Documento de análisis Reto 4

- **Nombres:**

Estudiante 1: Samuel Josué Freire Tarazona, 202111460, s.freire@uniandes.edu.co ---------->

Estudiante 2: José David Martínez Oliveros, 202116677, jd.martinezo1@uniandes.edu.co---------->

V = Vértices

E = Arcos

- **Análisis de complejidad:**
  - **Requerimiento 1:**
    - Primera parte:

```
vertices_total = gr.vertices(analyzer['rutas'])
lista_final = lt.newList('ARRAY_LIST')
lista_canitdad_digrafo = lt.newList('ARRAY_LIST')
```

      - O(V)
    - Segunda parte:

```
for vertice in lt.iterator(vertices_total):
    entran = gr.indegree(analyzer['rutas'],vertice)
    salen = gr.outdegree(analyzer['rutas'],vertice)
    total_rutas = entran + salen
    cantidad = newcantidad(vertice,total_rutas,entran,salen)
    lt.addLast(lista_canitdad_digrafo,cantidad)
    lt.addLast(lista_final,cantidad)
```

      - O(V)
    - Tercera parte:

```
orden = sortcomparecanitdades(lista_canitdad_digrafo)
```

      - O(V$\log V$)
    - Cuarta parte:

```
primeros5 = lt.subList(orden,1,5)
conectados_numero_interno = lt.newList('ARRAY_LIST')
```

      - O(V)

- Quinta parte:

```
for c in lt.iterator(orden):
    if c['cantidadtotal'] != 0:
        lt.addLast(conectados_numero_interno,c)
```

  - O(V)

- parte:

```
vertices_total_no = gr.vertices(analyzer['rutas_idayretorno'])
lista_canitdad_grafo = lt.newList('ARRAY_LIST')
```

  - O(V)

- Sexta parte:

```
for vertice_no in lt.iterator(vertices_total_no):
    entran = gr.degree(analyzer['rutas_idayretorno'],vertice_no)
    salen = gr.degree(analyzer['rutas_idayretorno'],vertice_no)
    total_rutas = entran + salen
    cantidad = newcantidad(vertice_no,total_rutas,entran,salen)
    lt.addLast(lista_canitdad_grafo,cantidad)
    lt.addLast(lista_final,cantidad)
```

  - O(V)

- Septima parte:

```
orden_no = sortcomparecanitdades(lista_canitdad_grafo)
```

  - O(V$\log V$)

- Octava parte:

```
primeros5_no = lt.subList(orden_no,1,5)
conectados_numero_interno_no = lt.newList('ARRAY_LIST')
```

  - O(V)

- Novena parte:

```
for c in lt.iterator(orden_no):
    if c['cantidadtotal'] != 0:
        lt.addLast(conectados_numero_interno_no,c)
```

  - O(V)

- Décima parte:

```
orden_final = sortcomparecanitdades(lista_final)
```

  - O(V$\log V$)

- Undécima parte:

```
primeros5_final = lt.subList(orden_final,1,5)
conectados_numero_interno_final = lt.newList('ARRAY_LIST')
for c in lt.iterator(orden_final):
```

  - O(V)
- Doceava parte:

```
for c in lt.iterator(orden_final):
    if c['cantidadtotal'] != 0:
        lt.addLast(conectados_numero_interno_final,c)
```

  - O(V)

- **COMPLEJIDAD GENERAL**: O(V log V)
- **JUSTIFICACION**: En este caso la complejidad para este requerimiento, en notación Big O, dio O (V log V). Cabe aclarar que los valores quedan en V, porque se están usando los vértices como el valor de datos a recorrer. Esta complejidad, nace del ordenamiento, basado en mergesort, que toca hacer para organizar cierta cantidad de datos. A su vez se hacen un par de recorridos que no presentan la máxima complejidad. Por lo que la complejidad principal resulta del ordenamiento tipo merge que se hace. También, cabe aclarar que se están ordenado el mismo tamaño de vértices que existen en el grafo, por esta razón la complejidad no resulta en N, sino que, se están tomando específicamente que aumentan de acuerdo al número de vértices del grafo. Por lo tanto, la complejidad de este requerimiento resulta ser O (V log V), porque se hacen unos ordenamientos de tipo merge en el grafo.

