Informe de Resultados y análisis, Taller Parcial 1, Estructura de Datos 2024-2

Guillermo Mejía Uribe, c.c. 1037643854

Universidad de Antioquia, Facultad de Ingeniería, Ingeniería de Sistemas
Medellín, 2024

guillermo.mejia@udea.edu.co

Introducción

Este informe presenta el desarrollo y análisis de un taller orientado a la implementación y comparación de 15 algoritmos de ordenamiento, utilizando el lenguaje Python y sus librerías en Google Colab. La columna seleccionada para la evaluación corresponde a los datos de "Alzheimer's Disease and Other Dementias", extraída de un conjunto de datos más amplio relacionado con las causas de muerte en diferentes regiones del mundo.

El objetivo principal del taller fue implementar cada uno de los algoritmos en su versión descendente, asegurando que los resultados sean coherentes y no afecten el conjunto de datos original. Entre los métodos aplicados se incluyen tanto algoritmos clásicos como Bubble Sort, Quick Sort, Merge Sort, así como el **método exclusivo Counting Sort**, que fue evaluado en profundidad por ser un algoritmo determinista y eficiente en ciertas circunstancias.

Para medir el rendimiento de los algoritmos, se realizaron pruebas con diferentes tamaños de muestras, desde 2000 hasta 6120 registros. Cada prueba registró los tiempos de ejecución de cada método y generó gráficas comparativas para visualizar los resultados de manera más clara. Los resultados obtenidos buscan proporcionar un entendimiento detallado del comportamiento y eficiencia de los algoritmos en función del tamaño y distribución de los datos.

Este informe también destaca el uso de Python y librerías especializadas como Seaborn, Pandas y Matplotlib para realizar visualizaciones y análisis de datos. Finalmente, se ofrecen conclusiones y recomendaciones basadas en los resultados obtenidos, evaluando los algoritmos más adecuados para este tipo de datos, así como las limitaciones encontradas en su aplicación.

Desarrollo del Taller

1. Lectura e inspección de la data:

La carga de datos se realizó utilizando la API de Kaggle, descargando el archivo directamente desde el repositorio disponible. En caso de fallas en la conexión o problemas con la API, se implementó una alternativa que permite subir manualmente el archivo al entorno de Colab para continuar con las pruebas sin interrupciones. Una vez cargados, los datos se procesaron y se extrajo la columna "Alzheimer's Disease and Other Dementias", que se utilizó como base para las pruebas de ordenamiento.

2. Implementación de algoritmos de ordenamiento:

Se desarrollaron 15 algoritmos de ordenamiento en su versión descendente, asegurando que los datos originales no se modifiquen mediante el uso de copias internas en cada función. Entre los algoritmos implementados se encuentran tanto métodos clásicos como Bubble Sort, Quick Sort, Merge Sort, y el método exclusivo Counting Sort. Todos los algoritmos fueron probados sobre la misma estructura de datos para garantizar la consistencia de los resultados.

3. Pruebas de rendimiento en diferentes tamaños de lista:

Para cada algoritmo, se realizaron pruebas con muestras de 2000, 3000, 4000, 5000 y 6120

registros. Los tiempos de ejecución de cada método fueron registrados para cada tamaño de muestra, generando un DataFrame consolidado que permitió realizar comparaciones detalladas. Esto facilitó el análisis de rendimiento a medida que aumentaba el tamaño de la lista.

4. Visualización de resultados:

Se utilizaron gráficas de barras horizontales en escala logarítmica para comparar los tiempos de ejecución de los algoritmos. Además, se generaron histogramas, boxplots y violin plots para visualizar la distribución de los datos, lo que permitió entender cómo la concentración de valores bajos afectó el rendimiento de algunos algoritmos.

