Práctica 1: Eficiencia

José Antonio Álvarez Ocete

3 de octubre de 2017

1. Informe de la eficiencia

Antes de comenzar con los ejercicios voy a describir el ordenador en el que los he realizado. Adjunto un archivo **cpuinfo.txt** con la información de la CPU, con un procesador de 2 cores fisicos e Intel Core i7-5500U, 2.40GHz. La gráfica es una Nvidia GeForce 920M y tiene una RAM de 8GB.
En cuanto al sistema operativo, he usado Linux v16.04, el compilador es g++, la versión por defecto. No he usado opciones de compilación salvo *-o*, con una única excepción en el ejercicio 6.
He adjuntado todos los códigos fuente y los scripts usados, incluyendo en este documento unicamente los fragmentos más relevantes. También he incluido todos los datos recopilados, registros y capturas de todas las ordenes *fit* de gnuplot ejecutadas y todas las gráficas.

2. Ejercicios

2.1. Ejercicio 1: Ordenación de la Burbja

El siguiente código realiza la ordenación mediante el algoritmo de la burbuja:

```
void ordenar(int *v, int n) {
        for (int i=0; i<n-1; i++)
                for (int j=0; j<n-i-1; j++)
                        if (v[j]>v[j+1]) {
                                int aux = v[j];
                                v[j] = v[j+1];
                                v[j+1] = aux;
                        }
                }
        }
}
```

Calcule la eficiencia teórica de este algoritmo. A continuación replique el experimento que se ha hecho antes (búsqueda lineal) con este nuevo código. Debe:

- Crear un fichero ordenacion.cpp con el programa completo para realizar una ejecución del algoritmo.
- Crear un script ejecuciones_ordenacion.csh en C-Shell que permite ejecutar varias veces el programa anterior y generar un fichero con los datos obtenidos.
- Usar gnuplot para dibujar los datos obtenidos en el apartado previo.

Los datos deben contener tiempos de ejecución para tamaños del vector 100, 600, 1100, ..., 30000. Pruebe a dibujar superpuestas la función con la eficiencia teórica y la empírica. ¿Qué sucede?

Solución:
El archivo fuente (**ordenacion.cpp**) y el script (**ejecuciones_ordenacion.sh**) están adjuntos, así como los datos obtenidos. Representamos ahora los datos mediante gnuplot:

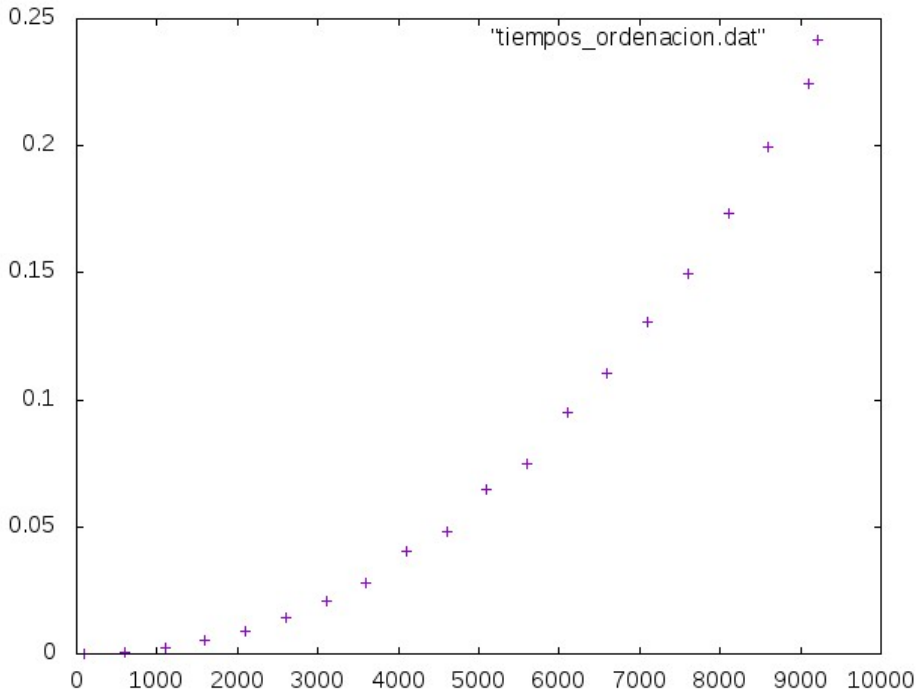

Figura 1: Tiempos de ordenación

Aunque la eficiencia es evidentemente de orden $O(n^2)$, calculamos el polinomio completo, con sus coeficientes, para poder representarlo. Primero explicaremos de donde viene cada coeficiente y depués obtendremos el polinomio:

- (for) + (comprobación, 2) + (inicialización) = 4, al entrar al primer bucle.
- Dentro del primer bucle, de i=0 hasta n-1:
  - (comprobación, 2) + (incremento) = 3, en cada iteración del primer bucle.
  - (for) + (comprobación, 3) + (inicialización) = 5, al entrar al segundo bucle.
  - Dentro del segundo bucle, de j=0 hasta n-i-1:
    - (comprobación, 3) + (incremento) = 4, en cada iteración del bucle.
    - 13, interior del bucle.

Formalizamos matemáticamente y simplificamos:

$$4 + \sum_{i=0}^{n-1} (3 + 5 + \sum_{j=0}^{n-i-1} 4 + 13)$$

$$= 4 + \sum_{i=0}^{n-1} (8 + 17 * \sum_{j=0}^{n-i-1} 1)$$

$$= 4 + 8 * (n-1) + 17 \sum_{i=0}^{n-1} \sum_{j=0}^{n-i-1} 1$$

$$= 4 + 8 * (n-1) + 17 \sum_{i=0}^{n-1} (n - i - 1)$$

$$= 4 + 8 * (n-1) + 17 \sum_{i=0}^{n-1} n - \sum_{i=0}^{n-1} i - \sum_{i=0}^{n-1} 1$$

$$= 4 + 8 * (n-1) + 17(n * (n-1) - n * \frac{n-1}{2} - (n-1))$$

$$= 8{,}5 * n^2 - 17{,}5 * n + 13$$

Al pintar esta gráfica junto a los datos obtenidos anteriormente obtenemos el siguiente resultado:

