Proyecto: Asociación de Deportes

Análisis y Diseño de Algoritmos I

Juan José Cortés Rodríguez
Código: 2325109

Universidad del Valle
Facultad de Ingeniería
Escuela de Ingeniería de Sistemas y Computación

Diciembre 2025

Índice

1. Introducción

El presente documento describe el desarrollo e implementación de un sistema de gestión y ordenamiento para una asociación de deportes, que administra información de jugadores, equipos y sedes deportivas. Este proyecto constituye el trabajo final de la materia Análisis y Diseño de Algoritmos I y tiene como objetivo aplicar conceptos fundamentales de estructuras de datos y análisis de complejidad computacional.

1.1. Contexto del Problema

Una asociación de deportes desea realizar un análisis a fondo de su organización deportiva, con la finalidad de definir cuáles equipos y jugadores se verán recompensados con más recursos para sus entrenamientos, además de cuáles equipos y jugadores requieren planes para mejorar su rendimiento. Esta organización tiene varias sedes por todo el país, cada sede tiene equipos para diferentes deportes; a su vez los equipos están formados por jugadores.

1.2. Objetivos del Proyecto

El proyecto plantea los siguientes objetivos específicos:

- **Implementar dos soluciones algorítmicas diferentes** que resuelvan el mismo problema de ordenamiento jerárquico (Sedes $\rightarrow$ Equipos $\rightarrow$ Jugadores).
- **Analizar y comparar la complejidad temporal** de ambas soluciones tanto teórica como empíricamente usando pruebas.
- **Aplicar estructuras de datos avanzadas** vistas en el curso.
- **Calcular estadísticas relevantes** sobre el conjunto de datos, incluyendo promedios, valores extremos y rankings.
- **Validar la correctitud de las implementaciones** mediante casos de prueba conocidos y verificables.

2. Descripción del Problema

2.1. Especificaciones de Entrada

2.1.1. Estructura de Datos de Entrada

Los datos de entrada se proporcionan en archivos Python (`.py`) que definen objetos de las clases `Jugador`, `Equipo` y `Sede`:

```python
from classes import Jugador, Equipo, Sede

# Definicion de jugadores
j1 = Jugador("Sofia Garcia", 21, 66)
j2 = Jugador("Alejandro Torres", 27, 24)
j3 = Jugador("Valentina Rodriguez", 19, 15)
# ... mas jugadores

# Definicion de equipos
e1 = Equipo("Futbol", [j1, j2])
e2 = Equipo("Volleyball", [j3, j4, j5])
# ... mas equipos

# Definicion de sedes
s1 = Sede("Sede Cali", [e1, e2])
s2 = Sede("Sede Medellin", [e3, e4])
```

Listing 1: Ejemplo de archivo de entrada

2.1.2. Información de Jugadores

Cada jugador contiene:

- **ID:** Identificador único asignado secuencialmente (1, 2, 3, ...)
- **Nombre:** Cadena de texto con el nombre completo
- **Edad:** Número entero positivo
- **Rendimiento:** Número entero en el rango [0, 100]

2.2. Criterios de Ordenamiento

El sistema debe ordenar la información según los siguientes criterios:

2.2.1. Ordenamiento de Jugadores

Los jugadores se ordenan según la siguiente prioridad:

1. **Rendimiento ASCENDENTE:** Del menor al mayor rendimiento.
2. **En caso de empate, Edad DESCENDENTE:** Del jugador más viejo al más joven.

3. **En caso de que tambien haya empate con la edad, implementé que se desempate con ID ASCENDENTE:** Del menor al mayor ID.

Ejemplo: Si dos jugadores tienen rendimiento 75, se ordena primero el de mayor edad. Si además tienen la misma edad, se ordena primero el de menor ID.

2.2.2. Ordenamiento de Equipos

Los equipos se ordenan según:

1. **Promedio de rendimiento ASCENDENTE:** Del equipo con menor promedio al mayor.
2. **En caso de empate, Cantidad de jugadores DESCENDENTE:** Del equipo más grande al más pequeño.

El promedio de rendimiento de un equipo se calcula como:

$$\text{Promedio}_{\text{equipo}} = \frac{\sum_{i=1}^{n} \text{rendimiento}_i}{n}$$

donde n es el número de jugadores del equipo.

2.2.3. Ordenamiento de Sedes

Las sedes se ordenan según:

1. **Suma de promedios de rendimiento ASCENDENTE:** De la sede con menor suma al mayor.
2. **En caso de empate, Total de jugadores DESCENDENTE:** De la sede con más jugadores a la de menos.

La suma de promedios de una sede se calcula como:

$$\text{Suma}_{\text{sede}} = \sum_{j=1}^{k} \text{Promedio}_{\text{equipo}_j}$$

donde k es el número de equipos en la sede.