Resultados

En esta sección se presentan algunos de los resultados más relevantes obtenidos durante la experimentación. Entre los elementos destacados se incluyen:

- Tabla de tiempos de ejecución: Comparación del rendimiento de los 15 algoritmos de ordenamiento con diferentes tamaños de muestra.
- Paso a paso de un método de ordenamiento: Se muestra cómo un algoritmo específico reorganiza los datos en cada iteración.
- Gráficos de distribución de datos: Visualizaciones como histogramas y boxplots para entender la naturaleza de la columna "Alzheimer's Disease and Other Dementias".
- Gráfica comparativa de tiempos: Un gráfico en escala logarítmica que permite analizar la eficiencia de los algoritmos en función del tamaño de la muestra.

Para una revisión más detallada de los procedimientos, resultados adicionales y el código implementado, se invita a consultar el notebook completo donde se pueden observar todos los resultados y experimentaciones paso a paso.

```
Pruebas con lista de tamaño 2000:

Algoritmo: Counting Sort, Tiempo: 0.032511 segundos
Algoritmo: Bubble Sort, Tiempo: 0.415349 segundos
Algoritmo: Bubble Sort Mejorado, Tiempo: 0.382643 segundos
Algoritmo: Cocktail Sort, Tiempo: 0.353772 segundos
Algoritmo: Selection Sort, Tiempo: 0.188590 segundos
Algoritmo: Insertion Sort, Tiempo: 0.182726 segundos
Algoritmo: Gnome Sort, Tiempo: 0.542939 segundos
Algoritmo: Shell Sort, Tiempo: 0.008705 segundos
Algoritmo: Merge Sort, Tiempo: 0.010066 segundos
Algoritmo: Quick Sort, Tiempo: 0.007125 segundos
Algoritmo: Radix Sort, Tiempo: 0.005001 segundos
Algoritmo: Weak Heap Sort, Tiempo: 0.009370 segundos
Algoritmo: Binary Insertion Sort, Tiempo: 0.104179 segundos
Algoritmo: Block Sort, Tiempo: 0.005089 segundos
Algoritmo: Pairwise Sorting Network, Tiempo: 0.397578 segundos
```

	Algoritmo	Tamaño de muestra	Tiempo (s)
0	Counting Sort	2000	0.032511
1	Bubble Sort	2000	0.415349
2	Bubble Sort Mejorado	2000	0.382643
3	Cocktail Sort	2000	0.353772
4	Selection Sort	2000	0.188590
5	Insertion Sort	2000	0.182726
6	Gnome Sort	2000	0.542939
7	Shell Sort	2000	0.008705
8	Merge Sort	2000	0.010066
9	Quick Sort	2000	0.007125
10	Radix Sort	2000	0.005001
11	Weak Heap Sort	2000	0.009370
12	Binary Insertion Sort	2000	0.104179
13	Block Sort	2000	0.005089
14	Pairwise Sorting Network	2000	0.397578

Ilustración 1. Resultados de tiempos de ejecución de los 15 métodos de ordenamiento para 2000 registros.

