ANÁLISIS DE RESULTADOS RETO 3

GRUPO #03

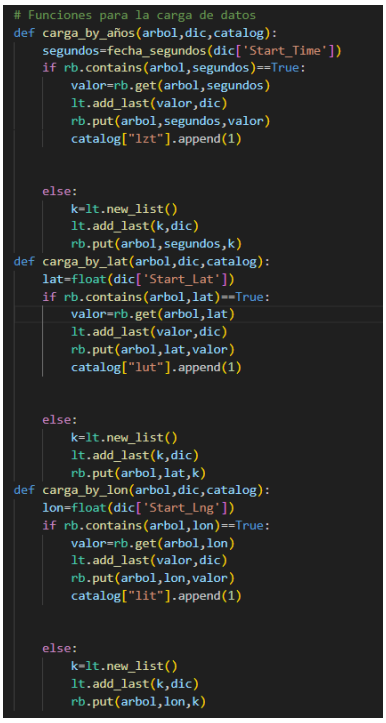
TABLA DE CONTENIDOS:

INTEGRANTES DEL GRUPO

1. Santiago Beltran Melo, s.beltranm1@uniandes.edu.co, 201911611.
2. Samuel Esteban Pardo Forero, s.pardof@uniandes.edu.co, 202410675.
3. Gerónimo Rojas, g.rojasr@uniandes.edu.co, 202215835.

ANÁLISIS DE COMPLEJIDAD DE LOS REQUERIMIENTOS

1. **Load data:**



```python
# Funciones para la carga de datos
def carga_by_años(arbol,dic,catalog):
    segundos=fecha_segundos(dic['Start_Time'])
    if rb.contains(arbol,segundos)==True:
        valor=rb.get(arbol,segundos)
        lt.add_last(valor,dic)
        rb.put(arbol,segundos,valor)
        catalog["lzt"].append(1)


    else:
        k=lt.new_list()
        lt.add_last(k,dic)
        rb.put(arbol,segundos,k)
def carga_by_lat(arbol,dic,catalog):
    lat=float(dic['Start_Lat'])
    if rb.contains(arbol,lat)==True:
        valor=rb.get(arbol,lat)
        lt.add_last(valor,dic)
        rb.put(arbol,lat,valor)
        catalog["lut"].append(1)


    else:
        k=lt.new_list()
        lt.add_last(k,dic)
        rb.put(arbol,lat,k)
def carga_by_lon(arbol,dic,catalog):
    lon=float(dic['Start_Lng'])
    if rb.contains(arbol,lon)==True:
        valor=rb.get(arbol,lon)
        lt.add_last(valor,dic)
        rb.put(arbol,lon,valor)
        catalog["lit"].append(1)


    else:
        k=lt.new_list()
        lt.add_last(k,dic)
        rb.put(arbol,lon,k)
```

```
def load_data(catalog, filename):

    """
    Carga los datos del reto
    """
    accidents = csv.DictReader(open(".\\Data\\Challenge-3\\"+filename, encoding='utf-8'))

    for elemento in accidents:

        rta = {}
        rta['ID'] = elemento['ID']
        rta['Source'] =  elemento['Source']
        rta['Severity'] = elemento['Severity']
        rta['Start_Time'] = elemento['Start_Time']
        rta['End_Time'] = elemento['End_Time']
        rta['Start_Lat'] = elemento['Start_Lat']
        rta['Start_Lng'] = elemento['Start_Lng']
        rta['End_Lat'] = elemento['End_Lat']
        rta['End_Lng'] = elemento['End_Lng']
        rta['Distance(mi)'] = elemento['Distance(mi)']
        rta['Description'] = elemento['Description']
        rta['Street'] = elemento['Street']
        rta['City'] = elemento['City']
        rta['County'] = elemento['County']
        rta['State'] = elemento['State']
        rta['Temperature(F)'] = elemento['Temperature(F)']
        rta['Wind_Chill(F)'] = elemento['Wind_Chill(F)']
        rta['Humidity(%)'] = elemento['Humidity(%)']
        rta['Pressure(in)'] = elemento['Pressure(in)']
        rta['Visibility(mi)'] = elemento['Visibility(mi)']
        rta['Wind_Direction'] = elemento['Wind_Direction']
        rta['Wind_Speed(mph)'] = elemento['Wind_Speed(mph)']
        rta['Precipitation(in)'] = elemento['Precipitation(in)']
        rta['Weather_Condition'] = elemento['Weather_Condition']
        carga_by_años(catalog["fecha"],rta,catalog)
        carga_by_lat(catalog["lat"],rta,catalog)
        carga_by_lon(catalog["lon"],rta,catalog)
        lt.add_last(catalog["accidents"],rta)

    return catalog
```

En el caso de la función de carga de datos, inicialmente creamos un array list haciendo uso de un ciclo For. En principio, esto cuenta con complejidad O(n).

Adicionalmente usamos funciones adicionales para realizar una carga y organización extra de los datos y así modelarlos como árboles rojo-negro usando como llaves la fecha de inicio, la latitud y la longitud. Estas funciones hacen uso de las funciones put y contains de rbt, las cuales tiene complejidad O (Log N).

En conclusión, la complejidad de nuestra carga de datos es O (N Log N).

2. **Get Data:**

```
def get_data(catalog, id):
    """
    Retorna un dato por su ID.
    """
    info = None
    for accidente in catalog['accidents']:
        if accidente['ID'] == id:
            info = accidente
    return info
```

La función Get data cuenta con un único ciclo For, y dentro de este tenemos comparaciones y asignaciones, por ende, su complejidad es O(N).