2.3. Especificaciones de Salida

El sistema debe generar archivos de texto con el siguiente formato:

```
PRIMERA/SEGUNDA SOLUCION


Sede [Nombre], Rendimiento: [suma_promedios]

[Deporte], Rendimiento: [promedio]
{id1, id2, id3, ...}

[Deporte], Rendimiento: [promedio]
{id1, id2, id3, ...}


Ranking Jugadores:
{id1, id2, id3, ...}


Equipo con mayor rendimiento: [Deporte] [Sede]

Equipo con menor rendimiento: [Deporte] [Sede]

Jugador con mayor rendimiento: { id , nombre , rendimiento }

Jugador con menor rendimiento: { id , nombre , rendimiento }

jugador mas joven: { id , nombre , edad }

jugador mas veterano: { id , nombre , edad }

Promedio de edad de los jugadores: [valor]

Promedio de rendimiento de los jugadores: [valor]
```

Listing 2: Formato de salida esperado

2.4. Estadísticas Requeridas

Además del ordenamiento, el sistema debe permitir identificar el equipo con mejor y peor rendimiento junto con su sede correspondiente, así como el jugador con mayor y menor rendimiento. También debe determinar el jugador más joven y el más veterano, y calcular el promedio de edad y el promedio de rendimiento de todos los jugadores.

3. Primera Solución - Árboles Rojinegros

3.1. Idea de la Solución

La primera solución implementa el problema utilizando árboles rojinegros como estructura de datos principal para mantener ordenados jugadores, equipos y sedes. Como se mencionó en el curso, un árbol rojinegro es un árbol binario de búsqueda balanceado que cumple con propiedades específicas de coloración que garantizan un balanceo, manteniendo la altura del árbol en O(log n), se implementó creando:

1. **Árbol rojinegro de jugadores:** Los jugadores se almacenan ordenados por rendimiento, edad e ID.
2. **Árboles rojinegros de equipos:** Cada equipo gestiona sus jugadores en un árbol, y los equipos se ordenan por rendimiento promedio.
3. **Árbol rojinegro de sedes:** Las sedes se organizan según el rendimiento total de los equipos.

3.2. Estructuras de Datos Utilizadas

La implementación utiliza una jerarquía de clases basada en árboles rojinegros:

- **Clase base NodoRB:** Almacena los datos del elemento (id, nombre, criterios de ordenamiento) junto con referencias a nodos hijos, padre y el color del nodo (rojo o negro).
- **Clase ArbolRojiNegro:** Implementa las operaciones fundamentales de inserción, rotaciones (izquierda/derecha) y restauración de propiedades (fixup).
- **Nodos y árboles especializados:** Se crearon nodos especializados que heredan de **NodoRB** (NodoEquipoRB y NodoSedeRB) para almacenar información específica de equipos y sedes, junto con tres árboles especializados que heredan de ArbolRojiNegro:
  - **ArbolJugadores:** Ordena jugadores por rendimiento → edad → ID
  - **ArbolEquipos:** Ordena equipos por promedio de rendimiento → cantidad de jugadores
  - **ArbolSedes:** Ordena sedes por suma de promedios → total de jugadores

En cada arbol implementé internamente su propia función de comparación según los criterios específicos de ordenamiento requeridos.

3.3. Algoritmos Implementados

Los siguientes algoritmos se implementaron para cada Arbol

3.3.1. Inserción en Árbol Rojinegro (insertar)

Inserta un elemento (jugador, equipo o sede) respetando los criterios de ordenamiento específicos de cada árbol. Primero realiza una búsqueda comparando el nuevo nodo con los existentes según los criterios definidos. Una vez ubicado el lugar correcto, el nodo se colorea de rojo y se llama a `insertar_fixup` para restaurar las propiedades del árbol rojinegro.

3.3.2. Rotaciones (rotar_izquierda / rotar_derecha)

Operaciones fundamentales para mantener el balance del árbol. Una rotación modifica las relaciones padre-hijo entre tres nodos. La rotación izquierda convierte al hijo derecho en padre, mientras que la rotación derecha convierte al hijo izquierdo en padre.

3.3.3. Balance del Árbol (insertar_fixup)

Corrige violaciones a las propiedades del ARN después de una inserción. Usa rotaciones a izquierda o derecha dependiendo de los casos y recolorea los nodos para mantener el balance del árbol.

3.3.4. Recorrido In-Orden (recorrido_inorden)

Recorre el árbol recursivamente en orden in-orden (izquierda → nodo → derecha), almacenando los nodos en una lista. Este recorrido garantiza que los elementos se visiten en orden ascendente según los criterios de comparación definidos para cada tipo de árbol (jugadores, equipos o sedes).

