ANÁLISIS DEL RETO

Estudiante 1: Gabriela Zambrano, 202211712, g.zambranoz@uniandes.edu.co

Estudiante 2: María José Mantilla, 202121670, m.mantillav@uniandes.edu.co

Estudiante 3: Dayanna Rueda, 202015345, l.ruedap@uniandes.edu.co

Carga de Datos

Plantilla para el documentar y analizar cada uno de los requerimientos.

```python
#Funciones de cargar Datos
def carga_resultados(data_structs,archivo):
    """
    Función para cargar los datos del archivo de "results"
    """

    nombre_archivo=cf.data_dir+"football/results-utf8-"+archivo
    archivo_leido= csv.DictReader(open(nombre_archivo, encoding="utf-8"))
    for cada_linea in archivo_leido:
        add_data_lists(data_structs,cada_linea,"results")
        add_data_map(data_structs,cada_linea,"tournament","tournament")
        add_data_mapTeam(data_structs,cada_linea)
        add_data_mapAnio_torneo(data_structs["anios_torneo"],cada_linea)
        add_data_partido(data_structs,"home_away_team_results",cada_linea)
    return data_size(data_structs, "results")

def carga_goleadores(data_structs,archivo):
    """
    Función para cargar los datos del archivo de "goalscorers"
    """
    nombre_archivo=cf.data_dir+"football/goalscorers-utf8-"+archivo
    archivo_leido= csv.DictReader(open(nombre_archivo, encoding="utf-8"))
    for cada_linea in archivo_leido:
        add_data_lists(data_structs,cada_linea,"goalscorers")
        add_data_map(data_structs,cada_linea,"scorers","scorer")
        add_data_partido(data_structs,"home_away_team_goalscorers",cada_linea
    return data_size(data_structs, "goalscorers")

def carga_penales(data_structs,archivo):
    """
    Función para cargar los datos del archivo de "shootouts"
    """
    nombre_archivo=cf.data_dir+"football/shootouts-utf8-"+archivo
    archivo_leido= csv.DictReader(open(nombre_archivo, encoding="utf-8"))
    for cada_linea in archivo_leido:
        add_data_lists(data_structs,cada_linea,"shootouts")
        add_data_partido(data_structs,"home_away_team_shootouts",cada_linea)
    return data_size(data_structs, "shootouts")
```

Descripción

Con el fin de realizar la carga de datos, de los tres archivos disponibles, se creó una función para cada uno de estos. En cada una de estas funciones en primer lugar se accede al nombre del archivo, a partir

de su tamaño, luego, se realiza la lectura del documento con csv.DictReader (esta herramienta va leyendo línea a línea el documento). Ahora bien, con cada archivo se realizan distintas acciones con cada línea del documento. En cuanto al archivo de “results”, en primer lugar, se agrega el partido a una lista, luego se agregan a un mapa teniendo como llave el torneo del partido, también se agrega a un mapa por el equipo, otro por el año y torneo y por último a un mapa el cual permite encontrar el partido más rápido (la llave siendo fecha- equipo local - equipo visitante). Por otra parte, con el archivo de “goalscorers, se añade cada dato en una lista, en un mapa con la llave del nombre del jugador que marcó el gol y por último a un mapa el cual permite encontrar el partido más rápido (la llave siendo fecha- equipo local - equipo visitante). Finalmente, con el archivo de “shootouts” cada partido se añade en una lista, y por último a un mapa el cual permite encontrar el partido más rápido (la llave siendo fecha- equipo local - equipo visitante).

IMPORTANTE: Los datos fueron almacenados en listas tipo ARRAY_LIST, ya que el tiempo de ordenamiento con este tipo de listas es más corto. Ademas los mapas de tournament y años-torneo fueron guardados en mapas tipo CHAINING con un factor de carga de 4 y el resto de mapas son PROBING con un factor de carga de 0.7.

Entrada	Entran como parámetros: • control (control de datos para el modelo, que posteriormente será el parámetro control[“model”] para las funciones carga_resultados, carga_goleadores y carga_penales). • archivo (archivo que contiene los datos para cargar en las estructuras).
Salidas	Retorna: - 3 listas con los datos de cada uno de los archivos - Mapa “tournament”, datos ordenados por torneos del archivo “results” - Mapa “team”, datos ordenados por equipos del archivo “results” - Mapa “scorers”, datos ordenados por jugador del archivo “goalscorers” - Mapa “anios_torneo”, datos ordenados por año y luego por torneos del archivo “results” - 3 mapas de “home_away_team” de cada uno de los archivos para encontrar los partidos de forma más rápida
Implementado (Sí/No)	Si, implementado por Gabriela Zambrano

Análisis de complejidad

La carga de todos los archivos es la misma, lo único que cambia los tamaños de los documentos:

Documentos resultados= n

Documentos goalscorers= g

Documento shootouts =s

Tal que n>g>s

Pasos	**Complejidad**
Carga de resultados(función)	O(n)-> es la complejidad de la función "carga_resultados"
Formar nombre de archivo	O(1)
Leer archivo	O(n)
Iteración en archivo	O(n)
Añadir datos a lista	O(1)
Añadir datos a mapas	O(1)
Size de la lista	O(1)
Carga de goalscorers(función)	O(g) -> es la complejidad de la función "carga_goleadores"
Formar nombre de archivo	O(1)
Leer archivo	O(g)
Iteración en archivo	O(g)
Añadir datos a lista	O(1)
Añadir datos a mapas	O(1)
Size de la lista	O(1)
Carga de shootouts(función)	O(s) -> es la complejidad de la función "carga_penales"
Formar nombre de archivo	O(1)
Leer archivo	O(s)
Iteración en archivo	O(s)
Añadir datos a lista	O(1)
Añadir datos a mapa	O(1)
Size de la lista	O(1)
Ordenamiento listas	O(nlog(n))
TOTAL	***O (nlog(n))***

Nota: Al trabajar con mapas (funciones como contains o get) tomamos la complejidad de los casos promedio, teniendo en cuenta que se eligió un factor de carga el cual permite reducir la cantidad de colisiones.

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).

Tiempo de ejecución con todos los datos (large):	2572,55 ms
Memoria consumida todos los datos	72035,3 kB
Condición, herramientas y recursos utilizados:	• csv.DictReader • Un ciclo for en cada función de carga de cada uno de los tres archivos
Computador donde se ejecuta:	Intel(R) Core(TM) i5-10300H CPU @ 2.50GHz 2.50 GHz (8 GB Windows 11 Home)

Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

Factor de carga 0.7

Tiempo

Entrada	Tiempo de Ejecución Real [ms]
Small	58,03
5 pct	299,4
10 pct	377,75
20 pct	652,56
30 pct	799,42
50 pct	1379,16
80 pct	1916,36
Large	2572,55

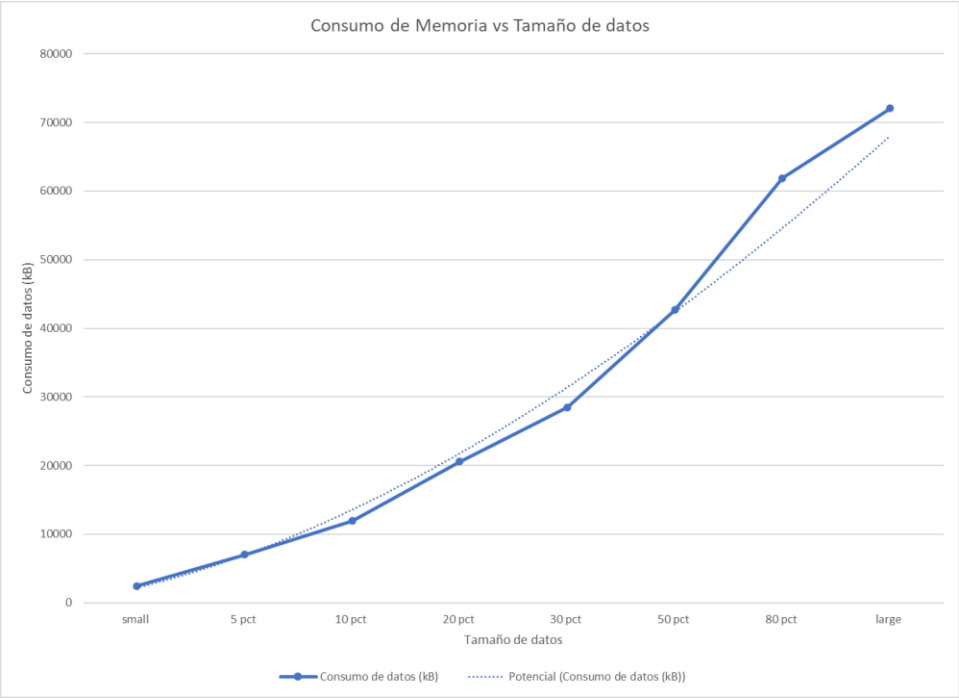
Memoria

Tamaño de datos	Consumo de Datos [kB]
Small	2438,93
5 pct	7028,04
10 pct	11911,08
20 pct	20590,66
30 pct	28504,97
50 pct	42678,47
80 pct	61900,6
Large	72035,3

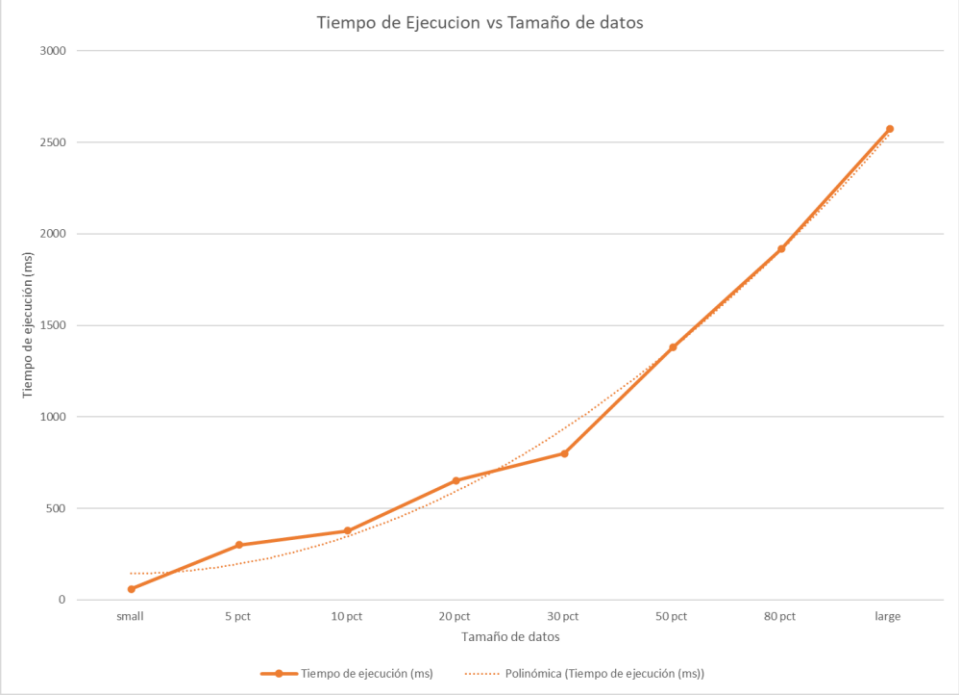
Graficas

Las gráficas con la representación de las pruebas realizadas.

Consumo de Datos



Tiempo de Ejecución



Análisis

Análisis de resultados de la implementación, tener cuenta las pruebas realizadas y el analisis de complejidad.

A partir de las gráficas dadas se puede observar que el tiempo de ejecucion aumenta de forma polinómica a medida que la cantidad de datos se hace mayor, esto va de acuerdo con la complejidad temporal que se calculó anteriormente. Esta fue de nlog(n), lo cual estaría de acorde a la gráfica que se puede observar. Por otro lado, con respecto al consumo de datos, este aumento de forma potencial a medida que aumentaban los datos, esto se encuentra relacionado a la creación de mapas, sobre todo los mapas de “CHAINING” los cuales consumen más espacio a medida que se aumentan los tamaños de las LINKED_LIST.

Requerimiento 1

Plantilla para el documentar y analizar cada uno de los requerimientos.

```
def req_1(data_structs,condicion, n, team):
    """
    Función que soluciona el requerimiento 1
    """

    equipos = data_structs["model"]["team"]
    lista_seleccionados = lt.newList("ARRAY_LIST")
    pareja = mp.get(equipos,team)
    lista = me.getValue(pareja)
    total_equipo = lt.size(lista)
    if condicion == "home" or condicion == "away":
        for equipo in lt.iterator(lista):
            if condicion == "home":
                if  team == equipo["home_team"]:
                    lt.addLast(lista_seleccionados,equipo)
            elif condicion == "away":
                if  team == equipo["away_team"]:
                    lt.addLast(lista_seleccionados,equipo)
    else:
        lista_seleccionados = lista
    sub_total = lt.size(lista_seleccionados)
    quk.sort(lista_seleccionados,cmp_partidos_results)
    total_equipos = mp.size(equipos)


    if sub_total > n:
        lista_final = lt.subList(lista_seleccionados,1,n)
        tamaño = lt.size(lista_final)
    else:
        lista_final = lista_seleccionados
        tamaño = lt.size(lista_seleccionados)
    return lista_final, sub_total, total_equipos,total_equipo,tamaño
```

Descripción

En este requerimiento para encontrar todos los partidos que jugó un equipo se utilizó un mapa cuyas llaves eran los nombres de los equipos y los valores eran una lista de diccionarios.Para empezar, se seleccionó la lista que correspondía al equipo indicado, después si la condición era indiferente se devolvía toda la lista de la llave, si era home o away se iteraba sobre esta para encontrar los partidos que coincidieran con la condición.

