FACULTAD DE LA ENERGÍA, LAS INDUSTRIAS Y LOS RECURSOS NATURALES NO RENOVABLES

CARRERA DE INGENIERÍA EN CIENCIAS DE LA COMPUTACIÓN

TERCER CICLO “A”

ASIGNATURA
ESTRUCTURA DE DATOS

FECHA DE ENTREGA:

21/11/2025

ELABORADO POR:
Steven Alexander Jumbo Jaramillo
Servio Julian Vega Jiménez

Docente encargado: Ingeniero Andrés Navas Castellanos

Septiembre 2025 – Febrero 2026

LOJA – ECUADOR

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

El desarrollo del taller se realizó aplicando el enfoque metodológico **ABPr (Aprendizaje Basado en Proyectos)** siguiendo las directrices de la guía institucional. La práctica consistió en implementar, comparar y analizar distintos algoritmos de ordenación usando un enfoque orientado a objetos, con separación por capas (modelo, vista y controlador). A continuación, se detalla la metodología seguida:

5.1 Inicio

Antes de iniciar la implementación, se revisó el objetivo general de la práctica y la estructura del proyecto proporcionado. Se verificaron las clases incluidas en el paquete ed.u2.sorting, identificando la arquitectura MVC compuesta por:

- **Modelo:** InsertionSort, SelectionSort, BubbleSort, SortingUtils, DatasetGenerator, CsvDataLoader, SortStats
- **Vista:** ConsoleView
- **Controlador:** SortingController
- **Ejecutor principal:** Main

Posteriormente se realizó la configuración del entorno de trabajo:

- **JDK:** OpenJDK
- **Editor/IDE:** IntelliJ IDEA o Visual Studio Code
- **Gestor de dependencias:** Maven
- **Estructura base:** taller6-ordenacion-comparacion/src/main/java/ed/u2/sorting

También se revisó la documentación interna del código para comprender el flujo: carga de datos, selección del algoritmo, ejecución de métricas y presentación de resultados.

5.2 Desarrollo

El desarrollo de la práctica siguió un orden progresivo, iniciando desde la correcta comprensión del flujo MVC, luego la implementación y validación de cada algoritmo de ordenación, hasta la carga de datos y comparación final.

5.2.1 Implementación del proyecto

El proyecto se compone de los siguientes paquetes y clases principales:

Estructura del paquete:

ed.u2.sorting

```
├── controller
│   └── SortingController
├── model
│   ├── InsertionSort
│   ├── SelectionSort
│   ├── BubbleSort
│   ├── SortingUtils
│   ├── SortStats
│   ├── DatasetGenerator
│   └── CsvDataLoader
└── view
```

└── ConsoleView

Descripción general del desarrollo

- Se revisaron e implementaron los algoritmos **InsertionSort**, **SelectionSort** y **BubbleSort**, cada uno diseñando su propia métrica interna mediante la clase SortStats.
- Se integró el cargador de datos CsvDataLoader para soportar lectura de datasets externos.
- Se utilizó SortingController para gestionar la ejecución de los algoritmos, seleccionar el método de ordenación y coordinar el flujo de métricas.
- ConsoleView se encargó de mostrar resultados al usuario, bajo una interfaz simple pero estructurada.

El programa permite generar conjuntos de datos, aplicar distintos algoritmos de forma independiente y comparar resultados como:

- Número de comparaciones
- Número de intercambios
- Tiempo de ejecución
- Estado final del arreglo ordenado

5.2.2 Detalle de cada algoritmo

a) InsertionSort

El algoritmo de Inserción fue diseñado recorriendo el arreglo desde el índice 1 hasta n-1.
En cada iteración:

- Se toma el elemento pivote (*key*).
- Se desplaza dentro del subarreglo ordenado previo.
- Se realizan comparaciones directas contra elementos anteriores.
- Se registra cada operación en SortStats.

