Estimación empírica del orden de complejidad de un algoritmo

Recursos necesarios

- Para esta práctica usaremos los siguientes componentes del proyecto ParteComun del repositorio:
  - Paquete us.lsi.curvefitting para el ajuste curvas
  - Librerías *matplotlib4j-0.5.1.jar* y *commons-math3-3.6.1.jar* (se encuentran dentro del proyecto a importar)
- Para evitar errores de memoria insuficiente (StackOverflowError), es necesario aumentar el tamaño en las opciones de ejecución de Eclipse. En Run Configurations -> Arguments -> VM Arguments debemos añadir -Xss50m.
- Instalar Python. Se recomiendan los siguientes pasos de instalación:
  - Descargar e instalar Python 3.10 con permisos de administrador (marcar la opción de añadir a PATH):
    - https://www.python.org/downloads/release/python-3100/
  - En línea de comandos ejecutar *pip install matplotlib*
  - En caso de que eclipse no encuentre Python, como paso adicional podemos indicarle la ruta a la hora de crear un Plot. Se muestra un ejemplo:
    - Plot plt = Plot.create(PythonConfig.pythonBinPathConfig("C:\\Users\\<NombreUsuario>\\AppData\\Local\\Programs\\Python\\Python310\\python.exe"));

Introducción

El objetivo de esta práctica es la estimación empírica del orden de complejidad de un algoritmo. Para ello, se realizarán experimentos teniendo en cuenta distintas entradas para los algoritmos, con distintos tamaños y casos. Con los resultados obtenidos, se usará un software (Matlab en nuestro caso) para obtener la curva que mejor se ajuste a dichos resultados.

Consideraciones previas

La medición del tiempo de ejecución de métodos o programas en general en Java no es un asunto trivial, ya que influyen distintos factores que afectan al tiempo medido. Es decir, el tiempo que se obtiene al ejecutar el código

```
long t0 = System.nanoTime();
metodo();
long t1 = System.nanoTime();
System.out.println(t1 – t0);
```

no se corresponde exactamente con lo que el código del método en cuestión tardaría en ejecutarse.

Entre los distintos factores que pueden influir en la medición de tiempos de ejecución, se pueden destacar los siguientes:

- El propio uso de *nanoTime* supone una distorsión. Dicho método también necesita tiempo para ejecutarse, y para entender su influencia hay que considerar su latencia (tiempo que pasamos en la llamada) y granularidad (diferencia mínima que obtenemos entre llamadas consecutivas). Los valores de dichos parámetros varían entre las distintas plataformas (hardware y SO), y también dependen del número de núcleos de una manera no proporcional en todos los casos. Valores típicos de latencia suelen estar en las decenas de ns, y de granularidad entre decenas de ns y centenas de ns (Windows) o incluso más, dependiendo del número de hilos de ejecución y de la sobrecarga de estos.
- Optimizaciones de código producidas por el compilador o intérprete, principalmente:
    - Eliminación de código inútil, sobre todo en métodos que devuelven void.
    - Evaluación parcial de expresiones constantes (Constant folding).
    - Desenrollado de bucles (loop unrolling).
    - Optimizaciones dinámicas que tienen en cuenta la información sobre la propia. ejecución de los métodos del programa.
- Balanceo de carga en los procesadores realizada por el scheduler del SO.
- Capacidad de la memoria caché de los procesadores.
- Ejecución de otros procesos simultáneos, tanto externos como internos (recolector de basura).

Algunas recomendaciones para limitar esos efectos son:

- Evitar código (cuyo tiempo se quiera medir) que pueda ser optimizado por el compilador
- Usar los métodos antes de medirlos, bien individualmente o de manera conjunta (warmup)
- Promediar el tiempo de ejecución de múltiples usos del método, sobre todo si su tiempo de ejecución es muy pequeño.
- Repetir los experimentos varias veces.
- Evitar o disminuir todo lo posible la ejecución de otros procesos simultáneamente.

Pasos a seguir para buscar la función del tiempo de ejecución de un algoritmo

1. Generar fichero csv con los tiempos y tamaños.
2. Ajustar datos a función (curve fitting).
3. Mostrar el resultado gráficamente

Ajuste de curvas

Una vez hayamos obtenido el fichero de salida de los tiempos de ejecución, procedemos al ajuste de curvas. El fichero de salida debe tener el siguiente formato, donde X representa el tamaño del problema e Y el tiempo de ejecución:

X1, Y1
X2, Y2
X3, Y3
...

El paquete us.lsi.curvefitting nos proporciona las clases Exponential y PolinomialLog para hacer ajustes con funciones de orden exponencial o polinómico/logarítmico, respectivamente.

El siguiente trozo de código muestra un ejemplo de ajuste de una lista de tiempos de ejecución a una función de tipo `a*n^b*(ln n)^c + d`, usando la clase PolinomialLog

```
import org.apache.commons.math3.fitting.WeightedObservedPoint;
import us.lsi.curvefitting.DataCurveFitting;
import us.lsi.curvefitting.PolinomialLog;

// Inicialización de los cuatro coeficientes: a, b, c, d
double[] start = {1.,1.,1.,1.};


// Lectura de fichero de tiempos de ejecución
List<WeightedObservedPoint> points =
DataCurveFitting.points(ficheroSalida);


// Cálculo de coeficientes
final double[] coeffs = PolinomialLog.of().fit(points,start);

System.out.println(String.format("Solutions=
a=%.2f,b=%.2f,c=%.2f,d=%.2f",
coeffs[0],coeffs[1],coeffs[2],coeffs[3]));
```

Mostrar ajuste gráficamente

```
import org.apache.commons.math3.fitting.WeightedObservedPoint;
import com.github.sh0nk.matplotlib4j.Plot;
import com.github.sh0nk.matplotlib4j.PythonExecutionException;

List<WeightedObservedPoint> points =
DataCurveFitting.points(dataFile);

// Ordenar los puntos para que se muestren correctamente

List<WeightedObservedPoint> pointsSorted = new
ArrayList<>(points);
pointsSorted.sort(Comparator.comparing(x->x.getX()));

// Tamaños de problema (Eje X)

List<Double> lx =
pointsSorted.stream().map(p>p.getX()).toList();

// Puntos registrados de tiempo de ejecución (Eje Y)

List<Double> datos =
pointsSorted.stream().map(p->p.getY()).toList();

// Puntos de la curva de ajuste (Eje Y)

Function<Double,Double> f =
x->PolinomialLog.of().value(x,coeffs)
```

```
List<Double> ajuste = pointsSorted.stream().map(p-
>f.apply(p.getX())).toList();
// Gráfica

Plot plt = Plot.create();

plt.plot().add(lx,ajuste).label("ajuste").linestyle("-");

plt.plot().add(lx,datos).label("datos").linestyle(":")
.linewidth(3.5);

plt.title(title + ":     " + coeffs);
plt.legend();
plt.xlabel("tamano");
plt.ylabel("tiempo");


try {
      plt.show();
      }
catch (IOException e) {
      e.printStackTrace();
      }
catch (PythonExecutionException e) {
      e.printStackTrace();
      }
}
```