
316         fechas_mas_frecuente = None
317         frecuencia_f = 0
318         for i in frecuencia_fecha:
319             if frecuencia_fecha[i] > frecuencia_f:
320                 fechas_mas_frecuente = i
321                 frecuencia_f = frecuencia_fecha[i]
322

```

Al sumar las complejidades mostradas por el código, nos quedamos con:

$O(1) + O(n) + O(m) + O(z)$ , refiriéndose respectivamente en notación Big-O a:  **$O(n+m+z)$**

## Req-3:

El estudiante responsable de este requerimiento es Fabio Andrés. En el inicio se puede notar la asignación de valores haciéndolo de complejidad  $O(1)$ , después se puede ver un bucle iterativo sobre el size de la lista de catalog, siendo este un  $O(n)$ .

```
def req_3(catalogo, pago_min, pago_max):
    inicio = get_time()

    totalviajes = al.size(catalogo["taxis"])
    filtrados = 0
    suma_duracion_min = 0
    suma_total = 0
    suma_distancia = 0
    suma_pasajes = 0
    suma_propinas = 0

    frec_pasajeros = {}
    frec_fecha = {}

    for i in range(totalviajes):
        viaje = al.get_element(catalogo["taxis"], i)
        costo = viaje["total_amount"]

        if costo is not None and pago_min <= costo <= pago_max:
            filtrados += 1

            duracion_min = (viaje["dropoff_datetime"] - viaje["pickup_datetime"]).total_seconds() / 60
            suma_duracion_min += duracion_min

            suma_total += costo
            suma_distancia += viaje["trip_distance"]
            suma_pasajes += viaje["tolls_amount"]
            suma_propinas += viaje["trip_amount"]
            pasajeros = viaje["passenger_count"]

            if pasajeros in frec_pasajeros:
                frec_pasajeros[pasajeros] += 1
            else:
                frec_pasajeros[pasajeros] = 1

            fecha_final = viaje["dropoff_datetime"].strftime("%Y-%m-%d")
            if fecha_final in frec_fecha:
                frec_fecha[fecha_final] += 1
            else:
                frec_fecha[fecha_final] = 1

    if filtrados > 0:
        promedio_duracion = suma_duracion_min / filtrados
        promedio_costo = suma_total / filtrados
        promedio_distancia = suma_distancia / filtrados
        promedio_pasajes = suma_pasajes / filtrados
        promedio_propinas = suma_propinas / filtrados

        pasajeros_max_frec = None
        frecuencia_pasajeros = 0
        for clave in frec_pasajeros:
            curr_frec = frec_pasajeros[clave]
            if curr_frec > frecuencia_pasajeros:
                frecuencia_pasajeros = curr_frec
                pasajeros_max_frec = clave

        fecha_final_max_frec = None
        frecuencia_fecha_max_frec = 0
        for fecha in frec_fecha:
            curr_frec_fecha = frec_fecha[fecha]
            if curr_frec_fecha > frecuencia_fecha_max_frec:
                frecuencia_fecha_max_frec = curr_frec_fecha
                fecha_final_max_frec = fecha

    else:
        promedio_duracion = 0.0
        promedio_costo = 0.0
        promedio_distancia = 0.0
        promedio_pasajes = 0.0
        promedio_propinas = 0.0
        pasajeros_max_frec = None
        frecuencia_pasajeros = 0
        fecha_final_max_frec = None
```



Dentro del bucle podemos ver condiciones lógicas tales como un filtro basado en un valor de “pago mínimo” el cual el usuario en el view puede proporcionar. Dentro del filtro aparte de las asignaciones de valores para el print que satisface el requerimiento, se encuentran dos bucles iterativos que sacan la fecha con mayor frecuencia dentro la base de datos y el pasajero con más frecuencia, las cuales son  $O(m)$  y  $O(z)$  ya que tienen distintos tamaños, lo mismo pasa con el bucle iterativo inicial el cual tiene  $O(n)$ .