3.4. Complejidad Teórica

3.4.1. Análisis por Operación

A continuacion en el Cuadro 1, se lista la complejidad de los algoritmos mencionados anteriormente.

Operación	**Complejidad**	**Justificación**
Inserción en árbol RB	O(log n)	Altura máxima del árbol es $\leq 2\log_2(n+1)$
Rotación	O(1)	Operación de punteros constante
Fixup (rebalanceo)	O(log n)	Máximo 2 rotaciones, O(log n) recoloreos
Recorrido in-orden	O(n)	Visita cada nodo exactamente una vez

Cuadro 1: Complejidad de operaciones en árbol rojinegro

3.4.2. Complejidad Total de la Solución

Sea:

- n = número total de jugadores
- k = número total de equipos

- s = número total de sedes

Inserción de jugadores en árbol global:

$$T_{\text{jugadores}} = n \cdot O(\log n) = O(n \log n)$$

Creación de equipos: Cada equipo inserta sus jugadores en su árbol local. Si m_i es el número de jugadores del equipo i:

$$T_{\text{equipos}} = \sum_{i=1}^{k} m_i \log m_i \leq n \log n = O(n \log n)$$

Inserción de equipos en árbol de equipos de cada sede:

$$T_{\text{equipos\_sede}} = k \cdot O(\log k) = O(k \log k)$$

Inserción de sedes en árbol de sedes:

$$T_{\text{sedes}} = s \cdot O(\log s) = O(s \log s)$$

Cálculo de estadísticas:

- Recorrido de todas las sedes: O(s)
- Recorrido de todos los equipos: O(k)
- Búsqueda de máximos/mínimos: O(n)
- Cálculo de promedios: O(n)

$$T_{\text{estadísticas}} = O(s + k + n) = O(n)$$

Complejidad total:

$$T_{\text{total}} = O(n \log n) + O(k \log k) + O(s \log s) + O(n)$$

Como típicamente $n \gg k \gg s$, organizamos los terminos dominantes:

$$\boxed{T_{\text{total}} = O(n \log n + k \log k + s \log s)}$$

En la configuración que se utilizó para las pruebas (caso promedio): $k \approx n/4$ (4 jugadores por equipo) y $s \approx k/2$ (2 equipos por sede):

$$T_{\text{total}} \approx O(n \log n)$$

4. Segunda Solución - Merge Sort + Insertion Sort

4.1. Idea de la Solución

La segunda solución combina dos algoritmos de ordenamiento vistos en el curso, Merge Sort para ordenar jugadores e Insertion Sort para ordenar equipos y sedes. Se implementó:

1. **Ordenar jugadores de cada equipo:** Se aplica Merge Sort a la lista de jugadores de cada equipo según los criterios.
2. **Calcular promedios y crear equipos:** Una vez ordenados los jugadores, se calcula el promedio de rendimiento de cada equipo y se almacena junto con el número de jugadores.
3. **Ordenar equipos dentro de cada sede:** Se utiliza Insertion Sort para ordenar los equipos según promedio de rendimiento.
4. **Ordenar sedes:** Se aplica Insertion Sort para ordenar las sedes según la suma de promedios.
5. **Crear ranking global de jugadores:** Se aplica Merge Sort a todos los jugadores para obtener el ranking global.

4.2. Estructuras de Datos Utilizadas

A diferencia de la primera solución que usa árboles, esta solución trabaja con estructuras de datos simples:

- **Jugadores:** Listas de diccionarios Python con claves `'id'`, `'nombre'`, `'edad'`, `'rendimiento'`.
- **Equipos:** Diccionarios con claves:
  - `'nombre'`: Nombre del deporte
  - `'promedio_rendimiento'`: Promedio calculado
  - `'cantidad_jugadores'`: Número de jugadores
  - `'jugadores'`: Lista ordenada de jugadores (diccionarios)
- **Sedes:** Diccionarios con claves:
  - `'nombre'`: Nombre de la sede
  - `'promedio_rendimiento'`: Suma de promedios de equipos
  - `'total_jugadores'`: Suma de jugadores de todos los equipos
  - `'equipos'`: Lista ordenada de equipos (diccionarios)

Esta representación basada en diccionarios y listas permite un acceso directo y simple a los datos, sin la complejidad de gestionar punteros y balanceo de árboles.

4.3. Algoritmos Implementados

4.3.1. Merge Sort para Jugadores (merge_sort_jugadores)

Ordena la lista de jugadores usando el algoritmo Merge Sort. Divide recursivamente la lista en dos mitades hasta tener sublistas de un elemento, luego las mezcla ordenadamente usando `merge_jugadores`. La comparación se realiza según los criterios del problema.