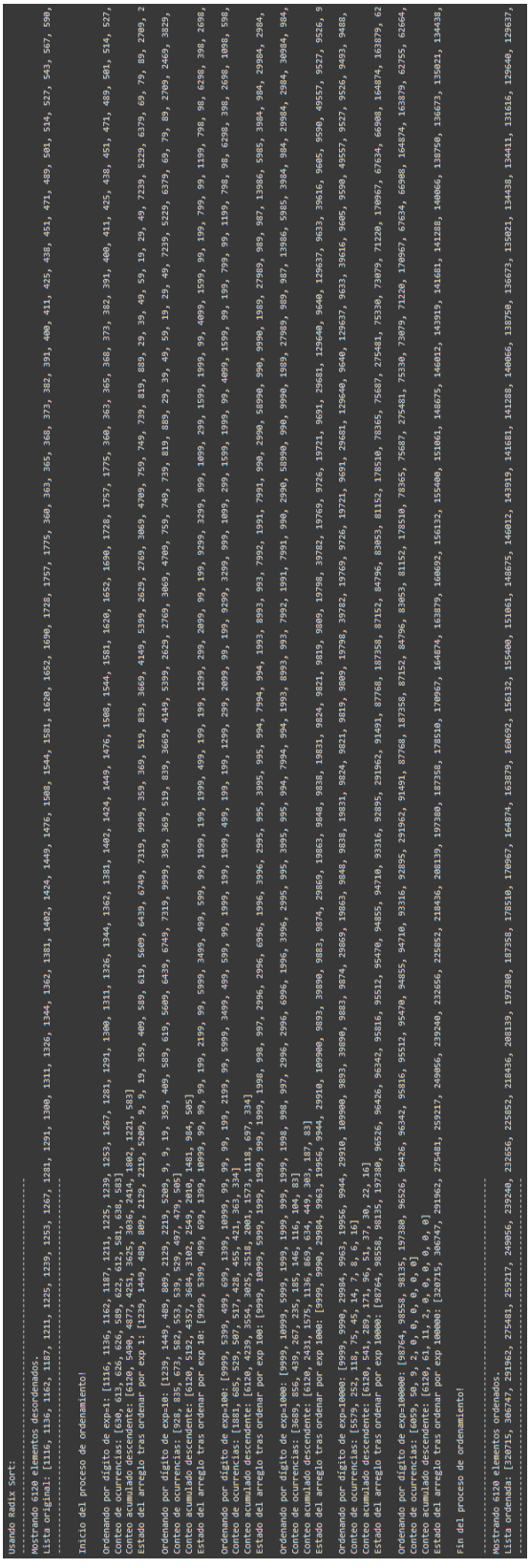

Ilustración 2. Prueba paso a paso del método Radix Sort para 6120 registros.

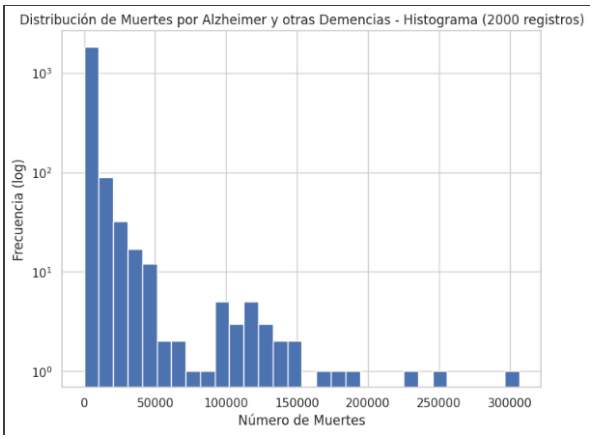

Ilustración 3. Histograma de distribución de datos de 2000 registros.

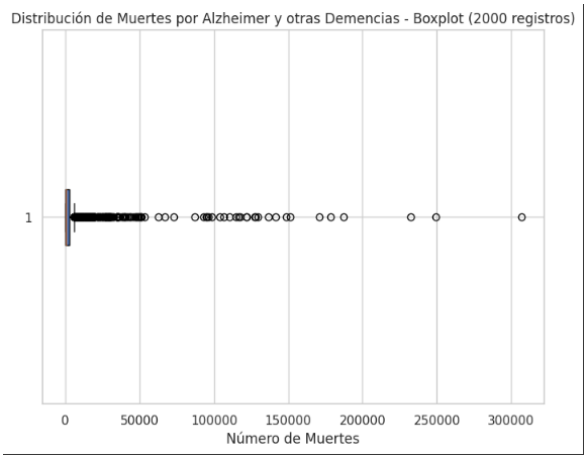

Ilustración 4. Box-Plot de la distribución de datos de 2000 registros.