Entrada	Entran por parámetro: • data_structs (Datos almacenados en los archivos ingresados) • Nombre del equipo • Condicion del equipo (home-away) • N número de partidos
Salidas	Retorna: • El total de equipos con información disponible. • El total de partidos en que participó el equipo (sin importar su condición). • El subtotal de los partidos en que participó el equipo según su condición (local, visitante, indiferente).
Implementado (Sí/No)	Sí, implementado por María José Mantilla

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Tamaño de results = n

Tamaño de goalscorers = g

Tamaño de shootouts = s

Tal que n>g>s

Pasos	**Complejidad**
Iniciar contadores y crear listas	O(1)
Extraer la tupla llave-valor para el torneo seleccionado	O(1)
Extraer la lista (valor)	O(1)
Condicional para filtrar condición equipo	O(1)
Iterar sobre la lista que estaba el	O(n)
condicionales	O(1)
addLast	O(1)
Crear sublista	O(n)
Lt.size	O(1)

Ordenamiento quick sort	O(nlogn)
TOTAL	***O(nlogn)***

Nota: Al trabajar con mapas (funciones como contains o get) tomamos la complejidad de los casos promedio, teniendo en cuenta que se eligió un factor de carga el cual permite reducir la cantidad de colisiones.

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).

Tiempo de ejecución con todos los datos (large):	4.46
Memoria consumida todos los datos (large)	0.921875
Condición, herramientas y recursos utilizados:	• DISClib.ADT import list e import map (size, get, getValue, iterator, sublist). • Un ciclo for para la iteración de los datos en la lista de goles del jugador. • Condicionales dentro del ciclo para filtrar aquellos datos que cumplen las condiciones especificadas. • Función de ordenamiento.
Computador donde se ejecuta:	11th Gen Intel(R) Core i7 – 11800H @ 2.30 GHz 2.30GHz 16GB

Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

Valores de prueba:

Partidos de consulta = 15

Equipo = Italy

Condición del equipo = home

Tiempo

Entrada	**Tiempo de Ejecución Real [ms]**
Small	0.280

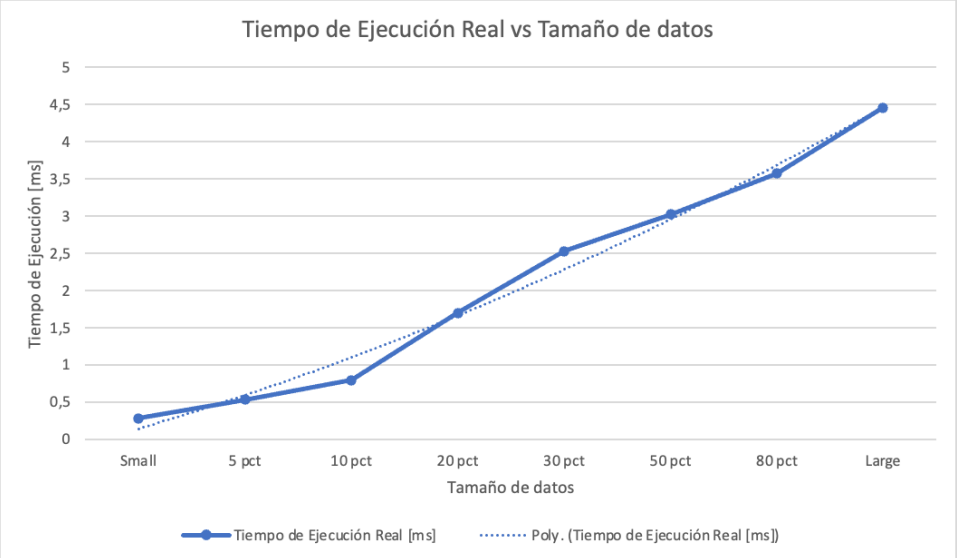
5 pct	0.533
10 pct	0.792
20 pct	1.70
30 pct	2.53
50 pct	3.02
80 pct	3.57
Large	4.46

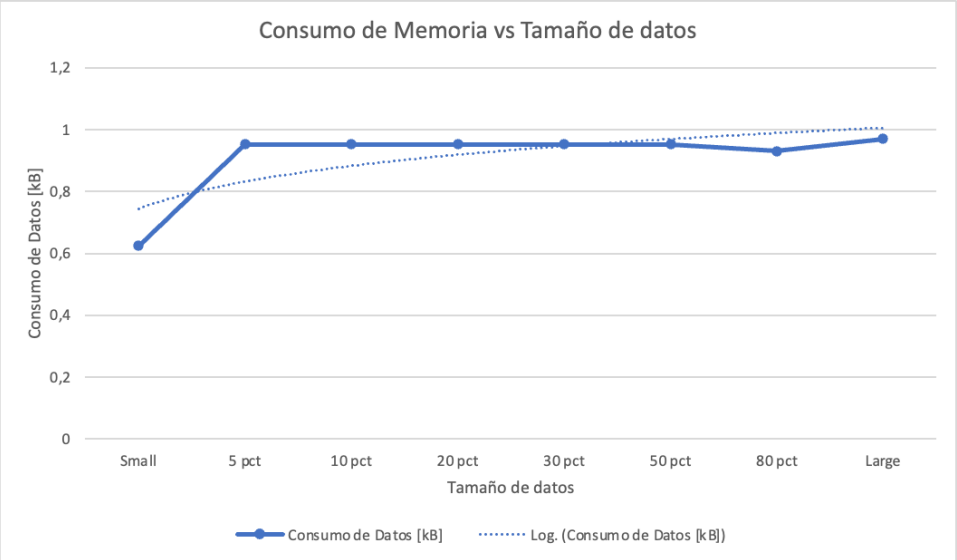
Memoria

Tamaño de datos	**Consumo de Datos [kB]**
Small	0.625
5 pct	0.953
10 pct	0.953
20 pct	0.953
30 pct	0.953
50 pct	0.953
80 pct	0.931
Large	0.971

Graficas

Las gráficas con la representación de las pruebas realizadas.





Análisis

Al realizar las pruebas vimos que el consumo de datos se mentienen en el mismo rango y bastantes cern entre los tamaños intermedios consumen la misma cantidad de datos y la máxima, ya que este disminuye al realizar las pruebas con el tamaño 80 pct. En cuanto al tiempo este si aumento progresivamente a medida que aumento el tamaño de los datos. Esto valores, aunque pueden estar influidos por algunos errores al tomar datos, se pueden explicar ya que, las acciones del requerimiento necesitan más tiempo si son más datos, pero no consumen más espacio al no crear tantas estructuras nuevas. Sin embargo, el ordenamiento al ser quick sort que es inplace si consume espacio, sobre todo si son muchos datos al igual que tiempo. Esto se vio reflejado en la complejidad, ya que si asumimos el peor caso y la complejidad más alta esta corresponde a nlogn que es la del ordenamiento. El segundo valor mas alto en O(n) que corresponde a la iteracion sobre la lista, pero esto solo ocurre al ingresar como condición home o away sino la complejidad algorítmica sería más baja al no tener que iterar gracias al mapa de torneos.

Requerimiento 2

Plantilla para el documentar y analizar cada uno de los requerimientos.

```python
def req_2(data_structs, n, jugador):
    """
    Función que soluciona el requerimiento 2
    """
    # TODO: Realizar el requerimiento 2

    #Se llama al mapa correspondiente a los anotadores y se obtiene su tamaño
    jugadores = data_structs["model"]["scorers"]
    size = mp.size(jugadores)

    #Se crea una lista vacía, y se inicia un contador para contar los goles por penales
    lista_final = lt.newList("ARRAY_LIST")
    subtotal_penales = 0

    #Se obtiene el valor a partir de la llave correspondiente al jugador que entra por parámetro
    pareja = mp.get(jugadores, jugador)
    lista_jugadores = me.getValue(pareja)

    #Se obtiene la cantidad de goles realizados por el jugador que entra por parámetro
    size_jugadores = lt.size(lista_jugadores)

    #Se itera sobre la lista con la información de cada jugador, se suma uno si el gol fue realizado por penal, y se agrega a la lista
    for jugador in lt.iterator(lista_jugadores):
        if jugador["penalty"] == "True":
            subtotal_penales += 1
        lt.addLast(lista_final,jugador)

    #Se ordena
    quk.sort(lista_final,cmpreq2)

    #Verificar si el numero total es mayor al ingresado, y se crea la sublista con el numero exacto
    if size_jugadores > n:
        lista_goles_final= lt.subList(lista_final,1,n)
        tamaño = lt.size(lista_goles_final)
    else:
        lista_goles_final = lista_final
        tamaño = lt.size(lista_goles_final)

    return size, size_jugadores, subtotal_penales, lista_goles_final, tamaño
```

Descripción

En el requerimiento 2 se saca del mapa de “scorers” el valor que se encuentra adjunto al nombre del jugador que se está buscando (esta es la llave), y se halla el tamaño de este mapa. Luego, se crea una lista vacía y se inicia un contador para los goles por penales. Posteriormente, se itera sobre estos partidos para identificar cuáles goles fueron realizados desde el punto penal, y se van sumando al contador. También se agregan a la lista vacía creada anteriormente, y se ordenan del más antiguo al más reciente. Finalmente, se crea una sublista según el número de goles ingresados por parámetro, y se obtiene su tamaño.

Entrada	Entran por parámetro: • data_structs (Datos almacenados en los archivos ingresados) • n (Número de goles de consulta) • jugador (Nombre completo del jugador)
Salidas	Retorna: • size (Cantidad total de anotadores) • size_jugadores (Cantidad de goles realizados por el jugador ingresado por parámetro) • subtotal_penales (Cantidad de goles realizados desde el punto penal por el jugador ingresado por parámetro) • lista_goles_final (Lista que contiene cada gol realizado por el jugador que entra por parámetro) • tamaño (Cantidad de goles realizados por el jugador ingresado por parámetro y filtrado por el número de goles de consulta)

Implementado (Sí/No)	Sí, implementado por Dayanna Rueda

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Tamaño de results = n

Tamaño de goalscorers = g

Tamaño de shootouts = s

Tal que n>g>s

Pasos	**Complejidad**
Iniciar contadores y crear listas	O(1)
Extraer la tupla llave-valor para el torneo seleccionado	O(1)
Extraer la lista (valor)	O(1)
lt.size	O(1)
Iterar sobre la lista	O(n)
Condicional penalty	O(1)
lt.addLast	O(1)
Ordenamiento quick sort	O(nlog(n))
Condicionales n	O(1)
lt.subList	O(n)
lt.size	O(1)
TOTAL	***O(nlog(n))***

Nota: Al trabajar con mapas (funciones como contains o get) tomamos la complejidad de los casos promedio, teniendo en cuenta que se eligió un factor de carga el cual permite reducir la cantidad de colisiones.

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).

Tiempo de ejecución con todos los datos (large):	0,16
Memoria consumida todos los datos (large)	2,67

Condición, herramientas y recursos utilizados:	• DISClib.ADT import list e import map (size, get, getValue, iterator, sublist). • Un ciclo for para la iteración de los datos en la lista de goles del jugador. • Condicionales dentro del ciclo para filtrar aquellos datos que cumplen las condiciones especificadas. • Función de ordenamiento.
Computador donde se ejecuta:	Apple M1 8GB

Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

Valores de prueba:

Número de goles de consulta: 7

Nombre completo del jugador: Michael Ballack

Tiempo

Entrada	Tiempo de Ejecución Real [ms]
Small	0,20
5 pct	0,09
10 pct	0,14
20 pct	0,13
30 pct	0,17
50 pct	0,24
80 pct	0,14
Large	0,16

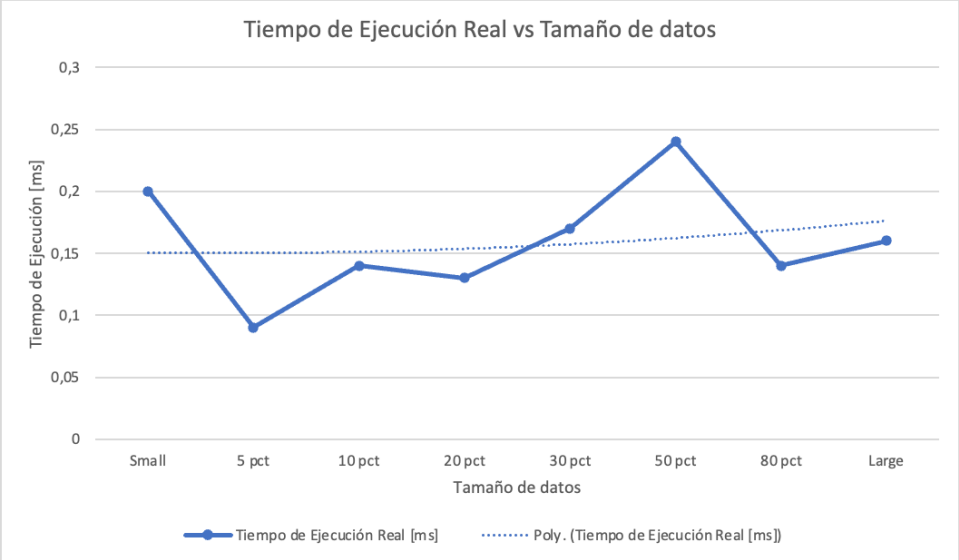
Memoria

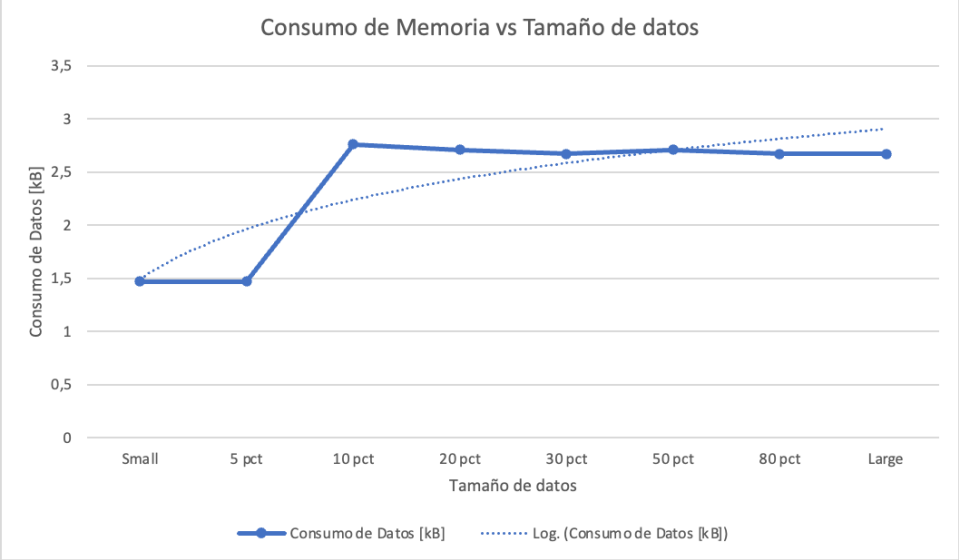
Tamaño de datos	Consumo de Datos [kB]
Small	1,47
5 pct	1,47
10 pct	2,76
20 pct	2,71
30 pct	2,67
50 pct	2,71
80 pct	2,67

Large	**2,67**

Graficas

Las gráficas con la representación de las pruebas realizadas.





