PRG (ETS de Ingeniería Informática) - Curso 2017-2018

Práctica 3. Medida empírica de la complejidad computacional

(2 sesiones)

Departamento de Sistemas Informáticos y Computación
Universitat Politècnica de València

Índice

1. Contexto y trabajo previo

En el contexto académico, esta práctica corresponde al *"Tema 2. Análisis de algoritmos. Eficiencia. Ordenación."*. Sus objetivos son los siguientes:

- Introducir el análisis de algoritmos en el laboratorio, usando un entorno real de programación: análisis empírico.
- Representar gráficamente el crecimiento de los recursos temporales empleados para confirmar los resultados teóricos.
- Inferir funciones aproximadas que definan el comportamiento temporal de un algoritmo.
- Usar los resultados empíricos para realizar comparaciones y predicciones.

Antes de la primera sesión de laboratorio, el/la alumno/a debe leer los apartados 1, 2 y 3 del boletín de prácticas. Las actividades de la práctica se realizarán durante 2 sesiones.

2. Medida del coste de la *búsqueda lineal*

En esta sección se presenta un problema completo de análisis empírico. El problema es la búsqueda lineal, es decir, la búsqueda de un elemento en un array posiblemente no ordenado.

2.1. Definición del problema

El problema de la búsqueda lineal consiste en, dado un array `a` de elementos de un cierto conjunto (por ejemplo, los números enteros) y dado un elemento arbitrario `e` de ese conjunto (un número entero concreto), devolver la primera posición del array que contenga al elemento `e`. Si `e` no se encuentrase en `a`, se devolvería un índice inválido (por ejemplo, -1) como resultado, a fin de señalar que el elemento no se ha encontrado. El siguiente método resuelve este problema sobre un `array` de `int`:

```
public static int linearSearch(int[] a, int e) {
    int i = 0;
    while (i < a.length && a[i] != e) { i++; }
    if (i < a.length) { return i; }
    else { return -1; }
}
```

2.2. Análisis de casos

Al enfrentarse al problema de analizar un algoritmo, el primer parámetro que se debe definir es la *talla* del problema. En este caso, la talla del problema es el tamaño del array, ya que determina el número de iteraciones de su bucle (por su condición `i < a.length`).

Además de esto, se debe estudiar si el algoritmo presenta *instancias significativas* o no. La búsqueda lineal presenta instancias significativas, es decir, *casos mejor y peor*. Estos casos se definen en base a la segunda condición del bucle (`a[i] != e`):

- **Caso mejor**: cuando el elemento `e` se encuentra en la primera posición de `a` (es decir, `a[0] == e`), ya que en ese caso el bucle no ejecutaría ninguna de sus iteraciones previstas.

- **Caso peor**: cuando el elemento `e` no se encuentra en `a`, ya que para verificar ese hecho se debe explorar completamente todo el array.

Con este análisis previo, se puede concluir que el coste del caso mejor es constante y el del caso peor es lineal. Por tanto, llamando $n =$`a.length`, las cotas asintóticas del algoritmo son $T(n) \in \Omega(1), O(n)$.

Para el estudio del caso promedio se pueden asumir ciertas simplificaciones como que el elemento a buscar siempre está en el array y que la probabilidad de encontrar el elemento en cualquier posición es la misma. Con estas suposiciones, la cota final del caso promedio es $T^{\mu}(n) \in \Theta(n)$.

2.3. Estructura de un experimento de medida

El análisis empírico se realiza tras el análisis teórico. Al diseñar un análisis empírico se deben tomar en consideración los siguientes puntos:

- **La medida de tiempo debe hacerse para varias tallas**: el objetivo es obtener una función de coste que tenga como parámetro la talla del problema; por tanto, deben emplearse diversas tallas para obtener el perfil de esa función.
- **Las instancias significativas deben medirse separadamente**: los casos mejor, peor y promedio presentan habitualmente distintas tasas de crecimiento y, por tanto, distinta función de coste; así pues, deben medirse en distintas partes del código.
- **Para obtener resultados significativos se deben tomar varias medidas**: una única medida por talla no es significativa, ya que puede verse afectada por condiciones del entorno (por ejemplo, la ejecución de otros procesos en el ordenador); por tanto, para garantizar resultados correctos se deben tomar varias medidas, promediándolas. Dicha media puede considerarse un resultado significativo.

