



# Estructuras de Datos: Proyecto Final

Diego Villarreal (AL007064821) y Ricardo Delgado (AL007050780)

Profesor: Francisco Gómez Rubio



# ¿Qué hace este proyecto?



Competencia en tiempo real

Seis algoritmos de ordenamiento compiten simultáneamente para ordenar el mayor número de colecciones posible.



Tiempo límite

El usuario define cuántos segundos durará la competencia entre algoritmos.



Medición de eficiencia

El sistema registra automáticamente cuántas colecciones ordenó cada algoritmo y genera reportes comparativos.

# Los 6 Algoritmos Implementados



Bubble Sort

Compara elementos adyacentes y los intercambia si están en orden incorrecto.



Merge Sort

Divide el arreglo en mitades y luego las mezcla ordenadamente.



Insertion Sort

Inserta cada elemento en su posición correcta dentro de la parte ya ordenada.



Quick Sort

Usa un elemento "pivot" para dividir y ordenar recursivamente.



Selection Sort

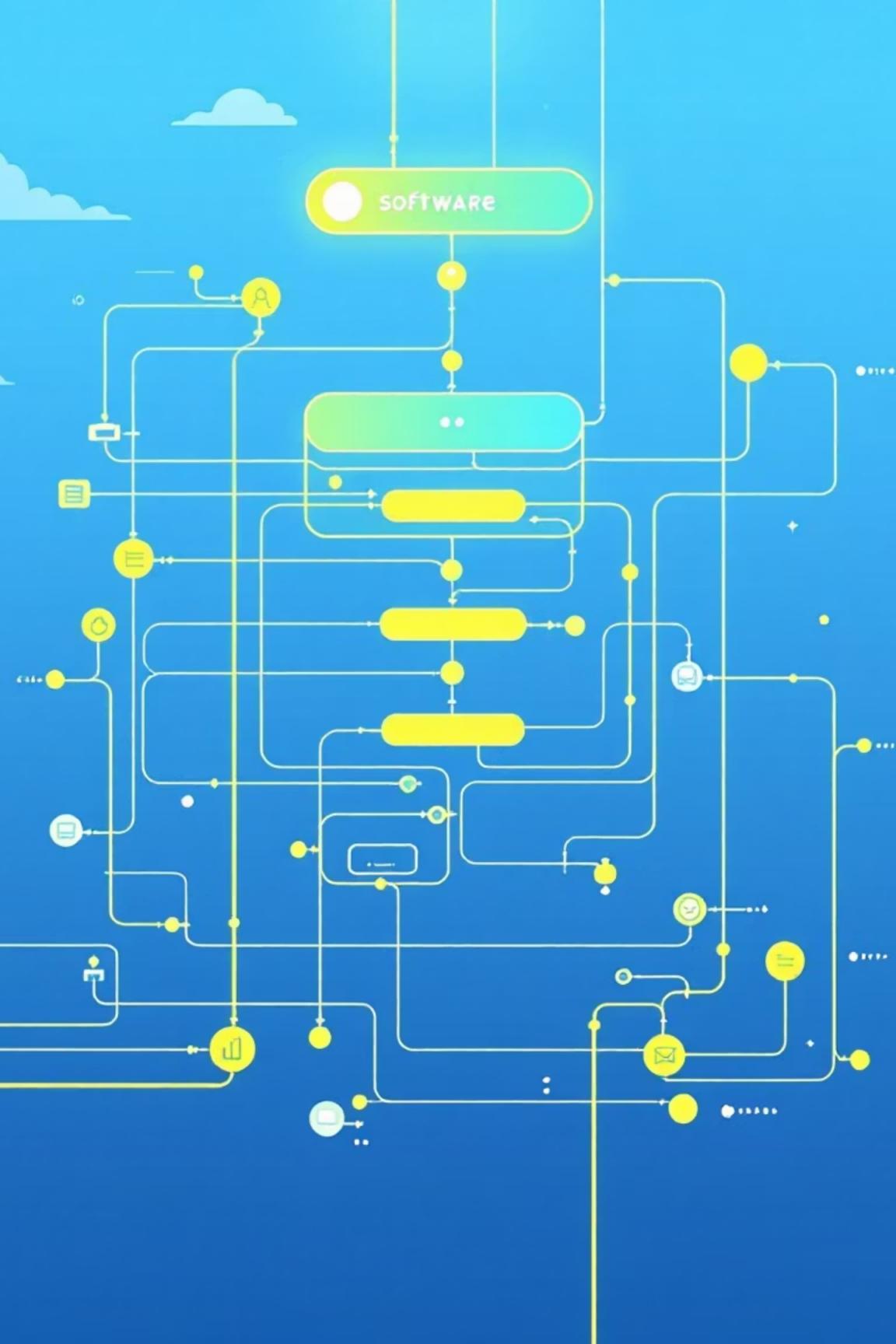
Selecciona el elemento mínimo y lo coloca al inicio del arreglo.



