PROYECTO DEL CURSO: ESTRATEGIAS PARA PROCESAR INFORMACIÓN DE ENCUESTAS.

INTEGRANTES:
JUAN MANUEL PEREA CORONADO - 1926462
YENNY MARGOT RIVAS TELLO - 2182527
DAVID ALEJANDRO ENCISO - 2240581
JEAN PAUL DAVALOS - 1832375

PRESENTADO A:
PhD. JESÚS ALEXANDER ARANDA BUENO

TRABAJO PRESENTADO PARA CUMPLIR LOGROS DE QUINTO SEMESTRE.

UNIVERSIDAD DEL VALLE
FACULTAD DE INGENIERÍA
ESCUELA DE INGENIERÍA DE SISTEMAS Y COMPUTACIÓN
DICIEMBRE 2024

ÍNDICE.

INTRODUCCIÓN.

El objetivo de este informe consiste en brindar asistencia a una consultora de datos que busca analizar los resultados de una encuesta estructurada en varios temas para identificar cuáles tienen opiniones más favorables y responder preguntas relacionadas.

Cada encuestado tiene un identificador único, nombre, nivel de experticia (valor entero entre 0 y 10), y su opinión sobre la pregunta (valor entero entre 0 y 10, donde 0 es completamente desfavorable y 10 es completamente favorable). La información debe organizarse y ordenarse según varios criterios:

Los encuestados en cada pregunta deben estar ordenados descendentemente por su opinión y, en caso de empate, por su nivel de experticia. Las preguntas en cada tema deben estar ordenadas según el promedio de las opiniones, priorizando el promedio de experticia y el número de encuestados en caso de empate. Los temas deben ordenarse de acuerdo con el promedio de los promedios de las opiniones de sus preguntas, considerando el promedio de experticia y el número de encuestados en caso de empate. Finalmente, se debe generar una lista general de encuestados, ordenada descendentemente por nivel de experticia y, en caso de empate, por su identificador.

Para llevar a cabo esta asesoría, se plantean dos soluciones al problema, utilizando diferentes estructuras de datos y algoritmos predefinidos para el procesamiento y ordenamiento.

Una de las soluciones implementadas es **Listas de Lista**, ya que permite estructurar y manejar los datos jerárquicos del enunciado de una manera eficiente y flexible, también permite ordenar los datos en varios niveles donde cada nivel se puede ordenar independientemente utilizando algoritmos personalizados. Estas también permiten facilitar el acceso a cualquier parte de los datos sin necesidad de crear estructuras adicionales. El uso de **listas de listas** es una forma práctica y eficiente de manejar datos jerárquicos como los del problema. Permiten organizar, ordenar y procesar los datos de manera directa, manteniendo la jerarquía inherente al problema. Además, facilitan tanto la entrada como la salida de datos, y son suficientes para resolver el problema sin necesidad de estructuras más complejas.

Por otra parte, decidimos la implementación de **árboles binarios de búsqueda** que permiten mantener los datos ordenados de forma eficiente durante su inserción. Dado que el problema requiere ordenar encuestados, preguntas y temas según diferentes criterios (opinión, experticia, cantidad, etc.), un **ABB** puede reducir significativamente el costo computacional al combinar almacenamiento y ordenamiento en una sola estructura.

La ventaja de esta implementación radica en que La inserción y la búsqueda en un ABB tienen un costo promedio de O(logn) en un árbol balanceado, en lugar de ordenar toda una lista repetidamente, que costaría O(nlogn).

SOLUCIONES.

Para resolver el problema de análisis y procesamiento de los datos de la gran encuesta propuesto por la empresa de consultoría de datos, se ha diseñado e implementado dos soluciones o estrategias utilizando estructuras de datos diferentes: listas y árboles binarios.

La primera estrategia, basada en listas, aprovecha su simplicidad y flexibilidad para ordenar y gestionar los datos de manera directa, implementando algoritmos como el **Merge Sort** y así garantizar un manejo eficiente de las operaciones de ordenamiento.

Por otro lado, la segunda estrategia emplea árboles binarios, lo que permite una organización jerárquica de los datos y optimiza la inserción y ordenamiento al reducir la complejidad computacional en comparación con métodos más lineales (Como el caso de las listas).

