Guía de Actividades Práctico-Experimentales Nro. 006

1. Datos Generales

Asignatura	Estructura de datos
Ciclo	3 A
Unidad	2
Integrantes	Steeven Pardo, Darwin Correa
Resultado de aprendizaje de la unidad	Aplica los métodos de ordenación y búsqueda en la resolución de problemas, bajo los principios de solidaridad, transparencia, responsabilidad y honestidad.
Título de la Práctica	Ordenación básica en Java: Burbuja, Selección e Inserción
Nombre del Docente	Andrés Roberto Navas Castellanos
Fecha	Jueves 20 de noviembre Viernes 21 de noviembre
Horario	07h30 – 10h30 07h30 – 09h30
Lugar	Aula
Tiempo planificado en el Sílabo	5 horas

2. Objetivo(s) de la Práctica:

- Ejecutar y analizar comparativamente los algoritmos de Burbuja, Selección e Inserción sobre casos de prueba, para determinar cuándo conviene cada uno en función de tamaño, grado de orden y duplicados.

3. Materiales y reactivos:

- Guía de pruebas con datasets y salidas esperadas.

4. Equipos y herramientas

- JDK OpenJDK (obligatorio).
- IDE: Visual Studio Code (extensión “Extension Pack for Java”) o IntelliJ IDEA Community.
- Sistema de control de versiones: Git; repositorio en GitHub.

- EVA/Moodle institucional: para entrega de evidencias.
- Herramientas de documentación: README Markdown, editor ofimático (Google Docs/LibreOffice/Word).

5. Procedimiento / Metodología

Enfoque metodológico: ABPr (Aprendizaje Basado en Proyectos).
Inicio

- Presentación del objetivo comparativo y criterios de éxito.
- Formación de equipos (3–4) y revisión de la rúbrica.
- Creación de repo Git.
- Lineamientos de uso responsable de IA.

Desarrollo

- Paso 1. Instrumentación (obligatorio) o Añade contadores a tus algoritmos: comparisons++ al comparar dos claves, swaps++ al intercambiar posiciones.
    - Mide tiempo con `System.nanoTime()` sin imprimir durante la medición (las trazas distorsionan). o Ejecuta R repeticiones por caso (sug.: R=10), descarta las 3 primeras (calentamiento/JIT) y reporta la mediana de tiempo. o Aísla IO: carga CSV fuera de la medición; mide sólo el ordenamiento del array en memoria.
- Paso 2. Casos de prueba oDefine clave de orden (p. ej., `fechaHora` en `citas`, `apellido` en `pacientes`, `stock` en `inventario`). o Convierte a array de la clave (o a registros con `Comparable` por clave).
    - Ejecuta: Insertion, Selection, Bubble (con "corte temprano" en Burbuja).
    - Registra: n, %casi-ordenado, %duplicados, comparisons, swaps, tiempo(ns) (mediana de R-3 corridas).
- Paso 3. Análisis o Tablas comparativas por caso (n, orden, duplicados) y gráficos (tiempo vs. n; tiempo vs. %casi-ordenado).
    - Matriz de recomendación (reglas prácticas):
        - ✦ Casi ordenado + n pequeño/medio → Inserciónngana (menos movimientos).
        - ✦ Muchos duplicados → Inserción tiende a mantener estabilidad útil; Selección hace $n(n-1)/2$ comparaciones siempre, con pocos swaps.
        - ✦ Inverso o aleatorio (n pequeño/educativo) → cualquiera, pero Burbuja penaliza; Selección constante en comparaciones; Inserción peor en inverso pero mejor si detecta localmente orden.

Cierre

- Discusión guiada: ¿Cuándo conviene cada uno? ¿Qué sesgos introdujo la medición?
- Completar README e informe con evidencias y la matriz de recomendación.

6. Resultados esperados:

- Tabla por dataset: `n, tipo (aleat/casi-ord/dup/inverso), algoritmo, comparisons, swaps, tiempo_mediana(ns)`.
- Gráficos (opcional): barras o líneas para tiempo y comparaciones.
- Matriz de recomendación (texto/tabla): "si casi ordenado y $n \leq 500$ → Inserción", "si minimizar swaps → Selección", etc.
- Capturas/Logs de ejecución (sin trazas durante medición).
- Código con instrumentación y scripts de generación de datasets (si aplica).

