ANÁLISIS DEL RETO

Daniel Camilo Quimbay Velásquez, 202313861, d.quimbay@uniandes.edu.co

Julián David Contreras Pinilla, 202223394, j.contrerasp@uniandes.edu.co

Funciones comunes

Las siguientes funciones son usadas por todas o la mayoría de requerimientos, por lo que las agregaremos en este apartado para evitar repeticiones.

Búsqueda binaria por nombre

```
def binary_search_by_name(data_structs, name):
    """
    Busqueda binaria para encontrar un nombre en una lista
    """
    low = 1
    high = lt.size(data_structs)

    while low <= high:
        mid = (low + high) // 2
        team = lt.getElement(data_structs, mid)
        mid_name = team['name'].lower()

        if mid_name == name:
            return mid
        elif mid_name < name:
            low = mid + 1
        else:
            high = mid - 1
    return -1
```

Esta función recibe un arreglo de n elementos por parámetro y el nombre del elemento a buscar. Se asume que la lista está ordenada alfabéticamente de la A a la Z, por lo que por medio de búsqueda binaria se obtiene la posición del elemento con una complejidad O(n).

En caso de no encontrar el elemento retorna –1

Búsqueda binaria de fecha de inicio

```python
def binary_search_start_date(data_structs, start):
    """
    Busqueda binaria para encontrar una fecha de inicio
    """
    low = 1
    high = lt.size(data_structs)

    recent = (lt.firstElement(data_structs))['date']
    oldest = (lt.lastElement(data_structs))['date']

    if start > recent:
        return -1

    #Confirmar que la fecha está en la lista en una posición intermedia
    if start >= oldest:

        #Buscar un día antes para evitar errores no encontrar la fecha mínima que coincide
        prev = start - timedelta(days=1)

        i = lt.size(data_structs)
        #Busqueda binaria
        while low <= high:
            mid = (low + high) // 2
            team = lt.getElement(data_structs, mid)
            mid_date = team['date']
            if mid_date == prev:
                i = mid
                #Salir de un bucle infinito que no cambia el low
                low = lt.size(data_structs) + 1
            elif mid_date > prev:
                low = mid + 1
            else:
                high = mid - 1

            if low == high:
                i = mid
            i = mid

        find = False
        #Iterar hacia atrás para encontrar la primera fecha que coincide
        while not find:
            result = lt.getElement(data_structs, i)
            date = result['date']
            if date >= start:
                #Se encuentra la fecha
                return i
            else:
                #Se sigue iterando
                i -= 1

            if i <= 0:
                return -1
    else:
        #Retornar posición de la fecha más antigua
        return lt.size(data_structs)
```

Esta función recibe por parámetro un arreglo que contiene los resultados de los partidos ordenados por fecha, de más reciente a más antiguo.

Para encontrar una fecha de inicio primero restamos un día a la fecha deseada, para evitar errores que al encontrar una fecha se ignoren otros resultados con la misma fecha.

Al encontrar una fecha que coincide cambiamos el parámetro low para salir del ciclo, y en caso de no encontrar ninguna coincidencia se guarda la última mitad a la que se accedió. A continuación se empieza a iterar hacia atrás en la lista, ya que al buscar un día antes, para encontrar el mínimo que cumpla con el rango deseado debemos ir a indices con fechas más recientes.

En el momento que la fecha que se está revisando es mayor o igual a la que se está buscando, significa que se encontró la fecha mínima que cumple con el rango. En caso de pasar del inicio de la lista y no encontrar una fecha que cumpla con la condición se devuelve –1 para indicar que no se pudo encontrar.

La complejidad de esta función al ser de búsqueda binaria es O(log(n)).

Búsqueda binaria de fecha final

```
def binary_search_end_date(data_structs, end):
    """
    Busqueda binaria para encontrar una fecha de final
    """
    low = 1
    high = lt.size(data_structs)

    recent = (lt.firstElement(data_structs))['date']
    oldest = (lt.lastElement(data_structs))['date']

    #Confirmar si la fecha existe en el rango de la estructura
    if end < oldest:
        return -1
    #Confirmar que la fecha está en una posición intermedia
    if  end <= recent:
        #Buscar un día después para evitar errores no encontrar la fecha máxima que coincide
        next = end + timedelta(days=1)
        next_id = 1
        while low <= high:
            mid = (low + high) // 2
            team = lt.getElement(data_structs, mid)
            mid_date = team['date']
            if mid_date == next:
                next_id = mid
                pass
            elif mid_date > next:
                low = mid + 1
            else:
                high = mid - 1
            if low == high:
                next_id = mid
                pass
        find = False
        i = next_id
        #Iterar hacia adelante para encontrar la primera fecha que coincide
        while not find:
            #Confirmar que el indice existe
            if i > 0 and i < lt.size(data_structs):

                date = (lt.getElement(data_structs, i))['date']
                if date <= end:

                    return i
                else:
                    i += 1
            else:
                return 1
    else:
        return 1
```

Esta función es muy similar a la anterior, pero en este caso se busca el rango máximo, por lo que ahora se le suma un día a la fecha deseada y se busca esa fecha. Apenas se sale del primer ciclo se empieza a iterar hacia adelante, debido a que ahora debemos ir avanzando en indices con fechas menores para

encontrar la fecha máxima que cumple con el rango. En caso de salirse del tamaño de la lista se devuelve –1.

Esta función, como la anterior, tiene una complejidad de O(log(n)).

Requerimiento 01

Descripción

```python
def req_1(data_structs, n_results, team_name, condition):
    """Función que resuelve el requerimiento 1, encuentra los últimos N partidos jugados en local, visitante

    Args:
        data_structs (ARRAY_LIST): _description_
        n_results (int): Cantidad de partidos a encontrar
        team_name (str): Nombre del equipo del que se quiere la información
        condition (str): Condición entre local, visitante o neutro para devolver la información

    Returns:
        sublist(ARRAY_LIST): Sublista con los últimos N partidos
        total (int): Cantidad de partidos encontrados

    """
    # TODO: Realizar el requerimiento 1
    t_name = team_name.lower()

    #Estructura separada por equipos
    teams = data_structs['teams']

    #Busqueda binaria para encontrar el equipo que busca el usuario
    pos_team = binary_search_by_name(teams, t_name)

    #Solo los partidos del equipo
    team_data = (lt.getElement(data_structs['teams'], pos_team))['results']

    #Creación lista auxiliar
    filtered_list = lt.newList("ARRAY_LIST")

    #Filtrar búsqueda por condición
    for result in lt.iterator(team_data):
        if condition == "local":
            if t_name == result["home_team"].lower():
                lt.addLast(filtered_list, result)
        elif condition == "visitante":
            if t_name == result["away_team"].lower():
                lt.addLast(filtered_list, result)
        else:
            lt.addLast(filtered_list, result)

    #Filtrar ultimos N partidos
    total = lt.size(filtered_list)
    if n_results <= lt.size(filtered_list):
        sublist = lt.subList(filtered_list, 1, n_results)
        return sublist, total
    else:
        return filtered_list, total
```

Lo primero que hacemos en el requerimiento 1 es que todos los caracteres del nombre del equipo se pasen a minúscula. Luego, hacemos la estructura separada por equipos y hacemos una búsqueda binaria para encontrar el equipo que pide el usuario. Después con el team_data lo que hacemos es que nos da el elemento de la posición específica de la lista de data_structs[“teams”] y cuando hayamos el equipo nos muestras sus partidos, es decir nos da la información de los partidos del equipo. A continuación se crea una lista auxiliar. Después hacemos una iteración para saber pues la condición del equipo en los partidos y si es local pues se buscan todos los partidos locales y se añade a la lista auxiliar. También es lo mismo para la condición de visitante y si no se cumple ninguna de las dos pues es indiferente y pues

queda la lista auxiliar vacía. Después creamos una variable que nos da el número de elementos de la lista auxiliar. Luego miramos si la cantidad de partidos a encontrar es menor o igual al número de elementos de la lista auxiliar y si es así, pues se crea una sublista con los últimos partidos y retorna los últimos partidos N a consutltar y si es mayor la cantidad de partidos a encontrar al número de elementos de la lista auxiliar, pues retorna los partidos de la lista auxiliar.

Entrada	- Estructuras de datos del modelo - El número (N) de partidos de consultar - Nombre del equipo - La condición del equipo en los partidos consultados (Local, Visitante, o Indiferente).
Salidas	- Retorna una sublista con los últimos partidos jugados (Local, Visitante, o Indiferente). - Retorna la cantidad de partidos encontrados.
Implementado (Sí/No)	Sí. Implementado por Daniel Camilo Quimbay Velásquez

Análisis de complejidad

Los procesos de (getElement), lt.size(), lt.newList(), lt.addLast() y lt.sublist() los consideraremos son procesos con complejidad promedio O(1), por lo que no hacen un efecto notorio a la complejidad total.

