# **Caso de análisis: técnicas de ordenamiento implementadas en Java.**

Cerraremos esta ficha de estudios mostrando una clase *Vector* muy general que implementa y encapsula el concepto de arreglo unidimensional (o vector) e incorpora entre sus métodos básicos a todos los métodos de ordenamiento clásicos que se han presentado en materias anteriores.

La clase ***Vector***simplemente declara como único atributo una referencia v para apuntar a un arreglo unidimensional de números enteros. La misma clase incluye un constructor sencillo, que toma como parámetro el tamaño *n* del arreglo y crea el arreglo con *new*. Además, incluye un par de métodos de acceso llamados *get(i)* (para retornar el valor contenido en la casilla *i* del arreglo) y *set(i, x)* para asignar el valor *x* en la casilla *i*:

public class Vector  
{  
 private int v[];  
   
 public Vector(int n)  
 {  
 if(n <= 0) {

n = 100; }  
 v = new int[n];  
 }  
   
 public int get(int i)  
 {  
 return v[i];  
 }  
   
 public void set(int i, int x)  
 {  
 v[i] = x;  
 }

// resto del contenido de la clase aquí...

}

La clase incluye también un par de métodos muy útiles si se trabaja con pruebas sobre técnicas de ordenamiento: uno de esos métodos se llama *generate()* y simplemente llena el arreglo con n números enteros aleatorios, siendo *n* el tamaño del arreglo. El otro método se llama *checkOrder()* y sólo comprueba si el arreglo está ordenado (retornando *true* en ese caso) o no (en cuyo caso retorna *false*): ç

## **Metodos para generar un vector con random() y verificar el orden**

public void generate()  
{  
 for(int i=0; i<v.length; i++ )  
 {  
 v[i] = (int)Math.*round*(100 \* Math.*random*());  
 }  
}

public boolean checkOrder()  
{  
 for(int i=0; i < v.length - 1; i++)  
 {  
 if(v[i] > v[i+1]) { return false; }  
 }  
 return true;  
}

## **Metodos para Buscar**

Los dos métodos que siguen implementan las ya conocidas técnicas de búsqueda secuencial (método *linearSearch()*) y búsqueda binaria (método *binarySearch()*):

public int linearSearch(int x)  
{  
 for(int i=0; i<v.length; i++)  
 {  
 if(x == v[i]) return i;  
 }  
 return -1;  
}  
  
public int binarySearch(int x)  
{  
 int n = v.length;  
 int izq = 0, der = n-1;  
 while(izq <= der)  
 {  
 int c = (izq + der)/2;  
 if(x == v[c]) return c;  
   
 if(x < v[c]) der = c - 1;  
 else izq = c + 1;  
 }  
 return -1;  
}

Y finalmente aparece un amplio conjunto de métodos que implementan los métodos clásicos de ordenamiento. Los primeros tres de ese grupo corresponden a los *algoritmos simples* o *directos*

## **Ordenamiento por Burbuja o de intercambio directo**

public void bubbleSort()  
{  
 boolean ordenado = false;  
 int n = v.length;  
 for(int i=0; i<n-1 && !ordenado; i++)  
 {  
 ordenado = true;  
 for(int j=0; j<n-i-1; j++)  
 {  
 if(v[j] > v[j+1])  
 {  
 int aux = v[j];  
 v[j] = v[j+1];  
 v[j+1] = aux;  
 ordenado = false;  
 }  
 }  
 }   
}

## Método por Ordenamiento por selección Directa

public void selectionSort()  
{  
 int n = v.length;  
 for(int i=0; i<n-1; i++)  
 {  
 for(int j=i+1; j<n; j++)  
 {  
 if(v[i] > v[j])  
 {  
 int aux = v[i];  
 v[i] = v[j];  
 v[j] = aux;  
 }  
 }  
 }   
}  
Metodo por inserción Directa  
public void insertionSort()  
{  
 int n = v.length;  
 for(int j=1; j<n; j++)  
 {  
 int k, y = v[j];  
 for(k=j-1; k>=0 && y<v[k]; k--)  
 {  
 v[k+1]= v[k];  
 }  
 v[k+1]= y;  
 }  
}