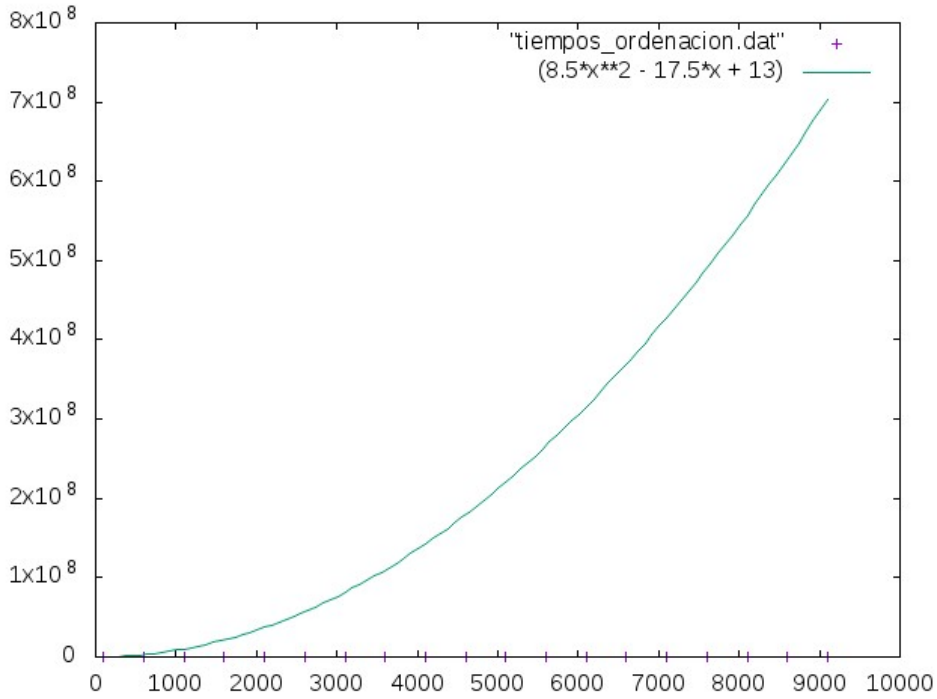

Figura 2: Comparativa empírica - teórica 1

Como podemos ver, todos los datos se agrupan sobre el eje X. Esto se debe a que los estos valores son muchísimo más pequeños que la gráfica obtenida teóricamente. Para apreciar más o menos que ambas funciones son del mismo orden, multiplicamos la función teórica por un factor atenuante.

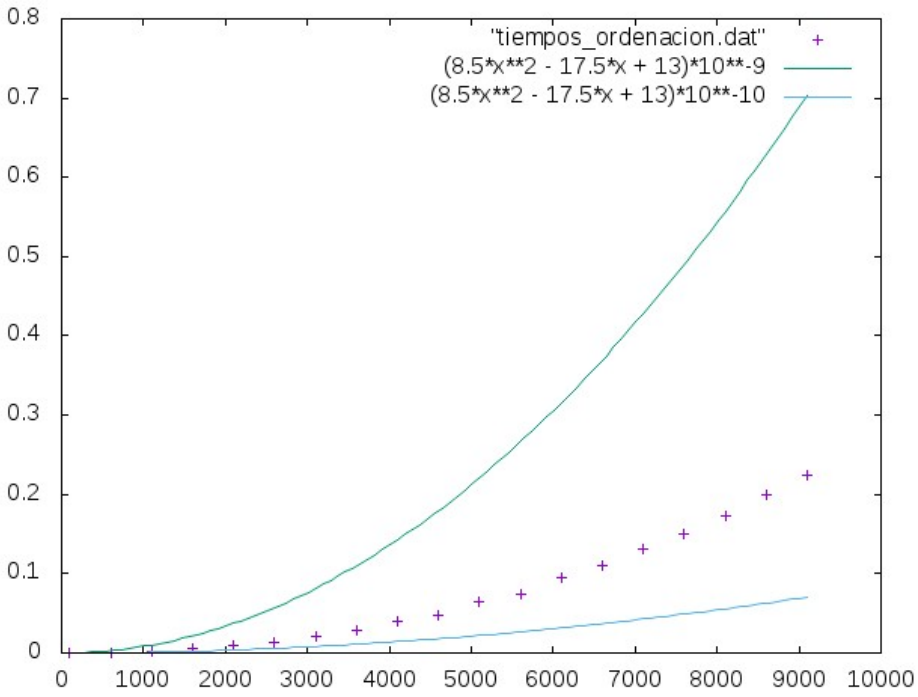

Figura 3: Comparativa empírica - teórica 2

2.2. Ejercicio 2: Ajuste en la ordenación de la burbuja

Replique el experimento de ajuste por regresión a los resultados obtenidos en el ejercicio 1 que calculaba la eficiencia del algoritmo de ordenación de la burbuja. Para ello considere que f(x) es de la forma $a * x^2 + b * x + c$.

Solución:
Realizamos el ajuste mediante gnuplot, (el registro del ajuste está almacenado como **fit.log(2_1)**) obteniendo los siguientes resultados:

```
gnuplot> f(x) = a*x**2 + b*x + c
gnuplot> fit f(x) "tiempos_ordenacion.dat" via a, b, c
iter      chisq       delta/lim  lambda   a             b             c
   0 2.8523224652e+16   0.00e+00  2.24e+07    1.000000e+00   1.000000e+00   1.000000e+00
   1 8.4789948928e+12  -3.36e+08  2.24e+06    1.710987e-02   9.998684e-01   1.000000e+00
   2 3.4188696090e+07  -2.48e+10  2.24e+05   -1.307904e-04   9.998593e-01   1.000000e+00
   3 3.3881870995e+07  -9.06e+02  2.24e+04   -1.337244e-04   9.991818e-01   9.999997e-01
   4 2.9716970766e+07  -1.40e+04  2.24e+03   -1.252334e-04   9.357296e-01   9.999725e-01
   5 4.9122503359e+05  -5.95e+06  2.24e+02   -1.606523e-05   1.199303e-01   9.996224e-01
   6 3.4259051617e+00  -1.43e+10  2.24e+01    1.750495e-08  -2.538590e-04   9.995236e-01
   7 2.3375463128e+00  -4.66e+04  2.24e+00    4.105264e-08  -4.291557e-04   9.948145e-01
   8 1.0750618248e+00  -1.17e+05  2.24e-01    2.883803e-08  -2.923060e-04   6.754706e-01
   9 4.9563805844e-04  -2.17e+08  2.24e-02    3.630565e-09  -9.890168e-06   1.645493e-02
  10 3.7820789629e-05  -1.21e+06  2.24e-03    3.099520e-09  -3.940528e-06   2.571486e-03
  11 3.7820769310e-05  -5.37e-02  2.24e-04    3.099409e-09  -3.939274e-06   2.568560e-03
iter      chisq       delta/lim  lambda   a             b             c

After 11 iterations the fit converged.
final sum of squares of residuals : 3.78208e-05
rel. change during last iteration : -5.37236e-07

degrees of freedom    (FIT_NDF)                        : 16
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 0.00153746
variance of residuals (reduced chisquare) = WSSR/ndf   : 2.3638e-06

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a               = 3.09941e-09      +/- 5.28e-11     (1.704%)
b               = -3.93927e-06     +/- 5.026e-07    (12.76%)
c               = 0.00256856       +/- 0.0009978    (38.85%)

correlation matrix of the fit parameters:
                a      b      c
a               1.000
b              -0.967  1.000
c               0.723 -0.851  1.000
gnuplot>
```