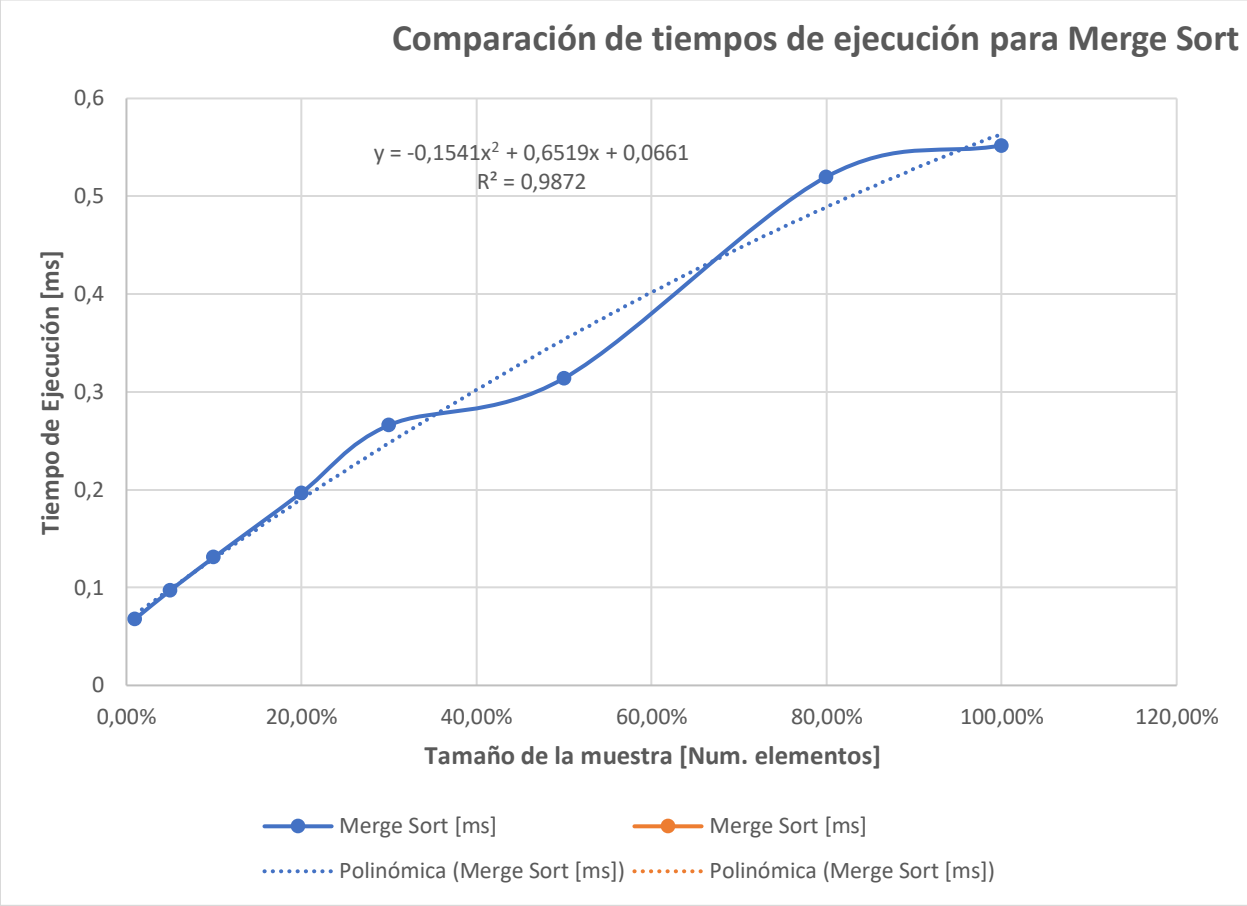
Pasos	**Complejidad**
Obtener el elemento (getElement)	O(1)
Crea una lista vacía lt.newList()	O(1)
Agrega un elemento en la última posición lt.addLast()	O(1)
Informa el número de elementos de la lista (size)	O(1)
Una lista más pequeña lt.sublist()	O(1)
TOTAL	***O(n)***

Para el caso del requerimiento 1 el peor caso con Array_LIST es n porque estamos recorriendo la iteración para saber pues la condición del equipo en los partidos y si es local pues se buscan todos los partidos locales y se añade a la lista auxiliar. También es lo mismo para la condición de visitante y si no se cumple ninguna de las dos pues es indiferente y pues queda la lista auxiliar vacía.

Entrada	**Tiempo (ms)**
small	0.068
5 pct	0.097
10 pct	0.131
20 pct	0.197
30 pct	0.266
50 pct	0.314
80 pct	0.520

large	0.552

Gráficas



Análisis

La gráfica confirma lo que esta en el análisis, ya que tiene tendencia de orden temporal n y este comportamiento se puede evidenciar experimentalmente en la gráfica. Ya que, gracias a que los datos no se encuentran tan dispersos con respecto a la línea de tendencia, la curva coincide con el comportamiento lineal esperado.

Requerimiento 02

Descripción

```python
def req_2(data_structs, n_goals, name):
    """Función que encuentra los últimos N goles anotados por un jugador específico

    Args:
        data_structs (ARRAY_LIST): Catálogo con la información de los partidos
        n_goals (int): Cantidad de anotaciones para buscar
        name (str): Nombre del jugador que se desea buscar

    Returns:
        goals (ARRAY_LIST): Lista con los datos encontrados
        lt.size(goals) (int): Cantidad de resultados encontrados
    """

    #Filtrar los partidos del jugador
    scorers = data_structs['scorers']
    posscorer = binary_search_by_name(scorers, name)
    if posscorer == -1:
        return None, 0
    results = (lt.getElement(scorers, posscorer))['results']

    #Lista auxiliar
    goals = lt.newList(datastructure='ARRAY_LIST', cmpfunction=compare_id)

    #Iteración hacia atrás para obtener de más antiguo a más reciente
    size = lt.size(results)
    for i in range(size, (size - n_goals) + 1, -1):
        if i < 1:
            #Caso en el que se llega al final de la lista
            return goals, lt.size(goals)
        else:
            result = lt.getElement(results, i)
            lt.addLast(goals, result)
    return goals, lt.size(goals)
```

En este requerimiento ‘scorers’ es un arreglo donde cada posición es un diccionario que contiene el nombre del jugador y un arreglo con todos los partidos que disputó el jugador. Primero, con una búsqueda binaria se halla la posición con la información del jugador en el arreglo, la cual al ser binaria es log(n). Después, si se encontró la posición se realiza una iteración hacia atrás, ya que como los partidos están ordenados del más reciente al más antiguo, esto nos permite recorrer de manera eficiente solo los datos que se buscan sin que se tenga que acceder a los datos que no nos interesan.

Además, hay varios puntos en el código donde se comprueban casos en donde no se encuentra el jugador o la cantidad m de goles es mayor al tamaño de la lista, por lo que se toman medidas como acabar la ejecución para evitar errores.

Entrada	Estructuras de datos del modelo, numero de goles a consultar, nombre del jugador

Salidas	Un arreglo de la cantidad de goles que anotó el jugador con su respectiva información
Implementado (Sí/No)	Si. Implementado por Daniel Quimbay

Análisis de complejidad

N es la cantidad de jugadores que se encontraron en la carga de datos, mientras que m es la cantidad de partidos. Considerando que la búsqueda binaria tiene mucho menor complejidad temporal que obtener los últimos partidos, la complejidad en Big O sería de O(m), ya que en el peor caso se tendría que recorrer todos los partidos del jugador.

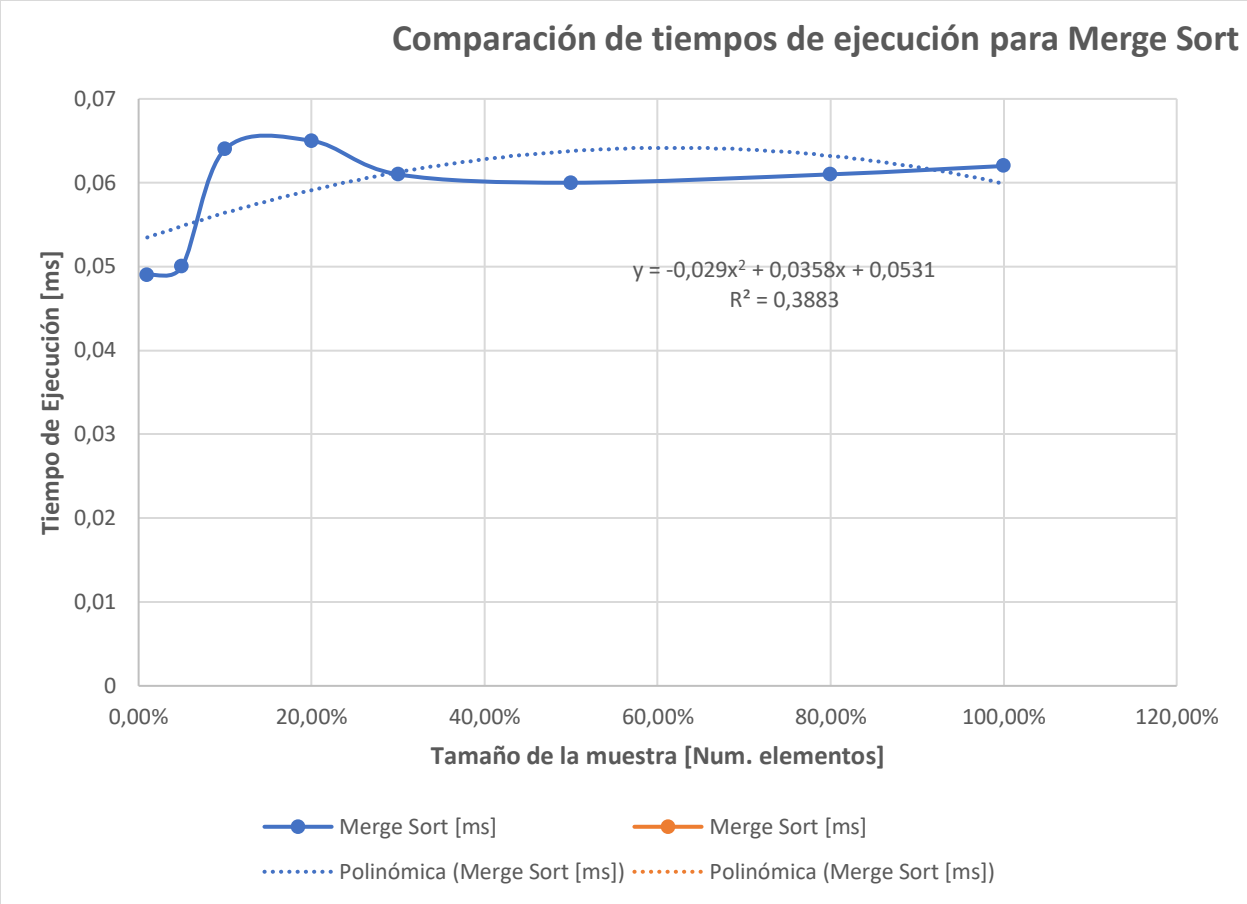
Los procesos de lt.getElement(), lt.size(), lt.newList() y lt.addLast() los consideraremos son procesos con complejidad promedio O(1), por lo que no hacen un efecto notorio a la complejidad total.

