# micro benchmarking
# Medidas de Uso de Recursos
# (en programas Java)

*José A. Mañas*

*7.3.2014*

## 1 Introducción

Cuando nos enfrentamos a algoritmos complejos el tiempo de ejecución es un factor importante que puede permitir utilizar un cierto algoritmo o buscar formas alternativas. A veces nos interesa el tiempo que requiere un cierto algoritmo; a veces simplemente queremos comparar dos algoritmos para estudiar su comportamiento relativo. En uno u otro caso mediremos tiempos de ejecución.

En paralelo al tiempo, a veces conviene tener medidas de los requisitos de memoria. Hay algoritmos o formas de programar que hacen un consumo de memoria tan generoso que los convierte en inviables más allá de problemas de juguete.

Por último, es uy frecuente que dispongamos de opciones alternativas, unas que consumen más tiempo pero requieren menos memoria, y otras que tiran de memoria para ganar en velocidad.

De una u otra forma, para decidir con qué nos quedamos, necesitamos medir.

## 2 Medidas de tiempos

La idea básica es muy simple.

Dado un trozo de código que queremos medir, se rodea de llamadas al reloj del sistema y la diferencia es el tiempo que le cuesta hacerlo

```
long t1 = System.currentTimeMillis();
   … lo que queremos medir …
long t2 = System.currentTimeMillis();
System.out.println(t2-t1);
```

Esto nos mide el tiempo en milisegundos.

currentTimeMillis() devuelve el número de milisegundos transcurridos desde el 1 de enero de 1970, media noche, meridiano de Greenwich. Es un reloj, y se ajusta a los cambios de hora cuando sincronizamos el reloj del equipo.

## 2.1 nanoTime

Se puede hacer con mayor precisión usando

```
long t1 = System. nanoTime ();
   … lo que queremos medir …
long t2 = System. nanoTime ();
System.out.println(t2-t1);
```

que mide nanosegundos, aunque conviene no engañarse pues el reloj del sistema no es tan preciso; es decir, que java no garantiza que su reloj interno se incremente cada nanosegundo, y probablemente sólo lo haga cada varios nanosegundos, con lo que la precisión no está definida y depende de la plataforma.

Habitualmente la resolución es de microsegundos.

Nadie garantiza que los contadores de currentTimeMillis() y nanoTime() estén sincronizados.

nanoTime() es un contador interno de java, que no es un reloj, y del que únicamente podemos derivar conclusiones midiendo diferencias.

## 2.2 Promedios

Prácticamente siempre conviene medir varias ejecuciones y tomar un valor promedio.

```
long t1 = System.currentTimeMillis();
for (int i= 0; i < N; i++) {
   funcion();
}
long t2 = System.currentTimeMillis();
System.out.println(t2-t1);
```

Durante las medidas conviene estabilizar lo que hace el equipo, pues la CPU se puede dedicar a muchas tareas y en la medida de tiempo sólo medidos dos instantes, pero no sabemos qué fracción de ese tiempo corresponde a nuestro objeto de medida.

Conviene no meter en el bucle más código que el que queremos medir. A menudo se introducen datos aleatorios para promediar, siendo conveniente sacar la parte aleatoria del bucle

```
int[] datos = new int[N];
for (int i= 0; i < N; i++)
   datos[i]= ….;

long t1 = System.currentTimeMillis();
for (int i= 0; i < N; i++) {
   funcion(datos[i]);
}
long t2 = System.currentTimeMillis();
System.out.println(t2-t1);
```

## 3 Demo

Usaremos el siguiente código a modo de demostración.

