**ANÁLISIS COMPARATIVO DE ALGORITMOS DE ORDENAMIENTO**

(Bubble sort, Insertion Sort, Merge sort y Quick sort)

**Presentado por**: Luis Geronimo Sacanambuy Correa  
**Docente**: Jairo Hernán Aponte Melo

**Asignatura**: Análisis y Diseño de Algoritmos

**Universidad Nacional de Colombia**

**Sede Manizales**

**Marzo de 2025**

**2024-2**

1. **Introducción**

En el contexto del análisis de eficiencia algorítmica, es fundamental evaluar el comportamiento de distintos algoritmos de ordenamiento bajo diversas condiciones. El problema consiste en determinar cómo afecta el tamaño del conjunto de datos al tiempo de ejecución de los algoritmos Bubble Sort, Insertion Sort, Merge Sort y Quick Sort.

Esto permite identificar cuál algoritmo es más eficiente según la cantidad de datos, lo que resulta crucial para seleccionar la mejor solución en sistemas que procesan grandes volúmenes de información, donde la optimización de recursos y tiempos es esencial.

Para ello, se diseñó un experimento práctico que evalúa el rendimiento de cada algoritmo en datasets de diferentes tamaños, midiendo y comparando sus tiempos de ejecución mediante gráficos y análisis.

1. **Presentación de los algoritmos estudiados**
   1. **Bubble Sort**

Bubble Sort es uno de los algoritmos de ordenamiento más sencillos. Funciona iterando a través de la lista, comparando elementos adyacentes y los intercambia si están en el orden incorrecto. Este proceso se repite hasta que la lista está completamente ordenada.

* Ventajas: Fácil de implementar y entender.
* Desventajas: Es ineficiente para listas grandes debido a su complejidad de tiempo O(n^2).
* Complejidad:

Mejor caso: O(n) (si la lista ya está ordenada).

Peor y caso promedio: O(n^2).

* 1. **Insertion sort**

Insertion Sort ordena una lista construyendo gradualmente una porción ordenada. Toma cada elemento y lo inserta en su posición correcta dentro de la parte ya ordenada de la lista.

* + - Ventajas: Eficiente para listas pequeñas o casi ordenadas.
    - Desventajas: Inadecuado para listas grandes debido a su crecimiento cuadrático.
    - Complejidad:

Mejor caso: O(n).

Peor y caso promedio: O(n^2).

* 1. **Merge Sort**

Merge Sort es un algoritmo basado en el paradigma "divide y vencerás". Divide recursivamente la lista en sublistas hasta que cada una tiene un solo elemento y luego son combinadas de manera ordenada.

* Ventajas: Muy eficiente incluso en grandes volúmenes de datos; su rendimiento es consistente.
* Desventajas: Usa memoria extra debido a las listas auxiliares.
* Complejidad:

Mejor, peor y caso promedio: O (n log n).

* 1. **Quick sort**

Quick Sort también se basa en “divide y vencerás”. Selecciona un "pivote" y reorganiza la lista de forma que todos los elementos menores al pivote quedan a su izquierda y los mayores a su derecha. Luego aplica recursivamente el mismo proceso a las sublistas.

* Ventajas: Muy rápido en la práctica, especialmente en listas grandes.
* Desventajas: En su peor caso puede llegar a O(n2), pero esto es raro si se elige bien el pivote.
* Complejidad:

Mejor y caso promedio: O (n log n).

Peor caso: O(n2).

1. **Descripción de las variables independientes y dependientes**

Variable independiente: El tamaño del dataset (cantidad de elementos a ordenar), con valores de prueba de 100, 500, 1000 y 5000 elementos.

Variable dependiente: El tiempo de ejecución (en segundos) que tarda cada algoritmo en ordenar el dataset.

Constantes:

* Tipos de datos generados (números aleatorios).
* Condiciones del sistema de prueba (misma máquina, mismo entorno de ejecución).
* Número de repeticiones por prueba.

1. **Presentación de los instrumentos de medición y almacenamiento y procesamiento de datos**

Lenguaje de programación: Python 3.

Librerías utilizadas:

* time: para medir tiempos de ejecución.
* random: para la generación de datasets aleatorios.
* pandas: para almacenar los resultados en archivos CSV.
* matplotlib: para graficar los resultados.

Herramientas:

* Visual Studio Code como entorno de desarrollo.
* Terminal de Windows para la ejecución de los scripts.

Archivos de almacenamiento:

* [tiempos\_algoritmos.csv](file:///C:\Users\luges\Desktop\ordenamiento\data\resultados\tiempos.xlsx): contiene los resultados de los tiempos de ejecución.
* [grafica\_resultados.png](file:///C:\Users\luges\Desktop\ordenamiento\graficas\tiempos.png): imagen generada con la comparación gráfica de los algoritmos.