Pasos	**Complejidad**
Búsqueda binaria del jugador	O(log(n))
Obtener los m primeros goles del jugador	O(m)
TOTAL	***O(m)***

Pruebas Realizadas

Entrada	**Tiempo (ms)**
small	0.049
5 pct	0.050
10 pct	0.064
20 pct	0.065
30 pct	0.061
50 pct	0.060
80 pct	0.061
large	0.062

Gráfica Req 2



Análisis

A partir de un tamaño de archivo del 30% la búsqueda se estabiliza y mantiene una trayectoria aparentemente lineal, por lo que concuerda con el análisis. Por la manera en como está construida la estructura con la que se maneja el requerimiento se supondría que debería ser lineal conforme la cantidad n de goles aumenta. La forma de la gráfica puede deberse a que con los datos que ingresamos, a pesar de ir subiendo la cantidad de goles solicitados al programa, el cambio no era tan grande como para evidenciar muchos cambios en el tiempo.

Requerimiento 03

Descripción

```python
def req_3(data_structs, name, inicial, final):
    """Función que consulta los partidos que jugó un equipo durante un periodo específico

    Args:
        data_structs (ARRAY_LIST): Catálogo con la información de los resultados
        name (str): Nombre del jugador que se desea buscar
        inicial (datetime): Fecha mínima de búsqueda
        final (datetime): Fecha máxima de búsqueda

    Returns:
        sublist: Lista con los partidos que jugó el jugador
        home: Cantidad de partidos como local
        away: Cantidad de partidos como visitante
    """
    # TODO: Realizar el requerimiento 3

    #Estructura separada por equipos
    teams = data_structs['teams']

    #Búsqueda del equipo deseado
    pos_team = binary_search_by_name(teams, name)
    if pos_team == -1:
        return None, 0, 0
    results_team = (lt.getElement(teams, pos_team))['results']

    #Búsqueda de rangos de fechas
    pos_date_inicial = binary_search_start_date(results_team, inicial)
    pos_date_final = binary_search_end_date(results_team, final)

    #No se encontraron las fechas
    if pos_date_final == -1 or pos_date_inicial == -1:
        return None, 0, 0

    #Iniciar contadores de local y visitante
    home = 0
    away = 0
    nlower = name.lower()

    #Sublista filtrada
    sublist = lt.newList(datastructure='ARRAY_LIST', cmpfunction=compare_id)

    #Añadir resultados y sumar contadores
    for i in range(pos_date_final, pos_date_inicial + 1):

        result = lt.getElement(results_team, i)
        if result['home_team'].lower() == nlower:
            home += 1
        else:
            away +=1
        lt.addLast(sublist, result)

    return (sublist, home, away)
```

En este requerimiento se usó un arreglo de nombre teams, el cuál es un arreglo en el que cada posición contiene el nombre de un equipo y todos los partidos que jugó ese equipo, esta estructura ya viene con la información de las columnas si se encontraron del archivo de goalscorers.

Primero, con la búsqueda binaria se encuentra en el arreglo la posición en la que se encuentra el equipo. A continuación, se usan las funciones de búsqueda para obtener el rango mínimo y máximo de fechas en las que se buscan los partidos del equipo.

Ya con los índices de las fechas se puede iterar desde la fecha más reciente hasta la más antigua, y se tiene la garantía de que todos los partidos en ese rango pertenecen al equipo y al rango deseado, por lo que se van agregando a una lista que después se va a retornar al usuario. Además, se hace un contador iniciado en 0 que va guardando la cantidad de partidos que el equipo jugó como local y como visitante.

Entrada	Estructuras de datos del modelo Nombre del equipo Fecha de inicial del rango Fecha fiinal del rango
Salidas	Un arreglo con los resultados de la búsqueda El total de partidos jugados como local El total de partidos jugados como visitante
Implementado (Sí/No)	Si. Implementado por Daniel Quimbay

Análisis de complejidad

Para este requerimiento tomaremos *n* como la cantidad de equipos, *m* como la cantidad total de partidos que jugó el equipo y *o* como la cantidad de partidos que entran en el rango de fechas.

Podemos decir que aunque n >> m >= o las búsquedas binarias tienen tiempos muy reducidos, y el peor caso es cuando la cantidad o de partidos que están en el rango es la misma cantidad m de partidos que jugó el equipo, por lo que la complejidad total sería O(m)

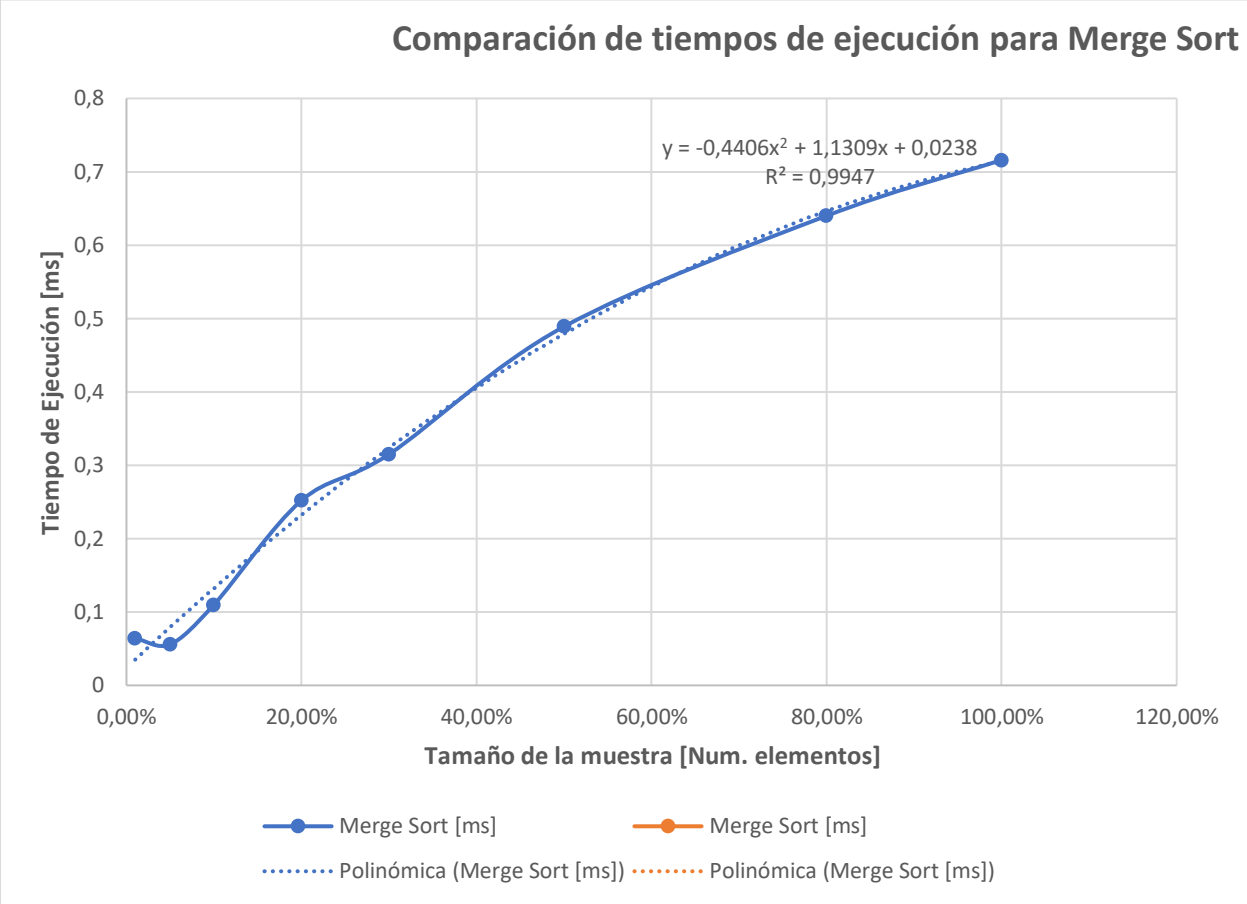
Pasos	**Complejidad**
Búsqueda binaria del equipo	O(log(n))
Búsqueda binaria fecha de inicio	O(log(m))
Búsqueda binaria fecha final	O(log(m))
Añadir	O(o)
TOTAL	***O(m)***

Pruebas Realizadas

Entrada	**Tiempo (ms)**
small	0.064
5 pct	0.056
10 pct	0.110

20 pct	0.252
30 pct	0.315
50 pct	0.489
80 pct	0.640
large	0.716

Gráfica Req 3



Análisis

En la gráfica podemos ver que la regresión lineal devolvió una función cuadrática en decrecimiento. Sin embargo, creemos que este decrecimiento es porque conforme los datos avanzan, las búsquedas binarias tienen menos impacto en tiempo en comparación con como aumenta la base de datos. Por lo que ignorando esta ligera variación el algoritmo sería una función lineal. Además, la pendiente de la gráfica es muy leve, por lo que se podría decir que se comporta aproximadamente como lo esperado.

