Análisis de eficiencia de algoritmos

Práctica 1 de Algorítmica

Grado en Informática
Departamento de Ciencias de la Computación e I. A.
E.T.S.I. Informática y de Telecomunicaciones
Universidad de Granada

Curso 2016 - 2017

ugr Universidad de Granada

DECSAI

Cálculo de la eficiencia teórica

Ejemplo 1: Algoritmo de ordenación Burbuja

```
1 void burbuja(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 }
```

- El tiempo de ejecución del cuerpo del bucle interno (líneas de 7-12) lo podemos acotar por una constante a.
- Esas líneas de 7-12 se ejecutan $(final - 1) - (i + 1) + 1$ veces.
- A su vez el bucle interno se ejecuta un número de veces indicado por el bucle externo, $(final - 2) - inicial + 1$.

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

Renombrando *final* como n e *inicial* como 1, pasamos a resolver la siguiente ecuación:

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

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

Ejemplo 2: Algoritmo de ordenación Mergesort

- Este algoritmo divide el vector en dos partes iguales y se vuelve a aplicar de forma recurrente a cada una de ellas.
- Una vez hecho esto, fusiona los dos vectores sobre el vector original, de manera que esta parte ya queda ordenada.
- Si el número de elementos del vector que se está tratando en cada momento de la recursión es menor que una constante *UMBRALMS*, entonces se ordenará mediante el algoritmo burbuja.

```
void mergesort (int T[], int inicial, int final)
 if (final - inicial < UMBRALMS)
    burbuja (T, inicial, final);
else {
      int k = (final - inicial)/2;
      int * U = new int [k - inicial + 1];
      for (l = 0, l2 = inicial; l < k; l++, l2++)
          U[l] = T[l2];
      U[l] = INT_MAX;
      int * V = new int [final - k + 1];
      for (l = 0, l2 = k; l < final-k; l++, l2++)
          V[l] = T[l2];
      V[l] = INT_MAX;
      mergesort(U, 0, k);
      mergesort(V, 0, final - k);
      fusion(T, inicial, final, U, V);  }
```

```
void fusion (int T[],int inicial,int final,int U[],
{
 int j = 0 ;
 int k = 0 ;

for (int i = inicial; i < final; i++)
    if (U[j] < V[k]) {
       T[i] = U[j];
       j++;
    }
   else {
         T[i] = V[k];
         k++;
   }
 }
```

- Hasta llegar a las llamadas recursivas, el algoritmo emplea un tiempo $O(n)$.
- Después se hacen dos llamadas recursivas para un tamaño $n/2$.
- El procedimiento fusion también es de orden $O(n)$.
- Por tanto la ecuación es

$$T(n) = \begin{cases} 2T(\frac{n}{2}) + n & \text{si } n \geq UMBRALMS \\ n^2 & \text{si } n < UMBRALMS \end{cases} \quad (1)$$

- La solución, por ejemplo por el método de la ecuación característica es:

$$T(n) = c_1 n + c_2 n \log_2(n) \in O(n \log_2(n))$$

Cálculo de la eficiencia empírica

- Estudio puramente empírico del comportamiento de los algoritmos.
- Para ello mediremos los recursos empleados (tiempo) para cada tamaño dado de las entradas.
  - Algoritmos de ordenación: número de componentes del vector a ordenar.
  - Torres de Hanoi: valor del entero que representa el número de discos.
  - Algoritmo de Floyd (calcula los caminos mínimos entre todos los pares de nodos en un grafo dirigido): número de nodos del grafo.

- Para obtener el tiempo empírico de una ejecución de un algoritmo se definen en el código dos variables:

```
clock_t tantes;
clock_t tdespues;
```

- 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.

- Por ejemplo:

```
tantes = clock();
burbuja(T,0,n);
tdespues = clock();
```

- Para obtener el número de segundos, se obtiene la diferencia entre los dos instantes y se pasa a segundos mediante la constante `CLOCKS_PER_SEC`:

```
cout << (double)(tdespues - tantes) /
CLOCKS_PER_SEC << endl;
```

- Para hacer uso de estas sentencias hay que usar la biblioteca `ctime`.

