Grupo 01 – Reto 1 - Análisis de resultados

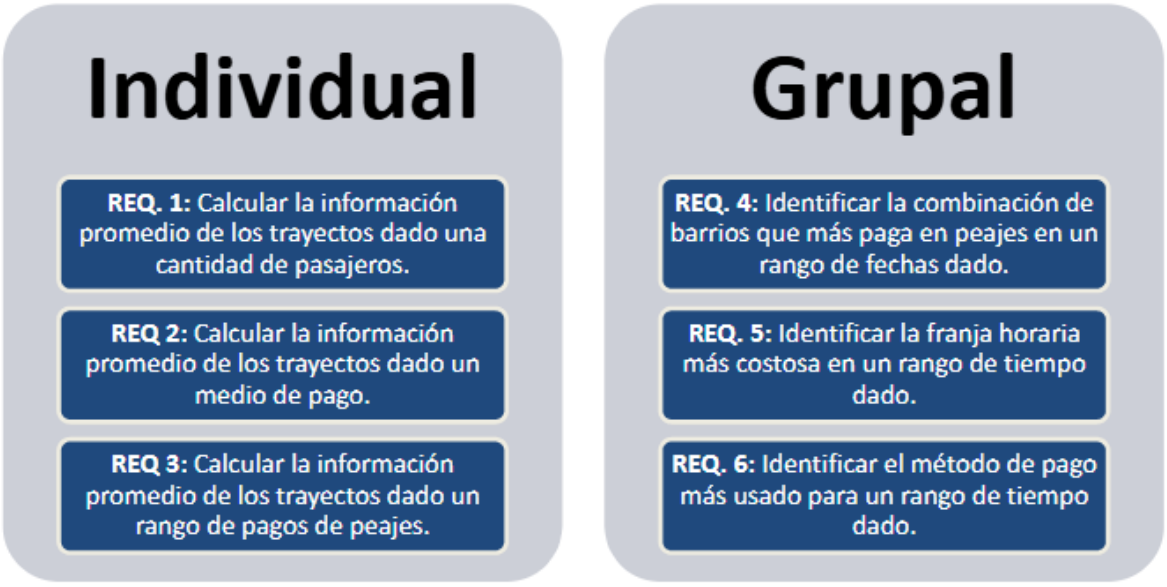
Integrantes:

Espitia Reyes, Juan Esteban - je.espitia@uniandes.edu.co - 202516958

Salazar Ochoa, Fabio Andrés - fa.salazar@uniandes.edu.co - 202511238

Tovar Palomo, Samuel Felipe - 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.

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 más 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 logicas, 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.

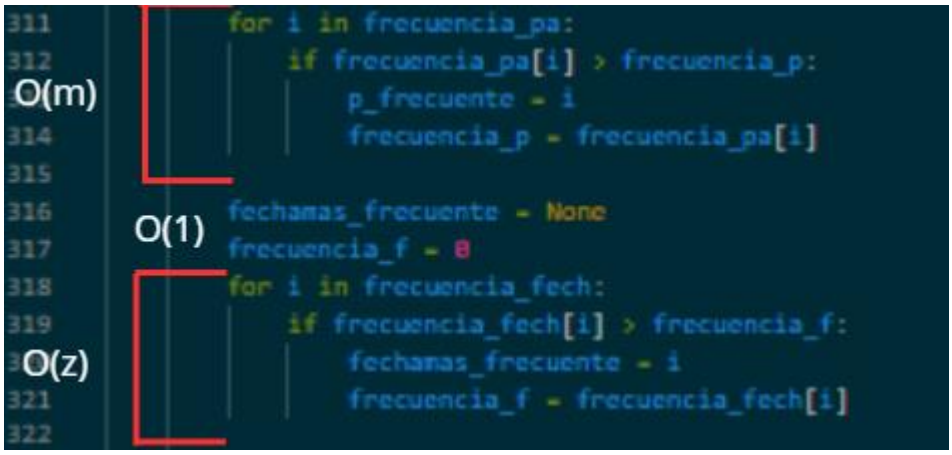
Al sumar las complejidades

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

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



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

O(1) + O(n) + O(m) + O(z), refiriéndose respectivamente en notacion 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).

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).