Requerimiento 04

Descripción

```python
def req_4(control, nombre_torneo, fecha_inicial, fecha_final):
    """Partidos relacionados con un torneo durante un periodo especifico

    Args:
        control (dict): Catálogo que contiene los ADT con la información de resultados
        nombre_torneo (str): Nombre del torneo que se quiere buscar
        fecha_inicial (datetime): Fecha mínima de búsqueda
        fecha_final (datetime): Fecha máxima de búsqueda

    Returns:
        lista_final_results: Lista con los resultados de la búsqueda
        num_ciudades (int): Total de ciudades donde se disputaron partidos del torneo
        num_paises (int): Total de paises donde se disputaron los partidos del torneo
        total_matches (int): Cantidad de partidos encontrados
        penalties (int): Total de partidos definidos por cobros de penal
    """
    lista_results = control["model"]["results"]
    lista_shootouts = control["model"]["shootouts"]

    lista_final_results = lt.newList("ARRAY_LIST")
    lista_final_shootouts = lt.newList("ARRAY_LIST")

    for dato in lt.iterator(lista_results):
        fecha_dato = dato["date"]

        if fecha_dato <= fecha_final and fecha_dato >= fecha_inicial and dato["tournament"] == nombre_torneo:
            dato["winner"] = "Unknown"
            lt.addLast(lista_final_results, dato)

    for dato in lt.iterator(lista_shootouts):
        fecha_dato = dato["date"]

        if fecha_dato <= fecha_final and fecha_dato >= fecha_inicial :
            lt.addLast(lista_final_shootouts, dato)

    penaltis = 0
    for dato in lt.iterator(lista_final_shootouts):
        for dato_result in lt.iterator(lista_final_results):
            if dato["home_team"] == dato_result["home_team"] and dato["away_team"] == dato_result["away_team"]:
                penaltis += 1
                dato_result["winner"] = dato["winner"]

    lista_cities = lt.newList("ARRAY_LIST")
    lista_countries = lt.newList("ARRAY_LIST")
    for dato in lt.iterator(lista_final_results):
        ciudad_dato = dato["city"]
        pais_dato = dato["country"]
        if not lt.isPresent(lista_cities, ciudad_dato):
            lt.addLast(lista_cities, ciudad_dato)
        if not lt.isPresent(lista_countries, pais_dato):
            lt.addLast(lista_countries, pais_dato)

    num_ciudades = lt.size(lista_cities)
    num_paises = lt.size(lista_countries)
    total_matches = lt.size(lista_final_results)

    return lista_final_results, num_ciudades, num_paises, total_matches, penaltis
```

Lo primero que hacemos es que obtenemos con dos variables la información total de los exceles “results” y “shootouts” desde el catálogo de data_structuts que contiene los ADT con la información que necesitamos. Luego creamos dos listas nuevas. Después iteramos con la lista de results y que si la fecha dato es menor o igual a la fecha final, también que si la fecha dato es mayor o igual a la fecha inicial y que si el torneo del dato sea igual al nombre del torneo, pues el dato ganador sea “Unknown” y que se añada el dato a la lista que creamos de results. Luego, iteramos con la lista de shootouts y que si se cumple entre esas fechas, pues se añade la información que cumple a la nueva lista de shootouts. A continuación creamos una variable llamada penaltis donde se iguala a 0 e iteramos con la lista nueva de shootouts. Dentro de la iteración debemos de iterar de nuevo pero ahora con la lista final de results para saber si dentro de las listas nuevas se iguala tanto “home_team” como “away_team” y pues si se cumple el contador de penales se sume 1 y que el dato_result pase ser el nombre del equipo ganador y

remplace al Unknown de arriba. Luego creamos dos listas nuevas para el total de ciudades y paises del torneo. Después iteramos con la lista final de resultados para saber si la ciudad y pais esta presente o no, y las añadimos a la lista y al final con el size podremos saber el número total de paises y ciudades. Por lo que se retorna al final del todo la lista final de resultus, el número de ciudades y paises, el total de partidos y el número de penales.

Entrada	- Catálogo que contiene los ADT con la información de los resultados - Nombre del torneo. - La fecha inicial del periodo a consultar (con formato "%Y-%m-%d"). - La fecha final del periodo a consultar (con formato "%Y-%m-%d").
Salidas	- Retorna el total de partidos encontrados al torneo - Retorna el total de países donde se disputaron partidos del torneo. - Retorna el total de ciudades donde se disputaron partidos del torneo. - Retorna el total de partidos definidos desde el punto penal - Retorna una lista con los resultados de la búsqueda.
Implementado (Sí/No)	Sí. Implementado por Julián David Contreras Pinilla

Análisis de complejidad

Los procesos de lt.size(), lt.newList() y lt.addLast() los consideraremos son procesos con complejidad promedio O(1), por lo que no hacen un efecto notorio a la complejidad total.

Pasos	**Complejidad**
Crea una lista vacía lt.newList()	O(1)
Agrega un elemento en la última posición lt.addLast()	O(1)
Buscar si el elemento existe (isPresent)	O(n)
Informa el número de elementos de la lista (size)	O(1)
TOTAL	***O(n2)***

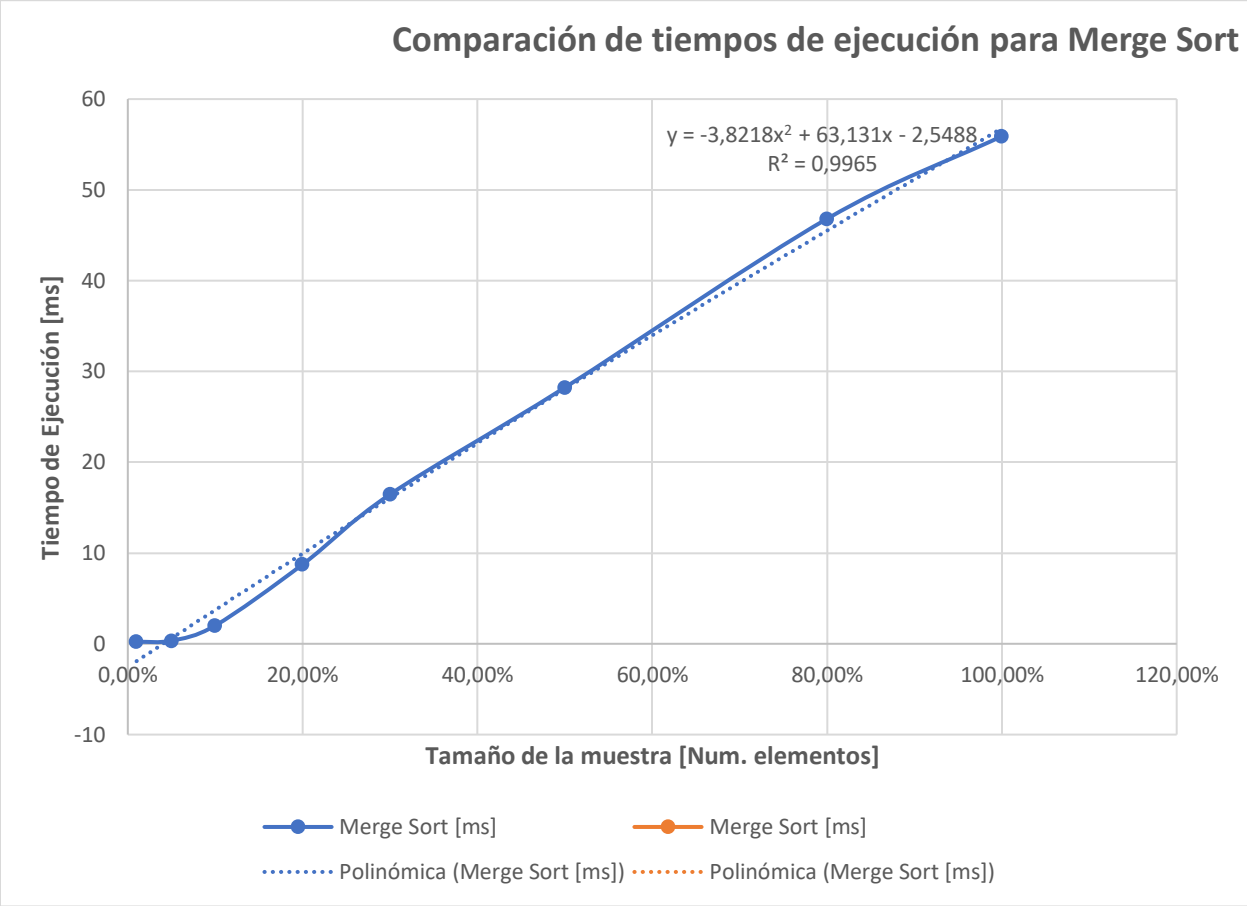
Para el caso del requerimiento 4 el peor caso con Array_LIST es n2 porque estamos recorriendo la lista final de shooutouts donde debemos recorrer dentro de ella misma la lista final de results para comprobar si tanto ambas lista tienen igual “home_team” y “away_team” y ahí poder sumarle +1 al número total de penales y cambiar el dato[“winner”] que era “Unknown” por el equipo que gano de dato_result.

Pruebas Realizadas

Entrada	**Tiempo (ms)**
small	0.217

5 pct	0.339
10 pct	2.003
20 pct	8.743
30 pct	16.456
50 pct	28.225
80 pct	46.813
large	55.914

Gráfica req 4



Análisis

La gráfica confirma lo que esta en el análisis, ya que tiene tendencia de orden temporal cuadrática y este comportamiento se puede evidenciar experimentalmente en la gráfica. Ya que, gracias a que los datos no se encuentran tan dispersos con respecto a la línea de tendencia, la curva coincide con el comportamiento cuadrático esperado.

