ESTRUCTURA DE DATOS

E.T.S. de Ingenierías Informática y de Telecomunicación

Documentación (Práctica 1)

Eficiencia

Grupo de Prácticas:

Marcos Avilés Luque → 100%

DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN E INTELIGENCIA ARTIFICIAL
UNIVERSIDAD DE GRANADA

Curso 2016-2017

Antes de entrar en detalles de la realización de la práctica debo comentar que los programas y ejecuciones las he realizado a través de una máquina virtual con limitación de 2 microprocesadores y poca memoria RAM, por lo que las pruebas y resultados obtenidos se pueden ver afectados.

1. PRIMERA CUESTIÓN.

Vamos a analizar el programa “ocurrencias.cpp” mediante los ficheros de entrada “Quijote.txt” y “Lema.txt” para el cálculo de la eficiencia empírica e híbrida.

1.1 Cálculo de tiempo teórico.

En primer lugar analizamos la eficiencia en tiempo del algoritmo principal, ya que el contenido restante del programa se encarga de leer el archivo de texto, declaración de variables, toma de reloj… que para nosotros diremos que tiene una eficiencia en tiempo constante y no es muy relevante para nosotros:

```
int contar_hasta( vector<string> & V, int ini, int fin, string & s) {
  int cuantos = 0;
  for (int i=ini; i< fin ; i++)
    if (V[i]==s) {
      cuantos ++;
    }
  return cuantos;
}
```

Esta función es la encargada contar ocurrencias de un String en el vector V, por lo que dependerá del tamaño del vector V, diremos que en el peor de los casos puede ser “n”. Por tanto:

T(n)= $\sum_{i=0}^{n} a$ => T(n)=a*n => O(n). Tiene un orden de eficiencia en tiempo lineal.

1.2 Cálculo de la eficiencia empírica.

Como dice el guión he utilizado la toma de tiempos promedio, ya que en este caso la precisión del reloj no era suficiente. Utilizando una batería de pruebas desde 10 hasta 100000 en intervalos de 100 en 100.

NOTA: Para ser un poco mas breve, junto los gráficos de los valores empíricos con la regresión de cada ejercicio, para evitar la extensión de esta documentación.

1.3 Cálculo de la eficiencia híbrida.

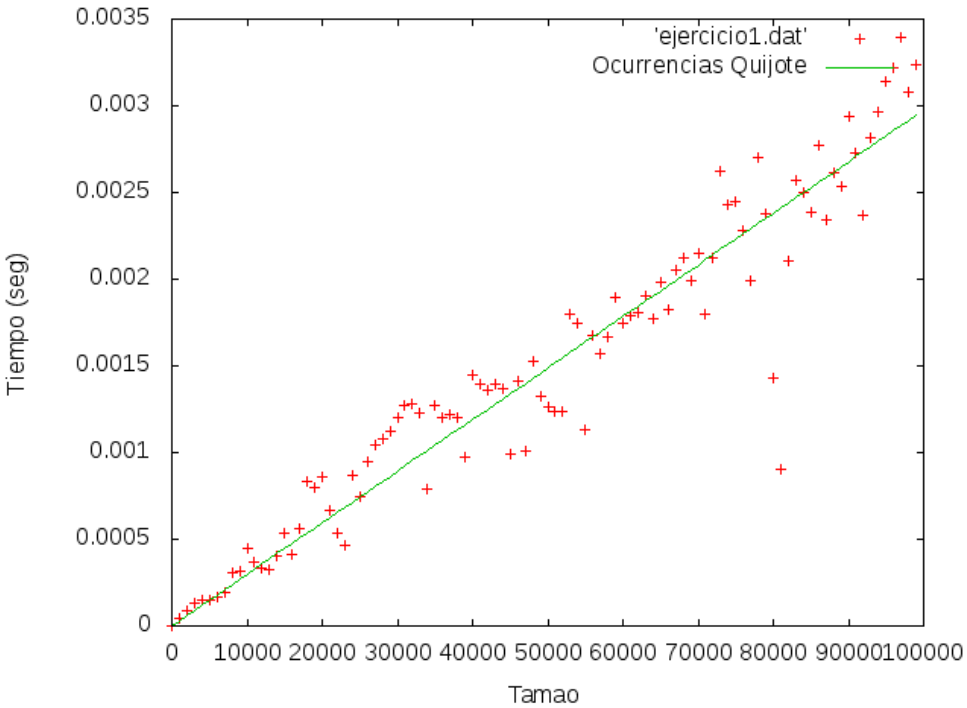
La función de regresión como he comentado en la parte teórica es:

f(x) = a*x

Muestro una captura de pantalla en el ajuste de regresión, tanto para “Quijote” como “Lema”, que podemos observar como el margen de error es un poco mas alto de lo habitual, debido a la toma de tiempos en la máquina virtual.