3. **Requerimiento 1:**

```python
def req_1(catalog,fecha_inicia,fecha_final):
    """
    Retorna el resultado del requerimiento 1
    """
    inicial_segs =  fecha_segundos(fecha_inicia)
    final_segs = fecha_segundos(fecha_final)
    inicio = rb.ceiling(catalog['fecha'],inicial_segs)
    fin = rb.floor(catalog['fecha'],final_segs)

    if inicio is None or fin is None:
        print("No se encontraron accidentes en el rango de fechas especificado.")
        return None

    accidentes_en_rango = rb.values(catalog['fecha'],inicio,fin)

    lista_filtrada = lt.new_list()

    for lista_accidentes in accidentes_en_rango['elements']:
        for accidente in lista_accidentes['elements']:

            id_accidente = accidente['ID']
            fecha_hora_inicio = accidente['Start_Time']
            ciudad = accidente['City']
            estado = accidente['State']
            descripcion = accidente['Description'][:40]

            inicio = datetime.strptime(accidente['Start_Time'], "%Y-%m-%d %H:%M:%S")
            final = datetime.strptime(accidente['End_Time'], "%Y-%m-%d %H:%M:%S")
            duracion_horas = (final - inicio).total_seconds() / 3600


            lt.add_last(lista_filtrada,{
                "ID": id_accidente,
                "Fecha y Hora de Inicio": fecha_hora_inicio,
                "Ciudad": ciudad,
                "Estado": estado,
                "Descripción": descripcion,
                "Duración en horas": duracion_horas
            })

    lista_ordenada = lt.merge_sort(lista_filtrada,sort_por_fecha_y_severidad)

    return lista_ordenada
```

El requerimiento 2, inicialmente usa la función de conversión de fecha a segundos, cuya complejidad es O (1).
Las funciones ceiling y floor de rbt nos devuelven las llaves para obtener la lista de los accidentes entre las dos fechas de rango y son complejidad O (Log N), y la función values, también de rbt, es O(n).
La iteración sobre la lista de accidentes es complejidad O(n). Aunque es un ciclo anidado y podría ser pensado como complejidad O(N^2), este no es el caso ya que solo se están recorriendo las listas, no se están haciendo comparaciones u operaciones entre ellas.
Finalmente, el algoritmo merge_sort es complejidad O (N Log N).
En conclusión, la complejidad de este requerimiento es O (N Log N).

4. **Requerimiento 2:**

```python
def req_2(catalog, visibility_range, state_list):
    """
    Retorna el resultado del requerimiento 2
    """
    resultados_estados = {}
    total_accidentes = 0

    for i in range(lt.size(catalog["accidents"])):
        accident = lt.get_element(catalog["accidents"], i)
        if 'Visibility(mi)' in accident:
            if 'Severity' in accident:
                if 'State' in accident:
                    if 'Distance(mi)' in accident:
                        visibility_str = accident['Visibility(mi)']
                        severity_str = accident['Severity']
                        state = accident['State']
                        distance_str = accident['Distance(mi)']
                        if visibility_str and severity_str and state and distance_str:
                            visibility = float(visibility_str)
                            severity = int(severity_str)
                            distance = float(distance_str)
                            if severity == 4:
                                if visibility_range[0] <= visibility <= visibility_range[1]:
                                    if state in state_list:
                                        total_accidentes += 1
                                        if state not in resultados_estados:
                                            resultados_estados[state] = {
                                                'count': 0,
                                                'total_visibility': 0,
                                                'total_distance': 0,
                                                'max_distance': None
                                            }
                                        state_data = resultados_estados[state]
                                        state_data['count'] += 1
                                        state_data['total_visibility'] += visibility
                                        state_data['total_distance'] += distance
                                        if (state_data['max_distance'] is None or
                                            distance > float(state_data['max_distance']['Distance(mi)'])):
                                            state_data['max_distance'] = accident
    for state, data in resultados_estados.items():
        data['average_visibility'] = data['total_visibility'] / data['count']
        data['average_distance'] = data['total_distance'] / data['count']
        print(f"Estado: {state}, Promedio de visibilidad: {data['average_visibility']}, Promedio de distancia: {data['average_distance']}")
    sorted_states = sorted(resultados_estados.items(),key=lambda x: (-x[1]['count'], x[1]['average_visibility']))

    resultado = {
        'total_accidentes': total_accidentes,
        'state_analysis': sorted_states
    }

    return resultado
```

En este algoritmo podemos observar que iniciamos con un ciclo For, cuya complejidad es O(N) siendo N el número de accidentes. Dentro de este ciclo verificamos si state se encuentra en la lista state_list, y este proceso también es complejidad O(N).

Siguiendo adelante podemos observas varias comparaciones y asignación de complejidad O (1), y un ciclo For de complejidad O(N).

Finalmente, la función sorted usada para organizar la lista tiene una complejidad O (N Log N) y es el termino más significativo de ese algoritmo.

En conclusión, la complejidad global del requerimiento 2 es O (N Log N).

5. **Requerimiento 3:**

```python
def req_3(catalog, n):
    """
    Retorna el resultado del requerimiento 3
    """
    n = int(n)
    accidentes_recientes = []

    for i in range(lt.size(catalog["accidents"])):
        accidente = lt.get_element(catalog["accidents"], i)
        visibility_str = accidente.get('Visibility(mi)', '')
        if visibility_str and accidente.get('Weather_Condition') and accidente['Severity'] in {'3', '4'}:
            if any(cond in accidente['Weather_Condition'].lower() for cond in ["rain", "snow"]):
                visibility = float(visibility_str) if visibility_str else float('inf')
                if visibility < 2:
                    accidentes_recientes.append(accidente)
    accidentes_recientes.sort(
        key=lambda x: (datetime.strptime(x['Start_Time'], "%Y-%m-%d %H:%M:%S"),-int(x['Severity'])), reverse=True
    )

    accidentes_recientes = accidentes_recientes[:n]
    visibilidad_total = sum(float(a['Visibility(mi)']) for a in accidentes_recientes if a.get('Visibility(mi)'))
    visibilidad_promedio = visibilidad_total / len(accidentes_recientes) if accidentes_recientes else 0

    resultado = {
        "average_visibility": visibilidad_promedio,
        "accidents": accidentes_recientes
    }

    return resultado
```

En este algoritmo iniciamos con una iteración sobre la lista de accidentes, cuya complejidad es O(N).

Dentro de este ciclo contamos con varias asignaciones y comparaciones lo cual cuenta con una complejidad O (1), y aunque la línea “ any(cond in accidente['Weather_Condition'].lower() for cond in ["rain", "snow"])” puede tener una complejidad O(N), esta se mantiene constante con respecto al tamaño del catálogo, lo que mantiene su complejidad aproximadamente en O (1).

La función sort emplead en el ordenamiento de la lista cuenta con complejidad O (N Log N)

El slicing usado en accidentes recientes y el cálculo de visibilidad promedio cuentan con complejidad O(N) ya que aplican funciones de recorrido sobre la lista.

Finalmente, podemos ver que la complejidad del algoritmo es O (N Log N).