Estas soluciones no solo ofrecen perspectivas complementarias para resolver el problema, sino que también permiten una comparación detallada de sus rendimientos en términos de tiempo de ejecución y eficiencia en el manejo de datos.

Estrategia 1: Listas.

En esta primera estrategia se propone el uso de listas como principal estructura de datos para gestionar y procesar la información de la gran encuesta. Las listas permiten almacenar y organizar los datos de manera secuencial, facilitando su interacción con ellas. Para esta solución, se desarrolló una implementación del **Merge Sort** para ordenar los encuestados, las preguntas y los temas según los criterios establecidos por la empresa de consultoría de datos. Esta solución también incluye una evaluación detallada de su complejidad computacional y su desempeño en diferentes escenarios de prueba (Distintos tamaños en los datos).

Idea de la Propuesta.

La elección de las listas como estructura de datos principal está relacionada a la flexibilidad que estas ofrecen para almacenar y manipular datos de forma secuencial, facilitando su gestión en elementos como los encuestados, las preguntas y los temas de manera clara. A través de la implementación de listas, se busca ordenar los datos según los criterios establecidos por la empresa de consultoría de datos; como el ordenamiento descendente por opinión y/o nivel de

experticia. Además ofrece una implementación que prioriza la simplicidad y la legibilidad del código.

En el diseño del código se hizo uso de clases para representar entidades clave del problema, como son los encuestados, las preguntas, los temas, y la encuesta como tal, facilitando un modelo lógico y bien estructurado para manejar la información.

Para hacer las distintas tareas de ordenamiento dentro del sistema, se implementó el algoritmo de ordenamiento **Merge Sort**, que tiene una complejidad de tiempo $T(n) = O(nlog(n))$, ideal para manejar volúmenes grandes de datos. La implementación se realizó usando de base el algoritmo visto en clase donde se dividen las listas en sublistas más pequeñas, para luego ordenarlas y finalmente combinarlas de nuevo para obtener la lista final ordenada según algún criterio previamente definido. Gracias al uso del **Merge Sort**, la solución garantiza un buen rendimiento en cualquier tipo de escenarios sin importar el tamaño de los datos.

Aunque usar listas es adecuado para procesar conjuntos de datos pequeños, medianos o prudentemente grandes, su rendimiento podría verse afectado con volúmenes de datos exageradamente grandes debido a las limitaciones que podrían existir en memoria. Esto se debe a que el **Merge Sort**, aunque es eficiente en cuanto a complejidad temporal, requiere espacio adicional para dividir y combinar las listas.

A pesar de lo anterior, el uso de listas permite un código legible, modular y fácil de mantener, lo que lo hace una solución viable para escenarios en los que el manejo de datos no exceda los límites de memoria disponibles. Por último, esta implementación servirá como base para comparar el desempeño con la segunda estrategia basada en árboles binarios.

Estructura de Datos Usada.

La estrategia basada en el uso de listas como estructura de datos principal para ofrecer una solución al problema de ordenamiento propuesto por la empresa de consultoría de datos, se desarrolló empleando el paradigma de programación orientado a objetos (POO) en Python. El usar listas permite una estructura para almacenar y organizar entidades como los encuestados, las preguntas, los temas y la encuesta como tal. Hacerlo de esta manera, facilita el acceso y el manejo de los datos a través de los métodos definidos en cada una de las clases, acelerando la ejecución de las operaciones necesarias para cumplir los requerimientos del proyecto y garantizando un código limpio y bien estructurado.

En esta solución, cada entidad del problema se representa mediante clases:

- La clase **Encuestado** modela a los participantes de la encuesta, almacenando atributos como su identificador, nombre, nivel de experticia y opinión. También incluye métodos

como “__init__” para inicializar sus atributos y “__rep__” para facilitar su representación en formato de texto y retornando, una lista con los valores de los atributos.

La clase **Pregunta** agrupa a los encuestados relacionados con una pregunta específica, manejando métodos para ejecutar operaciones como agregar encuestados a la pregunta, calcular el promedio de las opiniones obtenidas de responder la pregunta, calcular el promedio de experticia de los encuestados que respondieron la pregunta y ordenar la lista de encuestados mediante la implementación del **Merge Sort**.

