Universidad de los Andes
Facultad de Ingeniería

Departamento de Ingeniería de Sistemas y Computación

Estructuras de Datos y Algoritmos

ISIS-1225

ABET Engineering Accreditation Commission

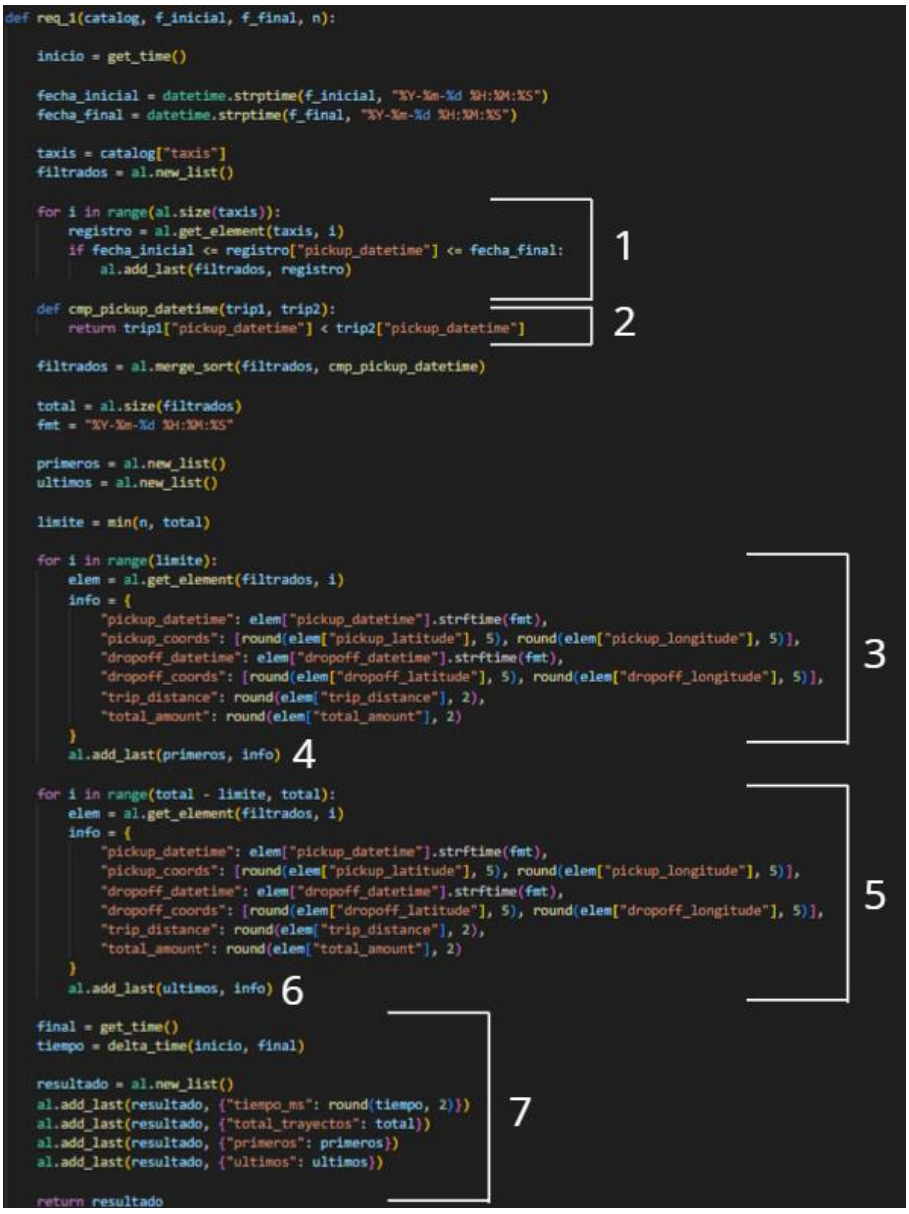
Análisis Reto-02

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

Samuel Felipe Tovar - sf.tovarp1@uniandes.edu.co - 202211625

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

- **Requerimiento <1>**




```python
def req_1(catalog, f_inicial, f_final, n):

    inicio = get_time()

    fecha_inicial = datetime.strptime(f_inicial, "%Y-%m-%d %H:%M:%S")
    fecha_final = datetime.strptime(f_final, "%Y-%m-%d %H:%M:%S")

    taxis = catalog["taxis"]
    filtrados = al.new_list()

    for i in range(al.size(taxis)):
        registro = al.get_element(taxis, i)
        if fecha_inicial <= registro["pickup_datetime"] <= fecha_final:
            al.add_last(filtrados, registro)

    def cmp_pickup_datetime(trip1, trip2):
        return trip1["pickup_datetime"] < trip2["pickup_datetime"]

    filtrados = al.merge_sort(filtrados, cmp_pickup_datetime)

    total = al.size(filtrados)
    fmt = "%Y-%m-%d %H:%M:%S"

    primeros = al.new_list()
    ultimos = al.new_list()

    limite = min(n, total)

    for i in range(limite):
        elem = al.get_element(filtrados, i)
        info = {
            "pickup_datetime": elem["pickup_datetime"].strftime(fmt),
            "pickup_coords": [round(elem["pickup_latitude"], 5), round(elem["pickup_longitude"], 5)],
            "dropoff_datetime": elem["dropoff_datetime"].strftime(fmt),
            "dropoff_coords": [round(elem["dropoff_latitude"], 5), round(elem["dropoff_longitude"], 5)],
            "trip_distance": round(elem["trip_distance"], 2),
            "total_amount": round(elem["total_amount"], 2)
        }
        al.add_last(primeros, info)

    for i in range(total - limite, total):
        elem = al.get_element(filtrados, i)
        info = {
            "pickup_datetime": elem["pickup_datetime"].strftime(fmt),