Requerimiento 6

Descripción

```python
def req_6(data_structs, n_equipos, torneo, fecha_inicial, fecha_final):
    """Función que clasifica los N mejores equipos de un torneo en un periodo específico

    Args:
        data_structs (dict): Catálogo con los ADT con la información de los partidos
        n_equipos (int): Cantidad de N mejores equipos para consultar
        torneo (str): Nombre del torneo que se desea consultar
        fecha_inicial (datetime): Fecha mínima de búsqueda
        fecha_final (datetime): Fecha máxima de búsqueda

    Returns:
        sublist (ARRAY_LIST): Lista de los equipos que conforman el torneo ordenada
        n_teams (int): Cantidad de equipos involucrados en el torneo
        n_results (int): Cantidad de encuentros disputados en el periodo de tiempo
        n_countries (int): Total de países involucrados en el torneo
        n_cities (int): Total de ciudades involucrados en el torneo
        mostmatches (str): Nombre de la ciudad donde se disputó mayor cantidad de partidos
    """
    # TODO: Realizar el requerimiento 6

    #Data struct con la información separada por torneos
    tournaments = data_structs['tournaments']

    #Posiciones del torneo
    pos_tourn = binary_search_by_name(tournaments, torneo.lower())

    #Resultados en el torneo
    results = (lt.getElement(tournaments, pos_tourn))['results']

    #Posiciones para el rango de fechas
    pos_start = binary_search_start_date(results, fecha_inicial)
    pos_end = binary_search_end_date(results, fecha_final)

    #Casos de error
    if pos_start == -1 or pos_end == -1 or pos_start < pos_end:
        return None, 0, 0, 0, 0, ''

    #Listas auxiliares
    meetings = {'cities' : lt.newList('ARRAY_LIST', cmpfunction=compare_name), 'countries': lt.newList('ARRAY_LIST', cmpfunction=compare_name)}
    teams = lt.newList(datastructure="ARRAY_LIST", cmpfunction=compare_name)
```

```
#Listas auxiliares
meetings = {'cities' : lt.newList('ARRAY_LIST', cmpfunction=compare_name), 'countries': lt.newList('ARRAY_LIST', cmpfunction=compare_name)}
teams = lt.newList(datastructure="ARRAY_LIST", cmpfunction=compare_name)

n_results = 0
for i in range(pos_end, pos_start + 1):

    #Obtención de posición en teams y creación si no existe
    result = lt.getElement(results, i)
    poshome = add_team_req6(teams, result['home_team'])
    posaway = add_team_req6(teams, result['away_team'])

    #Cambiar la información con la nueva de la iteración
    changedhome = change_info_req6(teams, poshome, 'home', result)
    changedaway = change_info_req6(teams, posaway, 'away', result)

    #Añadir los cambios en teams
    lt.changeInfo(teams, poshome, changedhome)
    lt.changeInfo(teams, posaway, changedaway)

    #Contadores de paises y ciudades

    #Obtención de la posición y creación si no existe
    poscountry = lt.isPresent(meetings['countries'], result['country'])
    poscity = lt.isPresent(meetings['cities'], result['city'])

    if poscity > 0:
        city = lt.getElement(meetings['cities'], poscity)

    else:

        city = {'name': result['city'], 'meetings': 0}
        lt.addLast(meetings['cities'], city)
        poscity = lt.size(meetings['cities'])

    #Actualización de encuentros en esa ciudad
    city['meetings'] += 1

    #Añadir cambios a meetings en la sección de ciudades
    lt.changeInfo(meetings['cities'], poscity, city)

    #Lo mismo de arriba pero con paises
    if poscountry > 0:
        country = lt.getElement(meetings['countries'], poscountry)
    else:
        country = {'name': result['country'], 'meetings': 0}
        lt.addLast(meetings['countries'], country)
        poscountry = lt.size(meetings['countries'])
    country['meetings'] += 1
    lt.changeInfo(meetings['countries'], poscountry, country)

    n_results += 1
```

```
        n_results += 1

    #Ordenamiento de teams por orden de puntos
    sort(teams, 'merge', 'req6')
    #Ordenamiento de ciudades por cantidad de encuentros
    merg.sort(meetings['cities'], cmp_cities)

    #Tamaño de datos para devolver en el return
    n_teams = lt.size(teams)
    n_countries = lt.size(meetings['countries'])
    n_cities = lt.size(meetings['cities'])
    #Ciudad con más encuentros
    mostmatches = (lt.getElement(meetings['cities'], 1))['name']

    #Sublista con los N mejores equipos
    sublist = teams
    if n_equipos <= lt.size(teams):
        sublist = lt.subList(teams, 1, n_equipos)

    #Ciclo para obtener el mejor jugador en cada equipo
    for team in lt.iterator(sublist):

        if lt.size(team['scorers']) > 0:
            merg.sort(team['scorers'], cmp_top_scorer)
            team['top_scorer'] = lt.getElement(team['scorers'], 1)

    return sublist, n_teams, n_results, n_countries, n_cities, mostmatches
```

```
def add_team_req6(data_struct, name):
    """Función que encuentra la información de las estadísticas de un elemento, y en caso de no existir se crea un esqueleto en 0

    Args:
        data_struct (ARRAY_LIST): Lista con la información de las estadísticas
        name (str): Nombre del elemento a buscar/crear

    Returns:
        posteam (int): Posicióńn en la lista donde se encuentra el elemento
    """
    posteam = lt.isPresent(data_struct, name)
    if posteam > 0:
        info = lt.getElement(data_struct, posteam)
    else:
        info = {
            'name': name,
            'total_points': 0,
            'penalty_points': 0,
            'matches': 0,
            'own_goal_points':0,
            'wins': 0,
            'draws': 0,
            'losses': 0,
            'goals_for':0,
            'goals_against':0,
            'top_scorer': {'name': '', 'goals': 0, 'matches': 0, 'avg_time': 0, 'temp_time': 0},
            'scorers': lt.newList("ARRAY_LIST", cmpfunction=compare_name),
            'own_goals': 0,
            'goal_difference': 0
        }
        lt.addLast(data_struct, info)
        posteam = lt.size(data_struct)
    return posteam
```

```python
def change_info_req6(data_struct, pos, condition, data):
    """Función que cambia la información de las estadísticas con la nueva información de cada iteración

    Args:
        data_struct (ARRAY_LIST): Lista con las estadísticas de cada elemento
        pos (int): Posición donde se encuentra el elemento a cambiar
        condition (str): Condición del equipo a cambiar, puede ser 'home' o 'away'
        data (dict): Información del partido

    Returns:
        changed(dict): Diccionario con la información actualizada
    """
    againstcondition = None
    if condition == 'home':
        againstcondition = 'away'
    else:
        againstcondition = 'home'
    changed = lt.getElement(data_struct, pos)
    name = data[(condition) + '_team']
    #Total points + Wins + Losses + Draws
    if data[(condition + '_score')] > data[(againstcondition + '_score')]:
        changed['total_points'] += 3
        changed['wins'] += 1
    elif data[(condition + '_score')] < data[(againstcondition + '_score')]:
        changed['losses'] += 1
    else:
        changed['total_points'] += 1
        changed['draws'] += 1
    #Goals for + Goals Against
    changed['goals_for'] += data[(condition) + '_score']
    changed['goals_against'] += data[(againstcondition) + '_score']
    #Penalty points
    if data['winner'] == name:
        changed['penalty_points'] += 1
    #Matches
    changed['matches'] += 1
    #Goal difference
    changed['goal_difference'] = changed['goals_for'] - changed['goals_against']

    #Change Info Scorers
    if data['scorers'] != 'Unknown':
```

```python
    #Change Info Scorers
    if data['scorers'] != 'Unknown':

        for scorerinfo in lt.iterator(data['scorers']):

            #Own goal and penalty points
            if scorerinfo['penalty'] == 'True':
                changed['penalty_points'] += 1
            if scorerinfo['own_goal'] == 'True':
                changed['own_goal_points'] += 1

            #Encontrar / crear el goleador
            scorers = changed['scorers']
            posscorer = lt.isPresent(scorers, data['scorer'])

            if posscorer > 0:
                infoscorer = lt.getElement(scorers, posscorer)
            else:
                infoscorer = {'name': scorerinfo['name'], 'goals': 0, 'matches': 0, 'avg_time': 0, 'temp_time': 0}
                lt.addLast(scorers, infoscorer)
                posscorer = lt.size(scorers)


            scorer = lt.getElement(scorers, posscorer)
            changedscorer = scorer
            changedscorer['matches'] += 1
            changedscorer['goals'] += 1
            changedscorer['temp_time'] += scorerinfo['minute']
            changedscorer['avg_time'] = changedscorer['temp_time'] / changedscorer['matches']
            lt.changeInfo(scorers, posscorer, changedscorer)


    return changed
```

Este requerimiento clasifica los N mejores equipos en un torneo específico.

Primero se obtiene un arreglo del model el cuál es un arreglo donde cada posición contiene el nombre del torneo y los partidos que se jugaron en ese torneo. Cada partido cuenta con las columnas correspondientes a results, shootouts y además, tiene una llave de nombre 'scorers' cuyo valor es un arreglo con todos los goleadores de ese partido, en caso de no tener información el valor de la llave es 'Unknown'.

Primero se obtiene de manera binaria la posicón del torneo, y dentro de este las posiciones de los resultados que cumplen con el rango de fecha de inicio y fecha final. A continuación, se crean dos listas vacías, 'meetings' cumplirá con el rol de almacenar las ciudades y los países donde se jugaron los partidos de ese torneo, para al final retornar el total de ciudades y paises distintos y la ciudad donde más se disputaron partidos. Por otro lado, 'teams' será un arreglo el cuál se encargará de guardar cada equipo con sus estadísticas.

Después, se recorre el torneo entre los índices que cumplen con el rango de las fechas y se ejecuta la función add_team_req6(), esta recibe como parametro el data_struct y el nombre del equipo. La función se encarga de verificar si el equipo ya está agregado en la lista auxiliar 'teams', si ya existe retornará la posición en donde se encuentra, y si no, añadirá un esqueleto en 0 en la última posición del arreglo y retornará ese índice, el cuál va a ser igual al tamaño de la lista.

Ya encontrada la posición del equipo, se ejecuta la función 'change_info_req6()', la cual recibe como parámetro el arreglo con los equipos, la posición encontrada por la función anterior, la condición del equipo y la información del partido.

Dentro de esta función hay una variedad de condicionales que modifican los datos de las estadísticas según corresponda. Luego, se verifica que en la llave 'results' haya información de goleadores y si la hay, se va a iterar cada goleador guardado en la llave.

En el esqueleto del equipo, hay una llave donde se guardan todos los goleadores del equipo, por lo que la función busca si el jugador ya existe en esa llave, en caso de que si encuentra la posición y en caso de que no lo crea con su esqueleto de estadísticas. Luego, cambia estas estadísticas y al final de la función se retorna el diccionario con la información del equipo cambiada.

Luego, la función principal se encarga de actualizar esta información en el arreglo 'teams'. Después, se hacen comprobaciones similares a las anteriores para saber si una ciudad o país ya fue agregada a 'meetings'.

Todo este proceso se realizó tanto para el equipo local como para el visitante, con el fin de cambiar la información de 2 equipos en una sola iteración.

Ya acabado el ciclo, se ordenan los equipos por estadísticas y las ciudades por cantidad de encuentros. Luego, se hace una sublista con la cantidad n de equipos que el usuario solicitó.

Y, por último, se itera esa sublista y se accede a la llave 'scorers' para ordenar en cada equipo esa llave por estadísticas y se saca el primer elemento para obtener el mejor jugador de cada equipo.

Entrada	Catálogo con los ADT a usar Número de equipos a filtrar Nombre del torneo Fecha inicial de búsqueda Fecha final de búsqueda
Salidas	Sublista con la información filtrada y ordenada por estadísticas Número total de equipos en el torneo Número total de países involucrados en el torneo Número total de ciudades involucradas en el torneo La ciudad en la que más partidos se jugaron
Implementado (Sí/No)	Sí. Implementado por Daniel Camilo Quimbay Velásquez

Análisis de complejidad

Las búsquedas binarias dependen de la cantidad de n de torneos y la cantidad m de partidos en el torneo.