Se trata de calcular el valor medio y la desviación típica de un conjunto de números aleatorios. Elegiremos número aleatorios en una curva de Gauss con valor medio 0.0 y desviación típica 1.0, de forma que conocemos los valores esperados y podemos verificar que el programa calcula correctamente.

```
 1 import java.util.Random;
 2
 3 public class Stats {
 4     private static final Random RANDOM = new Random();
 5
 6     public static void main(String[] args) {
 7         show("java.version");
 8         show("java.vm.name");
 9         show("java.vm.version");
10         show("java.vm.info");
11
12         int n = Integer.parseInt(args[0]);
13         int veces = Integer.parseInt(args[1]);
14         double delta0 = 1;
15
16         for (int repeat = 0; repeat < veces; repeat++) {
17             long t1 = System.nanoTime();
18             calculo(n);
19             long t2 = System.nanoTime();
20             long delta = t2 - t1;
21             if (delta > delta0) {
22                 if ((delta - delta0) / delta0 > 2) {
23                     System.out.printf("%3d: %,10d%n", repeat, delta);
24                     delta0 = delta;
25                 }
26             } else {
27                 if ((delta0 - delta) / delta > 2) {
28                     System.out.printf("%3d: %,10d%n", repeat, delta);
29                     delta0 = delta;
30                 }
31             }
32         }
33     }
34
35     private static void show(String property) {
36         System.out.println(property + ": " + System.getProperty(property));
37     }
38
39     private static void calculo(int n) {
40         double sx = 0;
41         double sx2 = 0;
42         for (int i = 0; i < n; i++) {
43             double x = RANDOM.nextGaussian();
44             sx += x;
45             sx2 += x * x;
46         }
47         double media = sx / n;
48         double desviacion = Math.sqrt(sx2 / n - media * media);
49         if (Math.abs(media) > 0.5 || Math.abs(desviacion - 1) > 0.5)
50             System.out.printf("media: %f; desviacion= %f%n", media, desviacion);
51     }
52 }
```

Explicación del código:

| líneas | descripción |
|---|---|
| 7-10 | información sobre lo que está ejecutando |
| 12 | tamaño del problema: número de datos |
| 13 | repeticiones para observar las mejoras en ejecución |
| 14 | referencia para sólo imprimir cambios significativos en 23 y 28 |
| 16-32 | repeticiones |
| 18 | calcula la media y la desviación estándar de N números aleatorios |
| 49-50 | chequeo de que funciona bien;<br>y además nos aseguramos de que no se optimizan variables no usadas |
| 20 | tiempo de esta ejecución |
| 21-31 | imprimimos las ejecuciones donde hay cambios significativos |

# 4 Compilación, optimización y ejecución

Antes de ejecutar, el compilador traduce el código java a bytecodes

X.java → X.class

En tiempo de ejecución se traducen los bytecodes a instrucciones máquina de la CPU y en este proceso se aplican técnicas de optimización dinámica. De esto se encarga la máquina virtual de Java (JVM). El optimizador dinámico de llama HotSpot. JVM primero ejecuta los bytecodes interpretándolos, y sólo cuando ve que una porción de bytecodes se usa frecuentemente, pasa a generar código máquina para la plataforma donde está ejecutando. Con el tiempo, según va recolectando más y más información sobre el uso de las diferentes partes el programa, HotSpot va optimizando e incluso des-optimizando el código generado.

El optimizador hace (o puede hacer) muchas cosas

- cambiar llamadas a métodos por código en línea (por ejemplo, *getters* y *setters*
- cambiar bucles por su código repetido
- obviar polimorfismo
- pasar variables de RAM a registros de la CPU
- si un resultado no se va a usar, se elimina su cálculo (*dead code*)
- etc.

La lista es larga y cambiante. Java busca un código óptimo desde el punto de vista de espacio en memoria y tiempo de ejecución.

Eso es muy interesante; pero exactamente ¿qué estamos midiendo?

- ¿el tiempo de mi programa?
- ¿el tiempo del compilador-optimizador?
- ¿qué versión de mi código? ¿la original? ¿la optimizada? ¿…?

Dos consecuencias son inmediatas:

1. en tiempo de ejecución estamos midiendo el tiempo empleado nuestro código en ejecutarse, y también el tiempo dedicado por la JVM a compilar y optimizar nuestro código
2. la primera ejecución es la más lenta; luego depende de los criterios de HotSpot para ir optimizando

Por tanto deberemos tener una fase de precalentamiento para que el código se estabilice y HotSpot se está quieta.

## 4.1 Demo

Veamos una ejecución ingenua del código de la sección 2. Ejecutamos 1000 veces el cálculo de la media y la desviación de 200 números aleatorios

```
$ java Stats 200 1000
java.version: 1.7.0_40
java.vm.name: Java HotSpot(TM) Client VM
java.vm.version: 24.0-b56
java.vm.info: mixed mode

  0:    555,439
  5:    165,031
  9:     40,232
 14:  2,237,770
 15:     35,715
```

Se deben comentar varios aspectos:

- el optimizador ha entrado en acción en varias ocasiones marcando reducciones drásticas de tiempo de ejecución
- el runtime está haciendo más cosas que ejecutar nuestro código