- **Requerimiento 2:**
  - Primera parte:

```
analyzer['componentes_grafo_dirigdo'] = scc.KosarajuSCC(analyzer['rutas'])
```

    - O(3(V+E))
  - Segunda parte:

```
numero_componentes = scc.connectedComponents(analyzer['componentes_grafo_dirigdo'])
```

    - O(k)
  - Tercera parte:

```
conectados = scc.stronglyConnected(analyzer['componentes_grafo_dirigdo'],codigo1,codigo2)
```

- O(k)

- **COMPLEJIDAD GENERAL**: O((V+E))
- **JUSTIFIACION**: En este caso la complejidad para este requerimiento, en notación Big O, da O ((V+E)).Cabe aclarar, que se hace esto, puesto que los valores con los que aumenta son V (vértices) y E (arcos). Esta complejidad resulta, del algoritmo que se usa en el requerimiento, llamado: "KosarajuSCC". Este algoritmo, busca encontrar componentes de los grafos. Esto lo logra, usando ciertos recorridos, e invirtiendo los grafos para poder hacer esta búsqueda. Nuevamente, su crecimiento se da por los diversos pasos internos que tiene este algoritmo. El algoritmo aumenta de acuerdo a como aumenta la cantidad de vértices y la cantidad de arcos. Por lo tanto, la complejidad de este requerimiento es de O ((V+E)).Puesto que, se usa el algoritmo de Kosaraju, que trae ciertos pasos internos como la inversión de grafos, y los recorridos que terminan haciendo que esta complejidad sea aumentada por el tamaño de vértice y arcos. .

- **Requerimiento 3:**
  - Primera parte:

```
caminos = djk.Dijkstra(analyzer['rutas_idayretorno'], codigo1)
```

    - O(Elog$V$)
  - Segunda parte:

```
lista = djk.pathTo(caminos,maximo)
```

    - O(V-1)
  - Tercera parte:

```
costo_total = 0
for arcos in lt.iterator(lista):
    costo_total += arcos['weight']
```

    - O(V)
  - **COMPLEJIDAD GENERAL**: O (Elog$V$)
  - **JUSTIFIACION**: En este caso, la complejidad de este requerimiento, en notación Big O resulta en O (Elog$V$). Cabe aclarar que el v representa los vértices del grafo. Esta complejidad resulta de usar ciertos recorridos e internamente hacer una operación. Específicamente, resulta de hacer un recorrido sobre los vértices buscados, e internamente

sacar el camino entre el vértice final y el vértice inicial. Por lo tanto, la complejidad dada por esta parte del requerimiento se evalúa con O (Elog$V$) por ser concreta solo en la distancia.

- **Requerimiento 4:**
  - Primera parte:

```
kilometros = float(millas) * 1.60
```

    - O(k)
  - Segunda parte:

```
search = pr.PrimMST(analyzer['rutas_idayretorno'])
```

    - O(Elog$V$)
  - Tercera parte:

```
edge = pr.edgesMST(analyzer['rutas_idayretorno'], search)
nodos_red_expansion = lt.size(edge['mst'])
```

    - O(V)
  - Cuarta parte:

```
busqeuda = df.DepthFirstSearch(analyzer['rutas_idayretorno'],codigo1)
```

    - O(V+E)
  - Quinta parte:

```
for j in lt.iterator(gr.vertices(analyzer['rutas_idayretorno'])):
    camino_espe = df.pathTo(busqeuda,j)
    existe_camino = df.hasPathTo(busqeuda,j)
    if existe_camino is True:
        if lt.size(camino_espe) > mayor:
            mayor = lt.size(camino_espe)
            maximo = j
```

    - O(V(V-1)) = O(V^2)
  - Sexta parte:

```
suma = 0
for c in lt.iterator(mp.valueSet(edge['distTo'])):
    suma += c
```

- O(E)

- Séptima parte:

```
caminos = djk.Dijkstra(analyzer['rutas_idayretorno'], codigo1)
```

- O(E log V)

- Octava parte:

```
lista = djk.pathTo(caminos,maximo)
```

- O(V-1)

- Novena parte:

```
costo_total = 0
for arcos in lt.iterator(lista):
    costo_total += arcos['weight']
```

- O(V)

- Décima parte:

```
if kilometros > float(costo_total):
    resta = round((kilometros - float(costo_total))/1.6,2)
    respuesta = 'sobran ' + str(resta) + ' '

if float(costo_total) > kilometros:
    resta = round((float(costo_total) - kilometros)/1.6,2)
    respuesta = 'faltan ' + str(resta) + ' '
```

- O(K)

- **COMPLEJIDAD GENERAL**: O(V^2)
- **JUSTIFIACION**: En este caso, la complejidad de este requerimiento, en notación Big O resulta en O (V^2). Cabe aclarar que el v representa los vértices del grafo. Esta complejidad resulta de usar ciertos recorridos e internamente hacer una operación. Específicamente, resulta de hacer un recorrido sobre todos los vértices del grafo, e internamente sacar el camino entre esos vértices y el vértice inicial. Esta verificación del camino entre todos los vértices es la que nos genera la complejidad cuadrática, porque saca una lista con el camino entre los vértices y el vértice inicial. Por lo tanto, la complejidad O (V^2).que sale en este requerimiento, resulta de hacer un recorrido sobre una función que tiene una complejidad interna igual.