Análisis

Análisis de resultados de la implementación, tener cuenta las pruebas realizadas y el análisis de complejidad.

A partir de la tabla de datos podemos observar una tendencia clara en el uso de memoria, ya que en los primeros dos valores más pequeños obtenemos el mismo valor. Y, a partir de allí para valores mayores el valor aumenta manteniéndose en un rango corto y constante. Por otro lado, el tiempo de ejecución muestra un movimiento más errático, donde no se ve una tendencia clara, pero la mayoría de los porcentajes de carga tienen un tiempo de ejecución menor que el menor de ellos. Por otro lado, en términos de complejidad, la mayoría de las operaciones presentan una complejidad algorítmica

constante, excepto la iteración de la lista que contiene la información del jugador que ingresa por parámetro y aquel que presenta la mayor complejidad sería el ordenamiento con quick sort que en peor de los casos sería O(nlog(n)).

Requerimiento 3

Plantilla para el documentar y analizar cada uno de los requerimientos.

```
def req_3(data_structs, equipo, fecha_inicial, fecha_final):
    """
    Función que soluciona el requerimiento 3
    """
   #Se llama al mapa de los partidos en goalscorers identificados por fecha- equipo local- equipo viaitante
    goalscorers_map = data_structs["model"]["home_away_team_goalscorers"]

    #Se llama al mapa correspondiente a todos los equipos
    equipos = data_structs["model"]["team"]

    #Se crea la lista donde se almacenarán los partidos filtrados, y se inician contadores para subtotales
    total_partidos = lt.newList("ARRAY_LIST")
    size_subtotal_local = 0
    size_subtotal_visitante = 0

    #Se obtiene el valor a partir de la llave correspondiente al equipo que entra por parámetro
    por_equipo = mp.get(equipos, equipo)
    partidos=me.getValue(por_equipo)

    #Se itera sobre la lista de partidos por cada equipo, y se filtra por fecha
    for partido in lt.iterator(partidos):
        if partido["date"] >= fecha_inicial and partido["date"] <= fecha_final:

    #Se filtra según la condición del equipo y se suma al contador
            if partido["home_team"] == equipo:
                size_subtotal_local += 1

            if partido["away_team"] == equipo:
                size_subtotal_visitante += 1

    #Se trae la información de penalty y own goal del mapa de goalscorers, y se añaden estos partidos a la lista vacía
            x = mp.contains(goalscorers_map, (partido["date"]+ "-"+partido["home_team"]+"-"+partido["away_team"]))
            if x:

                goal = me.getValue(mp.get(goalscorers_map, partido["date"]+ "-"+partido["home_team"]+"-"+partido["away_team"]))

                for g in lt.iterator(goal):
                    partido["penalty"] = g["penalty"]
                    partido["own_goal"] = g["own_goal"]
            else:
                partido["penalty"] = "Unknown"
                partido["own_goal"] = "Unknown"

            lt.addLast(total_partidos, partido)

    #Se halla el tamaño del total de equipos y el total de partido filtrado
    total_equipos = mp.size(equipos)
    size_total_partidos = lt.size(total_partidos)

    #Se ordena
    quk.sort(total_partidos,cmpreq3)
    return total_equipos, size_total_partidos, size_subtotal_local, size_subtotal_visitante, total_partidos
```

Descripción

En el requerimiento 3 se saca del mapa de “team” el valor que se encuentra adjunto al equipo que se está buscando (esta es la llave). Y, del mapa “home_away_team_goalscorers” el valor adjunto a los goles realizados en la fecha, con equipo local y visitante que se está buscando (esta es la llave). Adicionalmente, se crea una nueva lista, y se inician contadores correspondientes al subtotal si el equipo jugó en condición de local o visitante. Luego, se obtiene el valor a partir de la llave correspondiente al equipo que entra por parámetro. Después se itera sobre la lista de partidos por cada equipo, se filtra por fecha, y según la condición del equipo la cual se va sumando a los contadores. Posteriormente, se trae la información de penalty y own goal a partir del mapa “home_away_team_goalscorers”, y la iteración de sus valores. Finalmente, se hallan los tamaños del total de equipos a partir del mapa “team” y del total

de equipos filtrado por fecha más los valores adicionales de penalty y own goal. Para terminar, se ordenan los partidos del más reciente al más antiguo.

Entrada	Entran por parámetro: • data_structs (Datos almacenados en los archivos ingresados) • equipo (Nombre del equipo) • fecha_inicial (Fecha inicial del periodo a consultar) • fecha_final (Fecha final del periodo a consultar)
Salidas	Retorna: • total_equipos (Tamaño del total de equipos) • size_total_partidos (Tamaño del total de partidos filtrados) • size_subtotal_local (Cantidad de partidos donde el equipo ingresado por parámetro jugó en condición de local) • size_subtotal_visitante (Cantidad de partidos donde el equipo ingresado por parámetro jugó en condición de visitante) • total_partidos (Lista que almacena los partidos filtrados)
Implementado (Sí/No)	Sí, implementado por Dayanna Rueda

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Longitud lista results = n

Longitud lista goalscorers = g

g < n

Pasos	**Complejidad**
Crear listas e iniciar contadores	O(1)
Buscar si el equipo está en el mapa	O(1)
Extraer tupla llave-valor	O(1)
Extraer lista de partidos de la llave	O(1)
Iteración sobre lista partidos	O(n)*O(g)
Condicional para filtrar por fechas	O(1)
Condicional para filtrar por condición del equipo	O(1)
mp.contains goalscorers_map	O(1)
Condicional x	O(1)
Extraer tupla llave-valor	O(1)
Extraer lista de partidos de la llave	O(1)
Iteración sobre lista de partidos obtenida de goalscorers_map	O(g)
lt.addLast partidos	O(1)

mp.size de todos los equipos	O(1)
lt.size de lista de partidos obtenida de goalscorers_map	O(1)
Ordenamiento quick sort	O(nlg(n))
TOTAL	***O(n)*O(g)***

Nota: Al trabajar con mapas (funciones como contains o get) tomamos la complejidad de los casos promedio, teniendo en cuenta que se eligió un factor de carga el cual permite reducir la cantidad de colisiones.

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).

Tiempo de ejecución con todos los datos (large):	22,81
Memoria consumida todos los datos (large)	8,29
Condición, herramientas y recursos utilizados:	• DISClib.ADT import list e import map (size, get, getValue, iterator, contains). • Un ciclo for para la iteración de los datos en la lista de goles del jugador. • Condicionales dentro del ciclo para filtrar aquellos datos que cumplen las condiciones especificadas. • Función de ordenamiento.
Computador donde se ejecuta:	Apple M1 8GB

Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

Valores de prueba:

Nombre del equipo: Italy

Fecha inicial del periodo a consultar: 1939-01-01

Fecha final del periodo a consultar: 2018-12-31

Tiempo

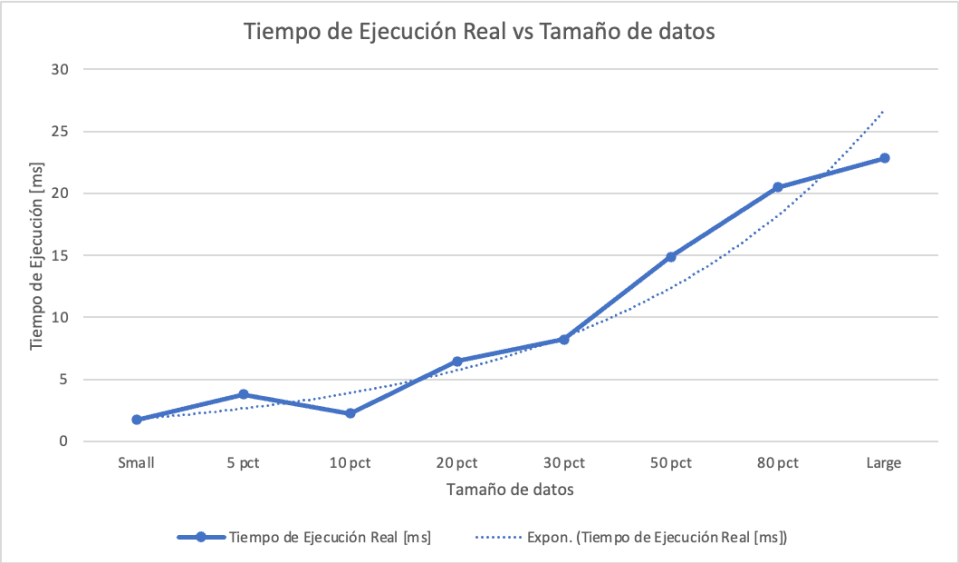
Entrada	Tiempo de Ejecución Real [ms]
Small	1,73
5 pct	3,79
10 pct	2,22
20 pct	6,44
30 pct	8,20
50 pct	14,86
80 pct	20,48
Large	22,81

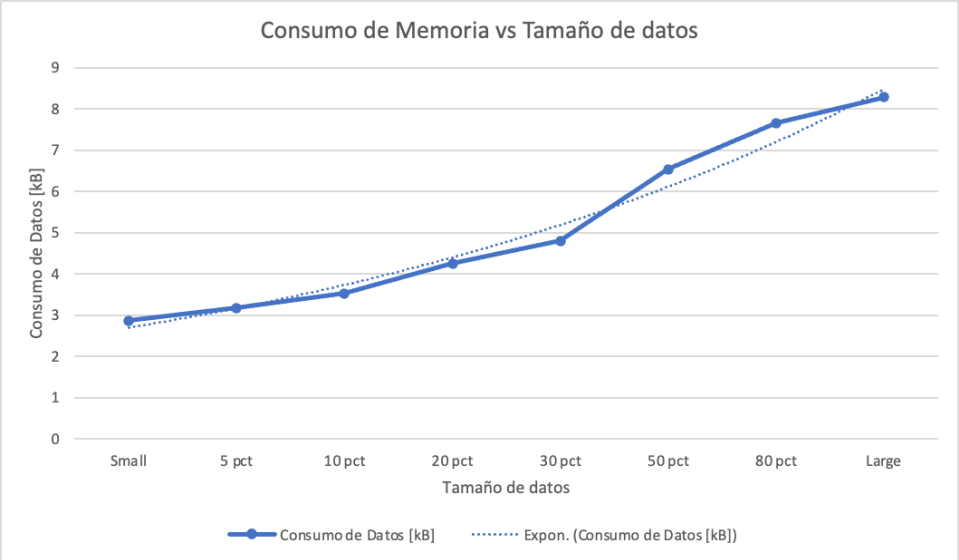
Memoria

Tamaño de datos	Consumo de Datos [kB]
Small	2,87
5 pct	3,18
10 pct	3,53
20 pct	4,25
30 pct	4,80
50 pct	6,54
80 pct	7,66
Large	8,29

Graficas

Las gráficas con la representación de las pruebas realizadas.





Análisis

Análisis de resultados de la implementación, tener cuenta las pruebas realizadas y el análisis de complejidad.

A partir de la tabla de datos y gráficas podemos evidenciar que a medida que el porcentaje de datos va aumentando en la entrada del requerimiento, el tiempo de ejecución y el consumo de memoria va aumentando progresivamente. Sin embargo, el tiempo de ejecución muestra una mayor variación de un resultado al otro. Respecto a la complejidad algorítmica la mayoría de las operaciones son constantes, a excepción de la iteración de la lista results y dentro de esta iteración, se itera la lista de goalscorers, se multiplican ambas para que en el peor caso sea O(n)*O(g). Finalmente, al implementar el ordenamiento quick sort se genera una complejidad de O(nlg(n)). En conclusión, la complejidad O(n*g) es mayor que O(nlog(n)), esto se debe a que la complejidad O(n*g) crece de manera lineal respecto a ambos valores n y g, mientras que la complejidad O(n*log(n)) crece de manera logarítmica con respecto a n y no depende de g.