Podemos ver cuándo entra en acción el optimizador

```
$ java -XX:+PrintCompilation Stats 200 1000

     68    1             java.lang.String::charAt (29 bytes)
     68    2             java.lang.String::hashCode (55 bytes)
     71    3             java.lang.String::indexOf (70 bytes)
     74    4             java.lang.String::indexOf (166 bytes)
     77    5
             java.lang.AbstractStringBuilder::ensureCapacityInternal (16 bytes)
     80    6             java.lang.String::lastIndexOf (52 bytes)

java.version: 1.7.0_40
java.vm.name: Java HotSpot(TM) Client VM
java.vm.version: 24.0-b56
java.vm.info: mixed mode

     81    7             java.lang.String::equals (81 bytes)
     82    8             java.lang.Object::<init> (1 bytes)
     88    9             sun.nio.cs.SingleByte$Decoder::decode (11 bytes)
     88   10             java.lang.CharacterData::of (120 bytes)
     89   11
             java.lang.CharacterDataLatin1::getProperties (11 bytes)
     89   12             java.lang.String::toLowerCase (472 bytes)
     90   13             java.lang.Character::toLowerCase (9 bytes)
     90   14             java.lang.CharacterDataLatin1::toLowerCase (39 bytes)
     91   15   !         java.io.BufferedReader::readLine (304 bytes)
     93   16
             sun.nio.cs.SingleByte$Decoder::decodeArrayLoop (154 bytes)
```

```
     94   17             java.io.Win32FileSystem::normalize (143 bytes)
     95   18             java.lang.AbstractStringBuilder::append (29 bytes)
     97   19             java.io.Win32FileSystem::isSlash (18 bytes)
    102   20             java.io.BufferedInputStream::getBufIfOpen (21 bytes)
    102   21  s          java.io.BufferedInputStream::read (49 bytes)
    103   22     n       java.lang.System::arraycopy (native)   (static)
  0:    583,766
    110   23             java.util.concurrent.atomic.AtomicLong::get (5 bytes)
    110   24
              java.util.concurrent.atomic.AtomicLong::compareAndSet (13 bytes)
    110   25     n       sun.misc.Unsafe::compareAndSwapLong (native)
    110   26             java.util.Random::next (47 bytes)
  4:    163,800
    111   27             java.util.Random::nextDouble (24 bytes)
    111   28  s          java.util.Random::nextGaussian (97 bytes)
    112   29             es.upm.dit.adsw.benchmark.Stats::calculo (117 bytes)
 10:     40,232
    114   30     n       java.lang.StrictMath::log (native)   (static)
 14:  2,233,253
 15:     42,284
    122   31             java.lang.String::length (6 bytes)
216:    152,304
217:     32,842
670:    231,536
671:     44,336
    142   32     n       java.lang.System::nanoTime (native)   (static)
```

Aunque la traza es un poco confusa de leer, básicamente podemos identificar

- la compilación de clases de java antes de empezar a ejecutar nuestro código,
- la optimización de varias funciones de la biblioteca Math
- la optimización de nuestro método calculo()

## 4.2 Inlining

El compilador en tiempo de ejecución también se toma la molestia de poner en línea código frecuentemente utilizado. Es decir, eliminar la llamada a un método por una ejecución directa de su contenido.

Por ejemplo

| código original | código en línea |
|---|---|
| int suma(int x1, int x2, int x3, int x4) {<br>  return suma(suma(x1, x2), suma(x3, x4));<br>}<br><br>int suma(int x1, int x2) {<br>  return x1 + x2;<br>} | int suma(int x1, int x2, int x3, int x4) {<br>  return x1 + x2 + x3 + x4;<br>} |

Evidentemente, hay casos más interesantes. Por ejemplo, java tiende a eliminar las llamadas a *setters* y *getters*, accediendo directamente a las variables internas de la clase.

Se puede trazar la actividad de *inlining* con los flags

```
java  -XX:+UnlockDiagnosticVMOptions -XX:+PrintInlining
```

## 4.3 Código muerto (*dead code*)

Si java detecta que un código o una variable no se utilizan para nada, puede obviar su cálculo. Esto puede llevar a bucles vacíos que a su vez se obvian y a métodos vacíos que a su vez se obvian. La mejora en prestaciones puede ser espectacular.

Ejemplo:

```
public class DeadCodeTest {
  public static void main(String[] args) {
    test1();
    test2();
  }

  public static void test1() {
    System.out.println("test1 1");
    if (true) return;
    System.out.println("test1 2");
  }

  public static void test2() {
    System.out.println("test2 1");
    if (false) return;
    System.out.println("test2 2");
  }
}
```

En test1, la última línea nunca se ejecuta. No así en test2, que sí se ejecuta.