```
final = get_time()
tiempo = delta_time(inicio, final)

retorno = {
    "tiempo de ejecucion (ms)": tiempo,
    "total de viajes válidos": filtrados,
    "promedio duración (min)": promedio_duracion,
    "promedio costo (USD)": promedio_costo,
    "promedio distancia (millas)": promedio_distancia,
    "promedio pago peajes": promedio_peajes,
    "num pasajeros mas frecuente": f"{pasajeros_mas_frec}-{frecuencia_pasajeros}",
    "promedio propinas": promedio_propinas,
    "fecha final mas frecuente": fecha_final_mas_frec
}

return retorno
```

$O(1)$

Al final, el retorno se acomoda con los valores ya guardados, listos para el print en el view. Para concluir, al estar complejidades de valor lineal (refiriéndose a  $O(m)$  y  $O(z)$ ) dentro de la complejidad de  $O(n)$ , al recorrerse las claves de  $m$  y  $z$  cada vez que se itera  $n$ , se podría representar así la complejidad:

$O(n) \cdot (O(m) + O(z))$ , lo cual terminaría representando  **$O(n \cdot (m + z))$**  como la complejidad en Big-O de la función.

## Req-4:

La función comienza con una función que identifica el barrio más cercano. Cuya complejidad temporal es  $O(m)$ , siendo  $m$  el tamaño de la lista de barrios. Después, todas las asignaciones de variables que se ven posteriormente son de complejidad constante =  $O(1)$ .

```
def req_4(catalog, f_costo, f_inicial, f_final):  
  
    def barrio_mas_cercano(lat, lon, lista_barrios):  
        mejor_barrio = None  
        mejor_distancia = float('inf')  
        total_barrios = al.size(lista_barrios)  
  
        for i in range(total_barrios):  
            barrio = al.get_element(lista_barrios, i)  
            distancia = haversine(lat, lon, barrio["latitude"], barrio["longitude"])  
            if distancia < mejor_distancia:  
                mejor_distancia = distancia  
                mejor_barrio = barrio["neighborhood"]  
  
        return mejor_barrio  
  
    inicio = get_time()  
  
    fecha_inicial = datetime.strptime(f_inicial, "%Y-%m-%d").date()  
    fecha_final = datetime.strptime(f_final, "%Y-%m-%d").date()  
  
    totalviajes = al.size(catalog["taxis"])  
    filtrados = 0  
  
    combinaciones = {}
```

Luego, la función sigue con un bucle que recorre cada viaje, obtiene su fecha de inicio y, si está en el rango indicado por los parámetros de la función, lo filtra y calcula barrios de origen y destino.

Siendo  $n$  = tamaño de `catalog["taxis"]` y  $m$  = tamaño de `catalog["neighborhoods"]`, la complejidad de este bucle sería de  $O(n*m)$ , porque recorre ambas listas (una en el `for i`



in range (totalviajes), y la otra en los llamados de la función “barrio\_mas\_cercano”).

```
for i in range(totalviajes):
    viaje = al.get_element(catalog["taxis"], i)
    pkup_date = viaje["pickup_datetime"].date()

    if fecha_inicial <= pkup_date <= fecha_final:
        filtrados += 1

    origen = barrio_mas_cercano(viaje["pickup_latitude"], viaje["pickup_longitude"], catalog["neighborhoods"])
    destino = barrio_mas_cercano(viaje["dropoff_latitude"], viaje["dropoff_longitude"], catalog["neighborhoods"])

    if origen != destino:
        clave = (origen, destino)

        duracion = (viaje["dropoff_datetime"] - viaje["pickup_datetime"]).total_seconds() / 60

        if clave not in combinaciones:
            combinaciones[clave] = {
                "distancia": 0.0,
                "duracion": 0.0,
                "costo": 0.0,
                "conteo": 0
            }

        distancia = haversine(viaje["pickup_latitude"], viaje["pickup_longitude"], viaje["dropoff_latitude"], viaje["dropoff_longitude"])

        combinaciones[clave]["distancia"] += distancia
        combinaciones[clave]["duracion"] += duracion
        combinaciones[clave]["costo"] += viaje["total_amount"]
        combinaciones[clave]["conteo"] += 1
```

Después, considerando a  $P = \text{len}(\text{combinaciones})$ , vemos que la complejidad de este bucle sería  $O(P)$ , dado a que recorre todo el diccionario de combinaciones en el ciclo “for clave in combinaciones”