Figura 4: Ajuste empírico del ejercicio 1

Representamos ahora el ajuste obtenido frente a los datos:

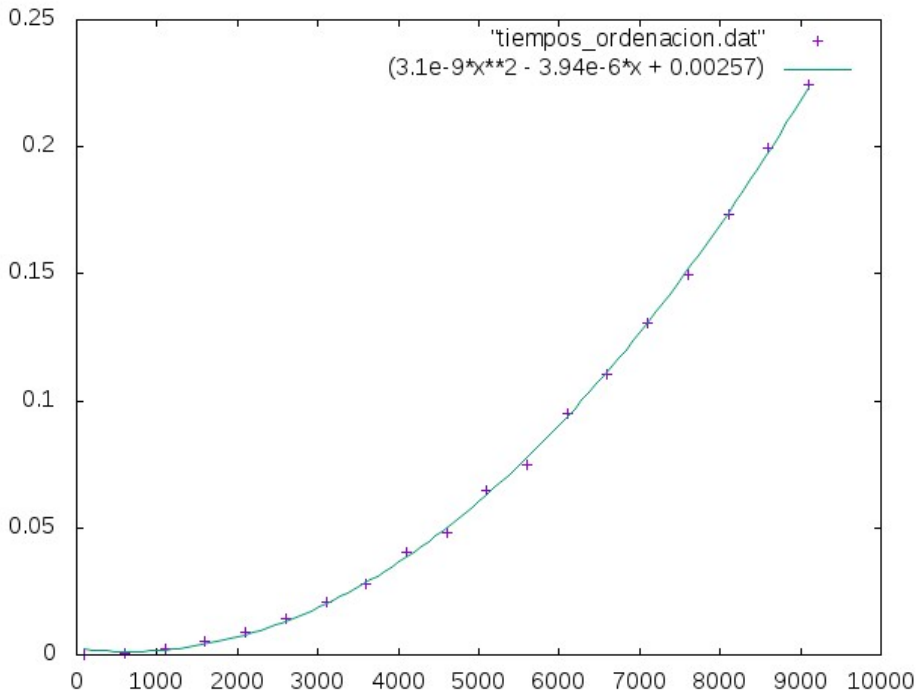

Figura 5: Ajuste empírico frente a los datos obtenidos en el ejercicio 1

2.3. Ejercicio 3: Problemas de precisión

Junto con este guión se le ha suministrado un fichero ejercicio_desc.cpp. En él se ha implementado un algoritmo. Se pide que:

- Explique qué hace este algoritmo.
- Calcule su eficiencia teórica.
- Calcule su eficiencia empírica.

Si visualiza la eficiencia empírica debería notar algo anormal. Explíquelo y proponga una solución. Compruebe que su solución es correcta. Una vez resuelto el problema realice la regresión para ajustar la curva teórica a la empírica.

Solución:
Este algoritmo implementa una búsquedad binaria. Cabe destacar que este algoritmo se aplica sobre un vector ordenado, condición que en este caso no estamos cumpliendo. Obviamente no nos interesa encontrar el elemento (de hecho forzamos que no pueda encontrarse) sino estudiar la eficiencia en el peor caso posible: que el elemento no se halle en el vector.
En cuanto al estudio teórico, es bastante sencillo: la eficiencia es del orden $O(log(n))$. Esto se debe a que reducimos el vector en el que buscamos a la mitad en cada iteración, provocando una progresión de orden logarítmico.
Tomamos muestras de tiempo y las representamos:

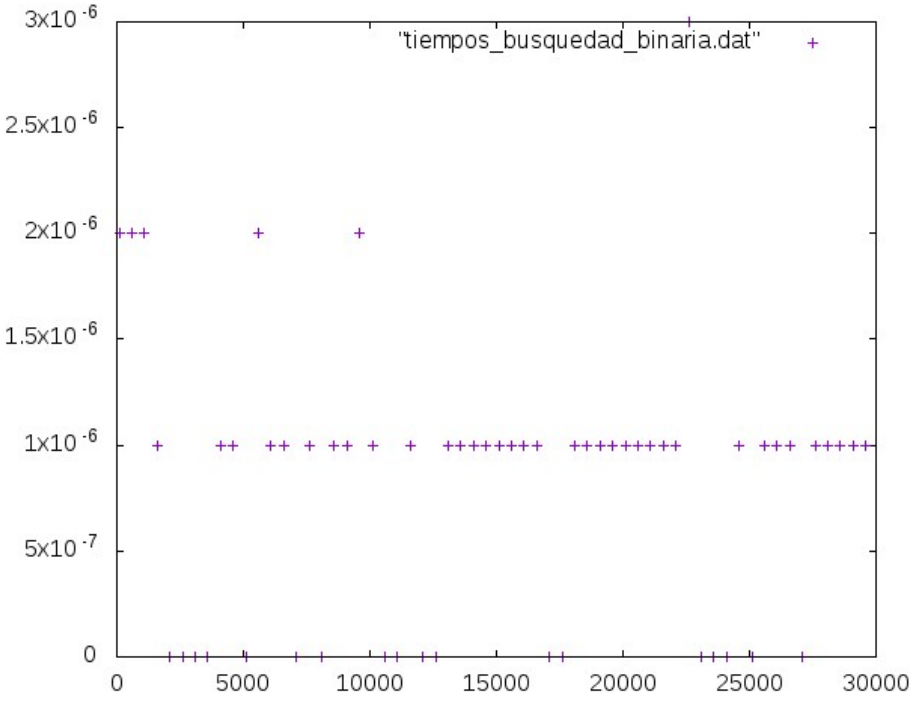

Figura 6: Tiempos de búsquedad binaria 1

Evidentemente hemos cometido un error: la función *clock()* tiene dificultades para medir tiempos muy pequeños, como es el caso de este algoritmo de eficiencia logarítmica. Para resolver esto realizamos numerosas ejecuciones (denotémoslo por MAX), medimos el tiempo del total y dividimos por MAX. Adjunto un fragmento del archivo **ejercicio_desc.cpp** modificado (el cual también va adjunto):

```
const long int MAX=10000;
double timedif;
tini=clock();
for (int i=0; i<MAX; i++) {
  operacion(v,tam,tam+1,0,tam-1);
}
tfin=clock();
timedif = (tfin-tini)/(double)CLOCKS_PER_SEC;
timedif /= MAX;
```

