Estimación empírica del orden de complejidad de un algoritmo

Introducción

Tras estudiar y analizar en clases de teoría el orden de complejidad de un algoritmo cuando el tamaño del problema crece asintóticamente, el objetivo de esta práctica es analizarlo empíricamente con grandes tamaños que a su vez permitan obtener la respuesta del algoritmo en no demasiado tiempo. Para ello, se realizarán distintos experimentos teniendo en cuenta distintas entradas del algoritmo: con distintos tamaños y casos.

Antes de empezar: Consideraciones previas

La medición del tiempo de ejecución de métodos o programas no es un asunto trivial, ya que influyen distintos factores que afectan al tiempo medido. En Java, el tiempo que se obtiene al ejecutar el siguiente 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 del tiempo 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).

Para limitar los efectos anteriores, debemos tener en cuenta las siguientes **recomendaciones**:

- 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.

Para empezar: Recursos necesarios

- En esta práctica usaremos componentes del paquete *us.lsi.curvefitting* del proyecto *ParteComun* y algoritmos del proyecto *EjemplosParteComun*, ambos en el repositorio de proyectos de la asignatura. Por ello, será necesario:
    1. Añadir los proyectos *ParteComun* y *EjemplosParteComun* al Modulepath de nuestro proyecto.
    2. Requerir el módulo *ejemplos_parte_comun* en el módulo de nuestro proyecto.

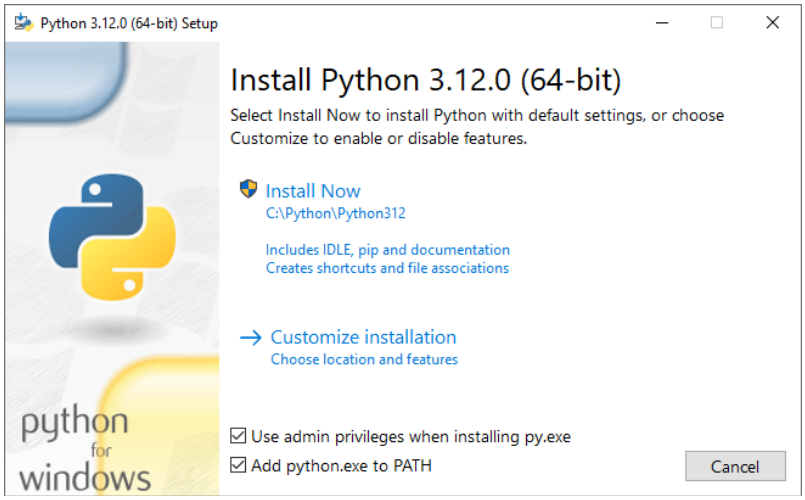
- Al ejecutar algoritmos recursivos, si el tamaño del problema es elevado puede ser necesario aumentar la cantidad de memoria que la máquina virtual de Java utiliza para almacenar las llamadas a los métodos. Las llamadas a los métodos requieren direcciones de memoria en las que ser almacenadas al igual que las referencias a los objetos que forman parte de su código. Si al ejecutar un test obtenemos una excepción del tipo *StackOverflowError*, debemos ampliar el tamaño de dicha memoria (cuyo valor por defecto es 1MB) en *Run Configurations* → *Arguments* → *VM Arguments*. Mediante la opción -Xss podemos ampliar dicho tamaño a por ejemplo 2MB de la forma: **-Xss2m** (también -Xss2M). Servirá únicamente para dicho test.

- Para aquellos algoritmos que requieran algún agregado de datos (por ejemplo listas o conjuntos) cuyo tamaño determine el tamaño del problema, puede ser necesario aumentar la cantidad de memoria que la máquina virtual de Java utiliza para almacenar sus elementos. Si al ejecutar un test obtenemos una excepción del tipo *OutOfMemoryError: Java heap space*, debemos ampliar el tamaño de dicha memoria en *Run Configurations* → *Arguments* → *VM Arguments*. Mediante las opciones -Xms y -Xmx podemos establecer respectivamente el tamaño inicial y máximo de dicha memoria (montículo) de forma análoga a -Xss (por ejemplo **-Xmx1500m**). Nuevamente, servirá únicamente para dicho test.

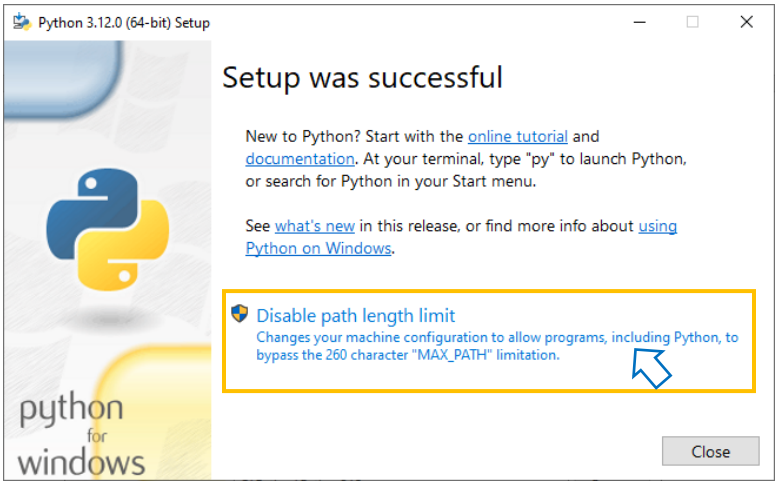
- Los componentes del paquete *us.lsi.curvefitting* del proyecto *ParteComun* que utilizaremos necesitan librerías que enlazan con otras externas implementadas en **Python**, por lo que será necesario tenerlo instalado. Para ello:

  1. Descargue y ejecute el instalador de Python. Puede instalar la última versión (3.12) desde la siguiente url: https://www.python.org/downloads/

  2. Al instalar Python, marque las opciones *usar privilegios de administrador* y *añadir a PATH*. La siguiente imagen muestra un ejemplo en Windows.

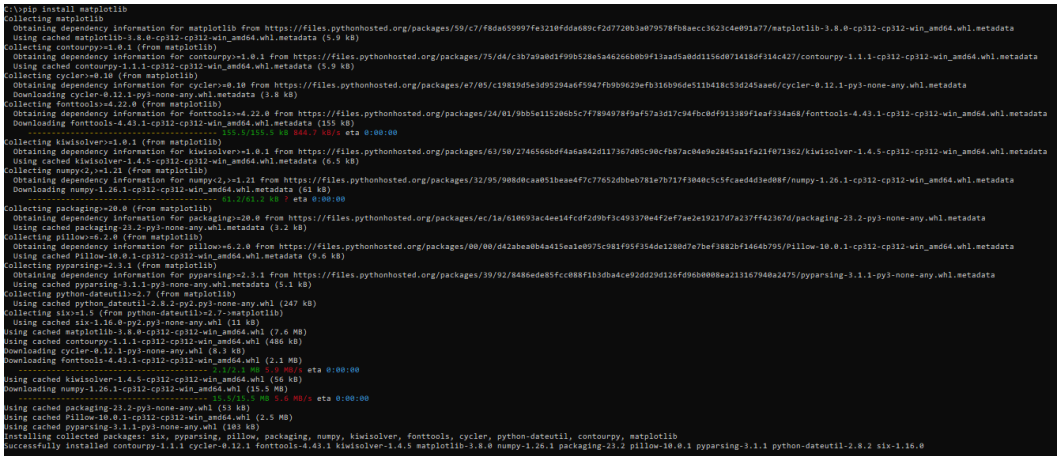


  3. Si eligió instalar Python en la ruta por defecto (para la cuenta de usuario en uso en su SO), en Windows *"C:\Users\<NombreUsuario>\AppData\Local\Programs\Python\Python31x\python.exe"*, o bien en otra ruta que incluye numerosas carpetas o directorios y éstos con nombres tales que la longitud en número de caracteres de la ruta de instalación no es pequeña, puede necesitar deshabilitar la limitación de 260 caracteres para aplicaciones a incluir en el PATH. Para evitar problemas, haga clic en la opción del final del proceso de instalación *Disable path length limit*. La siguiente imagen muestra un ejemplo en Windows.



Puede hacerlo también de forma manual según su SO. En Windows, puede usar el editor del registro y cambiar la clave *HKEY_LOCAL_MACHINE \SYSTEM\CurrentControlSet\Control\FileSystem\LongPathsEnabled* con dos valores: 1 (permite rutas de más de 260 caracteres) y 0 (no lo permite).

4. Tras la instalación, en línea de comandos ejecute: *pip install matplotlib*
   El instalador de paquetes de Python descargará e instalará las librerías que nos permitirán mostrar gráficamente las curvas a las que se ajustan los órdenes de complejidad de los algoritmos a analizar. La siguiente imagen muestra un ejemplo en Windows.



5. No incluir Python en el PATH o no permitir rutas de más de 260 caracteres en el PATH puede llevarle, según haya hecho la instalación, a que Eclipse no encuentre el intérprete de Python, necesario para el desarrollo de la práctica.

6. Según el sistema operativo, puede ser necesario reemplazar la instrucción *Plot plt = Plot.create()* en los métodos *show* y *showCombined* de la clase *MatPlotLib* por: *Plot plt = Plot.create(PythonConfig.pythonBinPathConfig ("PATH de PYTHON"))*.

   Ahora bien, tenga en cuenta que realizará la defensa en el ordenador del laboratorio, donde dicho cambio no será necesario, usando la versión original de *MatPlotLib* del repositorio.

