ESCUELA POLITÉCNICA NACIONAL

FACULTAD DE INGENIERÍA EN SISTEMAS

NOMBRE DEL PROYECTO EN EL QUE SE DESARROLLA EL PROYECTO DE LA MATERIA ESTRUCTURA DE DATOS Y ALGORITMOS 1

Análisis comparativo de algoritmos de ordenamiento con datos masivos

PROYECTO PRESENTADO COMO REQUISITO PARA EL PASE DE LA MATERIA ESTRUCTURA DE DATOS Y ALGORITMOS 1

NOMBRES Y APELLIDOS DEL ESTUDIANTE

Andres Felipe Merino Bravo

Aidan Hamilton Carrasco Aguirre

Deysi Abigail Guachamin Guauna

DIRECTOR: NOMBRES Y APELLIDOS DEL DIRECTOR

Boris Alfonso Astudillo Espinoza

boris.astudillo@epn.edu.ec

26/11/2025

CERTIFICACIONES

Yo, Andres Felipe Merino Bravo, declaro que el trabajo de integración curricular aquí descrito es de mi autoría; que no ha sido previamente presentado para ningún grado o calificación profesional; y, que he consultado las referencias bibliográficas que se incluyen en este documento.

Andres Felipe Merino Bravo

Yo, Aidan Hamilton Carrasco Aguirre, declaro que el trabajo de integración curricular aquí descrito es de mi autoría; que no ha sido previamente presentado para ningún grado o calificación profesional; y, que he consultado las referencias bibliográficas que se incluyen en este documento.

Aidan Hamilton Carrasco Aguirre

Yo, Deysi Abigail Guachamin Guauña, declaro que el trabajo de integración curricular aquí descrito es de mi autoría; que no ha sido previamente presentado para ningún grado o calificación profesional; y, que he consultado las referencias bibliográficas que se incluyen en este documento.

Deysi Abigail Guachamin Guauña

Certifico que el presente trabajo de integración curricular fue desarrollado por NOMBRE_ESTUDIANTE, bajo mi supervisión.

_____

NOMBRE_DIRECTOR

DIRECTOR

DECLARACIÓN DE AUTORÍA

A través de la presente declaración, afirmamos que el trabajo de integración curricular aquí descrito, así como el (los) producto(s) resultante(s) del mismo, son públicos y estarán a disposición de la comunidad a través del repositorio institucional de la Escuela Politécnica Nacional; sin embargo, la titularidad de los derechos patrimoniales nos corresponde a los autores que hemos contribuido en el desarrollo del presente trabajo; observando para el efecto las disposiciones establecidas por el órgano competente en propiedad intelectual, la normativa interna y demás normas.

Andres Felipe Merino Bravo

Aidan Hamilton Carrasco Aguirre

Deysi Abigail Guachamin Guauna

ÍNDICE DE CONTENIDO

RESUMEN

Dentro del proyecto se aborda la pregunta de cómo manejar grandes volúmenes de datos. Gracias a esto, se definió el fin de realizar un análisis a un dataset con 1 millón de filas. Dicho dataset fue obtenido de las bases de datos de Yelp, una popular página de reseñas en países angloparlantes.

Para el manejo de datos se implementaron dos arquitecturas de software distintas para comparar rendimiento: una solución secuencial en Java (un solo núcleo) y una solución paralela en Python (multiprocesamiento). En ambas arquitecturas se implementaron los algoritmos QuickSort y HeapSort (de complejidad O (n log n)), con el fin de comparar la eficiencia por medio del tiempo que tardan ambas arquitecturas en procesar el millón de datos.

PALABRAS CLAVE: Algoritmos de Ordenamiento, Multiprocesamiento, QuickSort, HeapSort, Java, Python.

ABSTRACT

The project addresses the question of how to handle large volumes of data to analyze a dataset with 1 million records. This dataset was obtained from Yelp's databases, a popular review site in English-speaking countries.

To manage the data, two different software architectures were implemented to compare performance: a sequential solution in Java (single core) and a parallel solution in Python (multiprocessing). In both architectures, the QuickSort and HeapSort algorithms (with complexity O (n log n)) were implemented to compare efficiency based on the time each architecture takes to process the million records.

The results showed that:

KEY WORDS: Sorting Algorithms, Multiprocessing, QuickSort, HeapSort, Java, Python.