Requerimiento 4

Plantilla para el documentar y analizar cada uno de los requerimientos.

```python
def req_4(data_structs, torneo,fecha_inicial, fecha_final):
    """
    Función que soluciona el requerimiento 4
    """


    shootout_map =data_structs["model"]["home_away_team_shootouts"]
    torneos = data_structs["model"]["tournament"]
    shootout = 0
    paises = lt.newList("ARRAY_LIST")
    ciudades = lt.newList("ARRAY_LIST")
    lista_final=lt.newList("ARRAY_LIST")
    #al recorrer results filtrar cada partido por fecha y torneo para agregarlo a otra lista,agregar cada pais y ciudad a sus respectivas listas
    tournament = mp.get(torneos, torneo)
    partidos=me.getValue(tournament)
    for partido in lt.iterator(partidos):
        if partido["date"] >= fecha_inicial and partido["date"] <= fecha_final:
                x=mp.contains(shootout_map,(partido["date"]+ "-"+partido["home_team"]+"-"+partido["away_team"]))
                if x:
                    shoot=me.getValue(mp.get(shootout_map,partido["date"]+ "-"+partido["home_team"]+"-"+partido["away_team"]))
                    partido["winner"] = shoot["winner"]
                    shootout += 1

                else:
                    partido["winner"] = "Unknown"
                lt.addLast(lista_final,partido)
                if lt.isPresent(paises,partido["country"]) == 0:
                    lt.addLast(paises,partido["country"])
                if lt.isPresent(ciudades,partido["city"]) == 0:
                    lt.addLast(ciudades,partido["city"])
    #Ver el tamaño de estas lista para encontrar los totales pedidos
    countries = lt.size(paises)
    cities = lt.size(ciudades)
    total_partidos = lt.size(lista_final)
    total_tournament = mp.size(torneos)
    quk.sort(lista_final,cmp_partidos_results)
    return total_tournament,lista_final,total_partidos, shootout,countries,cities
```

Descripción

Para este requerimiento empleamos un mapa para seleccionar todos los partidos que se llevaron a cabo en un torneo especifico, cuyas llaves eran los torneos y los valores listas de diccionarios con los partidos. Después iteramos en la lista seleccionada para filtrar por fecha los partidos y luego se empleó un mapa con el archivo de goalscorers cuyas llaves eran fecha, home y away team y solo tenía un único partido como valor con el fin de encontrar en partido exacto entre archivos. Esto se hizo con el fin de agregar la categoría winner a los diccionarios de cada partido. También empleando arraylist llevamos el conteo de las ciudades y países involucrados en el torneo.

Entrada	Entran por parámetro: • data_structs (Datos almacenados en los archivos ingresados) • Torneo • Fecha inicial • Fecha final
Salidas	Retorna: • El total de torneos con información disponible. • El total de partidos relevantes al torneo. • El total de países involucrados en el torneo. • El total de ciudades donde se disputan los partidos del torneo. • El total de partidos definidos por cobros de punto penal en el torneo. • El listado de los partidos disputados ordenados cronológicamente por fecha, nombre del por país y ciudad en que se disputaron los encuentros.
Implementado (Sí/No)	Sí, implementado por María José Mantilla

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Tamaño de results = n

Tamaño de goalscorers = g

Tamaño de shootouts = s

Tal que n>g>s

Pasos	**Complejidad**
Crear listas e iniciar contadores	O(1)
Buscar si el torneo está en el mapa	O(1)
Extraer tupla llave-valor	O(1)
Extraer lista de partidos de la llave	O(1)
Iteración sobre lista partidos	O(n)*(O(n)+O(n))
Condicional para filtrar por fechas	O(1)
Contains del mapa de shootouts	O(1)
Sacar el partido de shootouts que va con el de results	O(1)
Agregar al final de cada diccionario una llave con el valor extra	O(1)
addLast	O(1)
Ispresent de lista de paises	O(n)
Ispresent de la lista de cuidades	O(n)
lt.size	O(1)
Ordenamiento merge sort	O(n^3/2)
TOTAL	***O(n^2)***

Nota: Al trabajar con mapas (funciones como contains o get) tomamos la complejidad de los casos promedio, teniendo en cuenta que se eligió un factor de carga el cual permite reducir la cantidad de colisiones.

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).

Tiempo de ejecución con todos los datos (large):	0.78
Memoria consumida todos los datos (large)	5.39

Condición, herramientas y recursos utilizados:	• DISClib.ADT import list e import map (size, get, getValue, iterator, sublist). • Un ciclo for para la iteración de los datos en la lista de goles del jugador. • Condicionales dentro del ciclo para filtrar aquellos datos que cumplen las condiciones especificadas. • Función de ordenamiento.
Computador donde se ejecuta:	11th Gen Intel(R) Core i7 – 11800H @ 2.30 GHz 2.30GHz 16GB

Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

Valores de prueba:

Torneo = Copa América

Fecha inicial = 1955-06-01

Fecha final = 2022-06-30

Tiempo

Entrada	Tiempo de Ejecución Real [ms]
Small	0.04
5 pct	0.15
10 pct	0.34
20 pct	0.41
30 pct	0.49
50 pct	0.56
80 pct	0.64
Large	0.78

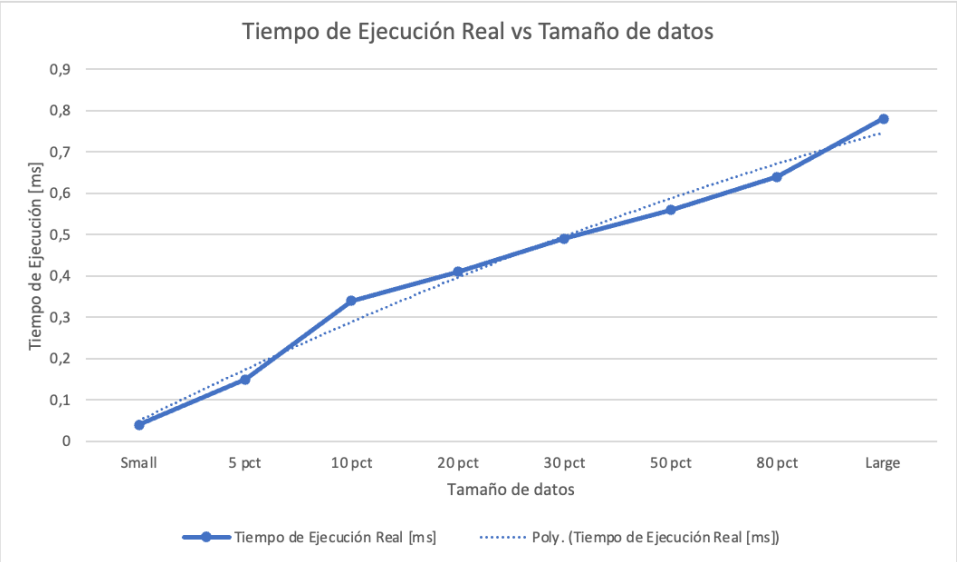
Memoria

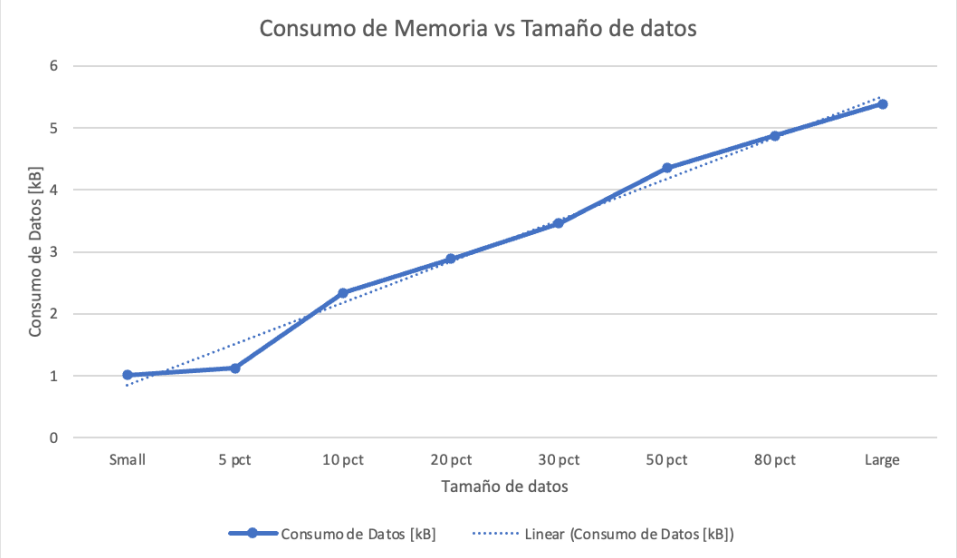
Tamaño de datos	Consumo de Datos [kB]
Small	1.015
5 pct	1.12
10 pct	2.34
20 pct	2.89
30 pct	3.46

50 pct	4.35
80 pct	4.87
Large	**5.39**

Graficas

Las gráficas con la representación de las pruebas realizadas.



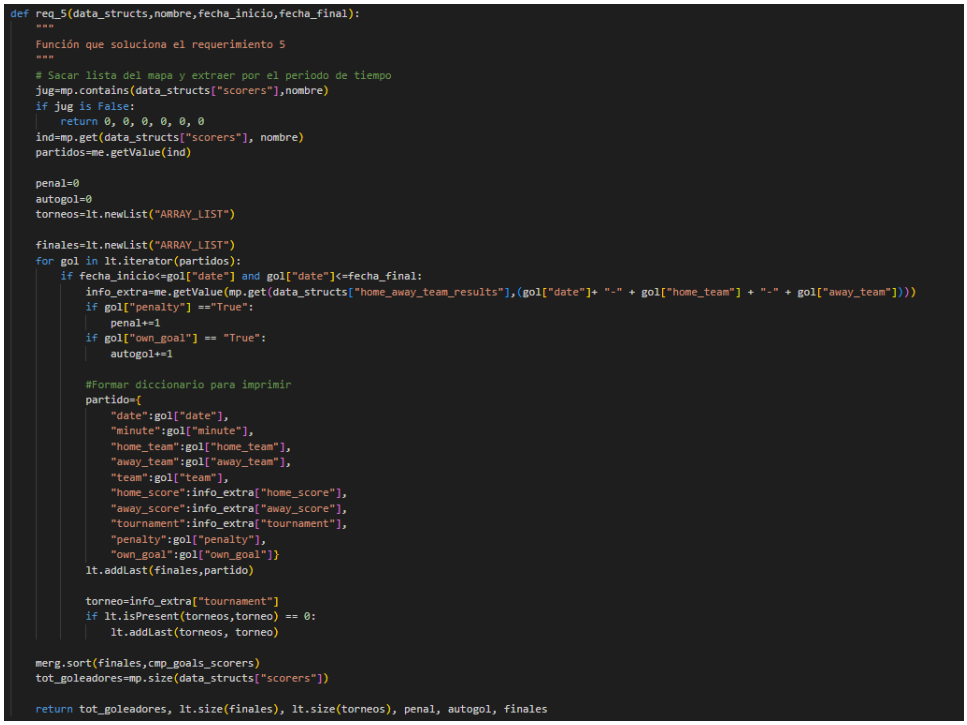


Análisis

Con base en las pruebas realizados, vimos que, en este requerimiento gastó más espacio que tiempo, aunque este si aumento progresivamente. Esto se debe a que se crean varias estructuras nuevas, al igual que se agregan llaves a todos los diccionarios. Por otro lado, la complejidad en este caso fue de O(n^2), ya que, aunque el ordenamiento que corría más rápido con menor complejidad era merge , pero esta no indica la complejidad del algoritmo debido a que se itera sobre la lista n y en su interior se utiliza un ispresent en dos listas lo que aumenta la complejidad en el peor caso.

Requerimiento 5

Plantilla para el documentar y analizar cada uno de los requerimientos.



```python
def req_5(data_structs,nombre,fecha_inicio,fecha_final):
    """
    Función que soluciona el requerimiento 5
    """
    # Sacar lista del mapa y extraer por el periodo de tiempo
    jug=mp.contains(data_structs["scorers"],nombre)
    if jug is False:
        return 0, 0, 0, 0, 0, 0
    ind=mp.get(data_structs["scorers"], nombre)
    partidos=me.getValue(ind)

    penal=0
    autogol=0
    torneos=lt.newList("ARRAY_LIST")

    finales=lt.newList("ARRAY_LIST")
    for gol in lt.iterator(partidos):
        if fecha_inicio<=gol["date"] and gol["date"]<=fecha_final:
            info_extra=me.getValue(mp.get(data_structs["home_away_team_results"],(gol["date"]+ "-" + gol["home_team"] + "-" + gol["away_team"])))
            if gol["penalty"] =="True":
                penal+=1
            if gol["own_goal"] == "True":
                autogol+=1

            #Formar diccionario para imprimir
            partido={
                "date":gol["date"],
                "minute":gol["minute"],
                "home_team":gol["home_team"],
                "away_team":gol["away_team"],
                "team":gol["team"],
                "home_score":info_extra["home_score"],
                "away_score":info_extra["away_score"],
                "tournament":info_extra["tournament"],
                "penalty":gol["penalty"],
                "own_goal":gol["own_goal"]}
            lt.addLast(finales,partido)

            torneo=info_extra["tournament"]
            if lt.isPresent(torneos,torneo) == 0:
                lt.addLast(torneos, torneo)

    merg.sort(finales,cmp_goals_scorers)
    tot_goleadores=mp.size(data_structs["scorers"])

    return tot_goleadores, lt.size(finales), lt.size(torneos), penal, autogol, finales
```

Descripción

Este requerimiento es lo que hace sacar del mapa de “scorers” el valor que se encuentra adjunto al nombre del jugador que se está buscando (esta es la llave). El valor que se extrae son todos los goles que ha hecho este jugador, luego se hace una filtración de estos goles por fecha, buscando que estén dentro del rango de fechas dado. Luego, por cada partido se van agregando los datos necesarios al diccionario. Además, se busca cada partido en el mapa de “home_away_team” de results para poder encontrar en que torneo se marcó ese gol. Por último, se ordenan los datos de los goles más recientes a los más antiguos.

