4. Caso de Estudio Nº 2: Un Manejador de Muestras

En este segundo caso del nivel vamos a desarrollar un programa que nos permita manejar una muestra de datos. Una muestra es una secuencia de valores enteros que se encuentran en un rango dado (por simplicidad vamos a suponer que el límite inferior del rango es siempre uno). Piense por ejemplo en los resultados de una encuesta, en la cual las personas califican el desempeño del presidente con un valor entre 1 y 10. O piense también en los números de la tarjeta de identidad de todos los niños nacidos en el país entre 1999 y 2001. Las operaciones que nos interesan sobre una

Fig. 1.9 – **Interfaz de usuario del manejador de muestras**

- Para iniciar el programa debemos crear una muestra, usando el botón **Nueva Muestra**. Allí definimos el tamaño y el límite superior. El programa contesta creando la muestra de manera aleatoria y presentando los primeros resultados estadísticos (mínimo, máximo y promedio).
- Inicialmente todos los botones que sólo funcionan sobre una muestra ordenada se encuentran desactivados. Es el caso, por ejemplo, de la búsqueda binaria.
- Luego, se debe ordenar la muestra utilizando cualquiera de los algoritmos disponibles en el programa. Después de esto se activan las opciones que no se encontraban disponibles.
- En este momento se pueden utilizar todas las opciones del programa: ejecutar las búsquedas, determinar cuántos elementos de la muestra se encuentran en un rango dado, cuántas veces aparece un valor dado, cuántos valores distintos hay en la muestra o cuál es el valor que más veces se presenta en la muestra. Para todos ellos se presenta el respectivo resultado, acompañado del tiempo que le tomó al computador calcularlo.

  Es posible ordenar varias veces la muestra, usando los distintos algoritmos.

muestra tienen que ver con ordenamientos y búsqueda de información, y con el tiempo que utilizan los distintos algoritmos disponibles para cumplir dichas tareas.

El programa que vamos a desarrollar debe ofrecer las siguientes opciones: (1) crear una nueva muestra de manera aleatoria, de un tamaño y con un límite superior definidos por el usuario (el usuario podría, por ejemplo, pedir una muestra de 200.000 elementos, en el rango 1 a 200.000); (2) ordenar la muestra utilizando uno de los siguientes algoritmos: inserción, selección o intercambio (burbuja), y mostrar el tiempo que gasta el computador en ejecutar dicha tarea; (3) mostrar la eficiencia de los algoritmos de búsqueda secuencial y binaria. Para esto, el programa debe calcular el tiempo que utiliza en buscar cada uno de los elementos del rango de la muestra y dividirlo por el número de elementos que buscó. Por ejemplo, si la muestra está en el rango de 1 a 200.000, el programa debe buscar cada uno de esos valores en la muestra, calcular el tiempo total del proceso y dividir por 200.000; (4) calcular el número de elementos que se encuentran en un rango dentro de la muestra ordenada (por ejemplo, determinar cuántos valores de la muestra están entre 50.000 y 100.000), (5) calcular el número de veces que aparece un valor dado en la muestra ordenada (por ejemplo, cuántas veces aparece en la muestra el valor 30.000), (6) calcular el número de valores distintos en la muestra ordenada, (7) encontrar el valor que más veces aparece en la muestra ordenada y (8) dar información estadística de la muestra no ordenada, incluyendo el mayor valor, el menor valor y el promedio.

La interfaz de usuario del programa se muestra en la figura 1.9, en donde aparecen los resultados de la ejecución para una muestra de 200.000 datos, con valores en un rango de 1 a 200.000. Allí podemos apreciar que hay diferencias considerables de tiempo al ordenar la muestra utilizando los distintos algoritmos disponibles en el programa. También se puede ver la diferencia de tiempo que existe entre una búsqueda secuencial y una búsqueda binaria. Casi 1.000 veces más veloz la segunda que la primera.

4.1. **Objetivos de Aprendizaje**

¿Qué tenemos que hacer en el caso de estudio?	¿Qué tenemos que aprender o reforzar?
Implementar los algoritmos de ordenamiento por selección, inserción e intercambio para tipos simples de datos.	Aprender cómo funcionan esos tres algoritmos de ordenamiento.
Implementar los algoritmos de búsqueda que se piden en el enunciado.	Aprender en qué consiste la búsqueda binaria. Para las demás búsquedas ya tenemos los conocimientos y habilidades necesarios.
Generar valores aleatorios en un rango.	Estudiar el método `Math.random()` de Java y adaptarlo para que genere valores en un rango dado.
Calcular el tiempo de ejecución de un método.	Estudiar los métodos que ofrece Java para manejo de tiempo y la manera en que se pueden utilizar para medir el tiempo de ejecución de una parte de un programa.
Construir un programa que funcione correctamente.	Practicar la utilización de invariantes y la construcción de pruebas unitarias.