1 DESCRIPCIÓN DEL COMPONENTE DESARROLLADO

Se creó un sistema de software en dos partes para el procesamiento, cálculo y ordenamiento de un dataset masivo de 1 millón de registros de reseñas. Se compone de 2 módulos: Java diseñado para la ejecución secuencial en un solo núcleo y Python con ejecución paralela haciendo uso de librerías que habilitan el multiprocesamiento, con el fin de mejorar la eficiencia del programa. Ambos módulos implementan una limpieza de datos, el cálculo de la fórmula de confianza y la aplicación de QuickSort y HeapSort.

1.1 Objetivo general

Comparar el rendimiento de los algoritmos de ordenamiento QuickSort y HeapSort procesando un millón de datos, utilizando diferentes arquitecturas: secuencial (Java) y paralela (Python).

1.2 Objetivos específicos

1. Implementar QuickSort y HeapSort para manejar el millón de datos
2. Implementar el multiprocesamiento para QuickSort y HeapSort en Python
3. Analizar los tiempos de ejecución resultantes en tres computadoras distintas para determinar la eficiencia de los algoritmos y el impacto del multiprocesamiento.
4. Generar un Top 20 de los restaurantes con más reseñas y mejor rating con el uso de algoritmos y la fórmula estadística de confianza

1.3 Alcance

El proyecto abarca desde la limpieza del archivo CSV con los datos en bruto (yelp_database.csv), pasando por el cálculo de las métricas estadísticas (promedio global), la aplicación de una fórmula de ponderación a cada dato dentro del dataset y el ordenamiento final para generar el top 20 final. Se limita al uso de únicamente 2 algoritmos QuickSort y HeapSort, en 2 lenguajes de programación Python y Java, donde Python será el encargado de implementar multiprocesamiento.

1.4 Marco teórico

QuickSort

El algoritmo Quicksort es considerado uno de los métodos más efectivos en el ordenamiento de arreglos y listas de datos masivos. Método desarrollado por Tony hoare en 1986 que actualmente es catalogado como uno de los mejores algoritmos de divide y vencerás, debido a que su logica se basa en dividir un problema en subproblemas y resolverlos de forma recursiva.

El funcionamiento general de Quicksort se resume en:

1. **Elección del pivote: Se toma un elemento que exista en el arreglo como referencia para dividir el mismo.** Este pivote puede seleccionarse de distintas formas: el primer elemento, el último, el centrar o mediante estrategias más avanzadas como “mediana de tres”. La elección del pivote influye directamente la eficiencia del algoritmo
2. Particionamiento: Se reorganiza los elementos del arreglo de tal forma que todos los elementos menores al pivote queden en la izquierda y los mayores al mismo a la derecha. Este proceso asegura que, tras la partición, el pivote quede ordenado.
3. Recursión: Se aplica el mismo procedimiento de selección y particionamiento a los subarreglos que se vayan generando hasta que quede uno que contenga uno o ningún elemento.

Quicksort se destaca principalmente por su eficiencia en arreglos grandes y su bajo consumo de memoria adicional, ya que no requiere estructuras de datos auxiliares como si lo hace el Mergesort. Si en el peor de los casos el pivote elegido es el más pequeño o grande del arreglo de la partición el algoritmo se vuelve pesado. Para reducir esto se usan ciertas estrategias que seleccionen un pivote más adecuado.

Entre las ventajas se destacan:

- Rapidez en grandes volúmenes de datos.
- Bajo consumo de memoria adicional.
- Sencillez conceptual.

Entre sus desventajas se encuentran:

- Sensibilidad a la elección del pivote.
- No es un algoritmo estable.

Heapsort

El Heapsort es un algoritmo de ordenamiento basado en la estructura de datos conocida como heap (arboles), un tipo especial de árbol binario que cumple con que se debe

establecer un max-head, cada nodo es mayor o igual a sus hijos, también un min-heap, donde cada nodo es menor o igual a sus hijos. Esta propiedad permite acceder de manera eficiente al elemento máximo o mínimo en el tiempo contante.