Heap Sort

Construye una estructura heap y extrae elementos en orden.

Todos implementados en **AlgoritmosOrdenamiento.java** sin usar librerías externas de Java.



# Arquitectura del Sistema



Usuario ingresa tiempo

SistemaOrdenamientoConcurrente.java solicita los segundos de ejecución.

Generación de datos

GeneradorColecciones.java crea 4 tipos de colecciones aleatorias usando LCG.

Ejecución concurrente

HiloOrdenamiento.java ejecuta cada algoritmo en su propio hilo independiente.

Análisis de resultados

EstadisticasAlgoritmo.java registra el desempeño de forma thread-safe.

# Resultados de Ejecución: 30 Segundos

1°

Heap Sort

El algoritmo más eficiente, completó todas las colecciones.

2°

Merge Sort

Segundo lugar con rendimiento consistente garantizado.

6°

Quick Sort

El menos eficiente en esta prueba, no completó todas las colecciones.

TADOS DE ORDENAMIENTO CON

a procesar: 4

Ordenadas	Promedio ms
8	4498.12
84	360.69
24	1274.87
5444	5.36
3	9.66
6092	4.77

# Complejidad Computacional

Algoritmo	Mejor Caso	Caso Promedio	Peor Caso
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

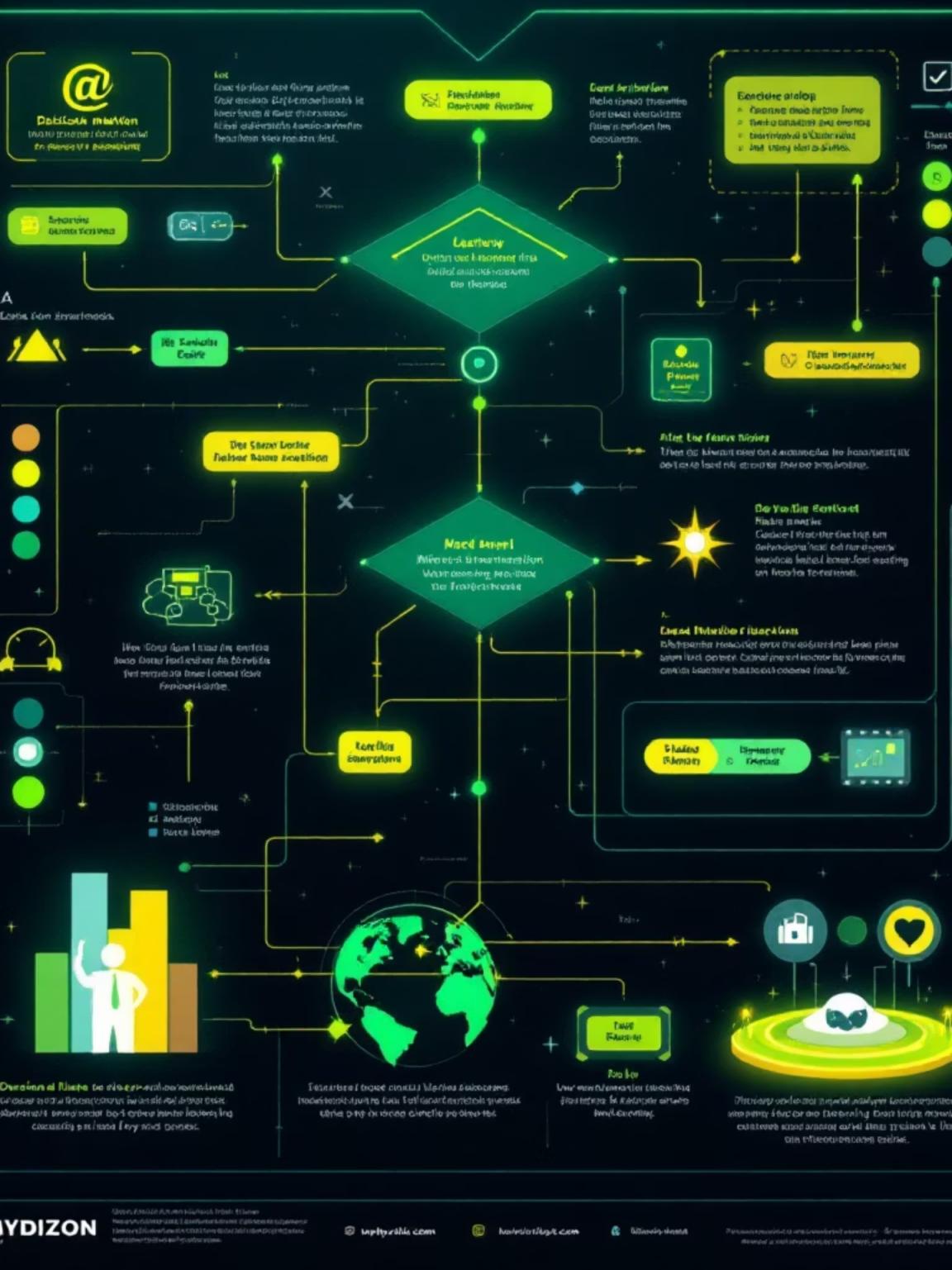
Algoritmos Cuadráticos  $O(n^2)$

Bubble, Insertion y Selection Sort son eficientes solo con datos pequeños o casi ordenados.

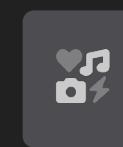
Algoritmos Logarítmicos  $O(n \log n)$

