UNIVERSIDAD NACIONAL DE LOJA

Facultad de la Energía, las Industrias y los Recursos Naturales No Renovables

Computación

Estructura de Datos

Informe

Docente: Ing. Navas Roberto

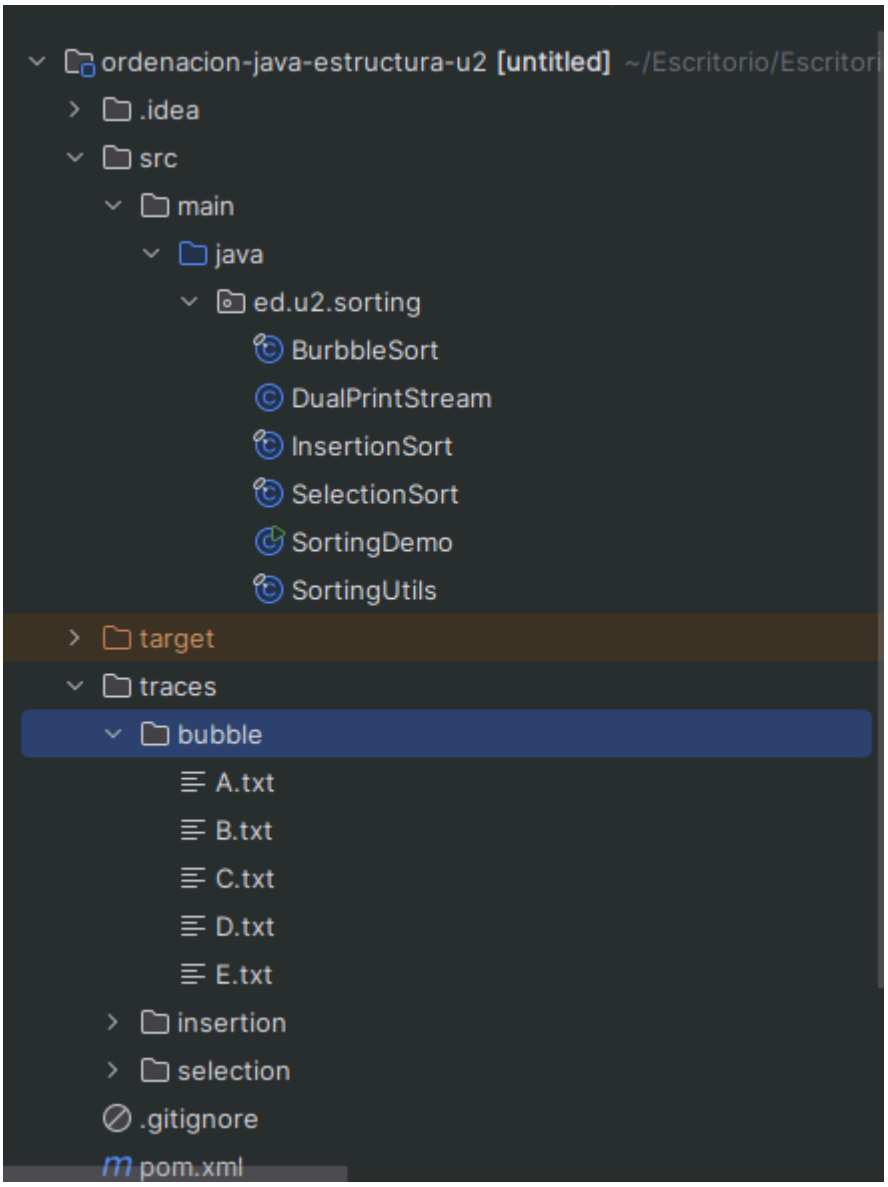
Integrantes:

- Mark González
- Anderson Coello

Ciclo: 3ro ciclo

LOJA – ECUADOR
2025

Estructura de nuestro proyecto



1. **src/main/java/ed/u2/sorting/**

Carpeta donde se encuentra todo el código fuente del proyecto. Incluye la implementación de los tres algoritmos de ordenación y las clases auxiliares necesarias para ejecutar y generar trazas.

BubbleSort.java

Implementa el algoritmo de ordenación Burbuja, incluyendo la optimización de corte temprano.

Recibe: Un arreglo de enteros.

Devuelve: El arreglo ordenado (in-place).

Produce: Trazas paso a paso de cada comparación y swap.

InsertionSort.java

Contiene la implementación del algoritmo de Inserción Directa.

Recibe: Un arreglo de enteros.

Devuelve: El arreglo ordenado.

Produce: Trazas detalladas de movimientos e inserciones.

SelectionSort.java

Implementa el algoritmo de Selección, mostrando cómo se busca el mínimo en cada iteración.

Recibe: Un arreglo de enteros.

Devuelve: El arreglo ordenado.

Produce: Trazas de comparaciones y cambios de posición.

SortingDemo.java

Clase principal del proyecto.

Ejecuta los tres algoritmos con los datasets A, B, C, D y E.

Redirige la salida para generar archivos .txt dentro de la carpeta traces.

Muestra también la ejecución en consola.

SortingUtils.java

Clase de utilidades.

Contiene métodos auxiliares como copiar arreglos, imprimirlos en formato legible y otras funciones de apoyo.

Es usada por los algoritmos y por la clase principal.

DualPrintStream.java

Clase encargada de duplicar la salida.

Hace posible que la misma impresión aparezca en consola y en un archivo .txt.

Sobrescribe print y println para enviar la salida a ambos destinos.