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)*(O(m)+O(z)), lo cual terminaría representando **O(n*(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_l

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

Después, considerando a P = len(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:

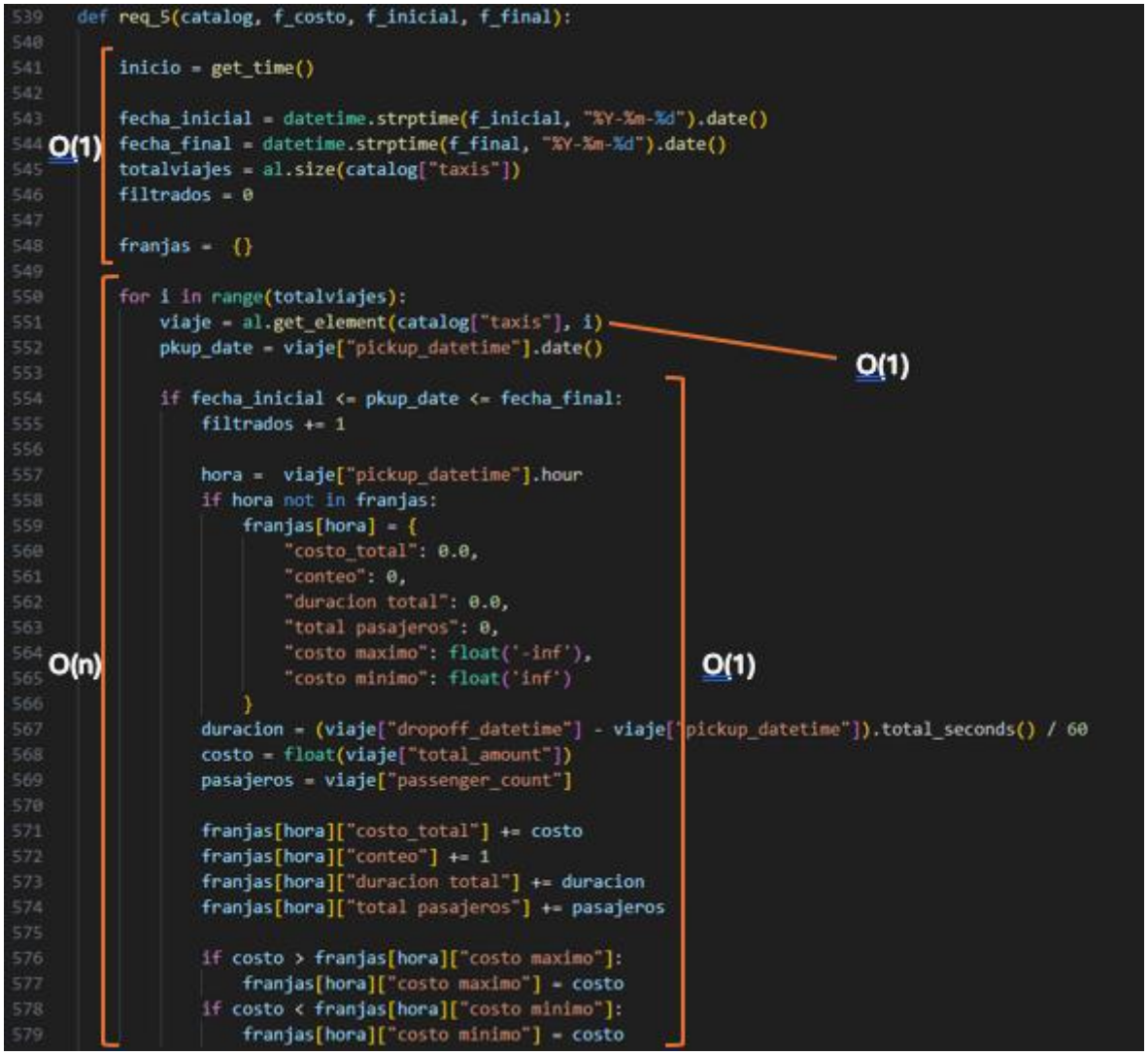
O(n*m) + O(P) = O((n*m) +P).

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

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

Req-5:

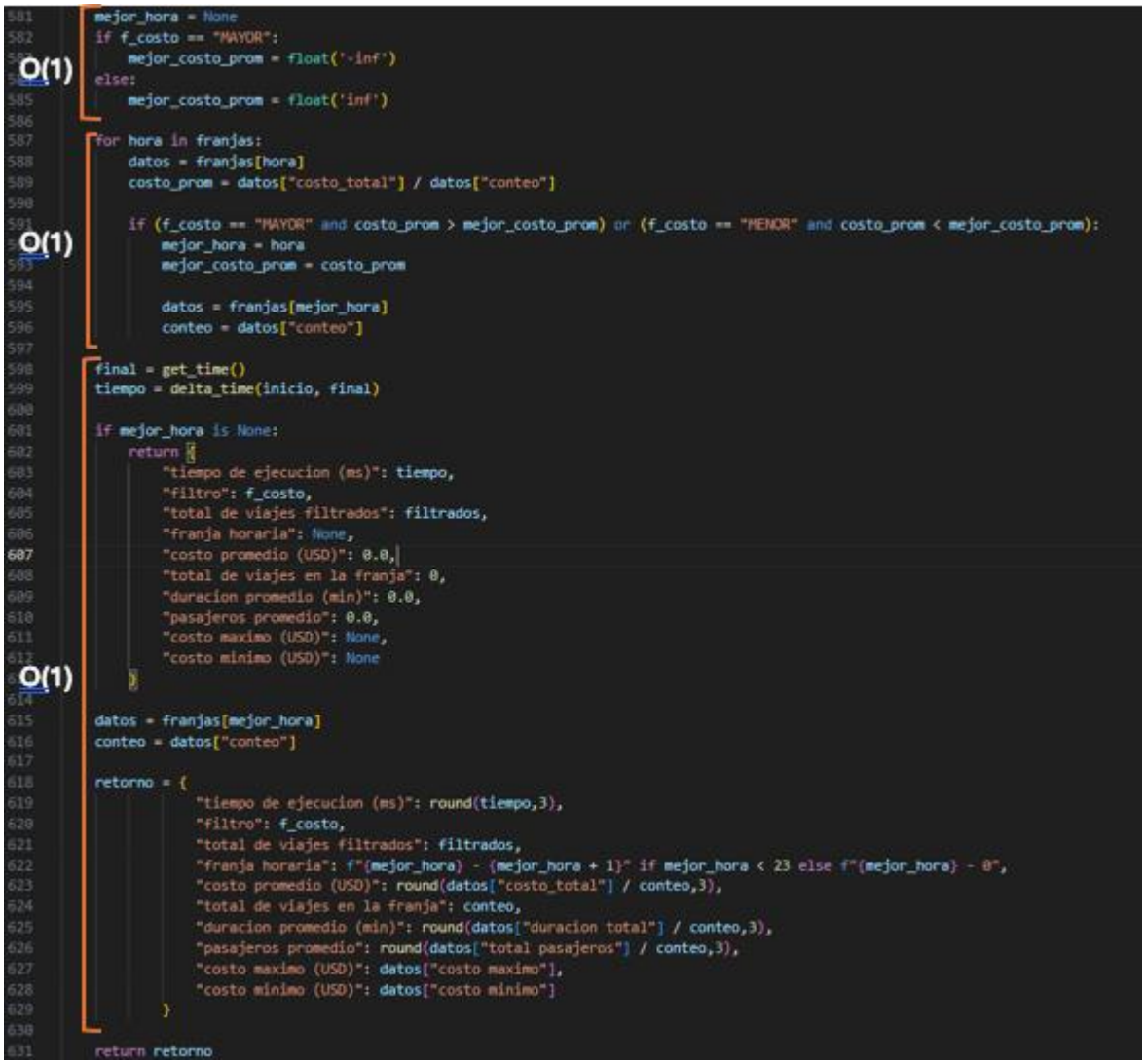
La instrucción más costosa en términos de complejidad temporal para esta función es su primer bucle que itera exactamente una vez por cada viaje almacenado en el catálogo (total viajes = n) O(n). Las primeras líneas simplemente son asignación de variables O(1).




```
def req_5(catalog, f_costo, f_inicial, f_final):

    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

    franjas = {}

    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

            hora = viaje["pickup_datetime"].hour
            if hora not in franjas:
                franjas[hora] = {
                    "costo_total": 0.0,
                    "conteo": 0,
                    "duracion total": 0.0,
                    "total pasajeros": 0,
                    "costo maximo": float('-inf'),
                    "costo minimo": float('inf')
                }
            duracion = (viaje["dropoff_datetime"] - viaje["pickup_datetime"]).total_seconds() / 60
            costo = float(viaje["total_amount"])
            pasajeros = viaje["passenger_count"]

            franjas[hora]["costo_total"] += costo
            franjas[hora]["conteo"] += 1
            franjas[hora]["duracion total"] += duracion
            franjas[hora]["total pasajeros"] += pasajeros

            if costo > franjas[hora]["costo maximo"]:
                franjas[hora]["costo maximo"] = costo
            if costo < franjas[hora]["costo minimo"]:
                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)**.



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"].

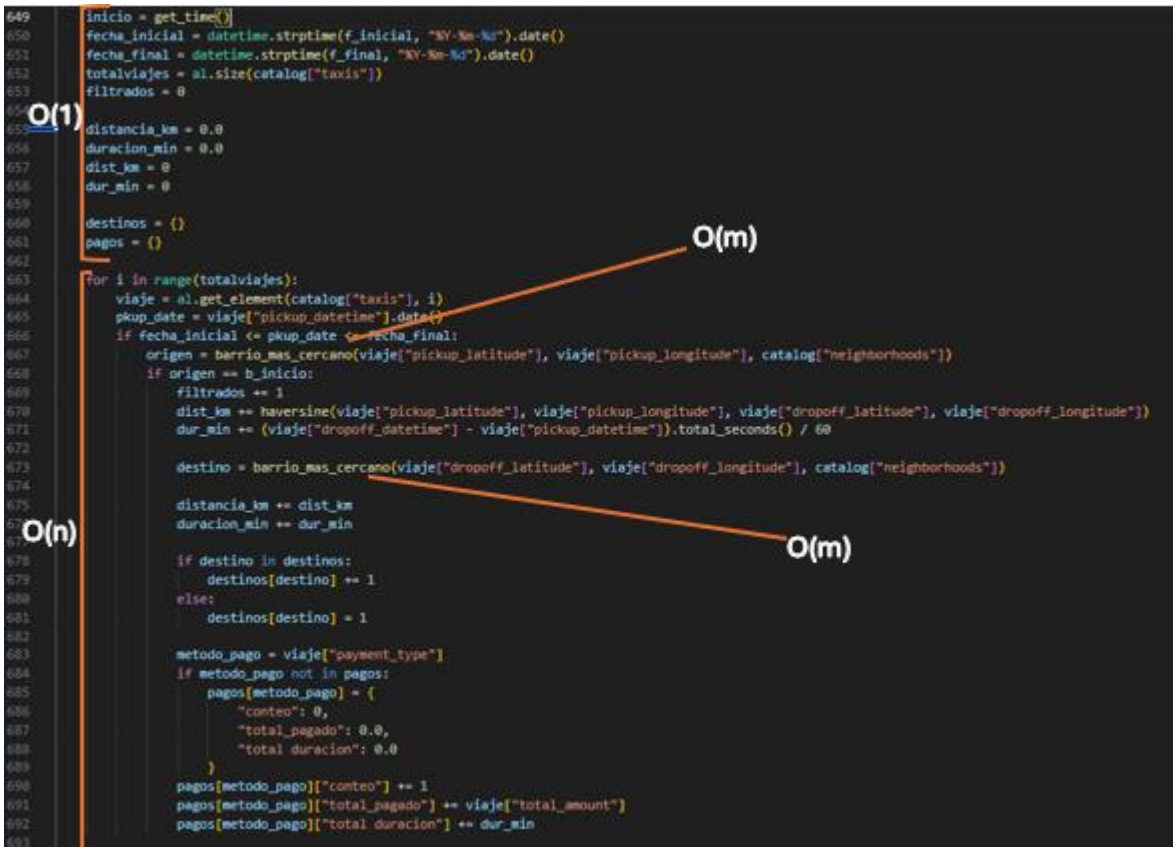
Sea m = número de barrios en catalog["neighborhoods"].

Sea k = número de valores en el diccionario destinos

Sea j = número de valores en el diccionario pagos