4.3.2. Mezcla de Subarreglos (merge_jugadores)

Implementa el procedimiento MERGE para combinar dos subarreglos ordenados en uno solo. Utiliza centinelas para simplificar la lógica de mezcla. En cada paso, compara los elementos de ambos subarreglos usando la función `comparar_jugadores` y selecciona el menor según los criterios del problema.

4.3.3. Insertion Sort para Equipos y Sedes (insertion_sort_equipos / insertion_sort_sedes)

Ordena la lista de equipos (o sedes) usando el algoritmo Insertion Sort. Para cada elemento desde la posición 1 hasta n-1, lo compara con los elementos anteriores y lo desplaza hacia la izquierda hasta encontrar su posición correcta. La comparación se realiza según los criterios del problema.

4.3.4. Funciones de Comparación Personalizada

- **comparar_jugadores:** Compara dos jugadores.
- **comparar_equipos:** Compara dos equipos.
- **comparar_sedes:** Compara dos sedes.

Complejidad: O(1) por comparación. Estas funciones encapsulan la lógica de desempate en múltiples niveles, simplificando los algoritmos de ordenamiento.

4.3.5. Cálculo de Estadísticas

Utiliza operaciones lineales sobre las listas ordenadas:

- **Equipo con mayor/menor rendimiento:** Recorre todas las sedes y equipos buscando el máximo y mínimo promedio.
- **Jugador con mayor/menor rendimiento:** Usa las funciones `max` y `min` de Python sobre la lista ordenada con una función clave personalizada.
- **Jugador más joven/veterano:** Similar al anterior, usando `min` y `max` con criterio de edad.
- **Promedios:** Suma todos los valores y divide entre el total de elementos.

4.4. Complejidad Teórica

4.4.1. Análisis de Merge Sort

Para un arreglo de tamaño n:
Relación de recurrencia:

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 \\ 2T(n/2) + \Theta(n) & \text{si } n > 1 \end{cases}$$

Aplicando el Teorema Maestro:

- $a = 2$ (dos subproblemas)
- $b = 2$ (tamaño dividido por 2)
- $f(n) = \Theta(n)$ (costo del merge)
- $\log_b a = \log_2 2 = 1$

Como $f(n) = \Theta(n) = \Theta(n^{\log_b a})$, aplicamos el Caso 2:

$$\boxed{T_{\text{merge}}(n) = \Theta(n \log n)}$$

4.4.2. Análisis de Insertion Sort

Para un arreglo de tamaño k:
Mejor caso (ya ordenado): O(k)
Peor caso (orden inverso): O(k^2)

$$\boxed{T_{\text{insertion}}(k) = O(k^2)}$$

4.4.3. Complejidad Total de la Solución

Sea:

- n = número total de jugadores
- k = número total de equipos
- s = número total de sedes

Ordenamiento de jugadores por equipo: Cada equipo ordena sus jugadores usando Merge Sort. Si el equipo i tiene m_i jugadores, ordenarlo cuesta $O(m_i \log m_i)$. Como hay k equipos y la suma de todos los jugadores es n:

$$T_{\text{jugadores\_equipo}} = \sum_{i=1}^{k} m_i \log m_i \leq n \log n = O(n \log n)$$

La desigualdad se cumple porque $\sum m_i = n$ y $\log m_i \leq \log n$.

Ordenamiento de equipos por sede: Cada sede ordena sus equipos usando Insertion Sort. Si la sede j tiene e_j equipos, ordenarlos cuesta $O(e_j^2)$ en el peor caso. Como hay s sedes y la suma de todos los equipos es k:

$$T_{\text{equipos}} = \sum_{j=1}^{s} e_j^2 \leq k^2 = O(k^2)$$

Ordenamiento de sedes: Se ordenan las s sedes usando Insertion Sort:

$$T_{\text{sedes}} = O(s^2)$$

Ranking global de jugadores: Se aplica Merge Sort a todos los n jugadores:

$$T_{\text{ranking}} = O(n \log n)$$

Cálculo de estadísticas: Se recorren linealmente sedes (s), equipos (k) y jugadores (n) para calcular máximos, mínimos y promedios:

$$T_{\text{estadísticas}} = O(s + k + n) = O(n)$$

Como típicamente $n \gg k \gg s$, el término dominante es $O(n)$.