TABLA: Pacientes por apellido

Algoritmo	Comparisons	Swaps	Observación
BubbleSort	112030	41989	Funciona, pero es lento
InsertionSort	42487	41989	Mucho más rápido
SelectionSort	124750	347	No estable

TABLA: Inventario por Stock

Algoritmo	Comparisons	Swaps	Observación
BubbleSort	124750	124750	Muchos cambios
InsertionSort	124750	124750	Igual que el burbuja
SelectionSort	124750	250	Casi no hizo intercambios, el más recomendado

TABLA: Pacientes por apellido citas_100_casi_ordenadas.csv

Algoritmo	Comparisons	Swaps	Observación
BubbleSort	4650	393	Funciona no realizo muchos intercambios
InsertionSort	492	393	El más estable
SelectionSort	4950	5	No estable

TABLA: Cita por fecha hora citas_100.csv

Algoritmo	Comparisons	Swaps	Observación
BubbleSort	4940	2401	No tan eficiente
InsertionSort	2496	2401	Mas rápido y estable
SelectionSort	4950	93	Comparo mucho, pero cambio poco

GRAFICOS:

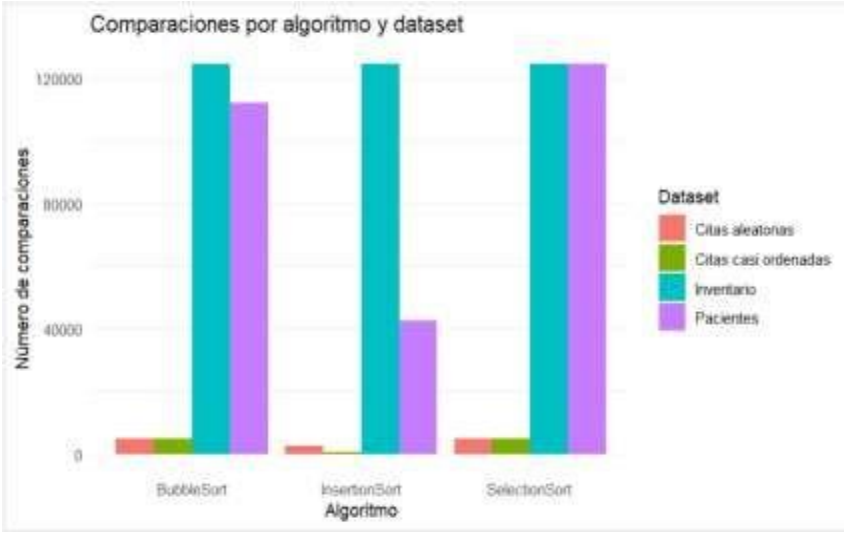

Figura 1: Comparación del número de comparaciones realizadas por cada algoritmo en distintos datasets

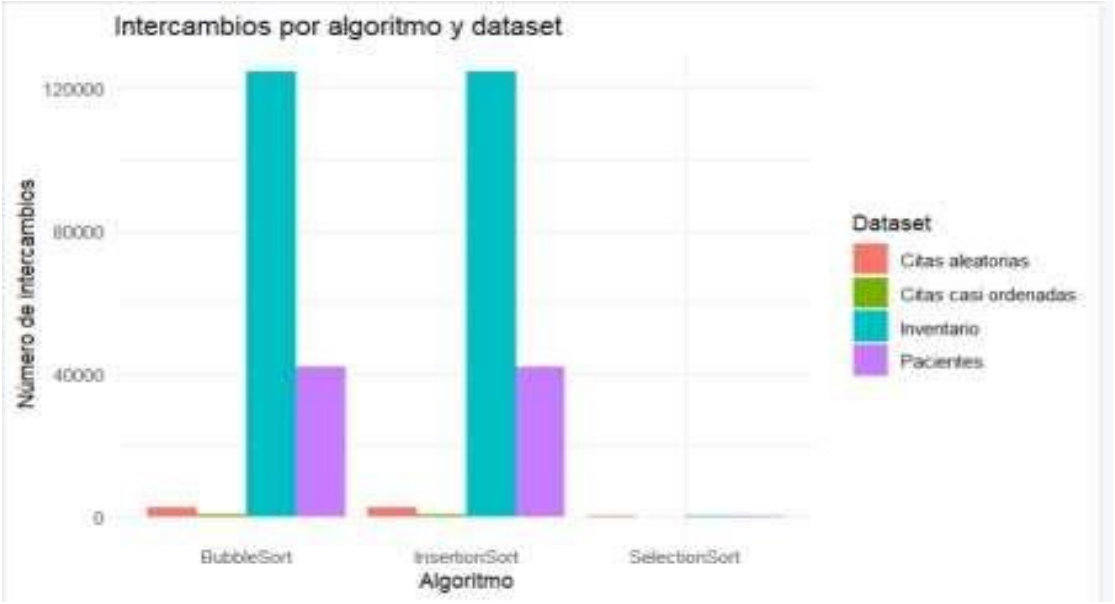

Figura 2: Cantidad de intercambios realizados por cada algoritmo según el tipo de dataset

MATRIZ:

Escenario	Algoritmo recomendado	Justificación breve
Casi ordenado + n pequeño/medio	InsertionSort	Realiza bajas comparaciones, en este caso es eficiente.
Muchos duplicados	InsertionSort	Se encarga de tener un orden relativo entre elementos iguales por lo que es estable.
	SelectionSort	Realiza muchas comparaciones, no es estable su ventaja es que realiza porcos swaps.
Inverso	SelectionSort	En este caso realiza las mismas comparaciones, no toma en cuenta el orden.
	InsertionSort	Mueve mucho los elementos se puede decir que es el peor caso.
Datos aleatorios o sin patrón claro (n pequeño)	BubbleSort	Funciona no es tan eficiente, se lo recomienda si se requiere estabilidad.
	InsertionSort	Este es más rápido que el anterior y también es estable.
	SelectionSort	Tienen comparaciones constantes y realiza pocos swaps. No es estable, pero si eficiente.

n	tipo	algoritmo	comparisons	swaps	tiempo_(ns)
500	Aleatorio (Pacientes)	BubbleSort	112,03	41,989	34,657,100
500	Aleatorio (Pacientes)	InsertionSort	42,487	41,989	10,216,633
500	Aleatorio (Pacientes)	SelectionSort	124,75	347	15,270,233
500	Inverso (Inventario)	BubbleSort	124,75	124,75	1,413,0966
500	Inverso (Inventario)	InsertionSort	124,75	124,75	11,472,966
500	Inverso (Inventario)	SelectionSort	124,75	250	1,534,1500
100	Casi-ord (Citas)	BubbleSort	4,65	393	8,978,033
100	Casi-ord (Citas)	InsertionSort	492	393	574,96
100	Casi-ord (Citas)	SelectionSort	4,95	5	1,366,633

CAPTURAS:

```
---BIENVENIDO---
---TALLER 6: Comparación de Ordenación con Datasets---

=== Pacientes por apellido ===
Tamaño del dataset: 500
BubbleSort -> Comparisons: 112030, Swaps: 41989, Time(ns): 34657100
InsertionSort -> Comparisons: 42487, Swaps: 41989, Time(ns): 10216633
SelectionSort -> Comparisons: 124750, Swaps: 347, Time(ns): 15270233

=== Inventario por stock ===
Tamaño del dataset: 500
BubbleSort -> Comparisons: 124750, Swaps: 124750, Time(ns): 14130966
InsertionSort -> Comparisons: 124750, Swaps: 124750, Time(ns): 11472966
SelectionSort -> Comparisons: 124750, Swaps: 250, Time(ns): 15341500

=== Citas por fechaHora ===
Tamaño del dataset: 100
BubbleSort -> Comparisons: 4650, Swaps: 393, Time(ns): 8978033
InsertionSort -> Comparisons: 492, Swaps: 393, Time(ns): 574966
SelectionSort -> Comparisons: 4950, Swaps: 5, Time(ns): 1366633
```

Figura 3: Resultados de ejecución

Los tiempos de ejecución aparecen como 0 ns debido a la baja cantidad de elementos en los datasets.

```
Iteración 339: [1, 1, 1, 1, 1, 1, 1
Total de intercambios: 41989
Total de comparaciones: 112030
Tiempo(ns): 44959400
```

Figura 4: Trazas de BubbleSort sobre Pacientes por apellido casi ordenados

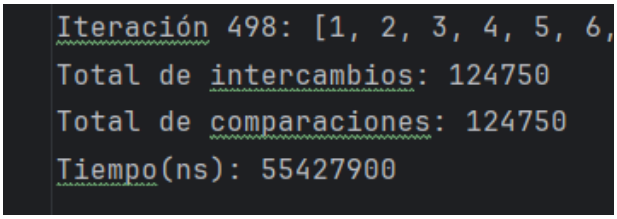

Figura 5: Trazas de BubbleSort sobre Inventario por stock

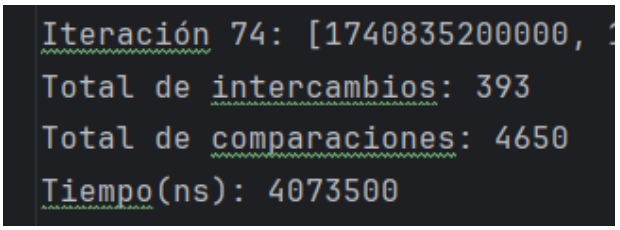

Figura 6: Trazas de BubbleSort sobre Citas casi ordenadas por fechaHora

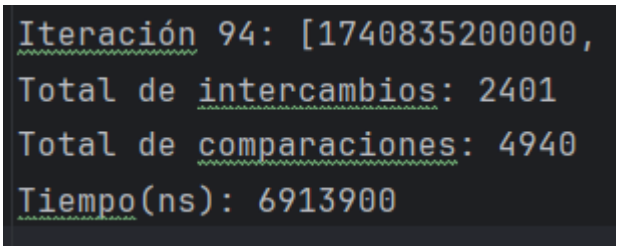

Figura 7: Trazas de BubbleSort sobre Citas por fechaHora

7. Preguntas de Control:

- **¿Por qué imprimir trazas durante la medición distorsiona los tiempos?**
  Porque al momento de que el programa imprima cosas en pantalla, eso demora un tiempo por lo que si medimos cuanto tarda el algoritmo tendríamos que incluir lo que tarde en mostrar los mensajes por ende va a verse más lento de lo que en realidad es.

