# PRACTICA I: EFICIENCIA DE LOS ALGORITMOS

**Grupos A y B**

October 3, 2016

## Contents

# 1 Objetivo

El objetivo de estas sesiones es que el alumno/a comprenda la importancia de analizar la eficiencia de los algoritmos y se familiarice con las formas de llevarlo a cabo. Para ello se mostrará como realizar un estudio teórico y empírico de un algoritmo.

# 2 Cálculo del tiempo teórico

A partir de la expresión del algoritmo, se aplicará las reglas conocidas para contar el número de operaciones que realiza un algoritmo. Este valor será expresado como una función de $T(n)$ que dará el número de operaciones requeridas para un caso concreto del problema caracterizado por tener un tamaño n.

El análisis que nos interesa será el del peor caso. Así, tras obtener la expresión analítica de $T(n)$, calcularemos el orden de eficiencia del algoritmo empleando la notación $O(\cdot)$.

## 2.1 Ejemplo 1: Algoritmo de Ordenación Burbuja

Vamos a obtener la eficiencia teórica del algoritmo de ordenación burbuja. Para ello vamos a considerar el siguiente código que implementa la ordenación de un vector de enteros mediante el método burbuja, desde la posición inicial a final.

```
1  void burbuja(vector<int> & T, int inicial, int final)
2  {
3   int i, j;
4   int aux;
5   for (i = inicial; i < final - 1; i++)
6     for (j = final - 1; j > i; j--)
7         if (T[j] < T[j-1])
8           {
9             aux = T[j];
10            T[j] = T[j-1];
11            T[j-1] = aux;
12          }
13 }
```

La mayor parte del tiempo de ejecución se emplea en el cuerpo del bucle interno donde se realiza el intercambio de elementos en el vector. Esta porción de código lo podemos acotar por una constante $a$. Por lo tanto el bucle for $j$ (lineas 6-12) se ejecuta exactamente un número de veces igual a final-i-1. A su vez el bucle interno se ejecuta una serie de veces indicado por el bucle externo. En definitiva tendríamos una fórmula como la siguiente:

$$\sum_{i=inicial}^{final-2} \sum_{j=i+1}^{final-1} a \tag{1}$$

Renombrando en la ecuación (1) $final$ como $n$ e $inicial$ como 1, pasemos a resolver la siguiente ecuación:

$$\sum_{i=1}^{n-2} \sum_{j=i+1}^{n-1} a \tag{2}$$

Realizando la sumatoria interior en (2) obtenemos:

$$\sum_{i=1}^{n-2} a(n-i-1) \tag{3}$$

Y finalmente tenemos:

$$\frac{a}{2}n^2 - \frac{3a}{2}n + a \tag{4}$$

Claramente $\frac{a}{2}n^2 - \frac{3a}{2}n + a \in O(n^2)$

Diremos por tanto que el método de ordenación es de orden $O(n^2)$ o cuadrático.

# 3 Cálculo de la eficiencia empírica

Se trata de llevar a cabo un estudio puramente empírico. Es decir, estudiar experimentalmente el comportamiento del algoritmo. Para ello mediremos los recursos empleados (tiempo) para cada tamaño dado de las entradas.

En el caso de los algoritmos de ordenación el tamaño viene dado por el número de componentes del vector a ordenar. En otro tipo de problemas, como es el caso del algoritmo para obtener el factorial de un número o la sucesión de fibonacci, la

eficiencia dependerá del valor del entero.