Se añadieron métricas de:

- Comparaciones
- Movimientos de elementos
- Iteraciones del bucle externo

El comportamiento observado muestra que InsertionSort es óptimo para arreglos pequeños o casi ordenados.

b) SelectionSort

Este algoritmo identifica el elemento mínimo del subarreglo en cada iteración del índice *i*.

El desarrollo incluyó:

- Búsqueda del mínimo en cada pasada.
- Contador de comparaciones durante la detección del mínimo.
- Solo un intercambio por iteración, característica clásica del algoritmo.
- Registro de métricas en SortStats.

SelectionSort permite analizar su naturaleza: **muchas comparaciones, pocos intercambios**.

c) BubbleSort

Se implementó comparando elementos adyacentes y realizando intercambios cuando es necesario.

La versión aplicada incluye la optimización conocida como **corte temprano**, mediante un indicador booleano que detecta si hubo o no intercambios en una pasada completa:

- Si no se detectan swaps → el arreglo está ordenado y el algoritmo finaliza inmediatamente.
- Esto mejora considerablemente su rendimiento para casi-ordenados.

Se registraron:

- Comparaciones adyacentes
- Intercambios por pasada
- Estado del arreglo tras finalizar cada recorrido

5.2.3 Integración del sistema MVC

Además de los algoritmos, el proyecto incluye la arquitectura MVC:

Controlador (SortingController)

- Selecciona el algoritmo de acuerdo al tipo requerido.

- Coordina ejecución, recolección de resultados y presentación.
- Genera las métricas asociadas al proceso de ordenación.

Vista (ConsoleView)

- Muestra las métricas y resultados al usuario.
- Permite visualizar comparaciones entre algoritmos.

DatasetGenerator y CsvDataLoader

Permiten cargar datos desde:

- Archivos CSV
- Generación interna de listas aleatorias
- Listas personalizadas para análisis

Esto amplía la capacidad del proyecto permitiendo pruebas más completas.

5.2.4 Casos borde considerados

Durante el proceso se evaluaron los casos esenciales para validar robustez:

- Arreglo vacío
- Arreglo con un solo elemento
- Arreglo ya ordenado
- Arreglo en orden inverso
- Arreglo con valores duplicados
- Arreglos generados aleatoriamente
- Dataset cargado desde archivo CSV

La correcta ejecución en estos escenarios confirma que las implementaciones son estables y compatibles con el controlador y la vista.

5.3 Evidencias y Validación

Para validar el correcto funcionamiento del proyecto se generaron múltiples ejecuciones utilizando diferentes conjuntos de datos, tanto aleatorios como cargados desde archivos CSV. Cada algoritmo fue probado de manera independiente y posteriormente comparado mediante el controlador, registrando métricas como tiempo de ejecución, número de comparaciones e intercambios.

```
====== BENCHMARKING DE ORDENACIÓN ======
Config: R=10, Descartar=3, Mediana de 7 runs.


>>> Dataset 1: Citas Aleatorias (N=100)
+----------------+--------------+------------+------------+--------------+
| Algoritmo      | Tiempo(ns)   | Comparac.  | Swaps      | Estado       |
+----------------+--------------+------------+------------+--------------+
| Insertion      | 95500        | 2479       | 2384       | OK           |
| Selection      | 133100       | 4950       | 94         | OK           |
| Bubble         | 201100       | 4944       | 2384       | OK           |
+----------------+--------------+------------+------------+--------------+
```

```
>>> Dataset 2: Citas Casi Ordenadas (N=100)
+----------------+--------------+------------+------------+--------------+
| Algoritmo      | Tiempo(ns)   | Comparac.  | Swaps      | Estado       |
+----------------+--------------+------------+------------+--------------+
| Insertion      | 14400        | 370        | 271        | OK           |
| Selection      | 35900        | 4950       | 5          | OK           |
| Bubble         | 61900        | 4422       | 271        | OK           |
+----------------+--------------+------------+------------+--------------+
```

```
>>> Dataset 3: Pacientes (N=500)
+---------------+-------------+-----------+-----------+-------------+
| Algoritmo     | Tiempo(ns)  | Comparac. | Swaps     | Estado      |
+---------------+-------------+-----------+-----------+-------------+
| Insertion     | 199000      | 42900     | 42401     | OK          |
| Selection     | 343700      | 124750    | 349       | OK          |
| Bubble        | 184000      | 111055    | 42401     | OK          |
+---------------+-------------+-----------+-----------+-------------+
```

```
>>> Dataset 4: Inventario Inverso (N=500)
+---------------+-------------+-----------+-----------+-------------+
| Algoritmo     | Tiempo(ns)  | Comparac. | Swaps     | Estado      |
+---------------+-------------+-----------+-----------+-------------+
| Insertion     | 113500      | 124750    | 124750    | OK          |
| Selection     | 109300      | 124750    | 250       | OK          |
| Bubble        | 222000      | 124750    | 124750    | OK          |
+---------------+-------------+-----------+-----------+-------------+
```