6. **Requerimiento 4:**

```python
def req_4(catalog,fecha_i,fecha_f):
    arbol=catalog["fecha"]
    inic=fecha_segundos(fecha_i)
    finic=fecha_segundos(fecha_f)
    ini=rb.ceiling(arbol,inic)
    fini=rb.floor(arbol,finic)
    lista=rb.values(arbol,ini,fini)
    dic={}
    for i in range(0,lt.size(lista)):
        for j in range(0,lt.size(lista["elements"][i])):
            if lista["elements"][i]["elements"][j]["Visibility(mi)"]!='':
             if int(lista["elements"][i]["elements"][j]["Severity"])>=3 and float(lista["elements"][i]["elements"][j]["Visibility(mi)"])<1 and fecha_segundos(lista["elements"][i]["e
                k=lista["elements"][i]["elements"][j]["Street"]
                if k not in dic:
                    if int(lista["elements"][i]["elements"][j]["Severity"])==3:
                        dic[k]={"city":lista["elements"][i]["elements"][j]["City"],"county":lista["elements"][i]["elements"][j]["County"],"state":lista["elements"][i]["elements"][j
                    else:
                        dic[k]={"city":lista["elements"][i]["elements"][j]["City"],"county":lista["elements"][i]["elements"][j]["County"],"state":lista["elements"][i]["elements"][j
                elif k in dic and dic[k]["city"]==lista["elements"][i]["elements"][j]["City"] and dic[k]["county"]==lista["elements"][i]["elements"][j]["County"]and dic[k]["state"]
                    dic[k]["visi"]+=float(lista["elements"][i]["elements"][j]["Visibility(mi)"])
                    if int(lista["elements"][i]["elements"][j]["Severity"])==3:
                        dic[k]["pelig"]+=3
                        dic[k]["total3"]+=1

                    else:
                        dic[k]["pelig"]+=4
                        dic[k]["total4"]+=1



    for i in dic:
        dic[i]["pelig"]=dic[i]["pelig"]/(dic[i]["total3"]+dic[i]["total4"])
        dic[i]["visi"]=dic[i]["visi"]/(dic[i]["total3"]+dic[i]["total4"])
    ordenado = dict(sorted(dic.items(), key=lambda item: (item[1]['state'], item[1]['county'], item[1]['city'], item[0])))
    return ordenado
```

Cómo fue mencionado anteriormente, el uso de ceiling y floor es complejidad O (Log N), y la función values cuenta con complejidad O(N)

Siguiendo adelante, encontramos dos ciclos For anidados y, dependiente de la cantidad de elementos, podríamos decir que la complejidad es O(M*N).

Dentro de dicho ciclo tenemos varias asignaciones y comparaciones, lo cual es complejidad O (1).

Más adelante nos encontramos con otro ciclo For de complejidad O(N), y finalmente tenemos la función de ordenamiento del diccionario, lo cual es complejidad O (N Log N).

En conclusión, se podría generalizar la complejidad del algoritmo a una aproximación de O(N Log N) en el peor caso.

7. Requerimiento 5:

```python
def req_5(catalog,fecha_inicio,fecha_fin,condiciones_climaticas):
    """
    Retorna el resultado del requerimiento 5
    """

    inicio_seg = fecha_segundos(fecha_inicio + " 00:00:00")
    fin_seg = fecha_segundos(fecha_fin + " 23:59:59")

    inicio = rb.ceiling(catalog['fecha'], inicio_seg)
    fin = rb.floor(catalog['fecha'], fin_seg)
    if inicio is None or fin is None:
        print("No se encontraron accidentes en el rango de fechas especificado.")
        return None

    accidentes_en_rango = rb.values(catalog["fecha"], inicio, fin)

    franjas = {
        "Mañana": {"total": 0, "severidad_suma": 0, "condiciones": {}},
        "Tarde": {"total": 0, "severidad_suma": 0, "condiciones": {}},
        "Noche": {"total": 0, "severidad_suma": 0, "condiciones": {}},
        "Madrugada": {"total": 0, "severidad_suma": 0, "condiciones": {}}
    }

    for lista_accidentes in accidentes_en_rango['elements']:
        for accidente in lista_accidentes['elements']:

            if accidente['Severity'] == '3' or accidente['Severity'] == '4':
                if accidente['Weather_Condition'] in condiciones_climaticas:

                    start_time = datetime.strptime(accidente['Start_Time'], "%Y-%m-%d %H:%M:%S")
                    franja = obtener_franja_horaria(start_time.hour)



                    franjas[franja]["total"] += 1
                    franjas[franja]["severidad_suma"] += int(accidente['Severity'])


                    condicion = accidente['Weather_Condition'].lower()
                    if condicion in franjas[franja]["condiciones"]:
                        franjas[franja]["condiciones"][condicion] += 1
                    else:
                        franjas[franja]["condiciones"][condicion] = 1


    resultados = lt.new_list()
    for franja, datos in franjas.items():
        if datos["total"] > 0:
            promedio_severidad = datos["severidad_suma"] / datos["total"]


            condicion_predominante = max(datos["condiciones"], key=datos["condiciones"].get)

            lt.add_last(resultados, {
                "franja_horaria": franja,
                "total_accidentes": datos["total"],
                "promedio_severidad": promedio_severidad,
                "condicion_predominante": condicion_predominante
            })
    resultados_ordenados = lt.quick_sort(resultados,sort_crit)

    return resultados_ordenados
```

En este algoritmo empezamos con una conversión de fechas a segundos, lo cual es complejidad O (1).

El uso de ceiling y floor tiene complejidad O (Log N) y values de O(N).

El uso del doble ciclo For anidado tiene una complejidad de O(N*M), siendo N y M los elementos de las respectivas listas.

Las operaciones realizadas dentro de estos ciclos cuentan con complejidad O (1).

Finalmente, el uso de Quick sort cuenta con complejidad O (N Log N).

Para concluir, podemos decir que en el peor de los casos la complejidad del algoritmo es O (N Log N).