            "pickup_coords": [round(elem["pickup_latitude"], 5), round(elem["pickup_longitude"], 5)],
            "dropoff_datetime": elem["dropoff_datetime"].strftime(fmt),
            "dropoff_coords": [round(elem["dropoff_latitude"], 5), round(elem["dropoff_longitude"], 5)],
            "trip_distance": round(elem["trip_distance"], 2),
            "total_amount": round(elem["total_amount"], 2)
        }
        al.add_last(ultimos, info)

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

    resultado = al.new_list()
    al.add_last(resultado, {"tiempo_ms": round(tiempo, 2)})
    al.add_last(resultado, {"total_trayectos": total})
    al.add_last(resultado, {"primeros": primeros})
    al.add_last(resultado, {"ultimos": ultimos})

    return resultado
```

Este requerimiento se encarga de obtener trayectos en una franja fecha y tiempo de recogida, en primer lugar, se filtra por fecha de recogida franja inicial y final, después se organiza de forma ascendente en un merge sort según este mismo tiempo de recogida y se retorna el tiempo, el total de trayectos, los primeros y los últimos valores filtrados.

Entrada	Catálogo, fecha inicial, fecha final, tamaño inicial
Salidas	Tiempo ejecución, trayectos totales, información de cada uno de los datos (los primeros y los últimos)
Implementado (Sí/No)	Si. Implementado por Juan Esteban Espitia

Análisis de la complejidad:

Pasos	**Complejidad**
P1 Filtrar por f_inicial y f_final los datos	O(n)
P2 Cmp_func para los datos	O(1)
P3 Lista de los primeros datos	O(a)
P4 Add_last de paso 3	O(1)
Paso 3 y 4	O(a)
P5 y P6 Lista de los últimos datos y add_last	O(b)
P7 Return	O(1)
Merge-sort usado en la función	O(c log c)
Complejidad de la función:	***O(n) + O(a) + O(b) + O(c log c)***

Tiene este orden de complejidad debido a que dentro de la creación de los datos de tanto los valores iniciales y finales se recorre la cantidad de datos a y b, tomando en cuenta también el recorrido del catálogo y el merge, da la complejidad