La clase **Tema** agrupa las preguntas asociadas a un tema específico, y maneja métodos muy similares a los de clase Pregunta, estos son para ejecutar operaciones relacionadas a agregar preguntas al tema, calcular el promedio de opiniones y de experticia por pregunta y ordenar las preguntas implementando el **Merge Sort**.

La clase **Encuesta** sintetiza toda los elementos creados en las anteriores clases, con métodos que permiten agregar temas a la encuesta, generar listados entre todos los encuestados totales, calcular el promedio de opiniones y experticia por tema, ordenar los temas implementando el **Merge Sort** y generar estadísticas globales, como los encuestados con valores extremos de experticia u opinión, así como los promedios generales.

En general, todas las clases implementadas cuentan con métodos para inicializarse con sus respectivos atributos, usando el “__init__”, y el “__rep__” para facilitar su representación en formato de texto durante la depuración y retornando, al imprimir o inspeccionar una instancia de la clase, una lista con los valores de los atributos.

El uso de listas resulta esencial en esta implementación por su flexibilidad y capacidad de redimensionarse dinámicamente, permitiendo almacenar y gestionar los datos sin restricciones de tamaño inicial, como al momento de agregar encuestados a las preguntas o preguntas a los temas. Además, las listas como estructura de datos permiten implementar algoritmos personalizados de ordenamiento eficientes temporalmente como el **Merge Sort**; con una complejidad de $T(n) = O(nlog(n))$; asegurando un rendimiento óptimo con conjuntos de datos de distintos tamaños. Este diseño modular permite cumplir con los requerimientos del problema, además de facilitar la extensión y el mantenimiento del código a futuro.

Como se mencionó anteriormente, esta estrategia proporciona una base completa para evaluar las distintas consultas y operaciones requeridas por la encuesta, presentando varias ventajas en términos de simplicidad y legibilidad; pero también puede mostrar limitaciones cuando se enfrentan volúmenes de datos extremadamente grandes, que se traduce a un consumo elevado de memoria.

Descripción de los Algoritmos Implementados.

En esta primera estrategia, el algoritmo seleccionado para ordenar los datos de las listas es el **Merge Sort**. Esto se decidió así, debido a su eficiencia y estabilidad. Este algoritmo garantiza un desempeño óptimo al ordenar las listas, ya que su complejidad en el mejor y en el peor de los casos es la misma, es decir $T(n) = O(nlog(n))$, lo cual asegura buenos tiempos de cómputo con distintos volúmenes de datos. La implementación se ha adaptado para satisfacer las necesidades específicas del problema que tiene qué ver con el criterio de ordenamiento, introduciendo una función clave (key) como parámetro.

El algoritmo se estructura en dos partes o funciones principales: “merge” y “merge_sort”, que trabajan en conjunto para realizar el ordenamiento mediante la técnica de "divide y vencerás":

> **Función merge:** Es una función auxiliar que combina dos listas ordenadas en una sola lista también ordenada. Para ello, inicializa la lista vacía “sorted_list” que almacena los elementos combinados y ordenados. Luego compara las sublistas “left” y “right”; obtenidas de dividir la lista ingresada original en la función “MergeSort”; usando la función parámetro clave (key) para determinar el orden. A continuación comienza a agregar los elementos más pequeños; de acuerdo a la clave; a la lista “sorted_list” hasta que una de las sublistas quede vacía. Los elementos restantes de la otra sublista se agregan directamente al final de la lista “sorted_list” y finalmente esta lista se retorna como resultado de la función.
>
> **Función merge_sort:** Es la función principal que aplica la técnica de "divide y vencerás" para ordenar la lista de manera recursiva. Esto lo consigue teniendo inicialmente en cuenta que si la lista tiene 0 o 1 elementos, se retorna a sí misma, pues ya se encuentra ordenada. En caso contrario, la función divide la lista en dos mitades o sublistas: “left” y “right”, y se llama recursivamente a sí misma para ordenar cada sublista. Luego combina las listas ordenadas utilizando la función “merge” y así retornar la lista completa ordenada.

La versatilidad del anterior algoritmo radica en el uso del parámetro de la función clave (key), que permite ordenar los datos según criterios específicos ingresados, como el nivel de experticia, el promedio de opiniones, o cualquier atributo definido en las clases.

