Integrantes:

Esteban Leiva, 202021368, e.leivam@uniandes.edu.co

Michelle Vargas, 201914771, bm.vargas@uniandes.edu.co

Análisis de complejidad, tiempo y memoria:

REQ 1:

```
def req1(analyzer,lpId_1,lpId_2):
    try:
        estructura_kosaraju = scc.KosarajuSCC(analyzer["connections_distance"]) #1 O(N)
        num_clusteres = scc.connectedComponents(estructura_kosaraju) #2 O(1)

        cable_1 = lt.getElement(mp.keySet(mp.get(analyzer["cables_dado_lpid"], lpId_1)["value"
]),1) #3 O(1)
        cable_2 = lt.getElement(mp.keySet(mp.get(analyzer["cables_dado_lpid"], lpId_2)["value"
]),1) #4 O(1)

        valor = scc.stronglyConnected(estructura_kosaraju,(lpId_1,cable_1),(lpId_2,cable_2))
#5 O(1)

        return num_clusteres,valor

    except:
        estructura_kosaraju = scc.KosarajuSCC(analyzer["connections_distance"])
        num_clusteres = scc.connectedComponents(estructura_kosaraju)

        return num_clusteres,True
```

Conclusión:

Teniendo en cuenta lo anterior, la complejidad del Requerimiento 1 es O(N), pues el algoritmo de kosaraju tiene una complejidad lineal.

Tiempo y consumo de memoria:

Tiempo (ms)	Memoria (kB)
993.940	136.305

REQ 2:

```
def req2(analyzer):
    lista_vertices = gr.vertices(analyzer["connections_distance"]) #1 O(N)
    final = lt.newList(datastructure="ARRAY_LIST")
    i = 1
    while i <=lt.size(lista_vertices): #2 O(N)
        elem = lt.getElement(lista_vertices, i)
        if gr.degree(analyzer["connections_distance"], elem)>1 and elem[1] != 0:
            lp_id = elem[0]
            lp_name = mp.get(analyzer["name_dado_id"], lp_id)["value"] #3 O(1)
            lista_adyacentes = gr.adjacents(analyzer["connections_distance"], elem) #4 O(n)

            ii = 1
            e = 0
            while ii <= lt.size(lista_adyacentes): #5 O(n)
                adyacente = lt.getElement(lista_adyacentes, ii)
                if adyacente[0] == lp_id:
                    e += 1
                ii+=1
            if e >= 1:
                lt.addLast(final, (lp_name,e+1))
        i+=1
    return final
```

Conclusión:

Teniendo en cuenta lo anterior, la complejidad del Requerimiento 2 es O(N), pues en el #4,5 n es pequeño en comparación a N.

Tiempo y consumo de memoria:

Tiempo (ms)	Memoria (kB)
1071.359	106.980

REQ 3:

```
def req3(analyzer,pais1,pais2):

    delta_time = -1.0
    delta_memory = -1.0

    tracemalloc.start()
    start_time = getTime()
    start_memory = getMemory()

    capital1=model.capital(analyzer,pais1.lower())
    capital2=model.capital(analyzer,pais2.lower())

    if capital1==None or capital2==None:
        print('No hay información para los países dados.')
    else:
        distpath=model.distpath(analyzer,capital1,capital2) #1 O(NlgN)
        distancia=distpath[0]
        path = distpath[1]

        print('La distancia total de la ruta es de: '+str(distancia)+' km.')
        print("")
        print('\nLa ruta está dada por: ')

        i = 1
        inicial = st.size(path)
        while i<=inicial: #2 O(n)
            sub = st.pop(path) #3 O(1)
            if i == 1:
                vertexA = sub["vertexA"][0] + str(" " + str(pais1))
                cableA = "CAPITAL"
            else:
                place = mp.get(analyzer["name_dado_id"], sub["vertexA"][0]) #4 O(1)
                if place != None:
                    vertexA = place["value"]
                    cableA = str(sub["vertexA"][1])
                else:
                    vertexA = sub["vertexA"][0]
                    cableA = "CAPITAL"

            place = mp.get(analyzer["name_dado_id"], sub["vertexB"][0]) #5 O(1)
            if place != None:
                vertexB = place["value"]
                cableB = str(sub["vertexB"][1])
```

```
            else:
                vertexB = sub["vertexB"][0].upper()
                cableB = "CAPITAL"

            print("---------------------")
            print(str(i) + ") ACTUAL: " + str(vertexA) +" |CABLE: "+ cableA  + " -
> SIGUIENTE: " + str(vertexB) +" |CABLE: "+ cableB + str(" | DISTANCIA (KM): ") +str(sub["weig
ht"]))
            i += 1
        stop_memory = getMemory()
        stop_time = getTime()
        tracemalloc.stop()

        delta_time = stop_time - start_time
        delta_memory = deltaMemory(start_memory, stop_memory)

        return None
```

Conclusión:

Teniendo en cuenta lo anterior, la complejidad del Requerimiento 3 es O(NlgN), pues la implementación de Djikstra usada (minPq) tiene complejidad de O (V + ElgV).

