Departamento de Ingeniería de Sistemas y Computación
Estructuras de Datos y Algoritmos
ISIS-1225

ANÁLISIS DEL RETO

REQ 3 Luis Restrepo Escudero, 202221383, l.restrepoe@uniandes.edu.co

REQ 5 Santiago Macías Cuel, 202221028, s.maciasc@uniandes.edu.co

REQ 4 César Augusto Pérez Carvajal, 202223716, ca.perezc1@uniandes.edu.co.

Requerimiento <<1>>

Descripción

Función req_1(n, n_equipo, condicion, data_structs, valor_ord)

Esta función se encarga de resolver el Requerimiento 1, que implica filtrar y ordenar partidos según un equipo y una condición dada.

n: Es el número de partidos que se deben mostrar en los resultados.

n_equipo: Es el nombre del equipo que se utilizará para filtrar los partidos.

condicion: Es una cadena que puede tener los valores "local", "visitante" o "todos", que indica cómo se deben filtrar los partidos.

data_structs: Es un conjunto de estructuras de datos que contiene información sobre los partidos.

valor_ord: Es un criterio de ordenamiento que determina cómo se deben ordenar los partidos en los resultados.

El código realiza las siguientes operaciones:

Filtra los partidos según la condición especificada (local, visitante o todos) y el equipo proporcionado. Esto implica iterar a través de los partidos en data_structs y seleccionar aquellos que cumplan con los criterios.

Ordena los partidos filtrados según el criterio de ordenamiento especificado. Esto implica aplicar una función de ordenamiento a la lista de partidos, que puede ser ascendente o descendente según el criterio.

Limita la cantidad de partidos a mostrar a n si hay más de n partidos disponibles. Esto se hace seleccionando los primeros n elementos de la lista ordenada.

Devuelve la lista de partidos ordenada y su tamaño después de aplicar todos los filtros y limitaciones.

Entrada	
Salidas	Respuesta esperada del algoritmo.
Implementado (Sí/No)	Si se implementó y quien lo hizo.

Análisis de complejidad

La operación más costosa en tiempo resulta ser la organización de los datos por fecha al final del algoritmo de merge sort. Este tiene una complejidad de O(Nlog(N)). También se puede notar que se itera N veces el arreglo conteniendo la información de los partidos; complejidad de O(N). Se toma en cuenta el factor más influyente, siendo la complejidad temporal de O(Nlog(N)) al final.

```
lista = lt.iterator(data_structs["results"])

partidos_filtrados = lt.newList(datastructure='ARRAY_LIST')
if condicion == "local":
    for partidos in lista:
        if partidos["home_team"] == n_equipo:
            lt.addLast(partidos_filtrados,partidos)

if condicion == "visitante":
    for partidos in lista:
        if partidos["away_team"] == n_equipo:
            lt.addLast(partidos_filtrados,partidos)
else:
    for partidos in lista:
        if partidos["away_team"] == n_equipo or partidos["home_team"]  == n_equipo:
            lt.addLast(partidos_filtrados,partidos)

partidos_ordenados = sort(partidos_filtrados, valor_ord, criterio_date_min)

resultado = lt.newList(datastructure='ARRAY_LIST')
```

Comparándolo con nuestra experimentación es posible discernir que

Pruebas Realizadas

Descripción de las pruebas de tiempos de ejecución y memoria utilizada. Incluir descripción del procedimiento, las condiciones, las herramientas y recursos utilizados (librerías, computadores donde se ejecutan las pruebas, entre otros).

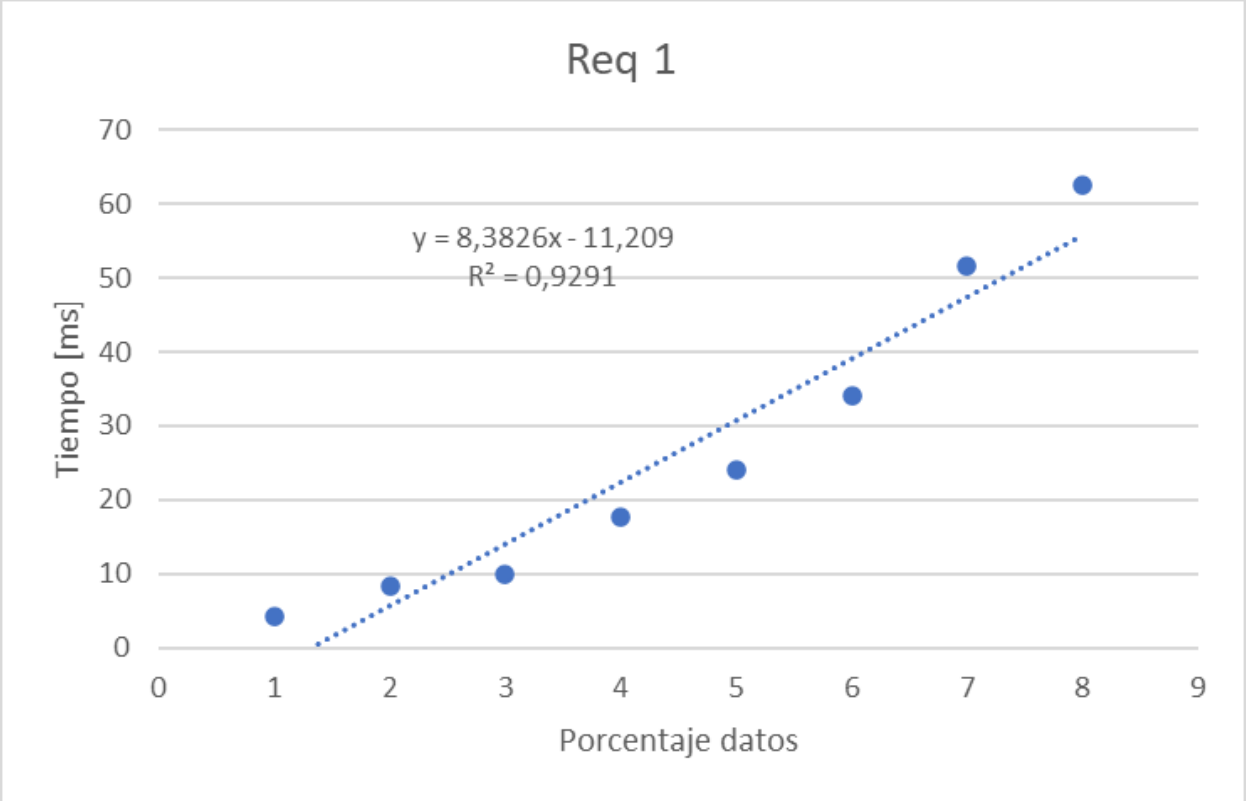
Procesadores	**13th Gen Intel(R) Core(TM) i7-13650HX 2.60 GHz**
Memoria RAM	32,0 GB
Sistema Operativo	Windows 11 Home

Tablas de datos

ARRAY_LIST

Porcentaje de la muestra [pct]	Merge [ms]
small	4.15
5%	8.23
10%	9.88
20%	17.65
30%	24.05
50%	34.02
80%	51.66
large	62.46

Gráficas



Análisis

Luego de realizar la gráfica de tiempos del requerimiento 1 los resultados que se obtienen es que que la operación más costosa del algoritmo en términos de tiempo es la organización de datos por fecha, que en uno de los criterios que se piden, y esta operación es la que más influye en la complejidad temporal del algoritmo, que se establece en O(N log(N), sin embargo en la gráfica se adapta a un algoritmo lineal.