Para obtener el tiempo empírico de una ejecución de un algoritmo lo que vamos a hacer es utilizar la biblioteca chrono de la stl

`http://www.cplusplus.com/reference/chrono/?kw=chrono`,

que fue introducida en la revisión del estándar de c++, C++11. Por tanto algunos compiladores pueden no incluirla (en esta caso podemos utilizar la biblioteca ctime, aunque tiene menor precisión). Para utilizar dicha biblioteca es necesario incluirla en nuestro programa `#include<chrono>` e indicar al compilador que estamos utilizando dicho estándar a la hora de compilar, esto es

**g++ -std=c++0x -o ejecutable fuente.cpp**

o

**g++ -std=c++11 -o ejecutable fuente.cpp**

**chrono** es el nombre del fichero cabecera, pero también es el nombre de un espacio de nombres por lo que para poder hacer uso del mismo debemos explicitarlo en nuestro programa.

```
1 #include <chrono>
2
3 ....
4
5 using namespace std::chrono;
6
7 ....
```

Así para compilar el fichero de prueba que se entrega junto a las prácticas **BurbujaEficiencia.cpp** podemos hacer

**g++ -std=c++11 -o eficiencia BurbujaEficiencia.cpp**

En este caso, generaremos un ejecutable donde la llamada es

**./eficienia n**

donde $n$ representa el número de elementos del vector.

Veamos con más detalle cómo hemos utilizado la librería chrono para medir el tiempo de ejecución del algoritmo burbuja anterior. Para ello debemos declarar dos variables tantes y tdespues (en fichero BurbujaEficiencia.cpp)

```
1 #include <chrono>
```

```
#include <iostream>
#include <vector>
....

using namespace std::chrono;

int main(){

high_resolution_clock::time_point tantes,tdespues;
duration<double> tiempo_transcurrido;

 // Codigo cualquiera

tantes = high_resolution_clock::now(); //Captura el valor del
    reloj antes de la llamada a burbuja

  burbuja(V, 0, V.size());  // Llama al algoritmo de ordenación
     burbuja para ordenar los n elementos

tdespues = high_resolution_clock::now(); //Captura el valor del
    reloj antes de la llamada a burbuja


tiempo_transcurrido  = duration_cast<duration<double> >(tdespues
    - tantes);
cout << tama << " " << tiempo_transcurrido.count() << endl;
}
```

De esta forma, en la variable ***tantes*** capturamos el valor del reloj antes de la ejecución del algoritmo al que queremos medir el tiempo. La variable $tdespues$ contendrá el valor del reloj después de la ejecución del algoritmo en cuestión.

Para obtener el número de segundos o nanosegundos simplemente emitiremos un mensaje como éste:

$cout << std :: chrono :: duration\_cast < std :: chrono :: seconds > (tdespues - tantes).count() << "seg" << endl;$

o bien

$cout << std :: chrono :: duration\_cast < std :: chrono :: nanoseconds > (tdespues - tantes).count() << "ns" << endl;$

**OBSERVACIÓN:** Para casos muy pequeños, el tiempo medido es muy pequeño, por lo que el resultado será 0 segundos. Estos tiempos tan pequeños se pueden medir de forma indirecta ejecutando la sentencia que nos interesa repetidas veces y después dividiendo el tiempo total por el número de veces que se ha ejecutado. Por ejemplo:

```
...


high_resolution_clock::time_point tantes,tdespues;
duration<double> tiempo_transcurrido, tiempo_por_iteracion;
int num_iteraciones = 1000; // o cualquier valor deseado
 // Codigo cualquiera

tantes = high_resolution_clock::now(); //Captura el valor del
    reloj antes de la llamada a burbuja
for (int i=0;i< num_iteraciones; i++){
   //llamada al código de prueba para problemas de tamaño N
  }
tdespues = high_resolution_clock::now(); //Captura el valor del
    reloj antes de la llamada a burbuja


tiempo_transcurrido  = duration_cast<duration<double> >(tdespues
     - tantes);
tiempo_por_iteracion = tiempo_transcurrido/(double)
    num_iteraciones;

 cout << N << " " << tiempo_transcurrido.count() << endl;
```

Para obtener la eficiencia empírica deberemos de ejecutar el mismo algoritmo para diferentes ejemplos. Así para un algoritmo de ordenación lo ejecutaremos para diferentes tamaños del vector a ordenar y obtendremos el tiempo. Estos tiempos los almacenaremos en un fichero.