4.2. **Comprensión de los Requerimientos**

Como primer paso para la construcción del programa, debemos identificar los requerimientos funcionales y especificarlos.

Tarea 7

Objetivo: Entender el problema del caso de estudio del manejador de muestras.

(1) Lea detenidamente el enunciado del caso de estudio del manejador de muestras e (2) identifique y complete la documentación de los ocho requerimientos funcionales que allí aparecen.

Requerimiento funcional 1	Nombre	R1 – Crear una nueva muestra
	Resumen	
	Entradas	
	Resultado	
Requerimiento funcional 2	Nombre	R2 – Ordenar la muestra
	Resumen	
	Entradas	
	Resultado	
Requerimiento funcional 3	Nombre	R3 – Calcular el tiempo de ejecución de los algoritmos de búsqueda
	Resumen	

Requerimiento funcional 3	Entradas	
	Resultado	
Requerimiento funcional 4	Nombre	R4 – Calcular el número de elementos en un rango
	Resumen	
	Entradas	
	Resultado	
Requerimiento funcional 5	Nombre	R5 – Calcular el número de veces que aparece un valor
	Resumen	
	Entradas	
	Resultado	
Requerimiento funcional 6	Nombre	R6 – Calcular el número de valores distintos
	Resumen	

Requerimiento funcional 6	Entradas	
	Resultado	
Requerimiento funcional 7	Nombre	R7 – Encontrar el valor que más veces aparece
	Resumen	
	Entradas	
	Resultado	
Requerimiento funcional 8	Nombre	R8 – Dar información estadística de la muestra
	Resumen	
	Entradas	
	Resultado	

4.3. **Arquitectura de la Solución**

En esta sección vamos a presentar los tres diagramas que definen la arquitectura de la solución propuesta: el diagrama de clases del modelo del mundo, el diagrama de clases de la interfaz de usuario y el diagrama de clases de las pruebas unitarias. En el modelo del mundo hay dos clases. La primera, llamada `Muestra`, representa una secuencia no ordenada de valores en un rango. Tiene en su interior tres atributos, tal como se muestra en la figura 1.10: un arreglo con los valores de la muestra (`valores`), el cual tiene reservado el espacio definido en el constructor, pero que sólo tiene un número dado de elementos presentes (`tamanio`), todos estos valores

en el rango `1..limiteSuperior`. En la figura 1.11 aparece un posible diagrama de objetos, en donde se puede observar una instancia de esta clase.

La segunda clase, llamada `MuestraOrdenada`, representa una secuencia ordenada de valores. Esta clase sólo tiene como atributo un arreglo que almacena los valores de la muestra. Dicho arreglo se encuentra completamente lleno y la clase no tendrá métodos para agregar valores, eliminarlos o modificarlos, puesto que no hay ningún requerimiento del caso que así lo exija.

En el diagrama de clases se puede ver una asociación marcada con una línea punteada, que indica que las dos clases tienen una dependencia, aunque no exista una asociación directa entre ellas. En este caso la dependencia consiste en que vamos a tener un método de la clase `Muestra` que es capaz de crear instancias de la clase `MuestraOrdenada`. En la figura 1.11 se ilustra esto con un diagrama de objetos.