El funcionamiento de Heapsort puede dividirse en las siguientes fases:

1. Construcción del heap: A partir del arreglo original, se reorganizan los elementos para formar el heap completo. En esta fase se asegura que cada subárbol cumpla con la propiedad de heap, lo que garantiza que el elemento que está en la raíz sea el mayor o el menor.
2. Extracción ordenada: una vez construido el heap, se intercambia el elemento raiz con el ultimo elemento del arreglo y se reduce el tamaño del heap en una unidad. Como resultado se ajusta la estructura para mantener el heap. Este proceso se repite hace que todos los elementos se encuentren ordenados.

Las características esenciales del Heapsort son la complejidad temporal lo cual lo hace muy predecible en termino de rendimiento. Aparte de no requerir una memoria adicional significativa, esto porque el ordenamiento se realiza en el arreglo, lo que lo hace menos sensible que el Quicksort.

Entre las ventajas se destacan:

- Complejidad temporal garantizada.
- Uso eficiente de memoria.
- Independencia de la distribución de los datos de entrada.

Entre sus desventajas se encuentran:

- No es un algoritmo estable.
- Es generalmente más lento que ele Quicksort en la práctica, debido al mayor número de operaciones.

Heapsort es usualmente utilizado en sistemas donde se requiere un rendimiento consciente de cada apartado, como en aplicaciones embebidas o sistemas que manejan gran cantidad de datos en tiempo crítico. Además, su relación con la estructura heap lo hace útil en la implementación de colas de prioridad y en optimización.

2 METODOLOGÍA

El proyecto se desarrolló bajo un enfoque cuantitativo experimental. Se diseñó un experimento controlado donde la variable independiente es la arquitectura (Secuencial vs. Paralela) y el algoritmo (QuickSort vs. HeapSort), y la variable dependiente es el tiempo de ejecución.

2.1 Caso de estudio y fórmula de confianza

La Fórmula de Confianza, es la fórmula creada y elegida con el objetivo de saber qué tan fiables son las reseñas de los clientes.

$$Puntuación\ total = \left(\frac{v}{v+m} \cdot R\right) + \left(\frac{m}{v+m} \cdot C\right)$$

Ecuación 2.1: La fórmula de la confianza

Donde:

R: Rating individual
v: Número de reseñas (votos)
C: El rating promedio de todo el dataset
m: El "umbral de confianza"

El valor de m lo marcamos como 100, lo que quiere decir que el restaurante necesita al menos 100 reseñas para que su rating real R y el promedio C tengan el mismo peso.

Lo que quiere decir que, si tiene menos de 100 reseñas, su puntuación apuntará a su promedio C, mientras que, si tiene más de 100, su puntuación apuntará a su rating real R.

La fórmula utiliza dos fracciones que actúan como "pesos" en un promedio ponderado, donde los pesos suman siempre 1:

$$\frac{v}{v+m} + \frac{m}{v+m} = 1$$

Ecuación 2.2: Pesos de la ecuación

El Primer Peso: $\frac{v}{v+m}$ determina qué porcentaje del total de votos (reales + virtuales) proviene de reseñas reales.

Cuando v es pequeño (ejemplo: v = 1): $\frac{1}{101} \approx 0.01$ el rating real R tiene poco peso.

Cuando v es grande (ejemplo: v = 1000): $\frac{1000}{1100} \approx 0.91$ el rating real R domina.

Representa el "grado de confianza" en el rating del restaurante.

El Segundo Peso: $\frac{m}{v+m}$ representa cuánto arrastramos la puntuación hacia el promedio global cuando hay pocas reseñas.

Cuando v es pequeño (ejemplo: v = 1): $\frac{100}{101} \approx 0.99$ el promedio global C domina.

Cuando v es grande (ejemplo: v = 1000): $\frac{100}{110} \approx 0.09$ el promedio global C tiene poco peso.

2.2 Algoritmos de ordenamiento utilizados

En este proyecto se usaron los algoritmos Quicksort y Heapsort, debido a su gran eficiencia con grandes cantidades de datos, se usaron estos algoritmos segun los criterios previamente definidos.

Quicksort en el proyecto

El algoritmo Quicksort se utilizó de manera recursiva para ordenar las listas, en las cuales cada una representa un restaurante con la información de columna ***Organization, Rating y NumberReview.***

1. **Ordenamiento:** Quicksort se aplicó para el ordenamiento de los restaurantes según su ***NumberReview,*** el número de reseñas de los restaurantes. Lo cual permitió saber el número de usuarios que interactúan con estos restaurantes.
2. **División de la lista:** Este algoritmo elige un pivote central que se va a encargar de dividir en tres sublistas la principal: elementos con menor porcentaje de resenias, mayor e igual número.
3. **Recursividad:** Cada sublista se ordena de manera recursiva hasta que todos los elementos estén posicionados correctamente.