Utilizando esta implementación repetimos el proceso de muestreo:

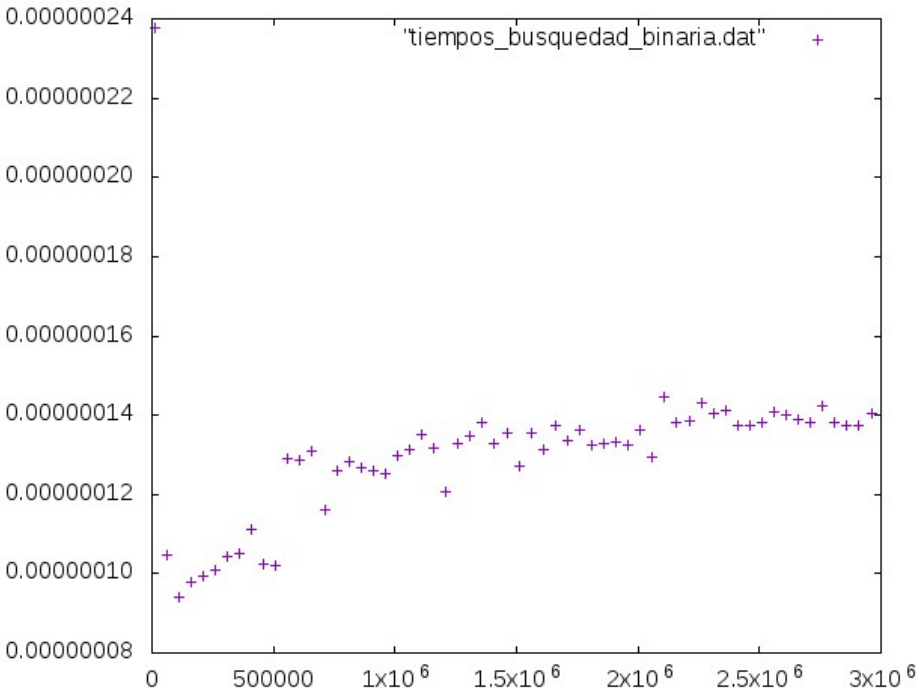

Figura 7: Tiempos de búsquedad binaria 2

Por último realizamos un ajuste conforme a los datos obtenidos utilizando la función $f(x) = a * log(b * x) + c$:

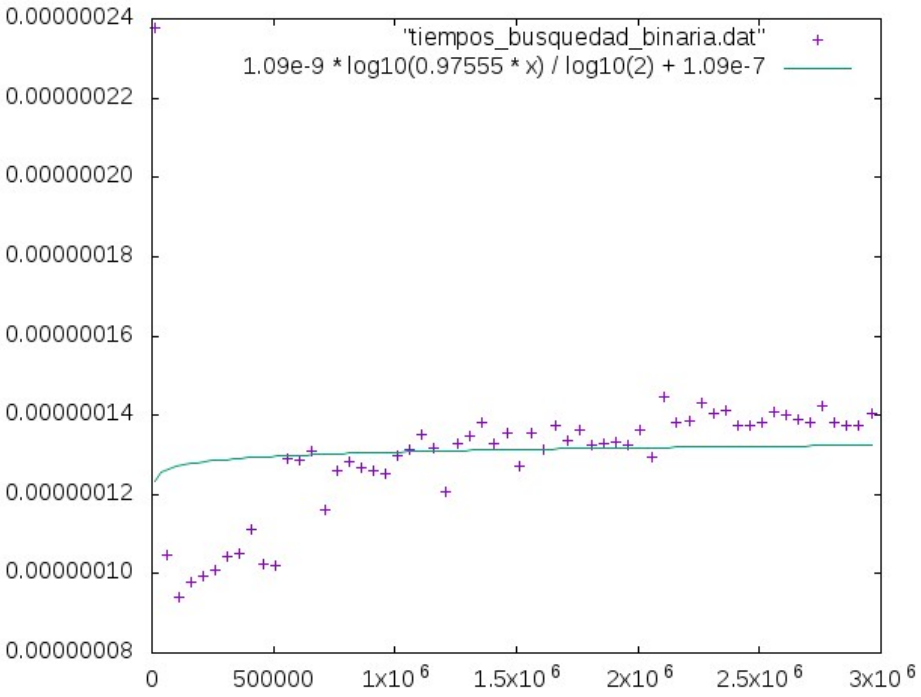

Figura 8: Tiempos de búsquedad binaria y ajuste.

2.4. Ejercicio 4: Mejor y peor caso

Retome el ejercicio de ordenación mediante el algoritmo de la burbuja. Debe modificar el código que genera los datos de entrada para situarnos en dos escenarios diferentes:

- El mejor caso posible. Para este algoritmo, si la entrada es un vector que ya está ordenado el tiempo de cómputo es menor ya que no tiene que intercambiar ningún elemento.
- El peor caso posible. Si la entrada es un vector ordenado en orden inverso estaremos en la peor situación posible ya que en cada iteración del bucle interno hay que hacer un intercambio.

Calcule la eficiencia empírica en ambos escenarios y compárela con el resultado del ejercicio 1.

