Análisis empírico de algoritmos con Java Microbenchmark Harness (JMH)

Lógica y Algorítmica

Curso 2025-2026

Índice

1. Introducción

Este documento proporciona una guía paso a paso para la configuración y ejecución de pruebas de rendimiento mediante **JMH** (*Java Microbenchmark Harness*). Su propósito es facilitar el análisis empírico de algoritmos, permitiendo obtener mediciones precisas que complementen el estudio teórico de su complejidad.

A diferencia de las mediciones manuales con `System.nanoTime()`, JMH permite controlar el impacto de los complejos mecanismos de optimización de la JVM (*Java Virtual Machine*), como el calentamiento del compilador JIT (*Just-In-Time*) o la recolección de basura. Esto permite alcanzar el denominado estado estacionario, una fase donde el rendimiento se estabiliza tras las optimizaciones iniciales de la JVM.

Al finalizar esta guía, el alumnado habrá aprendido a configurar un entorno profesional, implementar benchmarks robustos e interpretar sus métricas desde una perspectiva científica.

2. Limitaciones de las mediciones temporales directas

Antes de profundizar en el uso de JMH, es necesario analizar por qué los métodos convencionales de medición en Java, como `System.nanoTime()`, no proporcionan resultados fiables para el análisis de rendimiento algorítmico.

2.1 Inconsistencia en los resultados directos

Al intentar comparar la eficiencia de distintas implementaciones mediante un enfoque basado en cronometraje directo, se suelen obtener resultados altamente variables. Considérese el siguiente ejemplo:

```
public class MedicionDirecta {

    public static long fibIterativo(int n) {
        long num1 = 0, num2 = 1;
        for (int i = 0; i < n; i++) {
            long suma = num1 + num2;
            num1 = num2;
            num2 = suma;
        }
        return num1;
    }

    public static void main(String[] args) {
        int n = 40;
        int repeticiones = 100000;

        // Primera serie de mediciones (JVM en frío)
        long inicio = System.nanoTime();
        for (int i = 0; i < repeticiones; i++) fibIterativo(n);
        long duracion1 = System.nanoTime() - inicio;

        // Segunda serie de mediciones (JVM ya iniciada)
        inicio = System.nanoTime();
        for (int i = 0; i < repeticiones; i++) fibIterativo(n);
        long duracion2 = System.nanoTime() - inicio;

        // Tercera serie de mediciones
        inicio = System.nanoTime();
        for (int i = 0; i < repeticiones; i++) fibIterativo(n);
        long duracion3 = System.nanoTime() - inicio;

        System.out.printf("Ejecución 1: %,d ns/op%n", duracion1 / repeticiones);
        System.out.printf("Ejecución 2: %,d ns/op%n", duracion2 / repeticiones);
        System.out.printf("Ejecución 3: %,d ns/op%n", duracion3 / repeticiones);
    }
```

```
}
```

Una ejecución típica de este código podría arrojar los siguientes valores medios por operación:

```
Ejecución 1: 33 ns/op
Ejecución 2: 12 ns/op
Ejecución 3:  9 ns/op
```

Estos datos evidencian una falta de consistencia ya que los resultados difieren significativamente entre ejecuciones idénticas del mismo fragmento de código. Esta variabilidad impide extraer conclusiones sólidas sobre la eficiencia real del algoritmo si la medición se basa únicamente en muestras aisladas.

1.2 Factores de variabilidad en la JVM

La falta de precisión en estas mediciones se debe a diversos mecanismos internos de la Máquina Virtual de Java (JVM) y del entorno de ejecución que alteran el tiempo observado.

Veamos algunos de los factores que provocan esta variabilidad.

1. Compilación JIT (*Just-In-Time*)

La JVM emplea un compilador dinámico que transforma el bytecode en código máquina nativo durante la propia ejecución. Cuando un método se ejecuta repetidamente, el compilador JIT aplica optimizaciones avanzadas. En el ejemplo anterior, la primera ejecución es lenta porque el código se interpreta, mientras que las sucesivas son órdenes de magnitud más rápidas al estar ya optimizadas.

2. Eliminación de código muerto (*Dead Code Elimination*)

Si el compilador detecta que el resultado de una función nunca llega a utilizarse para producir una salida o afectar al estado del programa, puede optar por eliminar dicha instrucción por completo.

```
// Medición incorrecta: el compilador podría eliminar esta llamada
for (int i = 0; i < repeticiones; i++) {
    fibIterativo(n); // El valor de retorno no se utiliza ni se almacena
}
```

En este escenario, el tiempo medido correspondería a un bucle vacío, lo que invalidaría por completo el experimento de rendimiento.