8. Requerimiento 6:

```python
def req_6(catalog,fecha_inicio, fecha_fin, humedad_minima, lista_condados):
    """
    Retorna el resultado del requerimiento 6
    """

    inicio_seg = fecha_segundos(fecha_inicio + " 00:00:00")
    fin_seg = fecha_segundos(fecha_fin + " 23:59:59")


    inicio = rb.ceiling(catalog['fecha'], inicio_seg)
    fin = rb.floor(catalog['fecha'], fin_seg)



    if inicio is None or fin is None:
        print("No se encontraron accidentes en el rango de fechas especificado.")
        return None


    estadisticas_por_condado = {}


    accidentes_en_rango = rb.values(catalog["fecha"], inicio, fin)




    for lista_accidentes in accidentes_en_rango['elements']:
        for accidente in lista_accidentes['elements']:

            if accidente['Severity'] == "3" or accidente['Severity'] == "4":
                if accidente['Humidity(%)'] != '' and accidente['Humidity(%)'] is not None:
                    humedad = float(accidente['Humidity(%)'])
                    condado = accidente['County']


                    if humedad >= humedad_minima and condado in lista_condados:


                        if condado not in estadisticas_por_condado:

                                estadisticas_por_condado[condado] = {
                                    "total_accidentes": 0,
                                    "temperatura_suma": 0,
                                    "humedad_suma": 0,
                                    "viento_suma": 0,
                                    "distancia_suma": 0,
                                    "accidente_mas_grave": None,
                                    "max_severidad": -1
                                }
```

```
                estadisticas_por_condado[condado]["total_accidentes"] += 1
                if accidente['Temperature(F)'] != '' and accidente['Temperature(F)'] is not None:
                    estadisticas_por_condado[condado]["temperatura_suma"] += float(accidente['Temperature(F)'])
                if accidente['Humidity(%)'] != '' and accidente['Humidity(%)'] is not None:
                    estadisticas_por_condado[condado]["humedad_suma"] += float(accidente['Humidity(%)'])
                if accidente['Wind_Speed(mph)'] != '' and accidente['Wind_Speed(mph)'] is not None:
                    estadisticas_por_condado[condado]["viento_suma"] += float(accidente['Wind_Speed(mph)'])
                if accidente['Distance(mi)'] != '' and accidente['Distance(mi)'] is not None:
                    estadisticas_por_condado[condado]["distancia_suma"] += float(accidente['Distance(mi)'])


                if int(accidente['Severity']) > estadisticas_por_condado[condado]["max_severidad"]:
                    estadisticas_por_condado[condado]["max_severidad"] = int(accidente['Severity'])
                    estadisticas_por_condado[condado]["accidente_mas_grave"] = {
                        "ID": accidente['ID'],
                        "Fecha": accidente['Start_Time'],
                        "Temperatura": accidente['Temperature(F)'],
                        "Humedad": accidente['Humidity(%)'],
                        "Distancia": accidente['Distance(mi)'],
                        "Descripción": accidente['Description']
                    }

    resultados = lt.new_list()
    for condado, datos in estadisticas_por_condado.items():
        if datos["total_accidentes"] > 0:

            promedio_temperatura = datos["temperatura_suma"] / datos["total_accidentes"]
            promedio_humedad = datos["humedad_suma"] / datos["total_accidentes"]
            promedio_viento = datos["viento_suma"] / datos["total_accidentes"]
            promedio_distancia = datos["distancia_suma"] / datos["total_accidentes"]


            lt.add_last(resultados, {
                "condado": condado,
                "total_accidentes": datos["total_accidentes"],
                "promedio_temperatura": promedio_temperatura,
                "promedio_humedad": promedio_humedad,
                "promedio_viento": promedio_viento,
                "promedio_distancia": promedio_distancia,
                "accidente_mas_grave": datos["accidente_mas_grave"]
            })


    resultados_ordenados = lt.quick_sort(resultados, sort_por_accidentes)
    return resultados_ordenados
```

Una vez más, este algoritmo empieza con la conversión de una fecha a segundo, lo cual es complejidad O (1).

De nuevo el uso de las funciones ceiling y floor son complejidad O (Log n), y la función values O(N).

Los ciclos For anidados tiene una complejidad O(N*M), siendo N y M los elementos de las respectivas listas. Las operaciones realizadas dentro de estos ciclos cuentan con complejidad O (1).

El ciclo For próximo tiene complejidad O(N) y, finalmente, el uso de merge sort es O (N Log N).

En conclusión, la complejidad inicial del algoritmo es O (Log N + N*M + N Log N), lo que se puede aproximar a O (N Log N)

9. Requerimiento 7:

```
def req_7(catalog,lami,lamax,lomi,lomax):
    arbol1=catalog["lat"]
    ini1=rb.ceiling(arbol1,float(lami))
    fini1=bs.max_key(arbol1)
    lista1=rb.values(arbol1,ini1,fini1)
    arbol2=catalog["lon"]
    ini2=rb.ceiling(arbol2,float(lomi))
    fini2=bs.max_key(arbol2)
    lista2=rb.values(arbol2,ini2,fini2)
    lista3=lt.new_list()
    ids_array1 = {element['ID'] for group in lista1['elements'] for element in group['elements']}
    ids_array2 = {element['ID'] for group in lista2['elements'] for element in group['elements']}
    common_ids = list(ids_array1.intersection(ids_array2))
    total=0
    dic={}
    if len(common_ids)>0:
     for i in range(0,lt.size(lista1)):
        for j in range(0,lt.size(lista1["elements"][i])):
         if lista1["elements"][i]["elements"][j]["ID"] in common_ids:
            lt.add_last(lista3,lista1["elements"][i]["elements"][j])
    for i in range(0,lt.size(lista3)):
        if lista3["elements"][i]["End_Lat"]!=''and lista3["elements"][i]["End_Lng"]!='':
            if float(lista3["elements"][i]["End_Lat"])<float(lamax) and float(lista3["elements"][i]["End_Lng"])<float(lomax):
                total +=1
                dic[lista3["elements"][i]["ID"]]=lista3["elements"][i]
    sorted_data = dict(sorted(
    dic.items(),
    key=lambda item: (
        float(item[1]['Start_Lat']),
        float(item[1]['Start_Lng']),
        float(item[1]['End_Lat']),
        float(item[1]['End_Lng']))))
    return dic,total
```

En este algoritmo inicialmente hacemos uso de la función ceiling O (Log N), max_key O (Log N) y values O(N) para el árbol ordenado por latitud, y repetimos el proceso para el árbol ordenado por longitud.

Más adelante, en la creación del array1 y el array2 tenemos una complejidad O(N+M) ya que recorremos los elementos de la lista 1 y 2.

La siguiente función significativa es el doble ciclo For anidado, cuya complejidad es O(N*M). Las comparaciones dentro de esta función son O (1).

Más adelante nos encontramos con otro ciclo For, cuya complejidad es O(N), y las asignaciones y comparaciones dentro de este son O (1).

