Mini Proyecto I

Alejandro Guerrero - 2179652

Juan Pablo Ante - 2140132

Alejandro Zamorano - 1941088

Wilson Andres Mosquera - 2182116

Jesús Alexander Aranda Bueno

Cali - Valle del Cauca, 2024

Analisis y Diseño de Algoritmos I

0.1. Introducción

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, La asociación contará con K sedes, cada una de las cuales tendrá un nombre, además, cada sede tendrá M equipos, donde cada equipo tendrá, a su vez, un deporte definido, cada equipo puede contar con una cantidad minima $Nmin$ y máxima $Nmax$ de jugadores, cada jugador tendrá un número como identificador, su nombre, su edad y su rendimiento, este último tomará valores de 1 a 100, ϵ N.

K, M, y $Nmax$ son enteros positivos, además $Nmin < Nmax$.

La asociación busca que los equipos internamente estén ordenados ascendentemente teniendo en cuenta el rendimiento de los jugadores, en caso de empate se colocará primero el jugador de mayor edad. También para cada sede, se busca que los equipos estén ordenados ascendentemente por su rendimiento promedio, el cual se define como la suma de los rendimientos de los jugadores del equipo sobre la cantidad de jugadores del equipo ($\frac{SumaRendimientoJugadores}{NumeroDeJugadores}$), en caso de empate en el valor del rendimiento entre equipos, se pondrá primero el equipo que tiene mayor cantidad de jugadores. Para las sedes se busca algo similar, se requiere ordenar las sedes de forma

ascendente según el promedio de los rendimientos de todos los equipos de la respectiva sede, en caso de que dos sedes tengan el mismo rendimiento promedio la sede con mas jugadores se pondrá primero. Por último con el fin de saber el ranking de los jugadores para tomar decisiones respecto a esto, se requiere también generar la lista de todos los jugadores de todas las sedes ordenados por su rendimiento ascendentemente.

Además de esto la asociación, también necesita obtener algunos datos como:

- Equipo con mayor rendimiento.
- Equipo con menor rendimiento.
- Jugador con mayor rendimiento.
- Jugador con menor rendimiento.
- Jugador mas joven.
- Jugador mas veterano.
- Promedio de edad de los jugadores.
- Promedio del rendimiento de los jugadores

La finalidad de este proyecto era construir dos soluciones al problema dado utilizando diferentes estructuras de datos para almacenar los datos y distintos algoritmos para manipular estos.

0.2. Soluciones

0.2.1. Primera solución

La primera solución utiliza una tabla hash (datos.py) como estructura de datos para organizar y almacenar información de jugadores y sedes de manera eficiente. Las tablas hash fueron escogidas por su capacidad para proporcionar acceso rápido a los datos mediante la operación de búsqueda ademas de que su complejidad promedio es $O(1)$ la cual la hace aun mas eficiente. Además, se implementó el algoritmo quicksort (*quicksort.py*), conocido por su eficiencia y velocidad en la ordenación de datos, con una complejidad promedio de $O(nlogn)$ lo que lo hace adecuado para ordenar listas grandes.

0.2.2. Segunda solución

Para la segunda solución se ha optado por utilizar clases para modelar jugadores, equipos y sedes, lo cual proporciona una estructura más orientada a objetos que facilita la manipulación y organización de los datos *(datos-test.py)*. Además, se implementan algoritmos como *mergesort* y estructuras como árboles binarios de búsqueda para realizar operaciones eficientes de procesamiento y análisis de datos dentro del contexto del problema planteado *(mergesort.py)*. *Mergesort* fue elegido por su estabilidad y su rendimiento consistente de $O(nlogn)$, muy eficiente cuando se requiere una ordenación de grandes conjuntos de datos. Los árboles binarios de búsqueda permiten una gestión eficiente de los datos, facilitando operaciones rápidas de inserción, búsqueda y eliminación con una complejidad promedio de $O(logn)$, lo que es esencial para modificar y gestionar grandes listas.

0.3. Comparación

Se realizó una serie de pruebas para comparar el rendimiento de MergeSort y QuickSort. El objetivo de estas pruebas fue evaluar el tiempo de ejecución de cada algoritmo. Las pruebas se llevaron a cabo utilizando listas de jugadores con tamaños de 12, 22, 32 y 42 elementos.