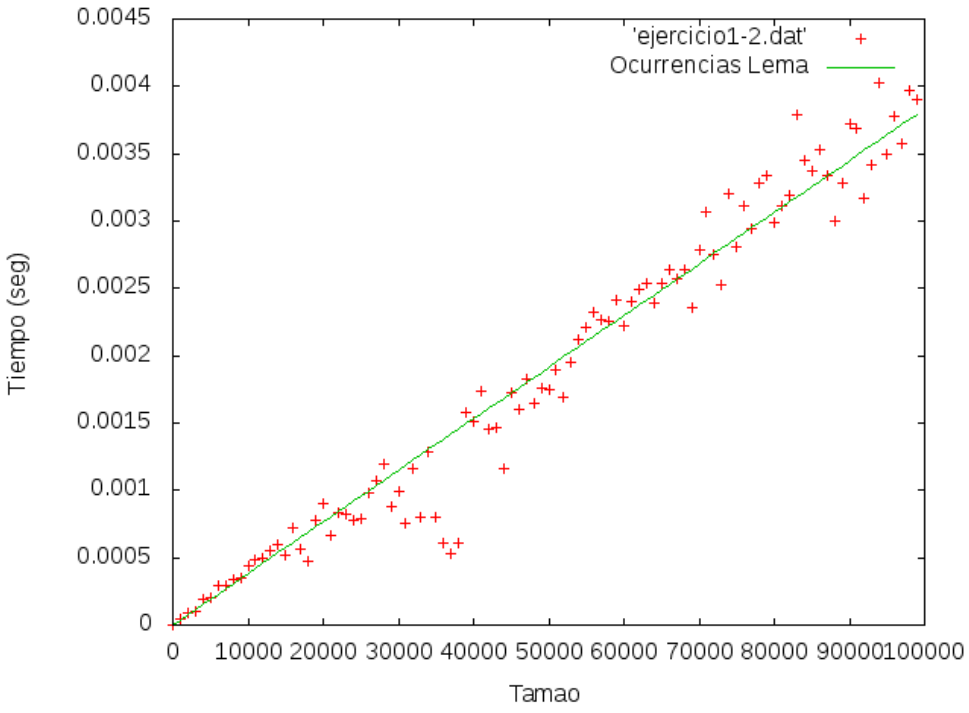
Quijote:

```
marcos@ubuntu: ~/Escritorio/3ºAño/1º Cuatrimestre/ED/practica1
a               = 2.97489e-08

After 3 iterations the fit converged.
final sum of squares of residuals : 7.11719e-06
rel. change during last iteration : -4.00276e-11

degrees of freedom    (FIT_NDF)                        : 99
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 0.000268125
variance of residuals (reduced chisquare) = WSSR/ndf   : 7.18908e-08

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

a               = 2.97489e-08      +/- 4.678e-10    (1.573%)


correlation matrix of the fit parameters:

               a
a               1.000
gnuplot>
```



Lema:

```
marcos@ubuntu: ~/Escritorio/3ºAño/1º Cuatrimestre/ED/practica1
a               = 3.83098e-08

After 3 iterations the fit converged.
final sum of squares of residuals : 5.67758e-06
rel. change during last iteration : -4.15566e-12

degrees of freedom    (FIT_NDF)                        : 99
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 0.000239477
variance of residuals (reduced chisquare) = WSSR/ndf   : 5.73493e-08

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

a               = 3.83098e-08      +/- 4.179e-10    (1.091%)


correlation matrix of the fit parameters:

               a
a               1.000
gnuplot>
```



2. SEGUNDA CUESTIÓN.

En este apartado vamos analizar el programa “frecuencias.cpp”, encargado de contar las frecuencias de aparición de una palabra en un libro. Consta de cuatro versiones (V1, V2, V3, V4).

2.1 Cálculo de tiempo teórico.

Como en la cuestión anterior, el programa vuelve a contar con la lectura del fichero “Quijote.txt”, declaración de variables, toma de tiempos...etc. Que para nosotros será de un orden de eficiencia en tiempo constante, y no será relevante para nosotros.

Nos quedamos con cada función importante:

V1:

```
void contar_frecuencias_V1( vector<string> & libro, int ini, int fin,
                            vector<string> &pal, vector<int> & frec ){
  int cuantas;
  for (int i = ini; i<fin; i++){

     cuantas = contar_hasta(libro,ini,fin,libro[i]);
     pal.push_back(libro[i]);
     frec.push_back(cuantas);
     }
}
```

En esta primera versión analizamos, una sumatoria que depende del tamaño del vector, pero como siempre, para nosotros en el peor de los casos puede ser “n”. Por lo que tenemos una sumatoria desde 0 hasta n, del máximo en eficiencia del interior del bucle.

Las inserciones son de orden constante por lo que vamos analizar la eficiencia de la función “contar_hasta”; la cuál hemos analizado en la primera cuestión y pertenece al orden de eficiencia lineal.

Por tanto tenemos:

$T(n)= \sum_{i=0}^{n} n$ => T(n)=n*n => $O(n^2)$. Tiene un orden de eficiencia en tiempo cuadrática.

V2:

```
void contar_frecuencias_V2( vector<string> & libro, int ini, int fin,
                            vector<string> &pal, vector<int> & frec ){
  int pos;
  for (int i = ini; i<fin; i++){

     pos = buscar(pal, libro[i]);
     if (pos==POS_NULA) {
         pal.push_back(libro[i]);    // Analisis amortizado O(1)
         frec.push_back(1);    // Analisis amortizado O(1)
         }
     else {
         frec[pos]++;
         }

     }
}
```

Un caso igual que el anterior, ya que todo el interior del bucle for, pertenece al orden de eficiencia constante, menos la función “buscar()”, por lo que vamos a estudiarla:

```
int buscar( vector<string> & V, string & s) {
  bool enc= false;
  int pos = POS_NULA;
  for (int i=0; i< V.size() && !enc; i++)
    if (V[i]==s) {
      enc = true;
      pos = i;
    }
  return pos;
}
```

Una función igual que las anteriores en eficiencia, depende del tamaño del vector, que contamos como tamaño “n”, y las sentencias interiores del bucle son de eficiencia constante.