6. Preguntas de Control:

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

  Porque internamente, simplemente comparar dos números e intercambiarlos se hace en cuestión de nanosegundos, pero cuando hablamos de imprimir una línea en la consola, debemos esperar a que el programa se "comunique" con el sistema operativo para exportar los caracteres necesarios, y que el buffer de salida imprima texto dibujándolos en pantalla. Dicha acción, a diferencia de la ordenación, puede tardar micro o milisegundos.

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

  Esto se debe a lo rígido que es el algoritmo. La manera en cómo funciona su lógica o está programado, lo obliga a recorrer desde la posición *i+1*, hasta al final del arreglo hasta encontrar el mínimo sin importar que ya lo haya encontrado, puesto que no tiene forma de saberlo.

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

  Por cómo funciona el algoritmo, es decir, "compara el siguiente dato del arreglo y muévelos hacia atrás siempre y cuando estos sean mayores". Todo esto lo hace en una sola pasada, en un bucle *while*.
  Entonces, cuando nos encontramos en el mejor escenario, es decir, los datos casi ordenados, la mayoría de las veces al hacer la comparación con el elemento que toma (key), este ya suele ser mayor al anterior, por lo que, no es necesario que entre la condición while muchas veces, ya que es muy posible que en pocos desplazamientos ya ordene todo el algoritmo.

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

  Normalmente los datos duplicados ponen a prueba al algoritmo para ver si es capaz de seguir el orden lógico original de un registro. Por ejemplo, si habláramos de un gestor de tickets, donde dos personas tienen la misma prioridad (1), como Pedro y Ana, un algoritmo estable seguiría este orden sin realizar intercambios innecesarios. En este caso, como Pedro estaba primero, se mantendrá primero y luego irá Ana.
  Un algoritmo que no valide estas condiciones puede invertir de forma ilógica el orden, en este caso Ana iría primero por más que Pedro haya llegado primero. Es fundamental evitar esto, más aún cuando hablamos de sistemas reales que manejan datos de esta forma.

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

  Por cómo funciona el algoritmo. Normalmente burbuja está hecho para recorrer el algoritmo máximo n-1 iteraciones, donde n es el tamaño del arreglo. Ahora cuando implementamos corte temprano, al momento que detecta que no hubo intercambios, corta el bucle por más que no haya terminado de iterar las n-1 iteraciones. Esto "mejora" cuando los datos están casi ordenados, ya que es muy posible que el algoritmo ordene todo en menos iteraciones que n-1, lo que hará que termine antes de hacer todas las iteraciones.
  Ahora bien, cuando el algoritmo está inverso es todo lo contrario, porque es una de las peores situaciones para el algoritmo, ya que el dato mínimo estará en la posición n-1 – i (i = 0) obligándolo a recorrer todo; y cuando ahora intente ordenar el siguiente dato, se encontrará con que el siguiente dato está en una posición antes de la posición del dato que recién ordenó (n-1-i donde i = 1). Esto será así con cada dato, obligándolo a iterar las n-1 iteraciones.

LINK DEL REPOSITORIO EN GITHUB:

https://github.com/SteveJ4ra/taller6-OrdenacionJava-Comparacion