Jugadores	**Tiempo (s)**
12	0.0020112991333007812
22	0.002378225326538086
32	0.01622748374938965
42	0.01799941062927246

Cuadro 1: Tiempos de ejecución para MergeSort

Jugadores	**Tiempo (s)**
12	0.001999378204345703
22	0.0030002593994140625
32	0.0030517578125
42	0.003133058547973633

Cuadro 2: Tiempos de ejecución para QuickSort

0.3.1. Orden de complejidad

La primera solución tiene una complejidad teórica (O ($nlogn$)), en cambio la segunda solución cuenta con una complejidad teórica (O ($nlogn$)). Con base a lo anterior se podría considerar la segunda solución ya que siempre conserva esta complejidad, lo cual garantiza una ejecución más rápida y óptima en comparación con la primera solución, especialmente para conjuntos de datos grandes.

0.3.2. Análisis de resultados

Se realizaron pruebas con diferentes configuraciones de entrada, variando el número de sedes *(m)*, equipos *(k)*, y el número de jugadores por equipo *(n)*. El análisis se centró en el tiempo de ejecución del algoritmo en función de estos parámetros.

Análisis de QuickSort

QuickSort mantuvo tiempos de ejecución muy consistentes incluso cuando el tamaño de la lista aumentó. El tiempo de ejecución fue casi lineal, incrementando de 0.002 segundos para 12 jugadores a aproximadamente 0.003 segundos para 42 jugadores.

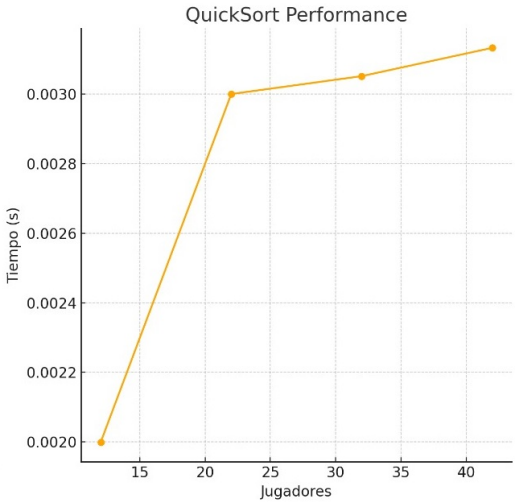

Figura 1: Comportamiento QuickSort

Análisis del MergeSort

Para listas pequeñas (12 y 22 jugadores), MergeSort mostró tiempos de ejecución muy bajos, cercanos a los de QuickSort. Sin embargo, a medida que el tamaño de la lista aumentó a 32 y 42 jugadores, el tiempo de ejecución de MergeSort incrementó significativamente, alcanzando 0.016 segundos para 32 jugadores y 0.018 segundos para 42 jugadores.

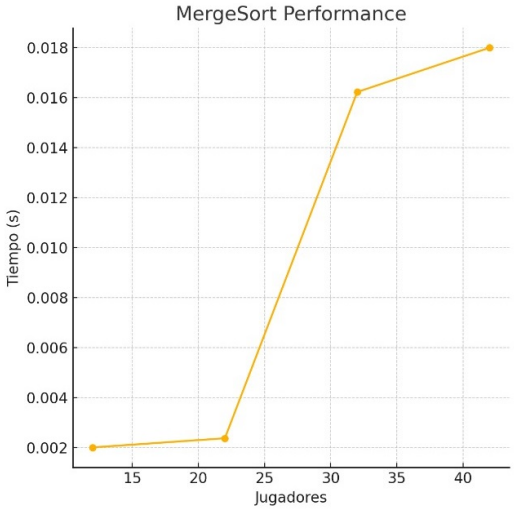

Figura 2: Comportamiento MergeSort

MergeSort es eficiente para listas pequeñas, pero su tiempo de ejecución aumenta de manera más notable con listas más grandes. Esto podría ser debido a la sobrecarga de la división y combinación en cada nivel de recursión. QuickSort demostró ser más consistente y escalable, manejando listas más grandes con incrementos mínimos en el tiempo de ejecución. Lo que indica que QuickSort podría ser más adecuado para aplicaciones que requieren el ordenamiento de listas grandes de manera eficiente.

0.3.3. Pruebas Grandes

Para validar las pruebas realizadas, se incrementaron significativamente los parámetros con valores mucho más altos para evaluar los tiempos de ejecución con mayor precisión. En esta ocasión, se seleccionó el algoritmo mergeSort para llevar a cabo estas pruebas.