Heapsort en el proyecto

El Heapsort se usó para un ordenamiento más avanzado, basado en la puntuación total de los restaurantes. **Esta puntación está dada por la calificación promedio del Rating y el número de reseñas NumberReview mediante la ecuación 2.1.**

1. **Construcción del heap:** Cada lista de restaurantes se transforma en un heap, lo que hace que el elemento mayor se convierta en la raíz.
2. **Extracción ordenada:** El algoritmo intercambia multiples veces la raiz del heap con el ultimo elemento del arreglo y reajusta su estructura para mantener el orden. Este proceso se repite hasta que todos los elementos se encuentren ordenados.

2.3 Implementación técnica (Java y Python)

El proyecto se divide en dos partes: procesamiento normal y multiprocesamiento, cada una implementada en un lenguaje diferente para sacarle provecho a las características de cada uno.

Procesamiento Normal (Java)

Lenguaje de programación:

- Java: Lenguaje de programación principal para la primera parte del proyecto.

Bibliotecas:

- **java.io.* :** Clases generales para manejo de entrada/salida (InputStream, OutputStream, File, etc.)
- **Java.io.BufferedReader:** Leer texto de manera eficiente desde un flujo de entrada (como archivos)
- **java.io.BufferedWriter : Escribir** texto de manera eficiente en un flujo de salida (como archivos)
- **Java.io.InputStream:** Accede a archivos que estan dentro del programa.
- **Java.io.InputStreamReader:** Lee los archivos que estan dentro del programa.
- **java.io.FileReader : Leer** archivos de texto caracter por caracter
- **java.io.FileWriter :** Escribir archivos de texto caracter por caracter
- **java.io.IOException** : Manejar excepciones relacionadas con entrada/salida de archivos
- **java.io.File :** Trabajar con archivos y directorios (crear, eliminar, comprobar existencia)

- **Java.util.ArrayList:** Lista dinámica que puede crecer o decrecer según los elementos
- **java.util.Scanner :** Leer datos desde teclado o desde archivos de texto
- **java.util.Locale :** Configurar ajustes regionales (formato de números, fechas, moneda)

Multiprocesamiento (Python)

Lenguaje de programación:

- **Python:** Lenguaje de programación para multiprocesos.

Bibliotecas:

- **pandas** para la lectura, manejo y escritura de archivos CSV.
- **time** para medir el tiempo de ejecución de los procesos.
- **os** para verificar la existencia de archivos en el sistema.
- **sys** para aumentar el límite de recursión y permitir ordenar grandes listas con Quicksort.
- **math** para operaciones matemáticas necesarias en la división de listas.
- **multiprocessing** para ejecutar ordenamientos en paralelo aprovechando múltiples núcleos de CPU.
- **Locale** biblioteca para poder usar signo de puntuación de distintas localidades.

Funciones de cada clase

Limpieza de datos (Preprocesamiento)

Java

- **Clase: Limpiar datos**
- **Función:** Detecta el CSV original (**yelp_database.csv**) y lo lee, selecciona solo tres columnas temporales para no forzar la ram (Optimización, Rating, Numberreview), elimina espacios y comillas, y guarda los cambios en un archivo limpio (**datos_procesados.csv**).

Python

- **Script: limpiarDatos.py**
- **Función:** Lee el CSV original usando pandas, selecciona las mismas tres columnas que en java, elimina comillas y espacios en Organization, convierte Rating y

NumberReview a numérico, elimina filas con valores nulos, y guarda el resultado en **datos_procesados_py.csv.**

Ordenamiento y puntuación bayesiana

Java

- **Clase: Ordenar.java**

Función:

- Carga los datos procesados y limpios (datos_procesados.csv).
- Aplica la formula (2.1) que calcula la puntuacionTotal de cada restaurante.

Ordena los Datos:

- **Quicksort:** por ***Numberreview*** para generar un archivo que luego usara el heap.
- **Heapsort:** por ***puntuacionTotal*** para obtener el ranking final.
- Guarda los datos en ***restaurantes_ordenados.csv*** y muestra top 20 restaurantes.