Requerimiento <<2>>

Descripción

Función req_2(data_structs, jugador, num_gol, valor_ord)

Esta función resuelve el Requerimiento 2, que consiste en filtrar y ordenar las anotaciones de un jugador específico según ciertos criterios.

data_structs: Es un conjunto de estructuras de datos que contiene información sobre las anotaciones.

jugador: Es el nombre del jugador cuyas anotaciones se deben analizar.

num_gol: Es el número mínimo de goles que debe tener el jugador para ser incluido en los resultados.

valor_ord: Es un criterio de ordenamiento que determina cómo se deben ordenar las anotaciones en los resultados.

El código realiza las siguientes operaciones:

Filtra las anotaciones del jugador especificado, excluyendo autogoles y asegurándose de que el jugador tenga al menos num_gol goles.

Ordena las anotaciones filtradas según el criterio de ordenamiento especificado. Esto implica aplicar una función de ordenamiento a la lista de anotaciones, que puede ser ascendente o descendente según el criterio.

Devuelve la lista de anotaciones ordenada y su tamaño después de aplicar los filtros y la ordenación.

Entrada	Parámetros necesarios para resolver el requerimiento.
Salidas	Respuesta esperada del algoritmo.
Implementado (Sí/No)	Si se implementó y quien lo hizo.

Análisis de complejidad

La operación más costosa en tiempo resulta ser la organización de los datos por fecha al final del algoritmo de merge sort. Este tiene una complejidad de O(Nlog(N)). También se puede notar que se itera N veces el arreglo conteniendo la información de las anotaciones; complejidad de O(N). Se toma en cuenta el factor más influyente, siendo la complejidad temporal de O(Nlog(N)).

```
lista_anotaciones = lt.iterator(data_structs["goalscorers"])

goles_filtrados = lt.newList(datastructure='ARRAY_LIST')
for anotacion in lista_anotaciones:
    if anotacion["scorer"].lower() == jugador and anotacion["own_goal"]=="False":
        lt.addLast(goles_filtrados, anotacion)

goles_ordenados = sort(goles_filtrados, valor_ord, criterio_date_min)
```

Pruebas Realizadas

Descripción de las pruebas de tiempos de ejecución y memoria utilizada. Incluir descripción del procedimiento, las condiciones, las herramientas y recursos utilizados (librerías, computadores donde se ejecutan las pruebas, entre otros).

Procesadores	**13th Gen Intel(R) Core(TM) i7-13650HX 2.60 GHz**
Memoria RAM	32,0 GB
Sistema Operativo	Windows 11 Home

