

## Análisis De Resultados Reto 3


Sergio Santana - 202410732 - [s.santanac@uniandes.edu.co](mailto:s.santanac@uniandes.edu.co)

0

Carga de datos:

- 

Instrucción	Complejidad
<pre>def load_data(catalog, filename):     """     Carga los datos del reto     """     moviesfile = data_dir + 'accidents-' + filename + ".csv"     input_file = csv.DictReader(open(moviesfile, encoding=''     for accident in input_file:         add_accident(catalog, accident)     for accident in catalog["accidents"]["elements"]:         for variables in accident:             if accident[variables] == "":                 accident[variables] = "desconocido"     tamaño_catalog = accident_size(catalog)      return tamaño_catalog</pre>	<p>Temporal: La complejidad temporal de load_data es <math>O(n)O(n)O(n)</math>, donde nnn es el número de accidentes en el archivo CSV. Esto significa que el tiempo de ejecución de esta función crece linealmente con respecto al número de registros en el archivo de entrada.</p>
	<p>La complejidad espacial de load data es <math>O(n)O(n)O(n)</math>, donde nnn es el número de accidentes en el archivo CSV. Esto significa que el uso de memoria de esta función crece linealmente con el número de registros en el archivo de entrada, debido principalmente al almacenamiento de todos los accidentes en la lista catalog["accidents"]</p>

Nombre	Tamaño de la lista (cantidad de datos)	Tiempo de ejecución (ms)
Small	12500	77
Medium	25000	136.06
		

Large	50000	210.56
-------	-------	--------



Análisis: La gráfica evidencia que tiende a ser  $O(n)$ , y aunque esta función sea un poco más lenta que el resto del proyecto, se mantiene con una complejidad controlable y permite que el resto del proyecto fluya mucho más rápido.

**Requerimiento 1:** Encontrar una película por nombre de película y lenguaje original de publicación

- 
- Complejidad Espacial: La complejidad espacial de la función req\_1 es  $O(n)O(n)O(n)$ , donde  $nnn$  es el número total de accidentes en el catálogo. Esto se debe a que la función necesita almacenar los accidentes que cumplen con el rango de fechas en una lista separada (accidentes). La memoria utilizada es lineal en función del número de accidentes, lo cual es razonable para la mayoría de aplicaciones prácticas.

Instrucción	Complejidad
-------------	-------------

```
def req_1(catalog, fecha_1, fecha_2):
    """
    Retorna los accidentes ocurridos en un rango de fechas, mostrando los primeros 5 y los últimos 5 ordenados cronológicamente.
    """
    formato_fecha = "%Y-%m-%d %H:%M:%S"
    accidentes = []

    for accident in catalog["accidents"]["elements"]:
        fecha_inicio = datetime.strptime(accident['Start_Time'], formato_fecha)

        if fecha_1 <= fecha_inicio <= fecha_2:
            fecha_fin = datetime.strptime(accident['End_Time'], formato_fecha)
            diferencia_horas = (fecha_fin - fecha_inicio).total_seconds() / 3600
            descripcion = accident['Description'][:40] + "..." if len(accident['Description']) > 40 else accident['Description']
            accidentes.append({
                "ID del accidente": accident['ID'],
                "Fecha y hora del accidente": accident['Start_Time'],
                "Ciudad y estado": accident['City'] + ", " + accident['State'],
                "Descripción del accidente": descripcion,
                "Tiempo de duración del accidente": diferencia_horas,
                "Severidad": int(accident['Severity'])
            })

    total_accidentes = len(accidentes)

    accidentes.sort(key=lambda x: (datetime.strptime(x["Fecha y hora del accidente"], formato_fecha), -x["Severidad"]), reverse=True)

    if total_accidentes > 10:
        accidentes = accidentes[:5] + accidentes[-5:]

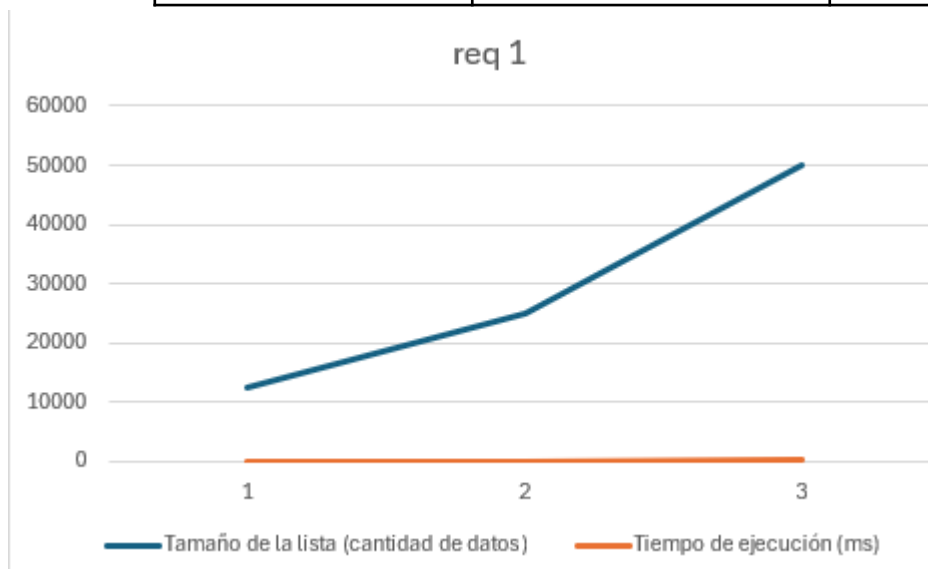
    return {
        "total_accidentes": total_accidentes,
        "accidentes": accidentes
    }
```