La medida de tiempo de un algoritmo puede presentarse como el proceso siguiente:

1. Tomar tiempo actual t_I (tiempo inicial).
2. Ejecutar el algoritmo (método).
3. Tomar tiempo actual t_F (tiempo final).
4. La diferencia entre t_F y t_I es el tiempo que ha usado el algoritmo para resolver el problema.

Este proceso se puede hacer usando un reloj externo, pero es más preciso usar el reloj interno. Java aporta el método `static long nanoTime()`, en `java.lang.System`, que retorna el valor actual del temporizador más preciso del sistema en nanosegundos (aunque la resolución puede ser menor, pero al menos es de milisegundos). Por tanto, el código Java para la medida de tiempos es semejante a:

```
long ti, tf, tt;
ti = System.nanoTime();
// Llamada al método que se quiere temporizar
tf = System.nanoTime();
tt = tf - ti;
```

donde en la variable `tt` se registrará el tiempo que el método ha tardado en resolver el problema. Esta medida debe hacerse varias veces, calculándose el tiempo promedio. Sin embargo, para casos extremadamente rápidos (por ejemplo, el caso mejor de la búsqueda lineal), es habitual incluir el bucle de repeticiones dentro de la medida de tiempo, considerando despreciable la sobrecarga del bucle. O también se puede repetir la medida un número de veces bastante grande (por ejemplo, en el caso mejor de la búsqueda lineal, un número de veces mucho mayor que para los casos peor y promedio).

Finalmente, las medidas de tiempo deben representarse apropiadamente. La forma usual es usar una tabla que muestre en cada fila la talla del problema y los tiempos medidos para

cada una de las instancias. Dichos tiempos deben venir expresados en alguna unidad de medida (microsegundos, milisegundos, etc.) que facilite la lectura de los valores, siendo aconsejable que dicha unidad aparezca como un comentario en la tabla. Nótese, además, que, al tratarse de valores promedio, es razonable que los valores vengan expresados con decimales. La forma típica de esta tabla es similar a la siguiente:

```
# Busqueda lineal. Tiempos en microsegundos
# Talla       Mejor      Peor     Promedio
#----------------------------------------
   10000      1.793      4.957      3.187
   20000      1.790      8.306      4.964
   30000      1.793     11.589      6.662
   40000      1.793     15.002      8.353
   50000      1.793     18.371     10.131
   60000      1.793     21.803     11.856
   .....
```

Actividad 1: creación del paquete `pract3` en el proyecto *prg*

Abre *BlueJ* en el proyecto de trabajo de la asignatura (`prg`) y crea un nuevo paquete `pract3` con las clases `MeasurableAlgorithms.java` que contiene, entre otros, el método `linearSearch(int[], int)` y `MeasuringLinearSearch.java` que implementa el código que realiza el análisis para las distintas instancias significativas de dicho algoritmo. Tienes disponibles dichas clases en `Recursos/Laboratorio/Práctica 3`, dentro del `poliformaT` de `PRG`.