Los métodos que restan (casi todos públicos y unos pocos privados que sirven como auxiliares de algunos de los públicos) implementan los métodos clásicos de *ordenamiento compuesto*. Los dos primeros (publico el primero, privado el segundo) corresponden al algoritmo *Quick Sort*: el método público *quickSort()* es el que se invoca desde el exterior de la clase para lanzar el proceso, y el método privado y recursivo *quick()* es el que realmente hace todo el trabajo:

//Quicksort: ordenacion rapida  
  
public void quickSort()  
{  
 quick(0, v.length-1);  
}  
  
private void quick(int izq, int der)  
{  
 int i = izq, j = der, y;  
 int x = v[(izq + der) / 2];  
 do   
 {  
 while(v[i]<x && i<der) { i++; }  
 while(x<v[j] && j>izq) { j--; }  
 if(i <= j)  
 {  
 y = v[i];  
 v[i] = v[j];  
 v[j] = y;  
 i++;  
 j--;  
 }  
 }  
 while(i <= j);  
 if(izq < j) { quick( izq, j ); }  
 if(i < der) { quick( i, der ); }  
}

Los dos que siguen (*heapSort()* y *shellSort()*) corresponden a los algoritmo *Heap Sort* y *Shell Sort*:

public void heapSort()  
{  
 int n = v.length;  
  
 // crear el grupo inicial...  
 for(int i=1; i<n; i++)  
 {  
 int e = v[i];  
 int s = i;  
 int f = (s-1)/2;  
 while(s>0 && v[f] < e)  
 {  
 v[s] = v[f];  
 s = f;  
 f = (s-1)/2;  
 }  
 v[s] = e;  
 }  
  
 // se extrae la raiz, y se reordena el vector y el grupo...  
 for(int i = n-1; i>0; i-- )  
 {  
 int valori = v[i];  
 v[i] = v[0];  
 int f = 0, s;  
 if(i==1) { s = -1; } else { s = 1; }  
 if(i>2 && v[2]>v[1]) { s = 2; }  
 while(s>=0 && valori<v[s])  
 {  
 v[f] = v[s];  
 f = s;  
 s = 2\*f + 1;  
 if(s+1<=i-1 && v[s]<v[s+1]) { s++; }  
 if(s>i-1) { s = -1; }  
 }  
 v[f] = valori;  
 }  
}  
  
public void shellSort()  
{  
 int h, n = v.length;  
 for(h = 1; h <= n / 9; h = 3\*h + 1);  
 for (; h > 0; h /= 3)  
 {  
 for (int j = h; j < n; j++)  
 {  
 int k, y = v[j];  
 for(k = j - h; k >= 0 && y < v[k]; k-=h)  
 {  
 v[k+h] = v[k];  
 }  
 v[k+h] = y;  
 }  
 }  
}

Por fin, los tres métodos finales implementan el algoritmo *Merge Sort*. El primero de los tres (*mergeSort()*) es público y es el que se invoca desde fuera de la clase para lanzar el proceso. El segundo método (*sort()*) y privado y recursivo y se encarga de dividir en dos al arreglo en cada entrada recursiva, invocando en cada una de esas entradas al tercer método (*merge()*), también privado, que hace la fusión entre las mitades ordenadas (los detalles lógicos de este algoritmo, así del Heap Sort y un repaso del Quick Sort, serán oportunamente analizados):

public void mergeSort()  
 {  
 int n = v.length;  
 int temp[] = new int[n];  
 sort(0, n-1, temp);  
 }  
  
 private void sort(int izq, int der, int temp[])  
 {  
 if(izq < der)   
 {  
 int centro = (izq + der) / 2;  
 sort(izq, centro, temp);  
 sort(centro + 1, der, temp);  
 merge(izq, centro, der, temp);  
 }   
 }   
  
