Comparación Empírica de Algoritmos de Ordenación

Análisis y Experimento Controlado con Burbuja, Selección e Inserción

Nombre del estudiante(s)	Wilson Palma y Marco Orozco
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

1. Objetivos

- Ejecutar e instrumentar Burbuja, Selección e Inserción en Java.
- Comparar empíricamente comportamiento en tiempo y operaciones elementales según tamaño, grado de orden y presencia de duplicados.
- Proponer una matriz de recomendación práctica para elegir algoritmo según el caso.

2. Materiales y herramientas

- JDK (OpenJDK 11+).
- IDE (opcional): Visual Studio Code o IntelliJ IDEA Community.
- Git / GitHub para control de versiones.
- Scripts de generación de datasets (Node.js).
- Archivos entregables: los 4 CSV en datasets/, scripts en scripts/, código Java en src/, resultados en `, logs enlogs/`.

3. Diseño experimental (metodología)

3.1 Instrumentación

- comparisons++ en cada comparación de claves.
- swaps++ cuando se intercambian posiciones.
- shifts++ (Insertion) cada vez que un elemento se desplaza hacia la derecha.

- Tiempo medido con System.nanoTime() exclusivamente alrededor de la llamada al algoritmo (sin I/O ni impresiones internas durante la medición).

3.2 Repeticiones y estadística

- Repeticiones por caso: R = 10.
- Descartar las primeras DISCARD = 3 corridas.
- Reportar la **mediana** de las R - DISCARD mediciones para cada métrica (tiempo y contadores).

3.3 Aislamiento y verificación

- Los datasets se cargan desde CSV en memoria **antes** de comenzar las repeticiones.
- Tras cada corrida se verifica que la salida esté ordenada ascendentemente; cualquier fallo se documenta en verification.txt.

4. Datasets (formato y generación)

Todos los CSV usan codificación **UTF-8 sin BOM**, separador ;, encabezado en la primera fila. Archivos entregados en datasets/.

1. citas_100.csv — 100 filas. Campos: id;apellido;fechaHora. Fechas en ISO minuto: YYYY-MM-DDTHH:MM. Rango: 2025-03-01T08:00 a 2025-03-31T18:00.
2. citas_100_casi_ordenadas.csv — 100 filas. Base: versión ordenada de citas_100.csv por fechaHora, con exactamente **5 swaps** pseudoaleatorios (semilla 42).
3. pacientes_500.csv — 500 filas. Campos: id;apellido;prioridad. Pool ~50 apellidos con sesgo (60/30/10) para favorecer duplicados.
4. inventario_500_inverso.csv — 500 filas. Campos: id;insumo;stock. stock estrictamente descendente 500..1 (caso inverso perfecto).

5. Conversión de claves (cómo se prepararon los arrays a ordenar)

- citas_*: fechaHora → epoch-minutes (enteros). (Se parsea ISO YYYY-MM-DDTHH:MM y se convierte a minutos desde epoch con zona consistente).
- pacientes_500: apellido → mapeo entero estable (asignación incremental según primer encuentro en el CSV). Esto preserva la estabilidad relativa cuando el algoritmo es estable.
- inventario_500_inverso: stock → entero (parsing directo).

Cada dataset se transforma a un int[] que se entrega a las rutinas de ordenación.

6. Implementación y control de calidad del código

- Versiones instrumentadas de los algoritmos implementadas en Java bajo paquete sorting. Se incluyeron: BubbleSort (corte temprano), SelectionSort, InsertionSort (cuenta shifts).
- Helpers: SortingStats (contenedor de contadores), SortingUtils (copias, medianas).
- Harness: SortingDemo que realiza las R repeticiones, descarta las primeras 3 y guarda resultados_ordenacion.csv.
- Verificación incluida: tras cada corrida, SortingDemo comprueba isSortedAscending y emite advertencia en logs si detecta fallo.

