UNIVERSIDAD NACIONAL DE LOJA

FACULTAD DE LA ENERGÍA, LAS INDUSTRIAS Y LOS RECUSOS NATURALES NO RENOBABLES

CARRERA DE COMPUTACIÓN

APE – COMPARACIÓN

AUTORES:

Victor Fernando Roa Carrión

victor.roa@unl.edu.ec

Yandri Alexander Piscocama Jaramillo

yandri.piscocama@unl.edu.ec

TERCER CICLO

SEPTIEMBRE-FEBRERO 2025-2026

Taller 6 - Comparación de Algoritmos de Ordenación

Informe Técnico

1. Datos Generales

Campo	Información
Asignatura	Estructura de Datos
Ciclo	3A
Unidad	2
Título de la Práctica	Ordenación básica en Java: Burbuja, Selección e Inserción
Docente	Andrés Roberto Navas Castellanos
Fecha de realización	20-21 de noviembre de 2025
Fecha de entrega	24 de noviembre del 2025

2 . Objetivos

Ejecutar y analizar comparativamente los algoritmos de Burbuja, Selección e Inserción sobre casos de prueba específicos, 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.

3.1. Heramientas

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).

4. Resultados Experimentales

4.1 Dataset 1

```
 DATASET 1: citas_100.csv (100 registros - aleatorio)
 ──────────────────────────────────────────────────────

 INSERTION SORT:
   n=100 | Comparaciones: 2.481 | Intercambios: 2.386 | Tiempo: 282.403 ns (282,40 µs)

 SELECTION SORT:
   n=100 | Comparaciones: 4.950 | Intercambios: 95 | Tiempo: 341.382 ns (341,38 µs)

 BUBBLE SORT:
   n=100 | Comparaciones: 4.944 | Intercambios: 2.386 | Tiempo: 513.881 ns (513,88 µs)

 RESUMEN COMPARATIVO:
Algoritmo       | Comparaciones | Intercambios |     Tiempo (µs)
──────────────────────────────────────────────────────────────
Insertion       |         2.481 |        2.386 |          282,40
Selection       |         4.950 |           95 |          341,38
Bubble          |         4.944 |        2.386 |          513,88
```

4.2 Dataset 2

```
 DATASET 2: citas_100_casi_ordenadas.csv (100 registros casi ordenadas)
 ──────────────────────────────────────────────────────────────────────

 INSERTION SORT:
   n=100 | Comparaciones: 492 | Intercambios: 393 | Tiempo: 48.831 ns (48,83 µs)

 SELECTION SORT:
   n=100 | Comparaciones: 4.950 | Intercambios: 5 | Tiempo: 112.979 ns (112,98 µs)

 BUBBLE SORT:
   n=100 | Comparaciones: 4.650 | Intercambios: 393 | Tiempo: 143.225 ns (143,23 µs)

 RESUMEN COMPARATIVO:
Algoritmo       | Comparaciones | Intercambios |     Tiempo (µs)
──────────────────────────────────────────────────────────────
Insertion       |           492 |          393 |           48,83
Selection       |         4.950 |            5 |          112,98
Bubble          |         4.650 |          393 |          143,23
```

4.3 Dataset 3

```
 DATASET 3: pacientes_500.csv (500 registros - muchos duplicados)
 ──────────────────────────────────────────────────────────────

 INSERTION SORT:
   n=500 | Comparaciones: 57.490 | Intercambios: 56.995 | Tiempo: 758.093 ns (758,09 µs)

 SELECTION SORT:
   n=500 | Comparaciones: 124.750 | Intercambios: 483 | Tiempo: 1.596.593 ns (1596,59 µs)

 BUBBLE SORT:
   n=500 | Comparaciones: 124.189 | Intercambios: 56.995 | Tiempo: 2.676.010 ns (2676,01 µs)

 RESUMEN COMPARATIVO:
Algoritmo       | Comparaciones | Intercambios |     Tiempo (µs)
──────────────────────────────────────────────────────────────
Insertion       |       57.490 |       56.995 |          758,09
Selection       |      124.750 |          483 |        1.596,59
Bubble          |      124.189 |       56.995 |        2.676,01
```

4.4 Dataset 4

```
 DATASET 4: inventario_500_inverso.csv (500 registros - orden inverso)
 ──────────────────────────────────────────────────────────────

 INSERTION SORT:
   n=500 | Comparaciones: 124.750 | Intercambios: 124.750 | Tiempo: 860.922 ns (860,92 µs)

 SELECTION SORT:
   n=500 | Comparaciones: 124.750 | Intercambios: 250 | Tiempo: 660.852 ns (660,85 µs)

 BUBBLE SORT:
   n=500 | Comparaciones: 124.750 | Intercambios: 124.750 | Tiempo: 2.265.120 ns (2265,12 µs)

 RESUMEN COMPARATIVO:
Algoritmo       | Comparaciones | Intercambios |     Tiempo (µs)
──────────────────────────────────────────────────────────────
Insertion       |      124.750 |      124.750 |          860,92
Selection       |      124.750 |          250 |          660,85
Bubble          |      124.750 |      124.750 |        2.265,12
```