 private void merge(int izq, int centro, int der, int temp[])  
 {  
 for(int i = izq; i <= der; i++) { temp[i] = v[i]; }  
  
 int i = izq, j = centro + 1, k = izq;  
 while(i <= centro && j <= der)   
 {  
 if(temp[i] <= temp[j])   
 {  
 v[k] = temp[i];  
 i++;  
 }   
 else   
 {  
 v[k] = temp[j];  
 j++;  
 }  
 k++;  
 }  
  
 while(i <= centro)   
 {  
 v[k] = temp[i];  
 k++;  
 i++;  
 }  
 }   
}

El proyecto *TSB [Ordenamiento]* que acompaña a esta Ficha inclye la clase Vector que acabamos de describir, y una clase *Main* que contiene un método *main()* para poner a funcionar todo a través de un menú de opciones:

package modelo;  
import java.util.Scanner;  
  
public class Main  
{  
 public static void main (String[] args)  
 {  
 long t1, t2, tf;  
 int op;  
   
 Scanner sc = new Scanner(System.*in*);  
 modelo.Vector v = null;  
 do  
 {  
 System.*out*.println ("\nOpciones de Ordenamiento");  
 System.*out*.println ("0. Generar el Arreglo");  
 System.*out*.println ("1. Intercambio Directo (Burbuja)");  
 System.*out*.println ("2. Seleccion Directa");  
 System.*out*.println ("3. Insercion Directa");  
 System.*out*.println ("4. Quick Sort");  
 System.*out*.println ("5. Heap Sort");  
 System.*out*.println ("6. Shell Sort");  
 System.*out*.println ("7. Merge Sort");   
 System.*out*.println ("8. Verificar si esta ordenado");  
 System.*out*.println ("9. Salir");  
 System.*out*.print ("Ingrese opcion: ");  
 op = sc.nextInt();  
 switch (op)  
 {  
 case 0:  
 System.*out*.print ("Ingrese cantidad del elementos del vector:

");  
 int n = sc.nextInt();  
 v = new modelo.Vector(n);  
 System.*out*.print("Se vuelve a generar el vector...");  
 v.generate();  
 System.*out*.print("\nVector generado...");  
 break;  
  
 case 1:  
 if(v == null) {  
 System.*out*.println("El arreglo no ha sido generado aún...");  
 continue;  
 }  
 System.*out*.print("Se ordena el vector por Intercambio...");  
 t1 = System.*currentTimeMillis*();  
 v.bubbleSort();  
 t2 = System.*currentTimeMillis*();  
 tf = t2 - t1;  
 System.*out*.print("\nVector ordenado en: " + tf + "

milisegundos...");  
 break;  
  
 case 2:  
 if(v == null) {  
 System.*out*.println("El arreglo no ha sido generado aún...");  
 continue;  
 }  
 System.*out*.print("Se ordena el vector por Seleccion...");  
 t1 = System.*currentTimeMillis*();  
 v.selectionSort();  
 t2 = System.*currentTimeMillis*();  
 tf = t2 - t1;  
 System.*out*.print("\nVector ordenado en: " + tf + "

milisegundos...");  
 break;  
  
 case 3:  
 if(v == null) {  
 System.*out*.println("El arreglo no ha sido generado aún...");  
 continue;  
 }  
 System.*out*.print("Se ordena el vector por Insercion... ");  
 t1 = System.*currentTimeMillis*();  
 v.insertionSort();  
 t2 = System.*currentTimeMillis*();  
 tf = t2 - t1;  
 System.*out*.print("\nVector ordenado en: " + tf + "

milisegundos...");  
 break;  
  
 case 4:  
 if(v == null) {  
 System.*out*.println("El arreglo no ha sido generado aún...");  
 continue;  
 }  
 System.*out*.print("Se ordena el vector por Quick Sort...");  
 t1 = System.*currentTimeMillis*();  
 v.quickSort();  
 t2 = System.*currentTimeMillis*();  
 tf = t2 - t1;  
 System.*out*.print("\nVector ordenado en: " + tf + "