Actividad 2: obtención de tiempos de búsqueda

Ejecuta el método `measuringLinearSearch()` de la clase `MeasuringLinearSearch` para obtener la tabla de resultados de tiempos y guárdala en un fichero de nombre `linearSearch.out`; para ello puedes utilizar la opción correspondiente disponible en la ventana de ejecución de *BlueJ* o redireccionar la salida del programa ejecutando desde la línea de comandos:

```
java MeasuringLinearSearch > linearSearch.out
```

3. Representación gráfica: *Gnuplot*

Los resultados numéricos usualmente se interpretan mejor con su representación gráfica. En esta sección se muestra cómo usar la herramienta *Gnuplot* para obtener representaciones gráficas de los resultados y para obtener funciones de coste que se aproximen a los resultados empíricos, que pueden usarse para comparar adecuadamente los algoritmos y para obtener predicciones. Para poder usar esta herramienta escribiremos `gnuplot` en la línea de comandos (terminal). *Gnuplot* acepta órdenes con modificadores; las órdenes más relevantes son:

- **plot**: dibuja datos de un fichero o funciones predefinidas; algunos modificadores son:
  - *fichero*: se especifica entre comillas dobles e indica dónde se encuentra el fichero con los datos a dibujar; las líneas del fichero que empiezan por el símbolo `#` se ignoran.

    - **title** *texto*: especifica el título que debe darse a los puntos (leyenda).
    - **using** *i:j*: especifica las columnas del fichero de datos que se usarán (i para el eje X, j para el eje Y).
    - **with** *formato*: especifica el formato de dibujo (los usuales son `lines`, `points` y `linespoints`).

- **replot**: con el mismo significado y modificadores que *plot*, pero sin borrar la vista gráfica, con lo que permite ver distintas gráficas a la vez; *replot* a solas redibuja la vista gráfica.

- **set xrange** *[inicio:fin]*, **set yrange** *[inicio:fin]*: fija el rango de valor del eje X (respectivamente Y) entre *inicio* y *fin*.

- **set xtics** *intervalo*, **set ytics** *intervalo*: fija el intervalo entre las marcas en el eje X (respectivamente Y).

- **set xlabel** *texto*, **set ylabel** *texto*: fija la etiqueta del eje X (respectivamente Y).

- **set title**: fija la etiqueta título de la gráfica.

- **load** *fichero*: carga un fichero de texto con órdenes de *Gnuplot* que son ejecutadas por *Gnuplot*.

- **fit** *función fichero* **using** $i:j$ **via** *parámetros*: permite ajustar una función predefinida con ciertos parámetros libres a un conjunto de datos de un fichero. En donde:

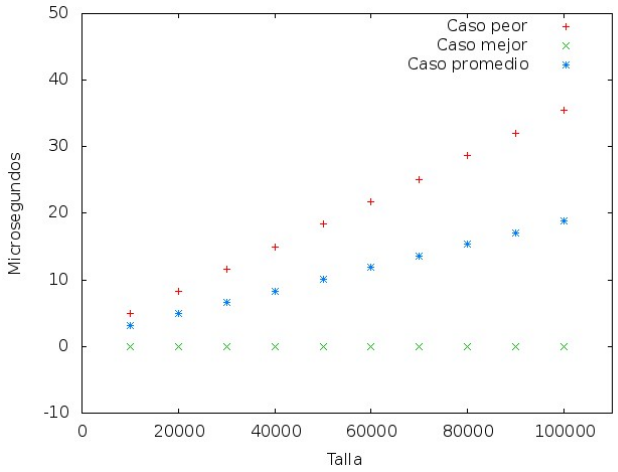
    - *función*: indica el nombre de la función a ajustar.
    - *fichero*: indica el nombre del fichero con los datos a ajustar (se especifica entre comillas dobles).
    - **using** $i:j$: especifica las columnas del fichero de datos que se usarán (i para el eje X y j para el eje Y).
    - **via** *parámetros*: especifica los parámetros (separados por comas) de la función a ajustar.