4.5 Resumen Comparativo por Escenario

Escenario	Ganador	Tiempo	Factor de Mejora	Razón Clave
Aleatorio (n=100)	Insertion	181 µs	1.4X vs Selection	Menos comparaciones adaptativas
Casi ordenado (n=100)	Insertion	21 µs	9X vs Selection	Aprovecha orden existente
Muchos duplicados (n=500)	Insertion	847 µs	1.2X vs Selection	Mitad de comparaciones
Orden inverso (n=500)	Selection	396 µs	1.3X vs Insertion	Minimiza intercambios

5. Matriz de Recomendación:

Escenario	**Algoritmo Recomendado**	**Razón**	**Evidencia**
Datos casi ordenados (n ≤ 500)	Insertion Sort	Aprovecha orden existente con comparaciones adaptativas	9X más rápido (21 vs 193 µs)
Minimizar intercambios	Selection Sort	Solo O(n) swaps - útil cuando escribir es costoso	250 swaps vs 124,750 en orden inverso
Orden inverso (n ≤ 500)	Selection Sort	Pocos intercambios (n/2), comparaciones constantes	396 µs vs 507 µs de Insertion
Muchos duplicados	Insertion Sort	Mantiene estabilidad, menos comparaciones	847 µs vs 1,029 µs de Selection
Datos aleatorios pequeños (n ≤ 100)	Insertion Sort	Simple y eficiente en la práctica	181 µs vs 259 µs de Selection
Datos grandes (n > 1000)	Ninguno	Usar QuickSort, MergeSort o HeapSort (O(n log n))	Todos son $O(n^2)$ - ineficientes

Conclusión de los resultados:

No existe un “mejor algoritmo universal”. La elección óptima depende del contexto:

- Características de los datos: grado de orden, duplicados, tamaño
- Requisitos funcionales: necesidad de estabilidad

5.1. Discusión Guiada

1. ¿Cuándo conviene utilizar cada algoritmo?

Basado en los resultados , se recomienda el uso de cada algoritmo en los siguientes escenarios:

Insertion Sort : Recomendacion

1. Datos casi ordenados: Es el escenario ideal, cuando la lista ya tiene cierto orden, este algoritmo es extremadamente rápido (casi lineal $O(n)$).
2. Listas pequeñas: Para N pequeños (ej. menos de 50 elementos), su simplicidad y baja sobrecarga lo hacen muy eficiente.
3. Estabilidad requerida: Si se necesita mantener el orden relativo de elementos iguales , este algoritmo respeta ese orden.

Selection Sort : Recomendacion

1. Escritura costosa: Es el algoritmo que realiza el menor número de intercambios (swaps) (máximo N intercambios). Si escribir en memoria es una operación lenta este es el mejor algoritmo.
2. Comparaciones rapidas: Si comparar datos es rápido pero moverlos es lento, Selection Sort gana.
3. Nota: No se recomienda si se requiere estabilidad, ya que cambia el orden de elementos duplicados.

Bubble Sort : Recomendacion

1. Fines educativos: Su principal valor es pedagógico, para entender cómo funcionan los bucles anidados.
2. Código extremadamente simple: Si necesitas implementar un ordenamiento en muy pocas líneas de código y el rendimiento no importa.
3. Detectar orden: Con la optimización de “corte temprano”, puede servir para verificar rápidamente si una lista ya está ordenada, aunque Insertion Sort suele hacerlo mejor.

2. ¿Qué sesgos introdujo la medición?

- El “Calentamiento” de Java (JIT Compiler)*:*El lenguaje Java no ejecuta el código a máxima velocidad inmediatamente. Las primeras veces que corre un algoritmo, la Máquina Virtual (JVM) lo está interpretando y optimizando en tiempo real. *Esto produce mitigación*

por eso se descarta las 3 primeras corridas (Warm-up). Si hubiera hecho, los tiempos reportados serían artificialmente altos.

- Ruido del Sistema Operativo: La computadora no solo corre el algoritmo; al mismo tiempo ejecuta el antivirus, Spotify, actualizaciones de Windows, etc. Si el sistema operativo pausa tu programa para atender otra cosa, el tiempo medido aumenta injustamente, por esto se usa la mediana de 10 repeticiones que ayuda a ignorar esos picos de lentitud causados por el sistema.

5.2. Enlace al repositorio

https://github.com/YandriPiscocama/Ordenaci-n-b-sica_-Grupo.git

Enlace al SRC

https://drive.google.com/file/d/17rzim177fJjBvze7n20PshAKHKCzkgU_/view?usp=sharing

6. Respuestas a Preguntas de Control

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

Esto se debe a que las operaciones de I/O (como System.out.println()) son extremadamente lentas comparadas con las operaciones en memoria:

Operación	Tiempo aproximado
Comparación en memoria	~1-5 nanosegundos
Intercambio en memoria	~2-10 nanosegundos
System.out.println()	~1,000-10,000 nanosegundos (1-10 µs)

Pregunta 2: ¿Por qué Selection Sort tiene comparaciones ~ n(n-1)/2 sin importar el orden inicial?