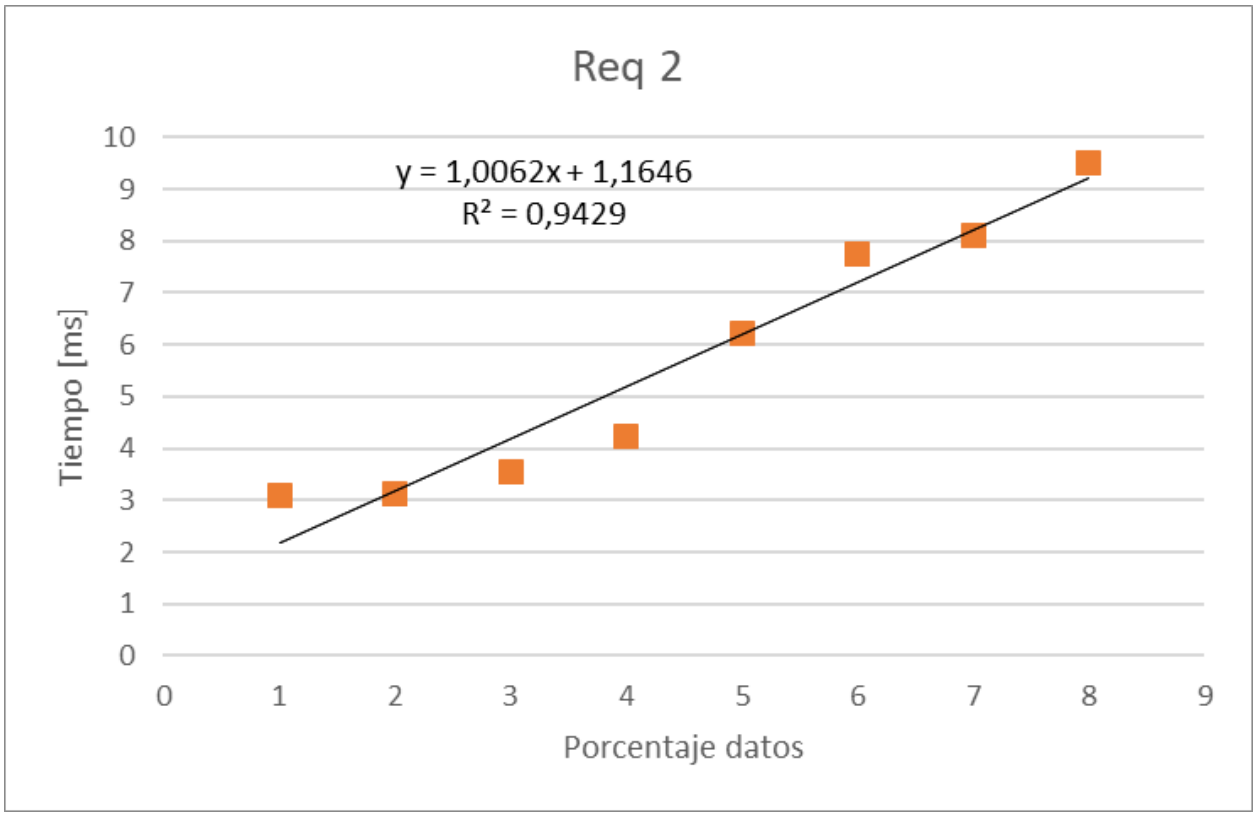
Tablas de datos

ARRAY_LIST

Porcentaje de la muestra [pct]	Merge [ms]
small	3.08

5%	3.12
10%	3.54
20%	4.22
30%	6.22
50%	7.75
80%	8.11
large	9.5

Gráficas



Análisis

Los resultados sugieren que la operación del algoritmo más costosa en cuanto a tiempos es la organización de los datos por fecha utilizando el algoritmo de mergesort, o cualquier tipo de algoritmo de ordenamiento que se decida utilizar. Esta operación tiene una complejidad de O(N log(N)), lo que la convierte en el factor más influyente en la complejidad temporal del algoritmo, haciendo que este no sea muy variable cuando se aumenta el número de datos. Aunque se realizan iteraciones sobre el arreglo de anotaciones, su contribución es menor en comparación con el ordenamiento por fecha.

Requerimiento <<3>>

Descripción

Función req_3(data_structs, consulta_equipo, consulta_fecha_i, consulta_fecha_f, valor_ord)

La función req_3 resuelve el Requerimiento 3, que involucra filtrar y organizar partidos y anotaciones relacionados con un equipo y un rango de fechas.

data_structs: Es un conjunto de estructuras de datos que contiene información sobre partidos y anotaciones.

consulta_equipo: Es el nombre del equipo para el cual se realizará la consulta.

consulta_fecha_i: Es la fecha de inicio del rango de fechas para la consulta.

consulta_fecha_f: Es la fecha de fin del rango de fechas para la consulta.

valor_ord: Es un criterio de ordenamiento que determina cómo se deben ordenar los resultados.

El código lleva a cabo las siguientes tareas:

Filtra los partidos y anotaciones relacionados con el equipo especificado y dentro del rango de fechas especificado.

Organiza los partidos filtrados según el criterio de ordenamiento especificado.

Calcula el número de partidos jugados en casa, el número de partidos jugados como visitante y el tamaño total de la lista de partidos ordenada.

Devuelve la lista de partidos ordenada, así como los números mencionados en el paso anterior.

Entrada	Parámetros necesarios para resolver el requerimiento.
Salidas	Respuesta esperada del algoritmo.
Implementado (Sí/No)	Si se implementó y quien lo hizo.

Análisis de complejidad

La complejidad más influyente se da por las iteraciones anidadas entre partidos y anotaciones. De modo que resulta que, como por cada partido se recorren todas las anotaciones la complejidad termina siendo de O(MN), donde M es el la cantidad de datos en partidos y N la cantidad de datos en anotaciones. A su vez se da una adición significativa al momento de ordenar los datos con el algoritmo mergesort de Nlog(N), pero esta es menor que NM por tanto la complejidad es el factor más influyente de O(NM)

```
for datos_partidos in lt.iterator(partidos_filtrados):
    datos_partidos["penalty"] = "Unknown"
    datos_partidos["own_goal"] = "Unknown"

    for datos_goals in lt.iterator(goles_filtrados):
        if all(datos_goals[llave] == datos_partidos[llave] for llave in ["date", "home_team", "away_team"]):
            datos_partidos["penalty"] = datos_goals["penalty"]
            datos_partidos["own_goal"] = datos_goals["own_goal"]

    del datos_partidos["neutral"]

    lt.addLast(respuesta,datos_partidos)
    if datos_partidos["home_team"] == consulta_equipo:
        n_local += 1
    if datos_partidos["away_team"] == consulta_equipo:
        n_visitante += 1
```

Pruebas Realizadas

Descripción de las pruebas de tiempos de ejecución y memoria utilizada. Incluir descripción del procedimiento, las condiciones, las herramientas y recursos utilizados (librerías, computadores donde se ejecutan las pruebas, entre otros).

Procesadores	**13th Gen Intel(R) Core(TM) i7-13650HX 2.60 GHz**
Memoria RAM	32,0 GB
Sistema Operativo	Windows 11 Home

