

# GUION DE LA PRÁCTICA 1.1

---

## OBJETIVOS:

- **Medición de Tiempos de Ejecución**
- **Trabajo con benchmarks**

## 1. CÓDIGO DE PARTIDA PARA ESTA PRÁCTICA

En esta práctica trabajará con la clase **Vector1.java** que se le proporcionan dentro del paquete **alg77777777.p1** y la clase **MatrizOperaciones.java** de la práctica anterior. Esta forma de trabajar *empaquetando* las clases tiene la ventaja de poder estructurar y utilizar la información mucho mejor, por lo que **será nuestra forma de trabajo** en lo sucesivo.

Si utilizamos la plataforma en línea de comandos (JDK)  
Después de colocarse en el directorio que las contiene hacemos:

```
javac Vector1.java
```

A continuación, podemos comprobar que **Vector1.class** aparece en ese mismo directorio, haciendo:

```
DIR
```

Para ejecutar, se puede hacer desde cualquier directorio tecleando:

```
java alg77777777.p1.Vector1
java alg77777777.p1.Vector1 5
java alg77777777.p1.Vector1 50
java alg77777777.p1.Vector1 500
java alg77777777.p1.Vector1 5000
... .
```

Si utilizamos el entorno Eclipse simplemente creamos un proyecto que llamaremos **prac01\_Tiempos<UOpropio>** y arrastramos el paquete **alg77777777.p1** a la carpeta *src*. Para compilar y ejecutar utilizamos la opción “**Run as...**” como indicamos anteriormente. Para añadir los argumentos en la ejecución hay que configurarlos en “**Run configurations...**”.

## 2. TOMA DE TIEMPOS DE EJECUCIÓN

Se trata de la medición empírica del tiempo de ejecución de un programa, pudiendo así comprobar si coincide o no con el comportamiento teórico obtenido a partir del estudio analítico de su complejidad temporal.

Se utiliza para ello el método **currentTimeMillis()** de la clase **System** del paquete de Java **java.lang** (es el único paquete que está ya cargado, sin necesidad de importarlo explícitamente mediante la sentencia *import*). Ese método **currentTimeMillis()** devuelve un entero de tipo **long**

(64 bits), que es el milisegundo actual que *vive* el ordenador (la cuenta a 0 se puso hace casi 50 años: 1 de enero de 1970) (para más información puede consultar la documentación JAVA). Llamando dos veces (al comienzo y al final del proceso a medir) a dicho método y restando los valores devueltos se obtiene el tiempo que pasó entre ambas llamadas (en milisegundos). Teniendo en cuenta esto ¿Cuántos años más podremos seguir utilizando esta forma de contar?

Cuando en una toma de tiempos, efectuada como se explicó en el párrafo anterior, salgan unos pocos milisegundos (por poner un umbral vamos a considerar un número menor a 50 milisegundos) no la vamos a tener en cuenta, por falta de fiabilidad. La razón es que hay procesos internos del sistema (p.ej. el llamado “recolector de basura”), que se ejecutan con mayor prioridad de ejecución que nuestro programa (de hecho, lo interrumpe, aunque no nos enteremos). Pues bien, esos procesos del sistema duran unos pocos milisegundos, que se suman a lo que tarda nuestro proceso a medir, falseando de forma sensible el tiempo obtenido en el caso de tiempos bajos.

**Conclusión: Despreciaremos tiempos por debajo de 50 y mientras más grande sea el tiempo, más fiable (3578 es más fiable que 123).**

A continuación, creamos una nueva clase **Vector2.java**, que tiene como objetivo medir el tiempo de una operación (algoritmo) determinado, en concreto la suma de los  $n$  elementos de un vector, que tenemos implementada en la clase **Vector1**. Pasaremos un argumento en línea de comandos con el tamaño del vector como en el caso anterior.

Recordamos que, para ejecutarlo desde línea de comandos, desde el directorio pertinente:

```
javac Vector2.java
java alg77777777.p1.Vector2 n
```

1. ¿Qué significa que el tiempo medido sea 0?
2. ¿A partir de qué tamaño de problema ( $n$ ) empezamos a obtener tiempos fiables?

### 3. CRECIMIENTO DEL TAMAÑO DEL PROBLEMA

Realmente, es poco práctico ir variando el tamaño del problema a mano, sobre todo si tenemos en cuenta que normalmente queremos dibujar una gráfica del tiempo en función del tamaño del problema para una determinada operación ¿cómo podemos ir obteniendo los distintos valores para dibujar la gráfica?

