Práctica 1

Antonio J. Galán Herrera

Ingeniería Informática (A)

Nota máxima alcanzada en SIETTE: 86 / 100.

Análisis de la Complejidad

Analizar la complejidad de un algoritmo consiste en determinar mediante algún procedimiento el orden de complejidad al que pertenece dicho algoritmo.

⚠️ Cabe destacar que **no es posible desarrollar un analizador que obtenga el resultado correcto para cualquier algoritmo existente** -y menos aún aplicando la restricción del enunciado de realizarlo como máximo en 10 segundos-, por lo que **el éxito en esta práctica dependerá de hacer un analizador que sea capaz de indicar correctamente las complejidades de los algoritmos de la prueba** de SIETTE.

Resolución

💡 La premisa de mi procedimiento consiste en tratar de *solapar* la **gráfica** que describe el **algoritmo** a analizar, con las **gráficas** que describen los **órdenes de complejidad**, y observar la complejidad que presente más **similitud**.

Ejemplo: *Si recibo un algoritmo cuya complejidad es exponencial, su evolución en el tiempo debería coincidir -aproximadamente- con una gráfica exponencial.*

Para llevar a cabo mi idea, he seguido los siguientes pasos:

1. Se dispone de un conjunto de valores n cada vez más grandes y más alejados entre sí.
    - Esto es así para que funcione mejor en general. Si los valores n crecen muy poco, no se observarán bien algunas complejidades.
2. Se crea un mapa cuyas claves son las cadenas que definen los órdenes de complejidad (resultados de la prueba) y para cada una, su valor es una serie de tiempos obtenidos al ejecutar una fórmula matemática que simula dicha complejidad.
    - Esto se usará más tarde para comparar los tiempos del algoritmo real, dividiendo cada tiempo entre el valor real; sabiendo que en una división si el numerador y el denominador son iguales, el resultado será 1; por tanto, se buscará la complejidad cuya proporción sea lo más cercana posible a 1.
3. Se ejecuta el algoritmo para todos los n que sea posible en 5 segundos, el resto del tiempo se usará para calcular las simulaciones de las complejidades.
4. Finalmente, se comparan los tiempos del algoritmo real con el de cada simulación (elementos del mapa), y se devuelve la complejidad de la simulación más cercana.

Analizador.java

▼ `calcularTiempos( repeticiones , limite )`

Ejecuta el algoritmo para todas las n que pueda, ejecutándose tantas veces como indique `repeticiones` para un mismo n_i, y mientras que no se supere el `limite` de tiempo.

1. Se crea una lista L en la que se almacenarán los tiempos de ejecución del algoritmo.
2. Para cada valor de n:
   - Se ejecuta el algoritmo `repeticiones` veces, **con el fin de obtener un promedio**.
   - Si en algún momento el `limite` de tiempo impuesto se supera, la ejecución se detiene y no se analizan más valores n.
3. El método termina devolviendo la lista L, donde $L_i = \text{algoritmo}(n_i)$.

▼ `ejecutar(n)`

Mide el **tiempo de ejecución del algoritmo** para un valor n usando la clase `Temporizador`.

▼ `guardarTiempos(tiempos, tabla)`

Almacena en el mapa cada complejidad y sus tiempos adaptados usando `adaptarMedias()`.

▼ `adaptarMedias( complejidad , medias )`

Devuelve una lista M donde cada elemento resulta $M_i = \frac{\text{algoritmo}(n_i)}{\text{complejidad}(n_i)}$, donde:

- complejidad es una **función que simula** la `complejidad` recibida como argumento.
- algoritmo es el **tiempo medido anteriormente**, recibido como la lista `medias`.

El objetivo de la división es obtener una relación (número cercano a 1), que representaría cuánto se parecen el tiempo medido para $\text{algoritmo}(n_i)$ y el tiempo simulado para una complejidad con esa misma entrada n_i.

▼ `determinarComplejidad( complejidades )`

Comprueba qué complejidad presenta una tendencia más cercana a 1, calculando el ***ratio*** de los valores asociados a cada complejidad en el mapa `complejidades`.

1. Para cada complejidad de las claves del mapa `complejidades`:
   - Calcula la **distancia entre 1 y el *ratio* de los valores asociados** a dicha complejidad.
   - Si dicho valor es más próximo a 1 que el mejor, **se sustituye**.
2. Finalmente, se devuelve la complejidad asociada al mejor valor.

El *ratio* se calcula como $\text{ratio} = \frac{\text{complejidad}(n_k)}{\text{complejidad}(n_{k-1})}$, donde k es el tamaño de la lista M de esa complejidad.

Datos.java

Esta clase es una pequeña estructura de datos que almacena:

- Una lista de tiempos (la asociada a cada complejidad), denominado antes como M.
- Un *ratio* que simboliza la relación entre el último y el penúltimo valor de la lista anterior.
  - Este valor se calcula en el propio constructor de la clase.