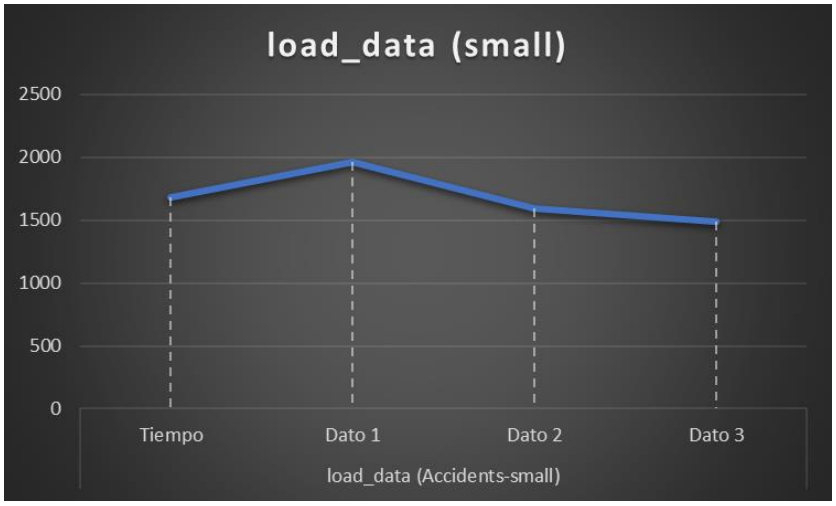
Finalmente, el ordenamiento del diccionario final es O (N Log N).

En conclusión, la complejidad global sería O (N*M + N Log N) y esto lo podemos aproximar a O (N Log N) en el peor de los casos.

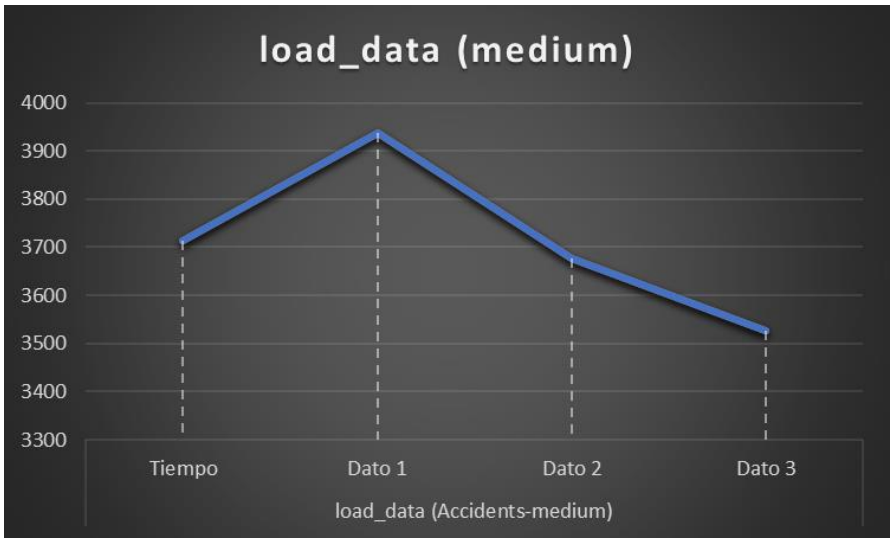
PRUEBAS DE TIEMPO DE EJECUCIÓN

1. Load Data:

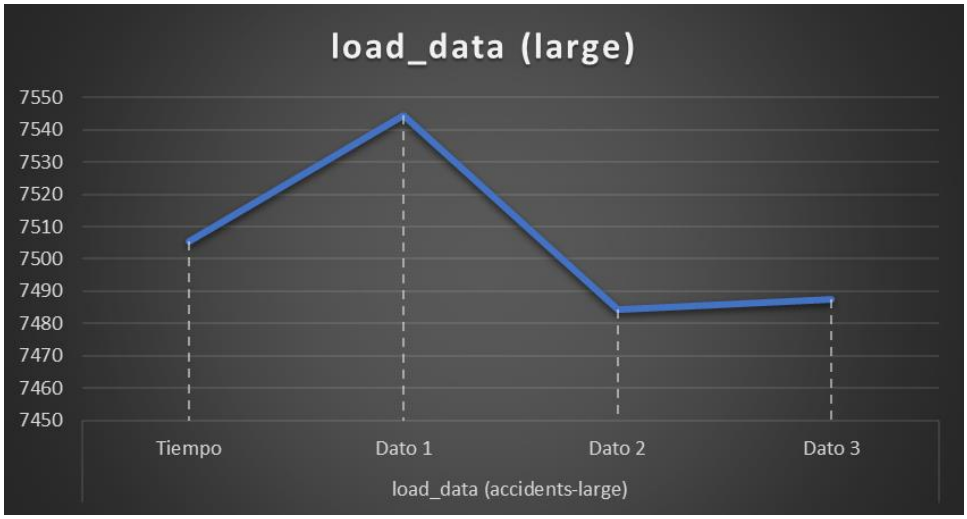
load_data (Accidents-small)			
Tiempo	Dato 1	Dato 2	Dato 3
1683,3467	1963,57	1594,88	1491,59



load_data (Accidents-medium)			
Tiempo	Dato 1	Dato 2	Dato 3
3713,16	3936,35	3677,31	3525,82



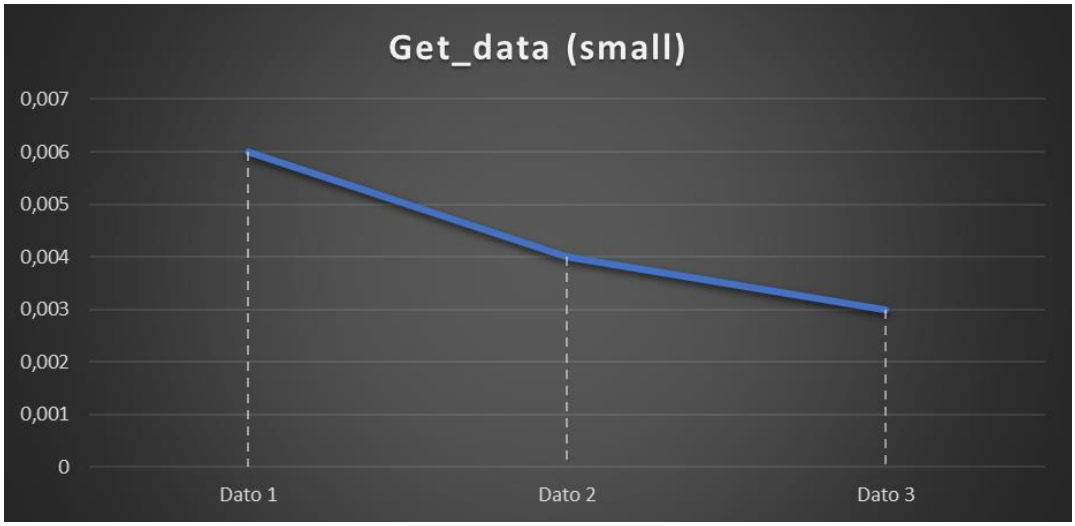
load_data (accidents-large)			
Tiempo	Dato 1	Dato 2	Dato 3
7505,473333	7544,56	7484,27	7487,59