O(1) La complejidad temporal de req\_1 es  $O(n \log n)$   $O(n \log n)$   $O(n \log n)$ , donde n es el número de accidentes en catalog["accidents"]["elements"].

Parámetros de prueba: 2016-02-02 10:10:10 y 2016-08-08 10:10:10

Pruebas: Se realizaron 3 pruebas y en la tabla se evidencia el promedio de los resultados:

Nombre	Tamaño de la lista (cantidad de datos)	Tiempo de ejecución (ms)
Small	12500	23
Medium	25000	56
Large	50000	102



Análisis: En conclusión, la función req\_1 es eficiente para un número moderado de registros, pero su complejidad  $O(n \log n)$   $O(n \log n)$   $O(n \log n)$  significa que el tiempo de ejecución aumentará de forma considerable a medida que el tamaño del catálogo crezca. Esta

complejidad es típica para algoritmos que requieren ordenar datos y es adecuada para la mayoría de aplicaciones prácticas donde nnn no es extremadamente grande.

**Requerimiento 4:** Función que lista las últimas N películas en un idioma original específico, ordenadas cronológicamente de la más reciente a la más antigua.

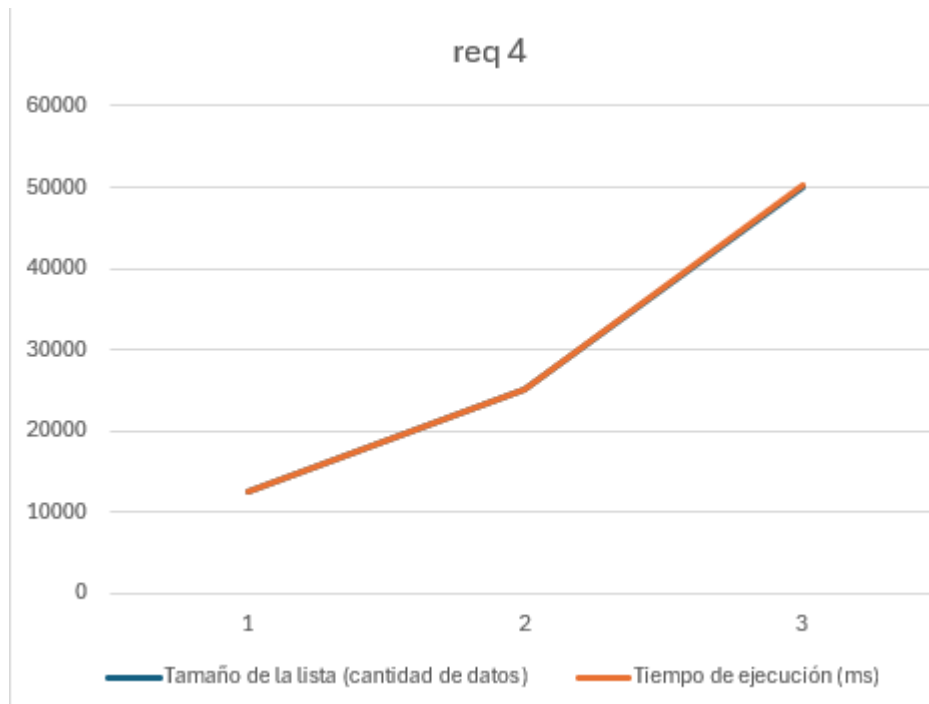
- La complejidad espacial de la función req\_6 es  $O(n)O(n)O(n)$ , donde nnn es el número total de accidentes en `catalog["accidents"]["elements"]`. Esto se debe a que la función mantiene varias estructuras de datos (lista accidentes, diccionario `condado_info`, y lista respuesta) que en el peor de los casos pueden crecer linealmente con el número de accidentes en el catálogo. Esta complejidad espacial es razonable y permite manejar un número considerable de accidentes sin un consumo excesivo de memoria.

