Estructuras de Datos y Algoritmos
Reto 4

Sección 4 - Grupo 03

Integrantes

- Camilo Quiñones Buitrago - ca.quinones@uniandes.edu.co - 201814870
- Juan Pablo Bautista - jp.bautista@uniandes.edu.co – 201812761

Análisis de Resultados

Los parámetros utilizados para cada ensayo de cada requerimiento se muestran a continuación.

Requerimiento	Parámetros
1	No se requieren
2	**Estación de inicio:** Coxwell Ave / Plains Rd **Tiempo máximo:** 5000 s **Mínima cantidad de estaciones:** 3 **Número de alternativas a visualizar:** 5
3	No se requieren
4	**Estación de inicio:** Exhibition GO (Atlantic Ave) **Estación de destino:** Donlands Station
5	**Fecha de inicio:** 03/25/2021 **Fecha de fin:** 07/01/2021
6	**Id de la bicicleta:** 5273.0

Análisis de Complejidad

Para empezar, hay que aclarar varias cosas para poder entender de mejor manera la complejidad conseguida, para procesar los datos solo se tomaron en cuenta los viajes que tenian todos los datos requeridos, o sea la información completa, esto llevó a que haya un menor número de datos, pero estos reflejen de mejor manera el funcionamiento del sistema al ser cada uno más informativo. Este cambio en el número de datos llevó también a que los resultados de los requerimientos sean distintos a los reportados en los distintos ejemplos.

Como se nos fue recomendado en la entrega pasada, esta vez solo se analizarán las complejidades de la parte de model de los distintos requerimientos.

Requerimiento 1:
Empezaremos con la función

```
def orderVerticesByEdgeOuts(analyzer):
    mapStations=analyzer["stations"]
    lstStations=m.keySet(mapStations)
    vertices=lt.newList("ARRAY_LIST")
    for station in lt.iterator(lstStations):
        stationInfo=addNumOutTrips(station,mapStations)
        lt.addLast(vertices,stationInfo)
    orderedVert=merge.sort(vertices,cmpVertByEdgeOuts)
    first5=getFirst5(orderedVert)
    for stationInfo in lt.iterator(first5):
        getUserTypes(stationInfo)
        mostFrequentDate(stationInfo)
        mostFrequentHour(stationInfo)
    return first5
```

Primero se consiguen el mapa de estaciones y la lista de las mismas, operaciones O(1), y por último se crea una lista para los vértices a ordenar. Entramos en un ciclo en el que primero se usa la función addNumOutTrips la cual es la siguiente:

```
def addNumOutTrips(station,mapStations):
    entry=m.get(mapStations,station)
    value=me.getValue(entry)
    stationInfo=value.copy()
    stationInfo["numOutTrips"]=(lt.size(stationInfo["outTrips"]))
    return stationInfo
```

En esta función primero conseguimos la entrada de un mapa, después conseguimos el valor de esta entrada, luego creamos la información de la estación, y a esta le definimos el parámetro numOutTrips como el tamaño de outTrips que hay en stationInfo. Se puede ver que esta complejidad es O(1) al tratarse de búsquedas dentro de un mapa.

Y ya que se usa un addLast podemos concluir que este ciclo es O(N) siendo N el número de estaciones.

Luego hacemos mergeSort sobre los vértices para así ordenarlos, lo que nos lleva a una complejidad de O(N log N), luego hacemos funciones para la correcta visualización de los datos, en donde conseguimos los primeros 5 y sobre estos 5 hacemos un ciclo en donde usamos primero getUserTypes

```
def getUserTypes(stationInfo):
    stationInfo["Annual Member"]=0
    stationInfo["Casual Member"]=0
    lstOutTrips=stationInfo["outTrips"]
    for trip in lt.iterator(lstOutTrips):
        typeMember=trip["User Type"]
        stationInfo[typeMember]+=1
    return stationInfo
```