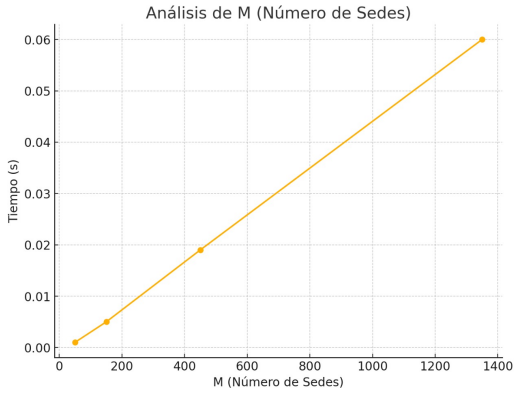

Figura 3: Parámetro Sedes

Sedes

M (Número de Sedes)	**Tiempo (s)**
50	0.00099945068359375
150	0.004998922348022461
450	0.018999099731445312
1350	0.059999942779541016

Cuadro 3: Análisis de M (Número de Sedes)

El tiempo de ejecución aumenta de manera significativa con el número de sedes, lo que indica una relación directa entre la cantidad de sedes y el tiempo requerido para procesar la información.

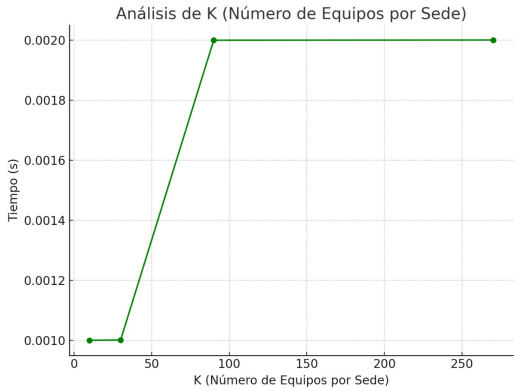

Figura 4: Parámetro Equipo

Equipos

K (Número de Equipos por Sede)	**Tiempo (s)**
10	0.0009999275207519531
30	0.0010006427764892578
90	0.0019991397857666016
270	0.0019998550415039062

Cuadro 4: Análisis de K (Número de Equipos por Sede)

El tiempo de ejecución permanece casi constante a pesar del incremento en el número de equipos por sede. Esto muestra que el algoritmo es eficiente en manejar múltiples equipos dentro de cada sede sin un costo significativo en tiempo.

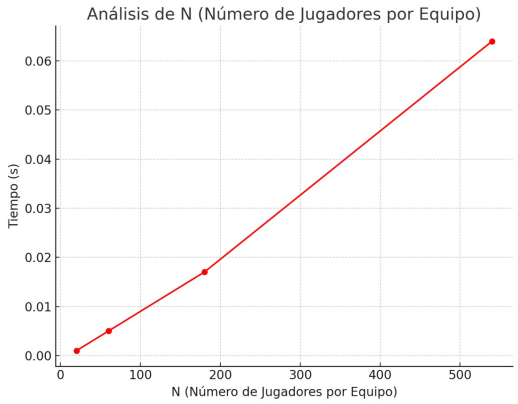

Figura 5: Parámetro Jugadores

N (Número de Jugadores por Equipo)	**Tiempo (s)**
20	0.0010001659393310547
60	0.0049991607666015625
180	0.01699995994567871
540	0.06400036811828613

Cuadro 5: Análisis de N (Número de Jugadores por Equipo)

Jugadores

Hay un incremento significativo en el tiempo de ejecución a medida que aumenta el número de jugadores por equipo. Esto muestra que la cantidad de jugadores tiene un impacto considerable en el rendimiento del algoritmo.

0.4. Repositorio

Puedes encontrar el código fuente y las instrucciones para ejecutar el proyecto en el siguiente repositorio:

`https://github.com/JuanPabloAnteSuarez03/ADAProject`

0.5. Conclusión

Con base a lo expuesto anteriormente, las dos soluciones son muy óptimas para resolver el problema de la asociación de deportes. Sin embargo, al momento de escoger, se debería considerar la segunda solución como la opción preferida. Esto se debe a su complejidad teórica (O $(nlogn)$), que garantiza tiempos de ejecución rápidos y predecibles sin importar el tamaño de los datos de entrada. Por otro lado, la primera solución, aunque eficiente con una complejidad (O $(nlogn)$) igual a la otra solución pero este gracias al quicksort maneja un peor caso de (O (n^2)), además podría experimentar variaciones en el tiempo de ejecución conforme aumente el tamaño de la entrada.

Así, la elección de la segunda solución no solo asegura eficiencia en el procesamiento de la información deportiva y análisis organizacional, sino también una gestión más robusta y estable del sistema.