La iteración de los partidos en el torneo va a es O(m), mientras que comprobar la existencia de equipos (o), ciudades (q), paises y jugadores (p) por equipo en el peor caso es la cantidad total de cada uno.

Dentro de las iteraciones de partidos la complejidad es ~ m(op + q) + n_equipos

En el peor caso op = n = q = n_equipos, por lo que la complejidad sería ~2n^2 + n, o en Big O sería O(n^2).

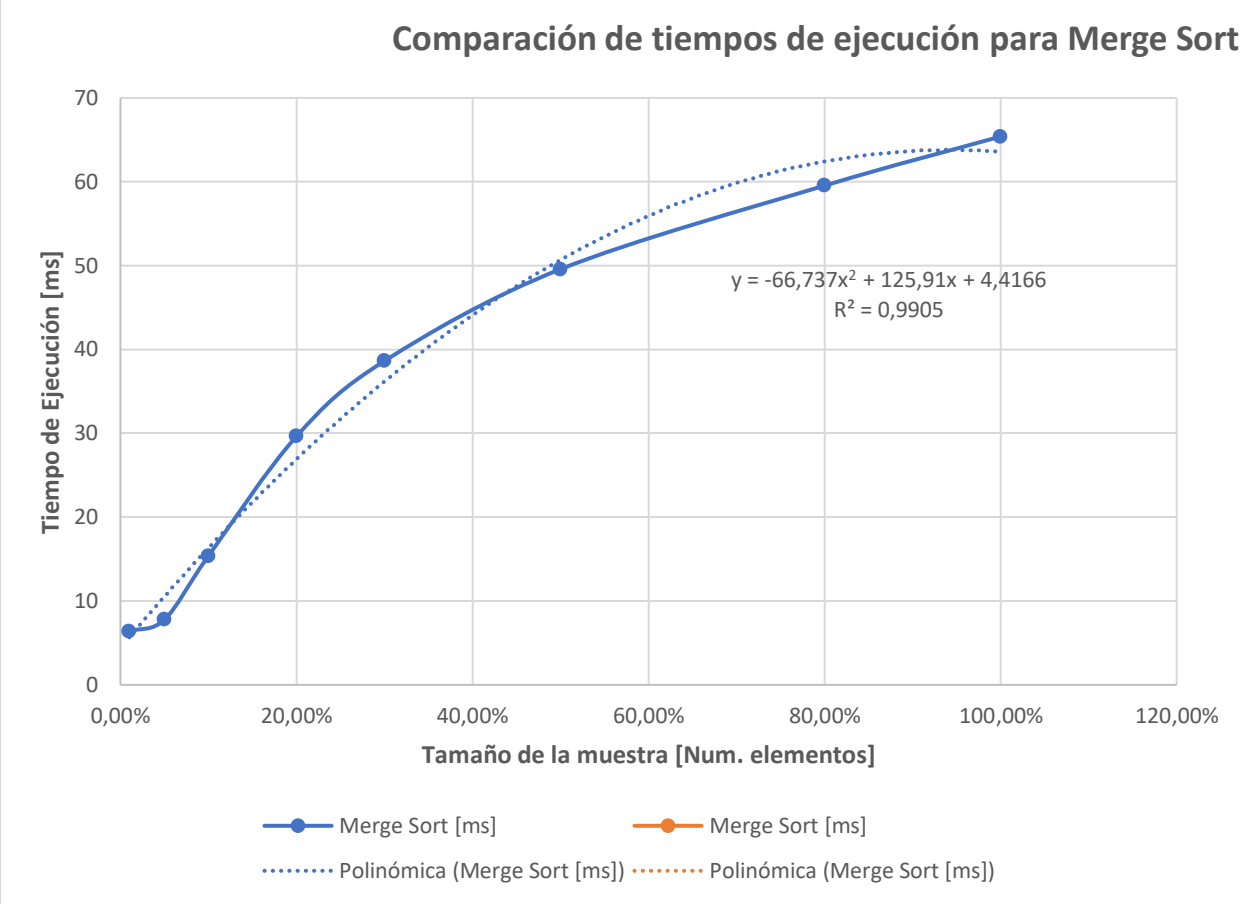
Pasos	**Complejidad**
Busqueda binaria del torneo	O(log(n))
Búsqueda binaria de la fecha de inicio	O(log(m))
Búsqueda binaria de la fecha final	O(log(m))
	O(1)
Verificación de existencia del equipo	O(o)
Cambio de la información del equipo	O(1)
Verificación de existencia del jugador	O(p)
Cambio de la información del jugador	O(1)
Ordenar equipos por estadísticas	O(log(m))
Sublista de equipos	O(n_equipos)
Iteración de los equipos y por dentro de sus jugadores	O(n_equipos * p)
TOTAL	***O(n^2)***

Pruebas Realizadas

Entrada	**Tiempo (ms)**
small	6.401

5 pct	7.792
10 pct	15.379
20 pct	29.642
30 pct	38.633
50 pct	49.578
80 pct	59.542
large	65.398

Gráfica req 6



Análisis

La gráfica es bastante diferente a lo esperado. Para explicar esto debemos recordar que la cota de n^2 es en el peor caso. Sin embargo, en el caso promedio la cantidad de equipos no supera a los partidos jugados, por lo que m > o

Conforme pasa el tiempo es más probable que los equipos se repitan, por lo que entre más datos m se hace mucho más mayor que o, por lo que la complejidad promedio sería O(m)

Este decrecimiento de complejidad conforme aumenta la cantidad de datos es la que podemos observar en la gráfica.

Requerimiento 07

Descripción

```python
def req_7(data_structs, fecha_inicial, fecha_final, top_jugadores):
    """Función que clasifica los N mejores anotadores en partidos oficiales en un periodo específico

    Args:
        data_structs (dict): Catálogo con los ADT que contienen la información de los partidos
        fecha_inicial (datetime): Fecha mínima de búsqueda
        fecha_final (datetime): Fecha máxima de búsqueda
        top_jugadores (int): Cantidad N de jugadores para filtrar

    Returns:
        sublist(ARRAY_LIST): Lista filtrada con los N mejores jugadores en el periodo especificado
        num_jugadores (int): Cantidad de jugadores encontrados
        num_partidos (int): Total de partidos en que participaron los anotadores
        num_goles(int): Total de goles durante el periodo de tiempo
        num_penales (int): Total de goles por penal durante el periodo de tiempo
        num_autogoles (int): Total de autogoles por los jugadores en ese periodo
        num_tourns (int): Total de torneos donde participaron los anotadores
    """

    #Data_struct con solo torneos oficiales
    results = data_structs['official_results']

    #Encontrar rangos de fechas
    posstart = binary_search_start_date(results, fecha_inicial)
    posend = binary_search_end_date(results, fecha_final)

    if posstart == -1 or posend == -1 or posstart < posend:
        return None, 0, 0, 0, 0, 0, 0

    #Creación de lista donde se guardará la información de cada scorer
    scorers = lt.newList(datastructure='ARRAY_LIST', cmpfunction=compare_name)
    tournaments = lt.newList(datastructure='ARRAY_LIST', cmpfunction=compare_name)

    num_partidos = 0
    num_goles = 0
```

```
    num_penales = 0
    num_autogoles = 0
    #Iterar solo en el rango de fechas
    for i in range(posend, posstart + 1):
        result = lt.getElement(results, i)
        if result['scorers'] != 'Unknown':
            for scorer in lt.iterator(result['scorers']):

                posscorer = add_scorer_req7(scorers, scorer['name'])
                #Cambio de información en la iteración
                changed = change_info_scorer(scorers, posscorer, result)
                lt.changeInfo(scorers, posscorer, changed)

                if scorer['penalty'] == 'True':
                    num_penales += 1
                if scorer['own_goal'] == 'True':
                    num_autogoles += 1

            #Cambiar datos
            num_partidos += 1
            num_goles += result['home_score'] + result['away_score']
            #Añadir si es necesario el torneo para obtener el total de torneos
            postournament = lt.isPresent(tournaments, result['tournament'])
            if postournament != 0:
                lt.addLast(tournaments, {'name': result['tournament']})

    merg.sort(scorers, cmp_scorer_points)
    sublist = lt.subList(scorers, 1, top_jugadores)

    num_jugadores = lt.size(scorers)
    num_tourns = lt.size(tournaments)

    return sublist, num_jugadores, num_partidos, num_goles, num_penales, num_autogoles, num_tourns
```

```
def add_scorer_req7(data_struct, name):
    """Función que encuentra la información de las estadísticas de un jugador, y en caso de no existir se crea un esqueleto en 0

    Args:
        data_struct (ARRAY_LIST): Lista con la información de las estadísticas
        name (str): Nombre del jugador a buscar/crear

    Returns:
        posscorer (int): Posición en la lista donde se encuentra el jugador
    """
    posscorer = lt.isPresent(data_struct, name)
    if posscorer <= 0:
        #Esqueleto scorer desde 0
        scorer = {
            'name': name,
            'total_points': 0,
            'total_goals': 0,
            'penalty_goals':0,
            'own_goals': 0,
            'avg_time': 0,
            'total_time': 0,
            'total_tournaments': 0,
            'tournaments': lt.newList(datastructure='ARRAY_LIST', cmpfunction=compare_name),
            'scored_in_wins': 0,
            'scored_in_losses': 0,
            'scored_in_draws': 0,
            'last_goal': None,
        }
        lt.addLast(data_struct, scorer)
        posscorer = lt.size(data_struct)
    return posscorer
```

```python
def change_info_scorer(data_struct, pos, data):
    """Función que cambia la información del jugador en cada iteración con la información del partido

    Args:
        data_struct (ARRAY_LIST): Lista con la información de los jugadores
        pos (int): Posición del jugador en la lista
        data (dict): Información del partido

    Returns:
        changed: Diccionario con la información actualizada del jugador
    """
    changed = lt.getElement(data_struct, pos)

    #penalty_points
    if data['penalty'] == 'True':
        changed['penalty_goals'] += 1

    #own_goals
    if data['own_goal'] == 'True':
        changed['own_goals'] += 1

    #Comprobar si el goleador es del home_team o del away_team
    condition = None
    againstcondition = None
    if data['team'] == data['home_team']:
        condition = 'home'
        againstcondition = 'away'
    else:
        condition = 'away'
        againstcondition = 'home'

    #Scores in Wins - Draws - Losses
    selfscore = data[(condition) + '_score']
    againstscore = data[(againstcondition + '_score')]
    if selfscore > againstscore:
        changed['scored_in_wins'] += 1
    elif selfscore < againstscore:
        changed['scored_in_losses'] += 1
    else:
        changed['scored_in_draws'] += 1

    #Total goals
    changed['total_goals'] = changed['scored_in_wins'] + changed['scored_in_losses'] + changed['scored_in_draws']

    #Total points
    changed['total_points'] = changed['total_goals'] + changed['penalty_goals'] - changed['own_goals']

    #Avg time
    changed['total_time'] += data['minute']
    changed['avg_time'] = changed['total_time'] / changed['total_goals']

    #Total tournaments
    postournament = lt.isPresent(changed['tournaments'], data['tournament'])
    if postournament != 0:
        lt.addLast(changed['tournaments'], {'name': data['tournament']})
    changed['total_tournaments'] = lt.size(changed['tournaments'])

    #Last goal
    if changed['last_goal'] == None:
        changed['last_goal'] = data

    return changed
```