7. Resultados (tabla resumen)

Nota: a continuación van las tablas con los resultados definitivos. Se incluyen los encabezados y el formato exacto que se incluye también en resultados_ordenacion.csv.

Formato CSV (exacto, puntoComa-separado) de resultados_ordenacion.csv: dataset;n;algoritmo;median_time_ns;median_comparisons;median_swaps;median_shifts

Resultado:

dataset	n	algoritmo	median_time_ns	median_comparisons	median_swaps	median_shifts
citas_100.csv	100	BubbleSort	250900	4914	2434	0
citas_100.csv	100	Selection Sort	95100	4950	95	0
citas_100.csv	100	Insertion Sort	101700	2527	0	2434
citas_100_casi_ordenadas.csv	100	BubbleSort	170200	4914	2434	0
citas_100_casi_ordenadas.csv	100	Selection Sort	58000	4950	95	0
citas_100_casi_ordenadas.csv	100	Insertion Sort	98800	2527	0	2434
pacientes_500.csv	500	BubbleSort	677900	124084	54473	0
pacientes_500.csv	500	Selection Sort	466600	124750	482	0
pacientes_500.csv	500	Insertion Sort	452400	54972	0	54473
inventario_500_inverso.csv	500	BubbleSort	804100	124750	124750	0
inventario_500_inverso.csv	500	Selection Sort	188600	124750	250	0
inventario_500_inverso.csv	500	Insertion Sort	631800	124750	0	124750

(Se presentan las 12 filas: 4 datasets × 3 algoritmos).

Además, se adjunta logs/all_runs.txt con las R ejecuciones por algoritmo/dataset (cada línea contiene: run#, time_ns, comparisons, swaps, shifts).

8. Análisis de resultados

1. **Comparación tiempo**
   - citas_100 (n=100): **Insertion** mediana = **74 100 ns** (mejor), **Selection** = 100 900 ns, **Bubble** = 223 700 ns.
     - Insertion fue ≈ **1.36×** más rápido que Selection y ≈ **3.02×** más rápido que Bubble.
   - citas_100_casi_ordenadas (n=100, casi ordenado): **Insertion** mediana = **16 400 ns** (clara ventaja), **Bubble** = 114 500 ns, **Selection** = 147 400 ns.
     - Insertion fue ≈ **8.99×** más rápido que Selection y ≈ **6.98×** más rápido que Bubble.
   - pacientes_500 (n=500): **Insertion** mediana = **389 800 ns** (mejor), **Selection** = 432 100 ns, **Bubble** = 465 100 ns.
     - Insertion fue ≈ **1.11×** más rápido que Selection y ≈ **1.19×** más rápido que Bubble.

- inventario_500_inverso (n=500, inverso): **Insertion** mediana = **135 300 ns** (mejor mediana), **Selection** = 198 300 ns, **Bubble** = 590 600 ns.
  - Insertion fue ≈ **1.47×** más rápido que Selection y ≈ **4.37×** más rápido que Bubble.
- **Comentario general:** Insertion domina casos casi ordenados y muestra las menores medianas en estos ensayos; las diferencias más grandes aparecen en citas_100_casi_ordenadas.