3. Caching y estado de la jerarquía de memoria

El rendimiento está fuertemente condicionado por la presencia de los datos en las memorias caché del procesador (L1, L2 y L3). La primera ejecución suele incurrir en fallos de caché que incrementan el tiempo medio, un factor que se mitiga en ejecuciones posteriores una vez que la información necesaria ha sido cargada.

4. Ruido del entorno y recolección de basura

El planificador del sistema operativo y tareas asíncronas de la propia JVM, como la recolección de basura (*Garbage Collection*), pueden interrumpir la ejecución del código en cualquier instante. Una medición simple de tiempo puede incluir estos eventos externos sin que el analista tenga forma de detectarlos o aislarlos.

1.3 Ventajas del uso de un framework de microbenchmarking

Para obtener resultados estadísticamente válidos, es necesario hacer uso de un framework específico para benchmarks que gestione estos factores de forma automática e imparcial. En este tutorial veremos cómo JMH soluciona estas deficiencias mediante los siguientes mecanismos:

1. **Aislamiento de la ejecución (*forks*)** para garantizar que cada experimento se lanza en un proceso de sistema diferente, evitando la contaminación del estado de la JVM.
2. **Fases de calentamiento (*warm-up*)** obligatorias para asegurar que el código ha sido debidamente optimizado por el compilador JIT antes de tomar las medidas definitivas.
3. **Muestreo estadístico múltiple** para aplicar estadística descriptiva y obtener una media acompañada de su margen de error.
4. **Consumo controlado de resultados** mediante objetos específicos (*BlackHoles*) que impiden al compilador eliminar el código bajo prueba.

La calidad de un análisis empírico no reside en el tiempo total invertido en la medición, sino en el rigor estadístico y en el control de las variables de entorno que proporciona una herramienta especializada como JMH.

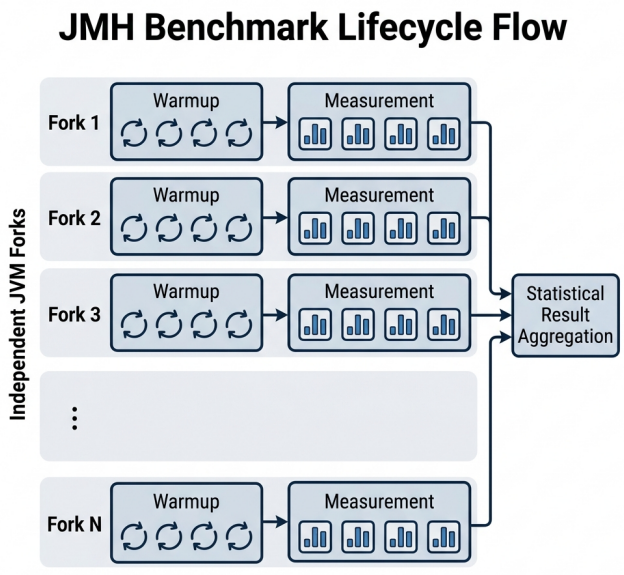

Figura 1. Ciclo de vida estructurado de un benchmark en JMH.

3. Configuración del entorno y ejecución del primer microbenchmark

En esta sección se detallan los procedimientos necesarios para configurar un entorno de experimentación robusto. A través de la creación y configuración de un proyecto Maven en el IDE Eclipse, aprenderemos a integrar JMH y a ejecutar nuestro primer microbenchmark de forma sistemática.

3.1 Creación del proyecto Maven en Eclipse

1. Abre Eclipse y haz clic en **File** > **New** > **Other...**.
2. En la ventana que se abre, busca la carpeta **Maven**, selecciona **Maven Project** y haz clic en **Next**.

