OBSERVACIONES RETO 4

Manuela Pacheco Malagón - m.pachecom2@uniandes.edu.co – 202112410
Sara Sanchez Fernandez – sj.sanchez@uniandes.edu.co - 202113012

Análisis de complejidad

Requerimiento 1:

La complejidad de las funciones de este requerimiento es O(1) puesto que, en primer lugar, se crea una lista de los vértices pertenecientes a un grafo previamente construido en la carga de datos, con la cual consecuentemente, se hace uso de la función que buscar los adyacentes a cada vértice perteneciente a la lista anterior. Sin embargo, la función de ordenamiento de los datos en shell sort toma O(N*log(N)). Por último, para poder realizar el print de los resultados, se realiza un m.get para encontrar la información específica de los vértices anteriores, esta función tiene de igual forma una complejidad de O(1). Teniendo en cuenta lo anterior, la complejidad del requerimiento es de O(N*log(N)).

```
def popularStations(catalog):
    vertexlist = gr.vertices(catalog["tripsxamount"])
    newvertex = lt.newList()

    for vertex in lt.iterator(vertexlist):
        dict = {"Station ID": vertex,
                "Departure Trips": tripsDepartureCount(catalog["tripsxamount"], vertex),
                "Out Degree": gr.outdegree(catalog["tripsxamount"], vertex)}

        lt.addLast(newvertex, dict)

    sa.sort(newvertex, cmpStationDepartures)
    size = lt.size(newvertex)
```

Requerimiento 2:

A lo largo de este requerimiento se hace uso de funciones como m.get, para encontrar el id referente a la estación indicada y gr.vertices, para obtener los vertices pertenecientes al grafo no dirigido construido en la carga de datos, las cuales tienen una complejidad de O(1). Sin embargo, lo que se usó principalmente para poder encontrar los paseos turisticos fue el algoritmo de Dijkstra cuya complejidad es de O(ElogV). Teniendo en cuenta lo anterior, la complejidad de este requerimiento es O(ElogV)

```
def planTrips(catalog, name, available, minst, maxrt):
    name = formatStr(name)
    name = name.lower()

    stationid = mp.get(catalog["stationnamexid"], name)["value"]

    initSearch("searchdjk", catalog, "tripsxdurationND", stationid)
    searchdjk = catalog["searchdjk"]["search"]
```

```
    vertexlist = gr.vertices(catalog["tripsxdurationND"])
    sa.sort(vertexlist, cmpVertex)
    newvertex = lt.newList()
```

```
def initSearch(type, catalog, graphname, vertex=None):
    if (catalog[type]["search"] == None) or (catalog[type]["graphname"] != graphname) or
(catalog[type]["vertex"] != vertex):
        if type == "searchdjk":
            catalog[type]["search"] = djk.Dijkstra(catalog[graphname], vertex)
        elif type == "searchscc":
            catalog[type]["search"] = scc.KosarajuSCC(catalog[graphname])
        elif type == "MST":
            catalog[type]["search"] = prim.PrimMST(catalog[graphname], vertex)
        elif type == "bfs":
            catalog[type]["search"] = bfs.BreadhtFisrtSearch(catalog[graphname], vertex)
        elif type == "dfs":
            catalog[type]["search"] = dfs.DepthFirstSearch(catalog[graphname], vertex)

        catalog[type]["graphname"] = graphname
        catalog[type]["vertex"] = vertex

    return catalog
```

Requerimiento 3:

Este requerimiento funciona de manera similar al requerimiento 2, para poder organizar los datos y encontrar los componentes fuertemente conectados se hizo uso del algoritmo de kosaraju, la complejidad asociada al mismo es de O(V+E). Por otro lado, en el requerimiento también se incluyen funciones complementarias como m.get, mp.keySet y mp.contains para poder definir los componentes presentes en el grafo, a estos anexos se les asocia una complejidad de O(1). De esta forma, la complejidad del requerimiento es de O(V+E)

```
def stronglyConnected(catalog):
    initSearch("searchscc", catalog, "tripsxamount")
    searchscc = catalog["searchscc"]["search"]
    keylist = mp.keySet(searchscc["idscc"])

    mapa = mp.newMap(numelements=100, maptype="PROBING")

    for key in lt.iterator(keylist):
        component = mp.get(searchscc["idscc"], key)["value"]
        if mp.contains(mapa, component):
            entry = mp.get(mapa, component)["value"]
        else:
            entry = {"Component number": component,
                     "Stations IDs": lt.newList()}
        lt.addLast(entry["Stations IDs"], key)
        mp.put(mapa, component, entry)
```

Requerimiento 4:
Para este requerimiento, al igual que el requerimiento 2 se hace uso del algoritmo Dijkstra, como se dijo anteriormente su complejidad es de O(ElogV). Al comparar la complejidad previa con las funciones complementarias, las cuales tienen una complejidad de O(1), se puede decir que la complejidad temporal de este requerimeinto es de O(ElogV).

```
def fastRoute(catalog, origen, destino):
    origen = formatStr(origen)
    origen = origen.lower()
    destino = formatStr(destino)
    destino = destino.lower()

    originid = mp.get(catalog["stationnamexid"], origen)["value"]
    destinationid = mp.get(catalog["stationnamexid"], destino)["value"]

    initSearch("searchdjk", catalog, "tripsxduration", originid)
    searchdjk = catalog["searchdjk"]["search"]

    cost = round(djk.distTo(searchdjk, destinationid))
    path = djk.pathTo(searchdjk, destinationid)
    size = stack.size(path)
```

Requerimiento 5:
Este requerimiento se compone de la creación de un árbol para poder filtrar la información y así crear un grafo referente a las fechas indicadas y poder encontrar de la mejor forma los elementos pedidos, tales como la estación más popular de salidas o la duración total del viaje Para esto a lo largo del requerimiento se ejecutan funciones que mantienen un complejidad de O(1). De esta forma, la complejidad temporal del requerimiento es de O(1).

```
def dateGraph(catalog, tripslist):
    graph = gr.newGraph(datastructure="ADJ_LIST",
                        directed=True,
                        size=639200,
                        comparefunction=cmpStations)

    mapa = mp.newMap(numelements=30,
                     maptype="PROBING")

    for trip in lt.iterator(tripslist):
        origin = int(float(trip["Start Station Id"]))
        origin = checkID(catalog, trip["Start Station Name"], origin)
        destination = int(float(trip["End Station Id"]))
        destination = checkID(catalog, trip["End Station Name"], destination)
        connectionid = (origin, destination)

        if not gr.containsVertex(graph, origin):
            gr.insertVertex(graph, origin)
```

```
        if not gr.containsVertex(graph, destination):
            gr.insertVertex(graph, destination)

        exists1 = gr.getEdge(graph, origin, destination)

        if exists1 != None:
            entry1 = mp.get(mapa, connectionid)["value"]
        else:
            gr.addEdge(graph, origin, destination)
            entry1 = {"connectionid": connectionid,
                      "origin": origin,
                      "destination": destination,
                      "tripsid": lt.newList(),
                      "tripsdurations": lt.newList(),
                      "tripsamount": None}
            mp.put(mapa, connectionid, entry1)
```

Requerimiento 6:
Para el poder planear el mantenimiento de las bicicletas, el requerimiento se compone de la creación de un grafo para la cicla indicada y a partir de el se realizan los conteos para encontrar la información requerida. Al igual que sucede con el requerimiento 5, la complejidad temporal de este algoritmo es O(1)

Requerimiento 7:
La complejidad depende de esta función, cuya complejidad es O(N) por el ciclo de esta función, dónde N es la lista de viajes dentro del rango de tiempo

```
graph1, mapa1 = dateGraph(catalog, startdatelist)
```