Podemos observar que el margen de crecimiento de la función es O (N log N). Las variaciones en el primer pico se deben a que es la primera carga de datos, por lo cual toma más tiempo. A partir de ese punto empieza a asemejarse potencialmente a una función Logarítmica.

2. **Get Data:**

get_data (Accidents-small)				
id	Tiempo	Dato 1	Dato 2	Dato 3
A-50280	0,0043333	0,006	0,004	0,003



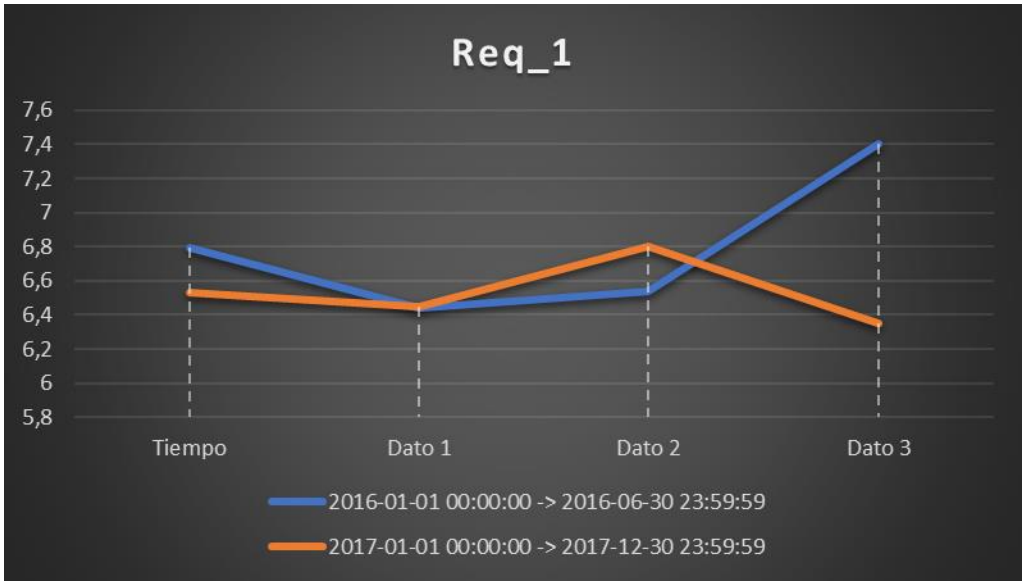
get_data (Accidents-medium)				
id	Tiempo	Dato 1	Dato 2	Dato 3
A-3433120	0,0023333	0,002	0,003	0,002

get_data (Accidents-large)				
id	Tiempo	Dato 1	Dato 2	Dato 3
A-152763	0,002	0,002	0,002	0,002

La función get_data es O(N) pero tiene potencial para modelarse como una función constante ya que crece en correlación a la cantidad de datos de la lista. Siguiendo esta lógica, las gráficas tienen sentido.

3. **Requerimiento 1 (**Santiago Beltran**)**

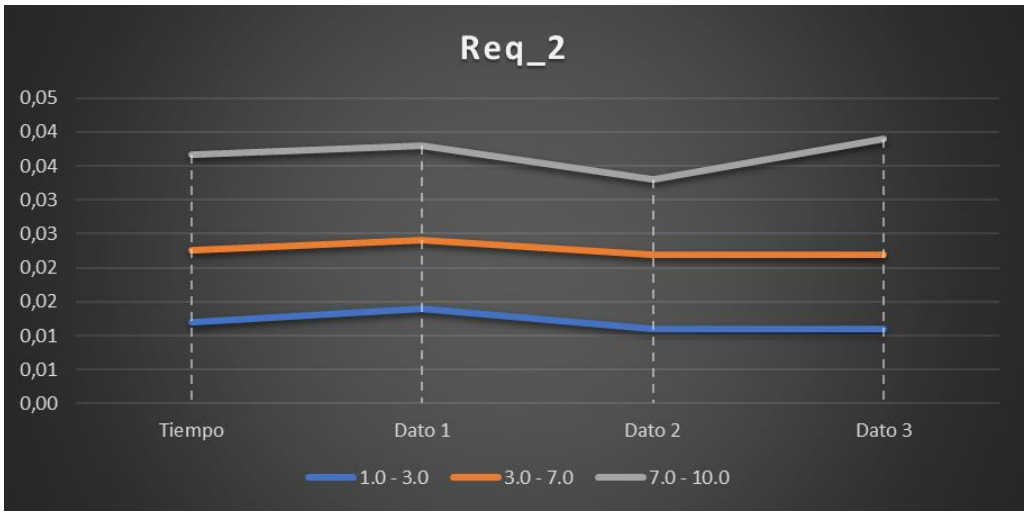
req_1 (Accidents-large)				
Casos	Tiempo	Dato 1	Dato 2	Dato 3
2016-01-01 00:00:00 -> 2016-06-30 23:59:59	6,79333333	6,44	6,54	7,4
2017-01-01 00:00:00 -> 2017-12-30 23:59:59	6,53333333	6,45	6,8	6,35



Las funciones logarítmicas crecen de manera exponencial, más rápido que una función O(N), pero más despacio que una O(N^2), por ende, su gráfica se sitúa justo en medio de estas dos, y esto es lo que podemos observar, principalmente en la línea azul.

