



# UNIVERSIDAD DE GRANADA

Práctica 1: Eficiencia  
Estructura de datos

LINQI ZHU

## Práctica 1: Eficiencia

### Índice:

0. Características hardware y Sistema Operativo
1. Ejercicio 1: Ordenación de la burbuja
2. Ejercicio 2: Ajuste en la ordenación de la burbuja
3. Ejercicio 3: Problemas de precisión
4. Ejercicio 4: Dependencia de la implementación
5. Ejercicio 5: Mejor y peor caso
6. Ejercicio 6: Influencia del proceso de compilación
7. Ejercicio 7: Multiplicación matricial
8. Ejercicio 8: Ordenación por Mezcla

## **Características hardware y Sistema Operativo**

Para la realización de esta práctica, he utilizado un ordenador con un procesador de Intel Core i5 de 4 núcleos y una velocidad turbo de 2GHz.

Este ordenador cuenta con una RAM de 16GB.

El Sistema Operativo sobre el que trabajo es macOS Big Sur Versión 11.6

### **Información del hardware:**

Nombre del modelo:	MacBook Pro
Identificador del modelo:	MacBookPro16,2
Nombre del procesador:	Intel Core i5 de 4 núcleos
Velocidad del procesador:	2 GHz
Cantidad de procesadores:	1
Cantidad total de núcleos:	4
Caché de nivel 2 (por núcleo):	512 KB
Caché de nivel 3:	6 MB
Tecnología Hyper-Threading:	Activado
Memoria:	16 GB
Versión del firmware del sistema:	1554.140.20.0.0 (iBridge: 18.16.14759.0.1,0)
Número de serie (sistema):	C02CKGEFML7H
UUID de hardware:	4FD22559-5EB2-57A1-8AF2- 92516B64FAB6
Instalando el perfil de datos de UDID:	4FD22559-5EB2-57A1-8AF2- 92516B64FAB6
Estado del bloqueo de activación:	Activado

### **Información del software de sistema:**

Versión del sistema:	macOS 11.6 (20G165)
Versión del kernel:	Darwin 20.6.0
Volumen de arranque:	Macintosh HD
Modo de arranque:	Normal
Nombre de usuario:	linqi zhu (z07)
Memoria virtual segura:	Activado
Protección de la integridad del sistema:	Activado
Tiempo desde el arranque:	17 días1:54

## Ejercicio 1: Ordenación de la burbuja (nivel de dificultad: 3)

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]) {
                swap(v[j],v[j+1]);
                //alternativa al swap
                /*int aux = v[j];//incluir algoritmo
                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?