Complejidad total:

$$T_{\text{total}} = O(n \log n) + O(n \log n) + O(k^2) + O(s^2) + O(n)$$

Simplificando términos dominantes:

$$\boxed{T_{\text{total}} = O(n \log n + k^2 + s^2)}$$

El término k^2 proviene del Insertion Sort aplicado a k equipos. Aunque $k < n$, este término puede dominar para valores grandes debido a que:

- En la configuración que se utilizó para las pruebas: $k \approx n/4$ (4 jugadores por equipo)
- Sustituyendo: $k^2 = (n/4)^2 = n^2/16$
- Comparando: $n \log n$ vs $n^2/16$
  - Para $n = 100$: 664 vs 625 $\rightarrow$ Similar
  - Para $n = 1000$: $9,966$ vs $62,500$ $\rightarrow$ Cuadrático domina
  - Para $n = 10,000$: $132,877$ vs $6,250,000$ $\rightarrow$ Cuadrático domina claramente

Conclusión de complejidad: Aunque formalmente la complejidad es $O(n \log n + k^2)$, cuando k es proporcional a n, esto no significa que necesariamente el algoritmo sea cuadrático en el peor caso teórico, sino que en la configuración práctica del problema, el término k^2 se convierte en el factor dominante para el tiempo de ejecución con datasets grandes.

5. Análisis de Resultados

El análisis de las soluciones implementadas se divide en dos etapas complementarias:

- **Validación de correctitud:** Ambas soluciones fueron probadas con los 4 archivos de entrada proporcionados con el proyecto (input0.py, input1.py, input2.py, input3.py), cada uno con diferentes configuraciones de jugadores, equipos y sedes. Los resultados generados por ambas soluciones son idénticos, confirmando que ambas implementan correctamente los criterios de ordenamiento y el cálculo de estadísticas especificados en el enunciado. Estos archivos de salida se encuentran en el proyecto en las carpetas `First Solution/outputs/` y `Second Solution/outputs/`.

- **Análisis de complejidad temporal:** Por otro lado, para evaluar el desempeño y validar la complejidad teórica, generé instancias de prueba con tamaños crecientes (desde 10 hasta 100,000 jugadores). Estas instancias se generan de manera completamente aleatoria, utilizando una semilla (seed) fija para controlar la aleatoriedad y garantizar que los resultados sean siempre los mismos en cada ejecución.

  Los datos generados incluyen:

  - **Jugadores:** Cada jugador tiene una edad aleatoria en el rango [20, 49] y un rendimiento aleatorio en el rango [10, 99]. La aleatoriedad asegura que los datos estén completamente desordenados y representen un caso general.
  - **Equipos:** Los jugadores se agrupan en equipos de aproximadamente 4 jugadores cada uno, asignados de manera aleatoria.
  - **Sedes:** Cada sede tiene entre 2 y 3 equipos, asignados de manera aleatoria para garantizar una distribución no uniforme.

La generación aleatoria de datos permite evaluar el desempeño de ambas soluciones en escenarios generales y reproducibles, eliminando cualquier sesgo introducido por patrones específicos. Los resultados se presentan a continuación.

5.1. Metodología de Pruebas

Se generaron 12 casos de prueba con la siguiente distribución:

- Número de jugadores (n): 10, 20, 50, 100, 200, 500, 1000, 2000, 5000, 10000, 20000, 100000
- Número de equipos (k): Aproximadamente $n/4$ (4 jugadores por equipo)
- Número de sedes (s): Aproximadamente $k/2$ (2-3 equipos por sede)

Cada medición representa el promedio de 5 ejecuciones para reducir la variabilidad por factores externos del sistema operativo. Se midió el tiempo completo de ejecución incluyendo:

- Ordenamiento de jugadores, equipos y sedes
- Cálculo de todas las estadísticas requeridas

5.2. Resultados Experimentales

En el Tabla 2 se presentan los tiempos promedio de ejecución (en milisegundos) para ambas soluciones, medidos en los 12 casos.

n	Primera (ms)	Segunda (ms)	Diferencia (%)
10	0.057	0.052	-8.7
20	0.116	0.105	-9.5
50	0.240	0.260	+8.5
100	0.597	0.577	-3.4
200	1.318	1.148	-12.9
500	2.712	3.153	+16.3
1000	5.461	6.897	+26.3
2000	11.256	16.300	+44.8
5000	32.178	51.542	+60.2
10000	70.245	159.005	+126.3
20000	172.767	469.681	+172.0
100000	1400.200	9893.106	+606.5

Cuadro 2: Tiempos de ejecución promedio (5 ejecuciones). Diferencia positiva indica que la primera solución es más rápida.

- Para $n \leq 200$, las diferencias entre ambas soluciones son pequeñas (menos del 10 %), ya que el término k^2 de la segunda solución aún no domina.
- A partir de $n = 500$, la primera solución comienza a ser consistentemente más rápida, con una diferencia que crece exponencialmente.
- En $n = 100,000$, la primera solución es alrededor de 6 veces más rápida que la segunda, lo que valida la predicción de que $O(k^2)$ domina para valores grandes de k.