Podéis hacer uso de una macro, como la que se muestra a continuación, para

obtener el tiempo empírico de un algoritmo.

```
#!/bin/csh -vx
@ i = 10
echo ""> burbuja.dat
while ( $i < 10000 )
echo " $i ‘burbuja $i‘" >> burbuja.dat
@ i += 100
end
```

Así en esta macro se va a escribir en el fichero ***burbuja.dat*** el tiempo en segundos que tarda el algoritmo de ordenación en ordenar vectores de 10 a 10000 elementos. Las muestras se han tomado de 100 en 100. Para poder ejecutar esta macro debéis de darle a ésta permisos de ejecución. Por ejemplo, mediante la siguiente sentencia:

chmod +x macro

y a continuación ejecutar la macro como

./macro

**NOTA:** Cuando redactéis un informe sobre la eficiencia empírica de un algoritmo deberán aparecer todos los detalles relevantes al proceso de medida: tipo de ordenador utilizado, compilador empleado, opciones de compilación, etc.

**Otra alternativa** para obtener los tiempos es generar los distintos tamaños desde el propio programa, como se hace en el fichero **BurbujaEficiencia_Lims.cpp** que requiere como parámetros el tamaño mínimo, máximo así como el incremento que se produce en cada iteración.

# 4 Cálculo de la eficiencia híbrida.

El cálculo teórico del tiempo de ejecución de un algoritmo nos da mucha información. Es suficiente para comparar dos algoritmos cuando los suponemos aplicados a casos de tamaño arbitrariamente grande. Sin embargo, cuando se va a aplicar

el algoritmo en una situación concreta, es decir, especificadas la implementación, el compilador utilizado, el ordenador sobre el que se ejecuta, etc., nos interesa conocer de la forma más exacta posible la ecuación del tiempo.

Así, el cálculo teórico nos da la expresión general, pero asociada a cada término de esta expresión aparece una constante de valor desconocido. Para describir completamente la ecuación del tiempo, necesitamos conocer el valor de esas constantes. La forma de averiguar estos valores es ajustar la función a un conjunto de puntos.

En nuestro caso, la función es la que resulta del cálculo teórico, el conjunto de puntos lo forman los resultados del análisis empírico y para el ajuste emplearemos regresión por mínimos cuadrados. Por ejemplo en el algoritmo de ordenación la función que vamos a ajustar a los puntos obtenidos en el cálculo de la eficiencia empírica será:

$$T(n) = a_o * x^2 + a_1 * x + a_2$$

Al ajustar a los puntos obtenidos por mínimos cuadrados obtendremos los valores de $a_0$, $a_1$y $a_2$ es decir las constantes ocultas.

## 4.1 Cómo obtener las constantes ocultas

Gnuplot puede ser utilizado de forma interactiva para poder realizar el análisis de eficiencia de los algoritmos. En este guión ilustraremos de forma breve cómo se puede para resolver los problemas planteados en la práctica de eficiencia.

# 5 Visualizando datos con gnuplot

En nuestro caso, partimos de un conjunto de datos, por ejemplo `'ordenBeje.dat'` que contiene en cada fila pares de elementos (x,y) separados por espacios en blanco. El primer elemento del par se corresponde con el tamaño y el segundo elemento se corresponde con el tiempo.

```
100 0
5100 0.53
```

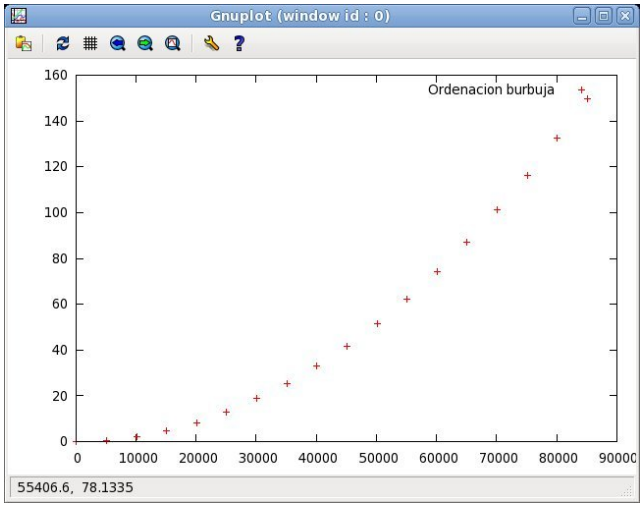


Figure 1: Fichero ordenBeje.dat

```
10100 2.12
15100 4.72
20100 8.39
25100 13.11
30100 18.73
35100 25.4
....
```