Por lo tanto, obtenemos:

$T(n)=\sum_{i=0}^{n} n$ => $T(n)=n*n = n^2$ =>$O(n^2)$. Tiene un orden de eficiencia en tiempo cuadrática.

V3:

```
void contar_frecuencias_V3( vector<string> & libro, int ini, int fin,
                            vector<string> &pal, vector<int> & frec ){
  vector<string>::iterator pos;
  for (int i = ini; i<fin; i++){

     pos = lower_bound(pal.begin(), pal.end(), libro[i]);  // O(log (n) ), n tama del vector
Busqueda Binaria
     if ((pos==pal.end()) || (*pos!=libro[i])) {
         frec.insert(frec.begin() + (pos-pal.begin()), 1);   //O(n)
         pal.insert(pos, libro[i]);            //O (n)
         }
     else {
         frec[pos-pal.begin()]++; // O(1)
         }

     }
}
```

En esta nueva versión como se muestra en los comentarios de código, tenemos la sumatoria del bucle for.

Realizando el análisis desde abajo hacia arriba, tenemos en el “else” un orden de eficiencia constante, y en el “if” dos sentencias de orden n, por lo que para nosotros siempre para el peor de los casos, nos quedamos con orden n.

La función “lowe_bound()” tiene una eficiencia logaritmica de n, por lo que resumiendo, el máximo de las sentencias internas del bucle, la mayor es O(n). Por tanto obtenemos:

T(n)= $\sum_{i=0}^{n} max\left(\log(n), n, n, 1\right)$ => T(n)= $\sum_{i=0}^{n} n$ => T(n)=n*n=n^2. Tiene un

un orden eficiencia en tiempo cuadrática.

V4:

```
void contar_frecuencias_V4( vector<string> & libro, int ini, int fin,
                            vector<string> &pal, vector<int> & frec ){

  map<string,int> M;
  for (int i = ini; i<fin; i++)
     M[libro[i]]++;          // O( log(n) )

  map<string,int>::iterator it;
  for (it = M.begin(); it!= M.end(); ++ it){ // Bucle O(k log k) siendo k el numero de palabras
distintas
    pal.push_back( (*it).first );
    frec.push_back( (*it).second );
     }
}
```

En esta versión 4, tenemos dos sentencias de bucles for. Voy a empezar analizando cada bucle individualmente.

El primero bucle for, contiene en su interior la sentencia con orden de eficiencia en tiempo logarítmica, por lo que obtenemos:

T(n)= $\sum_{i=0}^{n} \log(n)$ => T(n)=n*log(n) => O(n*log(n)).

En el segundo bucle for, tenemos como se indica en los comentarios del código, una eficiencia de k*log k, siendo k el número de palabras distintas, pero para nuestro caso, siempre en el peor de los casos, y sin conocer nunca el tamaño de datos de entrada para nuestro programa, diremos que el tamaño es “n”.

Por ello, obtenemos una eficiencia igual que la eficiencia del primero bucle, el resultado sería la suma de ambas, pero como he mencionado anteriormente, para un tamaño n muy grande, las constantes son irrelevantes y podemos decir que la eficiencia de esta versión es:

T(n)=n*log(n) => O(n*log(n)).

2.2 Cálculo de la eficiencia empírica.

Como dice el guión he utilizado la toma de tiempos promedio, ya que en este caso la precisión del reloj no era suficiente. Utilizando una batería de pruebas desde 10 hasta 10000 en intervalos de 100 en 100.

NOTA: Para ser un poco mas breve, junto los gráficos de los valores empíricos con la regresión de cada ejercicio, para evitar la extensión de esta documentación como en la cuestión anterior.

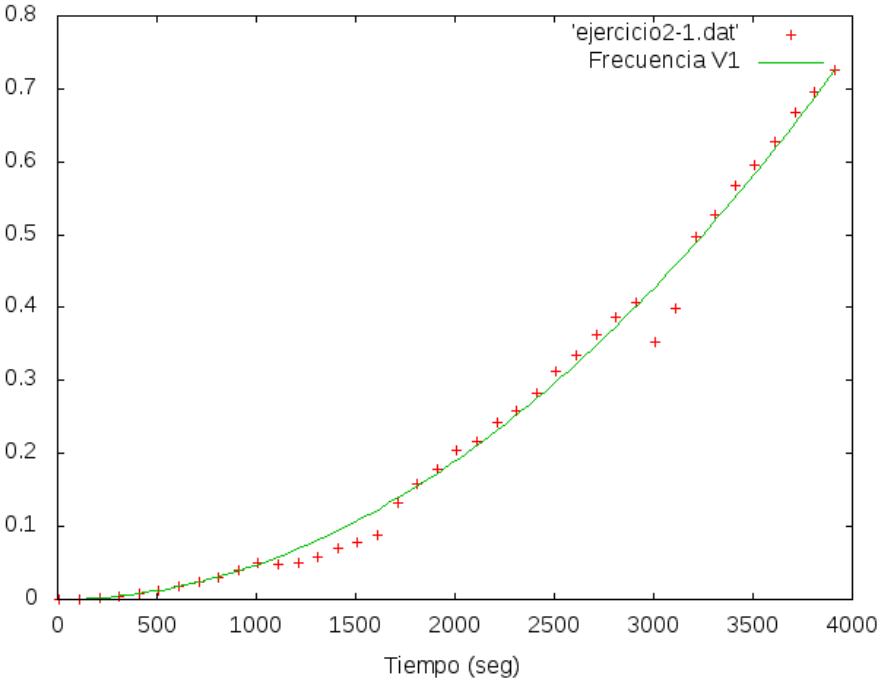
2.3 Cálculo de la eficiencia híbrida.