Python

Enfoque:

- Procesamiento de grandes volúmenes de datos en paralelo usando multiprocessing.
- Ordenamiento eficiente de listas usando QuickSort o HeapSort en paralelo.
- Aprovechamiento de múltiples núcleos de CPU para reducir tiempo de ejecución.

El ordenamiento en Python es similar o equivalente al de java, con el pequeño detalle que permite paralelización para reducir los tiempos de procesamiento en datasets muy grandes.

Resumen del flujo de trabajo

- **Preprocesamiento:** Limpieza de datos con Java (limpiarDatos) o Python (limpiar_datos.py).
- **Archivo limpio: datos_procesados.csv (Java) o datos_procesados_py.csv (Python).**

- **Cálculo de puntuación:** Fórmula bayesiana para ponderar rating y número de reseñas.

Ordenamiento:

- **QuickSort** por número de reseñas (archivo intermedio).
- **HeapSort** por puntuación total (ranking final).

Resultado: Archivo final con ranking de restaurantes y visualización del Top 20 en consola.

Implementación de Multiprocesamiento y Paralelismo de Datos (falta)

La optimización del rendimiento del código se logra mantener mediante la librería multiprocessing, la cual permite evadir las limitaciones que tienen normalmente los programas (Global Interpreter Lock) de Python. A diferencia de la ejecución tradicional, donde una sola unidad de procesamiento maneja toda la carga del programa, esta implementación permite trabajar con cada núcleo de la computadora.

El flujo de trabajo paralelo implemento la función ordenar_paralelo se divide en las siguientes etapas fundamentales:

1. **Detección de Recursos y Segmentación de Datos:** Detección de Recursos y Segmentación de Datos: El algoritmo de multiprocessing comienza detectando los núcleos de la máquina en la que se ejecuta mediante mult.cpu_count(), que determina cuántos procesos simultáneos va a realizar el sistema. Basándose en este número (normalmente los mismos del núcleo), la lista original de restaurantes se divide matemáticamente en “trozos” (chunks) de tamaño equitativo. Esta segmentación ayuda a dividir el problema de procesamiento masivo de datos simultáneos haciéndolos subproblemas independientes y manejables.
2. **Orquestación mediante Pool de procesos:** Se tiene que instancia en objeto Pool, que actúa como un gestor de recursos. A través del método pool.map, el programa se encarga se distribuir los trozos de los datos a un proceso independiente en cada núcleo. En esta etapa cada uno de ellos ejecuta la función paralelismo_multiproceso de manera aislada. Aplica el algoritmo seleccionado (QuickSort o HeapSort) únicamente en los fragmentos asignados. Esto significa que, si un equipo tiene 12 núcleos, se ordenará en 12 sublistas reduciendo el tiempo.

3. **Procesamiento distribuido:** Dentro de cada uno de los subprocesos, se ejecuta la lógica de ordenamiento puro. El código incluye una función de identidad del proceso que obtiene su PID y nombre para asegurar la trazabilidad. Dado que cada uno de los procesos ocupan su propio espacio de memoria, el ordenamiento de un trozo individual no influye con el otro, garantizando la integridad de los datos durante la ejecución.
4. **Convergencia y fusión de resultados:** Una vez que todos los procesos han culminado y devuelto todas las sublistas, el programa principal toma el control. No obstante, tener los segmentos ordenados no significa que la lista general esté ordenada. Por ello, se ejecuta la función fusionar_chucks, la cual toma los resultados parciales y los unifica secuencialmente. Esta función compara cada elemento y los ensambla en una lista definitiva, completando así el ciclo del ordenamiento paralelo.

3 RESULTADOS, CONCLUSIONES Y RECOMENDACIONES

3.1 Resultados

Tabla 3.1. Resultados en Computadora 1 (Felipe) *(Especificaciones: CPU: core i5 8va, RAM: 16 GB, 4 núcleos)*

Algoritmo	**Arquitectura**	**Tiempo Total [ms]**	**Notas**
QuickSort	Java (1 Núcleo)	759.8 ms	Orden por Reseñas
HeapSort	Java (1 Núcleo)	4080.0 ms	Cálculo + Orden por Puntuación
QuickSort	Python (Paralelo)	6523.38 ms	Orden por Reseñas
HeapSort	Python (Paralelo)	22153.20 ms	Cálculo + Orden por Puntuación