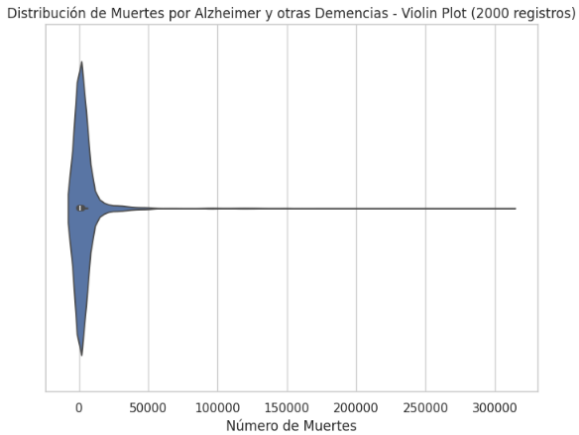

Ilustración 5. Violin-Plot de la distribución de datos de 2000 registros.

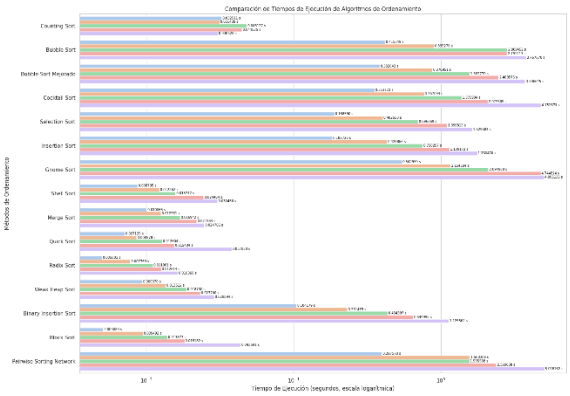

Ilustración 6. Comparación de tiempos de ejecución de los 15 métodos de ordenamiento y los diferentes tamaños de muestra.

Análisis de Resultados

Análisis de los resultados para los 2000 registros

1. **Distribución de los datos.**

   - Histograma (ver ilustración 3): La distribución muestra que la mayoría de los valores están concentrados entre 0 y 50,000, con una pequeña cantidad de valores significativamente más altos, alcanzando hasta 300,000.
   - Boxplot (ver ilustración 4): La presencia de muchos puntos fuera de la caja indica una gran cantidad de valores atípicos, que pueden impactar los algoritmos de ordenamiento.
   - Violin Plot (ver ilustración 5): La concentración de datos en el rango inferior confirma que la distribución está sesgada hacia valores bajos con un pico significativo en valores menores a 50,000.

2. **Impacto de la distribución en los algoritmos de ordenamiento.**

   - Algoritmos como Counting Sort y Radix Sort deberían beneficiarse de la naturaleza numérica de los datos, aunque el amplio rango de valores (0 a 300,000) puede ser un reto para Counting Sort, aumentando el espacio requerido.
   - Algoritmos como Bubble Sort o Gnome Sort pueden verse afectados negativamente, ya que su complejidad $O(n^2)$ hace que sean ineficientes con muestras con valores muy desordenados o con rangos amplios.
   - Quick Sort y Merge Sort, al ser más eficientes en general, no deberían verse tan afectados por la distribución, ya que se desempeñan bien incluso con listas desordenadas.

3. **Resultados de tiempos de ejecución de los 15 métodos de ordenamiento.**

3.1 Método más rápido de los 15:
Radix Sort, con un tiempo de ejecución de 0.005001 segundos. La posible razón por la que este método se destaca como el mejor para la muestra con 2000 registros, es que este método de ordenamiento basado en dígitos en lugar de comparaciones directas entre elementos. Este enfoque permite una complejidad lineal O(n*k). Esta eficiencia se ve reflejada con lo esperado en la distribución de los datos de la lista utilizada.

3.2 Métodos que se destacaron por su rapidez y eficiencia:
Block Sort, Quick Sort, Shell Sort y Merge Sort, se destacan porque son métodos que se adaptan bien a listas desordenadas o amplias distribuciones de valores. Tambien, mantienen una complejidad eficiente (O(n log n)), lo que los hace rápidos y escalables.

3.3 Método mas lento de los 15:
Gnome Sort, con un tiempo de ejecución de 0.542939 segundos. La razón principal de este resultado es su complejidad $O(n^2)$, que lo vuelve

muy ineficaz para listas grandes. Este algoritmo reordena elementos comparando pares consecutivos y retrocediendo si es necesario, lo que lo hace especialmente lento en listas como la de la prueba, que tiene valores muy dispersos o desordenados.