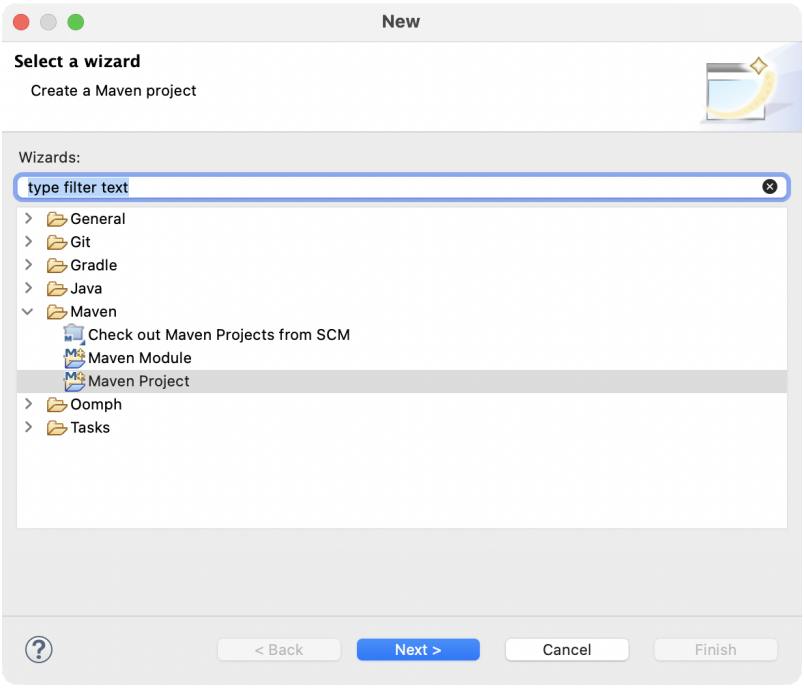

Figura 2. Creación de un proyecto Maven.

3. Marca la casilla **Create a simple project (skip archetype selection)**. Esto nos ahorra problemas con arquetipos desactualizados. Haz clic en **Next**.

¿Qué es un arquetipo de Maven?

Un arquetipo es una plantilla preconfigurada que genera automáticamente la estructura de directorios y archivos iniciales de un proyecto. Aunque suelen facilitar el inicio de aplicaciones complejas, en este tutorial optamos por un proyecto simple para tener control total sobre la configuración y evitar conflictos con plantillas que puedan estar desactualizadas.

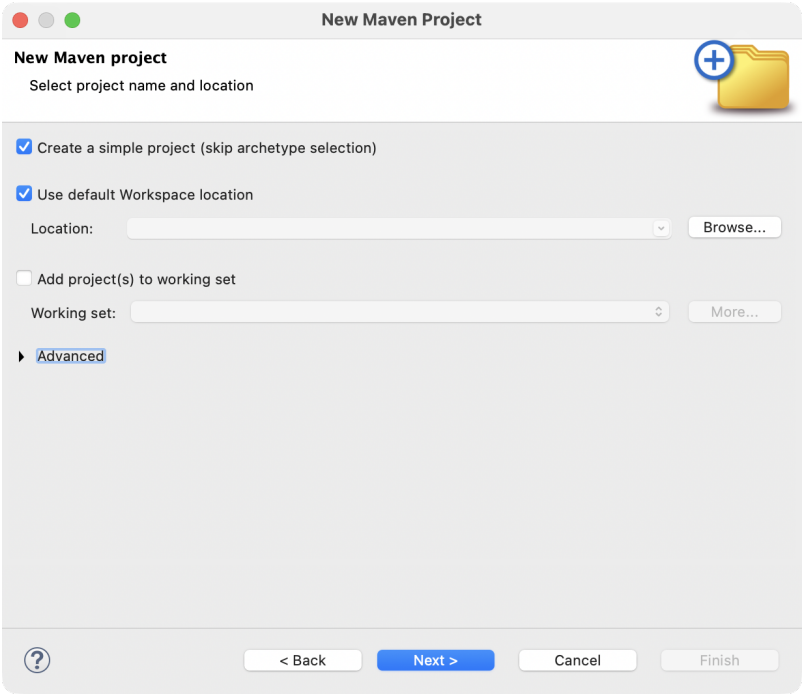

Figura 3. Marcamos la casilla de Create a simple project (skip archetype selection).

4. En el campo **Group Id**, escribe `org.algoritmica`, o el paquete que prefieras.
5. En el campo **Artifact Id**, escribe `jmh-benchmark`, o el nombre que prefieras para tu proyecto.
6. Haz clic en **Finish**.

