

# Laboratorio 1 – Análisis de Algoritmos de Ordenación

---

Este laboratorio tiene tres objetivos fundamentales:

1. Ampliar la descripción realizada en la parte de teoría sobre el **análisis de algoritmos**.
2. Conocer destacados algoritmos de ordenación: *inserción*, *selección* y *burbuja*. Estos algoritmos se encuentran explicados en el documento anexo a este laboratorio “Laboratorio1\_Anexo1.pdf”.
3. Comenzar a utilizar el framework más conocido para pruebas unitarias<sup>1</sup> en Java: JUnit. En concreto, se usará la versión 5. En la carpeta del Laboratorio 0 se encuentra una breve descripción sobre el uso de dicho framework (documento “Primeros pasos con JUnit5”).

## Tareas

Conocidos los algoritmos de ordenación, la **primera tarea** será la de implementar tres de ellos: *selección*, *burbuja* y *quicksort*. Con respecto al algoritmo de ordenación quicksort (recordad que fue trabajado el año pasado en Programación II) utilizad la versión iterativa de la partición, con el fin de realizar un análisis de este con diferentes tamaños del vector de entrada.

El código del algoritmo de *inserción* es ofrecido a continuación y se os proporciona para que lo podáis usar como inspiración para construir vuestra solución:

```
public class IntArraySorter {  
  
    private final int[] array;  
  
    private int numComparisons;  
    private int numSwaps;  
  
    public int getNumComparisons() {  
        return numComparisons;  
    }  
}
```

---

<sup>1</sup> Test que comprueba el comportamiento de una unidad de trabajo (generalmente, no siempre, será un método).

```
public int getNumSwaps() {
    return numSwaps;
}

public IntArraySorter(int[] array) {
    this.array = array;
}

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

public void swap(int i, int j) {
    int tmp = array[i];
    array[i] = array[j];
    array[j] = tmp;
    numSwaps += 1;
}

public boolean lessThanOrEqual(int i1, int i2) {
    numComparisons += 1;
    return i1 <= i2;
}

public void bubbleSort() {

    //...

}

public void selectionSort() {

    //...

}

public void quickSort() {

    //...

}

public void insertionSort() {
    // The prefix [0..s) is a sorted array
    // We insert element s into this prefix
    for (int s = 1; s < array.length; s++) {
        int insert = s;
        // Element at insert is lower than any
        // element in [insert+1..s]
```

```

        for (int i = s - 1; i >= 0; i--) {
            if (!lessThanOrEqualTo(array[i], array[insert])) {
                swap(i, insert);
                insert = i;
            } else {
                break;
            }
        }
    }
}

```

En vuestras implementaciones, de modo similar a como lo hace el método *insertionSort*, llamad a los métodos *lessThanOrEqualTo* y *swap* para comparar si un elemento es menor o igual que otro y para intercambiar dos elementos del vector. De esta manera se contabilizarán las veces que ambos métodos han sido llamados.

Una vez se han implementado, el **segundo paso** será comprobar que los cuatro algoritmos implementados ordenan correctamente por medio del uso de tests con JUnit 5.

Comprobado el correcto funcionamiento, el **tercer y último paso** consiste en analizar los “costes” de cada uno de los algoritmos en base a la cantidad de comparaciones y swaps (o intercambios) realizados a lo largo de diferentes ejecuciones.

Un ejemplo de uso de la clase sería:

```

int[] array = new int [] { 4, 2, 3, 0, 2 };
IntArraySorter sorter = new IntArraySorter(array);
sorter.insertionSort();
int swaps = sorter.getNumSwaps();
int comparisons = sorter.getNumComparisons();

```

Fijaos en que, para cada vector, necesitáis una nueva instancia de *IntArraySorter*.

El análisis se llevará a cabo haciendo uso de un array de 50 posiciones en las siguientes condiciones:

- Ordenado
- Ordenado al revés
- Muy desordenado
- Poco desordenado

La forma más fácil de crear un vector desordenado es, partiendo de un vector perfectamente ordenado, usar el algoritmo de Fisher-Yates que tenéis descrito

en [https://es.wikipedia.org/wiki/Algoritmo\\_de\\_Fisher-Yates](https://es.wikipedia.org/wiki/Algoritmo_de_Fisher-Yates). Fijaos que variando el número de vueltas que usa el bucle podéis variar el grado de desordenación que conseguís respecto del vector ordenado original.

Para generar números aleatorios en Java podéis usar la clase *Random* descrita en <https://docs.oracle.com/javase/7/docs/api/java/util/Random.html>.

## Entrega

El resultado obtenido debe ser entregado por medio de un proyecto IntelliJ por cada grupo. Debe estar comprimido con el nombre “Lab1\_NombreIntegrantes” y contendrá el proyecto con las implementaciones de los cuatro algoritmos de ordenación (*inserción*, *selección*, *burbuja* y *quicksort*), los tests realizados con JUnit 5 para cada uno de ellos y finalmente, el análisis realizado de los algoritmos en base a iteraciones y swaps.

Además, se debe entregar un documento de texto, máximo de dos páginas, en el cual se analicen los resultados obtenidos en base a cómo funciona cada uno de los algoritmos. El objetivo es demostrar que se ha entendido cómo funciona cada uno de ellos. Este documento debe realizarse con un mínimo de formato, es decir, deben utilizarse encabezados, justificar texto, insertar nombre descriptivo a imágenes si existen, referencias, etc.

## Consideraciones de Evaluación

A la hora de evaluar el laboratorio se tendrán en cuenta los siguientes aspectos:

- Los algoritmos planteados ofrecen una solución a los algoritmos de *selección*, *burbuja* y *quicksort*.
- Explicación adecuada del trabajo realizado, es decir, esfuerzo aplicado al documento de texto solicitado.
- Calidad y limpieza del código.