- En casos donde, por ejemplo, la cantidad de elementos a ordenar es muy pequeña, el tiempo medido será muy pequeño y por lo tanto el resultado será 0 segundos.
- Estos tiempos tan pequeños se pueden medir de forma indirecta ejecutando la sentencia que nos interesa muchas veces y después dividiendo el tiempo total por el número de veces que se ha ejecutado. Por ejemplo:

```
double tiempo_transcurrido;
const int NUM_VECES=10000;
tantes=clock();
for (int i=0; i<NUM_VECES;i++)
//Sentencia cuyo tiempo se pretende medir
tdespues = clock();
tiempo_transcurrido=((double)(tdespues-tantes)/
(CLOCKS_PER_SEC* (double)NUM_VECES));
```

Otra forma más precisa de medir el tiempo:

```
#include <chrono>
using namespace std::chrono;
high_resolution_clock::time_point tantes,
tdespues;
duration<double> transcurrido;
tantes = high_resolution_clock::now();
//sentencia o programa a medir
tdespues = high_resolution_clock::now();
transcurrido =
duration_cast<duration<double>>(tdespues -
tantes);
cout << ''el tiempo empleado es '' <<
transcurrido.count() << '' segundos.''  <<
endl;
```

En este caso al compilar se debe incluir la directiva

```
-std=gnu++0x
```

- Para realizar un análisis de eficiencia empírica debemos ejecutar el mismo algoritmo para diferentes tamaños de entrada.
- Obtenemos el tiempo (preferiblemente varias veces para cada tamaño, en cuyo caso obtendremos el tiempo medio o el tiempo máximo por tamaño).
- Estos tiempos se almacenan en un fichero.
- Se puede usar una macro externa o programarlo dentro del código

```
#!/bin/csh -vx
echo "" >> salida.dat
@ i = 1000
while ( $i < 100000 )
./mergeSort $i >> salida.dat
@ i += 1000
end
```

Mostrar resultados

- Haremos uso de tablas que recojan el tiempo invertido para cada caso y también de gráficas.
- Para mostrar los datos en gráfica pondremos en el eje X (abscisa) el tamaño de los casos y en el eje Y (ordenada) el tiempo, medido en segundos, requerido por la implementación del algoritmo.
- Para hacer esta representación de los resultados podemos hacer uso, bien del software grace (xmgrace, `http://plasma-gate.weizmann.ac.il/Grace/`) o de gnuplot (entre otras opciones).

Mostrar resultados con Gnuplot

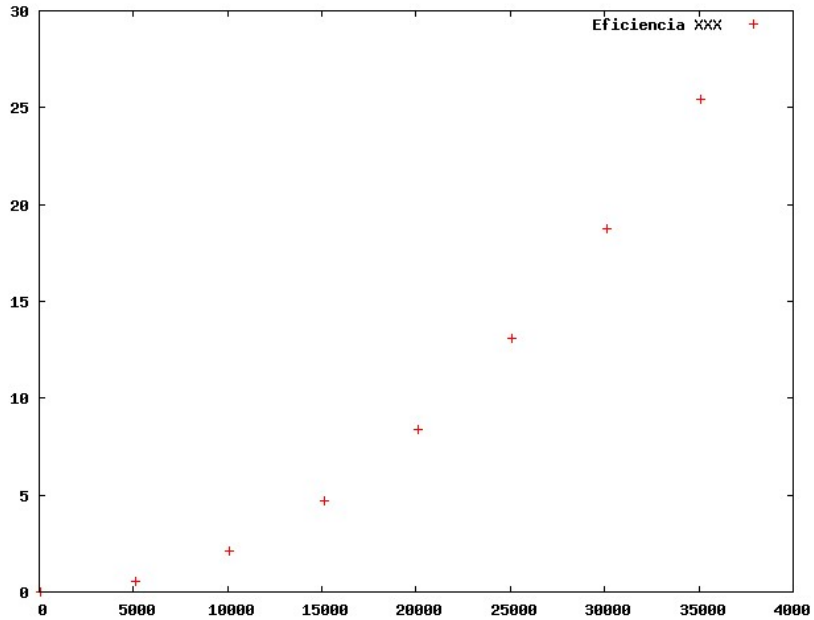
- Partimos de un conjunto de datos, por ejemplo `salida.dat` que contiene en cada fila pares de elementos (x, y) separados por espacios en blanco.
- El primer elemento del par se corresponde con el tamaño del problema y el segundo elemento se corresponde con el tiempo.