## Código fuente

### *busqueda\_burbuja.cpp*

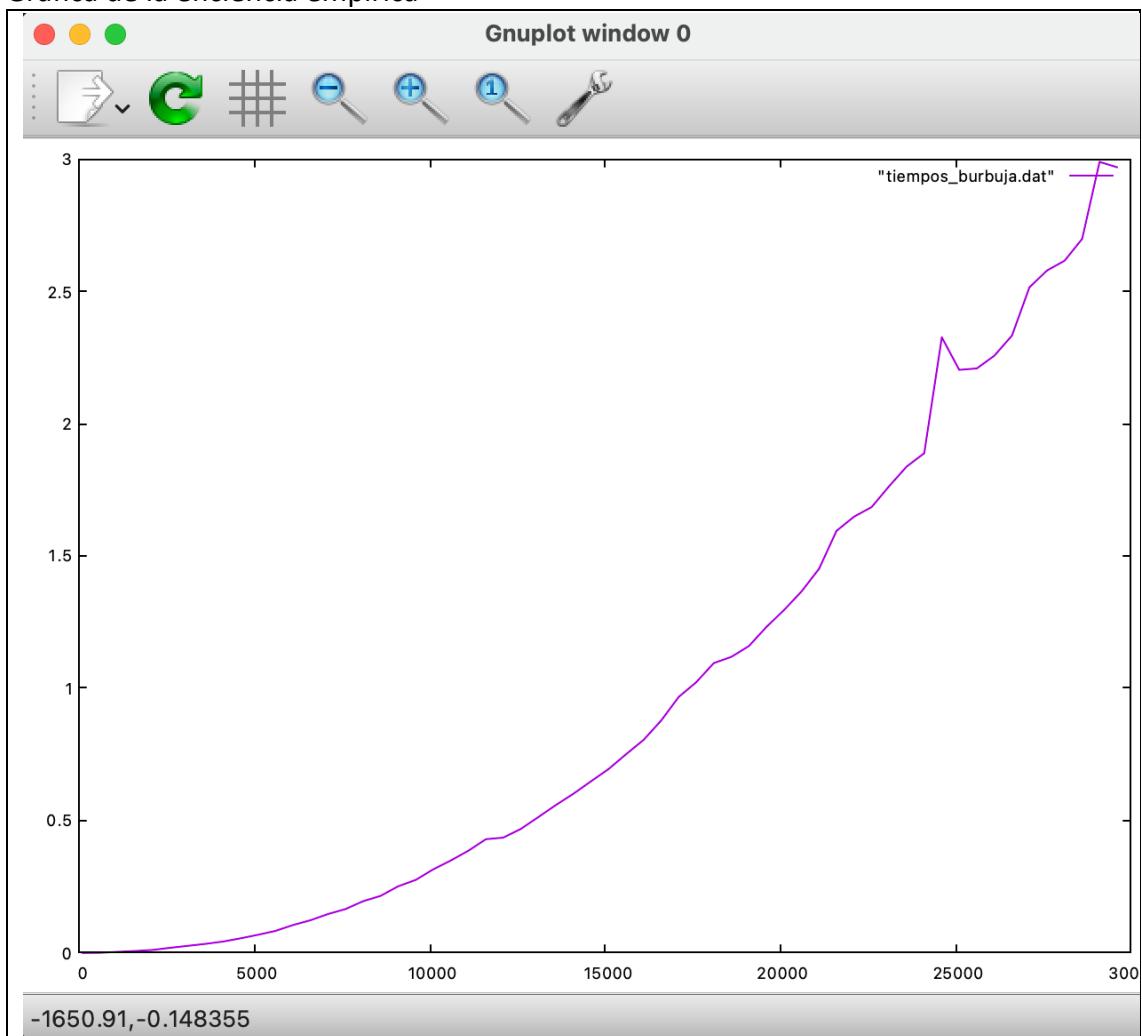
```
1 #include <iostream>
2 #include <ctime> // Recursos para medir tiempos
3 #include <cstdlib> // Para generación de números pseudoaleatorios
4
5 using namespace std;
6
7 void ordenar(int *v, int n)
8 {
9     for (int i = 0; i < n - 1; i++)
10        for (int j = 0; j < n - i - 1; j++)
11            if (v[j] > v[j + 1])
12            {
13                swap(v[j], v[j + 1]);
14                //alternativa al swap
15                /*int aux = v[j];//incluir algorithm v[j] = v[j+1];
16                v[j+1] = aux;*/
17            }
18    }
19
20 void sintaxis()
21 {
22     cerr << "Sintaxis:" << endl;
23     cerr << " TAM: Tamaño del vector (>0)" << endl;
24     cerr << " VMAX: Valor máximo (>0)" << endl;
25     cerr << "Se genera un vector de tamaño TAM con elementos aleatorios en [0,VMAX[" <<
26         endl;
27     exit(EXIT_FAILURE);
28 }
29
30 int main(int argc, char *argv[])
31 {
32     // Lectura de parámetros
33     if (argc != 3)
34         sintaxis();
35     int tam = atoi(argv[1]); // Tamaño del vector
36     int vmax = atoi(argv[2]); // Valor máximo
37     if (tam <= 0 || vmax <= 0)
38         sintaxis();
39
40     // Generación del vector aleatorio
41     int *v = new int[tam]; // Reserva de memoria
42     srand(time(0)); // Inicialización del generador de números
43     pseudoaleatorios
44     for (int i = 0; i < tam; i++) // Recorrer vector
45         v[i] = rand() % vmax; // Generar aleatorio [0,vmax[
46
47     clock_t tini; // Anotamos el tiempo de inicio
48     tini = clock();
49
50     // int x = vmax + 1; // Buscamos un valor que no está en el vector
51     ordenar(v, tam); // de esta forma forzamos el peor caso
52
53     clock_t tfin; // Anotamos el tiempo de finalización
54     tfin = clock();
55
56     // Mostramos resultados
57     cout << tam << "\t" << (tfin - tini) / (double)CLOCKS_PER_SEC << endl;
58     delete[] v; // Liberamos memoria dinámica
59 }
```

### *ejecuciones\_burbuja.csh*

```
1 #!/bin/csh
2 @ inicio = 100
3 @ fin = 30000
4 @ incremento = 500
5 set ejecutable = busqueda_burbuja
6 set salida = tiempos_burbuja.dat
7
8 @ i = $inicio
9 echo > $salida
10 while ( $i <= $fin )
11   echo Ejecución tam = $i
12   echo `./{$ejecutable} $i 300` >> $salida
13   @ i += $incremento
14 end
```

Grafica obtenido con gnuplot:

Grafica de la eficiencia empírica



## Calculo de la Eficiencia Teoría:

El tiempo de ejecución en el peor de los casos:

$$4 + \left( \sum_{i=0}^{n-2} 5 + \left( \sum_{j=0}^{n-i-2} 4 + 3 + 4 + 4 + 2 \right) + 2 \right) = 4 + \left( \sum_{i=0}^{n-2} 7 + \left( \sum_{j=0}^{n-i-2} 16 \right) \right)$$