5.3. Comparación Complejidad Teórica vs Real

En la Figura 1 se presentan las gráficas de complejidad teórica y tiempos reales para ambas soluciones. Cada línea tiene un significado específico:

- **Línea azul o roja (tiempo real):** Representa los tiempos medidos experimentalmente para cada tamaño de entrada, en cada solucion respectivamente.
- **Línea verde punteada ($O(n \log n)$):** En el caso de la segunda solucion es la cota inferior teorica, correspondiente al término de Merge Sort.
- **Línea naranja punteada ($O(n \log n + k^2)$):** Representa la cota superior teórica de la segunda solución, que incluye el término cuadrático de Insertion Sort.

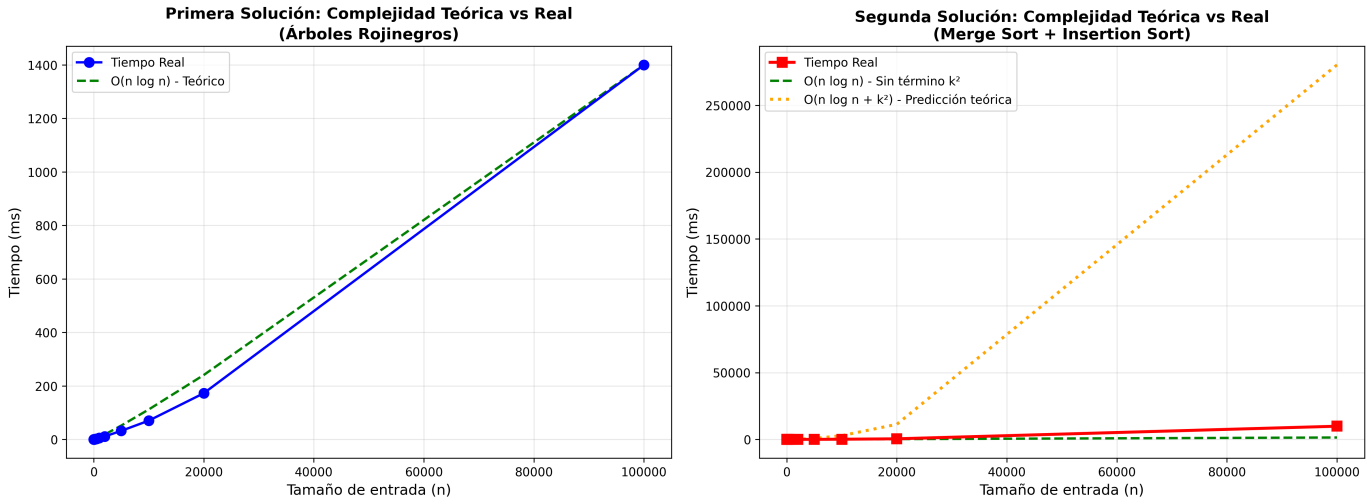

Figura 1: Validación de complejidad teórica. Izquierda: Primera solución sigue O(n log n). Derecha: Segunda solución muestra efecto de O(k^2) para n grande.

Interpretación de las gráficas:

- Para la grafica de la primer solución, el tiempo real se mantiene bastante cercano al tiempo teorico.
- Para la grafica de la segunda solución, el tiempo real está entre ambas cotas, mientras que el tamaño de los casos probados aumenta, se aleja mas de la cota inferior y se acerca a la cota superior, validando que el término k^2 domina poco a poco para valores grandes de n. Aunque cabe aclarar, que la cota superior es en un caso extremo, por eso es que tambien sigue muy lejos de el, tengo la hipotesis de que en un tamaño de entrada mucho más grande de 100.000 se acerca mucho más.

5.4. Gráficos de Desempeño

5.4.1. Comparación Directa de Tiempos

En la Figura 2 se comparan los tiempos de ejecución de ambas soluciones. Cada línea tiene un significado específico:

- **Línea azul (Primera solución):** Representa los tiempos medidos experimentalmente para la solución basada en árboles rojinegros.
- **Línea roja (Segunda solución):** Representa los tiempos medidos experimentalmente para la solución basada en Merge Sort e Insertion Sort.