```
if len(combinaciones) > 0:
    if f_costo == "MAYOR":
        m_costo = float('-inf')
    else:
        m_costo = float('inf')

    for clave in combinaciones:
        datos = combinaciones[clave]
        conteo = datos["conteo"]
        if conteo > 0:
            costo_prom = datos["costo"] / conteo

            if (f_costo == "MAYOR" and costo_prom > m_costo) or (f_costo == "MENOR" and costo_prom < m_costo):
                m_costo = costo_prom
                barrio_origen, barrio_destino = clave
                distancia_promedio = round(datos["distancia"] / conteo, 3)
                duracion_promedio = round(datos["duracion"] / conteo, 3)
                costo_promedio = round(costo_prom, 3)
```

Finalmente, la función termina determinando el tiempo de ejecución y creando el diccionario del retorno final, todas estas operaciones tienen complejidad constante =  $O(1)$

```
final = get_time()
tiempo = delta_time(inicio, final)

retorno = {
    "tiempo de ejecucion (ms)": round(tiempo,3),
    "filtro": f_costo,
    "total de viajes filtrados": filtrados,
    "barrio de origen": barrio_origen,
    "barrio de destino": barrio_destino,
    "distancia promedio (km)": distancia_promedio,
    "duracion promedio (min)": duracion_promedio,
    "costo promedio (USD)": costo_promedio
}

return retorno
```

Finalmente, podemos determinar la complejidad temporal  $T(n, m, P)$  como:

$$O(n*m) + O(P) = O((n*m) + P).$$

Sin embargo,  $P$  no dictamina el crecimiento porque está acotado por  $n$  y  $m$ , esto porque  $P \leq n$  (cada viaje solo puede generar una combinación) y  $m > 0$  (en el peor caso). Lo que haría que  $n*m \geq P$ .

Entonces, la complejidad temporal de la función termina siendo  **$O(n*m)$** .

## Req-5:

La instrucción más costosa en términos de complejidad temporal para esta función es su primer bucle que itera exactamente una vez por cada viaje almacenado en el catálogo (total viajes =  $n$ )  $O(n)$ . Las primeras líneas simplemente son asignación de variables  $O(1)$ .

```
539 def req_5(catalog, f_costo, f_inicial, f_final):
540
541     inicio = get_time()
542
543     fecha_inicial = datetime.strptime(f_inicial, "%Y-%m-%d").date()
544     fecha_final = datetime.strptime(f_final, "%Y-%m-%d").date()
545     totalviajes = al.size(catalog["taxis"])
546     filtrados = 0
547
548     franjas = {}
549
550     for i in range(totalviajes):
551         viaje = al.get_element(catalog["taxis"], i)
552         pkup_date = viaje["pickup_datetime"].date()
553
554         if fecha_inicial <= pkup_date <= fecha_final:
555             filtrados += 1
556
557             hora = viaje["pickup_datetime"].hour
558             if hora not in franjas:
559                 franjas[hora] = {
560                     "costo_total": 0.0,
561                     "conteo": 0,
562                     "duracion total": 0.0,
563                     "total pasajeros": 0,
564                     "costo maximo": float('-inf'),
565                     "costo minimo": float('inf')
566                 }
567             duracion = (viaje["dropoff_datetime"] - viaje["pickup_datetime"]).total_seconds() / 60
568             costo = float(viaje["total_amount"])
569             pasajeros = viaje["passenger_count"]
570
571             franjas[hora]["costo_total"] += costo
572             franjas[hora]["conteo"] += 1
573             franjas[hora]["duracion total"] += duracion
574             franjas[hora]["total pasajeros"] += pasajeros
575
576             if costo > franjas[hora]["costo maximo"]:
577                 franjas[hora]["costo maximo"] = costo
578             if costo < franjas[hora]["costo minimo"]:
579                 franjas[hora]["costo minimo"] = costo
```

Además, como `catalog["taxis"]` es un `array_list`, la operación `al.get_element` es  $O(1)$  y en cada iteración, se realizan comparaciones de fechas, operaciones aritméticas y actualizaciones en un diccionario (`franjas`). Todas estas operaciones son de tiempo constante  $O(1)$ .

Por lo tanto, el costo total de este bloque es  $O(n)$ .