$$\sum_{i=0}^{n-2} 7 = 7(n-1)$$

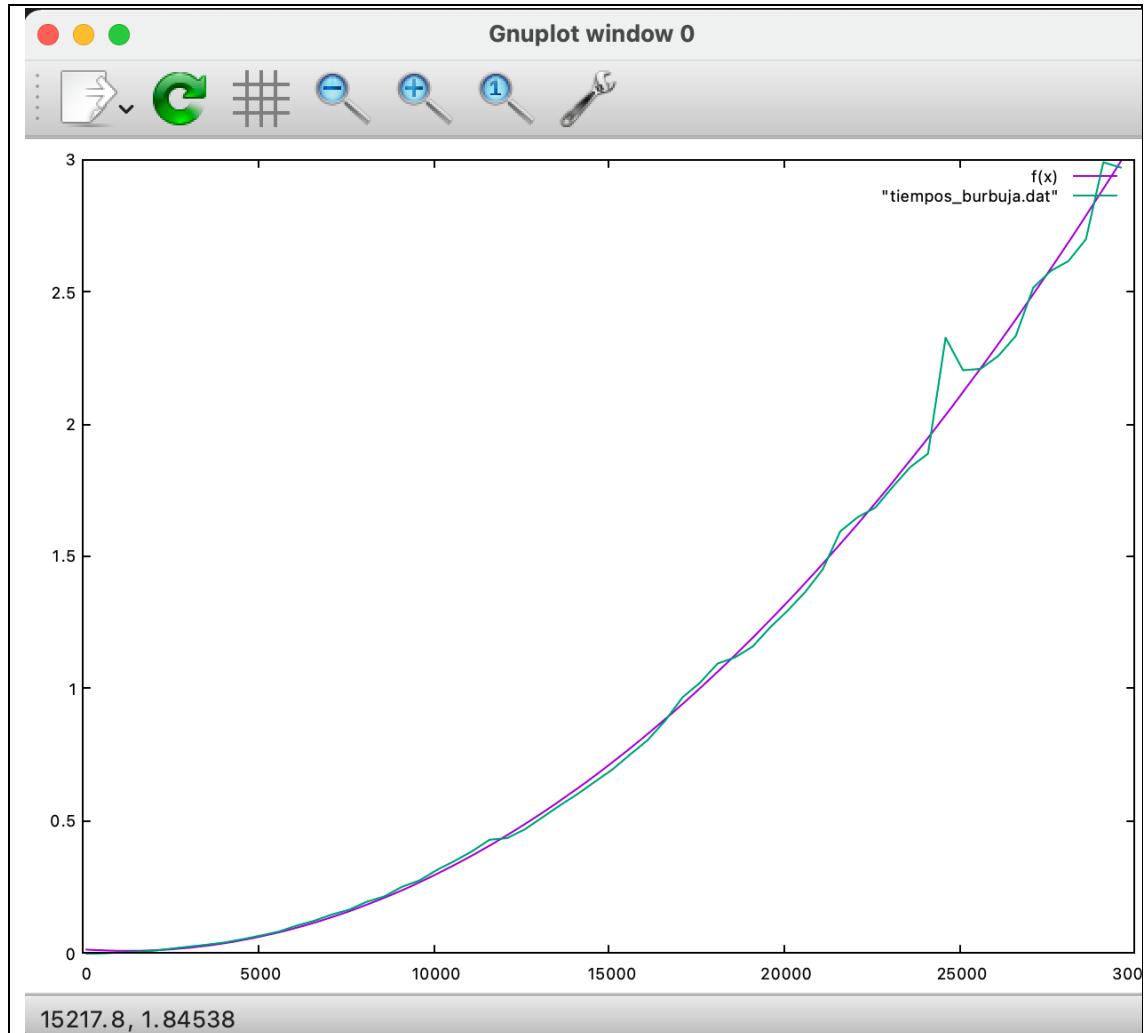
$$\sum_{i=0}^{n-2} \sum_{j=0}^{n-i-2} 16 = \sum_{i=0}^{n-2} \left( 16 \sum_{j=0}^{n-i-2} 1 \right) = \sum_{i=0}^{n-2} 16(n-i-1) = 16 \left( \sum_{i=0}^{n-2} n - \sum_{i=0}^{n-2} i - \sum_{i=0}^{n-2} 1 \right) = 16n(n-1) - 8(n-1)(n-2) - 16(n-1)$$

$$4 + 7(n-1) + 16n(n-1) - 8(n-1)(n-2) - 16(n-1) \quad \rightarrow \quad a^*n^2 + b^*n + c$$

La eficiencia teórica tiene un orden  $O(n^2)$ .

Grafica con las dos funciones:

Grafica de las dos eficiencias



Podemos que la grafica de la ordenación de burbuja se ajusta muy bien.

## Ejercicio 2: Ajuste en la ordenación de la burbuja(nivel de dificultad: 3)

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  $ax^2+bx+c$ .

En la ultima imagen del ejercicio anterior podemos ver que se ajusta muy bien la ordenación de la burbuja.

(Captura de las constantes ocultas de ajustar  $f(x)$  a los resultados obtenido)

```
gnuplot> f(x)=a*x**2+b*x+c
gnuplot> fit f(x) "tiempos_burbuja.dat" via a,b,c
iter      chisq      delta/lim   lambda    a          b          c
  0  5.9406433956e+10  0.00e+00  2.37e+04  1.009266e-04 -5.242409e-01  1.000000e+00
  1  1.8089465132e+08 -3.27e+07  2.37e+03  1.731337e-05 -4.017177e-01  1.000090e+00
  2  6.4761359657e+04 -2.79e+08  2.37e+02  3.285635e-07 -7.786350e-03  1.000288e+00
  3  6.8637082820e+00 -9.43e+08  2.37e+01  7.456177e-09 -1.426823e-04  1.000170e+00
  4  6.6989400610e+00 -2.46e+03  2.37e+00  7.349159e-09 -1.395969e-04  9.882040e-01
  5  1.4606830421e+00 -3.59e+05  2.37e-01  5.341573e-09 -6.769504e-05  4.517207e-01
  6  1.4300959724e-01 -9.21e+05  2.37e-02  3.723203e-09 -9.732972e-06  1.924681e-02
  7  1.4292396382e-01 -5.99e+01  2.37e-03  3.710052e-09 -9.261965e-06  1.573247e-02
  8  1.4292396382e-01 -3.96e-07  2.37e-04  3.710051e-09 -9.261926e-06  1.573219e-02
iter      chisq      delta/lim   lambda    a          b          c
After 8 iterations the fit converged.
final sum of squares of residuals : 0.142924
rel. change during last iteration : -3.95621e-12

degrees of freedom      (FIT_NDF) : 57
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf) : 0.0500743
variance of residuals (reduced chisquare) = WSSR/ndf : 0.00250744

Final set of parameters            Asymptotic Standard Error
=====
a          = 3.71005e-09      +/- 9.644e-11    (2.599%)
b          = -9.26193e-06     +/- 2.96e-06    (31.96%)
c          = 0.0157322        +/- 0.01902     (120.9%)

correlation matrix of the fit parameters:
      a      b      c
a  1.000
b -0.968  1.000
c  0.738 -0.861  1.000
```

