Documento de análisis | Reto 3

Integrantes:

Moisés Agudelo, 202113485, m.agudeloo@uniandes.edu.co
Sergio Franco, 202116614, s.francop@uniandes.edu.co

Requerimiento 1

Análisis de complejidad

```
for i in lt.iterator(cases):
    if om.contains(me.getValue(mp.get(analyzer, "casesByCity")), i["city"]) == False:
        lt.addLast(cityKeys, i["city"])
        om.put(me.getValue(mp.get(analyzer, "casesByCity")), i["city"], lt.newList("ARRAY_LIST"))
```

O(n)

```
for i in lt.iterator(cases):
    lt.addLast(me.getValue(om.get(me.getValue(mp.get(analyzer, "casesByCity")), i["city"])), i)
```

O(n)

```
for i in lt.iterator(cityKeys):
    city = {"city": None,
            "nCases": None
            }

    city["city"] = i
    city["nCases"] = lt.size(me.getValue(om.get(me.getValue(mp.get(outMap, "casesByCity")), i)))
    lt.addLast(citiesAndNCases, city)
```

O(n)

```
sortByNCases(citiesAndNCases)
```

O(nlogn)

Complejidad temporal: O(nlogn)

Al analizar los fragmentos de código que no son de tiempo constante tenemos una complejidad en total de O(n) + O(n) + O(n) + O(nlogn), la cual se puede reescribir como O(n + n +n +nlogn), operándose, O(3n + nlogn), las constantes se omiten, O(n+nlogn) y en la suma de complejidades se deja el termino mayor, concluyendo que la complejidad es O(nlogn), debida a lo ordenamientos de listas de diccionarios.

Requerimiento 2 | Sergio Franco

Análisis de complejidad

```
for i in lt.iterator(cases):
    if om.contains(me.getValue(mp.get(analyzer, "casesBySeconds")), i["duration (seconds)"]) == False:
        lt.addLast(secondsKeys, i["duration (seconds)"])
        om.put(me.getValue(mp.get(analyzer, "casesBySeconds")), i["duration (seconds)"], lt.newList("ARRAY_LIST"))
```

O(n)

```
for i in lt.iterator(secondsKeys):
    if float(i) >= float(beginSeconds) and float(i) <= float(endSeconds):
        lt.addLast(secondsKeysInRange, i)
```

O(n)

```
for i in lt.iterator(cases):
    lt.addLast(me.getValue(om.get(me.getValue(mp.get(analyzer, "casesBySeconds")), i["duration (seconds)"])), i)
```

O(n)

```
for i in lt.iterator(secondsKeysInRange):
    lt.addLast(casesInRange, om.get(me.getValue(mp.get(analyzer, "casesBySeconds")), i))
```

O(n)

```
for i in lt.iterator(secondsKeysInRange):
    nCases += lt.size(me.getValue(om.get(me.getValue(mp.get(analyzer, "casesBySeconds")), i)))
    for j in lt.iterator(me.getValue(om.get(me.getValue(mp.get(analyzer, "casesBySeconds")), i))):
        lt.addLast(onlyCasesInRange, j)
```

O(n^2)

```
onlyCasesInRange = sortDates(onlyCasesInRange)
```

O(nlogn)

Al analizar los fragmentos de código que no son de tiempo constante tenemos una complejidad en total de O(n) + O(n) + O(n) + O(n) + O(n^2) + O(nlogn), la cual se puede reescribir como O(n + n + n + n +nlogn), operándose, O(4n + n^2 + nlogn), las constantes se omiten, O(n+ n^2 + nlogn) y en la suma de complejidades se deja el termino mayor, concluyendo que la complejidad es O(n^2), debida a un ciclo O(n) dentro de otro ciclo O(n).

Requerimiento 3 | Moisés Agudelo

Análisis de complejidad

```
for i in lt.iterator(cases):
    if om.contains(me.getValue(mp.get(analyzer, "casesByHour")), str(dateToHour(i["datetime"]))) == False:
        om.put(me.getValue(mp.get(analyzer, "casesByHour")), str(dateToHour(i["datetime"])), lt.newList("ARRAY_LIST"))
        if lt.isPresent(hoursKeys, str(dateToHour(i["datetime"]))) == 0:
            lt.addLast(hoursKeys, str(dateToHour(i["datetime"])))
```

O(n)

```
for i in lt.iterator(cases):
    lt.addLast(me.getValue(om.get(me.getValue(mp.get(analyzer, "casesByHour")), str(dateToHour(i["datetime"])))), i)
```

O(n)

```
for i in lt.iterator(hoursKeys):
    if toHour(i) >= toHour(beginHour) and toHour(i) <= toHour(endHour):
        lt.addLast(hourKeysInRange, i)
```

O(n)

```
for i in lt.iterator(hourKeysInRange):
    nCases += lt.size(me.getValue(om.get(me.getValue(mp.get(analyzer, "casesByHour")), i)))
    for j in lt.iterator(me.getValue(om.get(me.getValue(mp.get(analyzer, "casesByHour")), i))):
        lt.addLast(onlyCasesInRange, j)
```

O(n^2)

```
onlyCasesInRange = sortDates(onlyCasesInRange)
```

O(nlogn)

Al analizar los fragmentos de código que no son de tiempo constante tenemos una complejidad en total de O(n) + O(n) + O(n) + O(n^2) + O(nlogn), la cual se puede reescribir como O(n + n + n +nlogn), operándose, O(3n + n^2 + nlogn), las constantes se omiten, O(n+ n^2 + nlogn) y en la suma de complejidades se deja el termino mayor, concluyendo que la complejidad es O(n^2), debida a un ciclo O(n) dentro de otro ciclo O(n).

Requerimiento 4

Análisis de complejidad

```
for i in lt.iterator(cases):
    if toDate(i["datetime"]) >= date(PrimeraFecha) and toDate(i["datetime"]) <= date(SegundaFecha):
        lt.addLast(
                    PrimerosTresUltimosTres,
                    i
                )
```

O(n)

Al analizar los fragmentos de código con solo un for que itera la lista, concluyendo así la complejidad O(n).

Requerimiento 5

Análisis de complejidad

```
                    )
for i in lt.iterator(cases):
    if float(i["latitude"]) >= float(LatitudMin) and float(i["latitude"]) <= float(LatidudMax):
        if float(i["longitude"]) >= float(LongitudMin) and float(i["longitude"]) <= float(longitudMax):
            lt.addLast(
                        PrimerosTresUltimosTres,
                        i
                    )
```

O(n)

Al analizar los fragmentos de código con solo un for que en el mismo se hace uso de if que iteran la lista, se puede concluir una complejidad de O(n).