Si miramos el código generado al compilar

```
$ javap -c DeadCodeTest
Compiled from "DeadCodeTest.java"
public class es.upm.dit.adsw.benchmark.DeadCodeTest {
  public es.upm.dit.adsw.benchmark.DeadCodeTest();
    Code:
       0: aload_0
       1: invokespecial #1    // Method java/lang/Object."<init>":()V
       4: return

  public static void main(java.lang.String[]);
    Code:
       0: invokestatic  #2    // Method test1:()V
       3: invokestatic  #3    // Method test2:()V
       6: return

  public static void test1();
    Code:
       0: getstatic     #4    // Field java/lang/System.out:Ljava/io/PrintStream;
       3: ldc           #5    // String test1 1
       5: invokevirtual #6    // Method java/io/PrintStream.println:(Ljava/lang/String;)V
       8: return

  public static void test2();
    Code:
       0: getstatic     #4    // Field java/lang/System.out:Ljava/io/PrintStream;
       3: ldc           #7    // String test2 1
       5: invokevirtual #6    // Method java/io/PrintStream.println:(Ljava/lang/String;)V
       8: getstatic     #4    // Field java/lang/System.out:Ljava/io/PrintStream;
      11: ldc           #8    // String test2 2
      13: invokevirtual #6    // Method java/io/PrintStream.println:(Ljava/lang/String;)V
```

```
      16: return
}
```

Donde vemos que el código generado para test1, efectivamente obvia lo que no sirve para nada.

# 5 Soluciones

Para medir tiempos debemos domesticar el runtime de java. ¿Cómo?

Quizás lo primero que debemos hacer es aclarar cómo está operando java en tiempo de ejecución, porque hay sus variantes. Para ello nos serán útiles las trazas iniciales

Ejemplo

```
java.version: 1.7.0_40
java.vm.name: Java HotSpot(TM) Client VM
java.vm.version: 24.0-b56
java.vm.info: mixed mode
```

Que se lee:

- en modo cliente
- en modo mixto (interpretado y compilado)

Y es un escenario diferente de este otro

```
java.version: 1.7.0_40
java.vm.name: Java HotSpot(TM) Server VM
java.vm.version: 24.0-b56
java.vm.info: compiled mode
```

Que se lee:

- en modo servidor
- en modo código compilado

Java, en modo CLIENTE procura que el programa arranque rápido, retrasando la compilación y primando un bajo consumo de memoria (*small memory footprint*); mientras que en modo servidor realiza una compilación más agresiva y temprana a costa de un arranque más lento y un mayor consumo de memoria.

Por otra parte, java puede interpretarlo todo (sin compilar nada) o compilarlo todo a la primera vez que se usa (casi como en los lenguajes de compilación-optimización estática antes de ejecutar, por ejemplo los compiladores de C) o en modo mixto que analiza el comportamiento dinámico del programa y va compilando-optimizando con la máxima información. Este es el modo mixto y es el que a la larga consigue un mejor resultado.

Para lo que nos ocupa, ¿qué hacemos para medir? ¿qué modo nos interesa?

## 5.1 Solución 1: todo interpretado

Podemos forzar a la máquina virtual de java para que no compile (no genere código de la máquina soporte) y toda la ejecución sea compilada.

Esta solución puede servir para comparar tiempos relativos de dos algoritmos, aunque las medidas de tiempo son perfectamente irreales. Sólo su valor relativo es informativo.

Y aun así, al prescindir de la compilación, no sabemos exactamente el comportamiento dinámico de la aplicación, pues cuando el programa ejecute de verdad, será compilado y optimizado, y puede que un algoritmo peor se pueda optimizar más que un algoritmo mejor sobre el papel y lo uno compense a lo otro.

Se hace de la siguiente forma

```
$ java –Xint
```

Ejemplo

```
$ java -Xint Stats 200 1000

java.version: 1.7.0_40
java.vm.name: Java HotSpot(TM) Client VM
java.vm.version: 24.0-b56
java.vm.info: interpreted mode

  0:    486,882
```

No hay más trazas porque el tiempo de ejecución siempre es el mismo.

## 5.2 Solución 2: precompilación

Podemos forzar a la máquina virtual a que lo compile los métodos la primera vez que vaya a ejecutarlos. Es el comportamiento clásico de los compiladores (por ejemplo de C) que compilan y optimizan antes de ejecutar.

Esta solución estabiliza las medidas, pero nos dará tiempos algo pesimistas ya que prescindimos de la mejor optimización que se puede obtener cuando vamos viendo el comportamiento dinámico del programa.