2. **Carpeta traces**

Directorio donde se almacenan automáticamente todos los archivos generados con las trazas de ejecución de los algoritmos.

La estructura está organizada por algoritmo:

traces/bubble/

Contiene las trazas generadas por BubbleSort para cada dataset.

Cada archivo contendrá los resultados correspondientes a los datasets A–E.

traces/insertion/

Carpeta reservada para las trazas de InsertionSort.

Cada archivo contendrá los resultados correspondientes a los datasets A–E.

traces/selection/

Carpeta reservada para las trazas de SelectionSort, organizadas también por dataset.

Cada archivo contendrá los resultados correspondientes a los datasets A–E.

3. **Readme.md**

El archivo README.md incluido dentro del proyecto cumple la función de documentar el propósito, estructura y funcionamiento general del taller de ordenación. Su contenido resume los elementos más importantes para la comprensión del trabajo realizado.

4. **pom.xml**

Archivo de configuración de Maven.

Define dependencias, versión del JDK y estructura estándar del proyecto.

Tabla de resultados de InsertionSort y SelectionSort

Swaps → SelectionSort

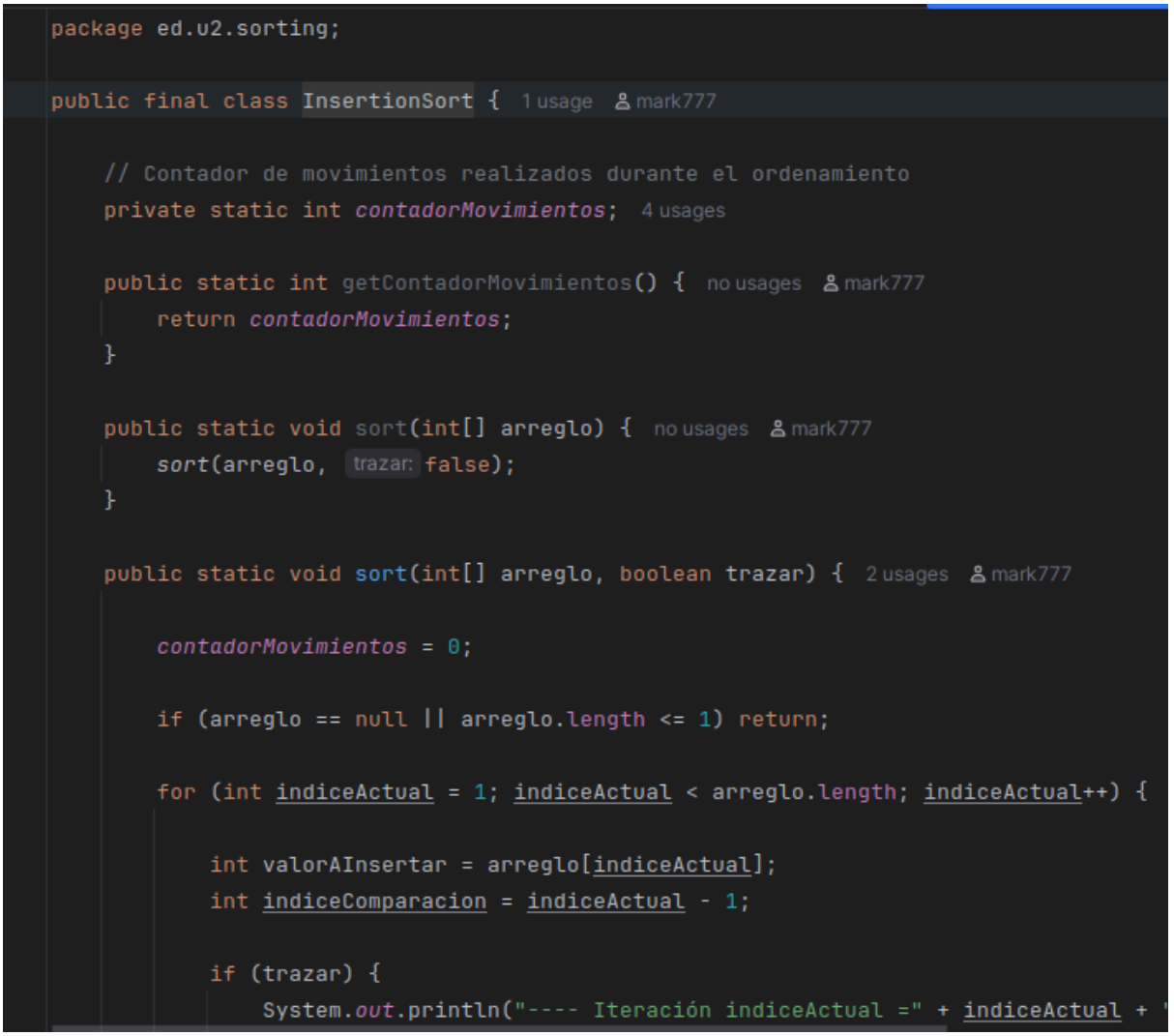
Movimientos → InsertionSort

Dataset	Algoritmo	Swaps	Movimientos
A	Insertion / Selection	2	4
B	Insertion / Selection	2	10
C	Insertion / Selection	0	0
D	Insertion / Selection	0	0
E	Insertion / Selection	2	4