Como se ha mencionado antes, la complejidad del **Merge Sort** se define mediante la siguiente ecuación de recurrencia:

$$T(n) = 2T(\frac{n}{2}) + O(n)$$

Donde $O(n)$ representa el tiempo necesario para fusionar las dos mitades de la lista. Resolviendo esta ecuación se obtiene una complejidad final de $T(n) = O(nlog(n))$.

Los resultados anteriores, hacen del **Merge Sort** una buena elección para ordenar los datos en esta primera estrategia, asegurando que los resultados cumplan con los requerimientos

establecidos por la empresa de consultoría de datos y que el sistema opere eficientemente con volúmenes variados de datos.

Complejidad Temporal.

Como se ha venido mencionando, nuestra implementación del algoritmo **Merge Sort** está basada en la propuesta en clase, con la diferencia de incluir un parámetro de ordenamiento mediante una función clave (key). A continuación, se analiza la complejidad temporal del algoritmo utilizando una ecuación de recurrencia y el método maestro:

Ecuación de recurrencia: La ecuación que modela la complejidad de **Merge Sort** se puede expresar como:

$$T(n) = 2T(\frac{n}{2}) + O(n)$$

donde:

$T(n)$**:** Es el tiempo de ejecución total del algoritmo para una lista de tamaño n.

$2T(\frac{n}{2})$**:** Representa el tiempo requerido para dividir la lista en dos mitades y procesarlas recursivamente en la función “merge_sort”.

$O(n)$**:** Es el tiempo necesario para combinar las dos sublistas ordenadas en una sola lista ordenada mediante la función “merge”.

En la función “merge” se utilizan los métodos “.append” y “.extend” para construir la lista ordenada final:

“.append”: Agrega elementos al final de la lista y tiene una complejidad de $O(1)$

“.extend”: Fusiona dos listas y tiene complejidad de $O(k)$, donde k es el número de elementos en la lista a agregar.

En el caso en que lo usamos, k es el tamaño de las sublistas “left” y “right”, implicando que $k \leq n$, por lo tanto el costo de la función “merge” es $O(n)$**.**

Teniendo en cuenta lo anterior y aplicando el método maestro para resolver la ecuación de recurrencia del algoritmo **Merge Sort**, se obtiene que:

$$T(n) = 2T(\frac{n}{2}) + O(n)$$

Donde $a = 2$, $b = 2$ y $f(n) = O(n)$. Por el método maestro se compara el lado de recurrente con el no recurrente como sigue:

$$n^{log_2 2} \quad \text{Vs} \quad O(n)$$

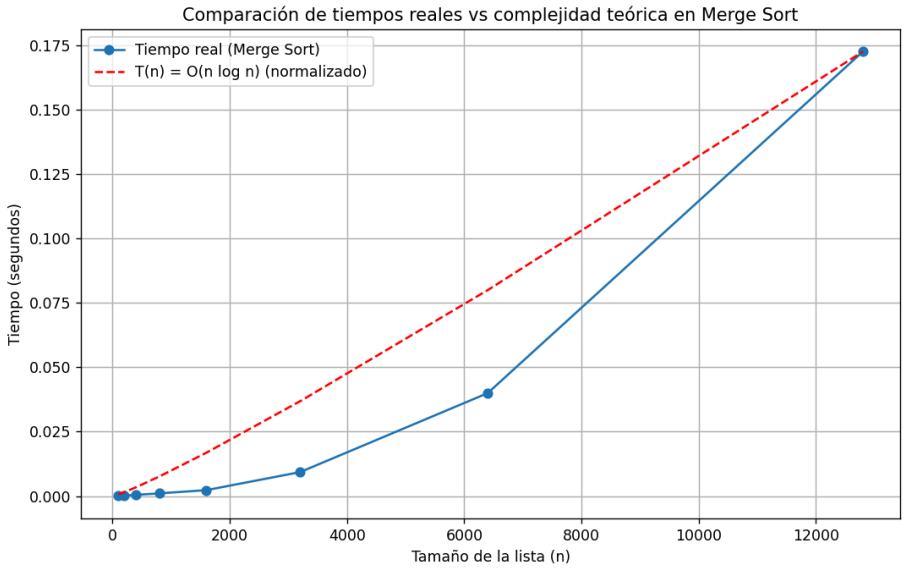
$$n^{1} \quad \text{Vs} \quad O(n)$$

$$O(n) \quad \text{Vs} \quad O(n)$$

Dado que ambos lados tienen la misma complejidad, se usa el segundo caso del método maestro, dando como resultado que la complejidad final de la ecuación de recurrencia del algoritmo **Merge Sort** es $T(n) = O(nlog(n))$. Esta complejidad temporal se aplica tanto para el mejor, como para el peor caso.