- # Requerimiento <2>


```
def req_2(catalog, lat_inicial, lat_final, n):
    """
    Retorna el resultado del requerimiento 2
    """
    # TODO: Modificar el requerimiento 2
    inicio = get_time()

    taxis = catalog["taxis"]
    filtrados_lat = al.new_list()

    for i in range(al.size(taxis)):
        registro = al.get_element(taxis, i)
        pickup_lat = float(registro["pickup_latitude"])
        if lat_inicial <= pickup_lat <= lat_final:
            al.add_last(filtrados_lat, registro)

    def cmp_pickup_lat(trip1, trip2):
        lat1 = float(trip1["pickup_latitude"])
        lat2 = float(trip2["pickup_latitude"])
        lon1 = float(trip1["pickup_longitude"])
        lon2 = float(trip2["pickup_longitude"])

        if lat1 != lat2:
            return lat1 > lat2

        else:
            return lon1 > lon2

    filtrados_lat = al.merge_sort(filtrados_lat, cmp_pickup_lat)
    total = al.size(filtrados_lat)
    fmt = "%Y-%m-%d %H:%M:%S"

    primeros = al.new_list()
    ultimos = al.new_list()

    limite = min(n, total)

    for i in range(limite):
        elem = al.get_element(filtrados_lat, i)
        info = {
            "pickup_datetime": elem["pickup_datetime"].strftime(fmt),
            "pickup_coords": [round(elem["pickup_latitude"], 5), round(elem["pickup_longitude"], 5)],
            "dropoff_datetime": elem["dropoff_datetime"].strftime(fmt),
            "dropoff_coords": [round(elem["dropoff_latitude"], 5), round(elem["dropoff_longitude"], 5)],
            "trip_distance": round(elem["trip_distance"], 2),
            "total_amount": round(elem["total_amount"], 2)
        }
        al.add_last(primeros, info)

    for i in range(total - limite, total):
        elem = al.get_element(filtrados_lat, i)
        info = {
            "pickup_datetime": elem["pickup_datetime"].strftime(fmt),
            "pickup_coords": [round(elem["pickup_latitude"], 5), round(elem["pickup_longitude"], 5)],
            "dropoff_datetime": elem["dropoff_datetime"].strftime(fmt),
            "dropoff_coords": [round(elem["dropoff_latitude"], 5), round(elem["dropoff_longitude"], 5)],
            "trip_distance": round(elem["trip_distance"], 2),
            "total_amount": round(elem["total_amount"], 2)
        }
        al.add_last(ultimos, info)

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

    resultado = al.new_list()
    al.add_last(resultado, {"tiempo_es": round(tiempo, 2)})
    al.add_last(resultado, {"total_trayectos": total})
    al.add_last(resultado, {"primeros": primeros})
    al.add_last(resultado, {"ultimos": ultimos})

    return resultado
```

Esta función filtra los datos de los taxis a partir de la latitud inicial y la latitud final, al inicio se filtran, luego se comparan los datos para que queden en orden ascendente en un merge y luego se toman en una lista los primeros y los ultimos datos para asi ponerlos en el resultado final que es el retorno “resultado”.

Entrada	Catálogo, latitud inicial, latitud final, tamaño de la muestra
Salidas	Tiempo ejecución, trayectos totales, información de cada uno de los datos (los primeros y los últimos)
Implementado (Sí/No)	Si. Implementado por Samuel Felipe Tovar

Análisis de la complejidad temporal:

Pasos	**Complejidad**
P1 Filtrar por lat_inicial y lat_final los datos	O(n) Siendo n el tamaño del catalogo
P2 Cmp_func para los datos	O(1)
P3 Lista de los primeros datos	O(a) Siendo a el tamaño de primeros
P4 Lista de los últimos datos	O(b) Siendo b el tamaño de últimos
P5 Return	O(1)
Merge-sort usado en la función	O(c log c) Siendo c los datos de filtrados
Complejidad de la función:	***O(n) + O(a) + O(b) + O(c log c)***

Igual que los anteriores requerimientos ya que son similares, al tener en el peor caso el P3 y el P4, y el merge se suman las complejidades y queda la complejidad ya escrita en la tabla