Resultados de BubbleSort

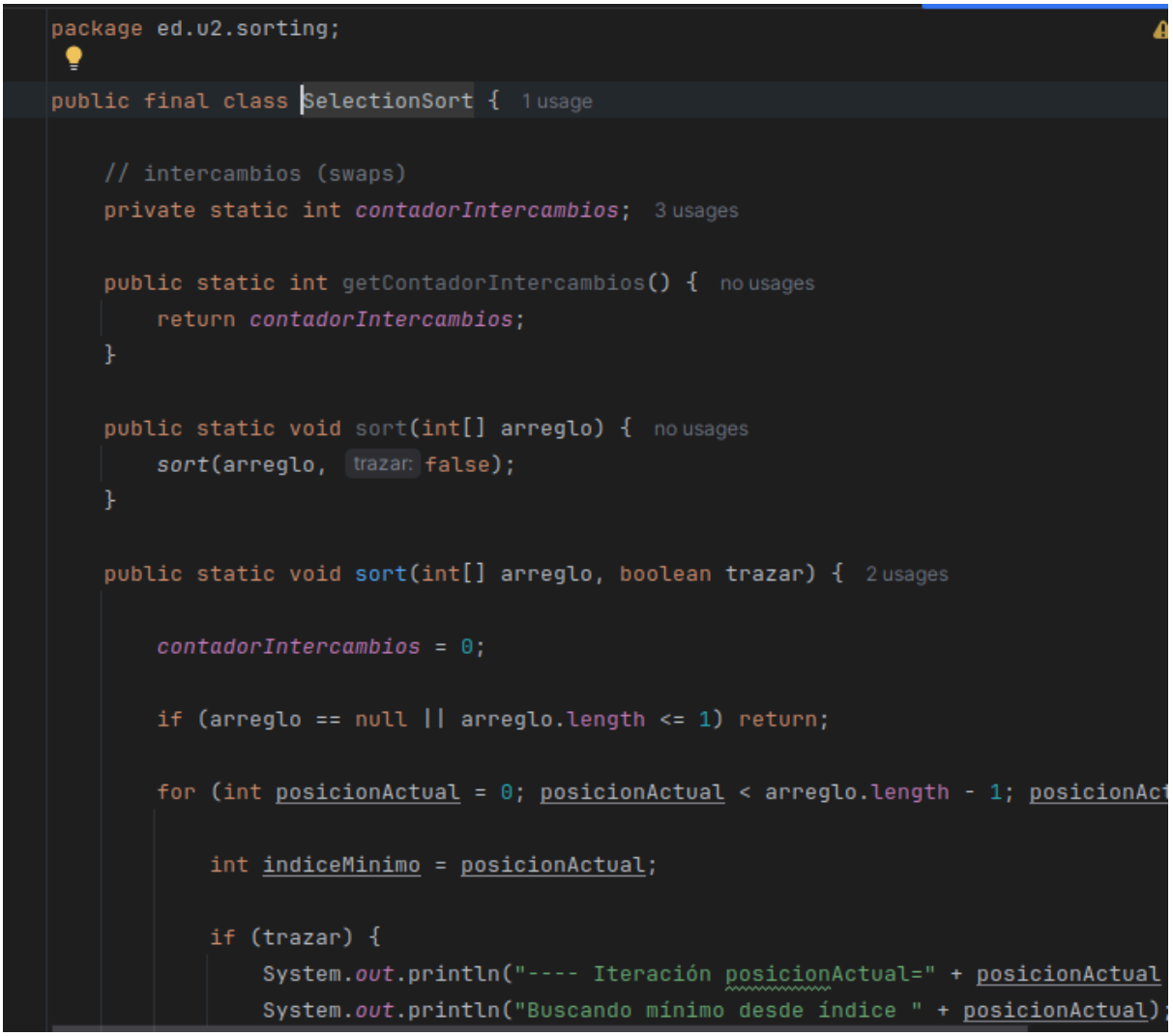
Dataset	Arreglo Original	Arreglo Ordenado	Pasadas Realizadas	Intercambios Realizados
A	[8, 3, 6, 3, 9]	[3, 3, 6, 8, 9]	3	4
B	[5, 4, 3, 2, 1]	[1, 2, 3, 4, 5]	4	10
C	[1, 2, 3, 4, 5]	[1, 2, 3, 4, 5]	1	0
D	[2,2,2,2]	[2, 2, 2, 2]	1	0
E	[9,1,8,2]	[1, 2, 8, 9]	3	4