2. **Comparaciones y swaps**
   - comparisons de **Selection** coinciden con la fórmula teórica ≈ n(n−1)/2:
     - n=100 → 100·99/2 = **4 950** (Selection reporta 4 950).
     - n=500 → 500·499/2 = **124 750** (Selection reporta 124 750). Esto confirma el comportamiento independiente del orden para Selection.
   - swaps observados: Selection realiza **muy pocos swaps** porque solo intercambia al final de cada pasada.
   - Bubble realiza **muchos swaps** cuando el arreglo está lejos del orden (ej.: inventario_500_inverso swaps = **124 750**).
   - Insertion no reporta swaps en las salidas pero sí shifts; su coste aparece como movimientos (shifts).
3. **Shifts (Insertion)**
   - Los shifts de Insertion reflejan los movimientos necesarios para insertar elementos y suelen corresponder con los swaps de Bubble en el mismo dataset:
     - citas_100: shifts = **2 434**.
     - pacientes_500: shifts = **54 473**.
     - inventario_500_inverso: shifts = **124 750** (peor caso).
   - Interpretación: muchos shifts implican mayor coste por movimiento de memoria/copia; en arreglos casi ordenados los shifts son pequeños (ej. 215 en citas_100_casi_ordenadas), por eso Insertion es muy eficiente allí.
4. **Efecto de duplicados**
   - En pacientes_500 (con duplicados): los algoritmos **estables** (Insertion y Bubble) preservan el orden relativo de elementos con la misma clave; **Selection** no es estable.
   - Observación práctica: si la aplicación requiere preservar orden secundario (registro de llegada, por ejemplo), preferir algoritmos estables aun si su tiempo es ligeramente mayor.
5. **Relación orden inicial ↔ rendimiento**
   - citas_100_casi_ordenadas: Insertion se beneficia enormemente (mediana **16 400 ns**) debido a pocas comparaciones y shifts; Bubble con corte temprano mejora respecto a la versión básica pero sigue por detrás de Insertion.
   - inventario_500_inverso: Insertion sufre en teoría por muchos shifts, mientras Selection mantiene comparaciones predecibles (≈124 750) y swaps modestos; en estos ensayos Insertion mostró una mediana buena, pero acompañada de mayor variabilidad (ver punto 6).
6. **Observaciones de confiabilidad**
   - Se observó **variabilidad** entre repeticiones y algunos outliers (por ejemplo, ejecuciones aisladas con tiempos muy altos). Por eso se empleó la **mediana** como medida central.
   - La mediana es útil para comparar tendencias, pero conviene informar también IQR o rango cuando hay outliers.

- Posibles fuentes de variación: JIT warm-up, recolección de basura (GC) y carga del sistema. Recomendación experimental: más repeticiones, ejecutar warm-up y reportar mediana + IQR.

9. Matriz de recomendación (reglas prácticas)

- **Si datos casi ordenados y n ≤ 500 → usar Insertion.**
  - Justificación: en citas_100_casi_ordenadas Insertion mediana = **16 400 ns**, con pocas operaciones (comparaciones/shifts) — hasta ~9× más rápido que Selection en ese caso.
- **Si se desea minimizar swaps/intercambios físicos y n no muy grande → usar Selection.**
  - Justificación: Selection realiza ≈ n(n−1)/2 comparaciones pero **pocos swaps**, útil cuando las escrituras son costosas.
- **Si hay muchos duplicados y se necesita estabilidad → preferir Insertion o Bubble.**
  - Justificación: ambos preservan orden relativo en pacientes_500; Selection no lo hace.
- **Si los datos están inversos y n grande → evitar Bubble e Insertion en producción; preferir Selection si la estabilidad no importa o, mejor aún, usar algoritmos O(n log n) (Merge/Quick) para escala.**
  - Justificación: Bubble realiza muchos swaps en inverso; Selection mantiene comparaciones predecibles y pocos swaps, pero para n grandes conviene un algoritmo asintóticamente mejor.
- **Regla resumida con umbrales observados:**
  - Casi ordenado & n ≤ 500 → **Insertion**.
  - Minimizar swaps → **Selection**.
  - Duplicados + estabilidad → **Insertion/Bubble**.
  - Inverso & n grande → **algoritmos O(n log n)** (fuera del alcance de los algoritmos simples).