- # Requerimiento <3>

```python
def req_3(catalog, dist_inicial, dist_final, n):

    inicio = get_time()
    taxis = catalog["taxis"]
    filtrados = al.new_list()

    for i in range(al.size(taxis)):
        viaje = al.get_element(taxis, i)
        distancia = viaje["trip_distance"]
        if dist_inicial <= distancia <= dist_final:
            al.add_last(filtrados, viaje)

    def cmp_dist_viaje(trip1, trip2):

        if trip1["trip_distance"] > trip2["trip_distance"]:
            return True
        elif trip1["trip_distance"] == trip2["trip_distance"]:
            return trip1["total_amount"] > trip2["total_amount"]
        else:
            return False

    filtrados = al.merge_sort(filtrados, cmp_dist_viaje)
    total = al.size(filtrados)
    fmt = "%Y-%m-%d %H:%M:%S"
    primeros = al.new_list()
    ultimos = al.new_list()
    limite = min(n, total)

    for i in range(limite):
        elem = al.get_element(filtrados, i)
        info = {
            "pickup_datetime": elem["pickup_datetime"].strftime(fmt),
            "pickup_coords": [round(elem["pickup_latitude"], 5), round(elem["pickup_longitude"], 5)],
            "dropoff_datetime": elem["dropoff_datetime"].strftime(fmt),
            "dropoff_coords": [round(elem["dropoff_latitude"], 5), round(elem["dropoff_longitude"], 5)],
            "trip_distance": round(elem["trip_distance"], 2),
            "total_amount": round(elem["total_amount"], 2)
        }
        al.add_last(primeros, info)

    for i in range(total - limite, total):
        elem = al.get_element(filtrados, i)
        info = {
            "pickup_datetime": elem["pickup_datetime"].strftime(fmt),
            "pickup_coords": [round(elem["pickup_latitude"], 5), round(elem["pickup_longitude"], 5)],
            "dropoff_datetime": elem["dropoff_datetime"].strftime(fmt),
            "dropoff_coords": [round(elem["dropoff_latitude"], 5), round(elem["dropoff_longitude"], 5)],
            "trip_distance": round(elem["trip_distance"], 2),
            "total_amount": round(elem["total_amount"], 2)
        }
        al.add_last(ultimos, info)

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

    resultado = al.new_list()
    al.add_last(resultado, {"tiempo_ms": round(tiempo, 2)})
    al.add_last(resultado, {"total_trayectos": total})
    al.add_last(resultado, {"primeros": primeros})
    al.add_last(resultado, {"ultimos": ultimos})

    return resultado
```

El requerimiento tres filtra por distancia de trayectos y luego organiza todos los datos con merge, luego inserta los datos ordenados en listas de primeros y últimos para después hacer el return.

Entrada	Catálogo, distancia inicial, distancia final, tamaño de la muestra
Salidas	Tiempo ejecución, trayectos totales, información de cada uno de los datos (los primeros y los últimos)
Implementado (Sí/No)	Si. Implementado por Fabio Andrés Salazar Ochoa

Análisis de la complejidad temporal:

Pasos	**Complejidad**
P1 Filtrar por dist_inicial y dist_final los datos	O(n) Siendo n el tamaño del catalogo
P2 Cmp_func para los datos	O(1)
P3 Lista de los primeros datos	O(a) Siendo a el tamaño de primeros
P4 Lista de los últimos datos	O(b) Siendo b el tamaño de últimos
P5 Return	O(1)
Merge-sort usado en la función	O(c log c) Siendo c los datos de filtrados
Complejidad de la función:	***O(n) + O(a) + O(b) + O(c log c)***

Igual que los anteriores requerimientos ya que son similares, al tener en el peor caso el P3 y el P4, y el merge se suman las complejidades y queda la complejidad ya escrita en la tabla.