Como se puede notar, la complejidad de los ordenamientos dependen principalmente del tamaño de las listas a ordenar y de la complejidad del algoritmo **Merge Sort**, que es $T(n) = O(nlog(n))$.

Para validar lo anterior, junto con la eficiencia del algoritmo **Merge Sort**, se realizaron pruebas utilizando listas generadas aleatoriamente con tamaños de 100, 200, 400, 800, 1600, 3200, 6400 y 12800 elementos. Se compararon los tiempos de ejecución reales con la complejidad teórica $T(n) = O(nlog(n))$. Los resultados fueron los siguientes:



Los resultados mostraron una concordancia notable entre ambas curvas: el tiempo real (azul) y el tiempo teórico (roja), confirmando que la implementación del algoritmo es eficiente y que el

programa cumple con los requisitos esperados en términos de rendimiento y escalabilidad. Cabe destacar que los valores teóricos fueron normalizados para que fuesen comparables a los reales.

Descripción de la Implementación del Código.

Teniendo en cuenta lo anterior, es posible analizar cómo se implementa el método de ordenamiento del **Merge Sort** para ordenar las entidades clave del problema (encuestados, preguntas, temas y la encuesta en general) bajo los criterios definidos en el código. A continuación, se detalla el procedimiento para cada caso:

> **Ordenamiento de los encuestados dentro de una pregunta:** Para ordenar los encuestados que contestaron a una pregunta, dentro de la clase Pregunta se crea un método “ordenar_encuestados” que invoque a “Merge_Sort” con la lista de encuestados y una función lambda como clave donde se ingresan los atributos de “opinión” para ordenar y de “experticia” para desempatar. Esta función devuelve una tupla para cada encuestado e.
>
> **Ordenamiento de las preguntas dentro de un tema:** El ordenamiento de las preguntas asociadas a un tema se realiza mediante la invocación de la función “merge_sort” en un método “ordenar_preguntas” dentro de la clase Tema, ingresando como parámetros la lista de preguntas y una función lambda que devuelve para cada pregunta p la tupla con los resultados de los métodos de “opinión_promedio” para ordenar y “experticia_promedio” o el número de encuestados para desempatar.
>
> **Ordenamiento de los temas dentro de la encuesta:** Como en los anteriores casos, dentro de la clase Encuesta se crea un método “ordenar_temas”, que invoca la función del “merge_sort” que recibe una lista con los temas y la función lambda que devuelve para cada tema t la tupa con los resultados de los métodos “opininon_promedio” para ordenar y “experticia_promedio” o el número de encuestados para desempatar.
>
> **Listado de todos los encuestados:** Para generar el listado global de los encuestados, primero se construye una lista que incluye a todos los encuestados de todos los temas y preguntas. Esto se realiza iterando sobre los temas y preguntas para recopilar los datos en una sola lista. Luego, se aplica la función “merge_sort” con el listado completo de encuestados y la función lambda que devuelve para cada encuestado una tupla con los atributos de “experticia” para ordenar y “id” para desempatar.

En cada caso, e representa el número de encuestados para las preguntas, p representa el número de preguntas para los temas y t representa el número de temas para la encuesta. Dado que las funciones lambda que se usaron como parámetros dentro de la función

“merge_sort” solo acceden a atributos o métodos de las clases, su costo es constante $O(1)$, lo que no impacta significativamente la complejidad general.

Para el cálculo de las estadísticas globales, se usaron funciones nativas de Python, llamando las instancias necesarias según el caso. En caso que dos o más instancias cumplan con el requerimiento, la función escogerá cualquier caso, pues el desempate para este caso no está definido dentro del enunciado entregado por la empresa de consultoría.

Estrategia 2: Árboles Binarios.

Idea de la propuesta.