En el requerimiento 7 pues la función comienza extrayendo información de partidos desde data_structs, que se supone que contiene datos de partidos de fútbol, específicamente los resultados de partidos oficiales. Luego, utiliza las funciones binary_search_start_date y binary_search_end_date (que no se muestran aquí) para encontrar los índices de inicio y fin de un rango de fechas dentro de los resultados

de partidos. Si no se encuentra un rango de fechas válido, la función retorna valores nulos y ceros. Después, la función crea una lista llamada scorers para almacenar información sobre los anotadores de goles, y otra llamada tournaments para almacenar información sobre los torneos en los que participaron los anotadores. Luego seitera a través de los resultados de los partidos dentro del rango de fechas y procesa a los anotadores de goles. Después, utiliza una función llamada add_scorer_req7 para agregar información sobre cada anotador a la lista scorers. Si un anotador aún no existe en la lista, se crea un nuevo registro. Luego, actualiza la información del anotador en cada iteración utilizando la función change_info_scorer, que calcula estadísticas como el número de goles, penales, autogoles, promedio de tiempo de anotación, etc. Después de procesar todos los partidos, la función clasifica a los anotadores y crea una sublista de los mejores anotadores según el valor de top_jugadores. Por último, la función también recopila información sobre la cantidad total de jugadores, partidos, goles, penales, autogoles y torneos en los que participaron los anotadores.

Entrada	- Estructuras de datos del modelo - El número (N) de partidos de consultar - La fecha inicial del periodo a consultar (con formato "%Y-%m-%d"). - La fecha final del periodo a consultar (con formato "%Y-%m-%d").
Salidas	- Retorna el total de anotadores que se encontraron en la consulta. - Retorna el total de partidos o encuentros en que participaron los anotadores. - Retorna el total de torneos donde participaron los anotadores en ese periodo. - Retorna el total de anotaciones o goles obtenidos durante los partidos de ese periodo. - Retorna el total de goles por penal obtenidos en ese periodo. - Retorna el total de autogoles en que incurrieron los anotadores en ese periodo. - Retorna la lista filtrada con los N mejores jugadores en el periodo especificado.
Implementado (Sí/No)	Sí. Implementado por Daniel Camilo Quimbay Velásquez y Julián David Contreras Pinilla

Análisis de complejidad

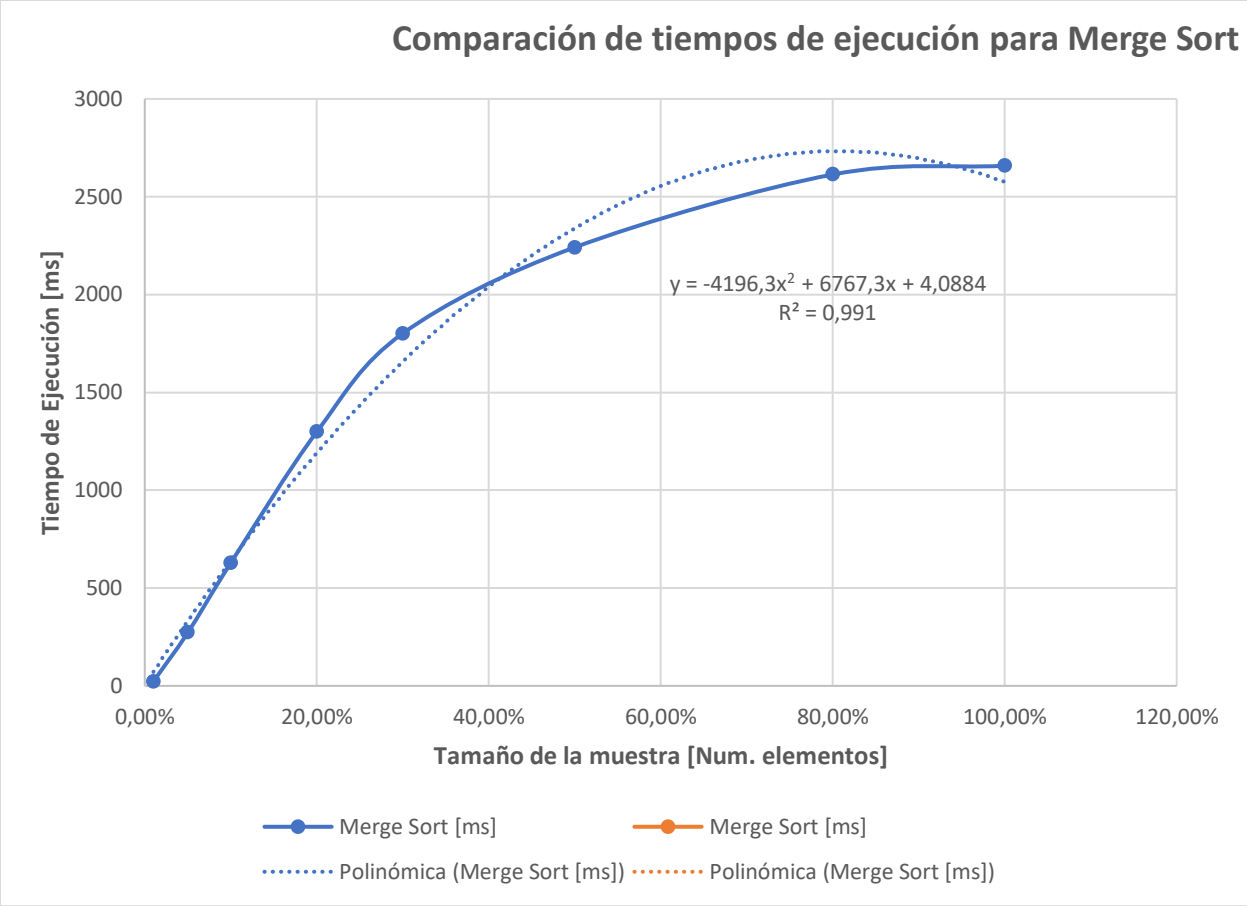
El peor caso para el análisis de complejidad es que fuera n2, ya que en el requerimiento 7 se recorre un for dentro de un for, lo cual implica que el peor caso sea n2 y es que en este caso recorre una lista dentro de una lista.

Pasos	Complejidad
Busqueda binaria del torneo	O(log(n))
Búsqueda binaria de la fecha de inicio	O(log(m))
Búsqueda binaria de la fecha final	O(log(m))
Crea una lista vacía lt.newList()	O(1)
Verificación de existencia del equipo	O(o)
Cambia la información contenida en el nodo de la lista lt.ChangeInfo()	O(1)
Buscar si el elemento existe (isPresent)	O(n)
Agrega un elemento en la última posición lt.addLast()	O(1)
Ordena la listamerg.sort	O(n*log(n))
Una lista pequeña lt.subList()	O(1)
Informa el número de elementos de la lista (size)	O(n)
TOTAL	***O(n^2)***

Pruebas Realizadas

Entrada	Tiempo (ms)
small	22.468
5 pct	271.872
10 pct	627.912
20 pct	1297.925
30 pct	1801.229
50 pct	2241.203
80 pct	2613.633
large	2658.368

Gráfica req 7



Análisis

La gráfica es bastante diferente a lo esperado. Para explicar esto debemos recordar que la cota de n^2 es en el peor caso. Sin embargo, en el caso promedio sería n*log(n) Este decrecimiento de complejidad conforme aumenta la cantidad de datos es la que podemos observar en la gráfica, por lo que podría decir que se comporta aproximadamente como lo esperado.