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

Mostrar resultados con Gnuplot

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

```
gnuplot> plot 'salida.dat' title
'Eficiencia XXX' with points
```

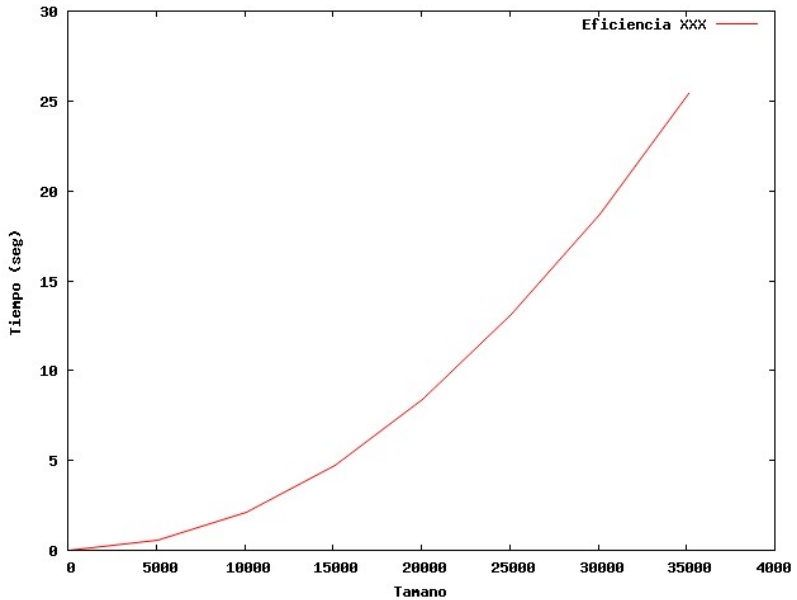


Mostrar resultados con Gnuplot

- Podemos etiquetar los valores que representa el eje x y el eje y:

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

- Para volver a mostrar el gráfico, podemos utilizar el comando `replot` (usando ahora la opción `with lines`).



Mostrar resultados con Gnuplot

- Si se desea que el gráfico se guarde en un fichero de cierto tipo, por ejemplo pdf o postscript (si queremos ver una lista de formatos disponibles podemos teclear `set terminal`), se puede emplear

```
gnuplot> set terminal postscript (o pdf)
gnuplot> set output "fichero.ps" (o
"fichero.pdf")
```

Para hacerse una idea de la potencia y calidad de los gráficos que puede generar gnuplot, se puede consultar la página web `http://gnuplot.sourceforge.net/demo/`.

Cálculo de la eficiencia híbrida

- El cálculo teórico del tiempo de ejecución de un algoritmo nos da mucha información, suficiente para comparar dos algoritmos cuando los suponemos aplicados a casos de tamaño arbitrariamente grande.
- 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 burbuja la función a ajustar a los puntos obtenidos en el cálculo de la eficiencia empírica será:

$$T(n) = a_0 \times n^2 + a_1 \times n + a_2$$

aunque también podríamos usar directamente

$$T(n) = a_0 \times n^2$$

- 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.
- De esta manera, luego podremos saber cuánto tiempo aproximadamente utilizará el algoritmo para cualquier entrada de tamaño n.

Cómo obtener las constantes ocultas

Al igual que para el cálculo de la eficiencia empírica, podemos utilizar (entre otros) grace o gnuplot. Ilustraremos el uso de gnuplot para esta tarea.

- Primero hay definir la función que queremos ajustar a los datos. Podemos definir esta función en gnuplot mediante el siguiente comando:

```
gnuplot> f(x) = a0*x*x+a1*x+a2
```