Desde línea de comandos hacemos una llamada a `gnuplot`, y para poder representar estos puntos podemos ejecutar el comando

```
gnuplot> plot 'ordenBeje.dat' title
        'Eficiencia Burbuja' with points
```

Como resultado aparecerá una nueva ventana como la de la figura 1. El comando plot indica que queremos representar el fichero `'ordenBeje.dat'`, with points indica que en la salida nos muestre los datos como un conjunto de puntos desconectados (hay otras opciones, como por ejemplo 'with lines'). Podemos dar un título significativo al gráfico (en nuestro caso de 'Eficiencia Burbuja'.)

Además podemos etiquetar los valores que representa el eje x y el eje y. Para ello podemos ejecutar los siguientes comandos:

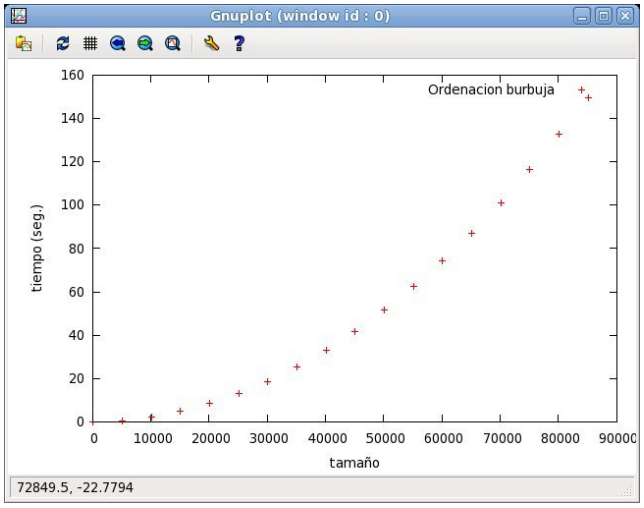


Figure 2: Fichero ordenBeje.dat etiquetado

```
gnuplot> set xlabel "Tamaño"
gnuplot> set ylabel "Tiempo (seg)"
```

Para volver a mostrar el gráfico, podemos utilizar el comando replot

```
replot
```

dando como resultado el gráfico de la figura 2

## 5.1 Regresión

Lo primero que tenemos que hacer es definir la función que queremos ajustar a los datos. En nuestro ejemplo, estamos hablando de una función cuadrática, pues hemos visto que el algoritmo tiene un orden de eficiencia $O(n^2)$. Podemos definir esta función en gnuplot mediante el siguiente comando

```
gnuplot> f(x) = a * x * x
```

El siguiente paso es indicarle a gnuplot que haga la regresión. Esto es simple, únicamente le tenemos que indicar

```
gnuplot> fit f(x) 'ordenBeje.dat' via a
```

Lo que se debe ver es el resultado, tras unas pocas iteraciones, del proceso de regresión de la función a ajustar. El último paso debería ser algo como:

```
Iteration 5
 WSSR        : 0.514373          delta(WSSR)/WSSR   : -1.12237e-14
 delta(WSSR) : -5.77316e-15      limit for stopping : 1e-05
 lambda   : 33811.5

resultant parameter values

a               = 2.06318e-08

After 5 iterations the fit converged.
final sum of squares of residuals : 0.514373
rel. change during last iteration : -1.12237e-14

degrees of freedom    (FIT_NDF)                        : 17
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 0.173946
variance of residuals (reduced chisquare) = WSSR/ndf   : 0.0302572

Final set of parameters            Asymptotic Standard Error
=======================            ==========================

a               = 2.06318e-08      +/- 1.213e-11    (0.05877%)


correlation matrix of the fit parameters:

               a
a               1.000
```