- # Requerimiento <4>

```python
def req_4(catalog, fecha, momento, hora, n):

    inicio = get_time()
    taxis = catalog["taxis"]
    filtrados = al.new_list()

    fecha = datetime.strptime(fecha, "%Y-%m-%d").date()
    hora = datetime.strptime(hora, "%H:%M:%S").time()

    for i in range(al.size(taxis)):
        viaje = al.get_element(taxis, i)
        drop_dt = viaje["dropoff_datetime"]
        if drop_dt.date() == fecha:
            hora_viaje = drop_dt.time()
            if momento.lower() == "antes" and hora_viaje < hora:
                al.add_last(filtrados, viaje)
            elif momento.lower() == "despues" and hora_viaje > hora:
                al.add_last(filtrados, viaje)

    def cmp_drop_descendiente(v1, v2):
        return v1["dropoff_datetime"] > v2["dropoff_datetime"]

    filtrados = al.merge_sort(filtrados, cmp_drop_descendiente)
    total = al.size(filtrados)
    fmt = "%Y-%m-%d %H:%M:%S"
    primeros = al.new_list()
    ultimos = al.new_list()
    limite = min(n, total)

    for i in range(limite):
        elem = al.get_element(filtrados, i)
        info = {
            "pickup_datetime": elem["pickup_datetime"].strftime(fmt),
            "pickup_coords": [round(elem["pickup_latitude"], 5), round(elem["pickup_longitude"], 5)],
            "dropoff_datetime": elem["dropoff_datetime"].strftime(fmt),
            "dropoff_coords": [round(elem["dropoff_latitude"], 5), round(elem["dropoff_longitude"], 5)],
            "trip_distance": round(elem["trip_distance"], 2),
            "total_amount": round(elem["total_amount"], 2)
        }
        al.add_last(primeros, info)

    for i in range(total - limite, total):
        elem = al.get_element(filtrados, i)
        info = {
            "pickup_datetime": elem["pickup_datetime"].strftime(fmt),
            "pickup_coords": [round(elem["pickup_latitude"], 5), round(elem["pickup_longitude"], 5)],
            "dropoff_datetime": elem["dropoff_datetime"].strftime(fmt),
            "dropoff_coords": [round(elem["dropoff_latitude"], 5), round(elem["dropoff_longitude"], 5)],
            "trip_distance": round(elem["trip_distance"], 2),
            "total_amount": round(elem["total_amount"], 2)
        }
        al.add_last(ultimos, info)

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

    resultado = al.new_list()
    al.add_last(resultado, {"tiempo_ms": round(tiempo, 2)})
    al.add_last(resultado, {"total_trayectos": total})
    al.add_last(resultado, {"primeros": primeros})
    al.add_last(resultado, {"ultimos": ultimos})

    return resultado
```

1 2 3 4 5

Se filtra los elementos segun el tiempo de dropoff, momento exacto en el que se hace el dropoff y fecha, luego se usa merge para ordenarlos en orden ascendente y se separa en las listas de primeros y últimos para luego hacer el return.

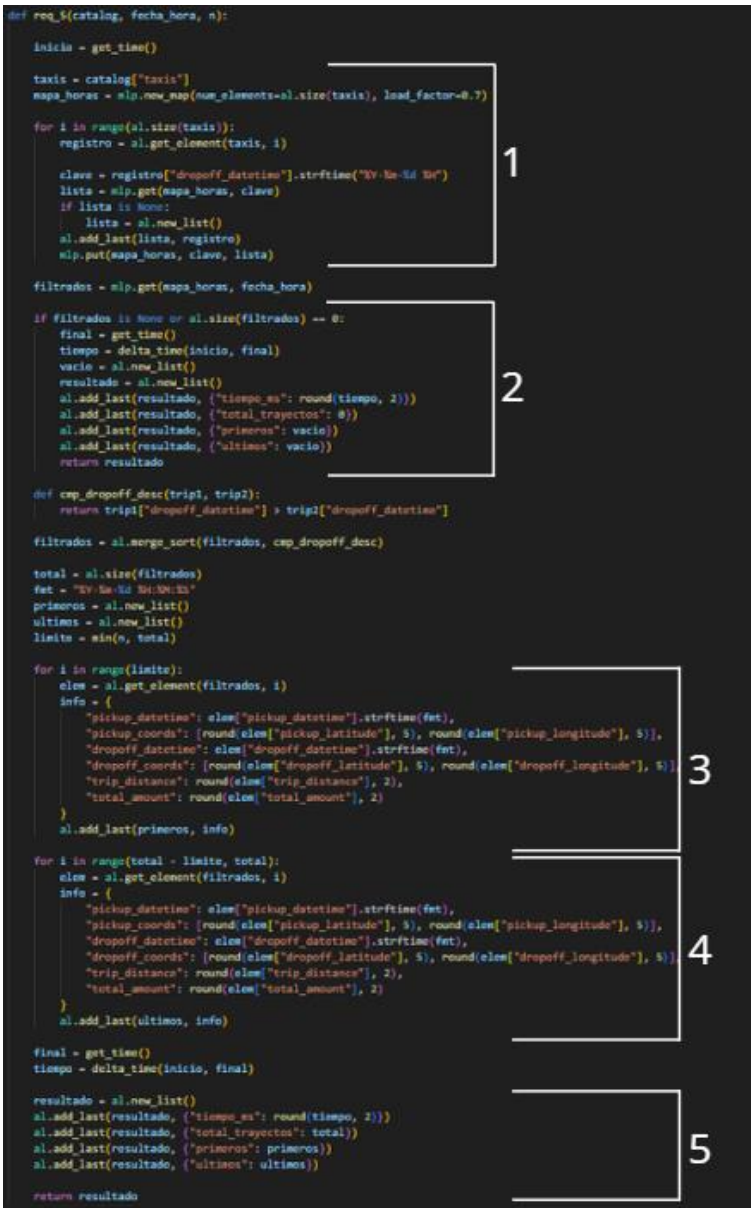
Entrada	Catálogo, fecha, momento, hora, tamaño de la muestra
Salidas	Tiempo ejecución, trayectos totales, información de cada uno de los datos (los primeros y los últimos)
Implementado (Sí/No)	Si