### Ejercicio 3: Problemas de precisión (nivel de dificultad: 5)

Junto con este guión se le ha suministrado un fichero ejercicio\_desc.cpp y ejercicio3.cpp En él se ha implementado un algoritmo. La diferencia entre estos dos ficheros es que uno usa clock() para medir el tiempo (ejercicio\_desc.cpp) y ejercicio3.cpp usa la librería chrono.

Se pide que:

- Explique qué hace este algoritmo.
- Calcule su eficiencia teórica.
- Calcule su eficiencia empírica. Para ello tiene a su disposición la macro ejecuciones\_ejercicio3.csh. Los incrementos nos son lineales sino potencias de 2. Para ver la necesidad de estos incrementos obtenga los tiempos usando incrementos lineales como se hace en la macro ejecuciones\_blineal.csh

Si visualiza la eficiencia empírica (con incrementos lineales) debería notar algo anormal. Explíquelo la solución. Realice la regresión para ajustar la curva teórica a la empírica.

```
7 int operacion(int *v, int n, int x, int inf, int sup)
8 {
9     int med;
10    bool enc = false;
11    while ((inf < sup) && (!enc))
12    {
13        med = (inf + sup) / 2;
14        if (v[med] == x)
15            enc = true;
16        else if (v[med] < x)
17            inf = med + 1;
18        else
19            sup = med - 1;
20    }
21    if (enc)
22        return med;
23    else
24        return -1;
25 }
```

El algoritmo es el método de búsqueda binaria, que realiza la búsqueda de un elemento x que le pasamos como argumento en un vector ordenado.

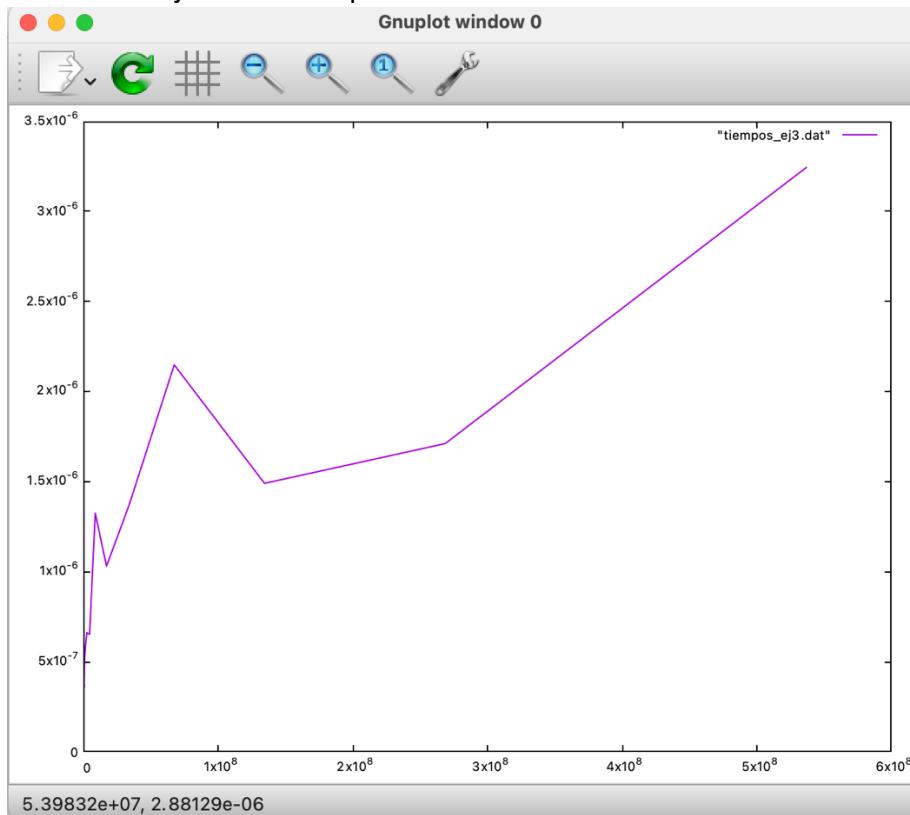
Calculo de eficiencia Teórica:

$$1+2+3+\sum_{i=0}^{\log(n)} (3+4)+1+1 = 8 + \sum_{i=0}^{\log(n)} 7 = 8 + 7(\log(n)+1) = 7\log(n)+15 \rightarrow O(\log(n))$$