Se hace de la siguiente forma

```
$ java –Xcomp
```

Ejemplo

```
$ java -Xcomp Stats 200 1000

java.version: 1.7.0_40
java.vm.name: Java HotSpot(TM) Client VM
java.vm.version: 24.0-b56
java.vm.info: compiled mode

  0:  6,132,415
  1:    109,610
727:    512,745
```

```
728:      92,778
```

Aunque le decimos a java que siempre compile y nunca ejecute código interpretado, el runtime no compila hasta la primera vez que se ejecuta. Y, además, si durante la ejecución ve que el código compilado es mejorable, lo recompila mejor. Nótese que java ha ejecutado cientos de veces antes de decidir una optimización.

## 5.3 Solución 3: compilación agresiva en el arranque

Podemos poner la máquina virtual en modo servidor. Este modo usa más memoria (*heap*) y optimiza de forma más agresiva o contundente. El modo cliente arranca más deprisa; mientras que el modo servidor arranca más lentamente pero optimiza más.

Modo cliente

```
$ java –client
```

Modo servidor

```
$ java –server
```

Nota: no todas las versiones de java soportan ambas opciones. Las máquinas de 32 bits tienen a ser sólo clientes y las de 64 bits, sólo servidores.

Ejemplos

```
$ java -client Stats 200 2000

java.version: 1.7.0_40
java.vm.name:
    Java HotSpot(TM) Client VM
java.vm.version: 24.0-b56
java.vm.info: mixed mode

   0:     567,755
   5:     156,820
  10:      39,000
  14:   2,195,896
  15:      34,894
 171:     180,220
 172:      33,252
 620:     441,314
 621:      32,842
1001:     991,827
1002:      38,179
1004:     218,810
1005:      35,715
1874:     965,553
1875:      38,589
1914:     149,842
1915:      33,663
```

```
$ java -server Stats 200 1000

java.version: 1.7.0_40
java.vm.name:
    Java HotSpot(TM) Server VM
java.vm.version: 24.0-b56
java.vm.info: mixed mode

   0:     563,650
  26:     176,525
  40:   3,341,670
  41:     149,020
  50:   7,916,555
  51:      38,999
  99:   2,427,842
 100:      32,431
 528:   1,245,942
 529:      34,484
 582:     137,116
 583:      31,200
 947:     154,357
 948:      28,737
1585:     106,736
1586:      31,199
1928:      94,421
1929:      27,505
```

## 5.4 Solución 4: Precalentamiento

Es una solución un poco artesanal, pero teniendo cuidado es la que más se acerca al comportamiento real a medio-largo plazo.

Para sacar la optimización de los cálculos, tenemos el código ejecutando un buen rato antes de empezar a medir tiempos.

En el ejemplo anterior, modo cliente mixto, a la vista de que los tiempos se estabilizan notablemente tras unas 15 ejecuciones, podemos meter un bucle previo de precalentamiento

```
for (int repeat = 0; repeat < 20; repeat++)
    calculo(n);
```

Y ahora las medidas son estables:

```
$ java -client Stats 200 2000

java.version: 1.7.0_40
java.vm.name: Java HotSpot(TM) Client VM
java.vm.version: 24.0-b56
java.vm.info: mixed mode

  0:     32,842
```

# 6 Garbage collector

En Java se crean objetos con new y no hay que preocuparse de destruirlos pues el propio sistema se encarga de eliminar los objetos que no se están usando. Para ser más preciso, java va consumiendo memoria (*heap*) según se van creando objetos y cuando se le agita la memoria disponible, recolecta la memoria que no se está usando (*garbage collection*) y retorna al programa con memoria fresca. Si no es capaz de encontrar memoria, lanza una interrupción (*OutOfMemoryException*) y el programa termina malamente.

La recolección automática simplifica enormemente la programación y evita los errores derivados de una inadecuada liberación de objetos; pero consume tiempo en momentos que están fuera del control del programa.