Código de Insertion Sort



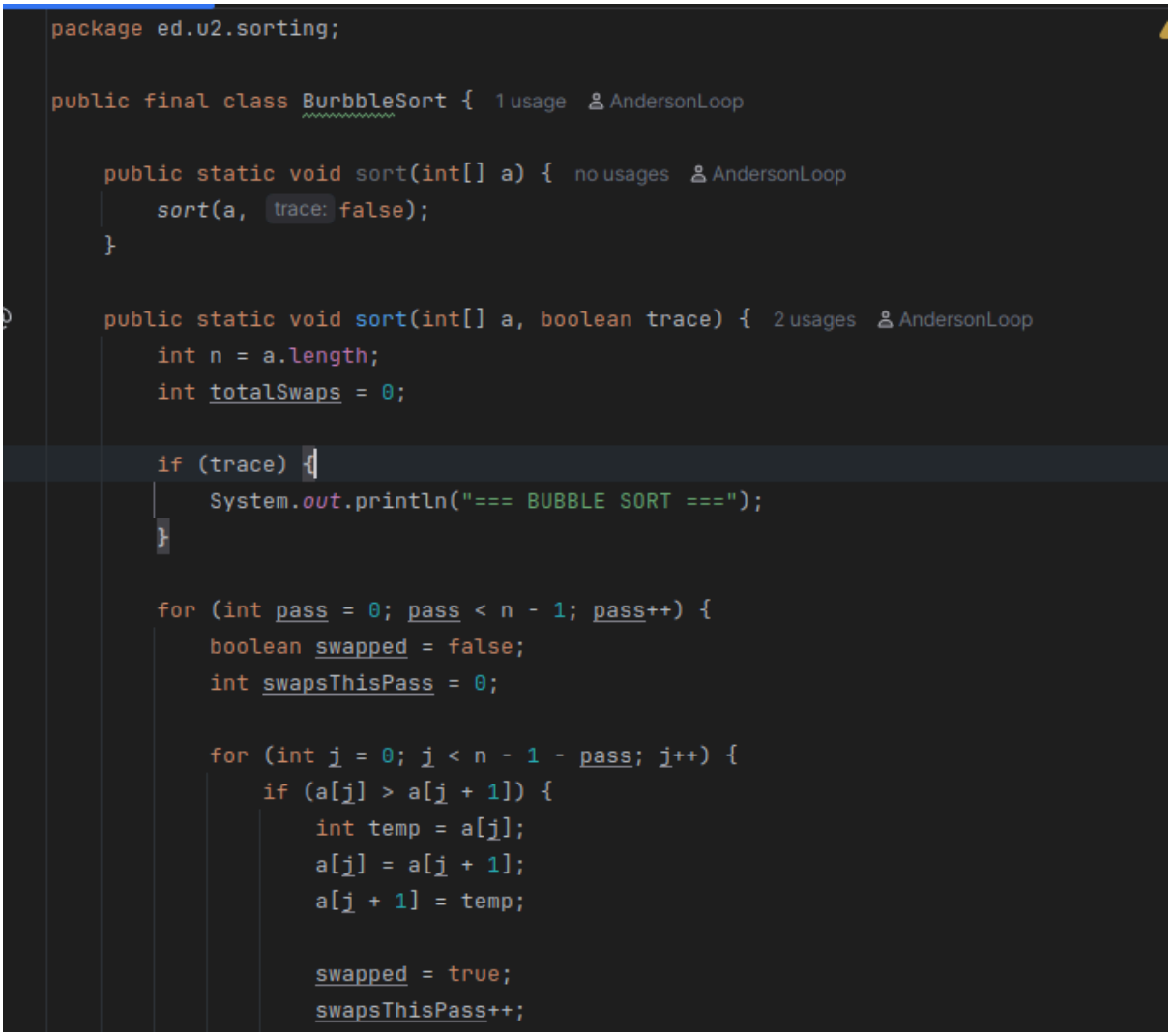
```java
package ed.u2.sorting;

public final class InsertionSort {  1 usage  mark777

    // Contador de movimientos realizados durante el ordenamiento
    private static int contadorMovimientos;  4 usages

    public static int getContadorMovimientos() {  no usages  mark777
        return contadorMovimientos;
    }

    public static void sort(int[] arreglo) {  no usages  mark777
        sort(arreglo,  trazar: false);
    }

    public static void sort(int[] arreglo, boolean trazar) {  2 usages  mark777

        contadorMovimientos = 0;

        if (arreglo == null || arreglo.length <= 1) return;

        for (int indiceActual = 1; indiceActual < arreglo.length; indiceActual++) {

            int valorAInsertar = arreglo[indiceActual];
            int indiceComparacion = indiceActual - 1;

            if (trazar) {
                System.out.println("---- Iteración indiceActual =" + indiceActual + '
```

Código de Selection Sort



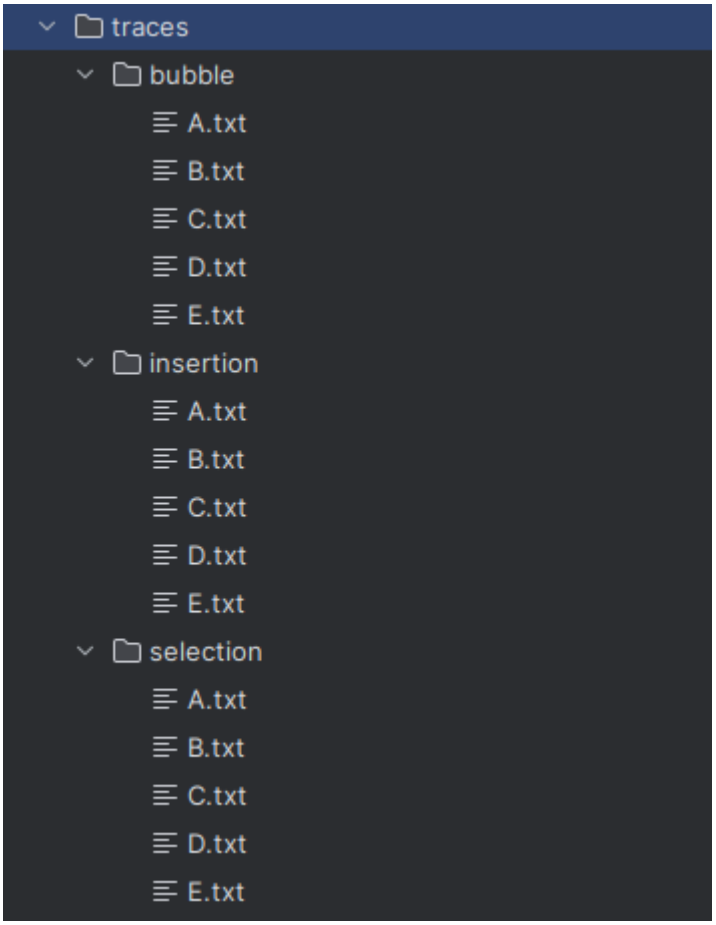
```java
package ed.u2.sorting;

public final class SelectionSort {  1 usage

    // intercambios (swaps)
    private static int contadorIntercambios;  3 usages

    public static int getContadorIntercambios() {  no usages
        return contadorIntercambios;
    }

    public static void sort(int[] arreglo) {  no usages
        sort(arreglo,  trazar: false);
    }

    public static void sort(int[] arreglo, boolean trazar) {  2 usages

        contadorIntercambios = 0;

        if (arreglo == null || arreglo.length <= 1) return;

        for (int posicionActual = 0; posicionActual < arreglo.length - 1; posicionAct

            int indiceMinimo = posicionActual;

            if (trazar) {
                System.out.println("---- Iteración posicionActual=" + posicionActual
                System.out.println("Buscando mínimo desde índice " + posicionActual);
```