milisegundos...");  
 break;  
  
 case 5:  
 if(v == null) {  
 System.*out*.println("El arreglo no ha sido generado aún...");  
 continue;  
 }  
 System.*out*.print("Se ordena el vector por Heap Sort...");  
 t1 = System.*currentTimeMillis*();  
 v.heapSort();  
 t2 = System.*currentTimeMillis*();  
 tf = t2 - t1;  
 System.*out*.print("\nVector ordenado en: " + tf + "

milisegundos...");  
 break;  
  
 case 6:  
 if(v == null) {  
 System.*out*.println("El arreglo no ha sido generado aún...");  
 continue;  
 }  
 System.*out*.print("Se ordena el vector por Shell Sort...");  
 t1 = System.*currentTimeMillis*();  
 v.shellSort();  
 t2 = System.*currentTimeMillis*();  
 tf = t2 - t1;  
 System.*out*.print("\nVector ordenado en: " + tf + "

milisegundos...");  
 break;  
  
 case 7:  
 if(v == null) {  
 System.*out*.println("El arreglo no ha sido generado aún...");  
 continue;  
 }  
 System.*out*.print("Se ordena el vector por Merge Sort...");  
 t1 = System.*currentTimeMillis*();  
 v.mergeSort();  
 t2 = System.*currentTimeMillis*();  
 tf = t2 - t1;  
 System.*out*.print("\nVector ordenado en: " + tf + "

milisegundos...");  
 break;  
  
 case 8:  
 if(v == null) {  
 System.*out*.println("El arreglo no ha sido generado aún...");  
 continue;  
 }  
 System.*out*.println("Se verifica si esta ordenado...");  
 if(v.checkOrder()) { System.*out*.println("Esta ordenado..."); }  
 else { System.*out*.println ("No esta ordenado..."); }  
 break;  
  
 case 9: ;  
 }  
 }  
 while (op != 9);  
 }   
}

Note que la clase *Prueba* se ha pensado como una simple clase para incluir un *main()* y un menú de opciones para probar el contenido de la clase *Vector*. Podría pensarse que la clase *Vector* no es necesaria: todo el contenido de esa clase podría haber sido incluido en la **Cátedra de TSB** *[Ciclo 2021] Ficha 02: Instrucciones de Control de Flujo* misma clase *Prueba* (o al revés: el método *main()* podría haber sido incluido en la clase *Vector*) y con eso tendríamos una sóla clase, dando la impresión de hacer todo más sencillo…

Sin embargo, el hecho de haber diseñado la clase *Vector* por separado, tomando incluso ciertos recaudos para hacerla genérica, sin dependencias externas ni acoplamiento entre procesos e interfaz de usuario, deja a esa clase lista para ser reutilizada en cuando proyecto sea requerida, sin tener que contar con un "molesto" método *main()* cuyo objetivo era solamente proporcionar un marco de testing. Las responsabilidades funcionales de la clase *Vector* (contener un arreglo unidimensional y proporcionar algoritmos básicos de ordenamiento y búsqueda para ese arreglo) son claramente diferentes de las responsabillidades de la clase *Prueba* (ofrecer un contexto simple de testing para la clase Vector).

Y esa es una línea de diseño muy fuerte en POO: trate de diseñar clases diferentes para responsabilidades diferentes y bien definidas, aún cuando finalmente le queden clases muy sencillas o breves. Mientras más especializada está una clase para cubrir sus responsabilidades, más reusable será esa clase y por cierto, más sencilla para mantener y modificar.

A lo largo del curso veremos distintas formas de replantear la misma clase, aplicando elementos de control de errores, testing automático, patrones de diseño, control de proyectos y diversas convenciones de trabajo de la POO que por ahora escapan al espíritu introductivo simple de esta ficha. Tenga paciencia…