Solución:
El mejor caso posible es aquel en el que el vector ya está ordenado. El peor caso es aquel en el que el vector está ordenado de forma inversa (de mayor a menor. Ahora pasamos sendos vectores creados de esta forma al algoritmo de ordenación y tomamos tiempos. Adjuntamos aquí el fragmento del código en el que se crean los vectores (*mejor* es un parámetro de entrada al programa). El código completo se encuentra en el archivo **ordenacionMejorPeor.cpp**.

```
int *v=new int[tam];
  if (mejor) {
    for (int i=0; i<tam; i++) {
      v[i] = i;
    }
  } else {
    for (int i=0; i<tam; i++) {
      v[i] = tam - i;
    }
  }
```

Representamos los datos obtenidos:

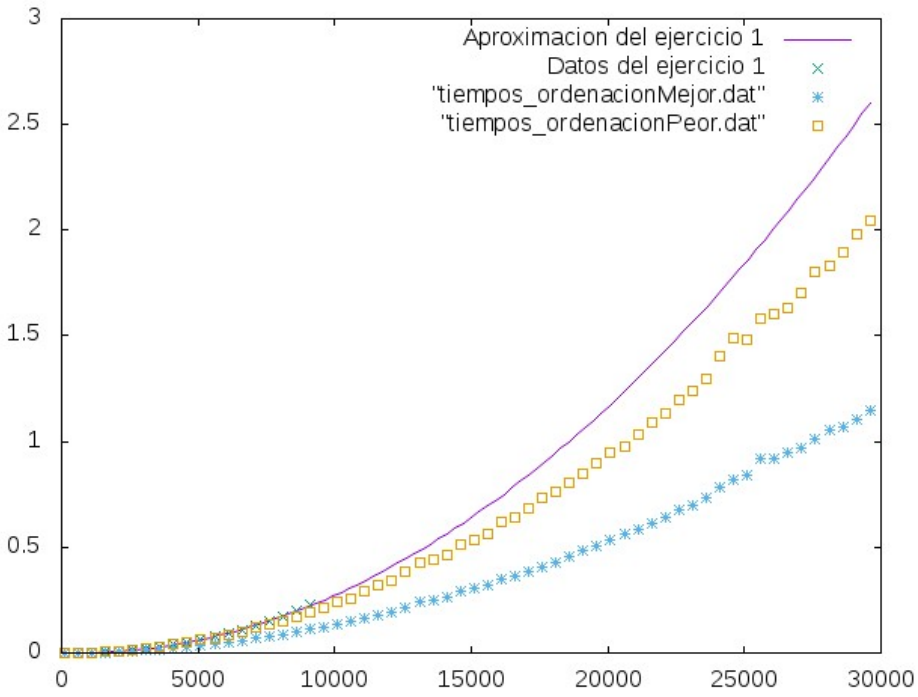

Figura 9: Mejor, peor y caso promedio de ordenación por burbuja

Como podemos observar en la gráfica, el mejor caso es el que menos tiempo requiere. Sin embargo, el peor caso está aparentemente mal realizado: según esta gráfica el caso aleatorio requiere más tiempo que el peor caso posible. No he encontrado una explicación clara para este hecho. Si me he asegurado, por otro lado, de asegurarme de que la creación del vector, la toma de datos y el uso de *gnuplot* están bien realizados. Por último, utilizamos de nuevo la función *fit* de gnuplot sobre estos datos para obtener sendas aproximaciones de ambos:

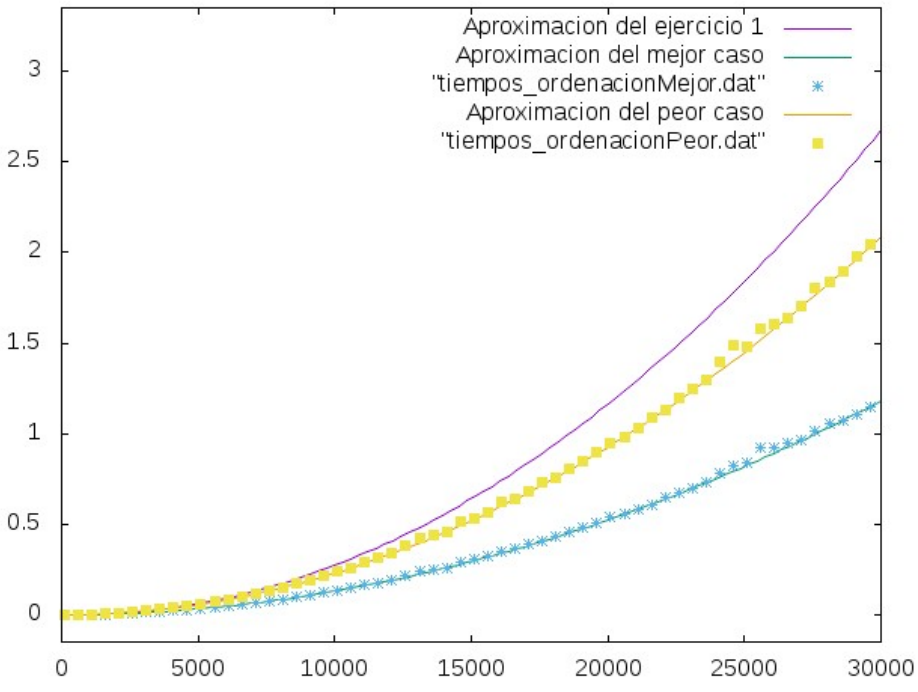

Figura 10: Aproximaciones del mejor, peor y caso promedio.

2.5. Ejercicio 5: Dependencia de la implementación

Considere esta otra implementación del algoritmo de la burbuja: void ordenar(int *v, int n) bool cambio=true;

```
void ordenar(int *v, int n) {
        for (int i=0; i<n-1 && cambio; i++) {
                cambio=false;
                for (int j=0; j<n-i-1; j++) {
                        if (v[j]>v[j+1]) {
                                cambio=true;
                                int aux = v[j];
                                v[j] = v[j+1];
                                v[j+1] = aux;
                        }
                }
        }
}
```

En ella se ha introducido una variable que permite saber si, en una de las iteraciones del bucle externo no se ha modificado el vector. Si esto ocurre significa que ya está ordenado y no hay que continuar. Considere ahora la situación del mejor caso posible en la que el vector de entrada ya está ordenado. ¿Cuál sería la eficiencia teórica en ese mejor caso? Muestre la gráfica con la eficiencia empírica y compruebe si se ajusta a la previsión.