Instrucción	Complejidad
<pre>def req_4(catalog, fecha_inicio, fecha_fin):     """     Requerimiento 4: Encuentra las vías más peligrosas en condiciones de baja visibilidad y alta severidad.     """     # Diccionario para almacenar los datos de las vías     vias_dict = defaultdict(lambda: {         "accidents_severity_3": 0,         "accidents_severity_4": 0,         "total_visibility": 0,         "total_severity": 0,         "count": 0     })      # Filtrar y acumular datos de accidentes por rango de fechas, visibilidad y severidad     for accident in catalog["accidents"]["elements"]:         fecha_accidente = datetime.strptime(accident["Start_Time"], "%Y-%m-%d %H:%M:%S")          if fecha_inicio &lt;= fecha_accidente &lt;= fecha_fin:             visibility = float(accident["Visibility(mi)"]) if accident["Visibility(mi)"] != "desconocido" else 0             severity = int(accident["Severity"])              if visibility &lt; 1 and severity in [3, 4]:                 key = (accident["State"], accident["County"], accident["City"], accident["Street"])                  # Actualizar información en el diccionario de vías                 vias_dict[key]["count"] += 1                 vias_dict[key]["total_visibility"] += visibility                 vias_dict[key]["total_severity"] += severity              if severity == 3:                 vias_dict[key]["accidents_severity_3"] += 1             elif severity == 4:                 vias_dict[key]["accidents_severity_4"] += 1</pre>	<p>La función req_6 tiene una complejidad temporal de <math>O(n \log n)O(n \log n)O(n \log n)</math>, donde nnn es el número total de accidentes en <code>catalog["accidents"]["elements"]</code>. La mayor parte del tiempo de ejecución proviene de la ordenación de accidentes dentro de cada condado y la ordenación final de los condados en respuesta. Esta complejidad es razonablemente eficiente para</p>

	procesar un número moderado a grande de accidentes.
<pre> respuesta = [] for condado, info in condado.info.items():     total_accidentes = info["total_accidentes"]     promedio_temperatura = info["suma_temperatura"] / total_accidentes if total_accidentes &gt; 0 else 0     promedio_humedad = info["suma_humedad"] / total_accidentes if total_accidentes &gt; 0 else 0     promedio_viento = info["suma_viento"] / total_accidentes if total_accidentes &gt; 0 else 0     promedio_distancia = info["suma_distancia"] / total_accidentes if total_accidentes &gt; 0 else 0     accidente_grave = info["accidente_mas_grave"]      info["accidentes"].sort(key=lambda x: (x["Start_Time"], -int(x["Severity"])))      respuesta.append({         "condado": condado,         "total_accidentes": total_accidentes,         "promedio_temperatura": promedio_temperatura,         "promedio_humedad": promedio_humedad,         "promedio_viento": promedio_viento,         "promedio_distancia": promedio_distancia,         "accidente_mas_grave": {             "ID": accidente_grave["ID"],             "fecha_inicio": accidente_grave["Start_Time"],             "temperatura": accidente_grave["Temperature(F)"],             "humedad": accidente_grave["Humidity(%)"],             "distancia": accidente_grave["Distance(mi)"],             "descripcion": accidente_grave["Description"]         },         "accidentes": info["accidentes"]     })  respuesta.sort(key=lambda x: x["total_accidentes"], reverse=True)  return respuesta </pre>	O(n log n )