Como vemos, hay una gran cantidad de información sobre los datos y sobre la bondad del ajuste. Probablemente, la parte que más nos importa es "¿Cuáles son los parámetros?". Sin embargo, éstos son fácilmente identificables bajo la sección `Final set of parameters`. En estos datos, la constante oculta es

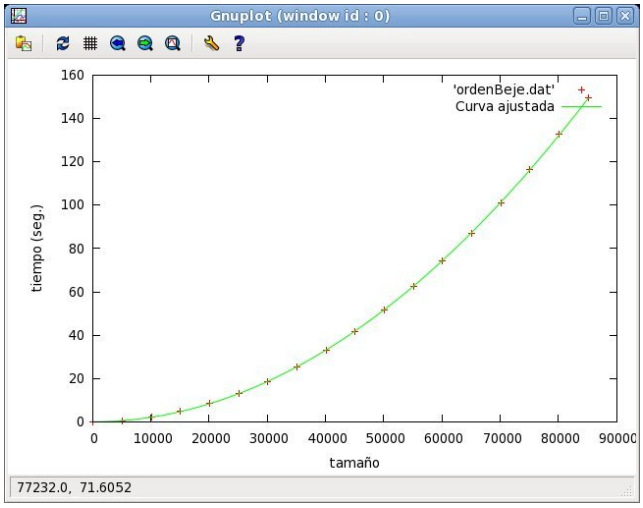


Figure 3: Enfoque híbrido: Ajuste de datos

2.06318e-08

La siguiente pregunta es, cómo se ajusta esta función a nuestros datos. Para ello, podemos dibujar ambos en un único gráfico mediante el comando siguiente. Como resultado tenemos el gráfico de la figura 3

```
gnuplot> plot 'ordenBeje.dat', f(x) title 'Curva ajustada'
```

## 5.2 Funciones más complejas

Podemos hacer las regresión con funciones más complejas, por ejemplo $f(x) = a * x^2 + b * log(x)$. Para ello es suficiente con definir

```
gnuplot> f(x) = a * x * x + b* log(x)
```

El siguiente paso es indicarle a gnuplot que haga la regresión. Esto es simple, únicamente le tenemos que indicar

```
gnuplot> fit f(x) 'ordenBeje.dat' via a,b
```

## 5.3 Salvar los resultados

Pasamos a describir cómo salvar estos gráficos en un fichero. Ya que gnuplot fue diseñado hace bastante tiempo, la forma en la que gnuplot exporta sus archivos puede parecer arcaica. Para ello, tenemos que dar tres pasos básicos. La sintaxis

puede variar con el formato del archivo que queremos guardar. Lo ilustraremos con un formato png, por estar ampliamente soportado por otras herramientas. En concreto, debemos de ejecutar los siguientes comandos

```
gnuplot> set terminal png
gnuplot> set output 'salida.png'
gnuplot> replot
```

El primer comando indica el formato que le debemos dar a la salida (si queremos ver una lista de formatos disponibles podemos indicar `set terminal`). El segundo comando nos indica el nombre el fichero donde queremos almacenar los datos. Finalmente, al volver a dibujar con replot, mandamos la salida a nuestro fichero de datos. Para volver a tener una salida en pantalla vale con hacer `set terminal x11`.

# 6 Tareas a realizar.

En la página de la asignatura se encuentran los siguientes ficheros:

- lema.txt: Fichero con distintas palabras del castellano.
- quijote.txt: Un fichero de texto con el contenido del Quijote.
- xxxx.cpp: Ficheros de código con distintos algoritmos y ejemplos de cómo se puede calcular la eficiencia de los mismos.
  - Contar ocurrecias de una palabra
  - Algoritmo de ordenación burbuja
  - Contar frecuencias de aparición de una palabra en un documento. Para este último caso, se disponen de distintas versiones. La idea es que el alumno pueda comprender las ventajas que aporta el uso de distintas estructuras de datos (a nivel de eficiencia) para la resolución de un problema.