Tiempo y consumo de memoria:

Tiempo (ms)	Memoria (kB)
1076.812	49.108

REQ 4:

```
def req4(analyzer):
    estructura = pr.PrimMST(analyzer["connections_distance"]) #1 O(NlgN)
    costo_total = pr.weightMST(analyzer["connections_distance"], estructura) #2 O(N)

    grafo_mst = gr.newGraph(datastructure='ADJ_LIST',
                                            directed=True, size=5000,
                                             comparefunction=None)
    i = 1
    while i<= lt.size(estructura["mst"]): #3 O(N)
        arista = lt.getElement(estructura["mst"], i)
        verticeA = arista["vertexA"]
        verticeB = arista["vertexB"]
        weight = arista["weight"]
```

```
        if not gr.containsVertex(grafo_mst, verticeA):
            gr.insertVertex(grafo_mst, verticeA)

        if not gr.containsVertex(grafo_mst, verticeB):
            gr.insertVertex(grafo_mst, verticeB)

        gr.addEdge(grafo_mst, verticeA, verticeB, weight)
        gr.addEdge(grafo_mst, verticeB, verticeA, weight)

        i += 1

    num_vertices = gr.numVertices(grafo_mst) #4 O(1)

    vertices_mst = gr.vertices(grafo_mst) #5 O(N)

    inicio = lt.firstElement(estructura["mst"])["vertexA"]

    final = lt.lastElement(estructura["mst"])["vertexB"]

    estructura_dfs = dfs.DepthFirstSearch(grafo_mst, inicio) #6 O(N)
    caminoo = dfs.pathTo(estructura_dfs, final)


    return num_vertices, costo_total,caminoo
```

Conclusión:

Teniendo en cuenta lo anterior, la complejidad del Requerimiento 4 es O(NlgN), pues Prim usada tiene complejidad de O(ElogV) y DFS tiene complejidad de O(V)

Tiempo y consumo de memoria:

Tiempo (ms)	Memoria (kB)
2338.788	117.512

REQ 5:

```
def req5(analyzer,id):
    lista_vertices = gr.vertices(analyzer["connections_distance"]) #1 O(N)
    lista_vertices_lp = lt.newList(datastructure="ARRAY_LIST")
    i = 1
    while i<= lt.size(lista_vertices): #2 O(N)
        vertice = lt.getElement(lista_vertices, i)
        if vertice[0] == id:
            lt.addLast(lista_vertices_lp, vertice)
        i += 1
    paises_afectados = mp.newMap()
    ii = 1
    while ii <= lt.size(lista_vertices_lp): #3 O(N)
        vertice = lt.getElement(lista_vertices_lp, ii)
        adyacentes = gr.adjacents(analyzer["connections_distance"], vertice) #4 O(1)
        iii = 1
        while iii <= lt.size(adyacentes): #5 O(n)
            adyacente = lt.getElement(adyacentes, iii)
            if adyacente[1] == 0:
                pais_afectado = mp.get(analyzer["country_dado_city"], adyacente[0])["value"]
                distancia = gr.getEdge(analyzer["connections_distance"], vertice, adyacente)["
weight"]

            else:
                id_adyacente = adyacente[0]
                pais_afectado = mp.get(analyzer["landing_points_country"], id_adyacente)["valu
e"]
                distancia = gr.getEdge(analyzer["connections_distance"], vertice, adyacente)["
weight"]

            if mp.contains(paises_afectados,pais_afectado):
                valor_ant = mp.get(paises_afectados, pais_afectado)["value"]
                if distancia < valor_ant:
                    mp.put(paises_afectados, pais_afectado, distancia)
            else:
                mp.put(paises_afectados, pais_afectado,distancia)

            iii += 1

        ii += 1

    return paises_afectados
```

Conclusión:

Teniendo en cuenta lo anterior, la complejidad del Requerimiento 5 es O(N), pues en el #5 n es pequeño en comparación a N.

Tiempo y consumo de memoria:

Tiempo (ms)	Memoria (kB)
765.907	36.422

REQ 6:

```
def req6(analyzer,pais,cable):

    vertices = gr.vertices(analyzer["connections_capacity"]) #1 O(N)
    ciudad_cap = mp.get(analyzer["countries"], pais)["value"]["CapitalName"].lower()
    vertice_ciudad = (ciudad_cap,0)
    adyacentes_ciudad_cable = adyacentes_ciudad_cabl(analyzer,vertice_ciudad,cable) #2 O(N)

    mapa_adyacentestotales = mp.newMap()
    i = 1
    while i <= lt.size(vertices): #3 O(N)
        vertice = lt.getElement(vertices, i)
        if vertice[1] == cable:

            pais_adyacente2 = mp.get(analyzer["landing_points_country"],vertice[0])["value"]

            poblacion = float(mp.get(analyzer["countries"], pais_adyacente2)["value"]["Populat
ion"].replace(".",""))

            ii = 1
            while ii <= lt.size(gr.adjacents(analyzer["connections_capacity"],vertice)): #4
O(n)
                inicio2 = lt.getElement(gr.adjacents(analyzer["connections_capacity"],vertice)
,ii)
                if inicio2[1] == cable:
                    inicio = inicio2
                    break
                ii +=1

            capacity_mbps = float(gr.getEdge(analyzer["connections_capacity"], inicio, vertice
)["weight"])*1000000
            valor = capacity_mbps/poblacion

            if pais_adyacente2 != pais:
                if mp.contains(mapa_adyacentestotales, pais_adyacente2):
```

```
                valor_ant = mp.get(mapa_adyacentestotales, pais_adyacente2)["value"]
                if valor_ant < valor:
                    mp.put(mapa_adyacentestotales, pais_adyacente2, valor)
            else:
                mp.put(mapa_adyacentestotales, pais_adyacente2, valor)
    i+=1


return mapa_adyacentestotales
```

Conclusión:

Teniendo en cuenta lo anterior, la complejidad del Requerimiento 6 es O(N), pues en el #4 n es pequeño en comparación a N.

Tiempo y consumo de memoria:

Tiempo (ms)	Memoria (kB)
836.456	28.312