# Marco Teórico - Trabajo Integrador

1. Introducción

Este trabajo se centra en el análisis y la comparación de distintos algoritmos clásicos de búsqueda y ordenamiento, aplicados sobre estructuras de datos avanzadas como árboles binarios y AVL, utilizando el lenguaje de programación Python. La elección de este tema se basa en su importancia dentro del campo de la programación, ya que estas técnicas son fundamentales para optimizar el rendimiento de aplicaciones que manejan grandes volúmenes de información.

El objetivo general es llevar a cabo una implementación funcional y documentada de estos algoritmos, evaluando su eficiencia a través de pruebas prácticas. Esto permite no solo comprender su funcionamiento teórico, sino también observar su comportamiento real en distintos escenarios. La relevancia del tema radica en que el manejo eficiente de datos es una habilidad clave para cualquier desarrollador de software.

2. Algoritmos de Ordenamiento

2.1 BubbleSort

Algoritmo simple que compara elementos adyacentes y los intercambia si están en orden incorrecto. Es ineficiente para grandes volúmenes de datos (O(n²)). Se implementa fácilmente y es útil para enseñar conceptos básicos de ordenamiento.

2.2 MergeSort

Algoritmo recursivo basado en el paradigma “divide y vencerás”. Divide la lista en mitades, ordena cada una y las combina. Su eficiencia es O(n log n) y es muy estable, aunque requiere espacio adicional.

2.3 QuickSort

También divide y conquista, eligiendo un pivote para dividir los datos. En promedio es muy rápido (O(n log n)), aunque su peor caso es O(n²). Es eficiente y muy usado en la práctica.

2.4 Comparación Teórica

A continuación, se comparan los tres algoritmos mencionados según su eficiencia temporal y consumo de espacio:



3. Algoritmos de Búsqueda

3.1 Búsqueda lineal

Recorre cada elemento hasta encontrar el valor deseado. Muy ineficiente (O(n)) pero útil en listas pequeñas o no ordenadas.

3.2 Búsqueda binaria

Divide la lista ordenada a la mitad en cada paso. Mucho más eficiente: O(log n). No sirve si la lista no está ordenada.

3.3 BFS y DFS en árboles

Algoritmos de recorrido. BFS (Breadth-First Search) usa una cola para recorrer por niveles, y DFS (Depth-First Search) usa una pila (o recursión) para recorrer en profundidad. Se aplican en árboles, grafos y mapas.

4. Estructuras de Datos Avanzadas

4.1 Árbol Binario

Estructura jerárquica donde cada nodo tiene hasta dos hijos. Se usa para organizar datos que requieren búsquedas rápidas.

4.2 Árbol AVL

Variante de árbol binario que se autobalancea para mantener operaciones eficientes. Ideal para mantener eficiencia O(log n) en búsqueda e inserción.

4.3 Aplicaciones en la vida real

- Bases de datos (índices)

- Compiladores (árboles de sintaxis)

- Motores de búsqueda (ranking)

- Juegos (árboles de decisión)

5. Análisis de Eficiencia

5.1 Notación Big-O

Mide el comportamiento de un algoritmo en función del tamaño de entrada. Sirve para comparar eficiencia teórica.

5.2 Métricas de rendimiento (tiempo y memoria)

En la práctica usamos herramientas como time.time() y timeit para medir velocidad, y memory\_profiler para consumo de RAM. Estos valores varían según el entorno, pero dan una idea clara del rendimiento.