Guía de Actividades Práctico-Experimentales Nro. 006

1. Datos Generales

Asignatura	Estructura de datos
Ciclo	3 A
Unidad	2
Integrantes	- Cael Alejandro Soto Castillo - Ariel Ismael González Astudillo
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)`.
- 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.
- Capturas/Logs de ejecución (sin trazas durante medición).
- Código con instrumentación y scripts de generación de datasets (si aplica).

TABLA POR DATASET

Citas 100 (Aleatorio)

n	tipo	algoritmo	comparaciones	swaps	tiempo_mediana(ns)
100	aleatorio	BubbleSort	4940	2398	19957800
100	aleatorio	InsertionSort	2493	2398	5219400
100	aleatorio	SelectionSort	4950	90	6315600

Citas 100 Casi Ordenadas

n	tipo	algoritmo	comparaciones	swaps	tiempo_mediana(ns)
100	casi-ordenado	BubbleSort	3519	237	7935300
100	casi-ordenado	InsertionSort	336	237	434500
100	casi-ordenado	SelectionSort	4950	5	5269700

Pacientes 500 (Duplicados)

n	tipo	algoritmo	comparaciones	swaps	tiempo_mediana(ns)
500	duplicados	BubbleSort	109525	41812	5100200
500	duplicados	InsertionSort	42311	41812	1218700
500	duplicados	SelectionSort	124750	327	2824400

Inventario 500 Inverso

n	tipo	algoritmo	comparaciones	swaps	tiempo_mediana(ns)
500	inverso	BubbleSort	124750	124750	3019100
500	inverso	InsertionSort	124750	124750	1384000
500	inverso	SelectionSort	124750	250	981400

Matriz de recomendación (Tabla)

Condición según dataset	Algoritmo recomendado	Motivo
Si el dataset está casi ordenado (ej: citas_100_casi)	**Insertion Sort**	Si el arreglo llega casi ordenado → Inserción reduce comparaciones 336 vs 3519 (Bubble) y 4950 (Selection).
Si n ≤ 500 y el contenido es mayormente aleatorio	**Insertion Sort**	Si el tamaño es moderado y el orden es aleatorio → Inserción usa ~50% comparaciones que Bubble y Selection (citas_100: 2493 vs 4940/4950).
Si se desea minimizar swaps	**Selection Sort**	Si los swaps son costosos → Selección realiza muy pocos (p. ej., 90 y 5 y 327 y 250), mientras que Bubble/Inserción pasan de 20k–120k.

Si los datos tienen muchos duplicados (pacientes_500)	**Insertion Sort**	Si hay gran cantidad de claves repetidas → Inserción evita comparaciones innecesarias y supera por mucho a Bubble/Selection (42311 vs 100k+).
Si el dataset está completamente en orden inverso (inventario_500_inverso)	**Selection Sort**	Si el orden es inverso → Selección mantiene swaps mínimos (250) mientras Bubble/Inserción hacen 124750.
Si se requiere observar sensibilidad al orden inicial (uso didáctico)	**Bubble Sort**	Si interesa analizar efectos del orden → Bubble muestra variación fuerte entre casos (pasa de 4940 comp. a 3519 a 109525).

Capturas/Logs de ejecución

- Generar Datasets (DatasetGenerator):

```
Dataset generado: datasets\citas_100.csv
Dataset generado: datasets\citas_100_casi_ordenadas.csv
Dataset generado: datasets\pacientes_500.csv
Dataset generado: datasets\inventario_500_inverso.csv
```

- Ejecución de los métodos en cada dataset generado (MainRunner):

```
=== Citas 100 ===
BubbleSort | citas_100 | appointments | N=100 | comp=4940 | swaps=2398 | time=17492100ns
InsertionSort | citas_100 | appointments | N=100 | comp=2493 | swaps=2398 | time=2691500ns
SelectionSort | citas_100 | appointments | N=100 | comp=4950 | swaps=90 | time=3352300ns

=== Citas 100 Casi Ordenadas ===
BubbleSort | citas_100_casi | appointments | N=100 | comp=3519 | swaps=237 | time=4229600ns
InsertionSort | citas_100_casi | appointments | N=100 | comp=336 | swaps=237 | time=235200ns
SelectionSort | citas_100_casi | appointments | N=100 | comp=4950 | swaps=5 | time=2563900ns

=== Pacientes 500 ===
BubbleSort | pacientes_500 | patients | N=500 | comp=109525 | swaps=41812 | time=3418400ns
InsertionSort | pacientes_500 | patients | N=500 | comp=42311 | swaps=41812 | time=657400ns
SelectionSort | pacientes_500 | patients | N=500 | comp=124750 | swaps=327 | time=3373900ns

=== Inventario 500 Inverso ===
BubbleSort | inventario_500 | inventory | N=500 | comp=124750 | swaps=124750 | time=3178700ns
InsertionSort | inventario_500 | inventory | N=500 | comp=124750 | swaps=124750 | time=1280800ns
SelectionSort | inventario_500 | inventory | N=500 | comp=124750 | swaps=250 | time=954400ns
```