Fig. 1.10 – **Diagrama de clases del modelo del mundo**

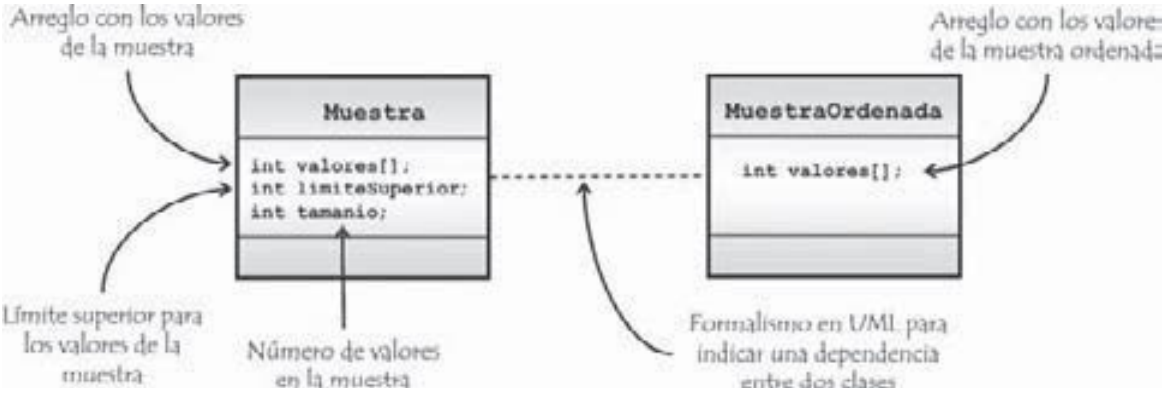
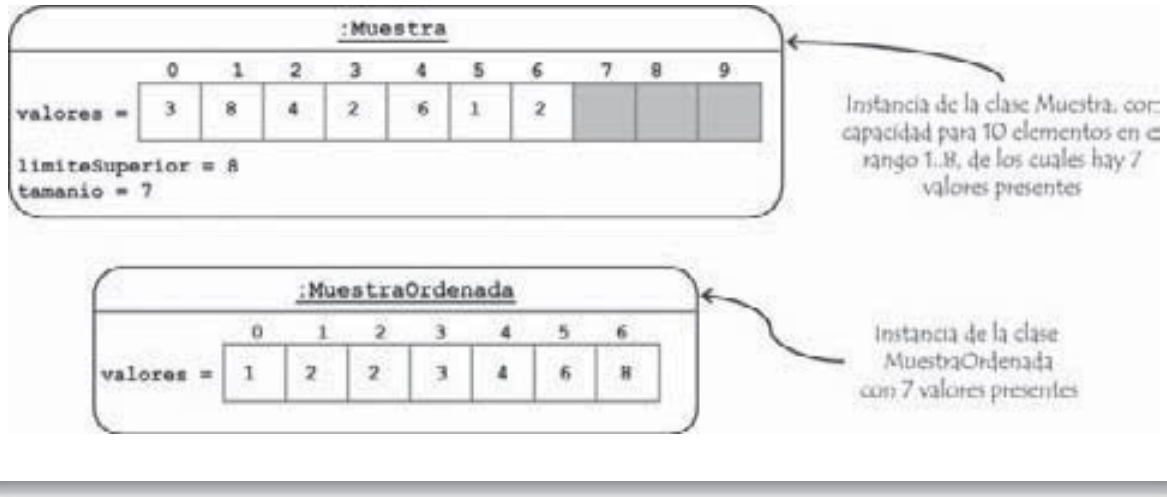


Fig. 1.11 – **Diagrama de objetos como un ejemplo de un estado posible del modelo del mundo**



En la siguiente tarea pediremos al lector que defina el invariante de las clases antes descritas y que implemente los métodos que nos van a permitir su verificación durante la ejecución.

Tarea 8

Objetivo: Definir el invariante de cada una de las clases del caso de estudio e implementar el método que lo verifica.

Para las clases `Muestra` y `MuestraOrdenada`, siga los pasos que se proponen a continuación:

1. Defina el invariante de cada una de las clases del caso de estudio:

Clase	Atributo	Invariante análisis	Invariante diseño
`Muestra`	`valores`		
`Muestra`	`limiteSuperior`		
`Muestra`	`tamanio`		
`MuestraOrdenada`	`valores`		

2. Escriba el método en cada clase que verifica en ejecución que el invariante se cumple:

```
public class Muestra
{
    private void verificarInvariante( )
    {









    }
}
```

```
public class MuestraOrdenada
{
    private void verificarInvariante( )
    {









    }
}
```