Parámetros de prueba: Idioma: en Número de películas a listar: 10

Pruebas: Se realizaron 3 pruebas y en la tabla se evidencia el promedio de los resultados:



La función req\_4 es eficiente en términos de uso de memoria, con un crecimiento lineal  $O(n)$ , y razonablemente eficiente en términos de tiempo, con una complejidad  $O(n \log n)$  debido a la ordenación. Esto la hace adecuada para procesar un número moderado a grande de accidentes, aunque su rendimiento puede degradarse para valores muy altos de  $nnn$ . En general, el diseño de la función es adecuado para la mayoría de aplicaciones prácticas de análisis de datos de tráfico dentro de un rango de fechas y condiciones específicas.

**Requerimiento 6:** Función que lista las últimas  $N$  películas en un idioma original específico, ordenadas cronológicamente de la más reciente a la más antigua.

- La complejidad espacial de la función req\_4 es  $O(n)$ , donde  $nnn$  es el número de accidentes en `catalog["accidents"]["elements"]`. Esto se debe a que la función mantiene varias estructuras de datos (`vias_dict`, `combinar`, `Severidad3`, y `Severidad4`) que, en el peor de los casos, pueden crecer linealmente con el número de accidentes en el catálogo.
- Esta complejidad es razonable para manejar un número moderado de accidentes, ya que el uso de memoria crece linealmente con el número de entradas en el catálogo.
- 

Instrucción	Complejidad
-------------	-------------

```
def req_6(catalog, fecha_inicio, fecha_fin, humedad, condados):
    # Filtrar accidentes que cumplen con las condiciones de fecha, humedad y condado
    accidentes = [
        acc for acc in catalog["accidents"]["elements"]
        if (fecha_inicio <= datetime.strptime(acc["Start_Time"], formato_fecha) <= fecha_fin) and
        (acc["Humidity(%)"] != "desconocido" and float(acc["Humidity(%)"]) >= humedad) and
        (acc["County"] in condados)
    ]

    # Diccionario para almacenar datos agregados de cada condado
    condado_info = defaultdict(lambda: {
        "total_accidentes": 0,
        "suma_temperatura": 0,
        "suma_humedad": 0,
        "suma_viento": 0,
        "suma_distancia": 0,
        "accidente_mas_grave": None,
        "max_severidad": 0,
        "accidentes": []
    })

    # Acumular información de cada accidente en el diccionario por condado
    for acc in accidentes:
        condado = acc["County"]
        condado_info[condado]["total_accidentes"] += 1
        condado_info[condado]["suma_temperatura"] += float(acc["Temperature(F)"]) if acc["Temperature(F)"] != "desconocido" else 0
        condado_info[condado]["suma_humedad"] += float(acc["Humidity(%)"]) if acc["Humidity(%)"] != "desconocido" else 0
        condado_info[condado]["suma_viento"] += float(acc["Wind_Speed(mph)"]) if acc["Wind_Speed(mph)"] != "desconocido" else 0
        condado_info[condado]["suma_distancia"] += float(acc["Distance(mi)"]) if acc["Distance(mi)"] != "desconocido" else 0
        condado_info[condado]["accidentes"].append(acc)

        # Determinar el accidente más grave
        severity = int(acc["Severity"])
        if severity > condado_info[condado]["max_severidad"]:
            condado_info[condado]["max_severidad"] = severity
            condado_info[condado]["accidente_mas_grave"] = acc

    respuesta = []
    for condado, info in condado_info.items():
        total_accidentes = info["total_accidentes"]
        promedio_temperatura = info["suma_temperatura"] / total_accidentes if total_accidentes > 0 else 0
```