El objetivo principal de esta propuesta es implementar un **Árbol Binario de Búsqueda** (BST, por sus siglas en inglés) como estructura de datos fundamental para almacenar y organizar los encuestados, preguntas y temas de manera eficiente. La elección del BST radica en sus propiedades únicas, que permiten mantener los datos ordenados de forma natural mientras optimizan las operaciones de inserción, búsqueda y recorrido (traversal).

Esta estructura jerárquica hace que operaciones como la inserción y el recorrido in-order tengan mucho mayor eficiencia comparada con otras estructuras de datos. Al mantener el árbol balanceado (o cercano a balanceado), la complejidad computacional promedio de estas operaciones es **O(log n)**, donde n es el número de elementos en el árbol. Esto representa una mejora significativa en contraste con estructuras de datos lineales como listas o arreglos, donde las mismas operaciones podrían requerir hasta **O(n)** en el peor caso.

Ventajas del BST en esta implementación

Practicidad en la estructura de datos:

Al insertar encuestados, preguntas o temas en el árbol utilizando claves (como la opinión o la experticia), estos se organizan automáticamente de manera jerárquica y ordenada.

El recorrido in-order del BST permite recuperar los elementos en orden ascendente o descendente con facilidad.

Optimización del tiempo en consultas:

La estructura del BST reduce el tiempo necesario para acceder a datos específicos o realizar análisis, cómo calcular promedios o buscar máximos y mínimos.

Eficiencia en diferentes escenarios:

Cuando los datos se modifican de forma continua (por ejemplo, al agregar nuevos encuestados o preguntas), el BST maneja las operaciones de inserción y reordenamiento de manera más eficiente que una lista o un arreglo.

Adaptabilidad:

Permite ordenar los elementos basándose en múltiples criterios (como el promedio de opinión o el nivel de experticia), simplemente ajustando la clave de comparación durante la inserción.

Estructura de Datos Usada:

Árbol Binario de Búsqueda(BST): El BST es una estructura datos para almacenar datos de una manera ordenada que permite realizar operaciones y acceder a datos en el arreglo de datos de manera eficiente

Listas: Se utilizan para almacenar temporalmente los datos de los archivos txt para dárselo al árbol, también se utiliza para almacenar los resultados del arbol

Diccionario: Se usa para almacenar los temas

Tuplas: Se utiliza como claves compuestas inmutables para el ordenamiento de datos

Descripción de los Algoritmos Implementados.

Inserción: Este algoritmo nos permite insertar un nuevo nodo al árbol mientras se mantiene su orden. La clave es comparar el valor del nodo que se desea insertar con los nodos existentes y colocarlo en la posición adecuada, con una complejidad del **O(log n)**

In-Order: Este algoritmo recorre todos los nodos de un árbol binario siguiendo un orden específico: primero visita el subárbol izquierdo, luego el nodo raíz, y finalmente el subárbol derecho. El recorrido se realiza de manera sistemática de izquierda a derecha, asegurando que los nodos se procesan en orden ascendente si el árbol es un árbol binario de búsqueda. La complejidad del recorrido es **O(n)**, donde *n* es el número de nodos en el árbol, ya que cada nodo se visita exactamente una vez.

Complejidad Temporal.

A Continuación vamos a mostrar la complejidad tanto temporal como espacial de cada algoritmo utilizados en el proyecto

Algoritmo	Complejidad Temporal	Complejidad Espacial
Inserción (BST)	*O(log n) promedio / O(n) Peor*	*O(h)*
In-Order	*O(n)*	*O(h)*
Procesamiento del archivo	*O(n)*	*O(n)*
Ordenamiento de temas	*O(n log n)*	*O(n)*
Cálculo de estadísticas	*O(n)*	*O(1)*
Escritura archivo salida	*O(n)*	*O(1)*

Descripción de la Implementación del Código.

Esta implementación consiste en un programa en Python diseñado para procesar, analizar y generar informes de datos provenientes de una encuesta estructurada en temas, preguntas y encuestados. Se enfoca en el uso de estructuras de datos como árboles binarios para organizar y ordenar la información según criterios específicos. A continuación, se detalla su funcionamiento por secciones:

Clases Principales.

Nodo y ArbolBinario:

Estas clases implementan la estructura de un árbol binario.

Los nodos (Nodo) contienen una clave (key) para ordenamiento y un dato (data) asociado.

El árbol binario (ArbolBinario) organiza los nodos de manera jerárquica, permitiendo insertar y recorrer elementos con eficiencia.