Aquí conseguimos la información de los viajes, para luego iterar sobre esta y así conseguir el tipo de miembro, operación O(N).

```
def mostFrequentDate(stationInfo):
    mapTripsOut=m.newMap(800,maptype="CHAINING")
    for i in lt.iterator(stationInfo["outTrips"]):
        startTrip=str(datetime.strptime(i["Start Time"],"%m/%d/%Y %H:%M").date())
        if not m.contains(mapTripsOut,startTrip):
            entry={"numTrips":1}
            m.put(mapTripsOut,startTrip,entry)
        else:
            value=me.getValue(m.get(mapTripsOut,startTrip))
            value["numTrips"]+=1
    stationInfo["mostFrequentDate"]=getMostFrequent(mapTripsOut)
    return stationInfo
```

Después utilizamos mostFreqeuntDate en donde primero creamos un mapa para los viajes, y luego iteramos sobre los viajes de salida, para así contar la fecha que se repite más, haciendo que esto sea O(N)

```
def mostFrequentHour(stationInfo):
    mapHourOut=m.newMap(800,maptype="CHAINING")
    for i in lt.iterator(stationInfo["outTrips"]):
        startTrip=datetime.strptime(i["Start Time"],"%m/%d/%Y %H:%M").hour
        startInverval=str(startTrip)+":"+"00"+" - "+str(startTrip)+":"+"59"
        if not m.contains(mapHourOut,startInverval):
            entry={"numTrips":1}
            m.put(mapHourOut,startInverval,entry)
        else:
            value=me.getValue(m.get(mapHourOut,startInverval))
            value["numTrips"]+=1
    stationInfo["mostFrequentHour"]=getMostFrequent(mapHourOut)
    return stationInfo
```

Aquí se sigue la misma algorítmica que para la parte anterior por lo que también es O(N).
Con esto llegamos a que se hacen 3 operaciones O(N) en un ciclo de 5 iteraciones por lo que se concluye que la complejidad de este requerimiento es de O(N log N).

Requerimiento 2

```
def getVertFromNameStation(analyzer,name):
    mapStations=analyzer["stations"]
    stations=m.keySet(mapStations)
    for station in lt.iterator(stations):
        stationName=getNameFromVert(station)
        if stationName==name:
            return station
    return None
```

Primero tenemos esta función en donde conseguimos el vértice con el nombre que queremos de forma iterativa usando la función getNameFromVert

```
def getNameFromVert(vertName):
    name=""
    for i in range(5,len(vertName)):
        name+=vertName[i]
    return name
```

Que simplemente busca el nombre de cada vertice en un ciclo con una complejidad de O(N) siendo N el número de letras de la estación.
Haciendo esta función O(N*N)
Seguimos con la función

```
def filterPathsByTimeAndNumStations(analyzer, maxTime, minStations):
    minStations=float(minStations)
    maxTime=float(maxTime)
    stations=gr.vertices(analyzer["connections"])
    lstPaths=lt.newList("ARRAY_LIST")
    for i in lt.iterator(stations):
        existPath=dfs.hasPathTo(analyzer["search"],i)
        if existPath:
            path=dfs.pathTo(analyzer["search"],i)
            timePath=getTimePath(analyzer,path)
            numStations=stack.size(path)-1
            if timePath<=maxTime and numStations>=minStations:
                lt.addLast(lstPaths,path)
    return lstPaths
```

Aquí filtramos según un mínimo de estaciones y máximo de tiempo, primero conseguimos las estaciones del grafo, luego creamos una lista de caminos, para luego iterar sobre las estaciones, y en caso de existir la estación se filtra según los criterios dados y se retornan todas las estaciones entre el filtro. Esto hace que esta función tenga una complejidad de O(N), y al ser esta la última.