Tabla 3.2. Resultados en Computadora 2 (Aidan) *(Especificaciones: CPU: core i5 12va, RAM: 16 GB, 12 núcleos)*

Algoritmo	**Arquitectura**	**Tiempo Total [ms]**	**Notas**
QuickSort	Java (1 Núcleo)	445.4 ms	Orden por Reseñas
HeapSort	Java (1 Núcleo)	1126.2 ms	Cálculo + Orden por Puntuación

QuickSort	Python (Paralelo)	1523.11 ms	Orden por Reseñas
HeapSort	Python (Paralelo)	4439.27 ms	Cálculo + Orden por Puntuación

Tabla 3.3. Resultados en Computadora 3 (Deysi) *(Especificaciones: CPU: i5-4460, RAM: 8,00 GB)*

Algoritmo	**Arquitectura**	**Tiempo Total [ms]**	**Notas**
QuickSort	Java (1 Núcleo)	517.8 ms	Orden por Reseñas
HeapSort	Java (1 Núcleo)	2273.7 ms	Cálculo + Orden por Puntuación
QuickSort	Python (Paralelo)	10018.63 ms	Orden por Reseñas
HeapSort	Python (Paralelo)	16513.14 ms	Cálculo + Orden por Puntuación

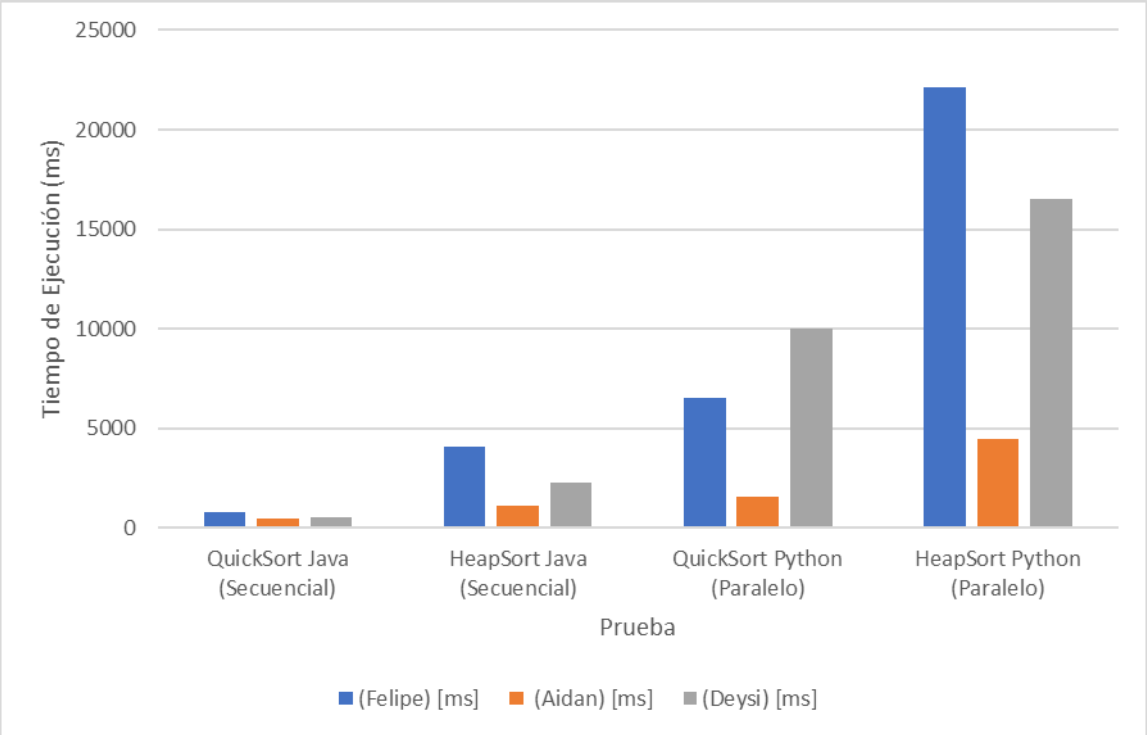

Figura 3.1. Gráfica comparativa de tiempos de ejecución entre arquitecturas.