Entrada	Entran como parámetros: • Data Structs -> Datos • Nombre Jugador • Fecha inicial

	• Fecha Final
Salidas	Retorna: - Total, de jugadores que han hecho gol encontrados - Total, goles por jugador - Total, torneos donde hizo gol el jugador - Total, goles por penal - Total, autogoles obtenidos - Lista de goles con respectivos datos
Implementado (Sí/No)	Si, implementado por Gabriela Zambrano

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Longitud lista results=n

Longitud lista goalscorers=g

Tal que n>g

Pasos	**Complejidad**
Buscar si el jugador está en el mapa	O(g)
Extraer tupla llave, valor	O(g)
Extraer valor	O(1)
Inicialización de contadores y listas	O(1)
Iteracion en goles del jugador	O(g)=O(g*g)
Comprobación filtros	O(1)
Sacar valor del mismo partido en lista results	O(1)
Sumar datos a contadores	O(1)
Creación diccionario	O(1)
Add last	O(1)
Comprobación de torneo y adición de torneo	O(g)*O(1)=O(g)
Ordenamiento con Merge	O(g^3/2)
TOTAL	***O(g**2)***

Nota: Al trabajar con mapas (funciones como contains o get) tomamos la complejidad de los casos promedio, teniendo en cuenta que se eligió un factor de carga el cual permite reducir la cantidad de colisiones.

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).

Tiempo de ejecución con todos los datos (large):	0,58

Memoria consumida todos los datos (large)	10,5
Condición, herramientas y recursos utilizados:	• DISClib.ADT import list e import map (newList, iterator, addLast, size, subList,isPresent, getElement, contains, get, getValue). • Un ciclo for para la iteración de los datos en la lista de goles del jugador. • Condicionales dentro del ciclo para filtrar aquellos datos que cumplen las condiciones especificadas.
Computador donde se ejecuta:	Intel(R) Core(TM) i5-10300H CPU @ 2.50GHz 2.50 GHz (8 GB Windows 11 Home)

Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

Valores de prueba:

Nombre: Lionel Messi

Fecha Inicio: 2004-01-30

Fecha Fin: 2022-10-15

Tiempo

Entrada	Tiempo de Ejecución Real [ms]
Small	0,07
5 pct	0,14
10 pct	0,19
20 pct	0,27
30 pct	0,3
50 pct	0,32
80 pct	0,41
Large	0,58

Memoria

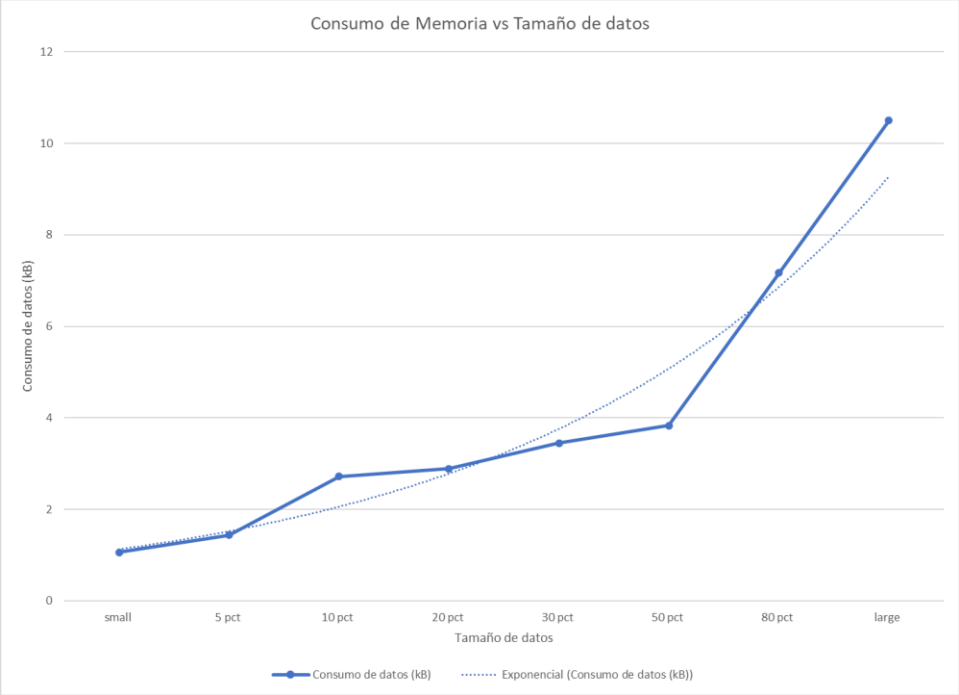
Tamaño de datos	Consumo de Datos [kB]
Small	1,06
5 pct	1,44

10 pct	2,72
20 pct	2,89
30 pct	3,45
50 pct	3,83
80 pct	7,17
Large	10,5

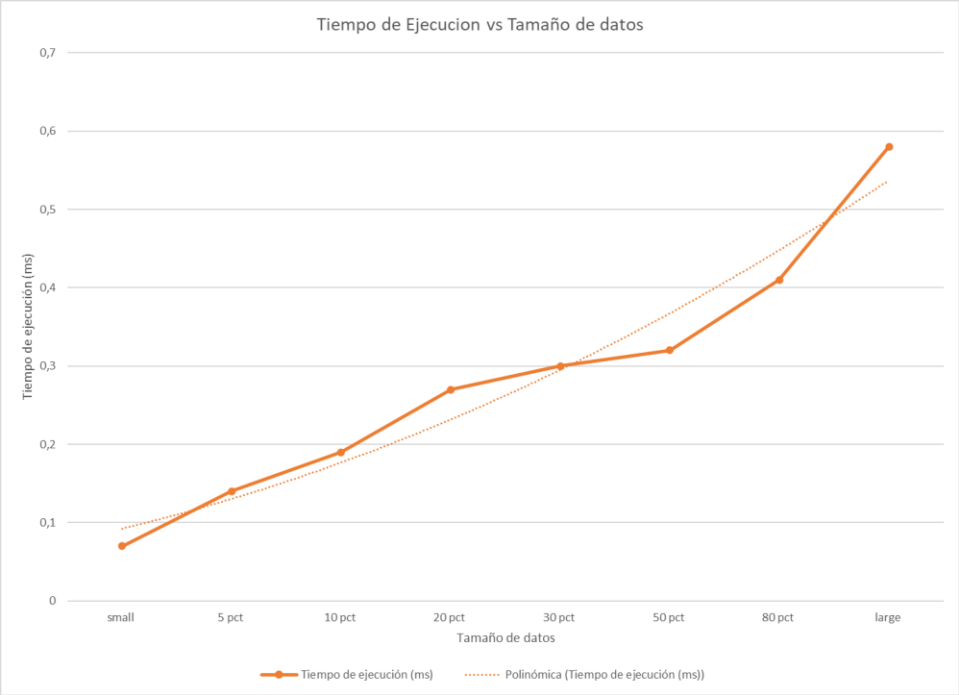
Graficas

Las gráficas con la representación de las pruebas realizadas.

Consumo de Datos



Tiempo de Ejecución



Análisis

Análisis de resultados de la implementación, tener cuenta las pruebas realizadas y el analisis de complejidad.

A partir de las gráficas dadas se puede observar que el tiempo de ejecución aumenta de forma polinómica, casi cuadrática, a medida que la cantidad de datos se hace mayor, esto va de acuerdo con la complejidad temporal que se calculó anteriormente. Esta fue de g**2, lo cual estaría acorde a la gráfica que se puede observar, en la parte superior. Por otro lado, con respecto al consumo de datos, este aumento de forma potencial a medida que aumentaban los datos, esto se encuentra relacionado a la creación de las listas array en las cuales entre mayor sean los datos, más aumenta su tamaño, los cual consume más espacio.

Requerimiento 6

Plantilla para el documentar y analizar cada uno de los requerimientos.

```python
def req_6(data_structs,n,torneo,anio):
    """
    Función que soluciona el requerimiento 6
    """
    # TODO: Realizar el requerimiento 6
    #Extraer todos los torneos de ese año
    an_io=mp.contains(data_structs["anios_torneo"],anio)
    if an_io:
        ind=mp.get(data_structs["anios_torneo"],anio)
        torneos=me.getValue(ind)
    else:
        return(0,0,0,0,0,0,0,0)

    #Extraer el torneo especifico
    tor=mp.contains(torneos,torneo)
    if tor:
        ind=mp.get(torneos,torneo)
        partidos=me.getValue(ind)
    else:
        return(0,0,0,0,0,0,0,0)

    #Trabajar con los datos
    cant_p=lt.size(partidos)
    equipo=mp.newMap(cant_p*3,
                     maptype="PROBING",
                     loadfactor=0.7)

    paises=lt.newList("ARRAY_LIST")
    ciudades=mp.newMap(cant_p,
                       maptype="PROBING",
                       loadfactor=0.7)

    #Divide por equipos
    for partido in lt.iterator(partidos):
        add_data_map_simplificado(equipo,partido,"home_team")
        add_data_map_simplificado(equipo,partido,"away_team")
        add_data_map_simplificado(ciudades,partido,"city")
        if lt.isPresent(paises,partido["country"])==0:
            lt.addLast(paises,partido["country"])

    lista_final=lt.newList("ARRAY_LIST")
    partidos_totales=lt.size(partidos)
    #Saca informacion completa por equipos
    llaves=mp.keySet(equipo)
    tam=lt.size(llaves)
    for equip in lt.iterator(llaves):
        team=mp.get(equipo,equip)
        team=me.getValue(team)
        partidos=lt.size(team)
        goles_favor=0
        goles_contra=0
        puntos=0
        wins=0
        draws=0
        losses=0
        penal=0
        autogol=0
        goleador=lt.newList("ARRAY_LIST")
```

```
    for partido in lt.iterator(team):
        if partido["home_team"]==equip:
            goles_favor+=int(partido["home_score"])
            goles_contra+=int(partido["away_score"])
            if partido["home_score"]>partido["away_score"]:
                wins+=1
                puntos+=3
            elif partido["home_score"]<partido["away_score"]:
                losses+=1
            else:
                draws+=1
                puntos+=1
        else:
            goles_favor+=int(partido["away_score"])
            goles_contra+=int(partido["home_score"])
            if partido["home_score"]<partido["away_score"]:
                wins+=1
                puntos+=3
            elif partido["home_score"]>partido["away_score"]:
                losses+=1
            else:
                draws+=1
                puntos+=1

        g=mp.contains(data_structs["home_away_team_goalscorers"],(partido["date"]+ "-" + partido["home_team"] + "-" + partido["away_team"]))
        if g:
            g=me.getValue(mp.get(data_structs["home_away_team_goalscorers"],(partido["date"]+ "-" + partido["home_team"] + "-" + partido["away_team"])))
            if g["team"]==equip:
                if g["penalty"] =="True":
                    penal+=1
                if g["own_goal"] == "True":
                    autogol+=1
                lt.addLast(goleador,g)

    goleadores=mp.newMap(partidos,
                         maptype="PROBING",
                         loadfactor=0.7)
    for gol in lt.iterator(goleador):
        add_data_map_simplificado(goleadores,gol,"scorer")

    prom_min,scorer,matches,goals=encontrar_goleador(goleadores)

    datos_jugador=lt.newList("ARRAY_LIST")
    dt={"scorer":scorer,
        "goals": goals,
        "matches":matches,
        "avg_time [min]":prom_min}
    lt.addLast(datos_jugador,dt)
    gols=creartabla(datos_jugador,datos_jugador["elements"][0])


    #Formar diccionario para imprimir
    equip_final={"team":equip,
                 "total_points":puntos,
                 "goal_difference":(goles_favor-goles_contra),
                 "penalty_points":penal,
                 "matches":partidos,
                 "own_goals_points":autogol,
                 "wins":wins,
                 "draws":draws,
                 "losses":losses,
                 "goals_for":goles_favor,
                 "goals_against":goles_contra,
                 "top_scorer":gols}
```

z

```
    lt.addLast(lista_final,equip_final)

#Encontrar ciudad mas repetida
ciudad_max=""
num_rep=0
llaves_ciud=mp.keySet(ciudades)
for ciudad in lt.iterator(llaves_ciud):
    ciud=me.getValue(mp.get(ciudades,ciudad))
    if lt.size(ciud)>=num_rep:
        num_rep=lt.size(ciud)
        ciudad_max=ciudad

#Ordenar lista y sacar numero de equipos
merg.sort(lista_final,cmpx_estadisticas_equipo)
if lt.size(lista_final)>n:
    lista_final=lt.subList(lista_final,1,n)

return (mp.size(data_structs["anios_torneo"]),mp.size(torneos), lt.size(llaves), partidos_totales, lt.size(paises),mp.size(ciudades),ciudad_max,lista_final)
```

Descripción

Este requerimiento es lo que hace sacar del mapa de “anios_torneo” el valor del año que se ingresó. Luego se extrae de este mapa el valor del torneo que el usuario ingreso. El valor son todos los partidos de ese torneo especifico que se jugaron durante el año ingresado. Luego se pasa por todos estos partidos, para así calcular los puntos, los goles a favor, goles en contra, victorias, empates, derrotas y demás, de cada equipo que participo en ese torneo durante un periodo de tiempo específico. Adicionalmente, se van contando los países, ciudades, equipos (para el total de equipos se tienen en cuenta tanto los que aparecen como locales como los que aparecen como visitantes) y partidos totales, datos que se presentan en cada uno de los partidos del torneo. Ahora bien, primero hace un filtro para

sacar los datos por equipo, luego, agrega los datos faltantes (diferencia de goles y goleador, este se saca a partir del que tenga el mejor promedio de tiempo si hay más de uno que tengan los mismos goles). Después, ordena la lista final y, por último, recorta la lista dependiendo la cantidad de equipos que se hayan pedido al inicio.