Requerimiento 3:

```
def getComponents(analyzer):
    SCC=analyzer["components"]
    grafo=analyzer["connections"]
    components=m.newMap(numelements=200,maptype="PROBING")
    for station in lt.iterator(gr.vertices(grafo)):
        sccId=m.get(SCC["idscc"],station)["value"]   # id del SCC al que pertenece la estación
        existComponent=m.contains(components,sccId)
        if existComponent:
            entry=m.get(components,sccId)
            value=me.getValue(entry)
            lt.addLast(value["stations"],station)
        else:
            entry={"sccId":sccId,"stations":None}
            entry["stations"]=lt.newList("ARRAY_LIST")
            lt.addLast(entry["stations"],station)
            m.put(components,sccId,entry)
    return components
```

Primero tenemos la función getComponents, en donde primero sacamos SCC, luego el grafo, y creamos un mapa que contiene a los componentes, para después hacer un ciclo en donde iteramos sobre los vértices del grafo, en donde primero vemos si existe el vértice dentro sccId, en caso de que exista añadimos el vertice dentro de nuestro mapa, y en caso de no existir creamos uno nuevo y lo añadimos. Con esto la complejidad es O(N).

```
def mostInOutTrips(stations,mapStations):
    stationMostOut=""
    stationMostIn=""
    outTrips=0
    inTrips=0
    for j in lt.iterator(stations):
        numOutTrips, numInTrips=getNumTrips(mapStations,j)
        if numInTrips>inTrips:
            inTrips=numInTrips
            stationMostIn=j
        if numOutTrips>outTrips:
            outTrips=numOutTrips
            stationMostOut=j
    return stationMostOut,stationMostIn
```

Esta otra función nos calcula la estación con más viajes de ida y vuelta, y aquí se hace una comparativa entre estaciones en donde se devuelve el nombre de la estación con más recorridos, así como cuantos recorridos son estos, esto tiene una complejidad de O(N) al ser un recorrido.

```
def getNumTrips(mapStations,station):
    entry=m.get(mapStations,station)
    value=me.getValue(entry)
    stationInfo=value
    numOutTrips=lt.size(stationInfo["outTrips"])
    numInTrips=lt.size(stationInfo["inTrips"])
    return numOutTrips,numInTrips
```

Vemos que esta es una operación constante por lo que decimos que es O(1).
Con esto concluimos que la complejidad de este requerimiento es de O(N).

Requerimiento 4

```
def minimumCostPaths(analyzer,startStation):
    startStation=getVertFromNameStation(analyzer,startStation)
    analyzer["minPaths"]=djk.Dijkstra(analyzer["connections"],startStation)
    return analyzer
```

En esta función utilizamos dijkstra, lo que lleva a que esta sea de complejidad O(NlogN).

```
def getMinimumPath(analyzer,endStation):
    endStation=getVertFromNameStation(analyzer,endStation)
    path=djk.pathTo(analyzer["minPaths"],endStation)
    return path
```

Aquí vemos el pathTo a cierta estación de destino esto con complejidad O(N).

```
def getTimeAndInfoPath(analyzer,path):
    path=fixStack(path)
    lstInfoPath=lt.newList("ARRAY_LIST")
    infoPath={"time":None,"info":None}
    time=getTimePath(analyzer,path)
    infoPath["time"]=str(round(time/60,2))+" min"+"\n"+str(round(time,2))+" s"
    infoPath["info"]=createTimePathInfo(analyzer,path)
    lt.addLast(lstInfoPath,infoPath)
    return lstInfoPath
```

Esta es otra función O(1) ya que solo se hacen operaciones de búsqueda y añadir elementos a una lista.

```
def createTimePathInfo(analyzer,path):
    pila=path.copy()
    grafo=analyzer["connections"]
    timeInfoPath=""
    startVert=None
    endVert=None
    count=True
    while not stack.isEmpty(pila):
        if count:
            startVert=stack.pop(pila)
        else:
            endVert=stack.pop(pila)
            edge=gr.getEdge(grafo,startVert,endVert)
            time=edge["weight"]
            timeInfoPath=addInfoStation(analyzer,timeInfoPath,startVert,endVert,time)
            startVert=endVert
        count=False
    return timeInfoPath
```

Aquí se busca la información del camino usando un stack, e iterando sobre este, para al final devolver la información del camino, esto con c omplejidad O(N) al tener el ciclo.
Esto nos lleva a que la complejidad del requerimiento sea O(N).