Crear una clase **Vector3.java** que vaya incrementando el tamaño del vector e ir obteniendo tiempos, y de esta manera seguir de forma más conveniente la evolución del tiempo de ejecución.

```
javac Vector3.java
java alg77777777.p1.Vector3
```

Vamos a realizar incrementos del tamaño de forma exponencial y comparar los tiempos para diferentes tamaños del problema

1. ¿Qué pasa con el tiempo si el tamaño del problema se multiplica por 5?
2. ¿los tiempos obtenidos son los que se esperaban de la complejidad lineal  $O(n)$ ?

Vamos a utilizar una hoja de cálculo para dibujar la gráfica. Para hacer que el proceso sea directo podemos mostrar los datos de tamaño y tiempos en forma tabular (separando las columnas por tabuladores)

#### 4. TOMA DE TIEMPOS DE EJECUCIÓN PEQUEÑOS (<50 ms)

Cuando se verifica que un proceso a medir tarda poco tiempo, podemos ejecutar ese proceso repetidamente **n veces**, ajustando ese parámetro; ya que, por una parte, hay que lograr que el tiempo total de ejecución sea superior a esos 50 milisegundos y, por otra parte, que acabe en un tiempo razonable.

Aunque **nVeces** puede ser un valor cualquiera, es aconsejable probar con valores que sean potencias de 10, porque la conversión de tiempos es tan fácil como aplicar la siguiente tabla:

<b>N Veces</b>	<b>Unidades de tiempo en:</b>	
<b>1</b>	<b>Milisegundos</b>	<b>(10<sup>-3</sup> s)</b>
10	decimas de ms	(10 <sup>-4</sup> s)
100	centésimas de ms	(10 <sup>-5</sup> s)
<b>1 000</b>	<b>microsegundos (µs)</b>	<b>(10<sup>-6</sup> s)</b>
10 000	decimas de µs	(10 <sup>-7</sup> s)
100 000	centésimas de µs	(10 <sup>-8</sup> s)
<b>1 000 000</b>	<b>Nanosegundos</b>	<b>(10<sup>-9</sup> s)</b>
.....	.....	.....
<b>10<sup>9</sup></b>	<b>Picosegundos</b>	<b>(10<sup>-12</sup> s)</b>

Tabla 1. Tabla de conversión de unidades



Crear la clase **Vector4.java** introduciendo la idea anterior. Cuando esté lista la ejecutamos de la siguiente forma:

```
javac Vector4.java
dir
java alg77777777.p1.Vector4 1
java alg77777777.p1.Vector4 10
java alg77777777.p1.Vector4 100
java alg77777777.p1.Vector4 1000
java alg77777777.p1.Vector4 100000
java alg77777777.p1.Vector4 10000000
...
```

Los tiempos para lograr mediciones adecuadas se van alargando. Si por cualquier razón hubiera que abortar una ejecución, pulse Control+C. En Eclipse se puede utilizar el botón cuadrado rojo encima del panel *Console*.

## 5. TRABAJO PEDIDO (TABLA 1)

Debe aplicar lo antes visto a los dos métodos siguientes: *suma* y *maximo*. Con los tiempos obtenidos debe ir rellenando la siguiente tabla:

$n$	$t \text{ suma}$	$t \text{ maximo}$
10	..... 	..... 
30	.....	.....
90	.....	.....
270	.....	.....
810	.....	.....
2430	.....	.....
7290	.....	.....
21870	.....	.....
65610	.....	.....
196830	.....	.....
590490	.....	.....
1771470	.....	.....
Hasta que <i>pete</i>	.....	.....

**Indicar las características principales (procesador, memoria) del ordenador donde se han medido tiempos.**

**Una vez rellenada la tabla, responder de forma razonada a la siguiente pregunta: ¿cumplen los valores obtenidos con lo esperado?**


## 6. OPERACIONES SOBRE MATRICES

En la clase `MatrizOperaciones.java` creada en la práctica anterior se programan varias operaciones sobre una matriz cuadrada de orden  $n$ .

Crear una nueva clase **`MatrizOperacionesTiempos.java`** para medir tiempos de las operaciones `sumarDiagonal1()` y `sumarDiagonal2()`, utilizando todo lo que hemos aprendido hasta ahora.