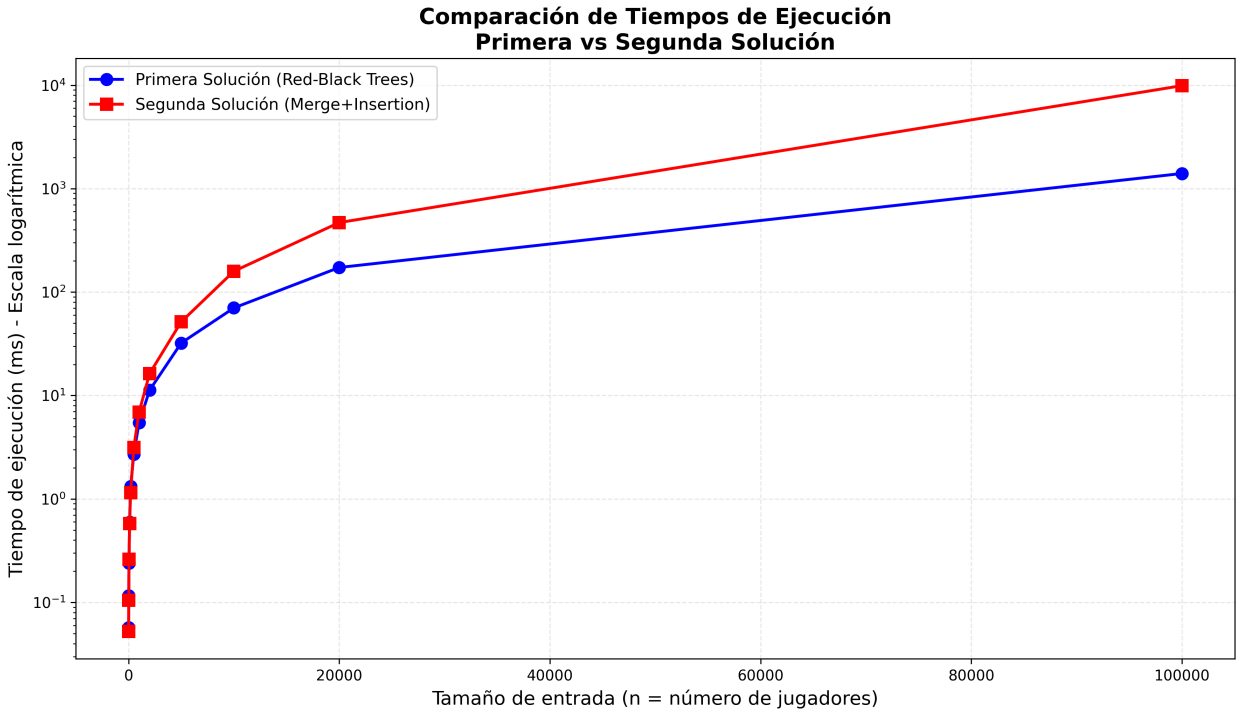

Figura 2: Evolución de tiempos de ejecución con el tamaño de entrada.

- La línea roja está consistentemente por encima de la línea azul (primera solución, Árboles Rojinegros) para valores grandes de n. Esto se debe a que la segunda solución incluye un término cuadrático $O(k^2)$, donde k es proporcional a n, lo que hace que su tiempo de ejecución crezca más rápido que el término logarítmico $O(n \log n)$ de la primera solución.

- Para valores pequeños de n (por ejemplo, $n \leq 200$), ambas soluciones tienen tiempos similares, aunque no se aprecia del todo en la grafica, a partir de $n = 500$, la diferencia entre las soluciones se vuelve más evidente.

- La escala logarítmica permite observar tanto los valores pequeños como los grandes de manera proporcional, destacando cómo la segunda solución se vuelve menos eficiente conforme aumenta el tamaño de entrada.

5.4.2. Diferencia Porcentual

Adicionalmente se creó una grafica para representar la diferencia porcentual, entre 10 a 200 para valores de n, la diferencia entre ambas soluciones es pequeña, pero a partir de 500 la segunda solución toma mucho más protagonismo.