```

581 mejor_hora = None
582 if f_costo == "MAYOR":
583     mejor_costo_prom = float('-inf')
584 else:
585     mejor_costo_prom = float('inf')
586
587 for hora in franjas:
588     datos = franjas[hora]
589     costo_prom = datos["costo_total"] / datos["conteo"]
590
591     if (f_costo == "MAYOR" and costo_prom > mejor_costo_prom) or (f_costo == "MENOR" and costo_prom < mejor_costo_prom):
592         mejor_hora = hora
593         mejor_costo_prom = costo_prom
594
595     datos = franjas[mejor_hora]
596     conteo = datos["conteo"]
597
598 final = get_time()
599 tiempo = delta_time(inicio, final)
600
601 if mejor_hora is None:
602     return {
603         "tiempo de ejecucion (ms)": tiempo,
604         "filtro": f_costo,
605         "total de viajes filtrados": filtrados,
606         "franja horaria": None,
607         "costo promedio (USD)": 0.0,
608         "total de viajes en la franja": 0,
609         "duracion promedio (min)": 0.0,
610         "pasajeros promedio": 0.0,
611         "costo maximo (USD)": None,
612         "costo minimo (USD)": None
613     }
614
615 datos = franjas[mejor_hora]
616 conteo = datos["conteo"]
617
618 retorno = {
619     "tiempo de ejecucion (ms)": round(tiempo,3),
620     "filtro": f_costo,
621     "total de viajes filtrados": filtrados,
622     "franja horaria": f"{mejor_hora} - {mejor_hora + 1}" if mejor_hora < 23 else f"{mejor_hora} - 0",
623     "costo promedio (USD)": round(datos["costo_total"] / conteo,3),
624     "total de viajes en la franja": conteo,
625     "duracion promedio (min)": round(datos["duracion total"] / conteo,3),
626     "pasajeros promedio": round(datos["total pasajeros"] / conteo,3),
627     "costo maximo (USD)": datos["costo maximo"],
628     "costo minimo (USD)": datos["costo minimo"]
629 }
630
631 return retorno

```

Para el segundo bloque, tenemos algo más simple en términos de complejidad temporal. Aunque presentamos otro ciclo, el número de claves en franjas es como máximo 24, porque solo existen 24 horas posibles en un día. Esto significa que este bucle es independiente de  $n$  y siempre toma tiempo constante:  $O(1)$ . El resto son definición de variables y asignación de valores a un diccionario.  $O(1)$

Por ende, esta función tiene una complejidad temporal  $O(n)$ .

## Req-6:

Notación y supuestos:

Sea  $n$  = número total de viajes en `catalog["taxis"]`.

Sea  $m$  = número de barrios en `catalog["neighborhoods"]`.

Sea  $k$  = número de valores en el diccionario destinos

Sea  $j$  = número de valores en el diccionario pagos

```
633 def req_6(catalog, b_inicio, f_inicial, f_final):
634
635     def barrio_mas_cercano(lat, lon, lista_barrios):
636         mejor_barrio = None
637         mejor_distancia = float('inf')
638         barrios = al.size(lista_barrios)
639
640     O(m) for i in range(barrios):
641         barrio = al.get_element(lista_barrios, i)
642         distancia = haversine(lat, lon, barrio["latitude"], barrio["longitude"])
643         if distancia < mejor_distancia:
644             mejor_distancia = distancia
645             mejor_barrio = barrio["neighborhood"]
646
647     return mejor_barrio
648
```

Empezando la función definimos una función con un ciclo que recorre la cantidad del `catalog["neighborhoods"]` teniendo con sigo una complejidad de  $O(m)$ .