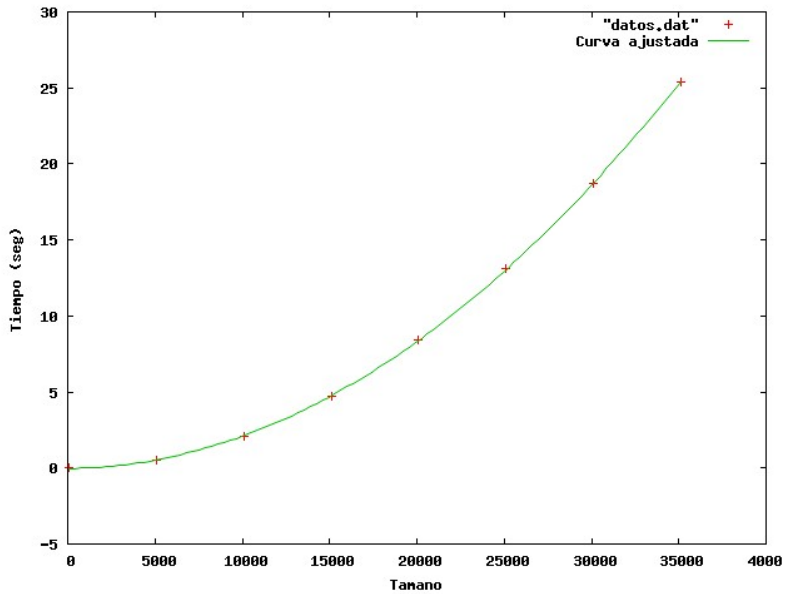
- Luego hay que indicarle a gnuplot que haga la regresión. Únicamente le tenemos que indicar

```
gnuplot> fit f(x) 'salida.dat' via a0,a1,a2
```

```
 Iteration 12
 WSSR        : 0.00707381        delta(WSSR)/WSSR   : -8.88702e-07
 delta(WSSR) : -6.28651e-09      limit for stopping : 1e-05
 lambda   : 0.000351302

resultant parameter values

a0              = 2.04333e-08
a1              = 7.98476e-06
a2              = -0.0268361

After 12 iterations the fit converged.
final sum of squares of residuals : 0.00707381
rel. change during last iteration : -8.88702e-07

degrees of freedom    (FIT_NDF)                        : 5
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 0.0376133
variance of residuals (reduced chisquare) = WSSR/ndf   : 0.00141476

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

a0              = 2.04333e-08      +/- 1.161e-10     (0.5681%)
a1              = 7.98476e-06      +/- 4.248e-06     (53.2%)
a2              = -0.0268361       +/- 0.03199       (119.2%)


correlation matrix of the fit parameters:
               a0     a1     a2
a0              1.000
a1             -0.962  1.000
a2              0.648 -0.798  1.000
```

- Para ver cómo se ajusta esta función a nuestros datos, podemos dibujar ambos en un único gráfico mediante:

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



- Por ejemplo, podemos estimar usando esa función ajustada que el tiempo de ejecución del algoritmo para una entrada de tamaño $n = 40000$ sería de $f(40000) = 32.98$ segundos.

Tareas a realizar

Aplicar las pautas y conceptos descritos anteriormente para el análisis de un conjunto de 8 algoritmos.

Algoritmo	Eficiencia
burbuja	$O(n^2)$
inserción	$O(n^2)$
seleccion	$O(n^2)$
mergesort	$O(n \log n)$
quicksort	$O(n \log n)$
heapsort	$O(n \log n)$
floyd	$O(n^3)$
hanoi	$O(2^n)$

`burbuja`, `insercion`, `seleccion`, `mergesort`, `quicksort` y `heapsort` son algoritmos de ordenación de vectores; `floyd` es un algoritmo que calcula el costo del camino mínimo entre cada par de nodos de un grafo dirigido; y `hanoi` es un algoritmo para resolver el famoso problema de las torres de Hanoi.

Los códigos de los algoritmos están disponibles en la plataforma de docencia de la asignatura (se pueden modificar ligeramente para que el posterior procesamiento de los datos sea más automático).

1. Calcule la eficiencia empírica. Defina adecuadamente los tamaños de entrada de forma tal que se generen al menos 25 datos. Incluya en la memoria tablas diferentes para los algoritmos de distinto orden de eficiencia (una con los algoritmos de orden $O(n^2)$, otra con los $O(n \log n)$, otra con $O(n^3)$ y otra con $O(2^n)$).
2. Con cada una de las tablas anteriores, genere un gráfico comparando los tiempos de los algoritmos. Para los algoritmos que realizan la misma tarea (los de ordenación), incluya también una gráfica con todos ellos, para poder apreciar las diferencias en rendimiento de algoritmos con diferente orden de eficiencia.

1. Calcule también la eficiencia híbrida de todos los algoritmos. Pruebe también con otros ajustes que no se correspondan con la eficiencia teórica (ajuste lineal, cuadrático, etc) y compruebe la variación en la calidad del ajuste.
2. Otro aspecto interesante a analizar mediante este tipo de estudio es la variación de la eficiencia empírica en función de parámetros externos tales como: las opciones de compilación utilizada (con/sin optimización), el ordenador donde se realizan las pruebas, el sistema operativo, etc. Sugiera algún estudio de este tipo y llévelo a cabo.
3. Escriba una memoria con todas las tareas realizadas, incluyendo todos los detalles relevantes referidos a los cálculos de eficiencia empírica e híbrida. El informe debe entregarse en formato pdf.