V1:

La función de regresión como se ha estudiado en la parte teórica es:

f(x) = a*x*x

```
marcos@ubuntu: ~/Escritorio/3ºAño/1º Cuatrimestre/ED/practica1

After 3 iterations the fit converged.
final sum of squares of residuals : 0.0151456
rel. change during last iteration : -1.07372e-08

degrees of freedom    (FIT_NDF)                        : 39
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 0.0197065
variance of residuals (reduced chisquare) = WSSR/ndf   : 0.000388347

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

a               = 4.74275e-08      +/- 4.467e-10    (0.9418%)


correlation matrix of the fit parameters:

               a
a               1.000
gnuplot> plot 'ejercicio2-1.dat', f(x) title 'Frecuencia V1'
gnuplot>
```

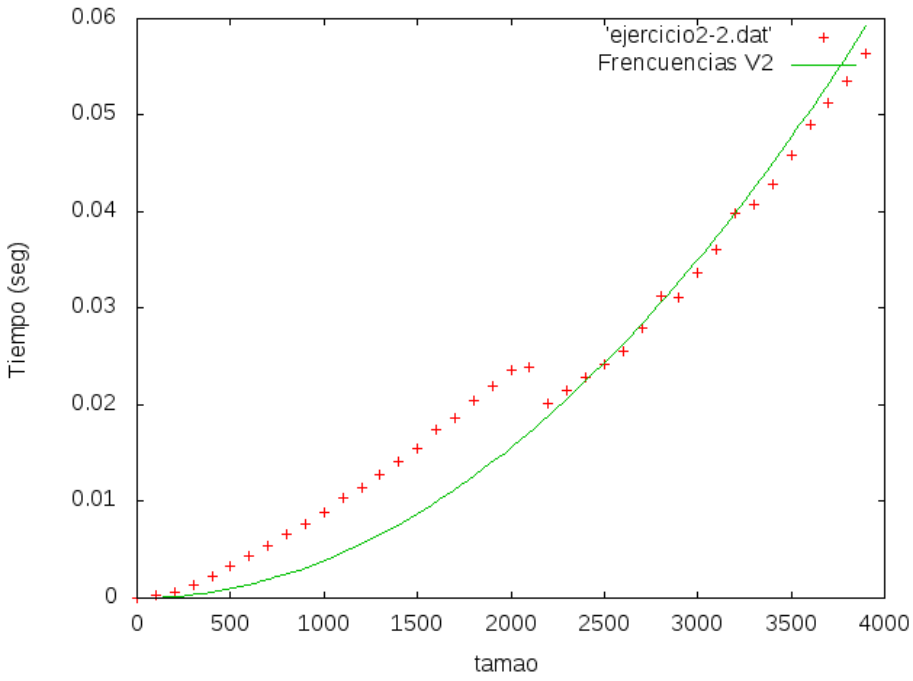


V2:

La función de regresión como se ha estudiado en la parte teórica es:

f(x) = a*x*x

```
marcos@ubuntu: ~/Escritorio/3ºAño/1º Cuatrimestre/ED/practica1

resultant parameter values

a               = 3.89473e-09

After 5 iterations the fit converged.
final sum of squares of residuals : 0.000667536
rel. change during last iteration : -3.24837e-16

degrees of freedom    (FIT_NDF)                        : 39
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 0.00413719
variance of residuals (reduced chisquare) = WSSR/ndf   : 1.71163e-05

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

a               = 3.89473e-09      +/- 9.437e-11    (2.423%)


correlation matrix of the fit parameters:

                a
a               1.000
gnuplot>
```

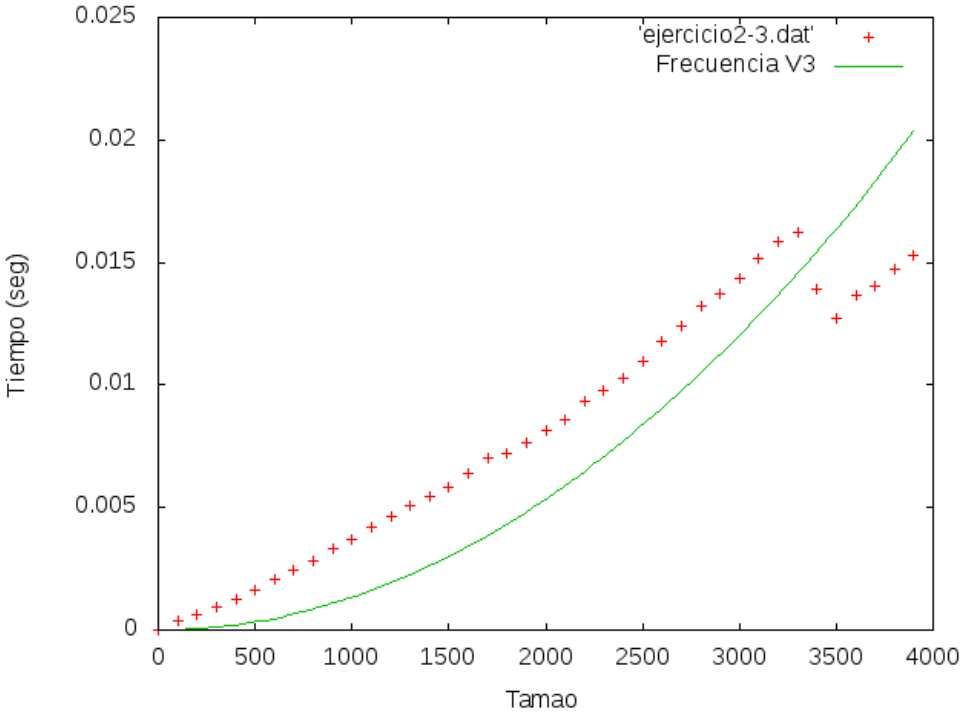


V3:

La función de regresión como se ha estudiado en la parte teórica es:

f(x) = a*x*x

```
marcos@ubuntu: ~/Escritorio/3ºAño/1º Cuatrimestre/ED/practica1

After 5 iterations the fit converged.
final sum of squares of residuals : 0.000280188
rel. change during last iteration : -1.93478e-16

degrees of freedom    (FIT_NDF)                        : 39
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 0.00268036
variance of residuals (reduced chisquare) = WSSR/ndf   : 7.18431e-06

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

a               = 1.33773e-09      +/- 6.114e-11    (4.57%)


correlation matrix of the fit parameters:

               a
a               1.000
```

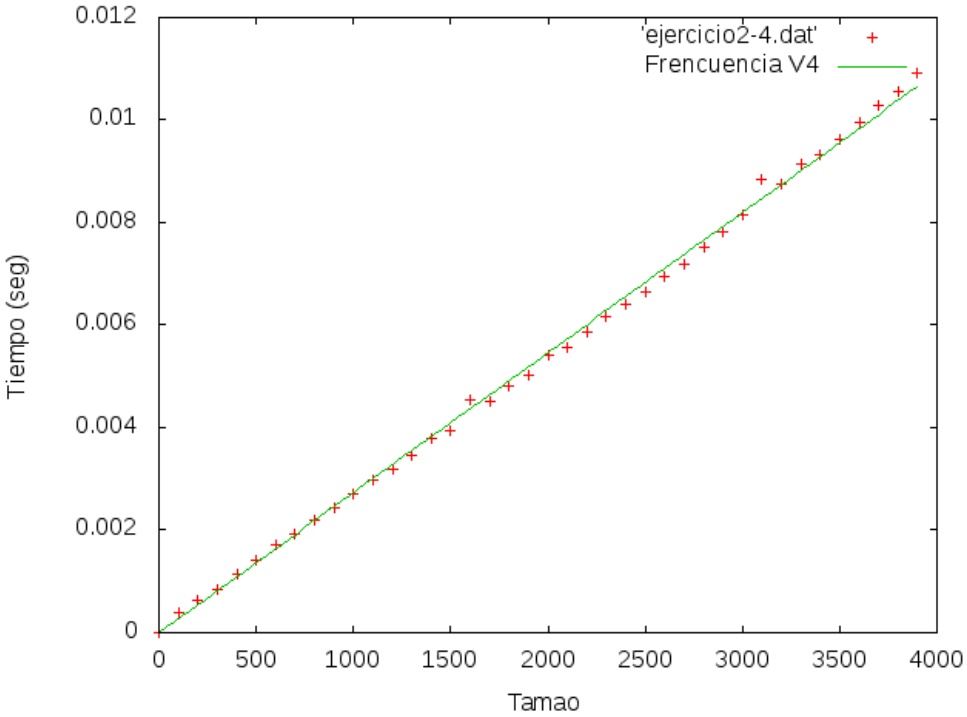


V4:

La función de regresión como se ha estudiado en la parte teórica es:

f(x) = a*x*log(x)

```
marcos@ubuntu: ~/Escritorio/3ºAño/1º Cuatrimestre/ED/practica1

resultant parameter values

a               = 2.73246e-06

After 4 iterations the fit converged.
final sum of squares of residuals : 6.88207e-07
rel. change during last iteration : -6.93189e-08

degrees of freedom    (FIT_NDF)                        : 39
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 0.00013284
variance of residuals (reduced chisquare) = WSSR/ndf   : 1.76463e-08

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

a               = 2.73246e-06      +/- 9.269e-09    (0.3392%)


correlation matrix of the fit parameters:

               a
a               1.000
gnuplot>
```



En conclusión como podemos ver en todos los gráficos, la evaluación teórica y la evaluación empírica se asemejan bastante o deberían de ser prácticamente similares. La diferencia podría ser el uso del microprocesador que este ocupado ejecutando instrucciones que no son del programa, y en este caso pueden empeorar los valores obtenidos.

Si nos encontramos con un gráfico que la evaluación teórica o empírica son totalmente distintas, hemos fallado seguramente en el cálculo de la eficiencia teórica. De hecho, esta práctica la he realizado bajo una máquina

virtual, y por eso podemos ver claramente que los tiempos se ven afectados.

Y para terminar después de la evaluación de las distintas versiones, podemos elegir como mejor versión la número 4, ya que tiene un orden de eficiencia mejor que todos los demás, y como podemos observar para un número infinito o muy grande de datos obtendremos mejores tiempos.

3. TERCERA CUESTIÓN.

En esta ecuación volveremos a repetir todos los procesos anteriores pero en este caso con los algoritmos de ordenación por selección, inserción, burbuja y quicksort.