- **Explica por qué Selección tiene comparaciones ~ n(n−1)/2 sin importar el orden inicial.**
  Porque el algoritmo siempre compara todos los elementos, aunque estos ya se encuentren previamente ordenados, entonces no importa si el arreglo que definimos esta ordenado, al revés o ya se mezclado siempre hace la misma cantidad de comparaciones.

- **¿Por qué Inserción es competitivo en datos casi ordenados?**
  Porque como los datos ya están casi ordenados, el algoritmo no tendría que mover muchos elementos, solo haría unos pocos cambios entonces es más rápido, por lo cual funciona muy bien en esos casos.

- **¿Qué papel juegan los duplicados en la estabilidad del resultado?**
  Los duplicados sirven para ver si nuestro algoritmo se encuentra en el orden original es decir mantiene el orden original de los elementos iguales, entonces si lo hace decimos que este es estable, esto es importante cuando queremos que los datos no se mezclen demás.

- **¿Por qué Burbuja con corte temprano mejora en "casi ordenado" pero no en "inverso"?** Porque si nuestro arreglo ya está casi ordenado, el algoritmo se da cuenta rápido por ende se detiene o se corta, pero en el caso de que esta al revés tienen que hacer todos los pasos igual es ese caso el corte temprano no ayuda.

8. Conclusiones

¿Cuándo conviene cada algoritmo?

- Inserción: siempre que los datos estén casi ordenados o tengan muchos duplicados y necesitemos estabilidad (ej. citas médicas que llegan casi en orden cronológico o listas de pacientes con apellidos repetidos).
- Selección: cuando no conocemos el grado de orden inicial, cuando queremos minimizar intercambios (swaps) o cuando el caso peor es muy desordenado/inverso.
- Burbuja con corte temprano: solo en contextos educativos o cuando n es muy pequeño y los datos están parcialmente ordenados (el corte reduce mucho el trabajo).

¿Qué sesgos introdujo nuestra medición?

Comparaciones y swaps sí son métricas fiables: al ser contadores enteros no sufren sesgo y muestran claramente las diferencias teóricas (124 750 comparaciones fijas en Selección, 492 en Inserción casi ordenado, etc.).

Uso de trazas en una prueba demostrativa: al imprimir iteraciones el tiempo saltó de 0 ns a 4 291 800 ns, confirmando que cualquier salida distorsiona totalmente la medición.

9. Evaluación

Criterio	4 – Excelente	3 – Bueno	2 – Básico	1 – Insuficiente	Pts
Instrumentación (contadores + tiempo)	Corrección y limpieza; medición sin IO/impresiones	Menor detalle	Parcial	No funcional	**2.5**
Diseño experimental	R≥10, descarta 3 corridas, mediana; casos variados	Algún ajuste menor	Parcial	Inadecuado	**2.0**
Ejecución y datos	Tablas completas por dataset	Tablas con huecos	Datos escasos	Sin datos	**2.0**
Análisis y matriz	Conclusiones claras y justificadas	Aceptables	Superficiales	Ausentes	**2.5**
Entrega y código	README/Informe claros; código limpio	Aceptable	Pobre	Deficiente	**1.0**

10. Bibliografía

[1] OpenDSA Project, "Sorting and Searching Modules," Virginia Tech, 2021–2024 (REA con visualizaciones y ejercicios).

[2] P. W. Bible and L. Moser, An Open Guide to Data Structures and Algorithms. PALNI Open Press, 2023.

[3] Oracle, "Java SE 17–21 Documentation: `Arrays`, Collections, and I/O (`java.nio.file`), and benchmarking notes," 2021–2025.

[4] OpenJDK, "JMH – Java Microbenchmark Harness: Samples and Guidance," 2020–2025 (guía práctica de mediciones reproducibles).

11. Elaboración y Aprobación

Elaborado po	Andrés R Navas Castellanos **Docente**	
Revisado po **Solo si es realizado en laboratorios**	Luis Sinche **Técnico Docente**	No Aplica
Aprobado po	Edison L Coronel Romero **Director de Carrera**	