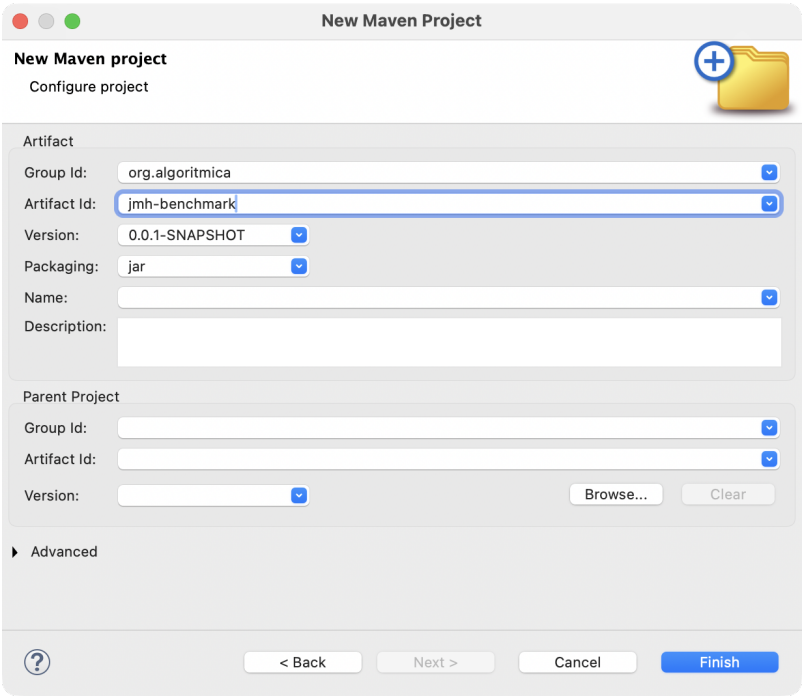

Figura 4. Configuración del Group Id y Artifact Id.

3.2 Configuración del archivo `pom.xml`

Aquí vamos a añadir las bibliotecas de JMH y, lo más importante, el *plugin* del compilador de Maven configurado para procesar las anotaciones automáticamente. Esto obliga a Eclipse a generar el archivo `BenchmarkList`.

1. En el panel izquierdo (Package Explorer), despliega tu proyecto `jmh-benchmark`.
2. Haz doble clic en el archivo **`pom.xml`**.
3. En la parte inferior del editor, cambia a la pestaña **pom.xml** para ver el código fuente.
4. Borra el contenido generado y pega esta configuración completa:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.algoritmica</groupId>
  <artifactId>jmh-benchmark</artifactId>
  <version>0.0.1-SNAPSHOT</version>

  <properties>
    <maven.compiler.source>21</maven.compiler.source>
    <maven.compiler.target>21</maven.compiler.target>
    <jmh.version>1.37</jmh.version>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.openjdk.jmh</groupId>
      <artifactId>jmh-core</artifactId>
      <version>${jmh.version}</version>
    </dependency>
    <dependency>
      <groupId>org.openjdk.jmh</groupId>
      <artifactId>jmh-generator-annprocess</artifactId>
      <version>${jmh.version}</version>
      <scope>provided</scope>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.11.0</version>
        <configuration>
          <source>21</source>
          <target>21</target>
          <annotationProcessorPaths>
            <path>
              <groupId>org.openjdk.jmh</groupId>
              <artifactId>jmh-generator-annprocess</artifactId>
              <version>${jmh.version}</version>
            </path>
```

```
          </annotationProcessorPaths>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

5. Guarda el archivo (`Ctrl + S`). Eclipse descargará automáticamente las dependencias.

3.3 Implementación del primer *benchmark*

1. Haz clic derecho sobre la carpeta `src/main/java` en el Package Explorer.
2. Selecciona **New** > **Class**.
3. En **Package**, escribe `org.algoritmica.benchmark`, o el paquete que prefieras.
4. En **Name**, escribe `MiPrimerBenchmark`, o el nombre que prefieras para tu clase.
5. Haz clic en **Finish**.