3.1 Cálculo de tiempo teórico.

Burbuja:

```
void burbuja(vector<int> & T, int inicial, int final)
{
 int i, j;
 int aux;
 for (i = inicial; i < final - 1; i++)
   for (j = final - 1; j > i; j--)
       if (T[j] < T[j-1]) //O(1)
         {
           aux = T[j]; //O(1)
           T[j] = T[j-1]; //O(1)
           T[j-1] = aux; //O(1)
         }
}
```

Empezamos analizando las sentencias mas internas del segundo bucle for, ya que nos encontramos con dos bucles anidados. Vemos que todas estas sentencias mas internas son constantes, y como el tamaño del segundo bucle, como siempre, en el peor de los casos tamaño “n”.

En las mismas condiciones en el primer bucle, obtenemos:

T(n)= $\sum_{i=0}^{n} \sum_{i=0}^{n} a$ => T(n)= $\sum_{i=0}^{n} n$ = n*n=n^2. O(n^2). Tiene una eficiencia en

tiempo cuadrática.

Inserción:

```
void Insercion (vector<int> & T, int inicial, int final)
{
  int i, j;
  int aux;
  for (i = inicial; i < final - 1; i++){
    aux=T[i];//O(1)
    j=i;//O(1)
    while (T[j-1]>aux) {
      T[j]= T[j-1];//O(1)
      j=j-1;//O(1)
      if (j==0) break;//O(1)
    }
    T[j]= aux;//O(1)
  }
}
```

Igual que en el apartado anterior de la burbuja, encontramos el mismo proceso, aunque el segundo bucle esta vez es un “while” obtenemos la misma eficiencia lineal.

Anidado a un primer bucle “for” obtenemos la misma ecuación anterior y el mismo orden de eficiencia:

$T(n)=\sum_{i=0}^{n}\sum_{i=0}^{n} a$ => $T(n)=\sum_{i=0}^{n} n$ = n*n=n^2. $O(n^2)$. Tiene una eficiencia en tiempo cuadrática.

Selección:

```
void Seleccion (vector<int> & T, int inicial, int final)
{

  int i, j;
  int aux;
  for (i = inicial; i < final - 1; i++) {
    aux=j; //O(1)
    for (i=j+1;i<final-1;i++)
      if (T[i]<T[aux]){//O(1)
        aux=i;//O(1)
        i= T[j];//O(1)
        T[j]= T[aux];//O(1)
        T[aux]=i;//O(1)
      }
    }
}
```

Volvemos a repetir las mismas características anteriores. Dos bucles for con las sumatorias de las sentencias internas como constantes, ya que las asignaciones y las comprobaciones la tomamos como orden de eficiencia constantes.

T(n)= $\sum_{i=0}^{n} \sum_{i=0}^{n} a$ => T(n)= $\sum_{i=0}^{n} n$ = n*n=n^2. O(n^2). Tiene una eficiencia en

tiempo cuadrática.

QuickSort:

```
void quicksort(vector<int> & T, int inicial, int final)
{
    int pivote;

    if (inicial < final) {
        pivote = divide(T, inicial, final); //O(n)

        // Ordeno la lista de los menores
        quicksort(T, inicial, pivote - 1);

        // Ordeno la lista de los mayores
        quicksort(T, pivote + 1, final);
    }
}
```

Este algoritmo es particular, ya que ahora como vamos a analizar la eficiencia, vamos a obtener una eficiencia de orden cuadrática, para en el peor de los casos, como los anteriores. Pero cuando analizamos la eficiencia de este algoritmo en caso promedio, obtenemos una eficiencia mucho mas rápida (n*log(n)).

Esta función es recursiva por lo que obtenemos, en cada llamada a esta funcion recursiva a su vez dos llamadas recursivas, con la mitad del tamaño (n/2):

T(n)=n+2T(n/2) => T(n)=n*n=n^2. Tiene un orden de eficiencia cuadrático en el peor de los casos.

Adjunto el código de la función “divide()” por si queremos analizar la eficiencia de esta función, pero básicamente se encarga de ir moviendo los índices e ir comparando, para poder poner los números menores al principio del vector y los mayores al final de vector, así sucesivamente.

3.2 Cálculo de la eficiencia empírica.

Como en todas las demás sigo utilizado la toma de tiempos promedio, ya que en este caso la precisión del reloj no era suficiente. Utilizando una batería de pruebas desde 10 hasta 10000 en intervalos de 100 en 100.

NOTA: Para ser un poco mas breve, junto los gráficos de los valores empíricos con la regresión de cada ejercicio, para evitar la extensión de esta documentación como en la cuestión anterior.

