Laboratorio 1 – Análisis de Algoritmos de Ordenación (v1)

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[1] 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").

4. Utilizar otro framework, denominado JMH (Java Microbenchmarking Harness) para realizar mediciones sobre tiempos de ejecución de funciones Java. Este framework es necesario debido a que la máquina virtual de Java es capaz de optimizar una función en tiempo de ejecución, si dicha función es llamada un número suficientemente grande de veces. Por ello, en el caso de Java, no basta con ejecutar una vez la función y medir el tiempo de reloj empleado.

Tareas

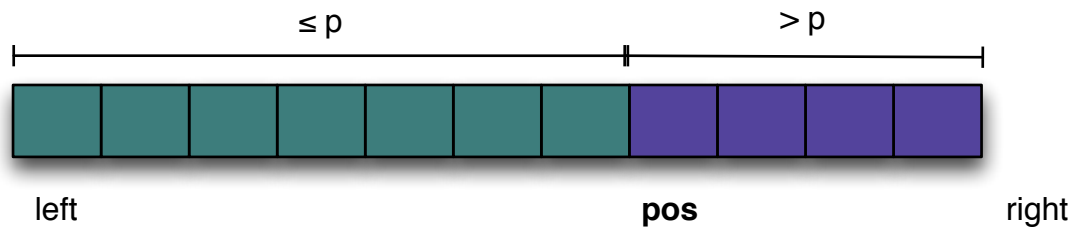
Conocidos los algoritmos de ordenación, la **primera tarea** será la de implementar tres de ellos: *selección*, *burbuja* y *quicksort*. Con respecto al algoritmo de ordenación quicksort (recordad que fue trabajado el año pasado en Programación II), para realizar una implementación "competitiva" deberéis implementar la función auxiliar de partición de forma iterativa, usando un bucle en vez de la recursividad.

Recordad que la función de partición es la siguiente:

[1] Test que comprueba el comportamiento de una unidad de trabajo (generalmente, no siempre, será un método).

```
1 public int partition(int[] v,
2                       int pivot,
3                       int left,
4                       int right) {
5    ¿?
6 }
```

Y su objetivo es, dado el subarray de enteros **v** definido por los índices en el **intervalo [left, right),** y un valor entero **pivot**, devolver la posición pos tal que:



Como es obvio, esta función de partición deberá poder modificar los elementos del subarray (usando la operación auxiliar swap).

Como siempre, recordad que el límite por la izquierda (left) pertenece al intervalo, pero el límite por la derecha (right) es el primero que ya no pertenece al mismo.

En el apartado de recursos del campus virtual, en la subcarpeta de Laboratorio 1, tenéis una presentación del algoritmo de quicksort.

Ordenación por inserción

El código del algoritmo de ordenación por *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;

    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;
    }
```

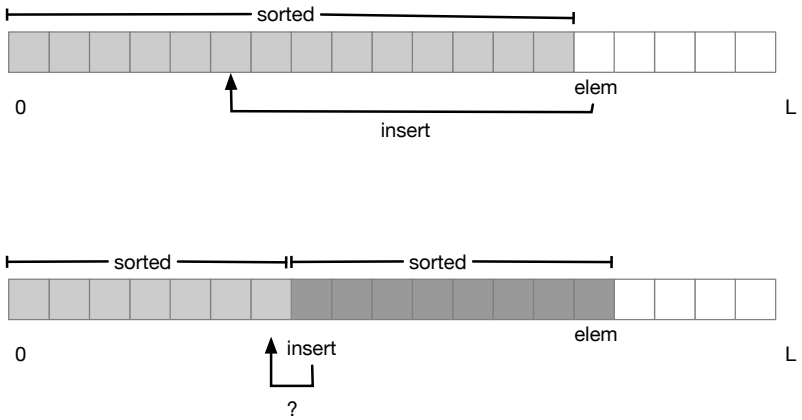
```
    public void swap(int i, int j) {
        int tmp = array[i];
        array[i] = array[j];
        array[j] = tmp;
    }

    public void insertionSort() {

        // Invariant: The prefix [0, end) is a sorted array

        for (int end = 1; end < array.length; end++) {

            // We insert element at end into this prefix

            // Invariant: arrays sorted in the ranges [0, insert)
            // and [insert, end] and all elements in [0, insert)
            // are lower than or equal to those in [insert+1, end]