10. Respuestas a preguntas de control

1. **¿Por qué imprimir trazas durante la medición distorsiona los tiempos?** Imprimir en consola implica operaciones de I/O costosas y no determinísticas que añaden latencia y ruido al tiempo medido de la rutina; por eso las trazas se desactivan durante las mediciones y se registran solo en logs fuera de la sección cronometrada.
2. **¿Por qué Selección tiene comparaciones ≈ n(n−1)/2 independientemente del orden inicial?** Porque el algoritmo busca el mínimo en cada iteración externa recorriendo el resto del arreglo; esto implica comprobar cada par candidato, resultando en ∑_{i=0}^{n-2} (n−1−i) = n(n−1)/2 comparaciones en el caso general.
3. **¿Por qué Inserción es competitivo en datos casi ordenados?** Porque Inserción desplaza principalmente los elementos que están fuera de lugar; si la mayoría ya está ordenada, cada inserción requiere pocos desplazamientos, resultando en costo cercano a O(n).
4. **¿Qué papel juegan los duplicados en la estabilidad del resultado?** Los duplicados hacen relevante la estabilidad: los algoritmos estables (Insertion, Burbuja) preservan el orden relativo de elementos con la misma clave; los algoritmos no estables (Selection) pueden reordenar iguales, lo cual puede ser indeseable en aplicaciones donde el orden secundario importa.
5. **¿Por qué Burbuja con corte temprano mejora en "casi ordenado" pero no en "inverso"?** El corte temprano detecta la ausencia de swaps en una pasada y finaliza; en casi ordenado suele ocurrir rápido. En orden inverso hay swaps constantes por muchas pasadas, así que el corte no se activa pronto y el tiempo consume muchas pasadas.

11. Conclusiones

- **Recomendación por dataset:**
  - citas_100 → **Insertion** (mediana **74 100 ns**) por mejor tiempo observado.
  - citas_100_casi_ordenadas → **Insertion** (mediana **16 400 ns**) claramente preferible.
  - pacientes_500 → **Insertion** mostró la mejor mediana (**389 800 ns**) y mantiene estabilidad; Selection es una alternativa si se quieren minimizar swaps.
  - inventario_500_inverso → aunque la mediana favorece a Insertion en estos ensayos (**135 300 ns**), la presencia de outliers sugiere cautela; Selection ofrece comportamiento más predecible en conteos teóricos.
- **Limitaciones del experimento:**
  - Variabilidad entre ejecuciones (JIT/GC/outliers), dependencia del entorno de ejecución y uso exclusivo de algoritmos $O(n^2)$ para tamaños modestos.
- **Mejoras propuestas:**
  - Aumentar repeticiones y realizar warm-up explícito; reportar mediana + IQR.
  - Medir también coste de memoria/movimientos y probar algoritmos O(n log n) con n mayores ($\geq 10^3$).
  - Aislar el entorno (fijar parámetros de la JVM, controlar GC) para reducir ruido experimental.

12. Archivos entregados (lista)

- Código fuente Java instrumentado: src/sorting/*.java.
- Datasets (CSV): datasets/citas_100.csv, datasets/citas_100_casi_ordenadas.csv, datasets/pacientes_500.csv, datasets/inventario_500_inverso.csv.
- Resultados: resultados_ordenacion.csv.
- Logs/evidencia: logs/salida.txt.
- CHECKLIST: CHECKLIST.md.

13. Bibliografía (mínimo requerido)

1. OpenDSA Project, “Sorting and Searching Modules,” Virginia Tech, 2021–2024.
2. Bible, P. W. & Moser, L., *An Open Guide to Data Structures and Algorithms.*, PALNI Open Press, 2023.
3. Oracle, *Java SE Documentation* (Arrays, I/O, benchmarking notes), 2021–2025.
4. OpenJDK, *JMH – Java Microbenchmark Harness: Samples and Guidance*, 2020–2025.

Anexos

- **A. Instrucciones rápidas de compilación y ejecución:** javac -d out src/sorting/*.java java -cp out sorting.SortingDemo
- **B. Estructura del repo:**

```
/src/sorting/
/datasets/
/
/logs/
README.md
CHECKLIST.md
```

- **C. Notas sobre reproducibilidad:**
  - Los scripts usan semilla 42 para garantizar reproducibilidad de los CSV.

- El harness utiliza R = 10 y DISCARD = 3 por defecto; parámetros editables en SortingDemo.java.