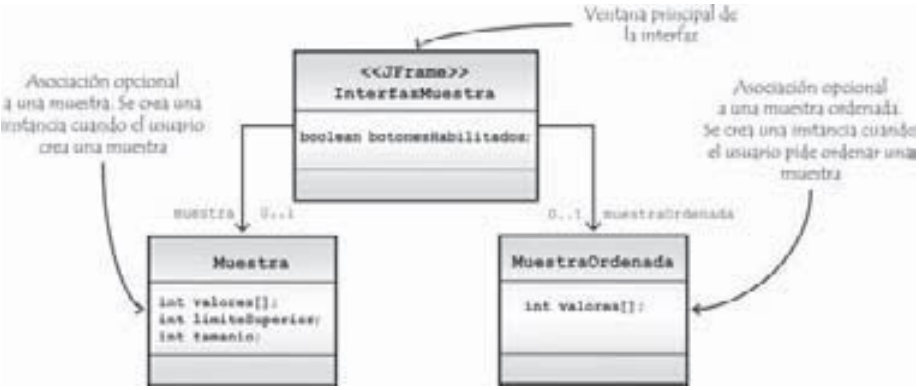
En la siguiente tabla hacemos un resumen de los principales métodos públicos de las dos clases del caso de estudio. Para ver los contratos exactos se recomienda consultar el CD que acompaña al libro.

Clase `Muestra`:	
`Muestra( int capacidad, int limite )`	Crea una muestra vacía (sin elementos), en un rango y con una capacidad dados como parámetro.
`void agregarDato( int valor )`	Agrega un valor al final de la muestra. La precondición afirma que hay espacio en la muestra para agregar el nuevo valor.
`void generarValores( )`	Genera de manera aleatoria valores para la muestra, llenándola completamente hasta su capacidad máxima.
`int darTamanio( )`	Retorna el número de elementos presentes en la muestra.
`int darCapacidad( )`	Retorna el número máximo de elementos que puede contener la muestra.
`int darLimiteSuperior( )`	Retorna el límite superior de valores de la muestra.
`int darMaximo( )`	Retorna el máximo valor presente en la muestra. Si la muestra está vacía retorna el valor 0.
`int darMinimo( )`	Retorna el mínimo valor presente en la muestra. Si la muestra está vacía retorna el valor 0.
`double darPromedio( )`	Retorna el promedio de los valores presentes en la muestra. Si la muestra está vacía retorna el valor 0.
`MuestraOrdenada ordenarSeleccion( )`	Crea y retorna una instancia de la clase MuestraOrdenada con todos los elementos de la muestra, utilizando para esto el algoritmo de ordenamiento por selección.
`MuestraOrdenada ordenarBurbuja( )`	Crea y retorna una instancia de la clase MuestraOrdenada con todos los elementos de la muestra, utilizando para esto el algoritmo de ordenamiento por intercambio.
`MuestraOrdenada ordenarInsercion( )`	Crea y retorna una instancia de la clase MuestraOrdenada con todos los elementos de la muestra, utilizando para esto el algoritmo de ordenamiento por inserción.
`boolean buscarSecuencial( int valor )`	Retorna verdadero si el valor que llega como parámetro se encuentra presente en la muestra.

Clase `MuestraOrdenada`:	
`MuestraOrdenada( int[ ] vals )`	Crea una muestra ordenada a partir de la información que recibe en un arreglo de valores enteros. La precondición dice que dicho arreglo no es nulo y que viene ordenado ascendentemente.
`int darTamanio( )`	Retorna el número de elementos de la muestra ordenada (que en este caso es igual al tamaño del arreglo de valores).
`int contarOcurrencias( int valor )`	Calcula y retorna el número de veces que aparece un valor dado en la muestra ordenada.
`int contarValoresDistintos( )`	Calcula y retorna el número de valores distintos que hay en la muestra ordenada.
`int contarElementosEnRango(int inf, int sup )`	Calcula y retorna el número de elementos de la muestra ordenada que están en el rango inf…sup, incluyendo los dos límites.
`int darValorMasFrecuente( )`	Calcula y retorna el valor que más veces aparece en la muestra ordenada.
`boolean buscarBinario( int valor )`	Retorna verdadero si el valor que llega como parámetro se encuentra presente en la muestra ordenada, utilizando para esto el algoritmo de búsqueda binaria.

En la figura 1.12 aparece una parte del diagrama de clases de la interfaz de usuario, en la que se puede apreciar su relación con el modelo del mundo. En este diagrama se puede ver que existe una asociación opcional (0..1) hacia la clase `Muestra` y otra asociación también opcional hacia la clase `MuestraOrdenada`.