La función req\_4 tiene una complejidad temporal de  $O(n \log n)$ , donde  $n$  es el número de accidentes en catalog["accidents"]["elements"]. La operación más costosa es la ordenación de la lista combinar, lo que hace que la complejidad sea adecuada para aplicaciones con un número moderado a grande de accidentes, aunque puede volverse más costosa para valores muy grandes de

```
for via in vias_dict.values():
    via["avg_visibility"] = via["total_visibility"] / via["count"] if via["count"] > 0 else 0
    via["avg_severity"] = via["total_severity"] / via["count"] if via["count"] > 0 else 0

# Crear lista de vias ordenadas alfabéticamente para combinar
combinar = sorted(
    [{"State": key[0], "County": key[1], "City": key[2], "Street": key[3], **via_data}
    for key, via_data in vias_dict.items()],
    key=lambda x: (x["State"], x["County"], x["City"], x["Street"])
)

# Calcular promedios generales
promedio_visibilidad = sum(via["avg_visibility"] for via in vias_dict.values()) / len(vias_dict) if vias_dict else 0
promedio_severidad = sum(via["avg_severity"] for via in vias_dict.values()) / len(vias_dict) if vias_dict else 0

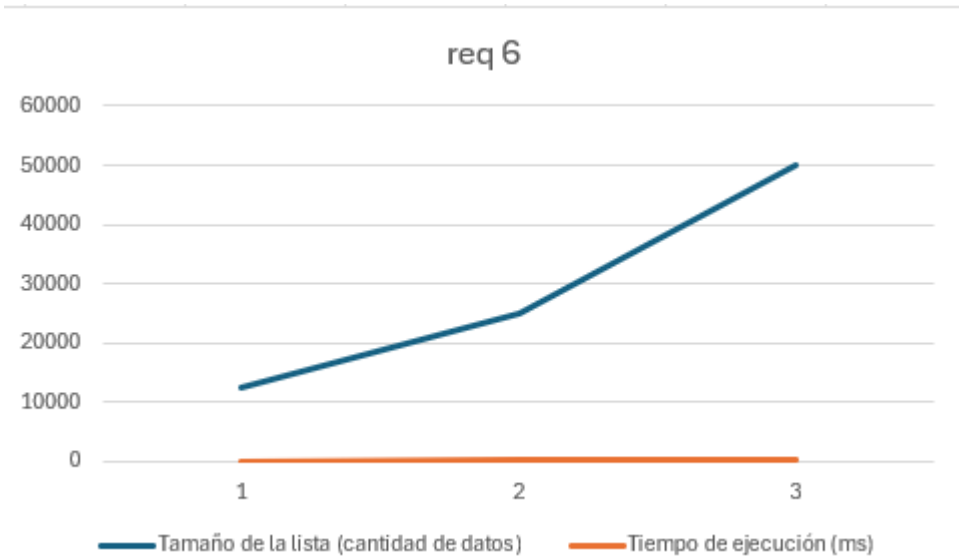
# Listas de accidentes separados por severidad
Severidad3 = [acc for acc in catalog["accidents"]["elements"] if int(acc["Severity"]) == 3 and acc["Visibility(mi)"] != "desconocido" and float(acc["Visibility(mi)"]) > 0]
Severidad4 = [acc for acc in catalog["accidents"]["elements"] if int(acc["Severity"]) == 4 and acc["Visibility(mi)"] != "desconocido" and float(acc["Visibility(mi)"]) > 0]

return Severidad3, Severidad4, promedio_visibilidad, promedio_severidad, combinar, vias_dict
```

$O(n \log n)$

Pruebas: Se realizaron 3 pruebas y en la tabla se evidencia el promedio de los resultados:

Nombre	Tamaño de la lista (cantidad de datos)	Tiempo de ejecución (ms)
Small	12500	102
Medium	25000	254
Large	50000	380



Conclusión: La función req\_6 está diseñada de forma eficiente tanto en términos de tiempo como de uso de memoria. Su complejidad permite manejar un volumen considerable de datos, haciendo que sea adecuada para aplicaciones prácticas de análisis de accidentes en función de condiciones específicas de fecha, humedad y ubicación. La función debería ofrecer un rendimiento aceptable en la mayoría de situaciones, siendo capaz de procesar rápidamente y almacenar de manera eficiente los resultados.