El recorrido **in-order** se utiliza para recuperar los datos en orden descendente según las claves.

Encuestado:

Representa a una persona encuestada con atributos como identificador, nombre, nivel de experticia y opinión.

Estos datos son fundamentales para calcular métricas a nivel de preguntas y temas.

Pregunta:

Representa una pregunta específica dentro de un tema.

Contiene un árbol binario de encuestados ordenados por opinión (y en caso de empate, por nivel de experticia).

Proporciona métodos para calcular el promedio de opiniones y experticia de los encuestados asociados.

Tema:

Representa un tema que agrupa múltiples preguntas.

Usa un árbol binario para ordenar las preguntas basándose en el promedio de opiniones, seguido del promedio de experticia en caso de empate.

Funciones Principales.

Cargar_datos_desde_txt(ruta_archivo):

Lee un archivo de texto estructurado para extraer datos de los encuestados y su relación con preguntas y temas.

Crea objetos de las clases Encuestado, Pregunta y Tema y los organiza en estructuras adecuadas (árboles binarios).

Utiliza expresiones regulares para reconocer líneas de datos y procesarlas correctamente.

Guardar_resultados_en_txt(ruta_salida, temas, encuestados):

Genera un archivo de salida con los resultados de la encuesta, incluyendo:

Temas ordenados por el promedio de opiniones de sus preguntas.

Preguntas ordenadas por opinión y experticia.

Estadísticas globales como encuestados con mayor/mayor experticia, preguntas con valores extremos, y promedios generales.

Medición de Desempeño.

Medir_tiempo_ejecucion(funcion, *args):

Mide el tiempo que toma una función específica en ejecutarse, devolviendo tanto el resultado como el tiempo transcurrido.

Generar_graficos(resultados_tiempos, nombre_grafico):

Genera un gráfico que visualiza cómo el tiempo de ejecución varía en función del tamaño de los datos de entrada (medido en líneas del archivo).

Utiliza matplotlib para mostrar y guardar el gráfico.

Interfaz Básica y Flujo de Trabajo.

main():

Solicita al usuario los nombres de los archivos de entrada y salida.

Llama a las funciones principales para cargar los datos, procesarlos y guardar los resultados.

Pruebas y Graficación:

Se procesan archivos de prueba con diferentes tamaños, registrando el tiempo de ejecución.

Los resultados se representan gráficamente, proporcionando una perspectiva visual del desempeño del programa en función del tamaño de la entrada.

Estadísticas y Resultados.

Al final, el programa genera un informe con datos relevantes como:

Promedios de opiniones y niveles de experticia por pregunta y tema.

Encuestados destacados según criterios como opinión máxima/mínima o mayor/mayor experticia.

Estadísticas globales del desempeño de la encuesta.

Además, los gráficos producidos permiten evaluar el rendimiento del programa en términos de tiempo de ejecución, lo que es útil para validar su escalabilidad en diferentes tamaños de datos.

Aspectos Técnicos Destacados

Ordenamiento Eficiente:

El uso de árboles binarios permite un ordenamiento eficiente de encuestados, preguntas y temas según múltiples criterios jerárquicos.

Escalabilidad:

La implementación mide y grafica el desempeño en archivos de prueba, ayudando a evaluar cómo se comporta con datos más extensos.

Legibilidad y Modularidad:

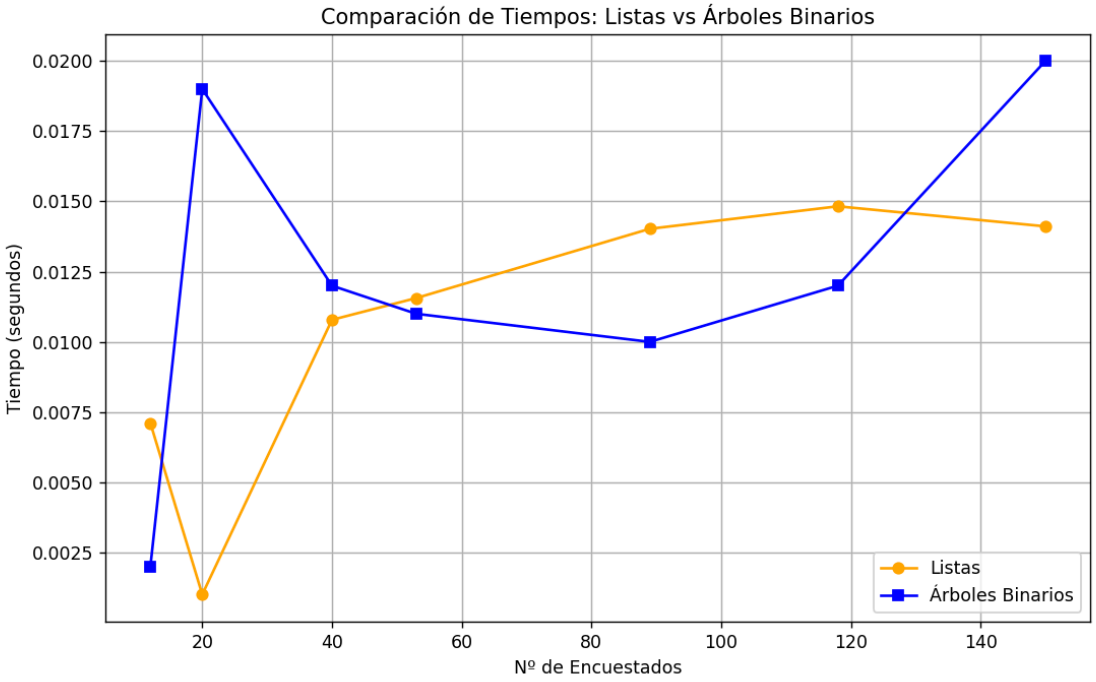
El código está estructurado en clases y funciones bien definidas, facilitando su comprensión y mantenimiento.

Flexibilidad:

Diseñado para procesar archivos de texto con estructuras variadas y generar resultados personalizables.

COMPARACIÓN TOTAL DE COMPLEJIDAD ENTRE ESTRATEGIAS Y CONCLUSIONES.

Al comparar los tiempos de ejecución de las dos estrategias propuestas con distintos volúmenes de datos (Número de encuestados), se obtiene la siguiente representación gráfica:



En la anterior gráfica se puede observar que al inicio, el tiempo de implementar árboles binarios es más bajo que para la implementación hecha con listas, especialmente cuando el número de encuestados es pequeño (menor a 20). Luego, el tiempo de ejecución pasa a ser mayor en los árboles binarios, que en las listas cuando el número de encuestados está entre 20 y 40.

A medida que aumenta el volumen de datos, los tiempos tienden a estabilizarse, aunque se observa un crecimiento gradual en la curva de las listas, lo que podría sugerir que las listas pueden manejar volúmenes pequeños de datos con rapidez, pero a medida que aumenta el tamaño, pueden presentar más dificultades en eficiencia. Por otro lado, la curva de árboles binarios tiende a estabilizarse e incluso a disminuir, mostrando mejor eficiencia comparativa con listas en volúmenes mayores.

En general, se podría pensar que para pequeños volúmenes de datos las listas muestran tiempos más bajos, lo cual puede deberse a su simplicidad en la implementación y acceso secuencial directo. Mientras que para grandes volúmenes de datos (más de 80 encuestados)

los árboles binarios se vuelven más competitivos e incluso podrían ser más eficientes, ya que mantienen tiempos de procesamiento estables o crecientes más lentamente.

Aunque en la gráfica solo se están teniendo en cuenta tiempos, la implementación usando listas tiene limitaciones de espacio y rendimiento para grandes volúmenes de datos, por la forma en que se están ordenando los datos. En cambio los árboles binarios, al ser más complejos en estructura, pueden usar más espacio inicialmente, pero son escalables y manejan grandes volúmenes con mayor eficiencia en términos de tiempo debido a su acceso logarítmico en operaciones bien balanceadas.

Como conclusión final, si se está trabajando con un sistema que crecerá en volumen de datos, sería recomendable optar por **árboles binarios**.

ENLACE AL REPOSITORIO DE GIT HUB: https://github.com/Juanmaperea/Proyecto_ADA.git

ENLACE AL VIDEO DE PRESENTACIÓN:
https://drive.google.com/file/d/1cqGH_Np8SA8R-1nln8vw291eibdeUhTK/view?usp=sharing