Código de BubbleSort

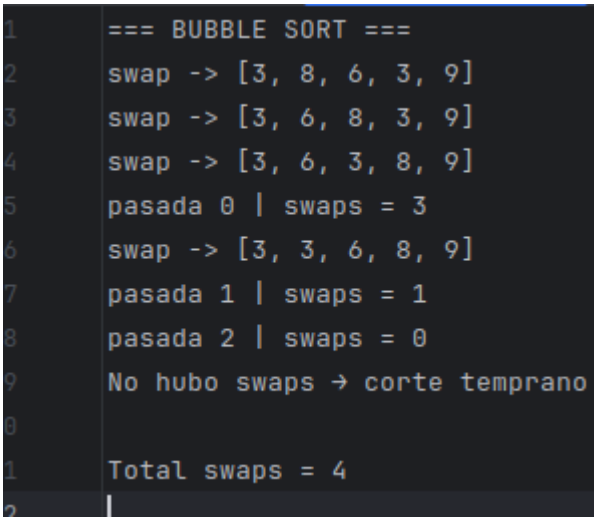


```java
package ed.u2.sorting;

public final class BurbbleSort {  1 usage  AndersonLoop

    public static void sort(int[] a) {  no usages  AndersonLoop
        sort(a,  trace: false);
    }

    public static void sort(int[] a, boolean trace) {  2 usages  AndersonLoop
        int n = a.length;
        int totalSwaps = 0;

        if (trace) {
            System.out.println("=== BUBBLE SORT ===");
        }

        for (int pass = 0; pass < n - 1; pass++) {
            boolean swapped = false;
            int swapsThisPass = 0;

            for (int j = 0; j < n - 1 - pass; j++) {
                if (a[j] > a[j + 1]) {
                    int temp = a[j];
                    a[j] = a[j + 1];
                    a[j + 1] = temp;

                    swapped = true;
                    swapsThisPass++;
```

Captura de la estructura elegida para la generación de los .txt



Ejemplo de lo que tiene uno de los archivos .txt



Nuestro programa genera los archivos .txt y también lo hace mediante consola, resultado de lo generado por consola

Resultado por Consola:

```
Buscando mínimo desde índice 3
Comparando arreglo[4]=5 con arreglo[3]=
=== BUBBLE SORT ===
swap -> [4, 5, 3, 2, 1]
swap -> [4, 3, 5, 2, 1]
swap -> [4, 3, 2, 5, 1]
swap -> [4, 3, 2, 1, 5]
pasada 0 | swaps = 4
swap -> [3, 4, 2, 1, 5]
swap -> [3, 2, 4, 1, 5]
swap -> [3, 2, 1, 4, 5]
pasada 1 | swaps = 3
swap -> [2, 3, 1, 4, 5]
swap -> [2, 1, 3, 4, 5]
pasada 2 | swaps = 2
swap -> [1, 2, 3, 4, 5]
pasada 3 | swaps = 1

Total swaps = 10
---- Iteración indiceActual =1 ----
Valor a insertar: 2
Insertar valor en posición 1
[ 1 2 3 4 5 ]
---- Iteración indiceActual =2 ----
Valor a insertar: 3
```