Entrada	Entran como parámetros: • Data Structs -> Datos • Torneo • Año • Numero de equipos
Salidas	Retorna: - Total, años calendario disponibles en el historial - Total, de torneos jugados ese año - Total, de equipos en el torneo - Total, de países involucrados en el torneo - Total, de ciudades involucradas en el torneo - Nombre de la ciudad donde se jugaron más partidos - Lista de equipos con respectivos datos
Implementado (Sí/No)	Si, implementado por Gabriela Zambrano

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Longitud mapa anio_torneo =a

Total torneos=t

Longitud results=r

Longitud goalscorers=g

Tal que r>g>a>t

Pasos	**Complejidad**
Buscar si el año está en el mapa	O(1)
Extraer tupla llave, valor	O(1)
Extraer valor	O(1)
Buscar si torneo esta y extraer valor	O(t)
Inicialización de contadores, listas y mapas	O(1)
Iteración en partidos del torneo	O(r)*O(r)=O(r^2)
Añadir en mapas	O(1)
Comprobación de ciudad y adición de ciudad	O(r)

Iteración sobre equipos	O(r)*O(g)=O(r*g)
Sacar partidos equipo de mapa	O(1)
Inicialización de contadores y listas	O(1)
Iteración sobre partidos del equipo	O(r)
Sumar a contadores	O(1)
Encontrar goles en lista goles	O(g)
Encontrar goleador	O(g)
Generar diccionario	O(1)
Encontrar ciudad max	O(r)
Ordenamiento con Merge	O(r^3/2)
Hacer sub_lista	O(r)
TOTAL	***O(r**2)***

Nota: Al trabajar con mapas (funciones como contains o get) tomamos la complejidad de los casos promedio, teniendo en cuenta que se eligió un factor de carga el cual permite reducir la cantidad de colisiones.

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).

Tiempo de ejecución con todos los datos (large):	2,94
Memoria consumida todos los datos (large)	21,43
Condición, herramientas y recursos utilizados:	• DISClib.ADT import list e import map (newList, iterator, addLast, size, subList,isPresent, getElement, contains, get, getValue). • Condicionales dentro de los ciclos para filtrar aquellos datos que cumplen las condiciones especificadas. • Condicional para delimitar la cantidad de datos que deben ser retornados a partir de la condición n ingresada por el usuario.
Computador donde se ejecuta:	Intel(R) Core(TM) i5-10300H CPU @ 2.50GHz 2.50 GHz (8 GB Windows 11 Home)

Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

Valores de prueba:

Toreno: Copa América

Numero de equipos: 15

Año: 2019

Tiempo

Entrada	Tiempo de Ejecución Real [ms]
Small	0,13
5 pct	1,38
10 pct	1,78
20 pct	2,1
30 pct	2,19
50 pct	2,76
80 pct	2,58
Large	2,94

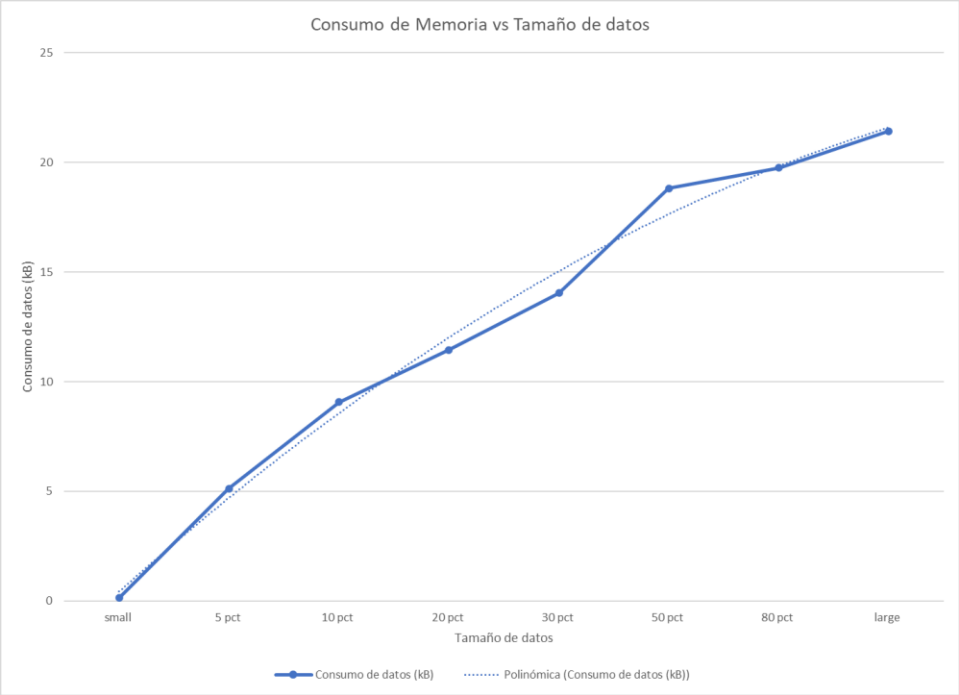
Memoria

Tamaño de datos	Consumo de Datos [kB]
Small	0,14
5 pct	5,12
10 pct	9,07
20 pct	11,45
30 pct	14,05
50 pct	18,82
80 pct	19,75
Large	21,43

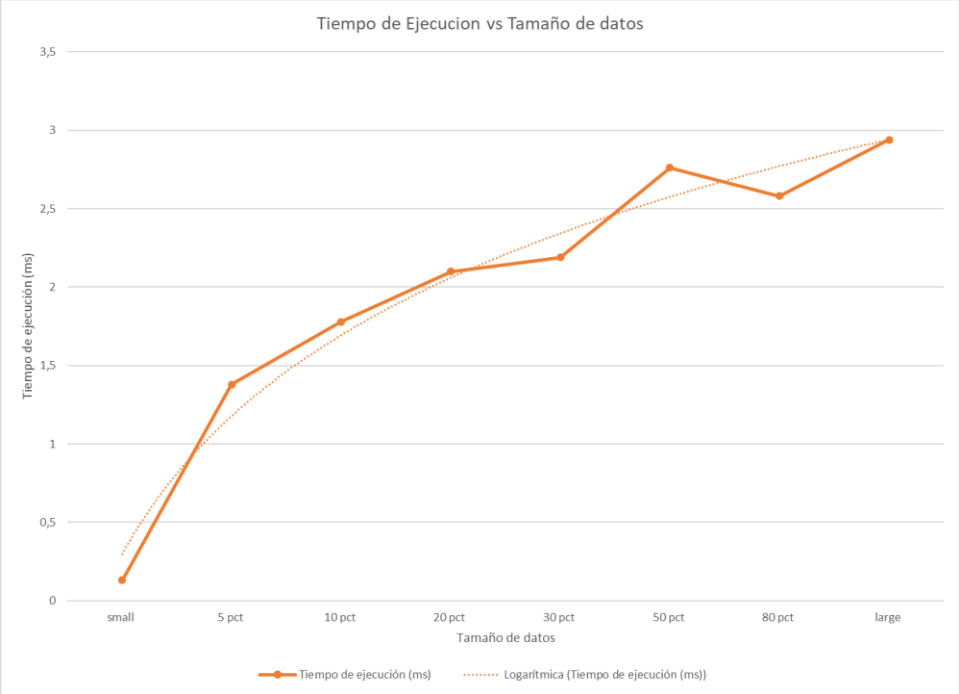
Graficas

Las gráficas con la representación de las pruebas realizadas.

Consumo de Datos



Tiempo de Ejecución



Análisis

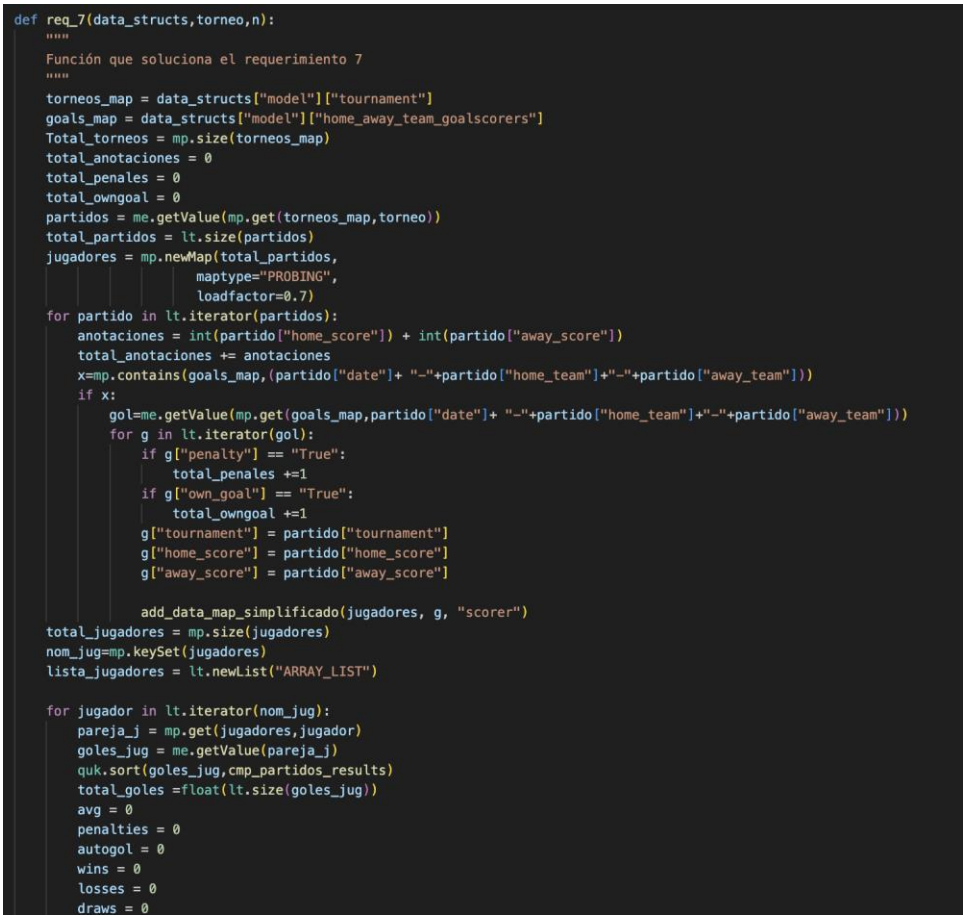
Análisis de resultados de la implementación, tener cuenta las pruebas realizadas y el analisis de complejidad.

A partir de las gráficas dadas se puede observar que el tiempo de ejecución aumenta de forma logarítmica a medida que la cantidad de datos se hace mayor, esto no va de acuerdo con la complejidad

temporal que se calculó anteriormente. Esta fue de r**2, lo cual no estaría de acuerdo con la gráfica, ya que lo que se esperaría sería una especie de curva polinómica, cuadrática, no una logarítmica. Por otro lado, con respecto al consumo de datos, este aumento de forma polinómica a medida que aumentaban los datos, esto se encuentra relacionado a la creación de mapas, sobre todo los mapas de “CHAINING” los cuales consumen más espacio a medida que se aumentan los tamaños de las LINKED_LIST.