4. **Requerimiento 2(**Samuel Pardo**)**

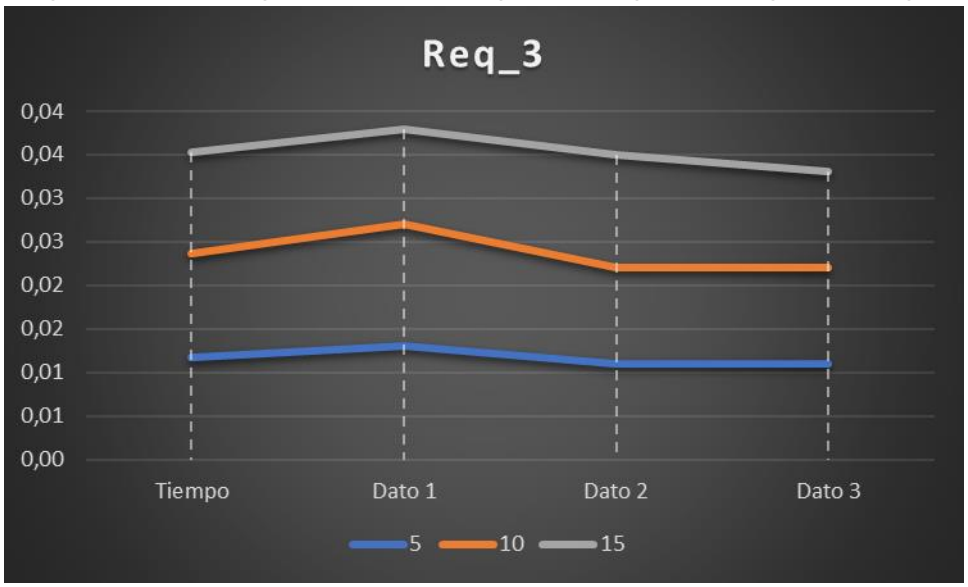
req_2 (Accidents-large) [,CA, NY, TX,]				
Casos	Tiempo	Dato 1	Dato 2	Dato 3
1.0 - 3.0	0,012	0,014	0,011	0,011
3.0 - 7.0	0,010666667	0,01	0,011	0,011
7.0 - 10.0	0,014	0,014	0,011	0,017



De la misma manera podemos observar que el crecimiento gradual de esta gráfica se asemeja a una gráfica de complejidad O (N Log N).

5. **Requerimiento 3(**Samuel Pardo**)**

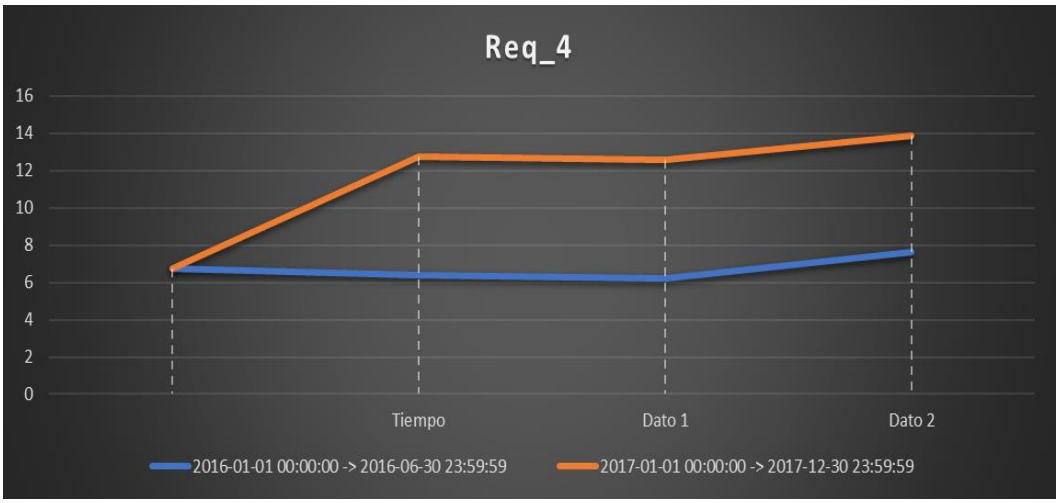
req_3 (Accidents-large)				
Casos	Tiempo	Dato 1	Dato 2	Dato 3
5	0,011666667	0,013	0,011	0,011
10	0,012	0,014	0,011	0,011
15	0,011666667	0,011	0,013	0,011



Esta grafica representa el mejor caso del algoritmo, donde los elementos están casi ordenados, por ende, el uso del sort O (N Log N) no llega a ser un gran problema en la complejidad de este algoritmo, y se modela como O(N)

6. **Requerimiento 4(**Gerónimo Rojas**)**

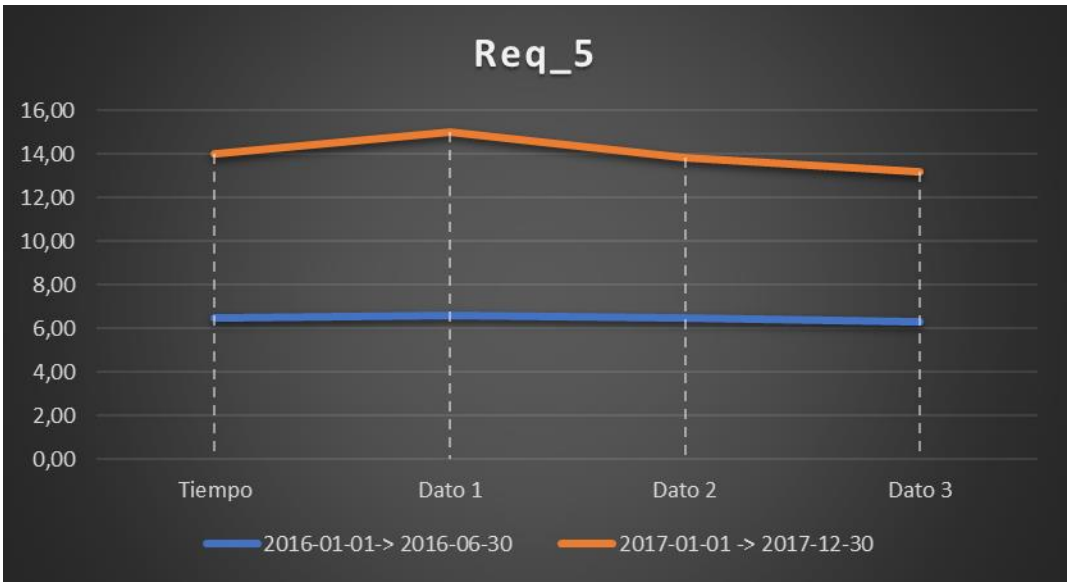
req_4 (Accidents-large)				
Casos	Tiempo	Dato 1	Dato 2	Dato 3
2016-01-01 00:00:00 -> 2016-06-30 23:59:59	6,74	6,39	6,19	7,64
2017-01-01 00:00:00 -> 2017-12-30 23:59:59	6,376666667	6,39	6,25	6,49



Esta gráfica representa de manera casi precisa la de una función O (N Log N), pues su potencial de incremento se ve mayor al de O(N) pero así mismo menor a O(N^2).