3.4 Métodos que no se destacaron debido a su lentitud y poca eficiencia:
Bubble Sort, Bubble Sort Mejorado, Cocktail Sort, Pairwise Sorting Network, tuvieron muy bajo rendimiento. Comparten una serie de limitaciones, aca se podría mencionar su alta complejidad temporal $O(n^2)$ y que son inadecuados para manejar eficientemente listas grandes o desordenadas.

4. Comparación entre algoritmos estables y no estables.

- Algoritmos estables en el experimento:
  Merge Sort, Insertion Sort, Binary Insertion Sort, Counting Sort, Bubble Sort, Bubble Sort Mejorado.
- Algoritmos no estables en el experimento:
  Quick Sort, Radix Sort, Shell Sort, Selection Sort, Cocktail Sort, Gnome Sort, Weak Heap Sort, Block Sort, Pairwise Sorting Network.

Al observar la ***ilustración 1***, se puede decir que la estabilidad no siempre garantiza un mejor rendimiento, pues métodos no estables como Radix Sort y Quick Sort, superaron en rendimiento a Merge Sort y Counting Sort. Este último, mostro un tiempo competitivo gracias a su enfoque basado en conteno. Los métodos no estables pueden sacrificar características como la estabilidad por mas rapidez en su ejecución.

Si se prioriza la velocidad de ejecución para un conjunto de datos con alrededor de 2000 registros, el cual presenta una dispersión significativa y desorden, Quick Sort y Radix Sort se destacan como métodos eficaces. Ambos algoritmos son eficientes debido a:

- Quick Sort: Aprovecha su estrategia de división y conquista, logrando un desempeño rápido en promedio, especialmente en listas grandes y desordenadas, con una complejidad de O(n log n).
- Radix Sort: Al ser un algoritmo basado en dígitos, funciona particularmente bien en datos numéricos, como la columna analizada. Aunque no es estable, es muy eficiente para distribuciones numéricas amplias.

No obstante, si el objetivo no solo es velocidad, sino también preservar el orden relativo entre registros con el mismo valor (es decir, estabilidad), Merge Sort y Counting Sort ofrecen un equilibrio entre rapidez y estabilidad:

- Merge Sort: Es estable y tiene un rendimiento consistente con O(n log n). Es ideal para conjuntos de datos dispersos y desordenados donde mantener la estabilidad es importante.
- Counting Sort: Aunque está limitado a datos numéricos con rangos conocidos, es sorprendentemente rápido en este contexto con O(n + k), y su estabilidad lo hace adecuado si hay valores repetidos.

Influencia de la dispersión de los datos:

La dispersión observada en la distribución de la muestra sugiere que la mayor parte

de los registros tiene valores relativamente bajos, con algunos valores atípicos que alcanzan cifras altas. Esto afecta a los algoritmos:

- Quick Sort: Funciona muy bien incluso con datos desordenados y dispersos, pero su eficiencia depende del pivote elegido. Con la dispersión presente, puede mantener un buen desempeño en promedio.
- Radix Sort y Counting Sort: Estos algoritmos se benefician de la naturaleza numérica de los datos, pero pueden perder eficiencia si la dispersión se extiende más allá del rango esperado.
- Merge Sort: No se ve afectado por la dispersión de los datos y ofrece un rendimiento consistente gracias a su enfoque recursivo y estable.

Asi, se puede concluir que:

- Para priorizar velocidad: Quick Sort y Radix Sort son opciones ideales para datos dispersos y desordenados, aunque sacrifican estabilidad.
- Para un equilibrio entre velocidad y estabilidad: Merge Sort y Counting Sort ofrecen un desempeño balanceado, preservando el orden relativo de los registros repetidos.