Actividad 3: representación y análisis de los resultados empíricos

Para representar los tiempos de la tabla almacenada en el fichero `linearSearch.out`, arranca *Gnuplot* y escribe las siguientes órdenes:

```
gnuplot> set xrange [0:110000]
gnuplot> set yrange [-10:50]
gnuplot> set xtics 20000
gnuplot> set ytics 10
gnuplot> set xlabel "Talla"
gnuplot> set ylabel "Microsegundos"
gnuplot> plot "linearSearch.out" using 1:2 title "Caso mejor"
gnuplot> replot "linearSearch.out" using 1:3 title "Caso peor"
gnuplot> replot "linearSearch.out" using 1:4 title "Caso promedio"
```

La imagen mostrada debe ser semejante a la presentada en la siguiente figura:



Para salvar la salida gráfica en un fichero .jpg, debes ejecutar las órdenes que siguen:

```
gnuplot> set term jpeg
gnuplot> set output "linearSearch.jpg"
gnuplot> replot
```

El fichero `linearSearch.jpg` se salvará en el directorio actual con los contenidos de la vista gráfica. Para redirigir la salida de nuevo al terminal se debe ejecutar:

```
gnuplot> set term x11
gnuplot> set output
```

Actividad 4: aproximación de funciones a los resultados empíricos

Gnuplot permite ajustar los valores de las columnas de un fichero de datos a una función cuyo tipo se establece de antemano. Por ejemplo, si se sospecha que determinados valores siguen una distribución cuadrática se puede, mediante *Gnuplot*, ajustar dichos valores a una parábola. El resultado del proceso de ajuste, serán los coeficientes de la parábola que mejor se aproxime a los datos del fichero.

Para hacer dicho tipo de ajuste se utiliza la orden `fit` que, como se ha dicho, permite obtener funciones aproximadas que muestren el comportamiento de un algoritmo. Por ejemplo, como el caso peor y promedio de la búsqueda lineal presentan un coste lineal, es posible conocer la diferencia entre ambos, ajustando previamente cada una de las columnas de datos correspondientes a una función de dicho tipo (lineal) para, después, comparar las funciones obtenidas.

Ejecuta la siguiente secuencia de comandos *Gnuplot* para realizar el ajuste de los datos del caso peor mediante una función lineal:

```
gnuplot> f(x)=a*x+b
gnuplot> fit f(x) "linearSearch.out" using 1:3 via a,b
```

El resultado será similar al que sigue:

```
...
Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a               = 0.000339198      +/- 1.051e-06    (0.3098%)
b               = 1.47013          +/- 0.0652       (4.435%)
...
```

Ahora haz lo mismo pero ajustando los datos del caso promedio mediante otra función lineal:

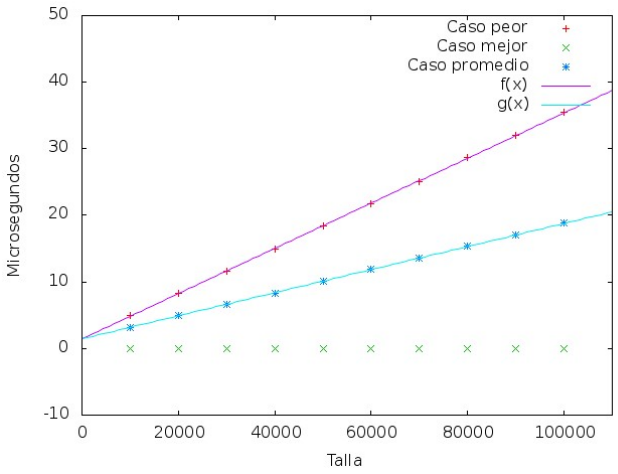
```
gnuplot> g(x)=c*x+d
gnuplot> fit g(x) "linearSearch.out" using 1:4 via c,d
...
Final set of parameters            Asymptotic Standard Error
=======================            ==========================
c               = 0.000173287      +/- 3.002e-07    (0.1733%)
d               = 1.4602           +/- 0.01863      (1.276%)
...
```

por tanto, se ve que la relación de crecimiento entre los casos peor y promedio es la relación entre las pendientes de las funciones lineales correspondientes, es decir, $0{,}000339198/0{,}000173287 \approx 2$ para los datos de nuestra tabla. Por tanto, se puede inferir que el caso peor es aproximadamente el doble de lento que el caso promedio.

