Guía de Actividades Práctico-Experimentales Nro. 006

1. Datos Generales

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?
- 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)`.

Tabla 1: Dataset citas_100.csv (Aleatorio)

Algoritmo	Comparaciones	Intercambios (Swaps)	Tiempo Mediana (ns)	Rendimiento
Inserción	2,460	2,362	**92,683**	⭐ Más Rápido
Selección	4,950	94	152,567	1.6x más lento
Burbuja	4,905	2,362	231,854	2.5x más lento

Tabla 2: Dataset citas_100_casi_ordenadas.csv (Casi Ordenado)

Algoritmo	Comparaciones	Intercambios (Swaps)	Tiempo Mediana (ns)	Rendimiento
Inserción	220	122	**7,810**	⭐⭐ Ultrarrápido
Burbuja	764	122	29,131	3.7x más lento
Selección	4,950	86	130,322	16.6x más lento

Tabla 3: Dataset pacientes_500.csv (Muchos Duplicados)

Algoritmo	Comparaciones	Intercambios (Swaps)	Tiempo Mediana (ns)	Rendimiento
Inserción	39,773	39,275	**215,161**	⭐ Más Rápido
Selección	124,750	331	550,773	2.5x más lento
Burbuja	110,215	39,275	638,925	3.0x más lento

Tabla 4: Dataset inventario_500_inverso.csv (Inverso Estricto)

Algoritmo	Comparaciones	Intercambios (Swaps)	Tiempo Mediana (ns)	Rendimiento
Selección	124,750	**250**	**464,296**	⭐ Ganador
Inserción	124,750	124,750	622,222	1.3x más lento
Burbuja	124,750	124,750	719,074	1.5x más lento

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

Escenario detectado	Algoritmo Recomendado	Justificación basada en datos
Datos casi ordenados (Pocas inversiones)	**Inserción**	**Resultados:** 7,810ns vs 130,322ns (Selección). Es imbatible. Al detectar orden local, su complejidad cae drásticamente a casi lineal. Burbuja mejora, pero sigue siendo 3x más lento que Inserción.
Minimizar escrituras (Swaps)	**Selección**	**Resultados:** En el caso inverso, hizo solo **250 swaps** vs 124,750 de los otros. Si se ordena sobre memoria flash (donde cada escritura desgasta el disco) o estructuras de datos grandes, Selección es obligatorio.
Datos totalmente inversos	**Selección**	Fue el único escenario donde Inserción perdió (464k ns vs 622k ns). La sobrecarga de mover cada elemento n posiciones penaliza a Inserción y Burbuja.
Lista pequeña aleatoria o con duplicados	**Inserción**	Mantiene el mejor tiempo promedio (92k ns en aleatorio, 215k ns en duplicados). Además, es **estable** (mantiene el orden de apellidos repetidos), algo que Selección no garantiza.

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

```
Cargando datos...
Datos cargados n: 100 elementos.
Tipo de datos del dataset: Aleatorio
Iniciando benchmark para: Bubble Sort...

--- Resumen: Bubble Sort ---
Tiempo (mediana): 231854 ns (0.231854 ms)
Comparaciones    : 4905
Intercambios     : 2362
------------------------------
Iniciando benchmark para: Insertion Sort...

--- Resumen: Insertion Sort ---
Tiempo (mediana): 92683 ns (0.092683 ms)
Comparaciones    : 2460
Intercambios     : 2362
------------------------------
Iniciando benchmark para: Selection Sort...

--- Resumen: Selection Sort ---
Tiempo (mediana): 152567 ns (0.152567 ms)
Comparaciones    : 4950
Intercambios     : 94
------------------------------
```

```
Cargando datos...
Datos cargados n: 100 elementos.
Tipo de datos del dataset: Casi ordenadas
Iniciando benchmark para: Bubble Sort...

--- Resumen: Bubble Sort ---
Tiempo (mediana): 29131 ns (0.029131 ms)
Comparaciones    : 764
Intercambios     : 122
------------------------------
Iniciando benchmark para: Insertion Sort...

--- Resumen: Insertion Sort ---
Tiempo (mediana): 7810 ns (0.00781 ms)
Comparaciones    : 220
Intercambios     : 122
------------------------------
Iniciando benchmark para: Selection Sort...

--- Resumen: Selection Sort ---
Tiempo (mediana): 130322 ns (0.130322 ms)
Comparaciones    : 4950
Intercambios     : 86
------------------------------
```

```
Cargando datos...
Datos cargados n: 500 elementos.
Tipo de datos del dataset: Inverso
Iniciando benchmark para: Bubble Sort...

--- Resumen: Bubble Sort ---
Tiempo (mediana): 719074 ns (0.719074 ms)
Comparaciones    : 124750
Intercambios     : 124750
------------------------------
Iniciando benchmark para: Insertion Sort...

--- Resumen: Insertion Sort ---
Tiempo (mediana): 622222 ns (0.622222 ms)
Comparaciones    : 124750
Intercambios     : 124750
------------------------------
Iniciando benchmark para: Selection Sort...

--- Resumen: Selection Sort ---
Tiempo (mediana): 464296 ns (0.464296 ms)
Comparaciones    : 124750
Intercambios     : 250
------------------------------
```

```
Cargando datos...
Datos cargados n: 500 elementos.
Tipo de datos del dataset: Duplicados
Iniciando benchmark para: Bubble Sort...

--- Resumen: Bubble Sort ---
Tiempo (mediana): 638925 ns (0.638925 ms)
Comparaciones    : 110215
Intercambios     : 39275
------------------------------
Iniciando benchmark para: Insertion Sort...

--- Resumen: Insertion Sort ---
Tiempo (mediana): 215161 ns (0.215161 ms)
Comparaciones    : 39773
Intercambios     : 39275
------------------------------
Iniciando benchmark para: Selection Sort...

--- Resumen: Selection Sort ---
Tiempo (mediana): 550773 ns (0.550773 ms)
Comparaciones    : 124750
Intercambios     : 331
------------------------------
```