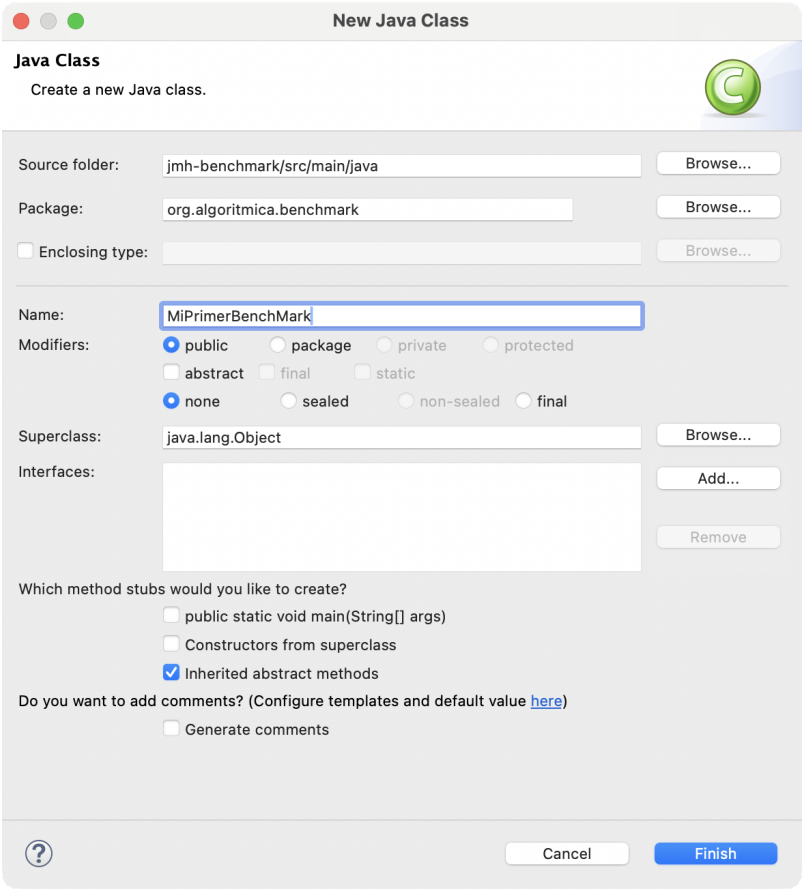

Figura 5. Creación de la clase MiPrimerBenchmark.

6. Reemplaza el código de la clase con este ejemplo, que incluye un método `main` para ejecutar la prueba cómodamente desde el IDE.

```
package org.algoritmica.benchmark;

import org.openjdk.jmh.annotations.*;
```

```java
import org.openjdk.jmh.runner.Runner;
import org.openjdk.jmh.runner.RunnerException;
import org.openjdk.jmh.runner.options.Options;
import org.openjdk.jmh.runner.options.OptionsBuilder;

import java.util.concurrent.TimeUnit;

@BenchmarkMode(Mode.AverageTime) ①
@OutputTimeUnit(TimeUnit.NANOSECONDS) ②
@State(Scope.Thread) ③
@Warmup(iterations = 2) ④
@Measurement(iterations = 3) ⑤
@Fork(1) ⑥
public class MiPrimerBenchmark {

    @Benchmark ⑦
    public void testMetodoVacio() {
        // Este método no hace nada, medirá el coste puro de JMH
    }

    public static void main(String[] args) throws RunnerException {
        Options opt = new OptionsBuilder() ⑧
                .include(MiPrimerBenchmark.class.getSimpleName())
                .build();

        new Runner(opt).run(); ⑨
    }
}
```

① **@BenchmarkMode**: Define qué queremos medir. `AverageTime` calcula el tiempo promedio que tarda cada ejecución del método.

② **@OutputTimeUnit**: Indica la unidad de tiempo de los resultados. Usamos nanosegundos (`NANOSECONDS`) para medir operaciones muy rápidas.

③ **@State**: Configura el estado del objeto. `Scope.Thread` significa que cada hilo que ejecute la prueba tendrá su propia instancia de esta clase.

④ **@Warmup**: Define las iteraciones de “calentamiento”. Sirven para que la JVM optimice el código antes de empezar a medir de verdad.

⑤ **@Measurement**: Indica cuántas iteraciones de medición real se realizarán para calcular la media final.

⑥ **@Fork**: Especifica cuántas veces se reiniciará la máquina virtual para repetir el experimento desde cero, evitando sesgos.

⑦ **@Benchmark**: Esta es la anotación más importante. Indica a JMH que este es el método que debe ser cronometrado y analizado.

⑧ **OptionsBuilder**: Es una utilidad que nos permite configurar cómo queremos lanzar el experimento, qué clases incluir, filtros, etc.

⑨ **Runner**: Es el motor de JMH que se encarga de lanzar los procesos, ejecutar el código y generar

el informe final.

3.4 Compilación del proyecto con Maven

Antes de ejecutar la clase Java, debemos pedirle a Maven que compile el proyecto. Este paso es fundamental para que el *plugin* de JMH que configuramos en el `pom.xml` procese las anotaciones (como `@Benchmark`) y genere los archivos internos necesarios para que el experimento funcione.

1. Haz clic derecho sobre el nombre de tu proyecto (`jmh-benchmark`) en el Package Explorer.
2. Selecciona **Run As** > **Maven build...** (asegúrate de elegir la opción que tiene los tres puntos ⋯).
3. En la ventana aparece el campo **Goals**.
4. Escribe exactamente: `clean compile`.
5. Haz clic en **Run**.
6. Revisa la consola de Eclipse en la parte inferior. Espera a ver el mensaje `BUILD SUCCESS`.