Las funciones de ajuste recién estimadas se pueden graficar junto a los anteriores resultados:

```
gnuplot> replot f(x),g(x)
```

apreciando así que los datos experimentales siguen con gran fidelidad las funciones calculadas.



La estimación de estas funciones aproximadas se puede utilizar para realizar predicciones. Por ejemplo, en el caso promedio, para un array con un tamaño de mil millones de enteros (10^9), el tiempo requerido para su ejecución según estos cálculos se estimaría de la siguiente manera:

$$g(x) = c * x + d = 0{,}000173287 * x + 1{,}4602$$

y sustituyendo x por el valor de la talla del que queremos realizar la predicción, 10^9,

$$g(10^9) = 0{,}000173287 * 10^9 + 1{,}4602 \approx 173288$$

microsegundos, es decir, algo más de un sexto de segundo, aproximadamente.

4. Análisis del coste de la ordenación por *selección directa*

La estrategia Selección Directa ordena un array con una complejidad temporal $\Theta(n^2)$, siendo n el número de elementos del array, y no presenta instancias significativas para el coste. Por ello, es suficiente realizar el estudio del coste independientemente del contenido del array, considerando entonces que los datos del mismo se generan aleatoriamente.

Actividad 5: método para rellenar un array con valores aleatorios

Añade a tu paquete `pract3` la clase `MeasuringSortingAlgorithms.java` disponible en la `poliformaT` de `PRG`. Para el análisis del coste del método `selectionSort(int[])` que implementa esta estrategia de ordenación, definido en la clase `MeasurableAlgorithms`, debes escribir en la clase `MeasuringSortingAlgorithms` un método para rellenar un array con enteros generados de forma aleatoria de acuerdo al siguiente perfil:

```
/** Rellena los elementos de un array a con valores aleatorios
 *  entre 0 y a.length-1.
 *  @param a int[], el array.
 */
private static void fillRandomArray(int[] a) { ... }
```

Actividad 6: tiempo de ejecución de una única llamada al método

Completa el método `measuringSelectionSort()` de la clase `MeasuringSortingAlgorithms` con las instrucciones necesarias para:

1. Crear un array `a` de tamaño 100, utilizando el método `createArray(int)`.
2. Rellenar el array con datos aleatorios utilizando el método `fillRandomArray(int[])`.
3. Llamar al método `System.nanoTime()` para obtener sobre una variable `ti` (de tipo `long`) el valor del reloj (en nanosegundos) antes de llamar al método que queremos temporizar.
4. Llamar al método `selectionSort(int[])` de la clase `MeasurableAlgorithms` para ordenar el array `a`.
5. Volver a llamar al método `System.nanoTime()` para tener en la variable `tf` el valor del reloj una vez acabada la ordenación.
6. Calcular la diferencia de tiempos `(tf - ti)` para saber el tiempo que ha necesitado el método `selectionSort(int[])` para ordenar el array `a`.
7. Mostrar por pantalla una fila de datos con la talla del array y el tiempo en microsegundos.

Actividad 7: tiempo de ejecución para una única talla dada

Como se ha comentado anteriormente, tomar una única medida para estimar el coste de un método para una determinada talla no es un procedimiento adecuado, ya que esta medida puede verse afectada por condiciones del entorno. Así, para garantizar resultados correctos se

debe repetir la toma de la medida y después promediar sobre el número de medidas que se hayan tomado.