1. **Descripción de los conjuntos de datos (instancias del problema)**

Para este experimento se generaron automáticamente datasets de números enteros aleatorios. Cada dataset varía únicamente en tamaño:

* Dataset de 100 elementos. [dataset\_100](file:///C:\Users\luges\Desktop\ordenamiento\data\datasets\dataset_100.csv)
* Dataset de 500 elementos. [dataset\_500](file:///C:\Users\luges\Desktop\ordenamiento\data\datasets\dataset_500.csv)
* Dataset de 1000 elementos. [dataset\_1000](file:///C:\Users\luges\Desktop\ordenamiento\data\datasets\dataset_1000.csv)
* Dataset de 5000 elementos. [dataset\_5000](file:///C:\Users\luges\Desktop\ordenamiento\data\datasets\dataset_5000.csv)

Cada dataset fue ordenado por los cuatro algoritmos para medir y comparar su desempeño.

1. **Descripción de la forma como se ejecutaron los experimentos (scripts)**

Para llevar a cabo la medición del desempeño de los algoritmos de ordenamiento seleccionados (Bubble Sort, Insertion Sort, Merge Sort y Quick Sort), se diseñaron y ejecutaron una serie de experimentos automáticos a través de scripts

programados en Python. A continuación, se describe de manera detallada cómo se estructuró y ejecutó el proceso experimental:

* 1. **Estructura general del experimento**

Los experimentos se organizaron mediante dos scripts principales:

* **ejecutar\_experimentos.py**: Encargado de la generación de los datasets, ejecución de los algoritmos de ordenamiento y registro de los tiempos de ejecución.
* **graficar\_resultados.py**: Responsable de la carga de los resultados y la visualización gráfica del rendimiento de los algoritmos.

Ambos scripts se apoyan en un archivo adicional llamado **algoritmos.py**, donde están implementados los cuatro algoritmos de ordenamiento evaluados.

* 1. **Proceso de ejecución**

El proceso experimental se llevó a cabo mediante los siguientes pasos:

**Paso 1: Generación de datasets**

Para cada uno de los tamaños de entrada seleccionados (100, 500, 1000 y 5000 elementos), se generó automáticamente un conjunto de datos con números aleatorios entre 0 y 10,000 usando la función np.random.randint(). Estos datos se almacenan en archivos .csv dentro del directorio /data/datasets/.

**Paso 2: Ejecución de los algoritmos**

Para cada tamaño de dataset, los cuatro algoritmos fueron ejecutados de manera independiente sobre una copia del mismo conjunto de datos. De esta manera, se garantiza que todos los algoritmos procesan exactamente la misma entrada, asegurando condiciones justas de comparación.

**Paso 3: Medición de tiempos**

El tiempo de ejecución de cada algoritmo se midió utilizando la función time.time(), registrando el instante inicial antes de iniciar el algoritmo y el instante final tras su conclusión. La diferencia entre estos dos valores proporciona el tiempo total que tardó el algoritmo en ordenar el dataset correspondiente.

**Paso 4: Registro de resultados**

Cada ejecución genera un registro con los siguientes datos:

* Nombre del algoritmo.
* Tamaño del dataset.
* Tiempo de ejecución en segundos.

Estos resultados se almacenan en un archivo CSV llamado tiempos.csv, ubicado en el directorio /data/resultados/. El archivo se organiza de manera estructurada para facilitar su análisis posterior y evitar desorganización de los datos.

**Paso 5: Generación de gráfica comparativa**

Con los resultados obtenidos, el script graficar\_resultados.py lee el archivo tiempos.csv y genera una gráfica comparativa que permite observar el comportamiento de los algoritmos en función del tamaño del dataset y su tiempo de ejecución. La gráfica se guarda automáticamente como una imagen (tiempos.png) en el mismo directorio de resultados.

* 1. **Requisitos técnicos**

Para ejecutar correctamente los experimentos se utilizaron las siguientes herramientas:

* Lenguaje de programación: Python 3.11.
* Librerías:
* numpy para la generación de datos aleatorios.
* pandas para la manipulación y almacenamiento de datos.
* matplotlib para la generación de la gráfica.
* time para la medición de los tiempos de ejecución.
  1. **Ejecución de los scripts**
* Para realizar los experimentos y generar los tiempos de ejecución, se utiliza el siguiente comando en la terminal:
* python ejecutar\_experimentos.py
* Para generar y visualizar la gráfica con los resultados, se ejecuta:
* python graficar\_resultados.py

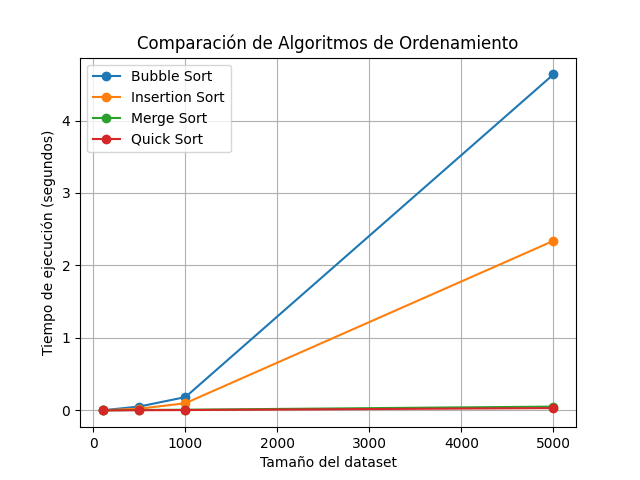
Con estos procedimientos, se garantizó la repetibilidad, organización y precisión de los resultados, permitiendo un análisis confiable del comportamiento de los algoritmos de ordenamiento bajo diferentes volúmenes de datos.