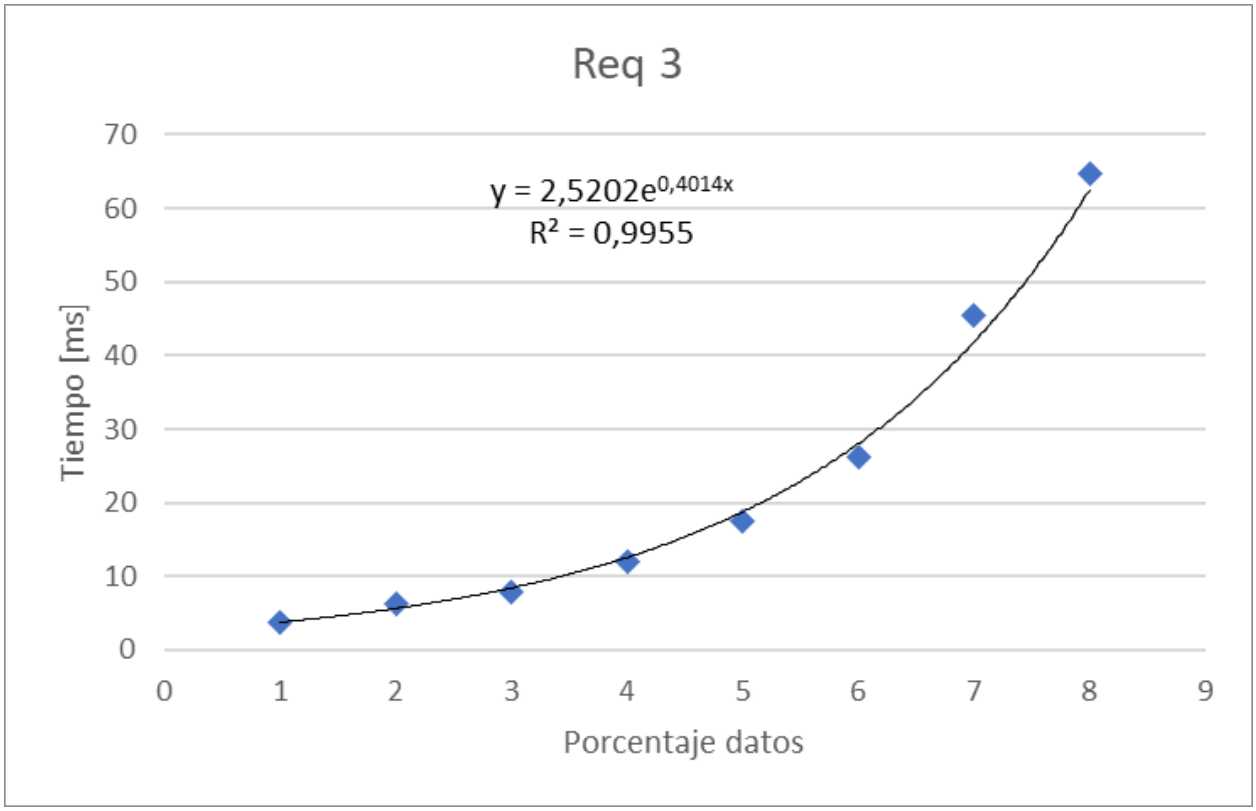
Tablas de datos

ARRAY_LIST

Porcentaje de la muestra [pct]	Merge [ms]
small	3.81
5%	6.36
10%	7.76
20%	11.96
30%	17.61

50%	26.31
80%	45.54
large	64.7

Graficas



Análisis

Para el requerimiento 3 se puede interpretar que los resultados indican que las iteraciones anidadas que hay entre las estructuras de datos de results y goalscorers tienen un impacto considerable en la complejidad temporal del algoritmo, el cual es O(MN). Independientemente de que el ordenamiento de datos contribuye a la complejidad, la influencia es menor en comparación con las iteraciones entre las estructuras de partidos y anotaciones.

Requerimiento <<4>>

Descripción

Función req_4(data_structs, torneo, fecha_i, fecha_f, valor_ord)

La función req_4 resuelve el Requerimiento 4, que implica filtrar y organizar partidos de un torneo específico dentro de un rango de fechas y calcular ciertas estadísticas.

data_structs: Es un conjunto de estructuras de datos que contiene información sobre partidos.

torneo: Es el nombre del torneo para el cual se realizará la consulta.

fecha_i: Es la fecha de inicio del rango de fechas para la consulta.

fecha_f: Es la fecha de fin del rango de fechas para la consulta.

valor_ord: Es un criterio de ordenamiento que determina cómo se deben ordenar los resultados.

El código realiza las siguientes operaciones:

Filtra los partidos relacionados con el torneo especificado y dentro del rango de fechas especificado.

Organiza los partidos filtrados según el criterio de ordenamiento especificado.

Calcula el número de países involucrados en los partidos, el número de ciudades involucradas y el número de partidos decididos por penales.

Devuelve la lista de partidos ordenada, así como los números mencionados en el paso anterior.

Entrada	Parámetros necesarios para resolver el requerimiento.
Salidas	Respuesta esperada del algoritmo.
Implementado (Sí/No)	Si se implementó y quien lo hizo.

Análisis de complejidad

La complejidad del algoritmo está dada por las iteraciones anidadas entre los partidos filtrados y las tandas de penales filtradas de menor longitud que sus estructuras de datos sin filtrar. De modo que la

complejidad se da por (N-k)(M-l), empero como k,l son constante se omiten al momento de definir la complejidad en notación O. La complejidad temporal termina dada por O(MN).

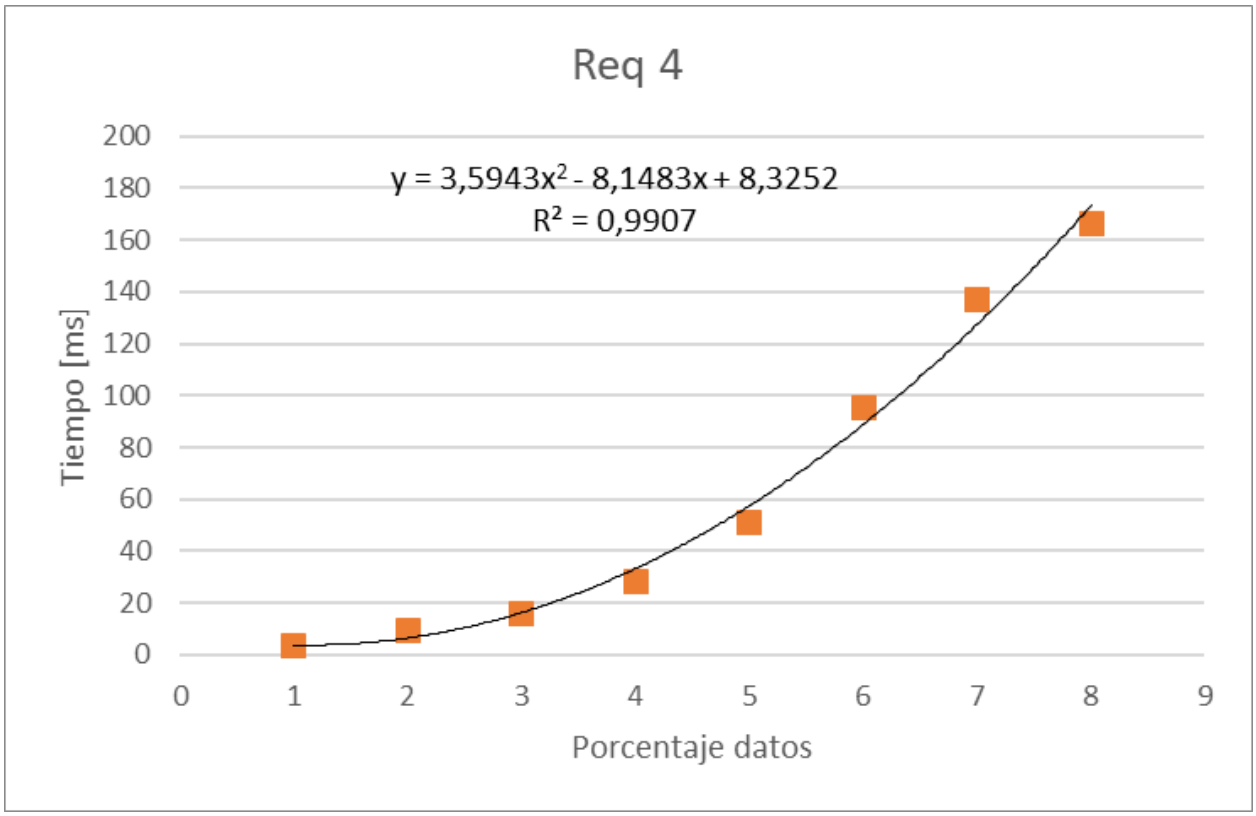
Procesadores	**13th Gen Intel(R) Core(TM) i7-13650HX 2.60 GHz**
Memoria RAM	32,0 GB
Sistema Operativo	Windows 11 Home