            for (int insert = end; insert >= 1; insert--) {
                if (array[insert - 1] > array[insert]) {
                    swap(insert - 1, insert);
                } else {
                    break;
                }
            }
        }
    }

    public void bubbleSort() {
        throw new UnsupportedOperationException("TODO: bubbleSort");
    }

    public void selectionSort() {
        throw new UnsupportedOperationException("TODO: selectionSort");
    }

    public void quickSort() {
        throw new UnsupportedOperationException("TODO: quickSort");
    }
}
```

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

Comprobado el correcto funcionamiento, ya que no tendría sentido analizar el rendimiento de un algoritmo de ordenación que no ordena correctamente, el **tercer y último paso** consiste en analizar los "costes" de cada uno de los algoritmos en base a los datos de ejecución que os proporcionará la ejecución de la clase principal (*MyBenchmark*) que utilizará JMH para ejecutar los algoritmos de ordenación para diferentes tamaños del vector.

Sobre los invariantes

En el código de ejemplo hemos incluido, como comentario, los llamados invariantes de los bucles: expresiones lógicas que se cumplen para todas las vueltas del bucle y que sirven para describir formalmente qué hace el bucle (junto a las condiciones de salida de los mismos) y cómo consigue el algoritmo realizar su tarea. En este caso, los invariantes describen la estrategia de ordenación por inserción, que podemos expresar gráficamente como:



Veamos cómo podemos usar el invariante para razonar la corrección del algoritmo. En el bucle externo, el invariante es que el prefijo [0, end) está ordenado:

- En la primera vuelta, lo está, ya que end es 1 y el prefijo [0, 1), que solamente contiene el índice 0 y, por lo tanto, tiene tamaño 1, está trivialmente ordenado.
- Si al principio de una vuelta se cumple el invariante, es decir, el prefijo [0, end) está ordenado, si el bucle interno es correcto y, por tanto, se inserta ordenadamente el elemento en la posición end en dicho prefijo, entonces el prefijo [0, end+1) estará ordenado, por lo que podemos incrementar end.
- Al final del bucle, se cumple que end es L, por lo que se cumplirá que el prefijo [0, L), es decir, todo el array, estará ordenado.

Por tanto, si logramos demostrar que el bucle interno es correcto, podremos garantizar que el externo lo es.

El razonamiento sobre el bucle interno sigue los mismos pasos, aunque ahora su invariante es que:

A. el prefijo [0, insert) está ordenado
B. el sufijo [insert, end] está ordenado
C. los elementos en [0, insert) son menores o iguales que los elementos en [insert+1, end].

Inicialmente, insert es end, por lo que se ha de cumplir que:

A'. [0, end) está ordenado, ya que estamos al principio del bucle interno, por lo que se cumple el invariante del bucle externo, que garantiza exactamente esto.
B'. [end, end] está ordenado ya que solamente contiene la posición end, y un subarray de tamaño 1 siempre está ordenado.
C'. Los elementos del rango [0, end) son menores o iguales que los de [end+1, end], ya que este último es un rango vacío y, por tanto, se cumple trivialmente la propiedad.

Hay dos formas de salir del bucle: por la condición del for, o bien, vía el break interno. Hemos de demostrar que ambas nos garantizan lo que queremos que se cumpla al salir del bucle interno que es el invariante del bucle externo: que el array esté ordenado en las posiciones [0, end].

- Si salimos por la condición del bucle, se da que insert es 0. En este caso, el invariante del bucle, concretamente B, nos garantiza que [insert, end], es decir [0, end] es un subarray ordenado.