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

7. Preguntas de Control:

1. ¿Por qué imprimir trazas durante la medición distorsiona los tiempos?

Imprimir en consola (System.out.println) es una operación de Entrada/Salida (I/O) extremadamente lenta comparada con las operaciones de CPU y memoria RAM.

- **Razón:** Al imprimir, el programa debe pausar el cálculo puro, interactuar con el buffer del sistema operativo y esperar a que el dispositivo de salida (pantalla) responda.
- **Consecuencia:** Si incluyes un print dentro del bucle de ordenamiento, estás midiendo principalmente la velocidad de tu terminal, no la eficiencia del algoritmo. El tiempo de ordenación (nanosegundos) queda eclipsado por el tiempo de I/O (milisegundos).

2. Explica por qué Selección tiene comparaciones ~ $n(n-1)/2$ sin importar el orden inicial.

El algoritmo de Selección busca el elemento mínimo en el resto de la lista no ordenada para colocarlo en su posición.

- **Lógica:** Para estar 100% seguro de que ha encontrado el mínimo, **debe revisar todos los elementos restantes** uno por uno. No tiene forma de "saber" que el resto ya está ordenado sin mirarlo.
- **Matemática:** El bucle interno siempre se ejecuta desde $j = i+1$ hasta n, sin ninguna condición de ruptura (break). Por tanto, la complejidad de comparaciones es siempre fija ($O(n^2)$), sea el array aleatorio, ordenado o inverso.

3. ¿Por qué Inserción es competitiva en datos casi ordenados?

La Inserción funciona tomando un elemento y comparándolo hacia atrás con los anteriores.

- **Mecanismo:** La condición del bucle interno es mientras (j > 0 y array[j-1] > actual).
- **Ventaja:** Si el dato ya está en orden (array[j-1] <= actual), la condición falla inmediatamente y el bucle interno **no se ejecuta** o se ejecuta una sola vez.
- **Resultado:** En datos casi ordenados, el algoritmo se comporta linealmente ($O(n)$), haciendo muy pocas comparaciones y desplazamientos.

4. ¿Qué papel juegan los duplicados en la estabilidad del resultado?

La estabilidad es la propiedad de un algoritmo de mantener el orden relativo original de registros con claves iguales.

- **Contexto:** En el dataset pacientes_500.csv, hay muchos apellidos repetidos (ej. múltiples "Ramírez").
- **Importancia:** Si usamos un algoritmo estable (como Inserción), un "Ramírez" que estaba antes en la lista original seguirá estando antes que otro "Ramírez" en la lista ordenada. Si usamos uno inestable (como Selección básico), podrían intercambiarse posiciones arbitrariamente, lo cual es indeseable si los datos tenían un sub-orden previo (ej. ordenados por fecha).

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

- **En Casi Ordenado:** La optimización del flag boolean swapped permite que, si en una pasada completa no se hizo ningún intercambio, el algoritmo detecta que ya está ordenado y termina. En un array casi ordenado, esto ocurre muy rápido (quizás en la segunda pasada).
- **En Inverso:** Burbuja mueve los elementos grandes hacia la derecha rápidamente ("conejos"), pero los elementos pequeños que están al final ("tortugas") solo se mueven **una posición a la izquierda por cada pasada completa**.

Resultado: Si el elemento más pequeño está en la última posición (caso inverso), Burbuja necesitará obligatoriamente n pasadas para llevarlo al inicio, haciendo inútil el corte temprano.

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**	Firmado electrónicamente por: ANDRES ROBERTO NAVAS CASTELLANOS Validar únicamente con FirmaEC
Revisado por **Solo si es realizado en laboratorios**	Luis Sinche **Técnico Docente**	No Aplica
Aprobado por	Edison L Coronel Romero **Director de Carrera**	