Requerimiento 08 o Bono

```python
def req_8(data_structs, equipo1, equipo2, inicial, final):
    """
    Función que soluciona el requerimiento 8
    """
    # TODO: Realizar el requerimiento 8
    official_teams = data_structs['official_teams']

    sublist1, home1, away1 = req_3({'teams': official_teams}, equipo1, inicial, final)
    sublist2, home2, away2 = req_3({'teams': official_teams}, equipo2, inicial, final)

    if sublist1 != None:
        n1 = lt.getElement(sublist1, 1)
        newest1 = lt.newList('ARRAY_LIST')
        lt.addLast(newest1, n1)
    else:
        n1 = ''
        newest1 = None

    if sublist2 != None:
        n2 = lt.getElement(sublist2, 1)
        newest2 = lt.newList('ARRAY_LIST')
        lt.addLast(newest2, n2)
    else:
        n2 = ''
        newest2 = None

    infocommon = {'matches': 0, 'wins1': 0, 'wins2': 0, 'losses1': 0, 'losses2': 0, 'draws': 0}

    if sublist1 == None and sublist2 == None:
        return None, None, None, None, None, {}, {}, {}

    common_history = lt.newList(datastructure='ARRAY_LIST', cmpfunction=compare_id)

    years = {'team1': lt.newList('ARRAY_LIST', cmpfunction=compare_year), 'team2': lt.newList('ARRAY_LIST', cmpfunction=compare_year)}

    for result in lt.iterator(sublist1):
        year1 = result['date'].year
        pos_year1 = add_team_req6(years['team1'], year1)
        if result['home_team'].lower() == equipo1:
            condition = 'home'
        else:
            condition = 'away'
        changed = change_info_req6(years['team1'], pos_year1, condition, result)
        lt.changeInfo(years['team1'], pos_year1, changed)

    for result in lt.iterator(sublist2):
        year2 = result['date'].year
```

```
    for result in lt.iterator(sublist2):
        year2 = result['date'].year
        pos_year2 = add_team_req6(years['team2'], year2)
        if result['home_team'].lower() == equipo2:
            condition = 'home'
            againstcondition = 'away'
        else:
            condition = 'away'
            againstcondition = 'home'

        if result[(condition) + '_team'].lower() == equipo2 and result[(againstcondition) + '_team'].lower() == equipo1:
            lt.addLast(common_history, result)
            infocommon['matches'] += 1
            if result['home_team'].lower() == equipo1:
                if result['home_score'] > result['away_score']:
                    infocommon['wins1'] += 1
                    infocommon['losses2'] += 1
                elif result['home_score'] < result['away_score']:
                    infocommon['wins2'] += 1
                    infocommon['losses1'] += 1
                else:
                    infocommon['draws'] += 1
            else:
                if result['home_score'] > result['away_score']:
                    infocommon['wins2'] += 1
                    infocommon['losses1'] += 1
                elif result['home_score'] < result['away_score']:
                    infocommon['wins1'] += 1
                    infocommon['losses2'] += 1
                else:
                    infocommon['draws'] += 1


        changed = change_info_req6(years['team2'], pos_year2, condition, result)
        lt.changeInfo(years['team2'], pos_year2, changed)



    merg.sort(years['team1'], cmp_year)
    merg.sort(years['team2'], cmp_year)
```

```
merg.sort(years['team1'], cmp_year)
merg.sort(years['team2'], cmp_year)

for year in lt.iterator(years['team1']):

    if lt.size(year['scorers']) > 0:
        merg.sort(year['scorers'], cmp_top_scorer)
        year['top_scorer'] = lt.getElement(year['scorers'], 1)

for year in lt.iterator(years['team2']):

    if lt.size(year['scorers']) > 0:
        merg.sort(year['scorers'], cmp_top_scorer)
        year['top_scorer'] = lt.getElement(year['scorers'], 1)

newestcommon = lt.newList('ARRAY_LIST')
nc = lt.getElement(common_history, 1)
lt.addLast(newestcommon, nc)

infot1 = {'years': lt.size(years['team1']), 'matches': lt.size(sublist1), 'home': home1, 'away': away1, 'oldest': (lt.lastElement(sublist1))['date']}
infot2 = {'years': lt.size(years['team2']), 'matches': lt.size(sublist2), 'home': home2, 'away': away2, 'oldest': (lt.lastElement(sublist2))['date']}


return years, common_history, newest1, newest2, newestcommon, infot1, infot2, infocommon
```

Descripción

Este requerimiento compara el rendimiento de dos equipos en un periodo específico de tiempo.

Para resolverlo usamos una arreglo en el que están todos los partidos oficiales que se jugaron. Para obtener una sublista que esté en el rango de fechas llamamos a la función del requerimiento 3 para que nos devuelva estos datos.

Luego, se crean varias variables que van a guardar datos pequeños como el último partido jugado de cada equipo, numero de partidos, victorias, derrotas, entre otros.

Luego, se itera cada equipo, en ambas se utilizan funciones auxiliares del requerimiento 6 que añaden un nuevo equipo, donde ahora nuestro nombre de equipo en realidad va a ser el año, y también se usa la de cambiar información ya que usa las mismas estadísticas que el requerimiento 6.

En la iteración del segundo equipo se hace el mismo procedimiento que en el primero. Sin embargo, en este caso también se van a buscar los partidos que tenga en común con el primer equipo, es decir, los partidos en que se enfrentaron el equipo 1 y el equipo 2. En caso de que se encuentre, se modifican los diccionarios que guardan la información de victorias, derrotas, etc.

Al terminar estas dos iteraciones se va a ordenar por fecha las estadísticas de cada año, tanto del equipo 1 como la del equipo 2. Y para finalizar, se repite el procedimiento de encontrar al mejor jugador en cada año de manera similar a como se hizo en el requerimiento 6.

Entrada	Catálogo con los ADT a utilizar Nombre del equipo 1 Nombre del equipo 2 Fecha de inicio de la búsqueda Fecha final de la búsqueda
Salidas	Sublista con la información por años de ambos equipos Sublista con los partidos entre ambos equipos Último partido jugado por el equipo 1 Último partido jugado por el equipo 2 Información de cantidades de datos varias del equipo 1 Información de cantidades de datos varias del equipo 2 Información de cantidades de datos varias de los enfrentamientos entre equipo 1 y equipo 2
Implementado (Sí/No)	Sí. Implementado por Daniel Camilo Quimbay Velásquez

Análisis de complejidad

De manera similar al requerimiento 6, la complejidad en el peor caso es O(n^2)

En notación tilda la complejidad sería 2n + 2n*(m + log(m)) + m

Suponiendo que el pero caso es cuando n = m, tenemos que la complejidad es O(n^2)

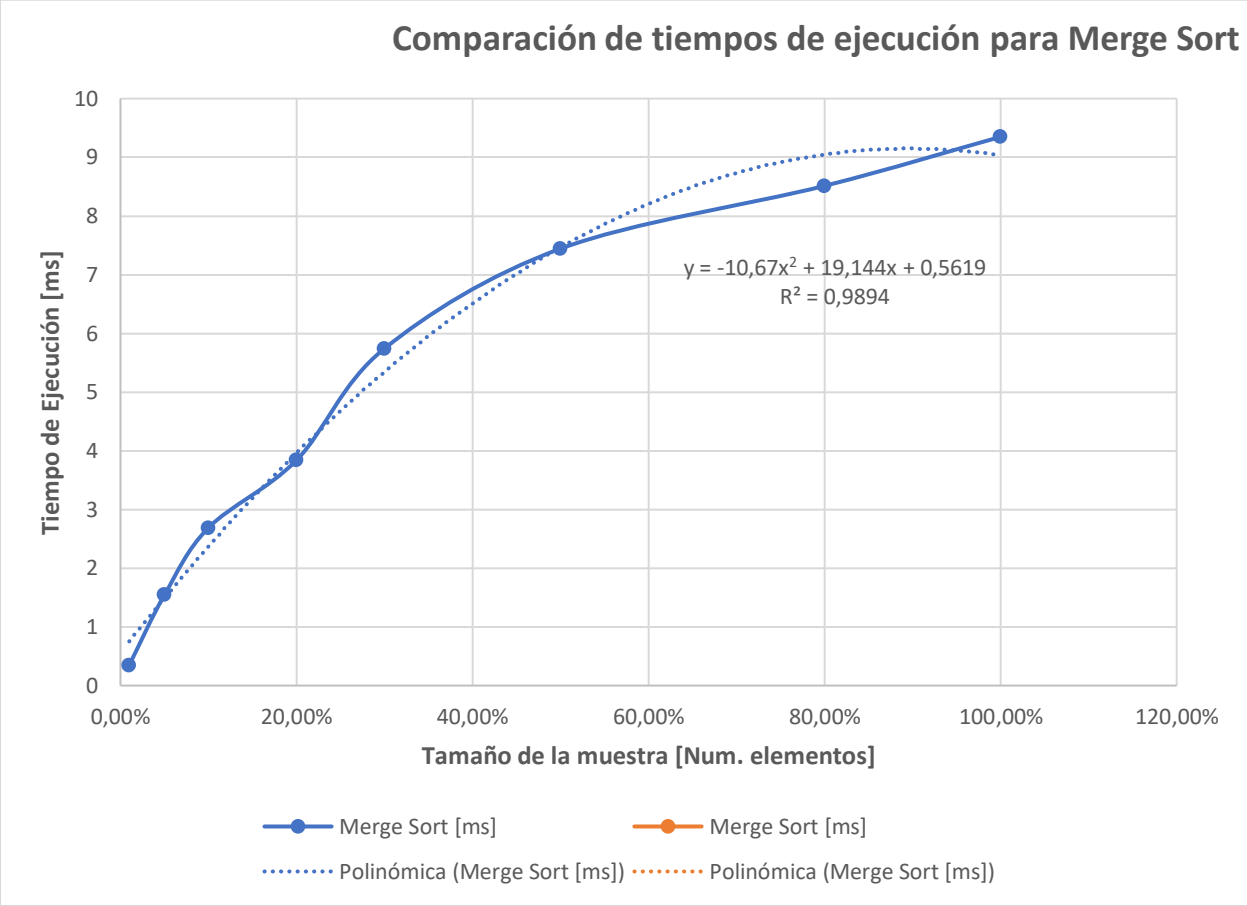
Pasos	**Complejidad**
Ejecutar requerimiento 3	O(n)
Iterar cada partido en la sublista	O(n)
Verificar existencia del año	O(m)
Cambiar la información de las estadísticas	O(1)
Ordenar por estadísticas	O(log(m))

Iterar cada año para obtener top_scorer	O(m)
TOTAL	***O(n^2)***

Pruebas Realizadas

Entrada	**Tiempo (ms)**
small	0.350
5 pct	1.544
10 pct	2.689
20 pct	3.840
30 pct	5.743
50 pct	7.444
80 pct	8.514
large	9.349

Gráfica req 8 o Bono



Análisis

Al igual que en el requerimiento 6, la gráfica es bastante alejado de lo esperado. Sin embargo, debemos tener en cuenta que en partidos de torneos es dificil que solo haya un partido por año, por lo que conforme va creciendo la muestra de datos, n se va haciendo mayor que m.

En este requerimiento vemos un aumento en el tiempo de ejecución, y tiene sentido ya que al llamar 2 veces la función 3 y ejecutar 2 iteraciones, una para cada equipo, evidentemente tendrá un impacto mayor en el tiempo.