Requerimiento 7

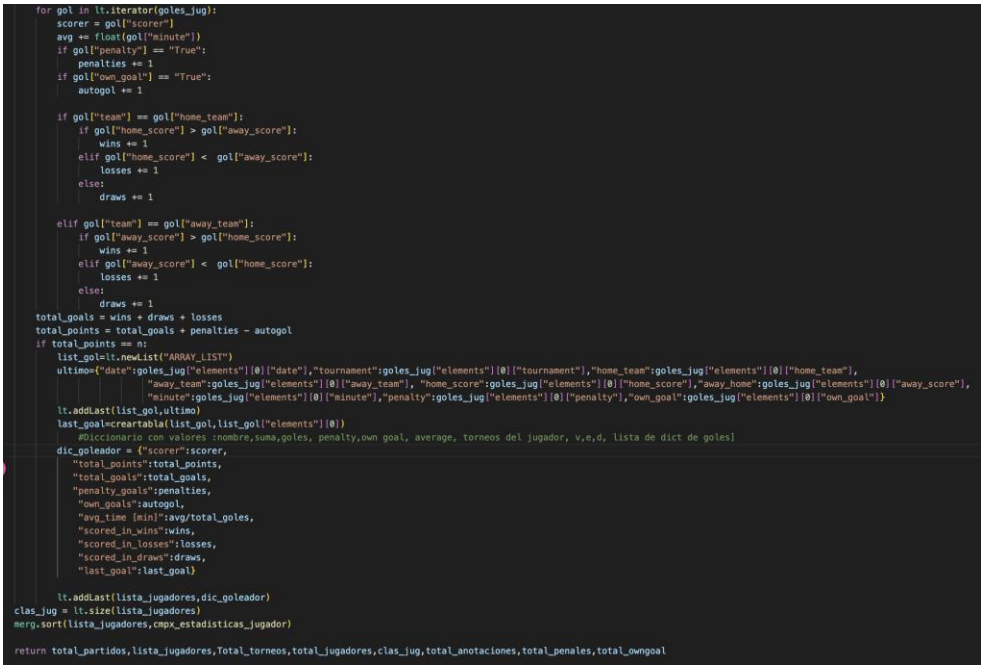
Plantilla para el documentar y analizar cada uno de los requerimientos.



```
def req_7(data_structs,torneo,n):
    """
    Función que soluciona el requerimiento 7
    """
    torneos_map = data_structs["model"]["tournament"]
    goals_map = data_structs["model"]["home_away_team_goalscorers"]
    Total_torneos = mp.size(torneos_map)
    total_anotaciones = 0
    total_penales = 0
    total_owngoal = 0
    partidos = me.getValue(mp.get(torneos_map,torneo))
    total_partidos = lt.size(partidos)
    jugadores = mp.newMap(total_partidos,
                          maptype="PROBING",
                          loadfactor=0.7)
    for partido in lt.iterator(partidos):
        anotaciones = int(partido["home_score"]) + int(partido["away_score"])
        total_anotaciones += anotaciones
        x=mp.contains(goals_map,(partido["date"]+ "-"+partido["home_team"]+"-"+partido["away_team"]))
        if x:
            gol=me.getValue(mp.get(goals_map,partido["date"]+ "-"+partido["home_team"]+"-"+partido["away_team"]))
            for g in lt.iterator(gol):
                if g["penalty"] == "True":
                    total_penales +=1
                if g["own_goal"] == "True":
                    total_owngoal +=1
                g["tournament"] = partido["tournament"]
                g["home_score"] = partido["home_score"]
                g["away_score"] = partido["away_score"]

                add_data_map_simplificado(jugadores, g, "scorer")
    total_jugadores = mp.size(jugadores)
    nom_jug=mp.keySet(jugadores)
    lista_jugadores = lt.newList("ARRAY_LIST")

    for jugador in lt.iterator(nom_jug):
        pareja_j = mp.get(jugadores,jugador)
        goles_jug = me.getValue(pareja_j)
        quk.sort(goles_jug,cmp_partidos_results)
        total_goles =float(lt.size(goles_jug))
        avg = 0
        penalties = 0
        autogol = 0
        wins = 0
        losses = 0
        draws = 0
```



```
    for gol in lt.iterator(goles_jug):
        scorer = gol["scorer"]
        avg += float(gol["minute"])
        if gol["penalty"] == "True":
            penalties += 1
        if gol["own_goal"] == "True":
            autogol += 1

        if gol["team"] == gol["home_team"]:
            if gol["home_score"] > gol["away_score"]:
                wins += 1
            elif gol["home_score"] <  gol["away_score"]:
                losses += 1
            else:
                draws += 1

        elif gol["team"] == gol["away_team"]:
            if gol["away_score"] > gol["home_score"]:
                wins += 1
            elif gol["away_score"] <  gol["home_score"]:
                losses += 1
            else:
                draws += 1
    total_goals = wins + draws + losses
    total_points = total_goals + penalties - autogol
    if total_points == n:
        list_gol=lt.newList("ARRAY_LIST")
        ultimo={"date":goles_jug["elements"][0]["date"],"tournament":goles_jug["elements"][0]["tournament"],"home_team":goles_jug["elements"][0]["home_team"],
                        "away_team":goles_jug["elements"][0]["away_team"], "home_score":goles_jug["elements"][0]["home_score"],"away_home":goles_jug["elements"][0]["away_score"],
                        "minute":goles_jug["elements"][0]["minute"],"penalty":goles_jug["elements"][0]["penalty"],"own_goal":goles_jug["elements"][0]["own_goal"]}
        lt.addLast(list_gol,ultimo)
        last_goal=creartabla(list_gol,list_gol["elements"][0])
            #Diccionario con valores :nombre,suma,goles, penalty,own goal, average, torneos del jugador, v,e,d, lista de dict de goles]
        dic_goleador = {"scorer":scorer,
          "total_points":total_points,
          "total_goals":total_goals,
          "penalty_goals":penalties,
           "own_goals":autogol,
           "avg_time [min]":avg/total_goles,
           "scored_in_wins":wins,
           "scored_in_losses":losses,
           "scored_in_draws":draws,
           "last_goal":last_goal}

        lt.addLast(lista_jugadores,dic_goleador)
    clas_jug = lt.size(lista_jugadores)
    merg.sort(lista_jugadores,cmpx_estadisticas_jugador)

    return total_partidos,lista_jugadores,Total_torneos,total_jugadores,clas_jug,total_anotaciones,total_penales,total_owngoal
```

Descripción

En el requerimiento 7, se llama al mapa de torneos y al mapa de goles. Después, se halla el tamaño de total de torneos, y se inician contadores para el total de anotaciones, penales y autogoles. Luego, se obtienen los valores de los torneos a partir de los torneos en el mapa de torneos como llaves, y se obtiene su tamaño. También se crea un nuevo mapa de jugadores a partir de este tamaño. Además, se itera sobre estos valores y se suman los goles de cada partido y se suma al contador de anotaciones. Asimismo, se busca el partido en el mapa de goles, se trae esta información para hallar si el gol fue realizado desde el punto penal o fue autogol y se suma a los contadores respectivos. De igual manera, se añaden los valores “tournament”, “home_score”, y “away_score”. Y, se agregan estos goles al mapa creado anteriormente. Después, se halla el tamaño del mapa, se obtiene una lista con todas las llaves del mapa. Luego, se crea una lista vacía llamada lista_jugadores. Esta lista de llaves es iterada para obtener los valores de cada llave, se halla el tamaño de estos y se ordenan. Luego, se inician contadores de promedio, penales, autogoles, partidos ganados, perdidos y empatados. Estos valores se iteran para hallar el minuto de cada gol y sumarlo al contador, y también sumar los goles y autogoles. También se suman los puntos si el equipo ganó, empató o perdió. Así como, el total de goles y puntos. Finalmente, se crea la condición de igualar a el puntaje n ingresado por parámetro, se crea una lista y se añaden los datos del gol respectivo. Finalmente, se crea el diccionario del goleador, y se añade a lista_jugadores, se obtiene su tamaño, y se ordena

Entrada	Entran por parámetro: • data_structs (Datos almacenados en los archivos ingresados) • Nombre del torneo que se desea consultar • El puntaje (N) de los jugadores dentro del torneo
Salidas	Retorna: • El total de torneos disponibles para consultar.

	• El total de anotadores que participaron en el torneo. • El total de partidos dentro del torneo. • El total de anotaciones o goles obtenidos durante los partidos del torneo. • El total de goles por penal obtenidos en ese torneo • El total de autogoles en que incurrieron los anotadores en ese torneo • El listado de anotadores debe estar ordenado por el criterio compuesto de sus estadísticas
Implementado (Sí/No)	Sí, implementado por María José Mantilla

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Total mapa torneos=t

Total mapa jugadores = j

Longitud results=n

Longitud goalscorers=g

Tal que n>g>j >t

Pasos	**Complejidad**
Crear mapas, listas y contadores vacíos	O(1)
Extraer partidos que pertenecen al torneo	O(1)
Iterar sobre estos partidos	O(n)
Contains de cada partido para cada gol del mapa de goalscorers	O(1)
Extraer gol y agregar datos al diccionario de cada partido	O(1)
Condicionales y contadores	O(1)
Agregar cada gol al mapa de jugadores	O(1)
Iterar sobre las llaves del mapa de jugadores	O(j)
Extraer lista de goles de cada jugador	O(1)
Ordenar cada lista por Quick sort	O(glog(g))
Iterar sobre cada lista de las llaves del mapa	O(g)
Condicionales y contadores	O(1)
Filtrar por número de puntos	O(1)
Crear diccionarios	O(1)
addLast	O(1)
lt.size	O(1)

Ordenar por merge sort	O(j)
	j*g*log(g)

Nota: Al trabajar con mapas (funciones como contains o get) tomamos la complejidad de los casos promedio, teniendo en cuenta que se eligió un factor de carga el cual permite reducir la cantidad de colisiones.

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).

Tiempo de ejecución con todos los datos (large):	5,94
Memoria consumida todos los datos (large)	19,34
Condición, herramientas y recursos utilizados:	• DISClib.ADT import list e import map (size, get, getValue, iterator, sublist). • Un ciclo for para la iteración de los datos en la lista de goles del jugador. • Condicionales dentro del ciclo para filtrar aquellos datos que cumplen las condiciones especificadas. • Función de ordenamiento.
Computador donde se ejecuta:	11th Gen Intel(R) Core i7 – 11800H @ 2.30 GHz 2.30GHz 16GB

Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

Valores de prueba:

Tiempo

Entrada	**Tiempo de Ejecución Real [ms]**
Small	0,24
5 pct	1,37
10 pct	2,03
20 pct	2,43
30 pct	3.32

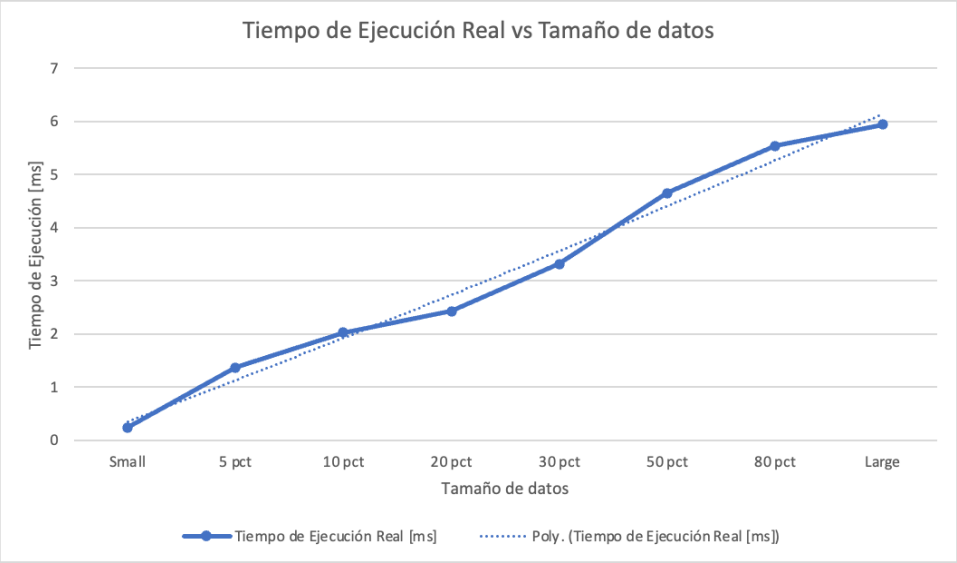
50 pct	4.65
80 pct	5,54
Large	5,94

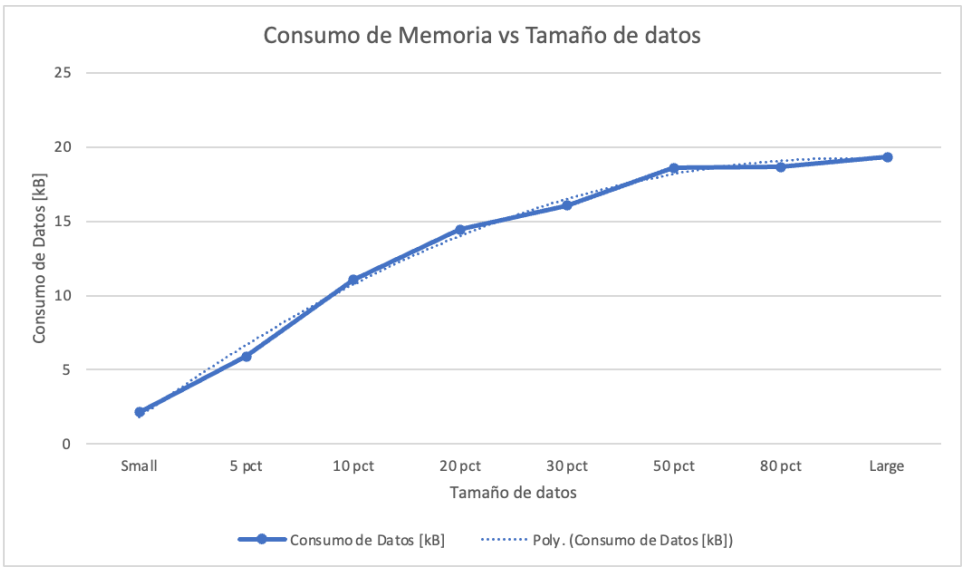
Memoria

Tamaño de datos	**Consumo de Datos [kB]**
Small	2,14
5 pct	5,89
10 pct	11,07
20 pct	14,45
30 pct	16,05
50 pct	18,60
80 pct	18,65
Large	19,34

Graficas

Las gráficas con la representación de las pruebas realizadas.





Análisis

Al analizar las pruebas de tiempo y espacio, vimos que, aunque en ambos aumentan progresivamente se consume más datos que tiempo, esto se debe a que dentro del algoritmo se crea un mapa desde 0 completamente, además de otras estructuras. En cuanto a la complejidad algorítmica, aunque se realizan varias iteraciones la que tiene más relevancia es la que itera sobre las llaves del mapa y después sobre la lista de cada llave. Además, como ordenamos estas listas en el interior la complejidad más alta es glog(g) ya que en el peor caso todo goalscorers estaría en una llave y al estar al interior del iterador del mapa, la complejidad más alta seria esta.