Análisis de la complejidad temporal:

Pasos	**Complejidad**
P1 Filtrado de datos	O(n) Siendo n el tamaño del catalogo
P2 Cmp_func para los datos	O(1)
P3 Lista de los primeros datos	O(a) Siendo a el tamaño de primeros
P4 Lista de los últimos datos	O(b) Siendo b el tamaño de últimos
P5 Return	O(1)
Merge-sort usado en la función	O(c log c) Siendo c los datos de filtrados
Complejidad de la función:	***O(n) + O(a) + O(b) + O(c log c)***

Igual que los anteriores requerimientos ya que son similares, al tener en el peor caso el P3 y el P4, y el merge se suman las complejidades y queda la complejidad ya escrita en la tabla.

- # Requerimiento <5>



Se hace por medio de tablas de hash, donde las claves son las fechas de los datos, los datos que pasan son filtrados según la fecha específica que se pide en el view, luego si no se filtra ninguno se retorna el resultado, si hay datos se divide en dos listas como en las anteriores funciones; primeros y ultimos y se da el retorno.

Entrada	Catálogo, fecha, tamaño de la muestra
Salidas	Tiempo ejecución, trayectos totales, información de cada uno de los datos (los primeros y los últimos)
Implementado (Sí/No)	Si

Análisis de la complejidad temporal:

Pasos	**Complejidad**
P1 Filtrar por fechas en mlp (put es o(1))	O(n) Siendo n el tamaño del catalogo
P2 Caso base (si no hay ningún dato filtrado)	O(1)
P3 Lista de los primeros datos	O(a) Siendo a el tamaño de primeros
P4 Lista de los últimos datos	O(b) Siendo b el tamaño de últimos
P5 Return	O(1)
Merge-sort usado en la función	O(c log c) Siendo c los datos de filtrados
Complejidad de la función:	***O(n) + O(a) + O(b) + O(c log c)***

En esta función hay un giro que hace que sea diferente a los demás, sin embargo, tiene la misma complejidad temporal, en esta función se implementa linear probing, donde los datos que cumplen con el filtro entran al mapa que se crea en el inicio de la función. Pero como dije anteriormente mantiene la misma complejidad temporal que las anteriores funciones.