3.2 Conclusiones

Los resultados obtenidos del análisis comparativo de arquitecturas y algoritmos de ordenamiento al procesar un millón de registros en tres ambientes de *hardware* distintos permiten establecer las siguientes conclusiones:

1. **Superioridad Definitiva de la Arquitectura Secuencial (Java):** Se concluye que, para el volumen de un millón de registros, la implementación secuencial en Java fue **significativamente más rápida y eficiente** que la solución paralela en Python. El tiempo mínimo de ejecución para QuickSort en Java fue de 445.4 ms (C2), mientras que el mejor tiempo para QuickSort en Python Paralelo fue de 1523.11 ms (C2), lo que demuestra que la optimización del código en un solo núcleo (Java) supera al rendimiento de la ejecución en múltiples núcleos (Python).

2. **Costo Elevado de la Paralelización en Python:** La diferencia de rendimiento entre arquitecturas se atribuye al **alto *overhead*** de la librería multiprocessing de Python. El tiempo que se necesita para gestionar los procesos paralelos, como la serialización, distribución y fusión de los datos segmentados, termina siendo mayor

a la velocidad que se gana al dividir el trabajo entre los núcleos. Esto invalida por completo el beneficio del paralelismo, por lo que no se termina por aplicar el concepto en este caso.

3. **Confirmación de la Eficiencia de QuickSort en la Práctica:** Se ha confirmado la expectativa teórica de **Quicksortk es consistentemente más eficiente que HeapSort** en escenarios de ordenamiento del mundo real. En todas las pruebas y arquitecturas, Quicksort demostró tener tiempos de ejecución más cortos, lo que refleja su superioridad con respecto HeapSort. A un que ambos algoritmos tienen una complejidad O (n \log n), Quicksort es superior en la práctica.

4. **Impacto Cuantificable del Cálculo de Ponderación:** Observamos que el tiempo de ejecución de **HeapSort** fue considerablemente más largo que el de QuickSort .Por ejemplo en C2, HeapSort fue 2.5 veces más lento en Python y 4 veces más lento en Java. Esta diferencia se debe a que HeapSort tiene que calcular el **Puntuación de Confianza (Ecuación 2.1)** en su proceso, lo que añade una carga de cálculo y ponderación a la tarea de ordenamiento. Esto confirma el costo de obtener una métrica de *ranking* más robusta.

3.3 Recomendaciones

Las siguientes recomendaciones se derivan de los hallazgos para optimizar futuras implementaciones y líneas de investigación.

1. **Optimización de la Fase de Fusión Paralela:** Se recomienda hacer cambios en la función fusionar_chucks para implementar una **estrategia de fusión paralela o arborescente**, en lugar de una fusión secuencial que es la q se está utilizando ahora. Utilizar el módulo multiprocessing para fusionar los trozos en pares (aplicando el concepto de Divide y Vencerás) podría ayudar a reducir el cuello de botella de un solo hilo, aprovechando mejor los núcleos de CPU disponibles y minimizando el ***overhead*** de la etapa final del ordenamiento.

2. **Vectorización del Cálculo de Puntuación:** Dado que ya se está utilizando **Pandas** y **NumPy** ,los cuáles, están disponibles en el entorno Anaconda, lo mejor sería **vectorizar el cálculo** de la Puntuación de Confianza (Ecuación 2.1) usando operaciones directamente en las series de pandas o *arrays* de NumPy. Esto sacaría del bucle de Phyton al cálculo, mejorando drásticamente el tiempo de preprocesamiento para HeapSort.

3. **Implementación de una Estrategia de Pivote Robusta para QuickSort:** Para la implementación de QuickSort, es recomendable obsta por la adopción de un pivote más estable , como la **"mediana de tres"**. Esto es fundamental para mitigar el riesgo de caer en el peor de los casos O(n^2) con la consecuente ralentización, garantizando que el rendimiento se mantenga consistentemente en O (n \log n) sin importar la distribución de los datos de entrada.
4. **Análisis de Sensibilidad de la Métrica de Confianza:** Para futuras investigaciones, se sugiere llevar a cabo un estudio de sensibilidad que varíe el umbral de confianza (m).Es fundamental evaluar cómo los cambios en este parámetro influyen en la composición del Top 20 final, ya que esto nos ayudará a determinar la solidez de la fórmula estadística y a identificar el valor óptimo para aplicaciones futuras.

4 REFERENCIAS BIBLIOGRÁFICAS

[1] Oracle, "Package java.io Description," Java Platform, Standard Edition 8 API Specification, 2024. [En línea]. Disponible: https://docs.oracle.com/javase/8/docs/api/java/io/package-summary.html. [Accedido: 25-nov-2025].