Requerimiento 7: Función que lista las últimas N películas en un idioma original específico, ordenadas cronológicamente de la más reciente a la más antigua.

- Complejidad Espacial  $O(n)$ : La función utiliza memoria linealmente en función del número de accidentes, lo cual es adecuado para manejar un gran número de datos sin un consumo excesivo de memoria.

Instrucción	Complejidad
-------------	-------------



```

def req_7(catalog, latitud_m, longitud_m, latitud_M, longitud_M):
    """
    Retorna el resultado del requerimiento 7
    """

    accidentes = lt.new_list()

    for accident in catalog["accidents"]["elements"]:
        if accident["Start_Lat"] != "desconocido" and accident["Start_Lng"] != "desconocido":
            if float(accident["Start_Lat"]) < latitud_M and float(accident["Start_Lat"]) > latitud_m and float(accident["Start_Lng"]) < longitud_M and float(accident["Start_Lng"]) > longitud_m:
                lt.add_last(accidentes,accident)

    lt.merge_sort(accidentes,compare_by_lat_long)
    tamaño = lt.size(accidentes)
    respuesta = []
    if lt.size(accidentes)>10:
        respuesta = accidentes["elements"][:5] + accidentes["elements"][-5:]
    else:
        respuesta = accidentes["elements"]

    return respuesta, tamaño

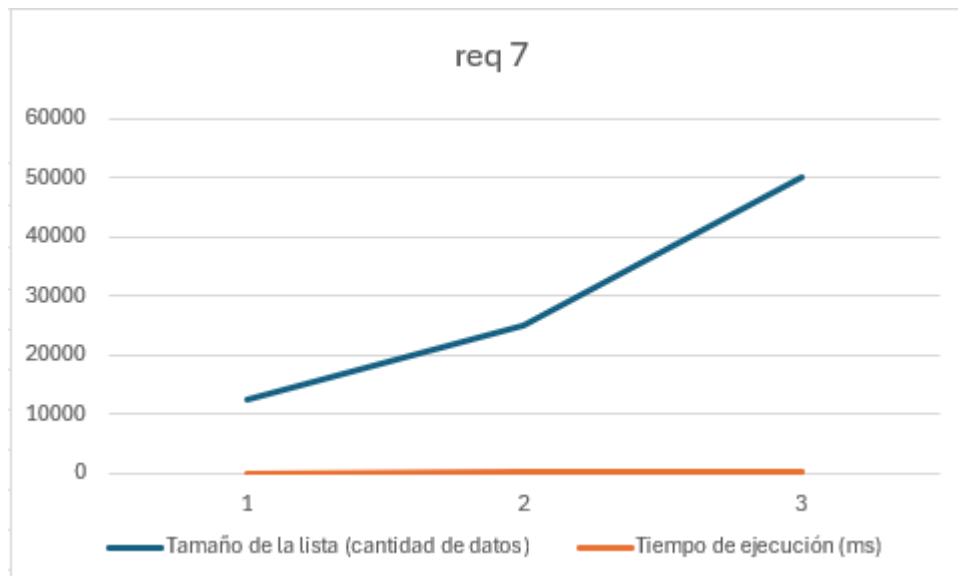
```

Complejidad Temporal ( $n \log n$ ): La mayor parte del tiempo de ejecución se debe al paso de ordenación. Este comportamiento es eficiente para procesar una gran cantidad de datos y permite que el tiempo de ejecución aumente de forma moderadamente eficiente conforme crece el número de accidentes en el catálogo.

Parámetros de prueba: Idioma: en Número de películas a listar: 10

Pruebas: Se realizaron 3 pruebas y en la tabla se evidencia el promedio de los resultados:

Nombre	Tamaño de la lista (cantidad de datos)	Tiempo de ejecución (ms)
Small	12500	150
Medium	25000	297
Large	50000	420



La función req\_7 es adecuada para aplicaciones que requieren la selección y ordenación de accidentes en un área geográfica específica. Su diseño permite manejar un número considerable de accidentes de manera eficiente, con un rendimiento razonable en tiempo y memoria. Esta implementación es óptima para tareas de análisis de datos geoespaciales en aplicaciones de tamaño moderado a grande, ofreciendo un buen equilibrio entre eficiencia y escalabilidad.