Casos de prueba

Caso 1: A: [8.3,6,3,9]

```
=== BUBBLE SORT ===
swap -> [3, 8, 6, 3, 9]
swap -> [3, 6, 8, 3, 9]
swap -> [3, 6, 3, 8, 9]
pasada 0 | swaps = 3
swap -> [3, 3, 6, 8, 9]
pasada 1 | swaps = 1
pasada 2 | swaps = 0
No hubo swaps → corte temprano

Total swaps = 4
```

```
---- Iteración indiceActual =1 ---
Valor a insertar: 3
Mover arreglo[0] -> arreglo[1]
[ 8 8 6 3 9 ]
Insertar valor en posición 0
[ 3 8 6 3 9 ]
---- Iteración indiceActual =2 ---
Valor a insertar: 6
Mover arreglo[1] -> arreglo[2]
[ 3 8 8 3 9 ]
Insertar valor en posición 1
[ 3 6 8 3 9 ]
---- Iteración indiceActual =3 ---
Valor a insertar: 3
Mover arreglo[2] -> arreglo[3]
[ 3 6 8 8 9 ]
Mover arreglo[1] -> arreglo[2]
[ 3 6 6 8 9 ]
Insertar valor en posición 1
[ 3 3 6 8 9 ]
---- Iteración indiceActual =4 ---
Valor a insertar: 9
Insertar valor en posición 4
[ 3 3 6 8 9 ]
```

```
---- Iteración posicionActual=0 ----
Buscando mínimo desde índice 0
Comparando arreglo[1]=3 con arreglo[0]=8
→ Nuevo mínimo encontrado en posición 1
Comparando arreglo[2]=6 con arreglo[1]=3
Comparando arreglo[3]=3 con arreglo[1]=3
Comparando arreglo[4]=9 con arreglo[1]=3
Intercambio entre posiciones 0 y 1
[ 3 8 6 3 9 ]
---- Iteración posicionActual=1 ----
Buscando mínimo desde índice 1
Comparando arreglo[2]=6 con arreglo[1]=8
→ Nuevo mínimo encontrado en posición 2
Comparando arreglo[3]=3 con arreglo[2]=6
→ Nuevo mínimo encontrado en posición 3
Comparando arreglo[4]=9 con arreglo[3]=3
Intercambio entre posiciones 1 y 3
[ 3 3 6 8 9 ]
---- Iteración posicionActual=2 ----
Buscando mínimo desde índice 2
Comparando arreglo[3]=8 con arreglo[2]=6
Comparando arreglo[4]=9 con arreglo[2]=6
---- Iteración posicionActual=3 ----
Buscando mínimo desde índice 3
Comparando arreglo[4]=9 con arreglo[3]=8
```

Caso 2: B[5,4,3,2,1]

```
=== BUBBLE SORT ===
swap -> [4, 5, 3, 2, 1]
swap -> [4, 3, 5, 2, 1]
swap -> [4, 3, 2, 5, 1]
swap -> [4, 3, 2, 1, 5]
pasada 0 | swaps = 4
swap -> [3, 4, 2, 1, 5]
swap -> [3, 2, 4, 1, 5]
swap -> [3, 2, 1, 4, 5]
pasada 1 | swaps = 3
swap -> [2, 3, 1, 4, 5]
swap -> [2, 1, 3, 4, 5]
pasada 2 | swaps = 2
swap -> [1, 2, 3, 4, 5]
pasada 3 | swaps = 1

Total swaps = 10
```

```
[ 4 5 5 2 1 ]
Mover arreglo[0] -> arreglo[1]
[ 4 4 5 2 1 ]
Insertar valor en posición 0
[ 3 4 5 2 1 ]
---- Iteración indiceActual =3 ----
Valor a insertar: 2
Mover arreglo[2] -> arreglo[3]
[ 3 4 5 5 1 ]
Mover arreglo[1] -> arreglo[2]
[ 3 4 4 5 1 ]
Mover arreglo[0] -> arreglo[1]
[ 3 3 4 5 1 ]
Insertar valor en posición 0
[ 2 3 4 5 1 ]
---- Iteración indiceActual =4 ----
Valor a insertar: 1
Mover arreglo[3] -> arreglo[4]
[ 2 3 4 5 5 ]
Mover arreglo[2] -> arreglo[3]
[ 2 3 4 4 5 ]
Mover arreglo[1] -> arreglo[2]
[ 2 3 3 4 5 ]
Mover arreglo[0] -> arreglo[1]
[ 2 2 3 4 5 ]
Insertar valor en posición 0
[ 1 2 3 4 5 ]
```

```
Buscando mínimo desde índice 0
Comparando arreglo[1]=4 con arreglo[0]=5
→ Nuevo mínimo encontrado en posición 1
Comparando arreglo[2]=3 con arreglo[1]=4
→ Nuevo mínimo encontrado en posición 2
Comparando arreglo[3]=2 con arreglo[2]=3
→ Nuevo mínimo encontrado en posición 3
Comparando arreglo[4]=1 con arreglo[3]=2
→ Nuevo mínimo encontrado en posición 4
Intercambio entre posiciones 0 y 4
[ 1 4 3 2 5 ]
---- Iteración posicionActual=1 ----
Buscando mínimo desde índice 1
Comparando arreglo[2]=3 con arreglo[1]=4
→ Nuevo mínimo encontrado en posición 2
Comparando arreglo[3]=2 con arreglo[2]=3
→ Nuevo mínimo encontrado en posición 3
Comparando arreglo[4]=5 con arreglo[3]=2
Intercambio entre posiciones 1 y 3
[ 1 2 3 4 5 ]
---- Iteración posicionActual=2 ----
Buscando mínimo desde índice 2
Comparando arreglo[3]=4 con arreglo[2]=3
Comparando arreglo[4]=5 con arreglo[2]=3
---- Iteración posicionActual=3 ----
Buscando mínimo desde índice 3
Comparando arreglo[4]=5 con arreglo[3]=4
```

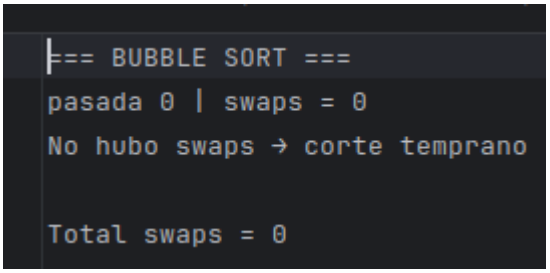
Caso 3: C: [1,2,3,4,5]