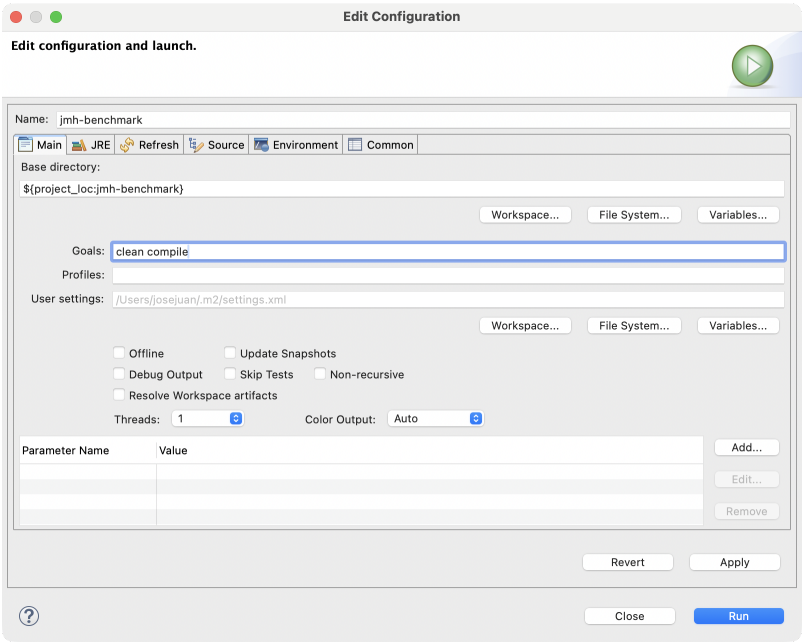

Figura 6. Compilación del proyecto con Maven.

Importante

Deberás repetir este paso de compilación **cada vez que añadas un nuevo método `@Benchmark` o crees una nueva clase de pruebas**, para que JMH registre los cambios.

3.5 Ejecución del experimento desde Eclipse

1. Abre tu clase `MiPrimerBenchmark`.
2. Haz clic derecho en cualquier parte del código fuente.
3. Selecciona **Run As** > **Java Application**.

La consola de Eclipse comenzará a mostrar los registros de iteración de JMH, finalizando con la

tabla resumen de resultados.

3.5.1 Análisis de resultados

Una vez finalizado el experimento, verás una tabla similar a esta.

```
Benchmark                          Mode  Cnt  Score   Error  Units
MiPrimerBenchmark.testMetodoVacio  avgt    3  0.557 ± 0.197  ns/op
```

¿Qué significan estos datos?

Es fundamental saber interpretar estas columnas para validar nuestro experimento.

- **Benchmark**: es el nombre completo del método que se ha puesto a prueba.
- **Mode**: indica el modo de medición. En este ejemplo, `avgt` significa *average time*, que es el tiempo promedio por operación.
- **Cnt**: es el número de iteraciones de medición (*samples*) que se han tenido en cuenta para el cálculo final.
- **Score**: es el valor principal del resultado. En este caso, el método tarda aproximadamente 0,557 nanosegundos en ejecutarse.
- **Error**: representa el margen de error estadístico. Cuanto más bajo sea este valor en relación con el *score*, más fiable será la prueba.
- **Units**: indica la unidad de medida, que en este ejemplo es nanosegundos por operación: `ns/op`.

4. Análisis de la sucesión de Fibonacci

En esta sección compararemos tres implementaciones del cálculo de Fibonacci para observar cómo JMH nos ayuda a validar las complejidades algorítmicas teóricas.

4.1 Definición matemática de Fibonacci

La sucesión de Fibonacci se define matemáticamente mediante la siguiente relación de recurrencia:

$$F_n = \begin{cases} n & \text{si} n \leq 1 \\ F_{n-1} + F_{n-2} & \text{si} n > 1 \end{cases}$$

Esta definición recursiva es la base de los algoritmos que compararemos a continuación.

4.2 Implementación de los algoritmos

Para este experimento, implementaremos tres versiones del cálculo de Fibonacci que representan diferentes paradigmas: (1) una versión **iterativa** clásica, (2) una versión optimizada mediante **recursividad de cola** (*tail recursion*) y (3) la versión **recursiva tradicional** (exponencial). Este enfoque nos permitirá visualizar cómo la elección del algoritmo afecta radicalmente al rendimiento real.