Código con instrumentación y scripts de generación de datasets

- Código con instrumentación (llamadas a System.nanoTime())

```
public static <T> BenchmarkResult runBenchmark(String algorithmName, String datasetName, String datasetType, T[] original,  3 usages
                                               KeyExtractor<T, ? extends Comparable<?>> keyExtractor, SortingAlgorithm<T> algorithm) {

    final int R = 10; // Número de repeticiones para la medición.
    long[] times = new long[R];
    long totalComparisons = 0;
    long totalSwaps = 0;

    // Bucle de ejecuciones: corre el algoritmo R veces para obtener promedios.
    for (int i = 0; i < R; i++) {
        T[] copy = copy(original); // Utiliza una copia para no alterar el arreglo original.

        long start = System.nanoTime();
        SortMetrics metrics = algorithm.sort(copy, keyExtractor); // Ejecuta el algoritmo de ordenación.
        long end = System.nanoTime();

        times[i] = end - start;
        totalComparisons += metrics.comparisons(); // Acumula el total de comparaciones.
        totalSwaps += metrics.swaps(); // Acumula el total de intercambios.
    }

    // Descarta las primeras 3 corridas (fase de "calentamiento") y calcula la mediana del tiempo.
    long[] valid = Arrays.copyOfRange(times,  from: 3, R);
    Arrays.sort(valid);
    long median = valid[valid.length / 2];
```

```
        // Retorna el resultado, promediando las métricas de todas las corridas.
        return new BenchmarkResult(algorithmName, datasetName, datasetType, original.length,  comparisons: totalComparisons / R,
                swaps: totalSwaps / R, median);
    }
}
```

- **Generación de datasets: SI APLICA EN NUESTRO TRABAJO**

Ubicación:

- Paquete: datasets
- Clase: DatasetGenerator

7. Preguntas de Control:

- **¿Por qué imprimir trazas durante la medición distorsiona los tiempos?**

  Porque la impresión es una operación de Entrada/Salida (I/O) muy lenta que requiere un costoso cambio de contexto en el sistema operativo, pues el programa se bloquea esperando a que se vacíen los *buffers* de salida y se liberen los *locks* de concurrencia, lo cual agrega latencia artificial.

- **Explica por qué Selección tiene comparaciones ~ n(n−1)/2 sin importar el orden inicial.**
  Debido a que el método de SelectionSort recorre todo el arreglo restante para poder hallar el mínimo, aunque este ya esté en orden.

  En cada iteración i: Se compara el elemento i con todos los elementos desde i+1 hasta n-1

  $$(n-1)+(n-2)+...+1=2n(n-1)$$

- **¿Por qué Inserción es competitivo en datos casi ordenados?**

  Insertion Sort es competitivo porque detecta que la mayoría de los elementos ya están casi en su lugar. Su bucle interno sólo tiene que hacer pocas comparaciones y desplazamientos antes de detenerse rápidamente. Esto hace que su rendimiento práctico pase de ser cuadrático O(n^2) a tener una conducta casi lineal O(n). A diferencia de algoritmos como Selection Sort, se ajusta dinámicamente al orden de partida y evita el trabajo innecesario. Por eso es la mejor opción para manejar datos pre-organizados.

- **¿Qué papel juegan los duplicados en la estabilidad del resultado?**

  Algoritmos como el BubbleSort y el InsertionSort mantienen el orden de los elementos iniciales.

  Algoritmos como el SelectionSort pueden cortar ese orden al realizar intercambios con el mínimo que se encontró.

  Cuando existen bastantes duplicados, la estabilidad es esencial si el orden original tiene algún significado por ejemplo mismos nombres pero diferentes fechas, por eso en pacientes_500 con varios duplicados el método de InsertionSort fue el más rápido y adecuado.

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

  Corte temprano: *si en un recorrido completado no se hizo ningún swap el arreglo ya está ordenado este se detiene.*

  En casi ordenado es demasiado probable que uno de esos recorridos tenga 0 swaps por lo cual este algoritmo termina antes. Por ende, BubbleSort con corte temprano es más veloz. Por otra parte, en inverso en cada recorrido los elementos necesitan un swap, no habrá recorridos con 0 swaps por eso BubbleSort no mejora en el caso inverso.

Conclusiones

- En la mayor parte de los casos, en conjuntos de datos aleatorios o casi ordenados, el más eficaz fue InsertionSort. Sus tiempos fueron constantemente más cortos debido a que utiliza el orden parcial y efectúa comparaciones, lo que disminuye de manera significativa la cantidad de operaciones que se necesitan.

- SelectionSort se comportó de una manera consistente en términos de comparaciones, siempre realizando alrededor de n(n-1)/2 sin tomar en cuenta la secuencia lineal. A pesar de la gran cantidad de comparaciones que efectúa, su beneficio es que reduce los swaps al mínimo, lo cual lo hace una opción preferible cuando los intercambios suelen ser costosos o si se quiere disminuir las operaciones en memoria.

- BubbleSort fue el que tuvo el rendimiento más bajo en general, especialmente en los casos aleatorios, inversos y duplicados, debido al gran número de comparaciones e intercambios. No obstante, el corte temprano le permitió un avance muy significativo en los conjuntos de casi ordenados, en el cual se observa pronto que no se necesitan más intercambios.

- El InsertionSort se destacó en conjuntos duplicados por su velocidad y estabilidad, este mantiene el orden relativo de elementos iguales y evita comparaciones que serían innecesarias. SelectionSort no mantiene el orden porque es inestable, todavía sirve cuando se necesitan pocos intercambios.

- A pesar de contar con comparaciones fijas, SelectionSort demostró ser eficiente en términos de tiempo en el peor caso inverso. Esto se debe a que reduce los intercambios en comparación a InsertionSort y BubbleSort, que tienen que desplazar cada elemento muchas veces debido al orden totalmente invertido.

Link repositorio:
https://github.com/C-ael/Taller-6-Comparacion-de-Ordenacion

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

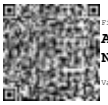
[1] OpenDSA Project, "Sorting and Searching Modules," Virginia Tech, 2021–2024 (REA con visualizaciones y ejercicios).

[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**	