[2] Oracle, "Class BufferedReader," Java Platform, Standard Edition 8 API Specification, 2024. [En línea]. Disponible: https://docs.oracle.com/javase/8/docs/api/java/io/BufferedReader.html. [Accedido: 25-nov-2025].

[3] Oracle, "Class BufferedWriter," Java Platform, Standard Edition 8 API Specification, 2024. [En línea]. Disponible: https://docs.oracle.com/javase/8/docs/api/java/io/BufferedWriter.html. [Accedido: 25-nov-2025].

[4] Oracle, "Class InputStream," Java Platform, Standard Edition 8 API Specification, 2024. [En línea]. Disponible: https://docs.oracle.com/javase/8/docs/api/java/io/InputStream.html. [Accedido: 25-nov-2025].

[5] Oracle, "Class InputStreamReader," Java Platform, Standard Edition 8 API Specification, 2024. [En línea]. Disponible: https://docs.oracle.com/javase/8/docs/api/java/io/InputStreamReader.html. [Accedido: 25-nov-2025]

[6] Oracle, "Class FileReader," Java Platform, Standard Edition 8 API Specification, 2024. [En línea]. Disponible: https://docs.oracle.com/javase/8/docs/api/java/io/FileReader.html. [Accedido: 25-nov-2025].

[7] Oracle, "Class FileWriter," Java Platform, Standard Edition 8 API Specification, 2024. [En línea]. Disponible: https://docs.oracle.com/javase/8/docs/api/java/io/FileWriter.html. [Accedido: 25-nov-2025].

[8] Oracle, "Class IOException," Java Platform, Standard Edition 8 API Specification, 2024. [En línea]. Disponible: https://docs.oracle.com/javase/8/docs/api/java/io/IOException.html. [Accedido: 25-nov-2025].

[9] Oracle, "Class File," Java Platform, Standard Edition 8 API Specification, 2024. [En línea]. Disponible: https://docs.oracle.com/javase/8/docs/api/java/io/File.html. [Accedido: 25-nov-2025].

[10] Oracle, "Class ArrayList," Java Platform, Standard Edition 8 API Specification, 2024. [En línea]. Disponible: https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html. [Accedido: 25-nov-2025].

[11] Oracle, "Class Scanner," Java Platform, Standard Edition 8 API Specification, 2024. [En línea]. Disponible: https://docs.oracle.com/javase/8/docs/api/java/util/Scanner.html. [Accedido: 25-nov-2025].

[12] Oracle, "Class Locale," Java Platform, Standard Edition 8 API Specification, 2024. [En línea]. Disponible: https://docs.oracle.com/javase/8/docs/api/java/util/Locale.html. [Accedido: 25-nov-2025].

[13] The pandas development team, "pandas documentation," PyData, 2024. [En línea]. Disponible: https://pandas.pydata.org/docs/. [Accedido: 25-nov-2025].

[14] Python Software Foundation, "time — Time access and conversions," Python 3.12 Documentation, 2024. [En línea]. Disponible: https://docs.python.org/3/library/time.html. [Accedido: 25-nov-2025].

[15] Python Software Foundation, "os — Miscellaneous operating system interfaces," Python 3.12 Documentation, 2024. [En línea]. Disponible: https://docs.python.org/3/library/os.html. [Accedido: 25-nov-2025].

[16] Python Software Foundation, "sys — System-specific parameters and functions," Python 3.12 Documentation, 2024. [En línea]. Disponible: https://docs.python.org/3/library/sys.html. [Accedido: 25-nov-2025].

[17] Python Software Foundation, "math — Mathematical functions," Python 3.12 Documentation, 2024. [En línea]. Disponible: https://docs.python.org/3/library/math.html. [Accedido: 25-nov-2025].

[18] Python Software Foundation, "multiprocessing — Process-based parallelism," Python 3.12 Documentation, 2024. [En línea]. Disponible: https://docs.python.org/3/library/multiprocessing.html. [Accedido: 25-nov-2025].

[19] Python Software Foundation, "locale — Internationalization services," Python 3.12 Documentation, 2024. [En línea]. Disponible: https://docs.python.org/3/library/locale.html. [Accedido: 25-nov-2025].

5 ANEXOS