1. **Análisis de resultados**

Según los tiempos obtenidos y la gráfica generada:

* **Bubble Sort** mostró el peor rendimiento, especialmente con datasets grandes (5000 elementos), debido a su complejidad cuadrática O(n^2).
* **Insertion Sort** también presentó un crecimiento significativo en tiempos a medida que aumentaba el tamaño del dataset, aunque fue mejor que Bubble Sort.
* **Merge Sort** y **Quick Sort** mostraron tiempos muy bajos y estables incluso con grandes volúmenes de datos, gracias a su complejidad O (n log n).

La gráfica refleja cómo los algoritmos cuadráticos (Bubble e Insertion) escalan mal con más datos, mientras que los algoritmos más eficientes (Merge y Quick) mantienen tiempos de ejecución controlados.



* 1. **Análisis basado en la gráfica**
* Bubble Sort (línea azul):
* Muestra un crecimiento exponencial del tiempo de ejecución a medida que aumenta el tamaño del dataset.
* Es consistente con su complejidad O(n^2), lo cual explica por qué con 5000 elementos tarda más de 4 segundos.
* Este comportamiento es el esperado: es un algoritmo ineficiente para grandes volúmenes de datos.
* Insertion Sort (línea naranja):
* También presenta un incremento notable del tiempo, aunque menor que Bubble Sort, pero igualmente con tendencia cuadrática (O(n^2)).
* Para 5000 elementos supera los 2 segundos, lo que sigue siendo costoso y coherente para este tipo de algoritmo.
* Merge Sort (línea verde) y Quick Sort (línea roja):
* Prácticamente se mantienen con tiempos muy bajos incluso para 5000 elementos, lo cual es lo esperado.
* Ambos algoritmos tienen una complejidad de O (n log n), mucho más eficiente que los anteriores, y por eso sus tiempos son casi planos en comparación.

|  |  |  |
| --- | --- | --- |
| **Algoritmo** | **Complejidad** | **Comportamiento en la grafica** |
| Bubble sort | O(n^2) | Crecimiento rápido y elevado en tiempo. |
| Insertion sort | O(n^2) | Similar a Bubble Sort, pero levemente mejor. |
| Merge sort | O (n log n) | Tiempo bajo y estable. |
| Quick sort | O (n log n) | Tiempo bajo y estable. |

1. **Conclusiones**

A partir del estudio experimental realizado sobre los algoritmos de ordenamiento, se concluye que:

* Bubble Sort y Insertion Sort son algoritmos poco eficientes para conjuntos de datos medianos y grandes debido a su complejidad O(n^2), lo que provoca un crecimiento exponencial del tiempo de ejecución conforme aumenta la cantidad de datos.
* Merge Sort y Quick Sort ofrecen un rendimiento óptimo y estable incluso para datasets grandes, lo que los convierte en opciones preferidas para tareas de ordenamiento en sistemas reales.
* Aunque Quick Sort es generalmente más rápido en la práctica, en este experimento ambos (Merge Sort y Quick Sort) tuvieron tiempos muy similares, demostrando su eficacia en términos de tiempo de ejecución.
* La importancia de seleccionar el algoritmo adecuado radica en el contexto y tamaño de los datos: para volúmenes pequeños, cualquier algoritmo puede ser válido; sin embargo, para volúmenes grandes, la elección de un algoritmo eficiente es crítica para optimizar recursos y tiempos.
* Para destacar:
* Eficiencia: Merge Sort y Quick Sort son notablemente superiores para grandes datasets.
* Escalabilidad: Bubble Sort e Insertion Sort se vuelven inviables para datasets grandes.
* Validación: Los resultados experimentales validan la teoría de las complejidades de los algoritmos.
* Recomendación: Para datasets pequeños, cualquier algoritmo podría funcionar, pero para tamaños mayores a 1000 elementos, Merge Sort y Quick Sort son la mejor elección.

1. **Enlaces**

[Script\_algoritmos](file:///C:\Users\luges\Desktop\ordenamiento\scripts\algoritmos.py)

[Ejecutar\_experimentos](file:///C:\Users\luges\Desktop\ordenamiento\scripts\ejecutar_experimentos.py)

[Graficar\_resultados](file:///C:\Users\luges\Desktop\ordenamiento\scripts\graficar_resultados.py)

1. **Referencias**

* Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.
* Weiss, M. A. (2014). Data Structures and Algorithm Analysis in Python. Pearson Education.