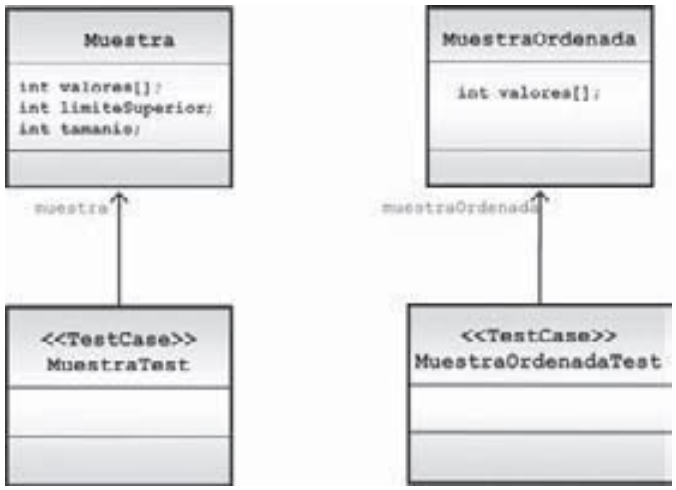
Fig. 1.12 – **Diagrama de clases de la interfaz de usuario**



Tan pronto el usuario genera una muestra, se crea la primera instancia. En el momento de utilizar cualquiera de los métodos de ordenamiento, se crea la instancia de la segunda clase. Existe también un atributo de tipo lógico (`botonesHabilitados`) en la clase de la ventana principal, que indica si se pueden ofrecer al usuario las opciones que trabajan sobre una muestra ordenada, cuyo valor se vuelve verdadero después de haber ordenado la muestra.

Para las pruebas unitarias construiremos dos clases (`MuestraTest` y `MuestraOrdenadaTest`), relacionadas con el modelo del mundo como aparece en la figura 1.13. Los escenarios y los casos de prueba son tema de una sección posterior. Por ahora nos contentamos con definir la parte estructural de la solución, e identificar las responsabilidades de cada uno de los componentes de ella.

Fig. 1.13 – **Diagrama de clases de las pruebas**



4.4. Algoritmos de Ordenamiento en Memoria Principal

Ahora que ya está completamente definida la arquitectura de la solución, nos podemos concentrar en la parte algorítmica, la cual constituye el principal objetivo de este caso de estudio. En esta sección trabajaremos sobre tres de los algoritmos de ordenamiento más simples que existen y veremos la manera de usarlos para escribir los métodos de la clase `Muestra`, especificados en la sección anterior.

Todos los algoritmos de ordenamiento que presentamos en esta sección manejan un orden ascendente y trabajan en una dirección determinada (de izquierda a derecha o de derecha a izquierda). Los cambios que se deben hacer para ordenar un arreglo de valores descendentemente o para trabajar en la dirección contraria son mínimos, por lo que no serán tratados aquí de manera aparte.

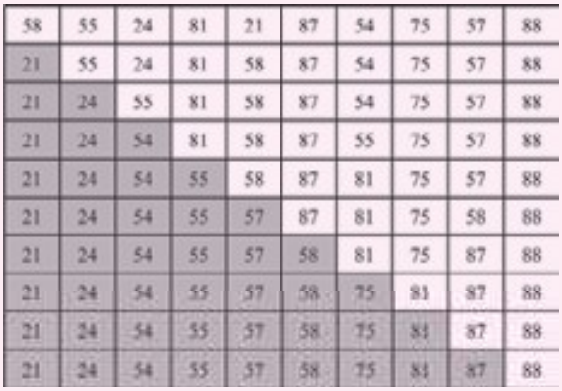
4.4.1. Ordenamiento por Selección

Aunque todos los algoritmos de ordenamiento satisfacen el mismo contrato, existen distintas maneras de cumplirlo, cada una de ellas con pequeñas ventajas y desventajas. El primero de los algoritmos que vamos a estudiar es el que utiliza una técnica denominada de selección, y se basa en la idea que se ilustra en el ejemplo 9.

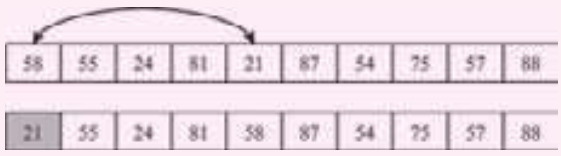
Ejemplo 9

Objetivo: Mostrar el funcionamiento del algoritmo de ordenamiento por selección.

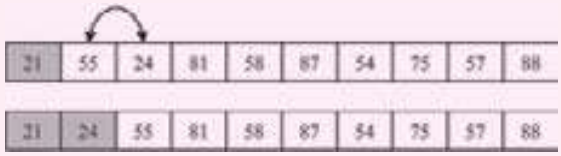
En este ejemplo se muestra cada una de las etapas por las que pasa el algoritmo de ordenamiento por selección para ordenar ascendentemente un arreglo de valores de tipo simple.