5. Comparación entre algoritmos recursivos e iterativos.

- Algoritmos recursivos en el experimento:
  Quick Sort, Merge Sort, Binary Insertion Sort, Weak Heap Sort.
- Algoritmos iterativos en el experimento:
  Counting Sort, Bubble Sort, Bubble Sort Mejorado, Cocktail Sort, Selection Sort, Insertion Sort, Gnome Sort, Shell Sort, Radix Sort, Block Sort, Pairwire Sorting Network

Al observar la ***ilustración 1***, se encuentra con que tanto algoritmos recursivos como iterativos tuvieron muy buen desempeño en la ejecución de una data con 2000 registros, datos significativamente dispersos y en desorden. Sin embargo, se puede observar que métodos iterativos como Radix Sort y Block Sort, superaron por poco los tiempos de ejecución de métodos recursivos como Quick Sort y Merge Sort. Esto se debe a que los métodos iterativos pueden ofrecer una sólida eficiencia sin tener los gastos adicionales de memoria, que tiene su contra parte, la recursividad.

6. Diferencias en el rendimiento del método exclusivo.

Counting Sort es un algoritmo de ordenamiento que utiliza un enfoque basado en conteo de frecuencias, en lugar de comparaciones directas entre elementos. Esto lo hace particularmente eficiente cuando se conocen los límites de los valores numéricos en la lista, ya que puede contar rápidamente las ocurrencias de cada número y reordenarlos.

En general, con un resultado de 0.032511 segundos de tiempo de ejecución, Counting Sort se ubicó como uno de los métodos de ordenamiento más rápidos en el escenario de 2000 registros. Mostró un desempeño sólido y eficiente para una data con una dispersión significativa y desordenada. Una de las ventajas de este método, es que preserva el orden relativo de los elementos (estable) y que es

especialmente útil para listas numéricas con rangos acotados y valores repetidos.

Análisis del impacto del tamaño de la muestra en los tiempos

Al observar la ***ilustración 6***, se puede identificar patrones importantes en el comportamiento de los métodos de ordenamiento de acuerdo con el tamaño de la muestra y su efecto en los tiempos de ejecución.

a. Patrones generales de escalabilidad.
   - Aumento del tiempo con el tamaño de la muestra:
     La mayoría de los métodos siguen un patrón donde los tiempos aumentan de manera no lineal conforme crece el tamaño de la muestra a ordenar. Esto es consistente con las complejidades temporales esperadas para cada método.

b. Algoritmos con buen rendimiento escalable.
   Los siguientes métodos muestran tiempos bastante eficientes, incluso con las muestras más grandes:
   - Block Sort mantiene tiempos competitivos gracias a su enfoque eficiente en la distribución de bloques.
   - Radix Sort sigue siendo uno de los más rápidos, beneficiándose de datos numéricos.
   - Quick Sort mantiene su ventaja como uno de los más rápidos, especialmente con muestras grandes y desordenadas.

c. Algoritmos con rendimiento inconsistente.
   - Bubble Sort, Gnome Sort, y Cocktail Sort:
     Estos métodos presentan tiempos de ejecución que aumentan rápidamente con el tamaño de la muestra. La gráfica muestra cómo, al llegar a tamaños mayores como 5000 o 6120 elementos, estos algoritmos se vuelven poco prácticos, tomando varios segundos en completarse.
   - Binary Insertion Sort:
     Aunque ofrece tiempos aceptables para muestras más pequeñas, su rendimiento comienza a decaer conforme aumenta el tamaño de la muestra.

d. Conclusiones del análisis del impacto del tamaño de la muestra.
   - Quick Sort y Radix Sort se destacan como los algoritmos más adecuados para manejar listas grandes y dispersas, con tiempos consistentes incluso en muestras mayores a 5000 elementos.
   - Algoritmos simples como Bubble Sort y Gnome Sort presentan un comportamiento ineficiente al manejar muestras grandes, demostrando que no son adecuados para volúmenes significativos de datos.
   - Counting Sort, Block Sort, y Merge Sort ofrecen un equilibrio adecuado entre velocidad y estabilidad, destacándose para casos donde se busca tanto eficiencia como ordenación estable.