Requerimiento 8

Plantilla para el documentar y analizar cada uno de los requerimientos.

```python
def req_8(data_structs,nom_equipo,fecha_inicial,fecha_final):
    """
    Función que soluciona el requerimiento 8
    """
    # TODO: Realizar el requerimiento 8
    eq=mp.contains(data_structs["team"], nom_equipo)
    if eq:
        ind=mp.get(data_structs["team"],nom_equipo)
        partidos_equipo=me.getValue(ind)
    else:
        return 0,0,0,0,0,0,0

    #Mapa para guardar partidos por año
    cant_p=lt.size(partidos_equipo)
    partido_anio=mp.newMap(cant_p,
                           maptype="PROBING",
                           loadfactor=0.7)

    #Asignar partidos en mapa, filtracion por años y torneo, contadores
    partidos_local=0
    partidos_visitante=0
    partidos_mas_antiguo="2025-01-01"
    partidos_mas_reciente="1800-01-01"
    for partido in lt.iterator(partidos_equipo):
        if partido["date"]>=fecha_inicial and partido["date"]<=fecha_final and partido["tournament"] != "Friendly":
            llave=partido["date"][0:4]
            add_data_map_simplificado_llave(partido_anio,partido,llave)
            if partido["home_team"]==nom_equipo:
                partidos_local+=1
            elif partido["away_team"]==nom_equipo:
                partidos_visitante+=1

            if partido["date"]<partidos_mas_antiguo:
                partidos_mas_antiguo=partido["date"]
            if partido["date"]>partidos_mas_reciente:
                partidos_mas_reciente=partido["date"]
                part_reciente=partido

    if mp.size(partido_anio)==0:
        return(0,0,0,0,0,0,0)

    lista_final=lt.newList("ARRAY_LIST")
    #Diccionario imprimir para partido mas reciente
    lis_part_reciente=lt.newList("ARRAY_LIST")
    dic_partid_reciente={
                         "date":part_reciente["date"],
                         "home_team":part_reciente["home_team"],
                         "away_team":part_reciente["away_team"],
                         "home_score":part_reciente["home_score"],
                         "away_score":part_reciente["away_score"],
                         "country":part_reciente["country"],
                         "city":part_reciente["city"],
                         "tournament":part_reciente["tournament"]
                         }
    lt.addLast(lis_part_reciente,dic_partid_reciente)

    #Extraer estadisticas
    llaves= mp.keySet(partido_anio)
    for anio in lt.iterator(llaves):
        ani=me.getValue(mp.get(partido_anio,anio))
        partidos_total=lt.size(ani)
        goles_favor=0
        goles_contra=0
        puntos=0
        wins=0
        draws=0
        losses=0
        penal=0
        autogol=0
        goleador=lt.newList("ARRAY_LIST")
        for partido in lt.iterator(ani):
            if partido["home_team"]==nom_equipo:
                goles_favor+=int(partido["home_score"])
                goles_contra+=int(partido["away_score"])
                if partido["home_score"]>partido["away_score"]:
                    wins+=1
                    puntos+=3
                elif partido["home_score"]<partido["away_score"]:
                    losses+=1
                else:
```

```
    for partido in lt.iterator(ani):
        if partido["home_team"]==nom_equipo:
            goles_favor+=int(partido["home_score"])
            goles_contra+=int(partido["away_score"])
            if partido["home_score"]>partido["away_score"]:
                wins+=1
                puntos+=3
            elif partido["home_score"]<partido["away_score"]:
                losses+=1
            else:
                draws+=1
                puntos+=1
        else:
            goles_favor+=int(partido["away_score"])
            goles_contra+=int(partido["home_score"])
            if partido["home_score"]<partido["away_score"]:
                wins+=1
                puntos+=3
            elif partido["home_score"]>partido["away_score"]:
                losses+=1
            else:
                draws+=1
                puntos+=1

        g=mp.contains(data_structs["home_away_team_goalscorers"],(partido["date"]+ "-" + partido["home_team"] + "-" + partido["away_team"]))
        if g:
            g=me.getValue(mp.get(data_structs["home_away_team_goalscorers"],(partido["date"]+ "-" + partido["home_team"] + "-" + partido["away_team"])))
            if g["team"]==nom_equipo:
                if g["penalty"] =="True":
                    penal+=1
                if g["own_goal"] == "True":
                    autogol+=1
                lt.addLast(goleador,g)

    goleadores=mp.newMap(partidos_total,
                 maptype="PROBING",
                 loadfactor=0.7)
    for gol in lt.iterator(goleador):
        add_data_map_simplificado(goleadores,gol,"scorer")

    prom_min,scorer,matches,goals=encontrar_goleador(goleadores)

    datos_jugador=lt.newList("ARRAY_LIST")
    dt={"scorer":scorer,
        "goals": goals,
        "matches":matches,
        "avg_time [min]":prom_min}
    lt.addLast(datos_jugador,dt)
    gols=creartabla(datos_jugador,datos_jugador["elements"][0])


    #Hacer dic para imprimir
    equip_final={"year":anio,
                 "matches":partidos_total,
                 "total_points":puntos,
                 "goal_difference":(goles_favor-goles_contra),
                 "penalties":penal,
                 "own_goals":autogol,
                 "wins":wins,
                 "draws":draws,
                 "losses":losses,
                 "goals_for":goles_favor,
                 "goals_against":goles_contra,
                 "top_scorer":gols}

    lt.addLast(lista_final,equip_final)

#Ordenar lista
quk.sort(lista_final,cmp_rq8)


return mp.size(partido_anio),(partidos_local+partidos_visitante),partidos_local,partidos_visitante,partidos_mas_antiguo,lis_part_reciente,lista_final
```

Descripción

Este requerimiento es lo que hace sacar del mapa de “team” el valor del equipo que se está buscando. Luego se organizan todos los partidos por años en un nuevo mapa. Además, de buscar el partido más reciente y el más antiguo. Realiza el diccionario del partido más reciente, para imprimirlo. Después, empieza a sacar todos los datos por año, los puntos, los goles a favor, goles en contra, victorias, empates, derrotas y demás. Además, busca al goleador del año. Por último, genera el diccionario de cada año y realiza un ordenamiento, de los años más recientes a los más antiguos, con Quick sort.

Entrada	Entran como parámetros: • Data Structs -> Datos • Nombre de equipo • Fecha inicial • Fecha final
Salidas	Retorna: - Total, años calendario disponibles en el historial - Total, de partidos disputados - Total, de partidos disputados como local - Total, de partidos disputados como visitante - Fecha partido más antiguo - Lista con partido más reciente - Lista de años con respectivos datos
Implementado (Sí/No)	Si, implementado por Gabriela Zambrano

Análisis de complejidad

Análisis de complejidad de cada uno de los pasos del algoritmo

Longitud mapa teams=t

Longitud lista goalscorers=g

Cantidad años= a

Longitud results=r

Tal que r>g>t>a

Pasos	**Complejidad**
Buscar si el equipo está en el mapa	O(t)
Extraer tupla llave, valor	O(1)
Extraer valor	O(1)
Inicialización de contadores, listas y mapas	O(1)
Iteración en partidos del equipo	O(r)
Añadir en mapas	O(1)
Sumar a contadores	O(1)
Formar diccionario partido reciente	O(1)
Iteración sobre años	O(a)*O(r)=O(a*r)
Sacar partidos año de mapa	O(1)
Inicialización de contadores y listas	O(1)
Iteración sobre partidos del equipo en el año	O(r)
Sumar a contadores	O(1)
Encontrar goles en lista goles	O(1)
Encontrar goleador	O(g)

Generar diccionario	O(1)
Ordenamiento con Quick	O(alog(a))
TOTAL	***O(a*r)***

Nota: Al trabajar con mapas (funciones como contains o get) tomamos la complejidad de los casos promedio, teniendo en cuenta que se eligió un factor de carga el cual permite reducir la cantidad de colisiones.

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).

Tiempo de ejecución con todos los datos (large):	7,21
Memoria consumida todos los datos (large)	38,54
Condición, herramientas y recursos utilizados:	• DISClib.ADT import list e import map (newList, iterator, addLast, size, subList,isPresent, getElement, contains, get, getValue). • Condicionales dentro de los ciclos para filtrar aquellos datos que cumplen las condiciones especificadas. • Condicional para delimitar la cantidad de datos que deben ser retornados a partir de la condición n ingresada por el usuario.
Computador donde se ejecuta:	Intel(R) Core(TM) i5-10300H CPU @ 2.50GHz 2.50 GHz (8 GB Windows 11 Home)

Tablas de datos

Las tablas con la recopilación de datos de las pruebas.

Valores de prueba:

Toreno: France

Año inicial 2000

Año final: 2023

Tiempo

Entrada	Tiempo de Ejecución Real [ms]
Small	1,19
5 pct	2,81
10 pct	4,06
20 pct	4,8
30 pct	5,24
50 pct	5,63
80 pct	6,77
Large	7,21

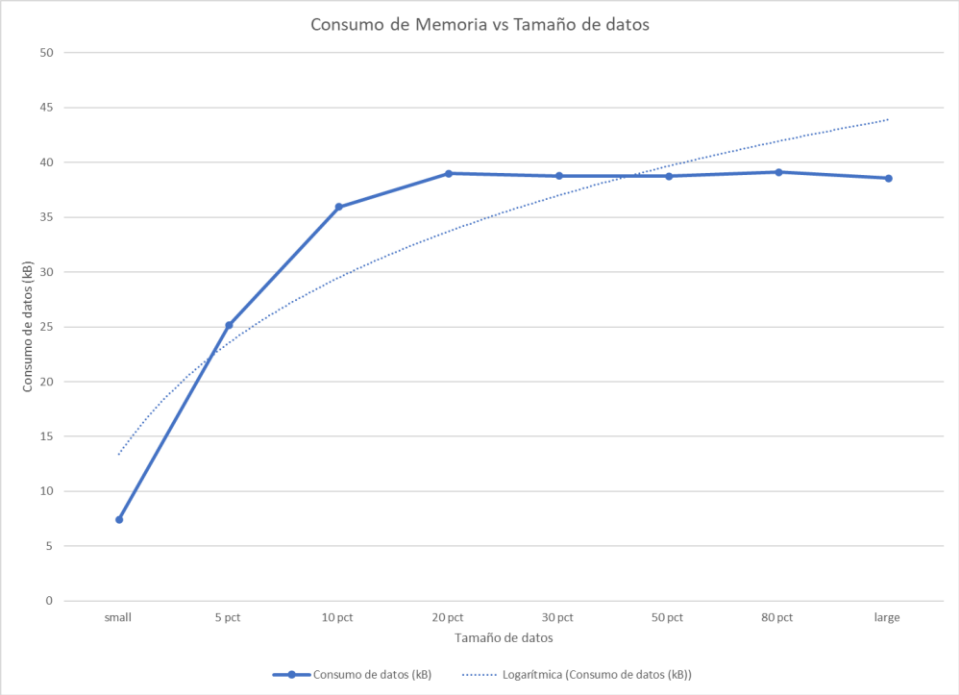
Memoria

Tamaño de datos	Consumo de Datos [kB]
Small	7,44
5 pct	25,14
10 pct	35,94
20 pct	38,98
30 pct	38,78
50 pct	38,75
80 pct	39,09
Large	38,54

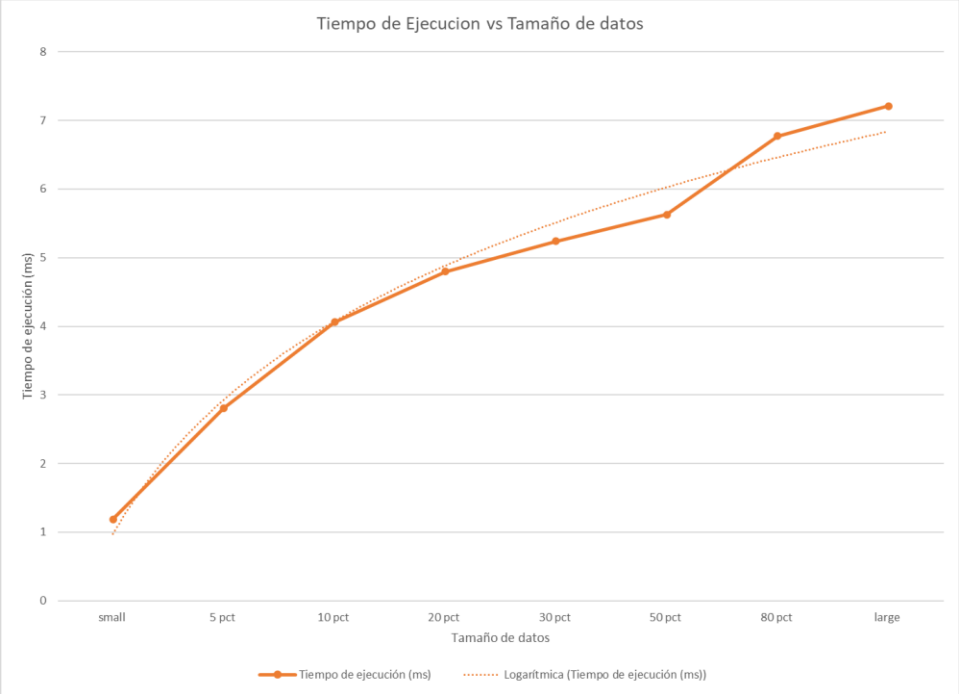
Graficas

Las gráficas con la representación de las pruebas realizadas.

Consumo de Datos



Tiempo de Ejecución



Análisis

Análisis de resultados de la implementación, tener cuenta las pruebas realizadas y el analisis de complejidad.

A partir de las gráficas dadas se puede observar que el tiempo de ejecución aumenta de forma logarítmica a medida que la cantidad de datos se hace mayor, esto no va de acuerdo con la complejidad

temporal que se calculó anteriormente. Esta fue de a*r, lo cual no estaría de acuerdo con la gráfica, ya que lo que se esperaría sería una especie de curva polinómica, teniendo en cuenta que a es menor a g. En cuanto al consumo de memoria, se observa un crecimiento logarítmico a medida que se incrementan los datos. Adicionalmente, esto nos puede decir que, a partir del ejemplo implementado, se llega a un punto en el que, al no agregar nuevos datos a los mapas, el consumo de memoria se estabiliza y deja de aumentar.