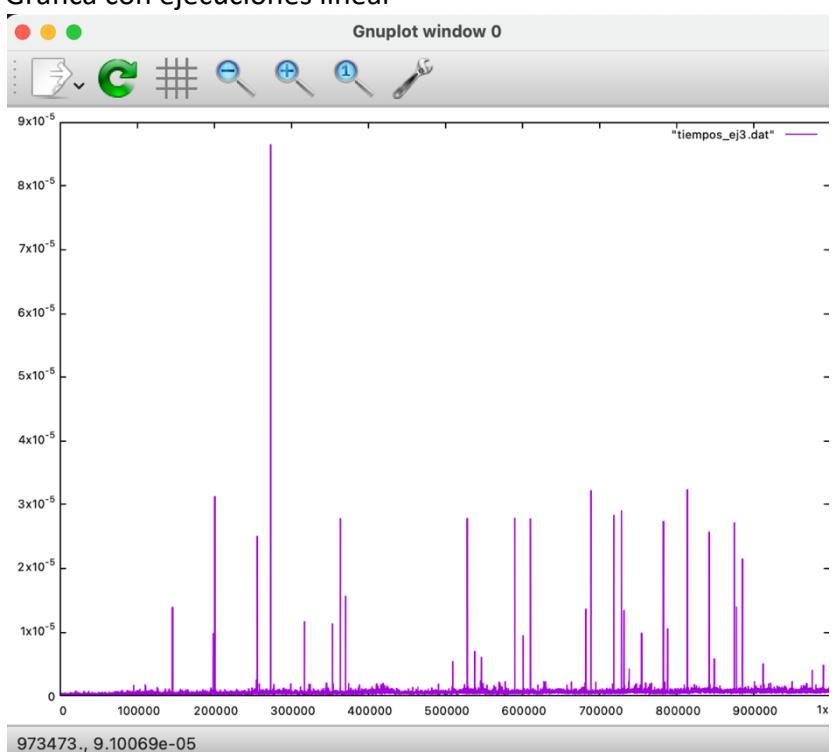
La eficiencia teórica es de orden  $O(\log(n))$ .

Calculo de eficiencia empírica:

Grafica con ejecuciones exponenciales

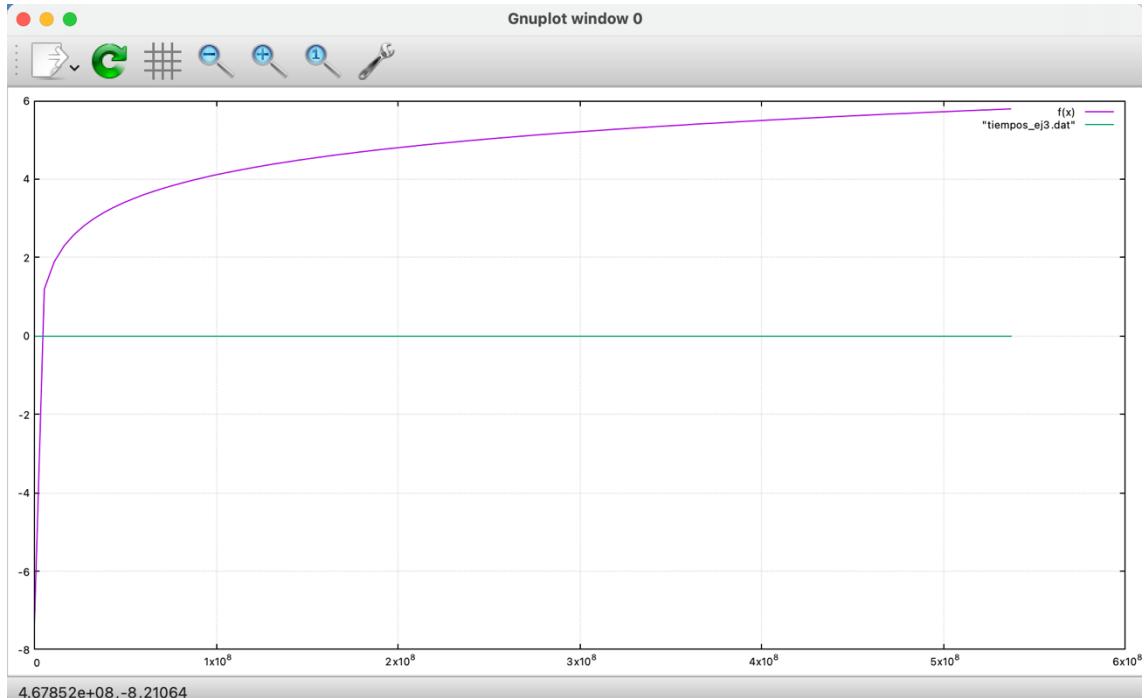


Grafica con ejecuciones lineal



Si visualizamos la gráfica con ejecución lineales, se puede apreciar que los más alto, si lo unimos pueden formar una línea como si fuera una función logarítmica.