Solución:

Ante esta nueva implementación y con el vector ya ordenado, el algoritmo se reduce a una iteración del bucle (recorrerlo y comprobar que esta ordenado). Es decir, su eficiencia es del orden *O(n)*. Evidentemente no tiene nada que ver con la implementación anterior:

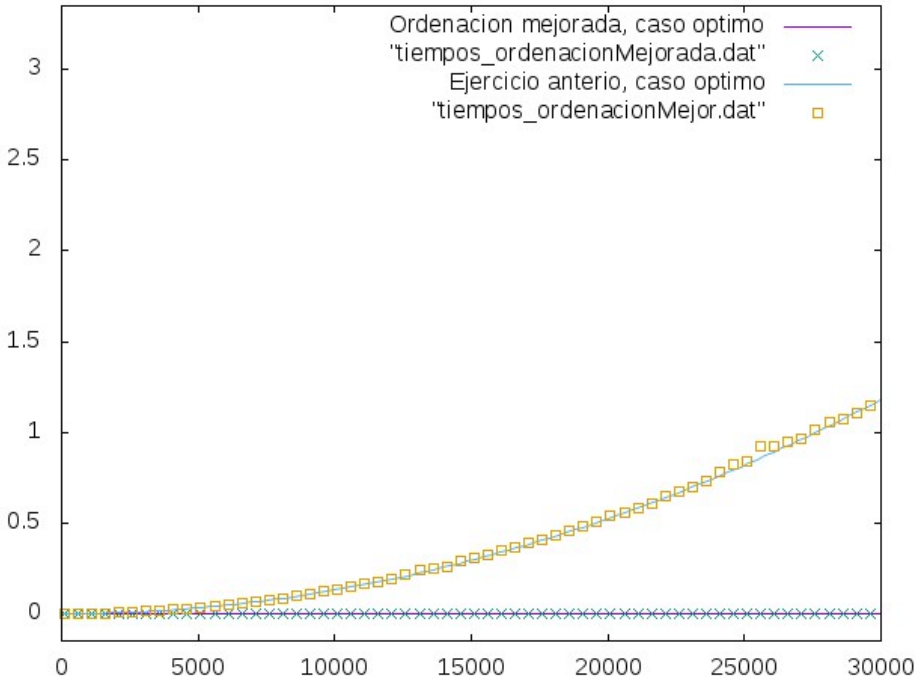

Figura 11: Dependencia de la implementación.

2.6. Ejercicio 6: Influencia del proceso de compilación

Retome el ejercicio de ordenación mediante el algoritmo de la burbuja. Ahora replique dicho ejercicio pero previamente deberá compilar el programa indicándole al compilador que optimice el código. Esto se consigue así.

$$g++ -O3 ordenacion.cpp -o ordenacion\_optimizado$$

Compare las curvas de eficiencia empírica para ver cómo mejora esto la eficiencia del programa.

Solución:
Compilamos de la forma indicada, ejecutamos para obtener los datos y representamos el resultado frente a los datos del ejercicio anterior:

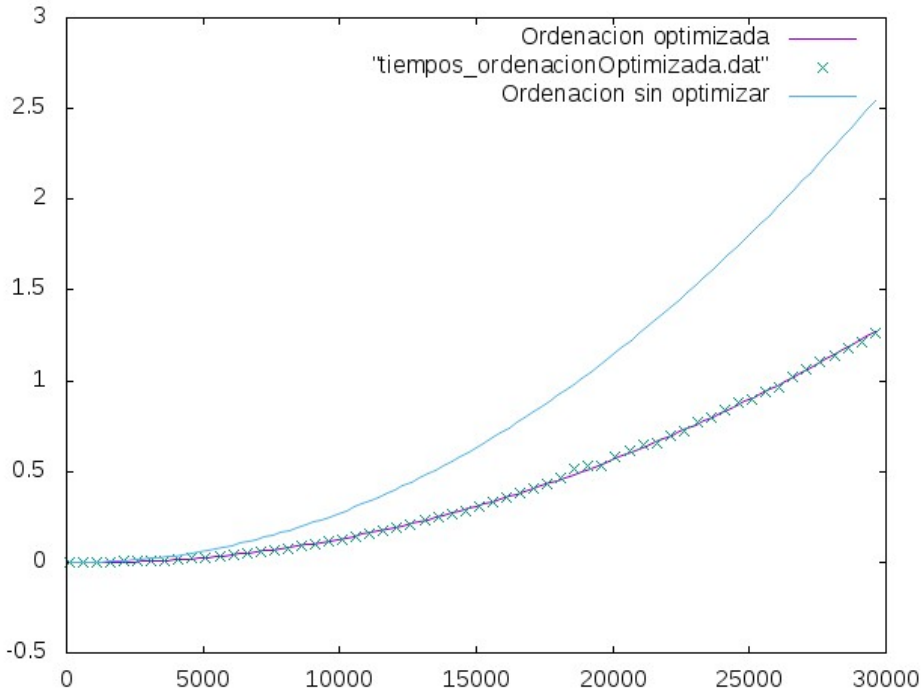

Figura 12: Influencia del proceso de compilación.

Nota: Este es el único ejercicio que no lleva asociado ningún archivo de código fuente (se ha compilado a partir del original, como indica el enunciado) pero si que lleva asociado un script, **ejecuciones_ordenacionOptimizada.sh**, para mayor comodidad.

2.7. Ejercicio 7: Multiplicación matricial

Implemente un programa que realice la multiplicación de dos matrices bidimensionales. Realice un análisis completo de la eficiencia tal y como ha hecho en ejercicios anteriores de este guión.

Solución:
Adjuntamos aquí el fragmento de **MM.cpp** que implementa la multiplicación como tal:

```
void mult(int **m1, int **m2, int **m3, int tam) {
  int result;
  for (int i=0; i<tam; i++) {
    for (int j=0; j<tam; j++) {
      result = 0;
      for (int k=0; k<tam; k++) {
```

```
          result += m1[i][k] * m2[k][j];
        }
        m3[i][j] = result;
      }
    }
}
```