## 7. TRABAJO PEDIDO (TABLA 2)

Debe medir los tiempos de los dos métodos vistos aquí:

$n$	$t \text{ diagonal1}$	$t \text{ diagonal2}$
3	..... 	.....
6	.....	.....
12	.....	.....
24	.....	.....
48	.....	.....
96	.....	.....

192	.....	.....
384	.....	.....
768	.....	.....
	.....	.....
	.....	.....

**Indicar las características principales (procesador, memoria) del ordenador donde se han medido tiempos.**

**Una vez rellenada la tabla, responder de forma razonada a la siguiente pregunta: ¿cumplen los valores obtenidos con lo esperado?**

## 8. BENCHMARKING: INFLUENCIA DE LA PLATAFORMA

### 8.1 Tarea 1

Continuando con el trabajo de la semana pasada, en esta tarea utilizaremos el programa `Benchmarking1`. Se proporcionan **4 versiones** del mismo programa: Java, Python, C# y C++.

- Revisa el código fuente de las 4 versiones del programa. Abre con un editor de textos el fichero `run1.cmd` para comprender su funcionamiento.
- Utilizando el fichero `run1.cmd`, ejecutar los 4 programas.
- Ejecuta los mismos programas con la utilidad `ptime.exe` (disponible en <http://www.pc-tools.net/win32/ptime/>). Para ello puedes modificar `run1.cmd`.

Contesta las siguientes cuestiones:

1. ¿Sabrías explicar a qué se debe la diferencia de tiempos entre la ejecución normal y el resultado obtenido con el programa `ptime.exe`?
2. ¿Cuáles de estos programas corren sobre una máquina virtual? ¿Cuáles corren código nativo? ¿Cuáles corren sobre un intérprete?

En el fichero `run1.cmd` tienes algunas pistas para ayudarte a contestar a estas preguntas.

### 8.2 Tarea 2

Una de las características de las plataformas de programación modernas es la optimización “inteligente” de código que realizan sobre lo escrito por el programador, esto se realiza tanto a nivel de compilación como de ejecución. Por tanto, a la hora de medir tiempos y sobre todo cuando estamos “simulando” las funciones a medir como nos pasa en estas prácticas iniciales hay que ser consciente de esto para que los tiempos sean coherentes.

Por ejemplo, en tiempo de compilación, si el compilador de Java detecta que hay un bucle que no produce ningún cambio de valor de una variable externa directamente lo elimina.

En tiempo de ejecución, una de las características de la JVM, es que dispone de un compilador *Just In Time* con optimización de *HotSpot*. En la Wikipedia puedes encontrar una pequeña explicación de cómo funciona: <https://en.wikipedia.org/wiki/HotSpot#Features>

- Lee este artículo: <https://www.beyondjava.net/blog/a-close-look-at-javas-jit-dont-waste-your-time-on-local-optimizations/>
- Compila y ejecuta `PerformanceTest.java`.
- Ejecuta el fichero `PerformanceTest.cpp.exe`. Este programa no es más que una adaptación del programa anterior en lenguaje C++. Puedes utilizar el script `run2.cmd`.
- Compara el código fuente de `Benchmarking2` y `Benchmarking2Warmup`.
- Compila y ejecuta el programa `Benchmarking2`. ¿Crees que está afectando el optimizador de la JVM?
- Compila y ejecuta el programa `Benchmarking2Warmup`. Utiliza el script `run3.cmd` para ejecutar ambos programas. ¿Crees que está afectando el optimizador de la JVM?

### Conclusiones

Contesta las siguientes cuestiones:

1. ¿Podrías explicar con tus propias palabras que son las optimizaciones de HotSpot?
2. ¿Has notado alguna diferencia entre la versión en Java y la Versión C++? Explícala brevemente.
3. ¿Se te ocurre alguna forma de eliminar este inconveniente?

## 9. TRABAJO PEDIDO BENCHMARKING

**Haz un documento Word donde contestar a las diferentes preguntas de esta sección, tanto de las tareas como de las conclusiones.**

*La entrega de esta práctica se realizará junto a la siguiente, en la tarea creada a tal efecto en el campus virtual. Se entregará un documento con las tablas y las respuestas a las preguntas. Además, en un fichero comprimido aparte, se entregará el código pedido, dentro del paquete `alg<dnipropio>.p11`. Si utiliza Eclipse llamar al proyecto `prac01_Tiempos<UOpropio>`.*