1. Crea una nueva clase en el mismo paquete llamada `FibonacciBenchmark`.
2. Pega el siguiente código, que utiliza la anotación `@Param` para definir los datos de entrada:

```
package org.algoritmica.benchmark;

import org.openjdk.jmh.annotations.*;
import org.openjdk.jmh.runner.Runner;
import org.openjdk.jmh.runner.options.Options;
import org.openjdk.jmh.runner.options.OptionsBuilder;

import java.util.concurrent.TimeUnit;

@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.NANOSECONDS)
@Warmup(iterations = 5, time = 1)
@Measurement(iterations = 5, time = 1)
@Fork(1)
@State(Scope.Benchmark)
public class FibonacciBenchmark {

    @Param({ "45" }) // Valor del n-ésimo número de Fibonacci a calcular
    private int n;

    @Benchmark
    public long testRecursivo() {
```

```
        return fibRecursivo(n);
    }

    @Benchmark
    public long testTailRecursion() {
        return fibTail(n, 0, 1);
    }

    @Benchmark
    public long testIterativo() {
        return fibIterativo(n);
    }

    // Método recursivo tradicional (sin recursividad de cola)
    public static long fibRecursivo(int n) {
        if (n <= 1) return n;
        return fibRecursivo(n - 1) + fibRecursivo(n - 2);
    }

    // Método con recursividad de cola (tail recursion)
    public static long fibTail(int n, long a, long b) {
        if (n == 0) return a;
        if (n == 1) return b;
        return fibTail(n - 1, b, a + b);
    }

    // Método iterativo tradicional
    public static long fibIterativo(int n) {
        long num1 = 0, num2 = 1;
        for (int i = 0; i < n; i++) {
            long suma = num1 + num2;
            num1 = num2;
            num2 = suma;
        }
        return num1;
    }

    public static void main(String[] args) throws Exception {
        Options opt = new OptionsBuilder()
                .include(FibonacciBenchmark.class.getSimpleName())
                .build();
        new Runner(opt).run();
    }
}
```

4.3 Explicación de la configuración avanzada

En este segundo experimento hemos utilizado una configuración más profesional para garantizar la estabilidad de los resultados en algoritmos complejos:

- `@Warmup(iterations = 5, time = 1)`: Realizamos 5 rondas de calentamiento. Según investigaciones recientes (Traini et al., 2022), para asegurar que la JVM ha salido de la fase de interpretación, se recomienda un mínimo de **50 a 300 iteraciones** de calentamiento en experimentos críticos. El parámetro `time = 1` indica que cada ronda durará 1 segundo. Esto da tiempo suficiente al compilador JIT (*Just-In-Time*) para detectar los "puntos calientes" del código y optimizarlos antes de empezar a medir.
- `@Measurement(iterations = 5, time = 1)`: Configuramos 5 rondas de medición real de 1 segundo cada una. JMH calculará la media de estas 5 muestras, lo que reduce el impacto de posibles picos puntuales de consumo de CPU en el sistema.
- `@Fork(1)`: Indica que el experimento se ejecutará en un proceso de sistema independiente. Usamos `1` para agilizar el tutorial, aunque en entornos de producción académica se recomiendan `2` o `3` para verificar la consistencia entre diferentes lanzamientos de la JVM.
- `@State(Scope.Benchmark)`: Define que todos los hilos que puedan participar en la prueba compartirán la misma instancia de la clase `FibonacciBenchmark`. Es el alcance estándar para pruebas donde el estado (en este caso el valor de `n`) es compartido.
- `@Param({ "45" })`: Esta es una de las anotaciones más potentes. Permite inyectar valores en el campo `n` para probar cómo escala el algoritmo. Si pusiéramos `@Param({ "10", "20", "30" })`, JMH ejecutaría automáticamente el experimento tres veces, una para cada valor, permitiéndonos ver la evolución del tiempo de ejecución según crece el problema.

Atención

El algoritmo `fibRecursivo` tiene una complejidad exponencial $O(2^n)$. Para valores como $n = 45$, el tiempo de ejecución será extremadamente superior al de los otros métodos (incluso podría tardar varios minutos).

Si quieres una respuesta rápida, te recomendamos probar primero con un valor de n más pequeño (por ejemplo, `30`) en la anotación `@Param`.

4.4 Análisis de los resultados experimentales

Al ejecutar el benchmark de Fibonacci con $n = 45$, obtenemos unos resultados que ilustran perfectamente la importancia de elegir el algoritmo correcto según su complejidad.

```
Benchmark                          (n)  Mode  Cnt            Score           Error
Units
FibBenchmark.testIterativo          45  avgt    5           13,872 ±         1,809  ns/op
FibBenchmark.testTailRecursion      45  avgt    5           70,564 ±         3,045  ns/op
FibBenchmark.testRecursivo          45  avgt    5  15121623594,600 ± 423893396,951  ns/op
```

Conclusiones de la comparativa

Tras el análisis de los datos obtenidos para $n = 45$, podemos extraer tres conclusiones fundamentales que validan los principios de la algoritmia:

1. **El impacto de la complejidad asintótica**. La diferencia de rendimiento entre las versiones eficientes y la recursiva tradicional no es solo cuantitativa, sino cualitativa. Mientras que la versión iterativa procesa el resultado en aproximadamente **14 nanosegundos**, la versión recursiva clásica requiere más de **15 segundos (15 x 10^9 ns)**. Esto supone una diferencia de **nueve órdenes de magnitud** (mil millones de veces más lento), ilustrando perfectamente cómo un algoritmo $O(2^n)$ se vuelve impracticable frente a uno $O(n)$ incluso para entradas pequeñas.