- Si salimos por el break, es decir, cuando array[insert-1] <= array[insert], esto nos garantiza que los trozos ordenados (A) [0, insert) y (B) [insert, end] "empalman" en un único trozo ordenado; es decir, que [0, end] es un subarray ordenado.

Solamente nos queda por garantizar que, si el invariante se cumple al principio de una vuelta, se cumple al final de ésta. Si no salimos del bucle, es que se da el caso de que array[insert-1] > array[insert] que, junto con (A) y (B) permiten escribir:

$$[0, insert-1) <= array[insert-1] > array[insert] <= [insert+1, end]$$

Además, (C) nos garantiza que

$$array[insert-1] <= [insert+1, end]$$

Por tanto, después de ejecutar el swap, es decir, de intercambiar los valores de las posiciones insert e insert-1, se cumple:

$$[0, insert-1) <= array[insert-1] < array[insert] <= [insert+1, end]$$

$$array[insert] <= [insert+1, end]$$

Expresiones que nos garantizan:

A". [0, insert-1) ordenado, ya que es un prefijo de [0, insert) que estaba ordenado por (A)
B". [insert, end] = [(insert-1)+1, end] ordenado
C". los elementos de [0, insert-1) son menores o iguales que los de [insert, end]=[(insert-1)+1, end] ya que antes eran menores que los del trozo [insert+1, end] y el elemento que está en la posición insert, que es el que antes estaba en insert-1, por el invariante (A), era mayor o igual que todos ellos.

Es decir, cumplimos el invariante cuando decrementamos insert en una unidad.

Lo único que queda por demostrar es que nunca entramos en un bucle infinito. Para ello hemos de encontrar expresiones que podemos garantizar que son positivas al dar vueltas y que siempre las decrementamos. Con ello podremos demostrar que llegará un momento que no podamos dar más vueltas al bucle sin convertirlas en negativas.

- Para el primer bucle podemos elegir: array-length – end.
- Para el segundo, insert.

Y, con todo ello, finalmente, hemos podido demostrar que el algoritmo es correcto.

El proyecto que os proporcionamos

En esta primera práctica, y debido al uso del framework JMH, os proporcionamos un proyecto que incluye:

- Juegos de prueba de los algoritmos de ordenación.
- Benchmark para medir el rendimiento de los algoritmos de ordenación.
- Implementación de la búsqueda por inserción, para que podáis comprobar la buena configuración de vuestro proyecto.
- Además, en la sección de recursos del campus virtual, en la carpeta Laboratorio 0,5, tenéis un ejemplo de proyecto IntelliJ que usa la librería JMH.

Configuración adicional

Para que funcione en vuestra máquina, deberéis realizar una pequeña configuración del proyecto:

- Incluid JUnit5.* para los test:
  - Abrid una de las clases de prueba en el editor.
  - Poned el cursor sobre uno de los @Test que se marcan en rojo.
  - <Alt>+enter y seleccionad “Add JUnit5.* to classpath”.
- Incluid las bibliotecas de JMH:
  - Id a “Project structure ...”.
  - Seleccionad “Libraries”.
  - Clicad en el (+) en la parte superior del panel intermedio.
  - Seleccionad “From maven ...”.
  - Escribid “org.openjdk.jmh:jmh-core:1.33”.
  - Clicad OK y “Add to module”.
  - Volved a clicar en (+).
  - Seleccionad “From maven ...”.
  - Escribid “org.openjdk.jmh:jmh-generator-annprocess:1.33”.
  - Clicad OK y “Add to module”.
- Configurad que el paso de compilación haga uso del procesador de anotaciones:
  - Id a “Preferences”.
  - Seleccionad “Build, Execution, Deployment”.
  - Seleccionad “Compiler”.
  - Seleccionad “Annotation processor”.
  - Marcad “Enable Annotation Processor”.
  - Clicad OK.

Estas mismas instrucciones las podéis encontrar en el fichero README.md que se incluye en el directorio raíz del proyecto.

Configuración de las pruebas de rendimiento

En las pruebas que queremos realizar, lo que deseamos medir es el tiempo medio de ejecución para cada uno de los algoritmos y para cada uno de los tamaños de array. Por ello la clase principal está anotada de la siguiente manera:

```
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(MILLISECONDS)
public class MyBenchmark { … }
```

Para conseguir que los resultados que se miden sean lo más correctos posibles, la ejecución de las pruebas con JMH realizan primero 5 iteraciones de calentamiento, que no se tienen en cuenta en los resultados finales y luego 5 de medición, que se usarán para calcular los tiempos de ejecución. Tal y como se ha configurado la ejecución en el main:

```
public static void main(String[] args) throws RunnerException {
    Options opt = new OptionsBuilder()
            .include(MyBenchmark.class.getSimpleName())
            .forks(1)
            .warmupTime(TimeValue.seconds(1L))
            .measurementTime(TimeValue.seconds(1L))
            .build();

    new Runner(opt).run();
}
```

las ejecuciones de calentamiento y de medida consisten en llamar a las funciones que se quieren medir durante un máximo de un segundo.

Para configurar los datos que se usarán durante la ordenación, se utiliza la clase BenchmarkState:

```
@State(Scope.Benchmark)
public static class BenchMarkState {

    @Param({"100", "200", "400", "800"})
    private int N;

    private int[] data;

    @Setup(Level.Invocation)
    public void fillArray() {
        data = new int[N];
        for (int i = 0; i < N; i++) data[i] = i;
        // Shuffle via Fisher-Yates algorithm
        Random rg = new Random();
        for (int i = N - 1; i >= 1; i--) {
            int j = rg.nextInt(i + 1);
            int tmp = data[i];
            data[i] = data[j];
            data[j] = tmp;
        }
    }
}
```

En ella vemos que realizaremos pruebas con tamaños de vectores de 100, 200, 400 y 800 elementos y, en cada prueba, el vector que se pasará estará desordenado aleatoriamente.

Las funciones que llaman a la función que se quiere medir (que en nuestro caso contienen la llamada al algoritmo de ordenación) son:

```
@Benchmark
public void testInsertionSort(
        Blackhole blackhole,
        BenchMarkState state) {
    IntArraySorter sorter = new IntArraySorter(state.data);
```

```
        sorter.insertionSort();
        blackhole.consume(state.data);
    }
```

En la que básicamente, pasamos en state el array sobre el que ha de trabajar y llamamos al algoritmo de ordenación. El uso de blackhole es para evitar que el compilador considere que realmente en ningún momento usamos el resultado de la ordenación y, por tanto, que decida que la llamada a ordenación es código muerto que puede eliminar.

Comentarios sobre el código

En los juegos de pruebas, para no tener que repetir los test para cada algoritmo de ordenación, se ha hecho uso de la herencia de las clases de prueba para factorizar en la clase base (*AbstractSortTest*) los casos de prueba comunes.

En la ejecución de los benchmarks, para crear arrays desordenados, se ha hecho uso del algoritmo de Fisher-Yates que, partiendo de un vector perfectamente ordenado, genera desordenaciones equiprobables. Podéis encontrar una descripción de dicho algoritmo en https://es.wikipedia.org/wiki/Algoritmo_de_Fisher-Yates.

Para generar números aleatorios en Java, como ya no usamos la librería de la ACM, haremos uso de la clase estándar 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*), así como el código de pruebas y de benchmarks que os proporcionamos (y cualquier test adicional o benchmark que hayáis añadido).

Además, se debe entregar un documento de texto de tamaño máximo de tres páginas (sin contar la portada), el cual incluya:

- Una explicación, usando vuestras palabras, de todos los algoritmos de ordenación de la práctica. Podéis ayudaros de diagramas (que podéis escanear) o razonar en base a los invariantes.
- Análisis de la complejidad de los algoritmos iterativos.

- Comentarios adicionales sobre el desarrollo de la práctica; elementos que os han costado más; pruebas adicionales que habéis intentado hacer, etc.

Este documento debe realizarse con un mínimo de formato, es decir, deben utilizarse encabezados, justificar texto, insertar nombre descriptivo en imágenes si existen, referencias, etc. Si añadís código en el informe, no lo hagáis como imagen (en IntelliJ podéis seleccionar el texto como RTF, ya sea porque así lo habéis configurado por defecto, o porque lo habéis seleccionado en el menú Edit -> Copy).

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*.
- Calidad y limpieza del código.
- Explicación adecuada del trabajo realizado

Adicionalmente, podéis considerar algoritmos adicionales como

- radix-sort (la versión simple solamente para números positivos),
- combinaciones del quicksort con alguno de los algoritmos simples, de manera que, cuando el tamaño del trozo a ordenar sea suficientemente pequeño, se llama al algoritmo simple, en vez de usar quicksort hasta los tamaños cero o uno. Podéis usar el mismo framework JMH para estudiar, con vectores suficientemente grandes, en qué punto es mejor hacer el cambio de algoritmo.
- timsort (algoritmo de ordenación usado en la biblioteca estándar de Java)

Información adicional sobre JMH

- Code Tools: jmh
  - https://openjdk.java.net/projects/code-tools/jmh/
- Avoiding Benchmarking Pitfalls on the JVM
  - https://www.oracle.com/technical-resources/articles/java/architect-benchmarking.html
- Java Microbenchmarks with JMH, Part 1
  - https://blog.avenuecode.com/java-microbenchmarks-with-jmh-part-1
- Java Microbenchmarks with JMH, Part 2

  - https://blog.avenuecode.com/java-microbenchmarks-with-jmh-part-2
- Java Microbenchmarks with JMH, Part 3
  - https://blog.avenuecode.com/java-microbenchmarks-with-jmh-part-3