```
def req_6(catalog, b_inicio, f_inicial, f_final):

    def barrio_mas_cercano(lat, lon, lista_barrios):
        mejor_barrio = None
        mejor_distancia = float('inf')
        barrios = al.size(lista_barrios)

        for i in range(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
```

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




```
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

distancia_km = 0.0
duracion_min = 0.0
dist_km = 0
dur_min = 0

destinos = {}
pagos = {}

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:
        origen = barrio_mas_cercano(viaje["pickup_latitude"], viaje["pickup_longitude"], catalog["neighborhoods"])
        if origen == b_inicio:
            filtrados += 1
            dist_km += haversine(viaje["pickup_latitude"], viaje["pickup_longitude"], viaje["dropoff_latitude"], viaje["dropoff_longitude"])
            dur_min += (viaje["dropoff_datetime"] - viaje["pickup_datetime"]).total_seconds() / 60

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

            distancia_km += dist_km
            duracion_min += dur_min

            if destino in destinos:
                destinos[destino] += 1
            else:
                destinos[destino] = 1

            metodo_pago = viaje["payment_type"]
            if metodo_pago not in pagos:
                pagos[metodo_pago] = {
                    "conteo": 0,
                    "total_pagado": 0.0,
                    "total duracion": 0.0
                }
            pagos[metodo_pago]["conteo"] += 1
            pagos[metodo_pago]["total_pagado"] += viaje["total_amount"]
            pagos[metodo_pago]["total duracion"] += dur_min
```

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 condiciónales y con una complejidad O(m) como ya dijimos.

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.