El estudio teórico de la eficiencia de este código es sencillo: tenemos tres bucles anidados, de $i, j, k = 0$ hasta tam, el tamaño de las matrices. Llamando $n = tam$, la eficiencia de la multiplicación de matrices implementada en este código es de $O(n^3)$.
Repetimos el procedimiento seguido en los otros ejercicios ajustando los datos a una función de la forma $f(x) = a * x^3 + b * x^2 + c * x + d$.

```
gnuplot> f(x) = a*x**3 + b*x**2 + c*x + d
gnuplot> fit f(x) "tiempos_MM.dat" via a, b, c, d
iter      chisq       delta/lim  lambda   a             b             c             d
   0 1.5263864270e+20   0.00e+00  1.42e+09    1.000000e+00   1.000000e+00   1.000000e+00   1.000000e+00
   1 2.5745970196e+16  -5.93e+08  1.42e+08    1.239571e-02   9.994076e-01   9.999996e-01   1.000000e+00
   2 1.5541078083e+12  -1.66e+09  1.42e+07   -5.973249e-04   9.993224e-01   9.999995e-01   1.000000e+00
   3 1.5299132880e+12  -1.58e+03  1.42e+06   -5.944342e-04   9.916434e-01   9.999877e-01   1.000000e+00
   4 4.8659679089e+11  -2.14e+05  1.42e+05   -3.349932e-04   5.585771e-01   9.993197e-01   9.999991e-01
   5 7.9580684441e+07  -6.11e+08  1.42e+04   -3.720322e-06   5.606858e-03   9.984546e-01   9.999980e-01
   6 2.4527725697e+05  -3.23e+07  1.42e+03    5.630867e-07  -1.542378e-03   9.972217e-01   9.999933e-01
   7 1.9480075228e+05  -2.59e+04  1.42e+02    5.032259e-07  -1.375374e-03   8.883331e-01   9.995765e-01
   8 1.1891521707e+03  -1.63e+07  1.42e+01    4.604072e-08  -1.045885e-04   6.432743e-02   9.963887e-01
   9 7.4028282555e+01  -1.51e+06  1.42e+00    8.646007e-09  -6.490597e-07  -3.064551e-03   9.924997e-01
  10 7.3699685425e+01  -4.46e+02  1.42e-01    8.921884e-09  -1.588523e-06  -2.119250e-03   7.308941e-01
  11 7.3346135033e+01  -4.82e+02  1.42e-02    9.691846e-09  -4.162394e-06   3.941210e-04   7.370185e-02
  12 7.3345911770e+01  -3.04e-01  1.42e-03    9.711682e-09  -4.228703e-06   4.588710e-04   5.677112e-02
iter      chisq       delta/lim  lambda   a             b             c             d

After 12 iterations the fit converged.
final sum of squares of residuals : 73.3459
rel. change during last iteration : -3.04397e-06

degrees of freedom    (FIT_NDF)                       : 15
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 2.21127
variance of residuals (reduced chisquare) = WSSR/ndf   : 4.88973

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a               = 9.71168e-09      +/- 3.991e-09    (41.1%)
b               = -4.2287e-06      +/- 1.212e-05    (286.7%)
c               = 0.000458871      +/- 0.01057      (2305%)
d               = 0.0567711        +/- 2.506        (4415%)

correlation matrix of the fit parameters:
                a      b      c      d
a               1.000
b              -0.988  1.000
c               0.929 -0.974  1.000
d              -0.736  0.810 -0.906  1.000
gnuplot> plot f(x) with l title 'Aproximacion empirica', "tiempos_MM.dat"
```

Figura 13: Ajuste de la eficiencia empírica de la multipliación de matrices.

Por último, representamos los datos:

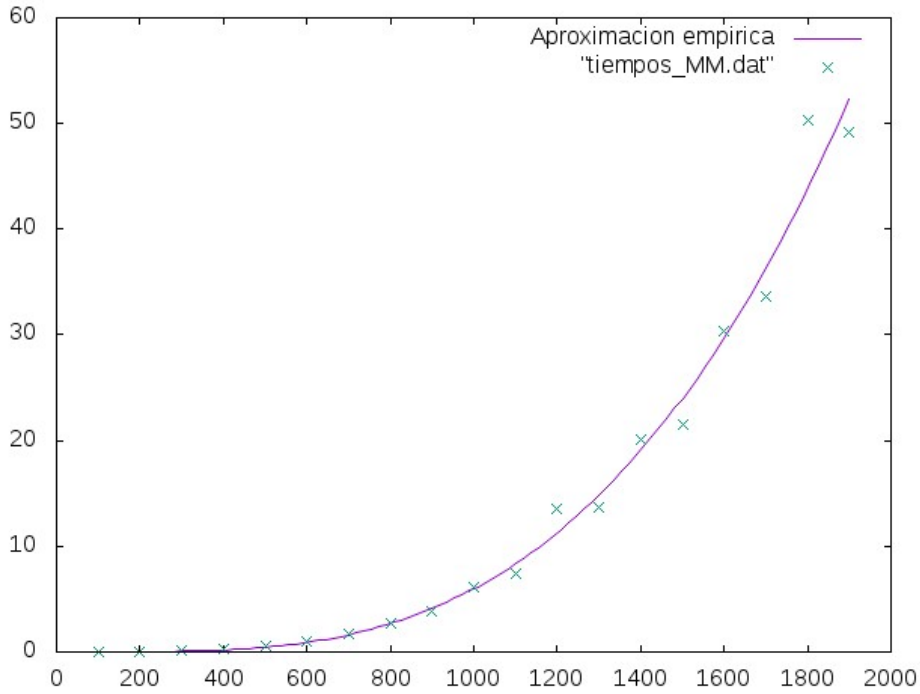

Figura 14: Eficiencia de la multiplicación de matrices.

2.8. Ejercicio 8: Ordenación por Mezcla

Estudie el código del algoritmo recursivo disponible en el fichero mergesort.cpp. En él, seintegran dos algoritmos de ordenación: inserción y mezcla (o mergesort). El parámetro UMBRAL_MS condiciona el tamaño mínimo del vector para utilizar el algoritmo de inserción en vez de seguir aplicando de forma recursiva el mergesort. Como ya habrá estudiado, la eficiencia teórica del mergesort es n log(n). Realice un análisis de la eficiencia empírica y haga el ajuste de ambas curvas. Incluya también, para este caso, un pequeño estudio de cómo afecta el parámetro UMBRAL_MS a la eficiencia del algoritmo .Para ello, pruebe distintos valores del mismo y analice los resultados obtenidos.

Solución:
Tomamos los datos y ajustamos la función $f(x) = a * x * log(x) + b * x + c$. (Aunque en primer lugar intenté ajustar los datos a $f(x) = a * x * log(b * x) + c * x + d$, los resultados dejaban mucho que desear).

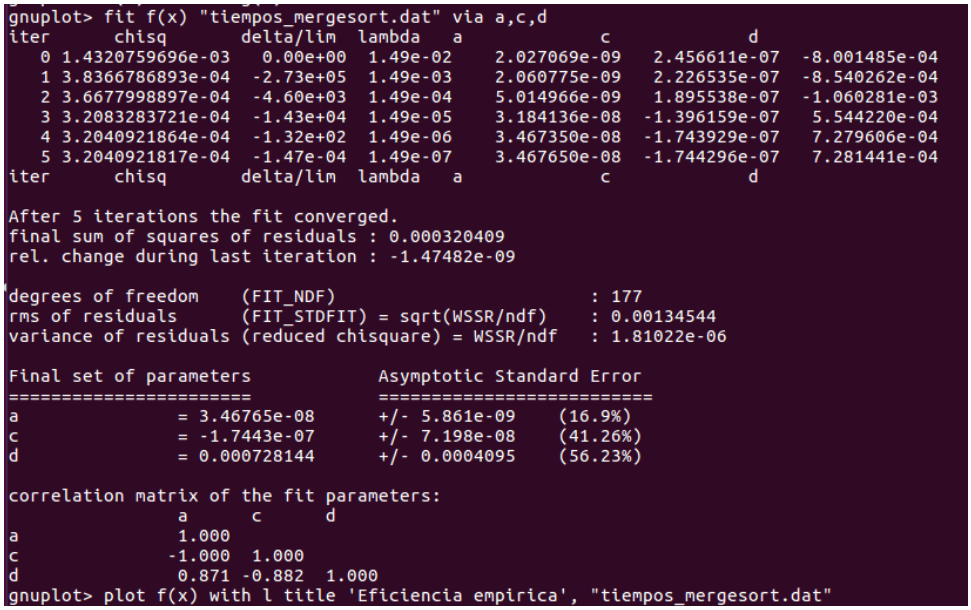

Figura 15: Influencia del proceso de compilación.