El alumno deberá ejecutar los distintos programas, generando como salida un fichero con los tiempos de ejecución de cada algoritmo, en función del tamaño

de las entradas. Una vez que tengamos los tiempos se debe proceder al análisis híbrido de los mismos como se indica en la práctica.

## 6.1 Ejemplo 1: Algoritmo de ordenación por burbuja.

Pasos:

1. Compilar el fichero de código.

   ```
   g++ -o ordenacion ordenacion.cpp
   ```

2. Ejecutar el algoritmo de ordenación, recogiendo la salida del programa en un fichero llamado ordenacionB.datos. En cada iteración se ordena un subconjunto del diccionario de palabras, variando el tamaño del subconjunto a ordenar.

   ```
   ./ordenacion > ordenacionB.dat
   ```

   Como resultado obtendremos un fichero con una salida parecida a

   ```
   100 0
   5100 0.53
   10100 2.12
   15100 4.72
   20100 8.39
   25100 13.11
   30100 18.73
   35100 25.4
   ....
   ```

   donde la primera columna representa el tamaño del conjunto de palabras y la segunda el tiempo necesario para ordenarlas.

3. Realizar el análisis híbrido. En esta caso, podemos realizar una regresión cuadrática, que utilizando los datos del fichero `ordenB.eje.dat` nos da los siguientes valores.

   $T(n) = 2.081e^{-8}n^2 - 1.4924e^{-5}n + 0.15918$

## 6.2 Se pide:

1. Realizar el análisis de eficiencia cuando consideramos el código en `ocurrencias.cpp`. En este caso, debemos de modificar el código para asegurarnos que tenemos la misma salida que en el ejemplo del algoritmo burbuja (tamaño y tiempo).

   NOTA: En el fichero se ilustra cómo se puede realizar la toma de tiempos cuando la precisión del reloj no es suficiente.

2. Análisis de eficiencia del código en `frecuencias.cpp`, considerando como entradas el texto del fichero `quijote.txt`.

3. Realizar el análisis de eficiencia teórico y práctico con los algoritmos de ordenación que se conocen (burbuja, inserción y selección). El alumno deberá implementar los distintos algoritmos de ordenación de forma que se permitan obtener los tiempos para distintos tamaños de entrada.

   Es importante destacar que el comportamiento de los distintos algoritmos depende del orden de las entradas, por tanto nos debemos asegurar que en cada llamada al algoritmo se parte del conjunto de datos. Para ello, en lugar de ordenar como se indica en el siguiente código

```
1 for (int tama = inf ; tama < sup ; tama+= incremento){
2   // tiempo de inicio
3   ordenar(V, 0,tama);
4   // tiempo de fin;
5   }
```

   que como efecto, en cada iteración el vector V se modifica teniendo las posiciones desde 0 hasta tama ordenadas, debemos de ordenar sobre una copia del vector como por ejemplo

```
1 for (int tama = inf ; tama < sup ; tama+= incremento){
2   aux = V;    // Copiamos el vector original que no se
        modifica
3   // tiempo de inicio
4   ordenar(aux, 0,tama);
5   // tiempo de fin;
6   }
```

Para el análisis de eficiencia se utilizarán dos ficheros:

- `lema.txt` Donde los elementos están casi ordernados
- `quijote.txt` Donde los elementos se distribuyen de forma mas aleatoria

El estudiante deberá presentar un análisis que refleje el comportamiento de los algoritmos para cada fichero de datos así como realizar una comparación conjunta de la eficienia de los distintos algoritmos. Analizar qué está ocurriendo.

## 6.3 Fecha de entrega

Día; Hora; Grupo
16/10/2015 23:50 Grupo 1
16/10/2015 23:50 Grupo 2
16/10/2015 23:50 Grupo 3