   Este análisis nos permite concluir que, conforme crece el tamaño de la muestra, los algoritmos con complejidad $O(n^2)$ se vuelven imprácticos, mientras que los algoritmos con complejidades O(n log n) o mejores mantienen su eficiencia y escalabilidad.

Conclusión General

Este taller ha permitido explorar y comparar de manera práctica la eficiencia y rendimiento de 15 algoritmos de ordenamiento sobre un conjunto de datos significativo. A través del desarrollo en Python y su implementación en Google Colab, se ha logrado analizar tanto los algoritmos clásicos como aquellos menos convencionales, con el propósito de entender mejor su comportamiento frente a datos dispersos y desordenados, como los que representan las cifras de muertes por Alzheimer.

En el proceso se realizaron pruebas detalladas con diferentes tamaños de muestras (2000, 3000, 4000, 5000 y 6120 registros), permitiendo observar cómo el tamaño de la lista afecta directamente el desempeño de cada método de ordenamiento. Los resultados arrojaron conclusiones interesantes:

- Quick Sort y Radix Sort se destacaron como las opciones más rápidas, mostrando un excelente rendimiento incluso con muestras grandes y listas desordenadas. Sin embargo, Quick Sort, al ser no estable, no es ideal si se desea preservar el orden relativo de elementos iguales.
- Merge Sort y Counting Sort resultaron ser buenas opciones para mantener un equilibrio entre velocidad y estabilidad. Estos algoritmos garantizan que los elementos con el mismo valor permanezcan en el mismo orden relativo que tenían en la lista original, lo cual es útil en análisis de datos más complejos.
- Los métodos iterativos como Radix Sort y Block Sort mostraron un buen desempeño gracias a su eficiencia en el uso de memoria y procesamiento secuencial. Por otro lado, los métodos recursivos como Quick Sort y Merge Sort demostraron ser eficaces, pero requieren más cuidado al trabajar con listas más grandes debido a posibles límites de recursión.
- El método exclusivo Counting Sort, aunque muy rápido en este contexto, tiene limitaciones y se recomienda para listas numéricas con un rango de valores bien definido, donde su simplicidad y eficiencia se destacan.
- Métodos con alta complejidad $O(n^2)$, como Bubble Sort y Gnome Sort, mostraron un desempeño pobre para listas grandes. Estos algoritmos son más adecuados para listas pequeñas o ya parcialmente ordenadas, donde el impacto de su complejidad es menor.

Recomendaciones

1. Elección del algoritmo según el tipo de lista:
    - Quick Sort es ideal para listas desordenadas de gran tamaño si no se necesita estabilidad.
    - Radix Sort se recomienda para listas numéricas con muchos elementos, dada su velocidad.
    - Merge Sort y Counting Sort son opciones robustas si se necesita estabilidad en los resultados.

2. Consideración del tamaño de la muestra:
    - Para listas grandes (>5000 elementos), Quick Sort y Radix Sort son los más adecuados por su eficiencia.
    - En listas pequeñas, algunos métodos como Insertion Sort o Shell Sort pueden ser útiles por su sencillez y rapidez en estos escenarios.

3. Uso de métodos estables:
    - Si se requiere preservar el orden relativo de elementos iguales (por ejemplo, al trabajar con registros asociados a varios atributos), Merge Sort es una excelente opción.

4. Evitar métodos ineficientes en listas grandes:
    - Métodos como Bubble Sort y Gnome Sort deben evitarse para listas de gran tamaño, ya que sus tiempos de ejecución aumentan significativamente con el tamaño de la muestra.

En resumen, este taller ha proporcionado una visión profunda y práctica de los algoritmos de ordenamiento, permitiendo entender sus fortalezas y limitaciones bajo diferentes circunstancias. Las decisiones sobre qué método utilizar dependerán tanto del tamaño de la lista como de las necesidades específicas de estabilidad y rapidez.