Tablas de datos

ARRAY_LIST

Porcentaje de la muestra [pct]	Merge [ms]
small	3.76
5%	9.29
10%	16.11
20%	28.5
30%	50.82
50%	95.32
80%	136.64
large	166.07

Gráficas



Análisis

cuando se toman los tiempos del algoritmo esto nos da resultados que indican que la complejidad del algoritmo está fundamentado en las iteraciones anidadas entre los datos filtrados y depende del tamaño de las estructuras de datos. Aunque inicialmente se mencionan las constantes (k y l) en la expresión de complejidad del algoritmo, no tienen un impacto significativo en la complejidad asintótica, que se establece como O(MN). Esto significa que el tiempo de ejecución del algoritmo crece en función de las longitudes de las estructuras de datos, como se evidencia en la gráfica, lo que puede hacer que el algoritmo sea más lento cuando hay más datos por procesar.

Requerimiento <<5>>

Descripción

Función req_5(data_structs, jugador, fecha_i, fecha_f, valor_ord)

La función req_5 resuelve el Requerimiento 5, que implica analizar las anotaciones de un jugador específico dentro de un rango de fechas y calcular estadísticas.

data_structs: Es un conjunto de estructuras de datos que contiene información sobre anotaciones.

jugador: Es el nombre del jugador cuyas anotaciones se analizarán.

fecha_i: Es la fecha de inicio del rango de fechas para la consulta.

fecha_f: Es la fecha de fin del rango de fechas para la consulta.

valor_ord: Es un criterio de ordenamiento que determina cómo se deben ordenar los resultados.

El código realiza las siguientes tareas:

Filtra las anotaciones del jugador dentro del rango de fechas especificado, excluyendo autogoles.

Organiza las anotaciones filtradas según el criterio de ordenamiento especificado.

Calcula estadísticas como el número de goles totales, el número de goles por penales, el número de autogoles y el número de torneos en los que el jugador ha participado.

Devuelve la lista de anotaciones ordenada y las estadísticas mencionadas en el paso anterior.

Entrada	Parámetros necesarios para resolver el requerimiento.
Salidas	Respuesta esperada del algoritmo.
Implementado (Sí/No)	Si se implementó y quien lo hizo.

Análisis de complejidad

La complejidad del algoritmo está dada por las iteraciones anidadas entre los partidos filtrados y las tandas de penales filtradas de menor longitud que sus estructuras de datos sin filtrar. De modo que la complejidad se da por (N-k)(M-l), empero como k,l son constante se omiten al momento de definir la

complejidad en notación O. La complejidad temporal termina dada por O(MN). En las iteraciones anidadas se le agrega a cada elemento diccionario dentro del arreglo la llave del torneo en que se realizó la anotación y el marcador tanto de visitante como local, contando anotaciones por penales y autogoles. Esto se da únicamente si las llaves de ambas estructuras de datos comparten el mismo valor por sus respectivas llaves con el mismo nombre. Es decir si la anotación tiene el mismo día, local y visitante coinciden y representan el mismo dato. En el requerimiento se actualiza la estructura de datos de anotaciones al haber coincidencias con el fin de sólo iterar después sobre esta en vez de tener que realizar diversas iteraciones animadas para confirmar torneo y otras variables de llave y valor.

```
torneos_anotados = lt.newList('ARRAY_LIST') #Estrcutura de Datos que Elimina Elementos Repetidos
i_penales = 0 #Contador de Penaltys
i_autogoles = 0 #Contador de Autogoles
for anotacion_filtrada in lt.iterator(anotaciones_f_jugador):
    for partido_filtrado in lt.iterator(partidos_f_fecha):
        if all(anotacion_filtrada[llave] == partido_filtrado[llave] for llave in ["date", "home_team", "away_team"]):

            anotacion_filtrada["tournament"] = partido_filtrado["tournament"]
            anotacion_filtrada["home_score"] = partido_filtrado["home_score"]
            anotacion_filtrada["away_score"] = partido_filtrado["away_score"]

            if anotacion_filtrada["penalty"] == "True":
                i_penales += 1
            if anotacion_filtrada["own_goal"] == "True":
                i_autogoles += 1
            if lt.isPresent(torneos_anotados, partido_filtrado["tournament"]) == 0:
                lt.addLast(torneos_anotados, partido_filtrado["tournament"])
```

Pruebas Realizadas

Descripción de las pruebas de tiempos de ejecución y memoria utilizada. Incluir descripción del procedimiento, las condiciones, las herramientas y recursos utilizados (librerías, computadores donde se ejecutan las pruebas, entre otros).

Procesadores	**13th Gen Intel(R) Core(TM) i7-13650HX 2.60 GHz**
Memoria RAM	32,0 GB
Sistema Operativo	Windows 11 Home