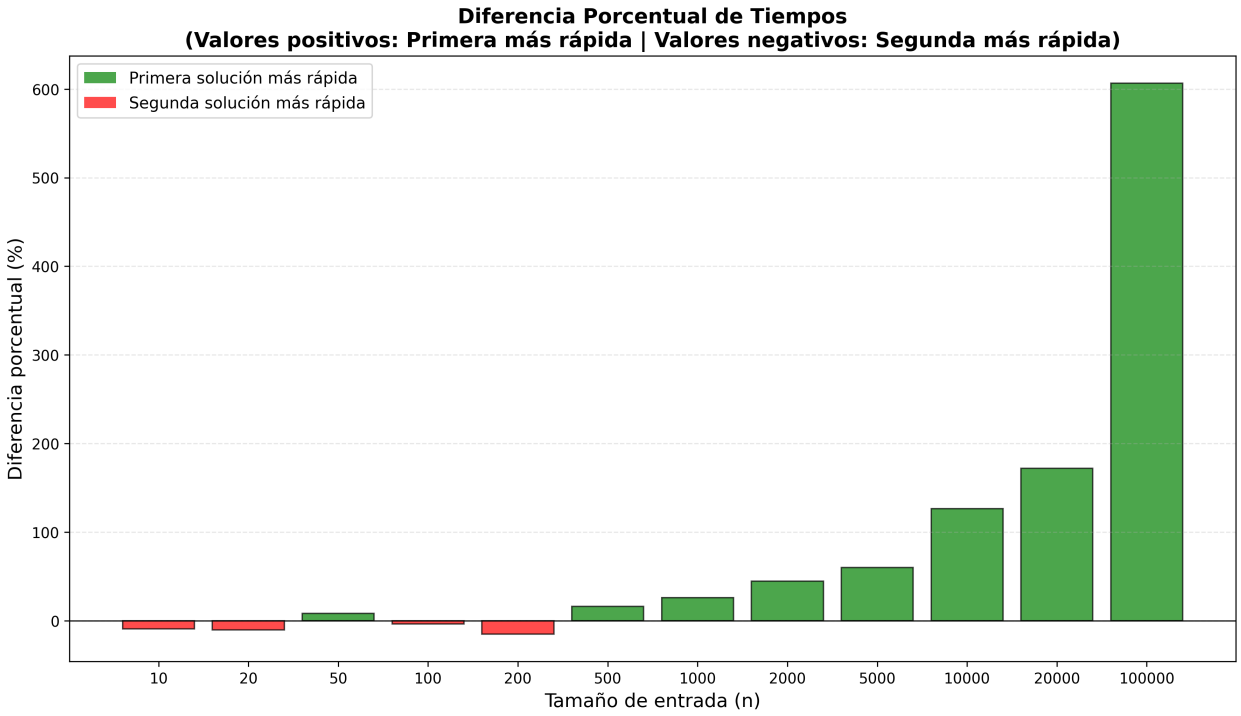

Figura 3: Diferencia porcentual entre ambas soluciones. Verde: primera más rápida. Rojo: segunda más rápida.

5.5. Comparación Entre Soluciones

En el Cuadro 3 se resumen las diferencias clave entre ambas soluciones:

Criterio	**Primer Solución**	**Segunda Solución**
Tiempo (n=10)	0.057 ms	0.052 ms
Tiempo (n=1000)	5.461 ms	6.897 ms
Tiempo (n=10000)	70.245 ms	159.005 ms
Tiempo (n=100000)	1400.200 ms	9893.106 ms
Complejidad	$O(n \log n)$	$O(n \log n + k^2)$
Escalabilidad	Excelente	Limitada
Dificultad de implementar en código	Algo compleja	Más simple
Se comporta mejor con	Cualquier valor de n	$n < 500$

Cuadro 3: Resumen comparativo de ambas soluciones.

6. Conclusiones

6.1. Logros del Proyecto

1. **Implementación exitosa de dos soluciones algorítmicas:** Se desarrollaron dos enfoques completamente diferentes que resuelven el mismo problema de ordenamiento jerárquico, cumpliendo con todos los requisitos del enunciado.

2. **Validación de la complejidad teórica:** Los experimentos con 12 tamaños de entrada (10 a 100,000 jugadores) confirmaron que:

   - La primera solución (árboles rojinegros) se comporta como $O(n \log n)$.
   - La segunda solución (Merge + Insertion Sort) evidencia el término $O(k^2)$ para valores grandes de k.

3. **Correctitud demostrada:** Ambas soluciones generan outputs idénticos para los 4 casos de prueba proporcionados, validando la correcta implementación de los algoritmos y criterios de ordenamiento.

6.2. Lecciones Aprendidas

El desarrollo de este proyecto permitió explorar y comparar dos enfoques completamente diferentes para resolver el mismo problema algorítmico: uno basado en estructuras complejas como los árboles rojinegros, y otro apoyado en listas y algoritmos como Merge Sort e Insertion Sort. A través de la implementación, análisis y pruebas de rendimiento, se evidenció cómo la elección de estructuras de datos y algoritmos adecuados puede impactar de manera significativa en la eficiencia y escalabilidad del código.

Durante la implementación de ambas soluciones, se observó que el uso de árboles rojinegros resulta muy útil cuando se trabaja con grandes cantidades de datos y se requieren inserciones o búsquedas eficientes. Sin embargo, esta estructura supone un mayor nivel de complejidad tanto en su diseño como en su mantenimiento. Por otro lado, la solución basada en listas y algoritmos clásicos es más sencilla de implementar, facilita el manejo inicial de los datos y puede ser más eficiente en conjuntos pequeños, pero no escala igual de bien ante grandes volúmenes de información.

Finalmente, el análisis experimental validó la teoría estudiada en el curso. Ver cómo los tiempos reales seguían las curvas teóricas de $O(n \log n)$ y $O(n \log n + k^2)$ no solo confirmó que los conceptos del curso funcionan en la práctica, sino que también demostró la importancia de realizar mediciones reales antes de elegir una solución definitiva para un problema.

7. Anexos

- Instrucciones de ejecución en el archivo README.md
- Repositorio: `https://github.com/JuanCortesRo/ADA-1_Project_2025-2.git`
- Video: `https://drive.google.com/file/d/1mKI3NIIF5jIcMaMZu151-SogGiPWolps/view?usp=sharing`