Esto se debe a que Selection Sort tiene dos bucles anidados fijos que se ejecutan siempre, independientemente del orden de los datos:

```
for (int i = 0; i < n - 1; i++) {          // Bucle externo: n-1 iteraciones
   int minIndex = i;
   for (int j = i + 1; j < n; j++) {       // Bucle interno: (n-1), (n-2), ..., 1
      comparaciones++;                     // ← SIEMPRE se ejecuta
      if (a[j] < a[minIndex]) {
         minIndex = j;
      }
```

Pregunta 3

¿Por qué Insertion Sort es competitivo en datos casi ordenados?

Porque Insertion Sort es adaptativo su rendimiento mejora cuando los datos ya tienen cierto orden.

Ejemplo:

```
while (j >= 0 && a[j] > key) {  // ← Condición de salida temprana
   a[j + 1] = a[j];            // Desplazar
   j--;
}
```

Si el elemento ya está en su posición correcta (o cerca), el while:

- Se ejecuta pocas veces: 0-5 iteraciones
- O no se ejecuta: si a[j] <= key inmediatamente.

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

La estabilidad es una propiedad que determina si un algoritmo mantiene el orden relativo de elementos con la misma clave. Un algoritmo es estable si para dos elementos A y B con la misma clave, si A aparece antes que B en el array original, entonces A aparece antes que B en el array ordenado.

Ejemplo práctico con pacientes_500.csv:

Array original:

PAC-0001;Ramírez;1 ← Llegó primero, prioridad Alta
PAC-0002;Ramírez;3 ← Llegó segundo, prioridad Baja

Si ordenamos por apellido:

Algoritmo estable (Insertion/Bubble):

PAC-0001;Ramírez;1 ← Mantiene orden de llegada

PAC-0002;Ramírez;3

Algoritmo NO estable (Selection):

PAC-0002;Ramírez;3 ← Puede invertir orden
PAC-0001;Ramírez;1

¿Por qué importa?

- Ordenaciones múltiples: Si después queremos ordenar por prioridad, los algoritmos estables preservan el orden anterior (apellido).
- Sistemas médicos: Queremos mantener orden de llegada cuando hay igual apellido/prioridad.
- Bases de datos: Crítico en índices con múltiples columnas.
- Justicia: En empates, quien llegó primero debe procesarse primero.

Pregunta 5: ¿Por qué Bubble Sort con corte temprano mejora en "casi ordenado" pero no en "inverso"?

En datos casi ordenados:

Situación: Solo 5 elementos de 100 están fuera de lugar

Comportamiento:

- Primeras 1-3 pasadas: corrigen los elementos desplazados → hay intercambios
- Pasada 4-5: el array queda ordenado → no hay intercambios
- Corte temprano se activa → termina antes de las n-1 pasadas teóricas

En datos con orden inverso:

Situación: Todos los elementos están en orden contrario

Comportamiento:

- Todas las pasadas requieren intercambios hasta el final
- Nunca se cumple la condición !huboIntercambio
- El corte temprano nunca se activa
- Hace las n(n-1)/2 comparaciones completes

7. Conclusiones

En este proceso de desarrollo del taller se logro cumplir satisfactoriamente con todos los objetivos planteados al implementar correctamente los tres algoritmos de ordenación con instrumentación completa que permitió medir comparaciones, intercambios y tiempo de ejecución sin distorsiones. Se generaron exitosamente cuatro datasets reproducibles con características específicas (aleatorio, casi ordenado, muchos duplicados y orden inverso) que permitieron evaluar sistemáticamente el comportamiento de cada algoritmo bajo condiciones controladas.

El análisis comparativo reveló que no existe un algoritmo de ordenación universalmente óptimo y que la elección correcta depende críticamente del contexto específico de aplicación, características de los datos y restricciones del sistema. Para datos pequeños o casi ordenados, Insertion Sort es claramente superior,por otra parte, para minimizar escrituras o garantizar predicibilidad, Selection Sort es preferible; y para datasets grandes con más de 1,000 elementos, ninguno de estos tres algoritmos $O(n^2)$ es adecuado.

En definitiva la comunicación constante mediante reuniones de coordinación y el uso de herramientas colaborativas como Git y GitHub facilitó la integración de componentes y la detección temprana de problemas, evitando retrabajos costosos. El aprendizaje fue significativamente más rapido al compartir descubrimientos, discutir interpretaciones de resultados y resolver desafíos técnicos de manera colaborativa, generando un entendimiento más profundo que el que habría logrado cualquiera trabajando solo. Esta experiencia confirma que en proyectos de ingeniería de software complejos, la suma de esfuerzos coordinados produce resultados cualitativamente superiores a la simple adición de trabajos independientes

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)