- En este ejemplo tenemos inicialmente un arreglo no ordenado de 10 posiciones (primera fila de la figura).
- Después de la primera iteración del algoritmo, queremos que el menor elemento (21) quede en la primera posición. Para esto buscamos el menor valor de todo arreglo y lo intercambiamos con el que está en la primera posición (58). Así obtenemos el arreglo que aparece en la segunda fila del ejemplo.
- Luego, repetimos el mismo proceso pero comenzando desde el segundo elemento del arreglo. Buscamos el menor (24) y lo remplazamos con el primer elemento que se encuentra en la parte sin ordenar (55).
- El proceso continúa hasta que todo el arreglo queda ordenado, tal como se ilustra en la siguiente secuencia de pasos.



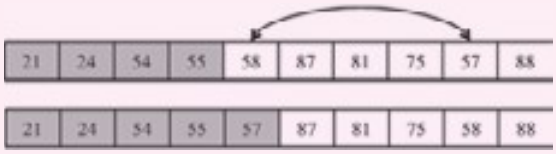
- Primera iteración: Todo el arreglo se encuentra sin ordenar. Buscamos el menor elemento (21) y lo intercambiamos con el primer elemento de la parte no ordenada (58).



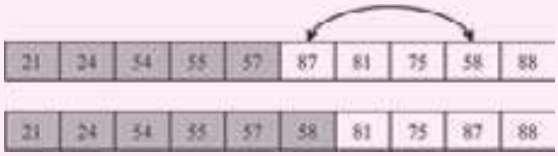
- Segunda iteración: Ya hay un elemento ordenado (21). Buscamos el menor de la parte sin ordenar (24) y lo intercambiamos con el primer elemento del arreglo que no está ordenado (55).

- Tercera iteración: Ya hay dos elementos ordenados en el arreglo (21, 24). Intercambiamos el menor de la parte restante (54) con el primero de la misma (55).

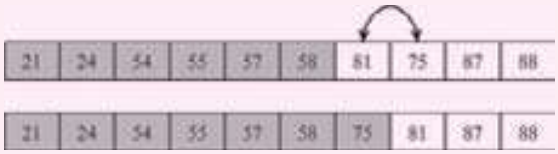
- Cuarta iteración: Incluimos en la parte ordenada el elemento 55, intercambiándolo por el 81. Con eso completamos ya cuatro elementos ordenados ascendentemente.



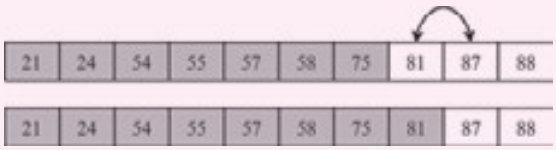
- Quinta iteración: Repetimos el mismo proceso anterior para intercambiar los elementos 57 y 58. Fíjese que en todo momento los valores de la parte ordenada son menores que todos los que se encuentran en la parte no ordenada.



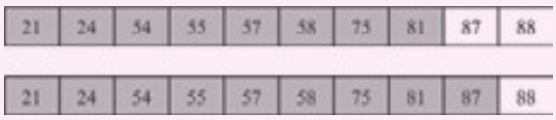
- Sexta iteración: Ya hay cinco elementos ordenados. Seleccionamos el menor de la parte no ordenada (58) y lo intercambiamos con el primero de la parte no ordenada (87).



- Séptima iteración: Intercambiamos los elementos 75 y 81 para avanzar en el proceso de ordenamiento.



- Octava iteración: Intercambiamos los elementos 87 y 81 para completar ocho elementos ordenados en el arreglo.



- Novena iteración: En este caso no hay que hacer ningún intercambio, puesto que el menor de la parte no ordenada (87) ya se encuentra en la primera posición.
- Cuando hayamos ordenado todos los elementos menos uno terminamos el proceso, puesto que el restante (88) es necesariamente mayor que todos los que ya fueron incluidos en la parte ordenada.

