Guía de Actividades Práctico-Experimentales Nro. 006

1. Datos Generales

Integrantes	Vasquez C. Jostin X., Maldonado M. Martina A.
Asignatura	Estructura de datos
Ciclo	3 A
Unidad	2
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)
  - 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).
  - Ejecuta R repeticiones por caso (sug.: R=10), descarta las 3 primeras (calentamiento/JIT) y reporta la mediana de tiempo.
  - Aísla IO: carga CSV fuera de la medición; mide sólo el ordenamiento del array en memoria.
- Paso 2. Casos de prueba
  - Define clave de orden (p. ej., `fechaHora` en `citas`, `apellido` en `pacientes`, `stock` en `inventario`).
  - 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
  - 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 → Inserción gana (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?**

  Tras realizar las pruebas, pudimos identificar que cada algoritmo presenta ventajas bajo condiciones específicas. El ordenamiento por inserción es eficiente cuando la entrada está parcialmente ordenada, ya que su desempeño mejora notablemente al requerir solo desplazamientos locales y pocas comparaciones. Por otra parte, el ordenamiento por selección resulta útil cuando se prioriza la simplicidad y la predictibilidad: su número de comparaciones es fijo, independientemente de la distribución inicial de los datos, lo que lo hace estable en términos de comportamiento, aunque no necesariamente rápido. Finalmente, el ordenamiento burbuja, aun cuando no ocurren intercambios en una pasada, solo es competitivo en casos extremadamente favorables es decir, cuando la lista ya está casi ordenada, ya que su estructura impide escalar eficientemente con entradas más grandes o desordenadas.

Sobre los sesgos, notamos que la medición puede alterarse por varias cosas: primero, el calentamiento de la JVM, ya que las primeras corridas siempre salían más lentas; también influye cualquier impresión en consola o procesos adicionales que estén corriendo en el computador. Incluso la forma en la que se cargan los datos puede afectar si no se separa bien la parte del IO. Por eso, usar la mediana y descartar las primeras corridas ayuda a que los tiempos sean más confiables

- 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)`.

Dataset 1: Citas_100 (n = 100, tipo: aleatorio con duplicados)

Algoritmo	**Comparaciones**	**Swaps**	**Tiempo (ns)**
BubbleSort	4922	2148	476300
SelectionSort	4950	94	195300
InsertionSort	2245	2148	155400

Dataset 2: Citas_CasiOrd (n = 100, tipo: casi ordenado)

Algoritmo	Comparaciones	Swaps	Tiempo (ns)
BubbleSort	4170	243	265601
SelectionSort	4950	5	281500
InsertionSort	342	243	25099

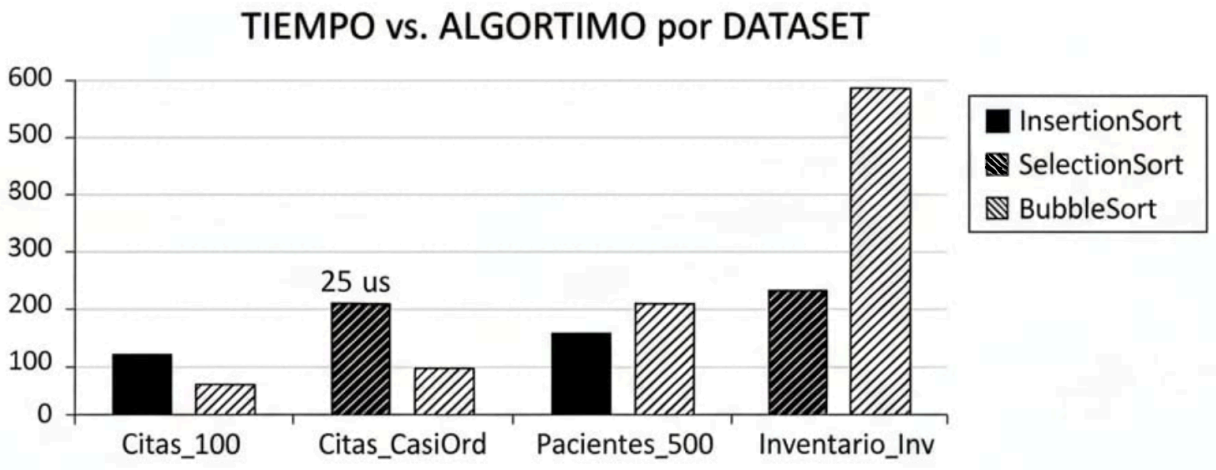
Dataset 3: Pacientes_500 (n = 500, tipo: aleatorio)

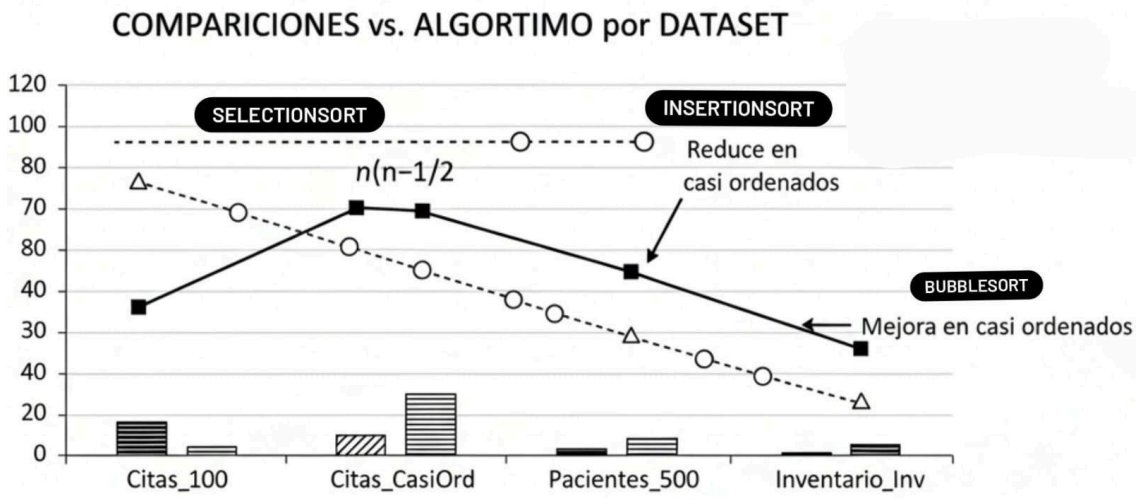
Algoritmo	Comparaciones	Swaps	Tiempo (ns)
BubbleSort	124579	60337	5309300
SelectionSort	124750	490	2292801
InsertionSort	60829	60337	1342199