Requerimiento 5

```
def getTripsByDate(analyzer,startDate,endDate):
    trips=lt.newList("ARRAY_LIST")
    totalTime=0
    startDate=startDate+" 00:00"
    endDate=endDate+" 24:59"
    for i in lt.iterator(analyzer["completeTrips"]):
        if i["Start Time"]>=startDate and i["End Time"]<=endDate:
            lt.addLast(trips,i)
            totalTime+=int(i["Trip  Duration"])
    return trips,totalTime
```

Aquí buscamos viajes según cierto límite de fechas, se ve que utilizamos un ciclo sobre los viajes completados, y luego aplicamos el filtro, lo que hace que esta operación sea O(N)

```
def getFrequentStations(trips):
    mapTripsOut=m.newMap(800,maptype="CHAINING")
    mapTripsIn=m.newMap(800,maptype="CHAINING")
    for i in lt.iterator(trips):
        origin=i["Start Station Id"]+"-"+i["Start Station Name"]
        dest=i["End Station Id"]+"-"+i["End Station Name"]
        if not m.contains(mapTripsOut,origin):
            entry={"numTrips":1}
            m.put(mapTripsOut,origin,entry)
        else:
            numTrips=me.getValue(m.get(mapTripsOut,origin))
            numTrips["numTrips"]+=1
        if not m.contains(mapTripsIn,dest):
            entry={"numTrips":1}
            m.put(mapTripsIn,dest,entry)
        else:
            numTrips=me.getValue(m.get(mapTripsIn,dest))
            numTrips["numTrips"]+=1
    originStation=getMostFrequent(mapTripsOut)
    destStation=getMostFrequent(mapTripsIn)
    return originStation,destStation
```

Aquí buscamos las estaciones más recientes, y vemos que hacemos un ciclo sobre los viajes en donde vemos si tiene un origen definido para poder contarlo o no, y para ver si tiene algún destino para poder contarlo o no, y por último retornamos la estación de origen más reciente, y la estación de destino más frecuente ambos con una complejidad de O(N).

```
def getFrequentHours(trips):
    mapHourOut=m.newMap(100,maptype="CHAINING")
    mapHourIn=m.newMap(100,maptype="CHAINING")
    for i in lt.iterator(trips):
        startTrip=datetime.strptime(i["Start Time"],"%m/%d/%Y %H:%M").hour
        endTrip=datetime.strptime(i["End Time"],"%m/%d/%Y %H:%M").hour
        startInverval=str(startTrip)+":"+"00"+" - "+str(startTrip)+":"+"59"
        endInverval=str(endTrip)+":"+"00"+" - "+str(endTrip)+":"+"59"
        if not m.contains(mapHourOut,startInverval):
            entry={"numTrips":1}
            m.put(mapHourOut,startInverval,entry)
        else:
            value=me.getValue(m.get(mapHourOut,startInverval))
            value["numTrips"]+=1
        if not m.contains(mapHourIn,endInverval):
            entry={"numTrips":1}
            m.put(mapHourIn,endInverval,entry)
        else:
            value=me.getValue(m.get(mapHourIn,endInverval))
            value["numTrips"]+=1
    mostStartHour=getMostFrequent(mapHourOut)
    mostEndHour=getMostFrequent(mapHourIn)
    return mostStartHour,mostEndHour
```

En esta parte se sigue la misma algorítmica que en la anterior, por lo que podemos concluir que la complejidad es de O(N).
Con lo visto anteriormente, podemos concluir que este requerimiento tiene complejidad O(N).

Requerimiento 6:
La única función que usamos es la siguiente:

```
def getTripsByBikeId(analyzer,bikeId):
    trips=lt.newList("ARRAY_LIST")
    numTrips=0
    totalTime=0
    for i in lt.iterator(analyzer["completeTrips"]):
        if float(i["Bike Id"])==float(bikeId):
            numTrips+=1
            totalTime+=int(i["Trip  Duration"])
            lt.addLast(trips,i)
    return trips,numTrips,totalTime
```

En esta función buscamos los viajes por medio de los Ids de las bicicletas, esto lo hacemos con un ciclo en donde contamos el número de viajes, el tiempo total, y los viajes que cumplen con esta Id, este ciclo tiene una complejidad de O(N) por lo que la complejidad del requerimiento es O(N).