Metodología: Pasos para medir el tiempo de ejecución de un algoritmo

En cada ejercicio seguiremos los siguientes pasos:

1. Generar para cada algoritmo un fichero de pares (tamaño, tiempo) con el tiempo medio por tamaño.

2. Obtener para cada algoritmo la curva que se ajusta al T(n) según los datos obtenidos (*curve fitting*)

3. Obtener una comparativa gráfica a partir de los ficheros anteriores.

Paso 1: Obtener los ficheros de pares

En este primer paso crearemos un fichero para cada algoritmo al cual queramos analizarle empíricamente su orden de complejidad. Cada fichero tendrá por líneas pares *X,Y* donde *X* (Integer) es el tamaño del problema e *Y* (Long) es el tiempo de ejecución empleado por el algoritmo para resolver un problema de dicho tamaño. Cada fichero tendrá por tanto el formato que muestra la imagen siguiente:

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

El paquete *us.lsi.curvefitting* proporciona la clase *GenData* para generar y almacenar en un fichero de texto los datos en el formato anterior, donde los tamaños formarán una progresión aritmética o geométrica. Según el orden de complejidad del algoritmo, deberemos decidir el valor para el tamaño máximo, así como entre un tipo de progresión u otro y su razón para que: 1) el número de valores generados sea suficiente y 2) el tiempo en la generación del fichero no sea excesivo. La clase *GenData* proporciona los siguientes métodos para los dos tipos de progresión:

```
GenData.tiemposEjecucionAritmetica(ft, file, min, max, razon, a, b);
GenData.tiemposEjecucionGeometrica(ft, file, min, max, razon, a, b);
```

En ambos métodos, el primer parámetro `ft` es una función que a partir de un tamaño *X* (Integer) devuelve el tiempo de ejecución *Y* (Long) empleado por un algoritmo en resolver un problema de dicho tamaño. Dicha función puede crearse de dos formas:

1. Cuando en el algoritmo a medir no sea necesario generar datos o llamar a otros algoritmos previamente para que éste pueda llevarse a cabo:
   ```
   Consumer<Integer> alg = …
   Function<Integer,Long> ft = GenData.time(alg);
   ```

2. En el caso contrario:
   ```
   Consumer<Integer> prep = …
   Consumer<Integer> alg = …
   Function<Integer,Long> ft = GenData.time(prep, alg);
   ```

Los detalles sobre cómo usar los métodos anteriores, así como el significado de sus parámetros se explicarán en clase de prácticas.

Paso 2: Ajuste Individual

Una vez hayamos obtenido para un algoritmo el fichero correspondiente en el formato anterior, ajustaremos la curva que mejor indique su orden de complejidad en base a los datos del fichero.

El paquete *us.lsi.curvefitting* proporciona las clases *Polynomial*, *Exponential* y *PowerLog* para hacer ajustes mediante funciones de orden polinómico, exponencial o polinómico/logarítmico respectivamente. Por tanto:

- Si el orden de complejidad es exponencial, el ajuste debe hacerse mediante *Exponential.of()*

- Si el orden de complejidad es polinómico, el ajuste puede hacerse mediante *Polynomial.of(n)*, donde *n* indica el grado del polinomio.

- Si el orden de complejidad es polinómico o logarítmico, el mejor ajuste lo obtendremos mediante *PowerLog.of(ls)*, donde *ls* es una lista de pares <índice, valor> para *a,b,c* y *d* en la función: $a\ n^b\ (ln\ n)^c + d$.

  Adicionalmente, *PowerLog* proporciona otro método factoría para restringir el valor de uno de los parámetros de la función $a\ n^b\ (ln\ n)^c + d$ dentro de un intervalo.

Es objetivo de esta práctica simplificar la expresión de la función de ajuste, por lo que todo orden de complejidad no exponencial debe ajustarse mediante uno de los dos métodos factoría con parámetros de *PowerLog*. Los detalles sobre cómo realizarlo se explicarán en clase de prácticas.

Si no estamos seguros del orden de complejidad de un algoritmo, *PowerLog* proporciona otro método factoría sin parámetros que puede ayudarnos antes del ajuste final. El siguiente código muestra un ejemplo:

```
List<WeightedObservedPoint> datos = DataFile.points(file);
Fit ajuste = PowerLog.of();
ajuste.fit(datos);
MatPlotLib.show(file, ajuste.getFunction(), ajuste.getExpression());
```

Cuando no es posible realizar un ajuste (sea cual sea el método factoría utilizado entre los anteriormente mencionados), Java lanza la siguiente excepción:

> *ConvergenceException: illegal state: unable to perform Q.R decomposition on the NxM jacobian matrix.*

Por otro lado, cuando los datos son insuficientes o las mediciones no son correctas, las nubes de puntos generadas presentan numerosos y acentuados máximos y mínimos locales (picos). Puesto que el T(n) debe ser una función monótona creciente, en tales casos será necesario modificar, bien el *warmup*, bien el número de mediciones, bien ambos.

Paso 3: Comparativa conjunta.

En cada ejercicio compararemos los tiempos de los distintos algoritmos de la forma:

```
MatPlotLib.showCombined("Tiempos", ls1, ls2);
```

Donde *ls1* es una lista de cadenas con las rutas de los ficheros obtenidos para varios algoritmos de un mismo ejercicio y *ls2* es una lista de cadenas que dan nombre a cada una de las curvas asociadas a los mismos.

La comparativa de tiempos nos ayudará al ajuste de cada curva tras generar todos los ficheros asociados a los distintos algoritmos (soluciones) de un mismo ejercicio.