Define la constante `REPETICIONESQ` (con valor 200) en la clase `MeasuringSortingAlgorithms` y completa el método `measuringSelectionSort()` de esta clase con un bucle para repetir ese número de veces el bloque de instrucciones que ya tenemos escrito de la actividad anterior y obtén el tiempo promedio. Date cuenta que en aras de minimizar la dependencia de una instancia particular del array rellenado de forma aleatoria, lo que conviene es que **en cada repetición se rellene de nuevo el array con valores aleatorios diferentes**[1]. De no ser así, fíjate que además, el array a ordenar ya estaría ordenado a priori a partir de la primera repetición, lo que no invalidaría el experimento formalmente pero estadísticamente no sería significativo. Obtén el tiempo medio por repetición (en μs) y muéstralo en pantalla junto a la talla del array.

Actividad 8: tiempo de ejecución para diferentes tallas

Define en la clase `MeasuringSortingAlgorithms` las constantes `INITALLA`, `MAXTALLA` e `INCRTALLA` para representar respectivamente el valor de la talla más pequeña a considerar (1000), el valor más grande (10000) y el incremento de talla (1000). Completa el método `measuringSelectionSort()` de esta clase añadiendo un bucle para repetir el cálculo del tiempo de ejecución para tallas desde `INITALLA` hasta `MAXTALLA` con incrementos de `INCRTALLA`; es decir, para tallas 1000, 2000, 3000, ..., 10000. El método debe mostrar por pantalla una tabla como la que sigue en la que el tiempo se dé en microsegundos:

```
# Selección directa. Tiempos en microsegundos
#  Talla   Promedio
# ------------------
    1000    403.346
    2000   1348.255
    3000   3061.948
   ...
```

Actividad 9: representación gráfica de los resultados

- Ejecuta el método `measuringSelectionSort()`, guarda la tabla resultado en un fichero y, utilizando *Gnuplot*, muestra los resultados en una gráfica en la que el eje X represente la talla y el eje Y el tiempo de ordenación en microsegundos. Ten en cuenta que las características del gráfico a representar se tienen que corresponder con los datos de la tabla.
- Ajusta los resultados obtenidos a una función cuadrática (`f(x)=a*x*x+b*x+c`), observando los valores de los parámetros de ajuste.
- Vuelve a construir la gráfica mostrando, además de los puntos experimentales, la función de ajuste. Etiqueta adecuadamente los ejes, los puntos, y la función de ajuste, añade un título a la gráfica y guárdala en un fichero .jpg.
- Utiliza la función de ajuste para predecir cuál sería el tiempo necesario para ordenar con el método `selectionSort(int[])` un array de 800000 enteros.

[1] Así, el array se crea una única vez pero se rellena de nuevo en cada repetición.

5. Análisis del coste de la ordenación por *inserción directa*

La estrategia Inserción Directa ordena un array con una complejidad temporal $\Omega(n)$ y $O(n^2)$, siendo n el número de elementos del array, presentando instancias significativas para el coste: el caso mejor cuando el array ya está ordenado (de forma creciente), y el caso peor cuando el array también está ordenado, pero al revés, es decir, de forma decreciente. Por eso, es necesario realizar el estudio del comportamiento del método en el caso mejor (con arrays ya ordenados), en el caso peor (con arrays ordenados de forma decreciente), y en el caso promedio (con arrays generados aleatoriamente).

Actividad 10: creación de arrays ordenados

En la clase `MeasurableAlgorithms` está definido el método `insertionSort(int[])` que implementa esta estrategia de ordenación. Para poder analizar este método se deben escribir en la clase `MeasuringSortingAlgorithms` dos métodos para rellenar arrays de enteros, de manera que su contenido esté ordenado, respectivamente, de forma creciente (con valores desde `0` hasta `a.length-1`) y decreciente (con valores desde `a.length-1` hasta `0`); sus perfiles serían:

```
/** Rellena los elementos de un array a de forma creciente,
 *  con valores desde 0 hasta a.length-1.
 *  @param a int[], el array.
 */
private static void fillArraySortedInAscendingOrder(int[] a) { ... }

/** Rellena los elementos de un array a de forma decreciente,
 *  con valores desde a.length-1 hasta 0.
 *  @param a int[], el array.
 */
private static void fillArraySortedInDescendingOrder(int[] a) { ... }
```