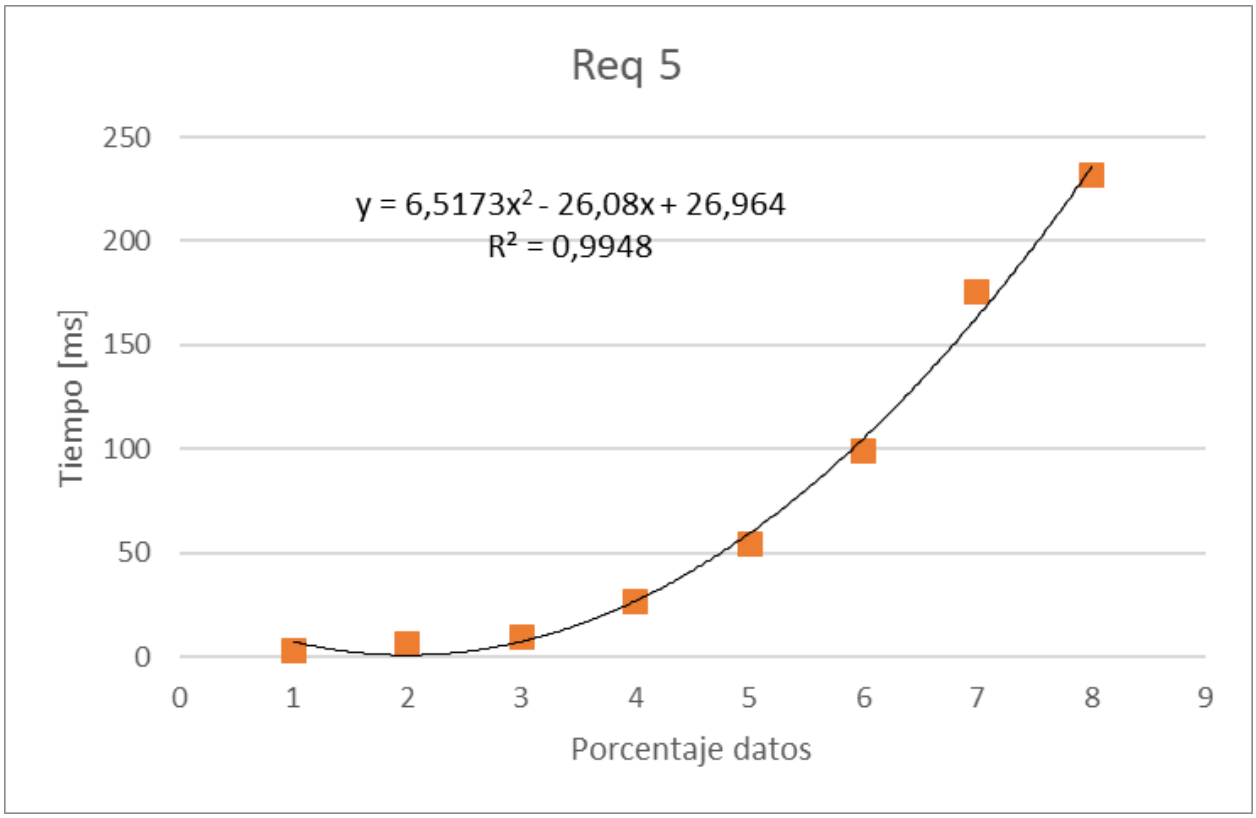
Tablas de datos

ARRAY_LIST

Porcentaje de la muestra [pct]	Merge [ms]
small	3.41
5%	6.56
10%	9.5

20%	26.85
30%	54
50%	99.26
80%	175.39
large	231.374

Gráficas



Análisis

De los resultados de la gráfica y de lo que pudimos analizar previamente, podemos decir que el algoritmo se adapta al tamaño de los datos, con una complejidad O(MN) que refleja el tiempo necesario para procesar estructuras de datos de diferentes tamaños. Además, el algoritmo realiza operaciones adicionales para enriquecer los datos sólo cuando se cumplen condiciones específicas de coincidencia entre las estructuras de datos. Se evidencia que el algoritmo es potencial de x^2

Requerimiento <<6>>

Descripción

Función req_6(data_structs, num_equipos, torneo, fecha_i, fecha_f, valor_ord)

La función req_6 resuelve el Requerimiento 6, que implica filtrar y organizar partidos de un torneo específico dentro de un rango de fechas, calcular estadísticas de los equipos involucrados y mostrar los equipos más destacados.

data_structs: Es un conjunto de estructuras de datos que contiene información sobre partidos.

num_equipos: Es el número de equipos a mostrar en los resultados.

torneo: Es el nombre del torneo para el cual se realizará la consulta.

fecha_i: Es la fecha de inicio del rango de fechas para la consulta.

fecha_f: Es la fecha de fin del rango de fechas para la consulta.

valor_ord: Es un criterio de ordenamiento que determina cómo se deben ordenar los resultados.

El código lleva a cabo las siguientes operaciones:

Filtra los partidos relacionados con el torneo especificado y dentro del rango de fechas especificado.

Organiza los partidos filtrados según el criterio de ordenamiento especificado.

Calcula estadísticas de los equipos involucrados, como el número de partidos jugados, el número de goles marcados y el número de victorias.

Muestra los equipos más destacados en función de las estadísticas calculadas y limita la cantidad de equipos a mostrar a num_equipos.

Devuelve la lista de equipos destacados y las estadísticas mencionadas en el paso anterior.

Entrada	Parámetros necesarios para resolver el requerimiento.
Salidas	Respuesta esperada del algoritmo.
Implementado (Sí/No)	Si se implementó y quien lo hizo.

Análisis de complejidad

La complejidad se da por las iteraciones anidadas while and for de partidos como O(NM). N y M son la longitud de las estructuras de partidos y anotaciones. El resto de adiciones a la complejidad son de N u M entonces la más influyente es O(NM) siendo mayor a N y M. Aquí se busca eliminar todas las veces que los archivos de partidos y resultados no posean coincidencias revisadas por fecha, equipo local y visitante. A pesar de que se realizan búsquedas binarias con el fin de hacer el código más eficiente con una complejidad temporal de logN esto no se logra pq el factor con mayor orden polinómico es de O(NM). Además es plausible esclarecer que el tamaño de los archivos de partidos y anotaciones es similar, de manera que se podría aproximar a un N=M con una complejidad de $O(N^2)$

```
index_anotacion_filtrada = 1
while index_anotacion_filtrada < lt.size(anotaciones_f_fecha) + 1:
    anotacion_filtrada = lt.getElement(anotaciones_f_fecha, index_anotacion_filtrada)
    coincidencia = False

    for partido_filtrado in lt.iterator(partidos_f_fecha_torneo):
        if all(anotacion_filtrada[llave] == partido_filtrado[llave] for llave in ["date", "home_team", "away_team"]):
            anotacion_filtrada["tournament"] = partido_filtrado["tournament"]
            coincidencia = True

    if not coincidencia:
        lt.deleteElement(anotaciones_f_fecha, index_anotacion_filtrada)
    else:
        index_anotacion_filtrada += 1
```

Pruebas Realizadas

Descripción de las pruebas de tiempos de ejecución y memoria utilizada. Incluir descripción del procedimiento, las condiciones, las herramientas y recursos utilizados (librerías, computadores donde se ejecutan las pruebas, entre otros).