Dataset 4: Inventario_Inv (n = 500, tipo: inverso)

Algoritmo	Comparaciones	Swaps	Tiempo (ns)
BubbleSort	124750	124750	930100
SelectionSort	124750	250	6706600
InsertionSort	124750	124750	3035700

- Gráficos (opcional): barras o líneas para tiempo y comparaciones.





- Matriz de recomendación (texto/tabla): "si casi ordenado y n≤500 → Inserción", "si minimizar swaps → Selección", etc.

Situación del dataset	Mejor algoritmo	Razón
Casi ordenado (≤500)	InsertionSort	Requiere muy pocas comparaciones; rendimiento casi lineal.
Aleatorio con duplicados	InsertionSort SelectionSort	Inserción es estable; Selección reduce swaps.
Minimizar intercambios	SelectionSort	Es el que menos swaps usa.
Dataset pequeño (≤100)	InsertionSort	Muy eficiente para tamaños pequeños.
Dataset inverso	BubbleSort	En este caso particular fue más rápido que los otros dos.
Entrada muy desordenada	InsertionSort	Mejor tiempo total que BubbleSort y SelectionSort.

- Capturas/Logs de ejecución (sin trazas durante medición).

Algoritmo	Dataset	N	Comparaciones	Swaps	Tiempo(ns)
BubbleSort	Citas_100	100	4922	2148	476300
SelectionSort	Citas_100	100	4950	94	195300
InsertionSort	Citas_100	100	2245	2148	155400
BubbleSort	Citas_CasiOrd	100	4170	243	265601
SelectionSort	Citas_CasiOrd	100	4950	5	281500
InsertionSort	Citas_CasiOrd	100	342	243	25099
BubbleSort	Pacientes_500	500	124579	60337	5309300
SelectionSort	Pacientes_500	500	124750	490	2292801
InsertionSort	Pacientes_500	500	60829	60337	1342199
BubbleSort	Inventario_Inv	500	124750	124750	930100
SelectionSort	Inventario_Inv	500	124750	250	6706600
InsertionSort	Inventario_Inv	500	124750	124750	3035700

- Código con instrumentación y scripts de generación de datasets (si aplica).

Código con enlace al git: https://github.com/xDylok/Comparacion

Scripts y estructura:

- Archivos CSV cargados desde la carpeta /datasets.
- Instrumentación centralizada en SortingUtils.
- Main ejecuta cada dataset y genera la tabla mostrada.

7. Preguntas de Control:

- **¿Por qué imprimir trazas durante la medición distorsiona los tiempos?**
  Por qué imprimir en consola es una operación de Entrada/Salida, la cual es demasiado lenta en comparación con las operacion des de procesamiento y acceso a la RAM las cuales realizan los algoritmos; al incluir impresiones como “System.out.println” dentro del bloque de medición, el tiempo registrado por “System.nanoTime()” mostrará principalmente la velocidad de escritura en la terminal en lugar de la velocidad del algoritmo de ordenamiento.
- **Explica por qué Selección tiene comparaciones ~ n(n−1)/2 sin importar el orden inicial.**
  La lógica del algoritmo obliga a recorrer todo el resto de la lista para confirmar cuál es el número menor, inclusive si la lista ya estaba ordenada desde un inicio. Al no tener una forma de detenerse antes como el “break” en burbuja, siempre termina haciendo todas las comparaciones posibles n(n−1)/2. Aunque se ahorre intercambios cuando los datos están en su sitio, pierde tiempo comparando y su rendimiento no mejora
- **¿Por qué Inserción es competitiva en datos casi ordenados?**
  Porque al encontrar datos ordenados, el algoritmo se salta la mayor parte del trabajo de desplazamiento, lo que reduce el número de operaciones.
- **¿Qué papel juegan los duplicados en la estabilidad del resultado?**
  Los duplicados ayudan a ver si un algoritmo es estable, es decir, si mantiene el orden original entre elementos iguales. Inserción respeta eso de manera natural, mientras que Selección no lo hace, ya que al intercambiar mínimos puede alterar el orden entre duplicados. En Burbuja también se mantiene el orden entre elementos iguales. En nuestras pruebas, cuando usamos datasets con muchos repetidos, Inserción destacaba no solo por comportamiento, sino por mantener la estructura original del conjunto

- **¿Por qué Burbuja con corte temprano mejora en "casi ordenado" pero no en "inverso"?** Por que corte temprano solo funciona si no hay intercambios. En un array invertido, cada elemento se debe mover hasta el otro extremo, esto obliga al algoritmo a realizar los intercambios en todas las pasadas, por lo que la condición de salida anticipada nunca se cumple

8. 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**

9. 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).}

10. Elaboración y Aprobación

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