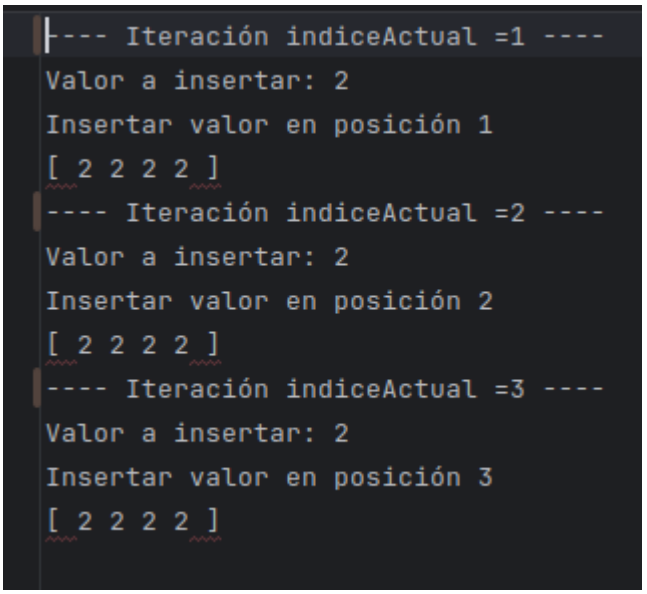
```
=== BUBBLE SORT ===
pasada 0 | swaps = 0
No hubo swaps → corte temprano

Total swaps = 0
```

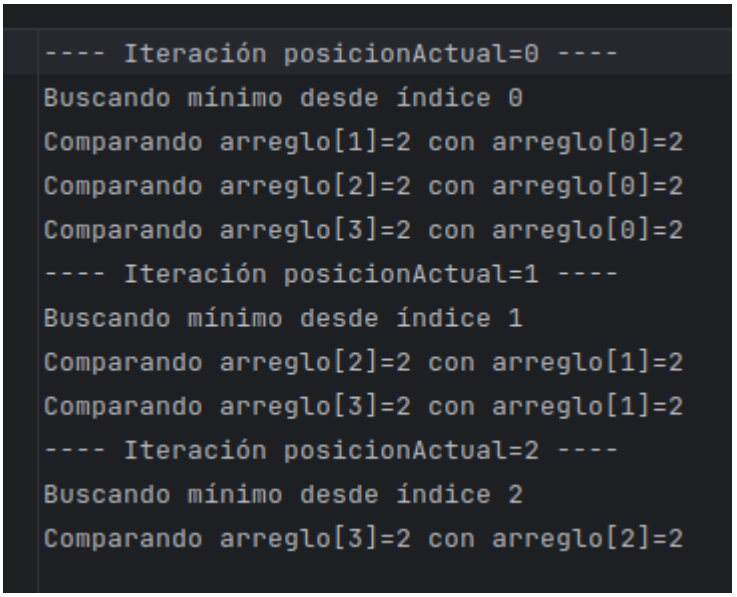
```
--- Iteración indiceActual =1 ----
Valor a insertar: 2
Insertar valor en posición 1
[ 1 2 3 4 5 ]
---- Iteración indiceActual =2 ----
Valor a insertar: 3
Insertar valor en posición 2
[ 1 2 3 4 5 ]
---- Iteración indiceActual =3 ----
Valor a insertar: 4
Insertar valor en posición 3
[ 1 2 3 4 5 ]
---- Iteración indiceActual =4 ----
Valor a insertar: 5
Insertar valor en posición 4
[ 1 2 3 4 5 ]
```

```
---- Iteración posicionActual=0 ----
Buscando mínimo desde índice 0
Comparando arreglo[1]=2 con arreglo[0]=1
Comparando arreglo[2]=3 con arreglo[0]=1
Comparando arreglo[3]=4 con arreglo[0]=1
Comparando arreglo[4]=5 con arreglo[0]=1
---- Iteración posicionActual=1 ----
Buscando mínimo desde índice 1
Comparando arreglo[2]=3 con arreglo[1]=2
Comparando arreglo[3]=4 con arreglo[1]=2
Comparando arreglo[4]=5 con arreglo[1]=2
---- Iteración posicionActual=2 ----
Buscando mínimo desde índice 2
Comparando arreglo[3]=4 con arreglo[2]=3
Comparando arreglo[4]=5 con arreglo[2]=3
---- Iteración posicionActual=3 ----
Buscando mínimo desde índice 3
Comparando arreglo[4]=5 con arreglo[3]=4
```

Caso 4: D:[2,2,2,2]


=== BUBBLE SORT ===
pasada 0 | swaps = 0
No hubo swaps → corte temprano
Total swaps = 0