Gráfica de ajuste:



Códigos fuente:

*ejecuciones\_ejercicio3.csh*

```

1 #!/bin/csh
2 @ inicio = 1024
3 @ fin = 1000000000
4
5 alias MATH 'set \!:1 = `echo "\!:3-$" | bc -l`'
6 set ejecutable = ejercicio3
7 set salida = tiempos_ej3.dat
8 @ factor = 4
9 @ i = $inicio
10
11 echo > $salida
12 while ( $i <= $fin )
13     echo Ejecución tam = $i
14     echo `./{$ejecutable} $i 10000` >> $salida
15
16 if ($i >= 65536) then
17     @ factor = 2
18 endif
19 MATH nuevo = $i*$factor
20
21 @ i = $nuevo
22
23 end

```

### ejercicio3.cpp

```
1 #include <iostream>
2 #include <cstdlib> // Para generación de números pseudoaleatorios
3 #include <chrono> // Recursos para medir tiempos
4 using namespace std;
5 using namespace std::chrono;
6
7 int operacion(int *v, int n, int x, int inf, int sup)
8 {
9     int med;
10    bool enc = false;
11    while ((inf < sup) && (!enc))
12    {
13        med = (inf + sup) / 2;
14        if (v[med] == x)
15            enc = true;
16        else if (v[med] < x)
17            inf = med + 1;
18        else
19            sup = med - 1;
20    }
21    if (enc)
22        return med;
23    else
24        return -1;
25 }
26 void sintaxis()
27 {
28     cerr << "Sintaxis:" << endl;
29     cerr << " TAM: Tamaño del vector (>0)" << endl;
30     cerr << " VMAX: Valor máximo (>0)" << endl;
31     cerr << "Se genera un vector de tamaño TAM con elementos aleatorios en [0,VMAX[" << endl;
32     exit(EXIT_FAILURE);
33 }
34
35 int main(int argc, char *argv[])
36 {
37     // Lectura de parámetros
38     if (argc != 3)
39         sintaxis();
40     int tam = atoi(argv[1]); // Tamaño del vector
41     int vmax = atoi(argv[2]); // Valor máximo
42     if (tam <= 0 || vmax <= 0)
43         sintaxis();
44
45     // Generación del vector aleatorio
46     int *v = new int[tam]; // Reserva de memoria
47     srand(time(0)); // Inicialización del generador de números pseudoaleatorios
48     for (int i = 0; i < tam; i++) // Recorrer vector
49         v[i] = rand() % vmax; // Generar aleatorio [0,vmax[
50
51     high_resolution_clock::time_point start, //punto de inicio
52                                     end; //punto de fin
53     duration<double> tiempo_transcurrido; //objeto para medir la duracion de end
54                                     // y start
55
56     start = high_resolution_clock::now(); //iniciamos el punto de inicio
57
58     operacion(v, tam, vmax + 1, 0, tam - 1);
59     end = high_resolution_clock::now(); //anotamos el punto de de fin
60     //el tiempo transcurrido es
61     tiempo_transcurrido = (duration_cast<duration<double> >(end - start));
62
63     // Mostramos resultados
64     cout << tam << "\t" << tiempo_transcurrido.count() << endl;
65
66     delete[] v; // Liberamos memoria dinámica
67 }
```

*ejecuciones\_lineal\_eje3.csh*

```
1  #!/bin/csh
2  @ inicio = 1024
3  @ fin = 1000000
4  @ incremento = 1024
5  set ejecutable = ejercicio3
6  set salida = tiempos_ej3_lineal.dat
7
8  @ i = $inicio
9  echo > $salida
10 while ( $i <= $fin )
11   echo Ejecución tam = $i
12   echo `./{$ejecutable} $i 1000` >> $salida
13   @ i += $incremento
14 end
```

## Ejercicio 4: Dependencia de la implementación(nivel de dificultad: 5)

Considere esta otra implementación del algoritmo de la burbuja:

```
void ordenar(int *v, int n) {
    bool cambio=true;
    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;
                swap (v[j],v[j+1]); //incluir algorithm
            }
    }
}
```

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.

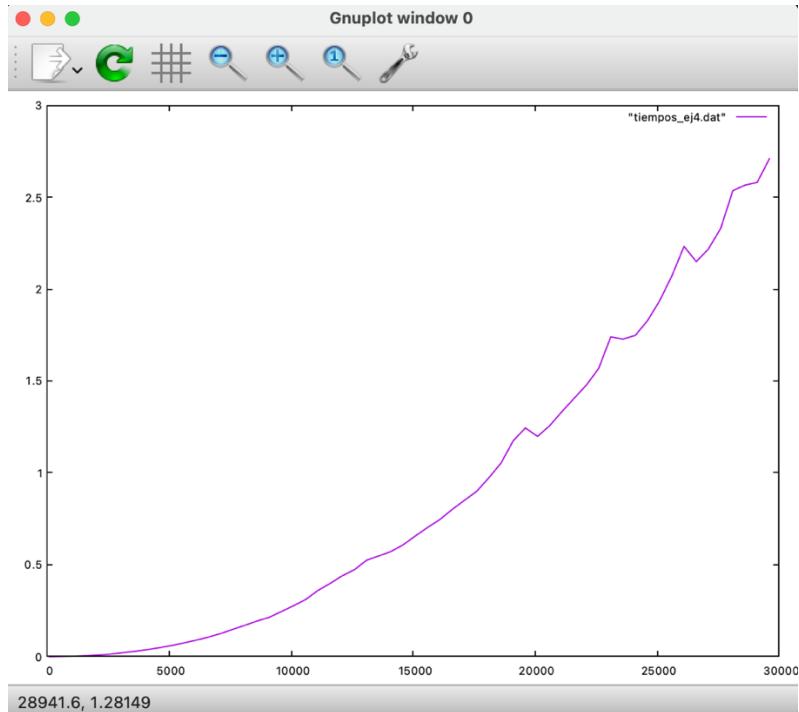
Calculo de Eficiencia Teórica:

$$2+4+1+5+\left(\sum_{j=0}^{n-2} 4+2\right) +3 = 15 + 6(n-1) \rightarrow a*n+b$$

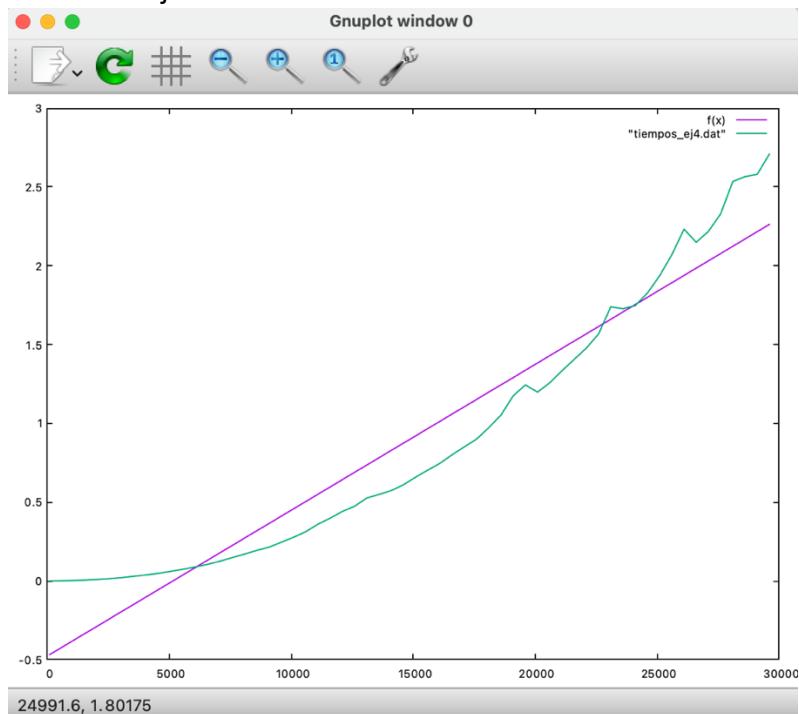
La eficiencia teórica de este algoritmo es  $O(n)$ .

Calculo de Eficiencia empírica:

Grafica de la eficiencia empírica



Grafica de ajuste:



Podemos que el ajuste a una función “ $f(x)=a*x+b$ ”, no se desvía mucho.

## Ejercicio 5: Mejor y peor caso (nivel de dificultad: 5)

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:

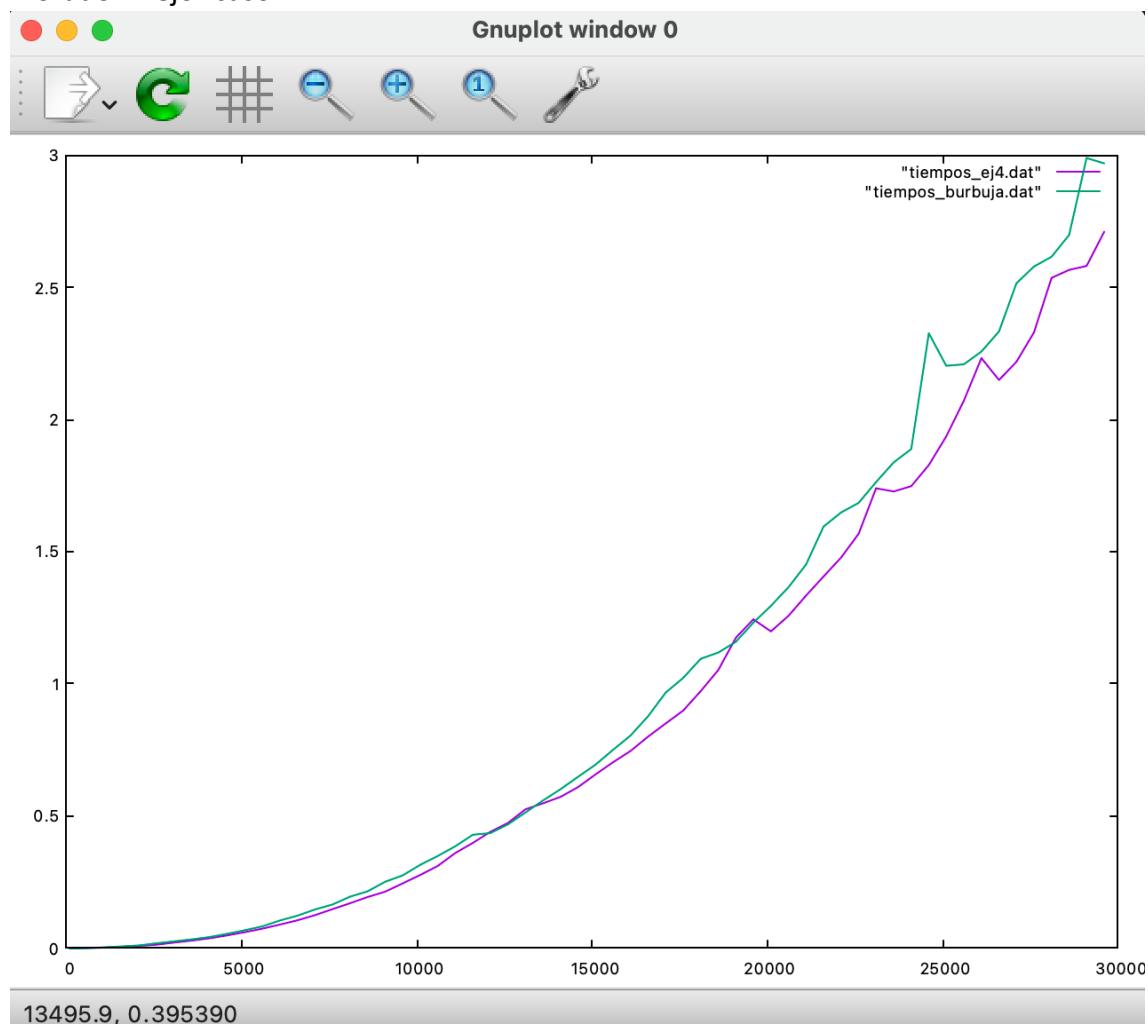
- El mejor caso posible. Para este algoritmo, si la entrada es un vector que ya está ordenado el tiempo de cómputo es menor ya que no tiene que intercambiar ningún elemento.
- El peor caso posible. Si la entrada es un vector ordenado en orden inverso estaremos en la peor situación posible ya que en cada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