Merge, Quick y Heap Sort manejan grandes volúmenes de datos eficientemente.

## Decision-Making Algorithm Selection Guide



# Recomendaciones de Uso



Datos pequeños o casi ordenados

Usar **Insertion Sort** por su simplicidad y eficiencia en estos casos específicos.



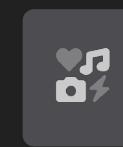
Garantía de rendimiento

Usar **Merge Sort o Heap Sort** cuando se necesita consistencia en todos los escenarios.



Mejor rendimiento promedio

Usar **Quick Sort** para la mayoría de casos prácticos con datos aleatorios.



Mínima memoria adicional

Usar **Heap Sort o Quick Sort** cuando el espacio en memoria es limitado.

# Conclusiones del Equipo

# Ricardo Delgado

Este proyecto me ayudó a entender cómo funcionan realmente los algoritmos al verlos competir simultáneamente. Programar sin `java.util` fue desafiante pero educativo, y aprendí que elegir el algoritmo correcto es crucial para el rendimiento con grandes volúmenes de datos.

# Diego Villarreal

La ejecución concurrente permitió comparar claramente la eficiencia de cada algoritmo. Los resultados muestran que algoritmos avanzados como Quick Sort y Merge Sort superan significativamente a los básicos, demostrando la importancia de la elección correcta del algoritmo.

Código fuente disponible en: <https://github.com/WebbiestStew/proyectorfinal-ed.git>

Gracias