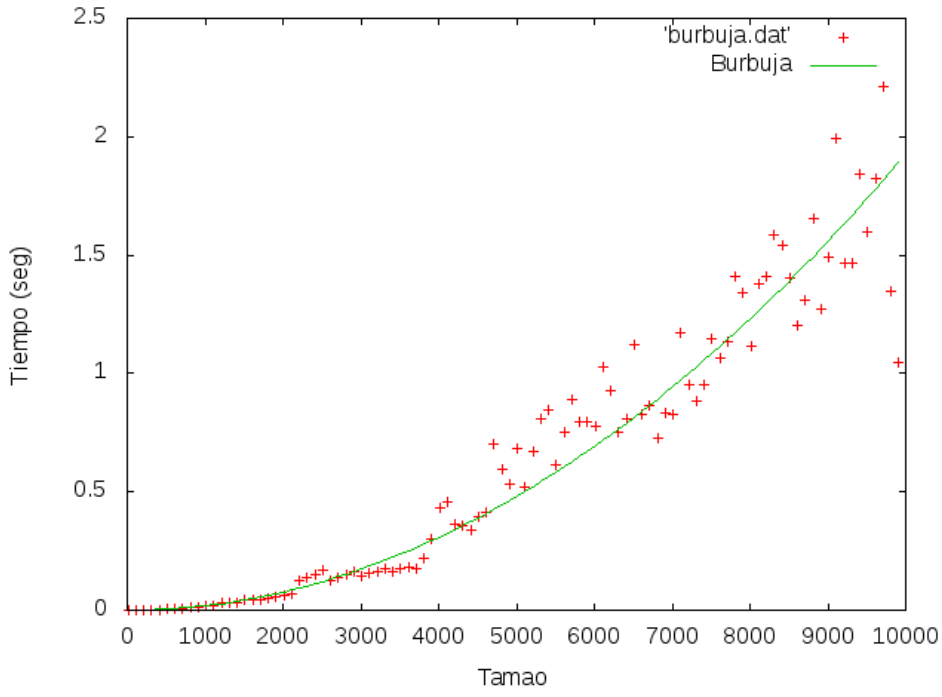
3.3 Cálculo de la eficiencia híbrida.

Burbuja:

La función de regresión como se ha estudiado en la parte teórica es:

f(x) = a*x*x

```
marcos@ubuntu: ~/Escritorio/3ºAño/1º Cuatrimestre/ED/practica1

resultant parameter values

a               = 1.92446e-08

After 4 iterations the fit converged.
final sum of squares of residuals : 2.64463
rel. change during last iteration : -7.26432e-08

degrees of freedom    (FIT_NDF)                       : 99
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 0.163442
variance of residuals (reduced chisquare) = WSSR/ndf   : 0.0267134

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

a               = 1.92446e-08      +/- 3.692e-10    (1.918%)


correlation matrix of the fit parameters:

               a
a               1.000
gnuplot>
```

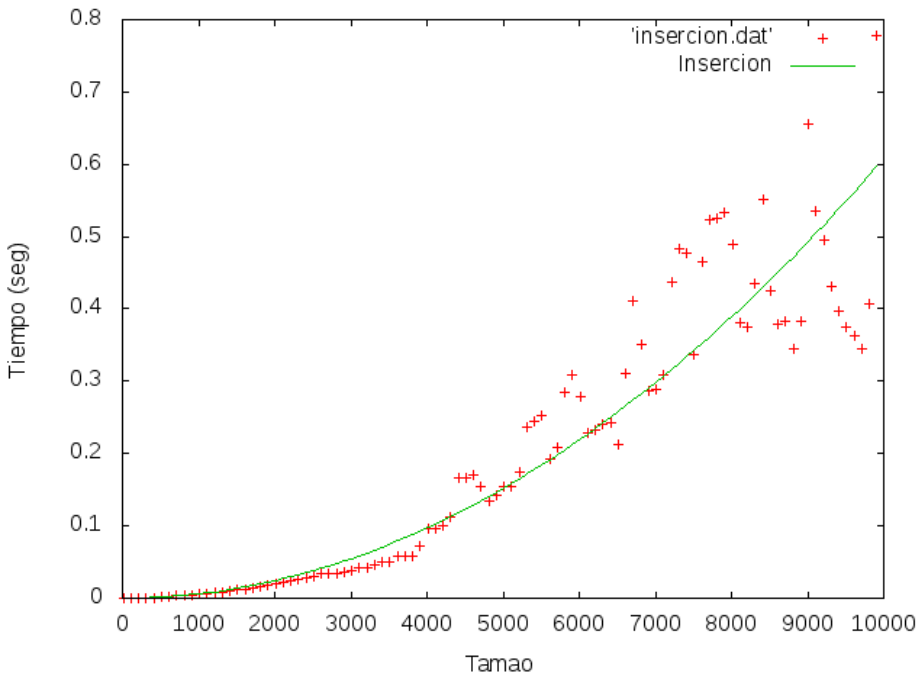


Inserción:

La función de regresión como se ha estudiado en la parte teórica es:

f(x) = a*x*x

```
marcos@ubuntu: ~/Escritorio/3ºAño/1º Cuatrimestre/ED/practica1

resultant parameter values

a               = 6.08128e-09

After 3 iterations the fit converged.
final sum of squares of residuals : 0.527935
rel. change during last iteration : -6.30546e-11

degrees of freedom    (FIT_NDF)                        : 99
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 0.0730252
variance of residuals (reduced chisquare) = WSSR/ndf   : 0.00533267

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

a               = 6.08128e-09      +/- 1.649e-10    (2.712%)


correlation matrix of the fit parameters:

               a
a               1.000
gnuplot>
```