7. **Requerimiento 5(**Santiago Beltran**)**

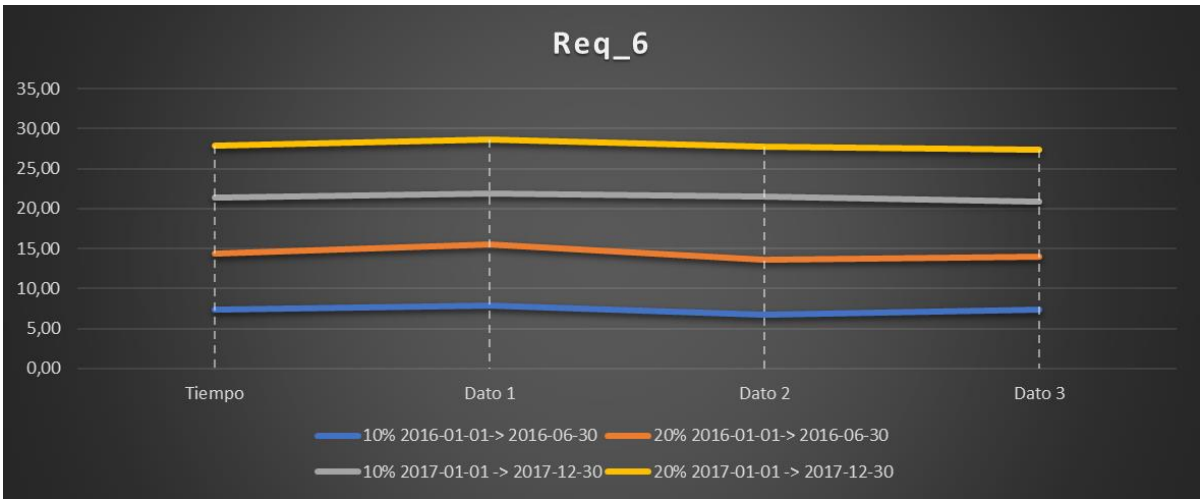
req_5 (Accidents-large) (Clear Rain Snow)				
Casos	Tiempo	Dato 1	Dato 2	Dato 3
2016-01-01-> 2016-06-30	6,477666667	6,613	6,49	6,33
2017-01-01 -> 2017-12-30	7,55	8,41	7,38	6,86



En esta gráfica una vez más tenemos un caso en el cual el sort no llega a ser un gran influyente, pues se puede dar el caso en el cual la lista esta semiordenada y se modela de manera casi constante ya que los datos crecen en correlación a las listas que se recorren.

8. **Requerimiento 6(**Santiago Beltran**)**

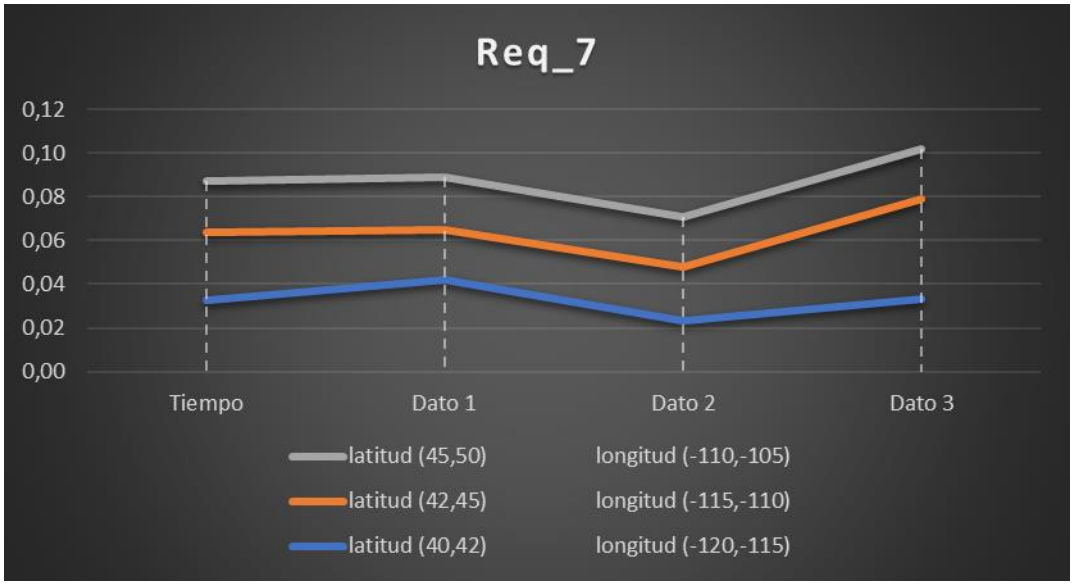
	req_6 (Accidents-large) (Cuyahoga Dallas Los Angeles Kent)				
%	Casos	Tiempo	Dato 1	Dato 2	Dato 3
10%	2016-01-01-> 2016-06-30	7,333333333	7,94	6,73	7,33
20%	2016-01-01-> 2016-06-30	7,07	7,62	6,88	6,71
10%	2017-01-01 -> 2017-12-30	6,996666667	6,35	7,85	6,79
20%	2017-01-01 -> 2017-12-30	6,54	6,78	6,32	6,52



En este algoritmo podemos ver un caso muy similar al anterior, donde con los parámetros dados tenemos el mejor caso de la función.

9. **Requerimiento 7(**Gerónimo Rojas**):**

req_7 (Accidents-large)				
Casos	Tiempo	Dato 1	Dato 2	Dato 3
latitud (40,42) longitud (-120,-115)	0,032666667	0,042	0,023	0,033
latitud (42,45) longitud (-115,-110)	0,031333333	0,023	0,025	0,046
latitud (45,50) longitud (-110,-105)	0,023333333	0,024	0,023	0,023



Una vez más podemos observar una gráfica con potencial crecimiento O (N Log N), lo cual concuerda con el análisis de complejidad.