Guía de Actividades Práctico-Experimentales Nro. 006

1. Datos Generales

Asignatura	Estructura de datos
Integrantes	Anthony Yaguana, Juan Calopino
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
Integrantes	Anthony Yaguana, Juan Galopino
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

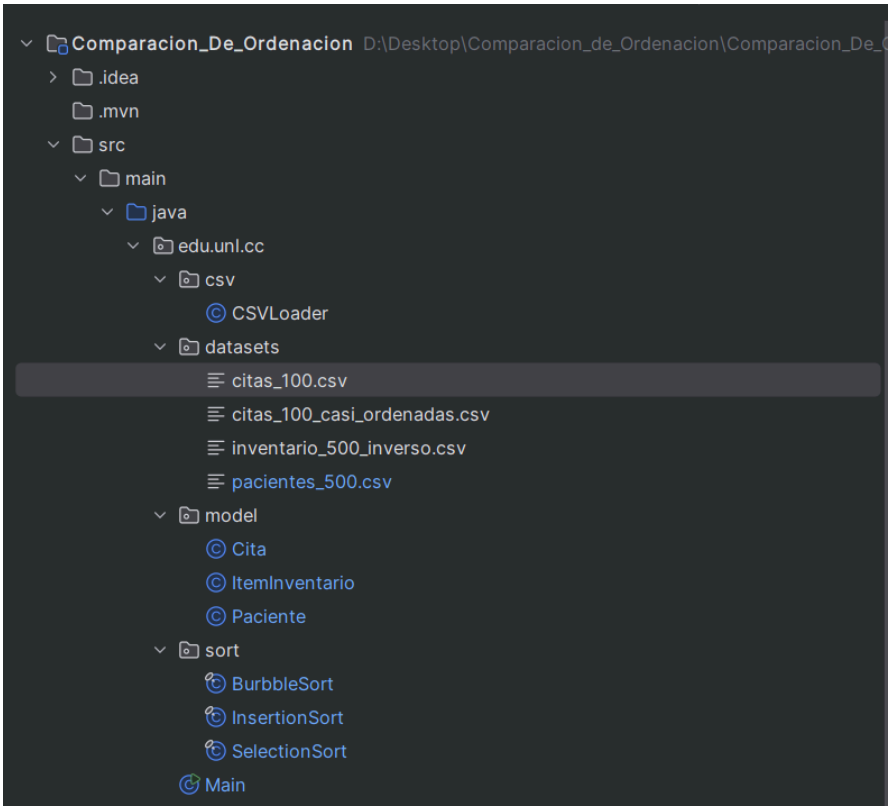
- Paso 1. Instrumentación (obligatorio) o Añade contadores a tus algoritmos: ▪ comparisons++ al comparar dos claves, ▪ swaps++ al intercambiar posiciones.
    - Mide tiempo con `System.nanoTime()` sin imprimir durante la medición (las trazas distorsionan). o Ejecuta R repeticiones por caso (sug.: R=10), 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 o Define 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 → 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?
- Completar README e informe con evidencias y la matriz de recomendación.

6. Resultados esperados:

6.1. Estructura del proyecto :



6.2. Resultados de ejecución:

```
[------ Bienvenido al sistema de comparacion de algoritmos de ordenacion ------]
============================== CITAS ORDENADAS ==============================
SelectionSort - Total de swaps = 96
SelectionSort - Total de comparaciones = 4950
Tiempo de ejecucion de Selection Sort: 2253200 nanosegundos
InsertionSort - Total de swaps = 2641
InsertionSort - Total de comparaciones = 2740
Tiempo de ejecucion de Insertion Sort: 906500 nanosegundos
BubbleSort - Total de pasadas = 99
BubbleSort - Total de swaps = 2542
BubbleSort - Total de comparaciones = 4950
Tiempo de ejecucion de Burble Sort: 1130700 nanosegundos
============================== CITAS CASI ORDENADAS ==============================
SelectionSort - Total de swaps = 5
SelectionSort - Total de comparaciones = 4950
Tiempo de ejecucion de Selection Sort: 273500 nanosegundos
InsertionSort - Total de swaps = 566
InsertionSort - Total de comparaciones = 665
Tiempo de ejecucion de Insertion Sort: 59400 nanosegundos
BubbleSort - Total de pasadas = 99
BubbleSort - Total de swaps = 467
BubbleSort - Total de comparaciones = 4950
Tiempo de ejecucion de Burble Sort: 279200 nanosegundos
```

```
============================== PACIENTES POR PRIORIDAD (Datos repetidos) ==============================
SelectionSort - Total de swaps = 336
SelectionSort - Total de comparaciones = 124750
Tiempo de ejecucion de Selection Sort: 6830100 nanosegundos
InsertionSort - Total de swaps = 43333
InsertionSort - Total de comparaciones = 43832
Tiempo de ejecucion de Insertion Sort: 1963500 nanosegundos
BubbleSort - Total de pasadas = 499
BubbleSort - Total de swaps = 42834
BubbleSort - Total de comparaciones = 124750
Tiempo de ejecucion de Burble Sort: 8169700 nanosegundos
============================== INVENTARIO INVERSO ==============================
SelectionSort - Total de swaps = 250
SelectionSort - Total de comparaciones = 124750
Tiempo de ejecucion de Selection Sort: 6485900 nanosegundos
InsertionSort - Total de swaps = 125249
InsertionSort - Total de comparaciones = 125748
Tiempo de ejecucion de Insertion Sort: 4526800 nanosegundos
BubbleSort - Total de pasadas = 499
BubbleSort - Total de swaps = 124750
BubbleSort - Total de comparaciones = 124750
Tiempo de ejecucion de Burble Sort: 2325800 nanosegundos
```

6.3. TABLAS:

CITAS ORDENADAS				
Algorito	**Comparisons**	**Swaps**	**Tiempo (ns)**	**Observación**
Selection	4950	96	2253200	Menos intercambios, pero muchas comparaciones; tiempo alto
Insertion	2740	2641	906500	Menos comparaciones; eficiente para listas casi ordenadas; tiempo más bajo.
Burbble	4950	2542	1130700	Muchas comparaciones e intercambios; menos eficiente que Insertion.