Procesadores	**13th Gen Intel(R) Core(TM) i7-13650HX 2.60 GHz**
Memoria RAM	32,0 GB
Sistema Operativo	Windows 11 Home

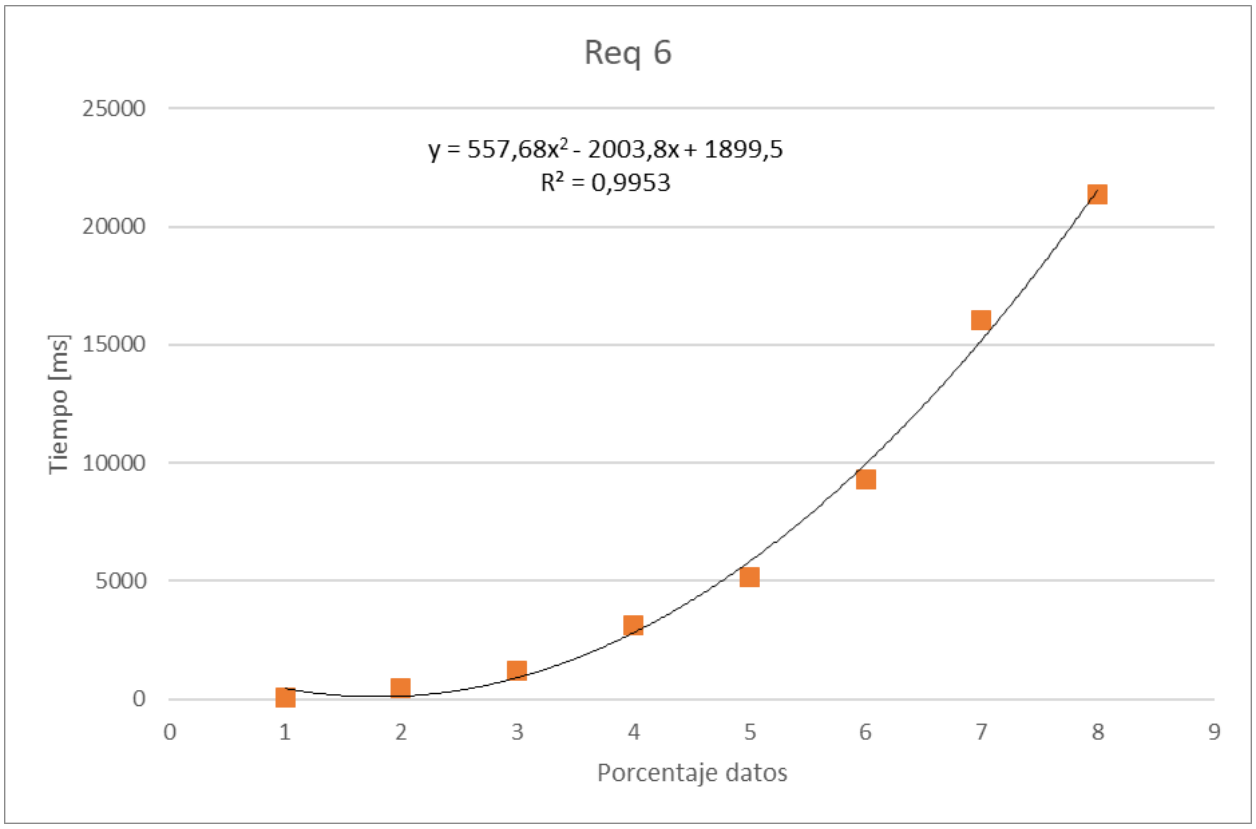
Tablas de datos

ARRAY_LIST

Porcentaje de la muestra [pct]	Merge [ms]
small	88.175
5%	488.794
10%	1183.486
20%	3139.531

30%	5171.828
50%	9325.369
80%	16055.915
large	21373.162

Graficas



Análisis

Luego de hacer el estudio de tiempos del algoritmo. Se destaca que la complejidad del algoritmo se determina principalmente por las iteraciones anidadas a través de las estructuras de datos "partidos" y "anotaciones", lo que resulta en una complejidad de O(NM). A pesar de los intentos de optimización, y la complejidad que fue poder obtener los resultados de la manera más eficiente, esta sigue siendo la parte más costosa del algoritmo debido a la cantidad de datos que se deben comparar, sin embargo es un precio a pagar por la complejidad del algoritmo. Además, el tamaño similar de los archivos de entrada sugiere que la complejidad podría aproximarse a $O(N^2)$. Realmente se buscan coincidencias con torneos y al ser estos reducidos en comparación con otras variables de llave valor esto facilita la implementación

del código reduciendo su complejidad, pues se debe actualizar y crear cada elemento torneo menos veces que otras variables como nombres de equipos y jugadores.

Requerimiento <<7>>

Descripción

Función req_7(data_structs, n_jugadores, fecha_i, fecha_f, valor_ord)

La función req_7 resuelve el Requerimiento 7, que implica analizar las anotaciones y estadísticas de los jugadores dentro de un rango de fechas y mostrar los jugadores más destacados.

data_structs: Es un conjunto de estructuras de datos que contiene información sobre anotaciones.

n_jugadores: Es el número de jugadores a mostrar en los resultados.

fecha_i: Es la fecha de inicio del rango de fechas para la consulta.

fecha_f: Es la fecha de fin del rango de fechas para la consulta.

valor_ord: Es un criterio de ordenamiento que determina cómo se deben ordenar los resultados.

El código realiza las siguientes tareas:

Filtra las anotaciones dentro del rango de fechas especificado, excluyendo autogoles.

Organiza las anotaciones filtradas según el criterio de ordenamiento especificado.

Calcula estadísticas de los jugadores, como el número de partidos jugados, el número de goles marcados, el número de goles por penales y el número de autogoles.

Muestra los jugadores más destacados en función de las estadísticas calculadas y limita la cantidad de jugadores a mostrar a n_jugadores.

Devuelve la lista de jugadores destacados y las estadísticas mencionadas en el paso anterior.

Cada una de estas funciones cumple un propósito específico dentro del programa y realiza las operaciones necesarias para resolver los requerimientos correspondientes.

Entrada	Parámetros necesarios para resolver el requerimiento.
Salidas	Respuesta esperada del algoritmo.
Implementado (Sí/No)	Si se implementó y quien lo hizo.