```

649 inicio = get_time()
650 fecha_inicial = datetime.strptime(f_inicial, "%Y-%m-%d").date()
651 fecha_final = datetime.strptime(f_final, "%Y-%m-%d").date()
652 totalviajes = al.size(catalog["taxis"])
653 filtrados = 0
654
655 O(1)
656 distancia_km = 0.0
657 duracion_min = 0.0
658 dist_km = 0
659 dur_min = 0
660
661 destinos = {}
662 pagos = {}
663
664 for i in range(totalviajes):
665     viaje = al.get_element(catalog["taxis"], i)
666     pkup_date = viaje["pickup_datetime"].date()
667     if fecha_inicial <= pkup_date <= fecha_final:
668         origen = barrio_mas_cercano(viaje["pickup_latitude"], viaje["pickup_longitude"], catalog["neighborhoods"])
669         if origen == h_inicio:
670             filtrados += 1
671             dist_km += haversine(viaje["pickup_latitude"], viaje["pickup_longitude"], viaje["dropoff_latitude"], viaje["dropoff_longitude"])
672             dur_min += (viaje["dropoff_datetime"] - viaje["pickup_datetime"]).total_seconds() / 60
673
674             destino = barrio_mas_cercano(viaje["dropoff_latitude"], viaje["dropoff_longitude"], catalog["neighborhoods"])
675
676             distancia_km += dist_km
677             duracion_min += dur_min
678
679             if destino in destinos:
680                 destinos[destino] += 1
681             else:
682                 destinos[destino] = 1
683
684             metodo_pago = viaje["payment_type"]
685             if metodo_pago not in pagos:
686                 pagos[metodo_pago] = {
687                     "conteo": 0,
688                     "total_pagado": 0.0,
689                     "total duracion": 0.0
690                 }
691             pagos[metodo_pago]["conteo"] += 1
692             pagos[metodo_pago]["total_pagado"] += viaje["total_amount"]
693             pagos[metodo_pago]["total duracion"] += dur_min
694

```

$O(m)$

$O(n)$

$O(m)$

Después vemos el ciclo principal que recorre el total de viajes ( $O(n)$ ) pero dentro de estos ciclos se llama a la función definida anteriormente dependiendo de unos condicionales y con una complejidad  $O(m)$  como ya dijimos.

```

694     if filtrados == 0:
695         fin = get_time()
696         tiempo = delta_time(inicio, fin)
697         retorno = {
698             "tiempo de ejecucion (ms)": tiempo,
699             "total de viajes filtrados": filtrados,
700             "distancia promedio (km)": 0.0,
701             "duracion promedio (min)": 0.0,
702             "barrio destino mas frecuente": None,
703             "pagos": []
704         }
705         return retorno
706     destino_mas_frec = None
707     max = float('-inf')
708     for i in destinos:
709         x = destinos[i]
710         if x > max:
711             max = x
712             destino_mas_frec = i
713
714     metodo_pago_max = None
715     max_conteo = float('-inf')
716     metodo_recaudo_max = None
717     max_recaudo = float('-inf')
718
719     for metodo in pagos:
720         if pagos[metodo]["conteo"] > max_conteo:
721             max_conteo = pagos[metodo]["conteo"]
722             metodo_pago_max = metodo
723         if pagos[metodo]["total_pagado"] > max_recaudo:
724             max_recaudo = pagos[metodo]["total_pagado"]
725             metodo_recaudo_max = metodo
726
727     lista_pagos = []
728     for metodo in pagos:
729         count = pagos[metodo]["conteo"]
730         lista_pagos.append({
731             "tipo": metodo,
732             "cantidad de viajes": count,
733             "precio promedio (USD)": pagos[metodo]["total_pagado"] / count,
734             "¿Es el mas usado?": metodo == metodo_pago_max,
735             "¿Es el que genera más recaudo?": metodo == metodo_recaudo_max,
736             "tiempo promedio (min)": pagos[metodo]["total duracion"] / count
737         })
738
739     fin = get_time()
740     tiempo = delta_time(inicio, fin)
741
742     retorno = {
743         "tiempo de ejecucion (ms)": round(tiempo,3),
744         "total de viajes filtrados": filtrados,
745         "distancia promedio (km)": round(distancia_km / filtrados,3),
746         "duracion promedio (min)": round(duracion_min / filtrados,3),
747         "barrio destino mas frecuente": destino_mas_frec,
748         "pagos": lista_pagos
749     }
750
751     return retorno

```

Por último, vemos la implementación de otros ciclos que recorren diccionarios creados anteriormente con una complejidad  $O(k)$  y  $O(j)$ . Además, la asignación de valores con una complejidad  $O(1)$ .

Pero al final la complejidad será  $O(n*m)$  gracias al ciclo que llama dentro de él a la función de barrios haciendo que las complejidades se multipliquen y dando el peor caso posible.