CITAS CASI ORDENADAS				
Algorito	**Comparisons**	**Swaps**	**Tiempo (ns)**	**Observacion**
Selection	4950	5	273500	Menos intercambios; muchas comparaciones; tiempo moderado
Insertion	665	566	59400	Menos comparaciones; eficiente para listas casi ordenadas; tiempo más bajo.
Burbble	4950	467	8169700	Muchas comparaciones; tiempo muy alto; menos eficiente que Insertion.Muchas comparaciones; tiempo muy alto; menos eficiente que Insertion.

PACIENTES (DATOS REPETIDOS)				
Algorito	**Comparisons**	**Swaps**	**Tiempo (ns)**	**Observacion**
Selection	124750	336	6830100	Menos intercambios; muchas comparaciones; tiempo alto.
Insertion	43382	43333	1963500	Menos comparaciones; eficiente para listas con datos repetidos; tiempo más bajo.
Burbble	124750	42834	4931000	Muchas comparaciones e intercambios; menos eficiente que Insertion.

INVENTARIO (DATOS COMPLETAMENTE INVERTIDOS)				
Algorito	**Comparisons**	**Swaps**	**Tiempo (ns)**	**Observacion**
Selection	124750	250	6485900	Menos intercambios; muchas comparaciones; tiempo alto.
Insertion	125748	125249	4526800	Muchas comparaciones e intercambios; menos eficiente que Bubble
Burbble	124750	124750	2325800	Muchas comparaciones e intercambios; tiempo más bajo en este caso.

Tabla Comparativa en distintos casos :

Caso	**Algoritmo recomendado**	**Argumentación**
Casi ordenado +n pequeño/medio	InsertionSort	Es eficiente para listas casi ordenadas, ya que realiza menos comparaciones e intercambios.
Muchos duplicados	InsertionSort	Maneja bien listas con datos repetidos debido a su simplicidad en intercambios.
Inverso	SelectionSort	Aunque no es el más rápido, es consistente en listas completamente desordenadas.
Datos aleatorio o sin patrón (n pequeño)	BubbleSort	Aunque no es el más rápido, es consistente en listas completamente desordenadas.
Mejor Caso	Insertion Sort	Es el más eficiente cuando la lista está casi ordenada
	BubbleSort	Lista ya ordenada, ya que puede detectar que no hay intercambios necesarios.
	SelectionSort	Lista ya ordenada, pero el número de comparaciones sigue siendo constante.
Peor Caso	BubbleSort	Lista completamente inversa, requiere el máximo número de intercambios.
	InsertionSort	Lista completamente inversa, requiere el máximo número de comparaciones e

		intercambios.
	SelectionSort	Lista completamente desordenada, ya que siempre realiza el mismo número de comparaciones.

7. **Preguntas de Control:**
   - **¿Por qué imprimir trazas durante la medición distorsiona los tiempos?**
     Esto se debe a que imprimir cosas en consola es una operación muy lenta en comparación con las acciones internas del algoritmo, por lo tanto esto altera la medición de tiempo de nuestro algoritmo, haciendo que tome más tiempo de ejecución del que realmente tiene.

   - **Explica por qué Selección tiene comparaciones ~ n(n−1)/2 sin importar el orden inicial.**
     Porque el algoritmo de selección siempre realiza las mismas acciones.
     Para cada posición **i**, busca el mínimo en el subarreglo desde **i+1** hasta n. Y, para encontrar ese mínimo, compara cada elemento restante, aunque ya estén ordenados.
     En términos simples se puede entender que el algoritmo de selección no depende de ningún orden inicial, nunca se detiene antes y no se usa ninguna condición para determinar si el arreglo ya está ordenado.

   - **¿Por qué Inserción es más competitivo en datos casi ordenados?**
     Porque en inserción cada uno de los elementos se coloca moviendo hacia atrás mientras haya datos mayores, por lo que, si los datos ya están casi ordenados el número de desplazamientos que se ejecutarán bajan drásticamente.

   - **¿Qué papel juegan los duplicados en la estabilidad del resultado?**
     Cuando se trata de un algoritmo de ordenación estable los datos duplicados conservaran su orden original por lo tanto sera mas rapido el tiempo de ordenacion, pero, cuando el algoritmo de ordenación no es estable desperdiciara tiempo desplazando los datos que relativamente ya están ordenados así estén duplicados.

- **¿Por qué Burbuja con corte temprano mejora en "casi ordenado" pero no en "inverso"?**

  Porque al tratarse de datos ya ordenados o casi ordenados a la primera pasada se puede determinar que ya están ordenados y así detenerse pronto. Por lo que en una sola pasada o pocas se puede tener todos los datos ordenados.

  En cambio cuando se tratan de datos con orden invertido siempre habrá cambios por lo que crece significativamente el número de pasadas que hará burbuja, por eso el corte temprano solo sirve para datos ordenados o casi ordenados.

8. 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).