## 6.1 Ejemplo.

Veamos un ejemplo que genera un gran número de objetos.

```
public class Romans {
   private static final Random random = new Random();

   public static void main(String[] args) {
      for (int i= 0; i < 20; i++) {
         long t1 = System.nanoTime();
         Map<Integer, String> table = new TreeMap<Integer, String>();
         for (int repeat = 0; repeat < 1000; repeat++) {
            int n = random.nextInt(4000);
            String s = toString(n);
            table.put(n, s);
         }
         long t2 = System.nanoTime();
         System.err.println(t2-t1);
      }
```

```
  }

  private static String toString(int n) {
    String s= "";
    if (n < 0){ s += "-"; n = -n; }
    while (n >= 1000) { s += "M"; n -= 1000; }
    if (n >= 900) { s += "CM"; n -= 900; }
    if (n >= 500) { s += "D"; n -= 500; }
    if (n >= 400) { s += "CD"; n -= 400; }
    while (n >= 100) { s += "C"; n -= 100; }
    if (n >= 90) { s += "XC"; n -= 90; }
    if (n >= 50) { s += "L"; n -= 50; }
    if (n >= 40) { s += "XL"; n -= 40; }
    while (n >= 10) { s += "X"; n -= 10; }
    if (n >= 9) { s += "IX"; n -= 9; }
    if (n >= 5) { s += "V"; n -= 5; }
    if (n >= 4) { s += "IV"; n -= 4; }
    while (n >= 1) { s += "I"; n -= 1; }
    return s;
  }
}
```

Lo ejecutamos sin precaución alguna

```
$ java es/upm/dit/adsw/benchmark/Romans
7302764
1803831
1128113
1117440
2869955
1018504
855116
844853
827200
874000
2055891
835001
834590
846905
837874
1776736
819811
843210
822685
827611
```

El resultado parece un tanto fuera de control. Es porque de vez en cuando java activa el recolector. Podemos verlo en acción

```
$ java -verbose:gc es/upm/dit/adsw/benchmark/Romans
7217786
1740200
1154797
```

```
1076798
1062839
[GC 4416K->240K(15872K), 0.0012985 secs]
2566169
848548
832948
841158
850600
[GC 4656K->268K(15872K), 0.0010279 secs]
2110080
855116
835000
850601
853474
[GC 4684K->312K(15872K), 0.0011232 secs]
2164268
855937
982377
849369
836232
```

Los primeros resultados son disparatados porque el compilador está generando y optimizando código. Luego se puede ver el efecto retardante del recolector.

## 6.2 Soluciones

Lo ideal es quitarse el recolector de encima.

### 6.2.1 Solución 1

Una opción es medir muchas veces y olvidar los resultados que se salen por incluir al recolector. Probablemente es la solución más sencilla y eficaz, aunque requiera un tratamiento manual.

### 6.2.2 Solución 2

Otra opción es sacar al recolector del bucle de medida, requiriendo su actuación fuera de la zona que estamos midiendo. Algo así

```
for (int i = 0; i < 20; i++) {
   long t1 = System.nanoTime();
   Map<Integer, String> table = new TreeMap<Integer, String>();
   for (int repeat = 0; repeat < 1000; repeat++) {
      int n = random.nextInt(4000);
      String s = toString(n);
      table.put(n, s);
   }
   long t2 = System.nanoTime();
   System.err.println(t2 - t1);
   System.gc();
}
```

En realidad esta solución no funciona muy fiablemente ya que la llamada a System.gc() es meramente indicativa y java no se compromete a llamar al compilador en ese momento. El resultado es impredecible.

### 6.2.3 Solución 3

Podemos ampliar la memoria disponible para evitar que haya que recolectar nada.

Java tiene dos parámetros para ello

- -Xms indica el tamaño inicial de memoria (heap)
- -Xmx indica el tamaño máximo de memoria (heap)

La idea es poner un tamaño suficiente para que las pruebas ejecuten sin tener que recolectar y arrancar con ese tamaño para que java no tenga que ir ampliando.

El tamaño necesario requiere de unos ciclos de prueba-error

En nuestro ejemplo, nos basta con unas 128Mbytes

```
$ java -verbose:gc -Xms128m -Xmx128m
       es/upm/dit/adsw/benchmark/Romans
7140198
1750874
1116619
1109639
1095682
1133860
958978
908894
922030
916284
945841
935168
915463
939683
972525
931473
919568
1317363
1016041
1052166
```

# 7 Referencias


- B. Goetz, “Java theory and practice: Dynamic compilation and performance measurement http://www.ibm.com/developerworks/library/j-jtp12214/
- B. Goetz, “Java theory and practice: Anatomy of a flawed microbenchmark” http://www.ibm.com/developerworks/java/library/j-jtp02225/index.html
- Oracle, “MicroBenchmarks” https://wikis.oracle.com/display/HotSpotInternals/MicroBenchmarks

- Oracle, “Java SE 6 HotSpot[tm] Virtual Machine Garbage Collection Tuning“ http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html