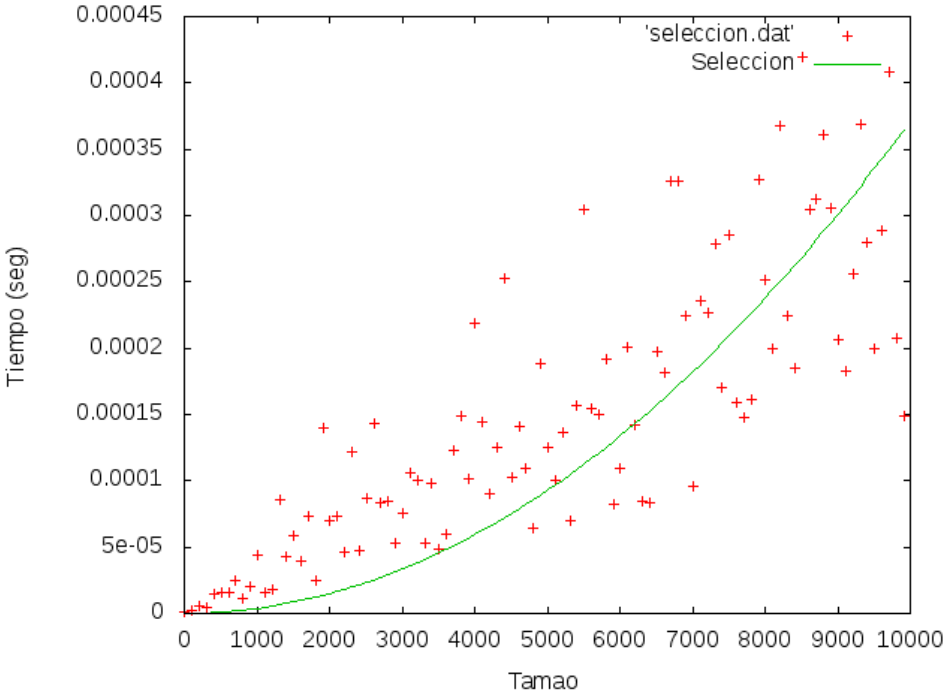


Selección:

La función de regresión como se ha estudiado en la parte teórica es:

f(x) = a*x*x

```
marcos@ubuntu: ~/Escritorio/3ºAño/1º Cuatrimestre/ED/practica1

resultant parameter values

a               = 3.7129e-12

After 5 iterations the fit converged.
final sum of squares of residuals : 5.34077e-07
rel. change during last iteration : -3.96494e-16

degrees of freedom    (FIT_NDF)                       : 99
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)   : 7.34487e-05
variance of residuals (reduced chisquare) = WSSR/ndf  : 5.39472e-09

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

a               = 3.7129e-12       +/- 1.659e-13    (4.468%)


correlation matrix of the fit parameters:

               a
a               1.000
gnuplot>
```

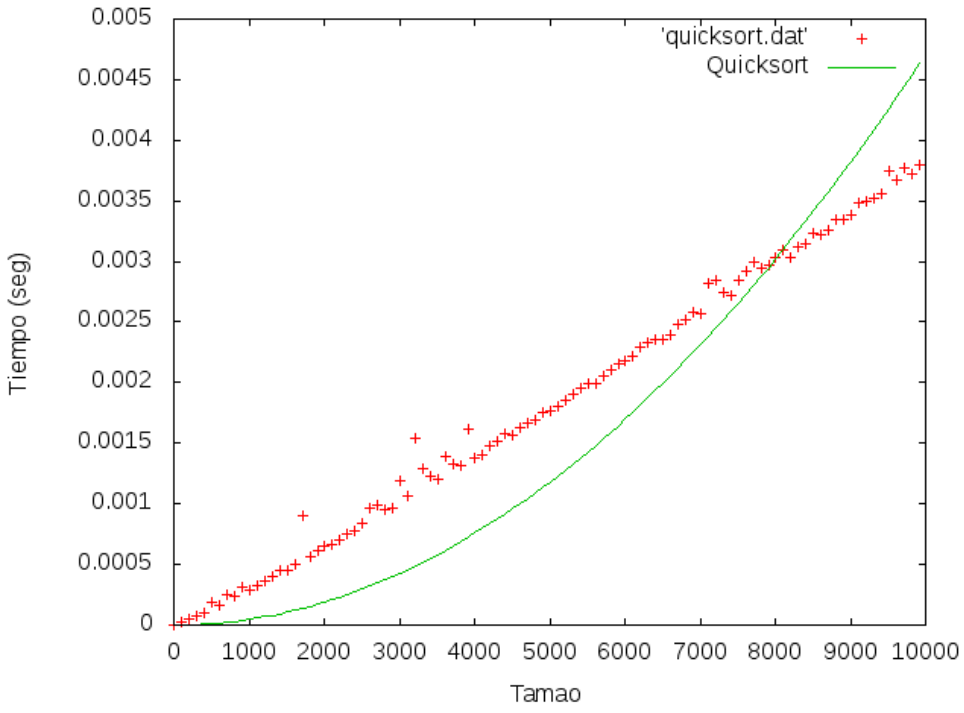


QuickSort:

La función de regresión como se ha estudiado en la parte teórica es:

f(x) = a*x*x

```
marcos@ubuntu: ~/Escritorio/3ºAño/1º Cuatrimestre/ED/practica1

resultant parameter values

a               = 4.71529e-11
*****************
The maximum lambda = 1.000000e+20 was exceeded. Fit stopped.
final sum of squares of residuals : 2.41393e-05
rel. change during last iteration : 0

degrees of freedom    (FIT_NDF)                        : 99
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 0.000493793
variance of residuals (reduced chisquare) = WSSR/ndf   : 2.43831e-07

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

a               = 4.71529e-11      +/- 1.115e-12    (2.365%)


correlation matrix of the fit parameters:

               a
a               1.000
gnuplot>
```



Como vemos claramente la mejora del QuickSort, con una gran cantidad de datos, con tamaño “n”, es prácticamente lineal en la práctica, aunque en la eficiencia teórica obtenemos orden cuadrático.