Nota: Aunque en esta imagen no se aprecia la forma de f(x) (me equivoqué al tomar la captura), podemos verlo en el archivo **fit.log(8_1)**:

```
Tue Sep 26 09:40:05 2017

FIT:    data read from "tiempos_mergesort.dat"
        format = z
        x range restricted to [-120000. : 180000.]
        #datapoints = 180
        residuals are weighted equally (unit weight)

function used for fitting: f(x)
f(x) = a*x*log(x) + c*x + d
fitted parameters initialized with current variable values

[...]

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a                = 3.46765e-08     +/- 5.861e-09    (16.9%)
c                = -1.7443e-07     +/- 7.198e-08    (41.26%)
d                = 0.000728144     +/- 0.0004095    (56.23%)
```

Vemos el resultado en la siguiente imagen:

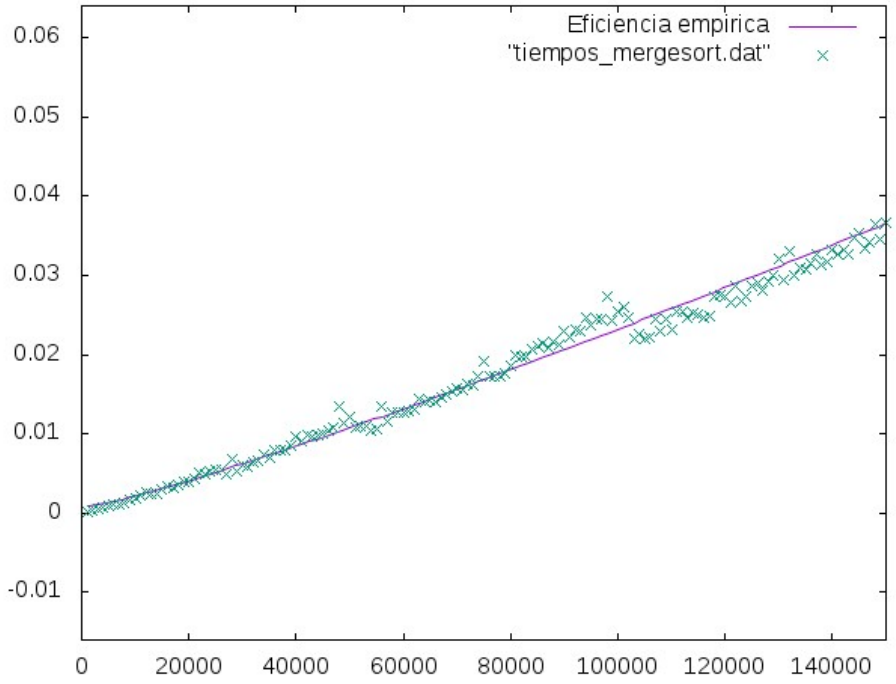

Figura 16: Eficiencia del algoritmo Mergesort

Veamos ahora como afecta el parámetro **UMBRAL_MS** a la eficiencia dándole distintos valores. He realizado dos estudios del algoritmo además del explicado arriba: uno para $UMBRAL\_MS = 10$ (umbral bajo) y otro para $UMBRAL\_MS = 1000$ (umbral alto).

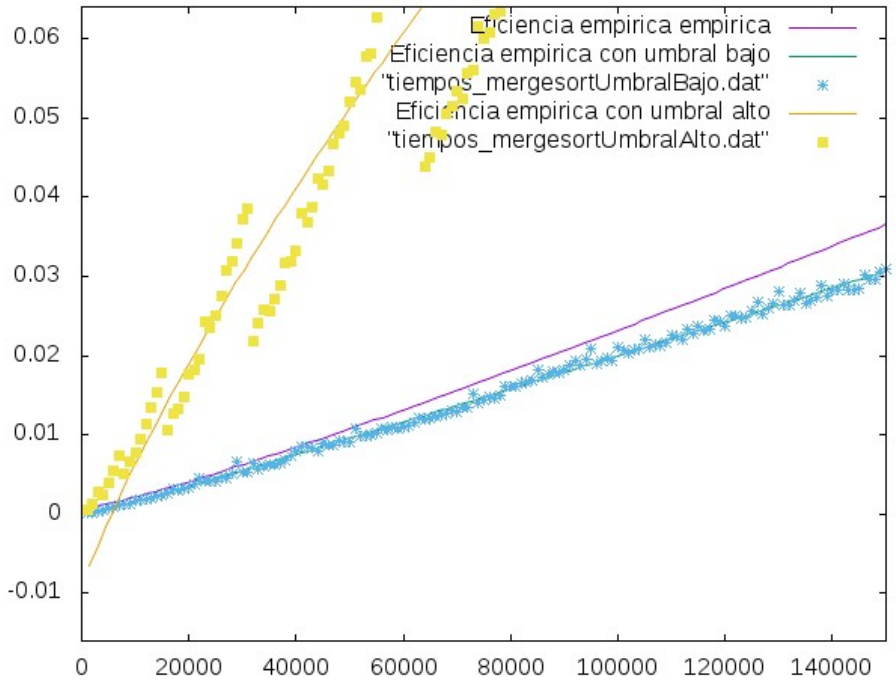

Figura 17: Influencia del parámetro **UMBRAL_MS**.

Vemos como para valor alto el tiempo se dispara, ya que estamos aplicando inserción más que mergesort, mientras la eficiencia aumenta ligeramente para un valor bajo del parámetro. Evidentemente esto se debe a que aplicamos mergesort a un mayor porcentaje del vector. Sin embargo esto produce un uso extra de la pila del programa ya que supone un mayor número llamadas recursivas.