A continuación se muestra el código del método de la clase `Muestra` encargado de crear una instancia de la clase `MuestraOrdenada` usando la técnica de ordenamiento por selección:

```
public MuestraOrdenada ordenarSeleccion( )
{
    int[] arreglo = darCopiaValores( );
    for( int i = 0; i < tamanio - 1; i++ )
    {
        int menor = arreglo[ i ];
        int cual = i;
        for( int j = i + 1; j < tamanio; j++ )
        {
```

- Inicialmente crea una copia de los valores de la muestra, para ordenarlos, usando el método darCopiaValores().
- El ciclo externo del método lleva el índice "i" señalando el punto en el cual comienza la parte sin ordenar del arreglo. Comienza en 0 y termina cuando llega a la penúltima posición.

```
            if( arreglo[ j ] < menor )
            {
                menor = arreglo[ j ];
                cual = j;
            }
        }
        int temp = arreglo[ i ];
        arreglo[ i ] = menor;
        arreglo[ cual ] = temp;
    }
    return new MuestraOrdenada( arreglo );
}
```

- El objetivo del ciclo interior es buscar el menor valor de la parte sin ordenar. Dicha parte comienza en la posición "i" y va hasta el final del arreglo. Al final del ciclo, deja en la variable "menor" el menor valor encontrado y en la variable "cual" su posición dentro del arreglo.
- Después de haber localizado el menor elemento, lo intercambia con el que se encuentra en la casilla "i" del arreglo. Para esto utiliza una variable temporal "temp".
- Cuando finalmente ha terminado de ordenar el arreglo, crea una instancia de la clase MuestraOrdenada.

Tarea 9

Objetivo: Utilizar la técnica de ordenamiento por selección, para ordenar ascendentemente una secuencia de valores.

Suponiendo que los valores que se encuentran en la siguiente tabla corresponden a una secuencia de números enteros que quiere ordenar, muestre el avance en cada una de las iteraciones para el caso en el cual utilice la técnica de ordenamiento por selección. Marque claramente en cada iteración la parte que ya está ordenada y el menor elemento de la parte por ordenar.

25	13	45	12	1	76	42	90	56	27	33	69	72	99	81

El algoritmo de ordenamiento por selección se basa en la idea de tener dividido el arreglo que se está ordenando en dos partes: una, con un grupo de elementos ya ordenados, que ya encontraron su posición final. La otra, con los elementos que no han sido todavía ordenados. En cada iteración, localizamos el menor elemento de la parte no ordenada, lo intercambiamos con el primer elemento de esta misma región e indicamos que la parte ordenada ha aumentado en un elemento.

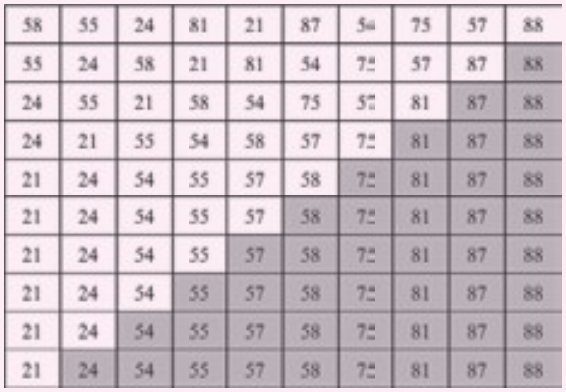
4.4.2. Ordenamiento por Intercambio (Burbuja)

Otra técnica que se usa frecuentemente para ordenar secuencias de valores se basa en la idea de ir intercambiando todo par de elementos consecutivos que no se encuentren ordenados, tal como se ilustra en el ejemplo 10.

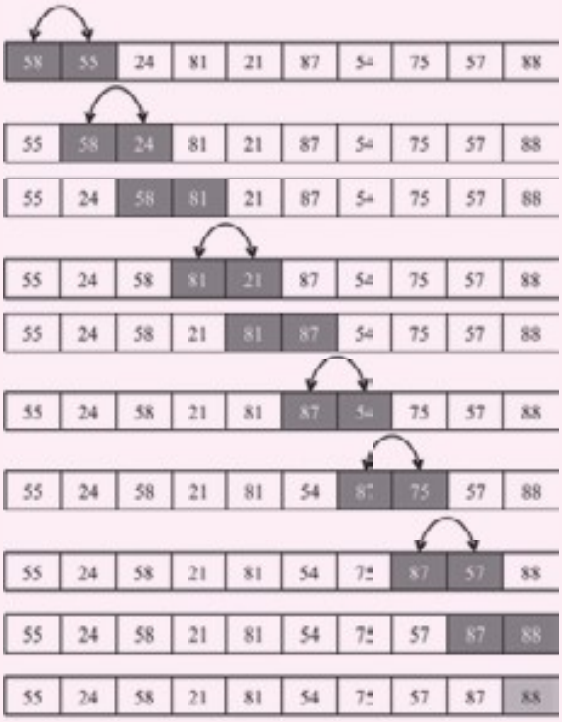
Ejemplo 10

Objetivo: Mostrar el funcionamiento del algoritmo de ordenamiento por intercambio (también llamado ordenamiento de burbuja).