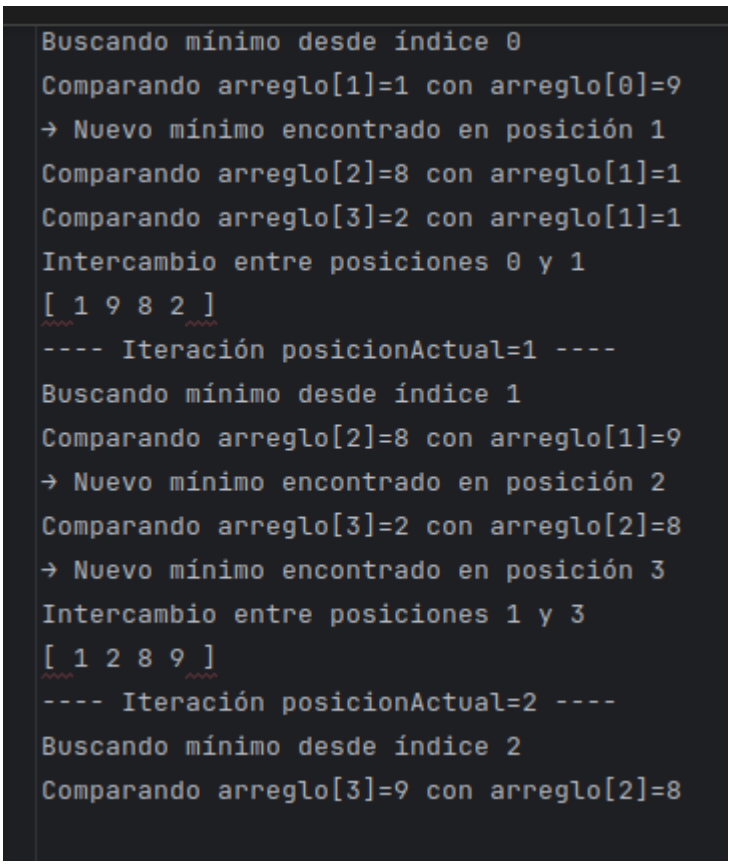

--- Iteración indiceActual =1 ----
Valor a insertar: 2
Insertar valor en posición 1
[2 2 2 2]
---- Iteración indiceActual =2 ----
Valor a insertar: 2
Insertar valor en posición 2
[2 2 2 2]
---- Iteración indiceActual =3 ----
Valor a insertar: 2
Insertar valor en posición 3
[2 2 2 2]


---- Iteración posicionActual=0 ----
Buscando mínimo desde índice 0
Comparando arreglo[1]=2 con arreglo[0]=2
Comparando arreglo[2]=2 con arreglo[0]=2
Comparando arreglo[3]=2 con arreglo[0]=2
---- Iteración posicionActual=1 ----
Buscando mínimo desde índice 1
Comparando arreglo[2]=2 con arreglo[1]=2
Comparando arreglo[3]=2 con arreglo[1]=2
---- Iteración posicionActual=2 ----
Buscando mínimo desde índice 2
Comparando arreglo[3]=2 con arreglo[2]=2

Caso 5: E:[9,1,8,2]

```
=== BUBBLE SORT ===
swap -> [1, 9, 8, 2]
swap -> [1, 8, 9, 2]
swap -> [1, 8, 2, 9]
pasada 0 | swaps = 3
swap -> [1, 2, 8, 9]
pasada 1 | swaps = 1
pasada 2 | swaps = 0
No hubo swaps → corte temprano

Total swaps = 4
```

```
Valor a insertar: 1
Mover arreglo[0] -> arreglo[1]
[ 9 9 8 2 ]
Insertar valor en posición 0
[ 1 9 8 2 ]
---- Iteración indiceActual =2 ----
Valor a insertar: 8
Mover arreglo[1] -> arreglo[2]
[ 1 9 9 2 ]
Insertar valor en posición 1
[ 1 8 9 2 ]
---- Iteración indiceActual =3 ----
Valor a insertar: 2
Mover arreglo[2] -> arreglo[3]
[ 1 8 9 9 ]
Mover arreglo[1] -> arreglo[2]
[ 1 8 8 9 ]
Insertar valor en posición 1
[ 1 2 8 9 ]
```



Discusión entre los integrantes del grupo (Anderson, Mark):

Insertion Sort: mejor para arreglos casi ordenados y para arreglos pequeños; en la práctica es eficiente.

Selection Sort: elegirlo cuando se desean pocos intercambios; su comportamiento es simple y predecible.

Bubble Sort (con corte temprano): bueno para detectar rápidamente si ya está ordenado; útil en listas casi ordenadas.

Preguntas de control:

1) ¿Por qué la Inserción es preferible con datos casi ordenados?

Porque si la lista ya está casi ordenada, Inserción prácticamente no tiene que mover nada. Solo compara unos pocos elementos y los deja en su lugar.

En ese escenario trabaja muy rápido, casi recorriendo la lista una sola vez (O(n)).

En resumen: Funciona muy bien cuando los elementos ya están casi en orden.

2) ¿Qué propiedad hace que la Selección use pocos swaps? ¿Qué compromisos tiene?

Selección siempre hace lo mismo: busca el mínimo y lo intercambia una sola vez por cada posición.

Por eso usa muy pocos swaps (máximo uno por vuelta).

El problema es que hace muchas comparaciones sin importar si la lista está ordenada o no.

En resumen: Usa pocos intercambios, pero compara demasiado.

3) ¿Cómo implementarías el corte temprano en Burbuja y qué caso mejora?

Se usa una bandera (una variable booleana) para saber si hubo algún intercambio en una pasada.

Si en una pasada no hubo swaps, significa que el arreglo ya está ordenado y se puede terminar antes.

Esto mejora el caso donde los datos ya están casi ordenados.

En resumen: Si no cambia nada en una vuelta, el algoritmo se detiene y se vuelve muy rápido.

4) ¿Cuál(es) de los tres puede(n) ser estable y en qué condiciones?

La Inserción es estable porque nunca pasa un elemento por encima de otro igual.

Burbuja también es estable cuando se intercambian solo elementos desordenados.

Selección, por su forma de buscar el mínimo y moverlo, no es estable por defecto.

En resumen: Los más estables son Inserción y Burbuja.

5) Menciona dos casos bordes que deben probarse siempre.

Un arreglo vacío, porque el algoritmo debe saber manejar que no hay nada que ordenar.

Un arreglo con un solo elemento, porque ya está ordenado y el algoritmo debería terminar rápido.