Análisis de complejidad

La complejidad está dada por las interacciones anidadas, siendo dod. De modo que la complejidad es de O(NM). Sin embargo hay dos de estas iteraciones anidadas además de las demás adiciones de complejidad de N para iteraciones lineales, Nlog(N) para ordenamiento y log(N) de búsquedas binarias. O(NM) es la mayor entonces se toma esa. Se crean elementos en forma de diccionario para guardar cada dato relevante de los jugadores del archivo de anotaciones en un formato llave y valor y se insertan a un arreglo en caso de no estar ya en este. Si ya están en este se crea la entrada y se actualizan sus datos por cada anotación y partido en que participaron. Esto es principalmente lo que incrementa la complejidad temporal pues se de forma N por M que se puede aproximar a N^2 pues N es casi igual a M. No obstante, cada llave de jugador a diferencia de cada llave de torneo en el req 6 es incluso más singular, pues existen más goleadores que torneos oficiales de fútbol.

```
for partido in lt.iterator(todas_partidos_borrados):
    fecha_partido = partido["date"]

    if fecha_i <= fecha_partido and fecha_partido <= fecha_f and partido["tournament"] != "Friendly":
        #verifica si el resultado cumple los criterios
        nombre_partido = str(partido["date"]) + str(partido["home_team"]) + str(partido["away_team"])
        if lt.isPresent(nombres_partidos, nombre_partido) == 0:
            lt.addLast(nombres_partidos,nombre_partido)

        if lt.isPresent(nombres_torneos, partido["tournament"]) == 0:
            lt.addLast(nombres_torneos, partido["tournament"])

        goles = goles + int(partido["home_score"]) + int(partido["away_score"])

        posizq = busq_bin_izq(anotaciones_organizadas,partido["date"])
        posder = busq_bin_der(anotaciones_organizadas,partido["date"])

        #busca los partidos jugaores que hay
        for i in range(posizq,posder+1):

            anotacion_encontrada = lt.getElement(anotaciones_organizadas,i)
            nombre_jugador = anotacion_encontrada["scorer"]
            if posizq > 0 and posder > 0:
                if partido["home_team"] == anotacion_encontrada["home_team"] and partido["away_team"] == anotacion_encontrada["away_team"]:

                    equipo_jug = anotacion_encontrada["team"]

                    jugador = get_data(informacion_jugadores, nombre_jugador, "Nombre")

                    if jugador == None:
                        jugador = {
                            "Nombre": nombre_jugador,
                            "puntos_totales": 0,
                            "goles_totales": 0,
                            "goles_penalty":0,
                            "autogoles":0,
                            "tiempo_promedio":0,
                            "total_torneos":0,
                            "goles_victoria":0,
                            "goles_derrota":0,
                            "goles_empate":0,
                            "ultimo_gol": 0,
                            "fecha_gol":"1000-01-01"
                        }
```

Pruebas Realizadas

Descripción de las pruebas de tiempos de ejecución y memoria utilizada. Incluir descripción del procedimiento, las condiciones, las herramientas y recursos utilizados (librerías, computadores donde se ejecutan las pruebas, entre otros).

Procesadores	**13th Gen Intel(R) Core(TM) i7-13650HX 2.60 GHz**
Memoria RAM	32,0 GB
Sistema Operativo	Windows 11 Home

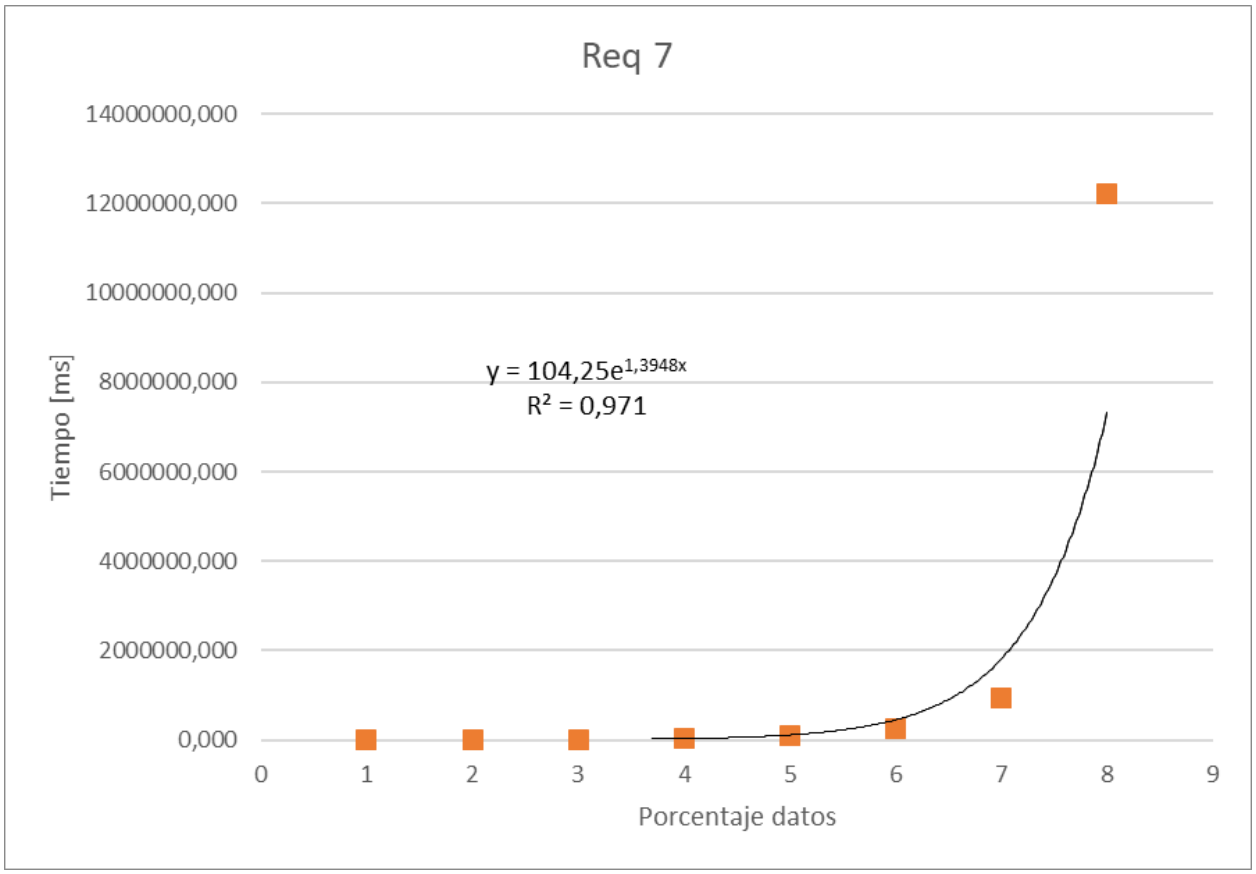
Entrada	**Tiempo (s)**

Tablas de datos

ARRAY_LIST

Porcentaje de la muestra [pct]	Merge [ms]
small	156.257
5%	3296.039
10%	12082.448
20%	46963.404
30%	101501.356
50%	260825.820
80%	947344.454
large	12229775.265

Graficas



Análisis

Cuando hicimos el estudio de tiempo de este algoritmo, inmediatamente caímos en cuenta que este no era eficiente para obtener los resultados requeridos. Sin embargo, la complejidad temporal de este algoritmo está dominada por las iteraciones anidadas sobre dos estructuras de datos, lo que resulta en una complejidad de O(NM). A pesar de otras operaciones con complejidades lineales y logarítmicas, O(NM) es la complejidad más alta y, por lo tanto, se considera la más influyente en el tiempo de ejecución del algoritmo, el cual termina siendo muy lento cuando la cantidad de datos utilizada es alta.