En este ejemplo se muestra cada una de las etapas por las que pasa el algoritmo de ordenamiento de burbuja para ordenar ascendentemente un arreglo de valores de tipo simple.



- En este ejemplo, tenemos inicialmente un arreglo no ordenado de 10 posiciones (primera fila de la figura).
- En la primera iteración recorremos el arreglo de izquierda a derecha intercambiando todo par de valores consecutivos que no se encuentren ordenados. Al final de la primera iteración, el mayor de los elementos (88) debe quedar en la última posición del arreglo, mientras todos los demás valores han avanzado un poco hacia su posición final.
- El valor 87, por ejemplo, alcanzó a avanzar tres posiciones hacia su posición definitiva.
- Este mismo proceso se repite para la parte del arreglo que todavía no está ordenada (en blanco en la figura).
- El proceso termina cuando sólo queda un elemento en la zona no ordenada del arreglo (al comienzo), lo cual requiere nueve iteraciones en nuestro ejemplo (número de elementos menos uno).



- En esta figura aparecen todos los intercambios que tienen lugar en la primera iteración.
- Si vamos en la posición "i" del arreglo, y el valor en esa posición es mayor que el valor de la posición "i+1", los intercambiamos.
- El proceso en esta iteración va hasta el último elemento del arreglo. A medida que vayamos avanzando en la ejecución del algoritmo, iremos acortando el punto hasta el cual hay que hacer los intercambios.
- Al final de la primera iteración, el último elemento ya se encuentra ordenado.
- Este método se denomina de burbuja, porque hace que los elementos vayan subiendo poco a poco hasta ocupar su posición final.
- Al final de cada iteración, en la parte final del arreglo están los elementos ya ordenados, los cuales además son mayores que todos los elementos que faltan por ordenar.

El algoritmo de ordenamiento por intercambio (conocido también como ordenamiento de burbuja) se basa en la idea de intercambiar todo par de elementos consecutivos que no se encuentren en orden. Al final de cada pasada haciendo este intercambio, un nuevo elemento queda ordenado y todos los demás elementos se acercaron a su posición final.

A continuación se muestra el código del método de la clase `Muestra` encargado de crear una instancia de la clase `MuestraOrdenada` usando la técnica de ordenamiento por intercambio (burbuja):

```
public MuestraOrdenada ordenarBurbuja( )
{
    int[] arreglo = darCopiaValores( );

    for( int i = tamanio; i > 0; i-- )
    {
        for( int j = 0; j < i - 1; j++ )
        {

            if( arreglo[ j ] > arreglo[ j + 1 ] )
            {
```

- Inicialmente crea una copia de los valores de la muestra, para ordenarlos, usando el método darCopiaValores().
- El ciclo externo va controlando con la variable "i" el punto hasta el cual hay que llevar el proceso de intercambio. Inicialmente va hasta el final de la secuencia, y va disminuyendo hasta que sólo queda un elemento (termina cuando i==0).

```
                int temp = arreglo[ j ];
                arreglo[ j ] = arreglo[ j + 1 ];
                arreglo[ j + 1 ] = temp;
            }
        }
    }

    return new MuestraOrdenada( arreglo );
}
```

- En el ciclo interno se lleva a cabo el proceso de intercambio con la variable "j", comenzando desde la posición 0 hasta llegar al punto anterior al límite marcado por la variable "i". Allí compara (y si es necesario intercambia) los valores de las posiciones "j" y "j+1".
- Cuando finalmente ha terminado de ordenar el arreglo, crea una instancia de la clase MuestraOrdenada.

Tarea 10

Objetivo: Utilizar la técnica de ordenamiento por intercambio (burbuja), para ordenar ascendentemente una secuencia de valores.

Suponiendo que los valores que se encuentran en la siguiente tabla corresponden a una secuencia de números enteros que quiere ordenar, muestre el avance en cada una de las iteraciones para el caso en el cual utilice la técnica de ordenamiento por intercambio (burbuja). Marque claramente en cada iteración la parte del arreglo que ya se encuentra ordenada.

25	13	45	12	1	76	42	90	56	27	33	69	72	99	81