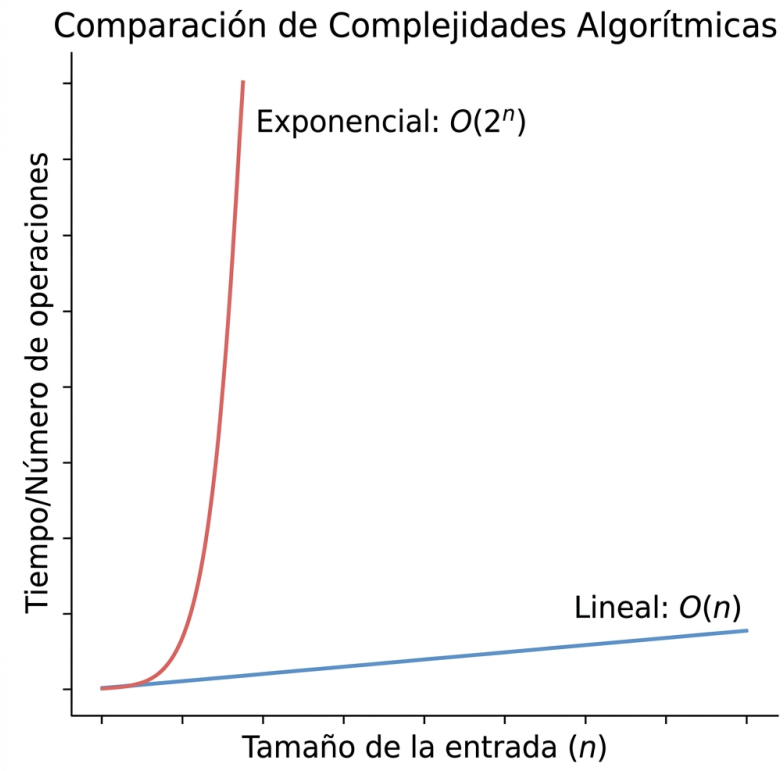

Figura 7. Comparativa de crecimiento lineal O(n) frente a exponencial $O(2^n)$.

2. **Eficiencia de la recursividad de cola (*Tail Recursion*)**. Los resultados demuestran que la recursividad de cola es una alternativa extremadamente eficiente a la iteración. Con un tiempo de aproximadamente **70 ns**, se mantiene en el mismo orden de magnitud que la versión iterativa. La pequeña penalización observada se debe a la gestión de la pila de llamadas, pero su rendimiento es órdenes de magnitud superior a la recursividad clásica, confirmando que es una técnica viable para algoritmos de alto rendimiento.
3. **Rigor estadístico y fiabilidad**. Observando la columna **Error**, vemos valores muy bajos en relación con el *Score*, especialmente en las versiones lineales. Esto confirma que las mediciones de JMH son estables y que el entorno de ejecución ha sido correctamente aislado mediante *forks* y estabilizado con la fase de *warm-up*, eliminando el ruido que habitualmente invalida las mediciones manuales.

El reto del "Estado Estacionario" (Steady State)

Estudios académicos (Traini et al., 2022) indican que es posible que algunos benchmarks en Java nunca alcancen un estado estable debido al no-determinismo de la JVM. Si observas que el margen de error estadístico es muy elevado, es posible que el código aún esté siendo optimizado por el JIT o que la recolección de basura esté interfiriendo. En estos casos, aumentar el número de `Forks` y de

iteraciones de `Warmup` es esencial para obtener datos fiables.

Validación Teórica

Este experimento permite al alumnado comprobar que la teoría de la complejidad no es una abstracción matemática, sino una herramienta predictiva real. La elección del algoritmo correcto es, en muchos casos, más importante que la potencia del hardware utilizado.

5. Referencias

- Microbenchmark with Java. Baeldung.
- Repositorio oficial de JMH en GitHub.
- Towards effective assessment of steady state performance in Java software: Are we there yet?. Luca Traini, Vittorio Cortellessa, Daniele Di Pompeo, Michele Tucci. 2022.

6. Licencia

Licencia **CC BY-NC-ND 4.0**