- # Requerimiento <6>

```python
def req_6(catalog, barrio, hora_inicial, hora_final, n):

    inicio = get_time()

    num_barrios = al.size(catalog["neighborhoods"])
    mapa_barrios = mlp.new_map(num_elements=num_barrios, load_factor=0.7)

    taxis = catalog["taxis"]
    total_taxis = al.size(taxis)
    barrios = catalog["neighborhoods"]
    total_barrios = al.size(barrios)

    # Indexar los viajes por barrio
    # Para cada viaje, se determina a qué barrio pertenece su punto de recogida
    for i in range(total_taxis):
        viaje = al.get_element(taxis, i)
        pickup_lat = viaje["pickup_latitude"]
        pickup_lon = viaje["pickup_longitude"]

        # Buscar el barrio más cercano (menor distancia Haversine)
        barrio_encontrado = None
        distancia_minima = float('inf')

        for j in range(total_barrios):
            barrio = al.get_element(barrios, j)
            distancia = haversine(pickup_lat, pickup_lon, barrio["latitude"], barrio["longitude"])

            if distancia < distancia_minima:
                distancia_minima = distancia
                barrio_encontrado = barrio["neighborhood"]

        if barrio_encontrado is not None:
            lista_viajes = mlp.get(mapa_barrios, barrio_encontrado)
            if lista_viajes is None:
                lista_viajes = al.new_list()

            al.add_last(lista_viajes, viaje)
            mlp.put(mapa_barrios, barrio_encontrado, lista_viajes)

    # Obtener los viajes del barrio solicitado
    viajes_barrio = mlp.get(mapa_barrios, barrio)

    # Si el barrio no existe o no hay viajes registrados
    if viajes_barrio is None or al.size(viajes_barrio) == 0:
        final = get_time()
        resultado = al.new_list()
        al.add_last(resultado, {"tiempo_ms": round(tiempo, 2)})
        al.add_last(resultado, {"total_trayectos": 0})
        al.add_last(resultado, {"primeros": primeros})
        al.add_last(resultado, {"ultimos": ultimos})
        return resultado

    hora_inicial = int(hora_inicial)
    hora_final = int(hora_final)

    # Filtrar viajes dentro del rango horario
    filtrados = al.new_list()
    total_viajes = al.size(viajes_barrio)

    for i in range(total_viajes):
        v = al.get_element(viajes_barrio, i)
        hora = v["pickup_datetime"].hour
        if hora_inicial <= hora < hora_final:
            al.add_last(filtrados, v)

    # Ordenar del más antiguo al más reciente
    def cmp_fecha_asc(a, b):
        return a["pickup_datetime"] < b["pickup_datetime"]

    filtrados = al.merge_sort(filtrados, cmp_fecha_asc)

    # Seleccionar primeros y últimos N
    total_filtrados = al.size(filtrados)
    limite = n if n < total_filtrados else total_filtrados
    primeros = al.new_list()
    ultimos = al.new_list()
    fmt = "%Y-%m-%d %H:%M:%S"

    # Primeros N
    for i in range(limite):
        v = al.get_element(filtrados, i)
        info = {
            "pickup_datetime": v["pickup_datetime"].strftime(fmt),
            "pickup_coords": [round(v["pickup_latitude"], 5), round(v["pickup_longitude"], 5)],
            "dropoff_datetime": v["dropoff_datetime"].strftime(fmt),
            "dropoff_coords": [round(v["dropoff_latitude"], 5), round(v["dropoff_longitude"], 5)],
            "trip_distance": round(v["trip_distance"], 2),
            "total_amount": round(v["total_amount"], 2)
        }
        al.add_last(primeros, info)

    # Últimos N
    for i in range(total_filtrados - limite, total_filtrados):
        v = al.get_element(filtrados, i)
        info = {
            "pickup_datetime": v["pickup_datetime"].strftime(fmt),
            "pickup_coords": [round(v["pickup_latitude"], 5), round(v["pickup_longitude"], 5)],
            "dropoff_datetime": v["dropoff_datetime"].strftime(fmt),
            "dropoff_coords": [round(v["dropoff_latitude"], 5), round(v["dropoff_longitude"], 5)],
            "trip_distance": round(v["trip_distance"], 2),
            "total_amount": round(v["total_amount"], 2)
        }
        al.add_last(ultimos, info)

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

    resultado = al.new_list()
    al.add_last(resultado, {"tiempo_ms": round(tiempo, 2)})
    al.add_last(resultado, {"total_trayectos": total_filtrados})
    al.add_last(resultado, {"primeros": primeros})
    al.add_last(resultado, {"ultimos": ultimos})

    return resultado
```

En esta función se filtra según barrio, hora inicial, hora final y tamaño de muestra, en la cual se agarra la latitud según el barrio y se filtra con los demás criterios.

Entrada	Catálogo, barrio, hora inicial, hora final, muestra
Salidas	Tiempo ejecución, trayectos totales, información de cada uno de los datos (los primeros y los últimos)
Implementado (Sí/No)	Si

Análisis de la complejidad temporal:

Pasos	**Complejidad**
P1 Filtrar por criterios (si tenemos en cuenta todos los implementados)	O(n) Siendo n el tamaño del catalogo
P2 Caso base (si no hay ningún dato filtrado)	O(1)
P3 Lista de los primeros datos	O(a) Siendo a el tamaño de primeros
P4 Lista de los últimos datos	O(b) Siendo b el tamaño de últimos
P5 Return	O(1)
Merge-sort usado en la función	O(c log c) Siendo c los datos de filtrados
Complejidad de la función:	***O(n) + O(a) + O(b) + O(c log c)***

Es similar a la anterior función, como siempre con la misma complejidad, pero con más criterios, tales como la función havershine O(1) la cual saca la latitud según el barrio y demás comparaciones que se hacen en la función, pero en general se puede asumir que llega a la misma complejidad.