- **Requerimiento 5:**
  - Primera parte:

```
lista_total = lt.newList('ARRAY_LIST')
fo no dirigido

numero_afectados_nodiri = lt.size(gr.adjacents(analyzer['rutas_idayretorno'],codigo))
afectados_nodiri = gr.adjacents(analyzer['rutas_idayretorno'],codigo)
```

    - O(V)
  - Segunda parte:

```
for c in lt.iterator(afectados_nodiri):
    lt.addLast(lista_total,c)
```

    - O(V)

  - Tercera parte:

```
entran = gr.indegree(analyzer['rutas'],codigo)
salen = gr.outdegree(analyzer['rutas'],codigo)
total_afectados_diri = entran + salen
afectados_diri = lt.newList('ARRAY_LIST')
```

    - O(K)
  - Cuarta parte:

```
arcos_total = gr.edges(analyzer['rutas'])
for c in lt.iterator(arcos_total):
```

- O(E)

- Quinta parte:

```
for c in lt.iterator(arcos_total):
    if c['vertexA'] == codigo:
        existe = lt.isPresent(afectados_diri,c['vertexB'])
        if existe == 0:
            lt.addLast(afectados_diri,c['vertexB'])
            lt.addLast(lista_total,c['vertexB'])
    if c['vertexB'] == codigo:
        existe = lt.isPresent(afectados_diri,c['vertexA'])
        if existe == 0:
            lt.addLast(afectados_diri,c['vertexA'])
            lt.addLast(lista_total,c['vertexA'])
numero_afectados_diri = lt.size(afectados_diri)
```

- O(E)

- Sexta parte:

```
restantes_digrafo = gr.numEdges(analyzer['rutas'])-total_afectados_diri
restantes_grafo = gr.numEdges(analyzer['rutas_idayretorno'])-numero_afectados_nodiri
```

- O(K)

- Séptima parte:

```
if lt.size(afectados_nodiri) >= 6:
    primeros_nodiri = lt.subList(afectados_nodiri,1,3)
    ultimos_nodiri = lt.subList(afectados_nodiri,lt.size(afectados_nodiri)-2,3)
else:
    primeros_nodiri = afectados_nodiri
    ultimos_nodiri = afectados_nodiri
```

- O(E)

- Octava parte:

```
if lt.size(afectados_diri) >= 6:
    primeros_diri = lt.subList(afectados_diri,1,3)
    ultimos_diri = lt.subList(afectados_diri,lt.size(afectados_diri)-2,3)
else:
    primeros_diri = afectados_diri
    ultimos_diri = afectados_diri
```

- O(E)

- Novena parte:

```
numero_afectados_toal = numero_afectados_diri + numero_afectados_nodiri

presion total
if lt.size(lista_total) >= 6:
    primeros_total = lt.subList(lista_total,1,3)
    ultimos_total = lt.subList(lista_total,lt.size(lista_total)-2,3)
else:
    primeros_total = lista_total
    ultimos_total = lista_total
```

- O(E+E) = O(E)

- **COMPLEJIDAD GENERAL**: O(V+E)
- **JUSTIFIACION**: En este caso, la complejidad de este requerimiento, en notación Big O resulta en O (V+E). Cabe aclarar que la E representa los arcos y la V representa los vértices. Esta complejidad resulta de extraer todos los arcos y vértices que están en ambos grafos. Esta extracción tiene internamente que tener un recorrido sobre estos parámetros. Para el caso de los arcos tiene que recorrer toda la lista de arcos para poder tener los resultados. Caso similar sucede con los vértices, que tienen que recorrer todos los vértices para tener todos los vértices. Por lo tanto, la complejidad O (V+E). que sale en este requerimiento, resulta de la extracción de los vértices y arcos del grafo en cuestión. .

- **Pruebas de tiempo**:

Pruebas de tiempo				
Req 1				Promedio
	1546,88	1546,88	1546,88	1546,88
Req 2	LED-RTP	LIS-DXB	ATL-KUF	Promedio
	4250	4283	4031,25	4188,08333
Req 3	20:00-22:00	22:00-23:59	14:00--19:00	Promedio
	2843,75	1593,75	921,87	1786,45667
Req 4	Lisbon	Barcelona	Milan	Promedio
	14906,25	14533,25	13750	14396,5
Req 5	DBX	LIS	LED	Promedio
	578,13	546,88	328,13	484,38