Actividad 11: análisis empírico del coste del método `insertionSort`

Completa el método `measuringInsertionSort()` de la clase `MeasuringSortingAlgorithms` para estudiar el comportamiento del método `insertionSort(int[])` para los casos peor, mejor, y promedio, tallas desde `INITALLA` hasta `MAXTALLA` con incrementos de `INCRTALLA`; es decir, para tallas 1000, 2000, 3000, ..., 10000. Para los casos peor y promedio, la medida se repetirá `REPETICIONESQ` veces (200). En el caso mejor (igual que en la búsqueda lineal), la medida debe repetirse un número mayor de veces: `REPETICIONESL` (con valor 20000). El método debe mostrar por pantalla una tabla como la que sigue:

```
# Inserción directa. Tiempos en microsegundos
#  Talla    Mejor     Peor     Promedio
# ------------------------------------
    1000   1.115    422.532    134.647
    2000   3.567    848.167    405.849
    3000   5.625   1904.827    919.622
   ...
```

Fíjate que para el algoritmo de inserción directa, la recomendación hecha en la actividad 7, relacionada con que **en cada repetición se rellena de nuevo el array con valores aleatorios diferentes**, en este caso resulta de obligado cumplimiento. Si no, los resultados en los casos peor y promedio no serían válidos, ya que el array estaría ya ordenado (caso mejor) en las restantes repeticiones. Recuerda crear el array una única vez para cada talla (no para cada repetición!!!).

Actividad 12: representación gráfica de los resultados

- Ejecuta el método `measuringInsertionSort()`, guarda la tabla resultado en un fichero y, utilizando *Gnuplot*, muestra los resultados en una gráfica en la que el eje X represente la talla y el eje Y el tiempo de ordenación en microsegundos. Debes dibujar los puntos experimentales obtenidos para los tres casos. Ten en cuenta que las características del gráfico a representar se tienen que corresponder con los datos de la tabla.
- Ajusta los resultados obtenidos a las funciones previstas del análisis teórico (caso mejor a función lineal, casos peor y promedio a funciones cuadráticas) observando los valores de los parámetros de ajuste.
- Vuelve a construir la gráfica mostrando, además de los puntos experimentales, las tres funciones de ajuste. Etiqueta adecuadamente los ejes, los puntos, y las funciones de ajuste, añade un título a la gráfica y guárdala en un fichero .jpg.
- Utiliza las funciones de ajuste para predecir cuál sería el tiempo necesario para ordenar un array de 800000 enteros mediante el método `insertionSort(int[])` si: a) el array ya está ordenado crecientemente; b) si el array también está ordenado, pero en sentido decreciente; y c) para un array con valores aleatorios.

6. Análisis del coste de la ordenación por *mezcla o MergeSort* (actividad adicional)

Completa el método `measuringMergeSort()` de la clase `MeasuringSortingAlgorithms` para estudiar el comportamiento del método `mergeSort(int[], int, int)` definido en la clase `MeasuringSortingAlgorithms`. En este ejercicio, te sugerimos que uses tallas que sean potencias de 2. Para ello, puedes definir en la clase `MeasuringSortingAlgorithms` las constantes `INITALLA_MERGE` y `MAXTALLA_MERGE` para representar respectivamente el valor de la talla más pequeña a considerar (2^{10}) y el valor más grande (2^{19}), y que el incremento de talla sea: `talla *= 2`. El número de llamadas al método de ordenación para obtener valores significativos, puede ser el mismo que en el análisis de los otros dos algoritmos de ordenación (200).

7. Evaluación

Esta práctica forma parte del primer bloque de prácticas que será evaluado en el primer parcial de PRG. El valor de dicho bloque es de un 40 % con respecto al total de las prácticas. El valor porcentual de las prácticas en la asignatura es de un 20 % de su nota final.