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

Grafica de las eficiencias empíricas:

Verde-peor caso

Morado- mejor caso



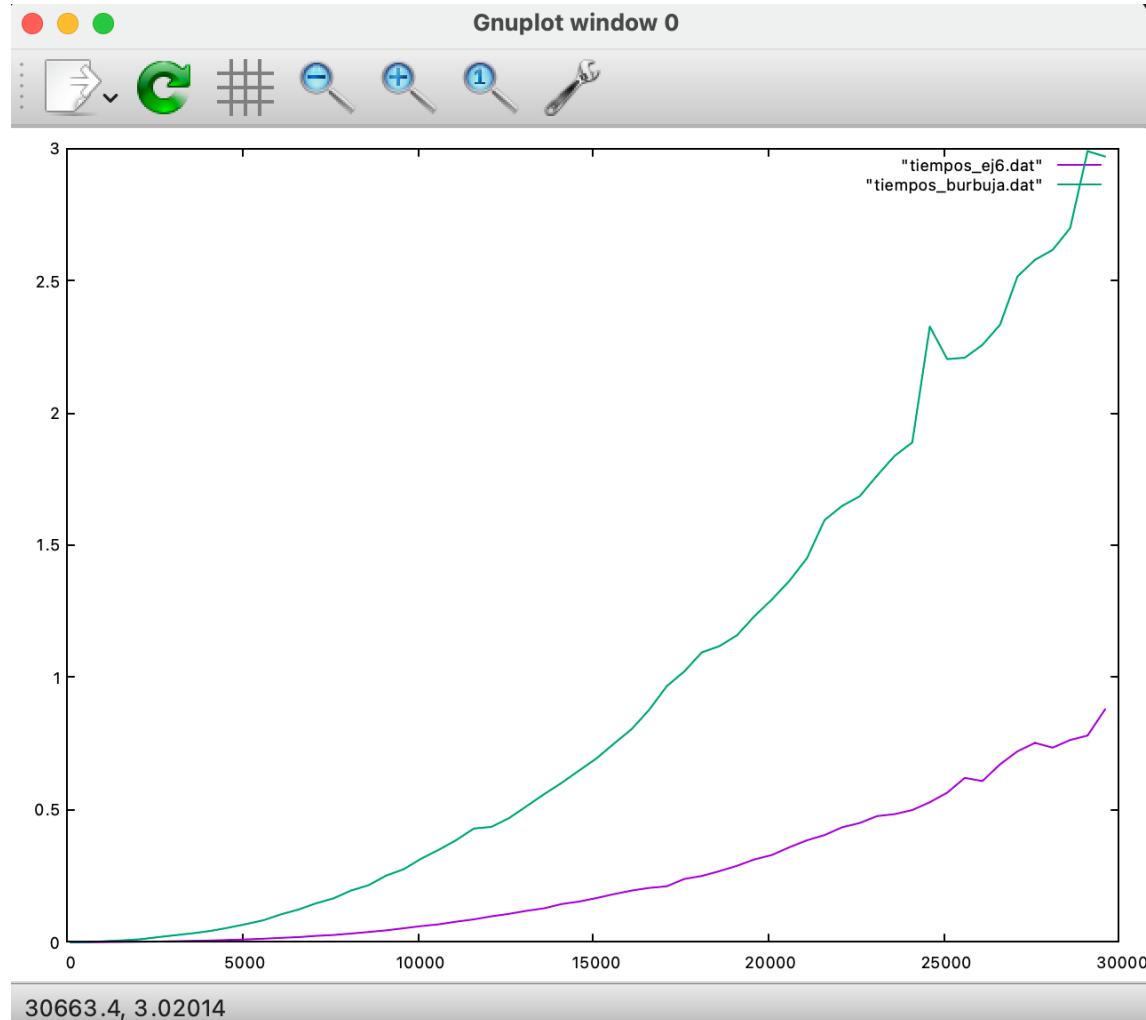
Podemos ver que la función de peor caso consume menos tiempo.

## Ejercicio 6: Influencia del proceso de compilación (nivel de dificultad: 5)

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.



Podemos ver que si optimizado a la hora de compilar, consumirá mucho menos tiempo.

## Ejercicio 7: Multiplicación matricial. (nivel de dificultad: 7)

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.

Calculo de eficiencia teórica de la función:

```
19 void multiplicar(int **m1, int **m2, int rows, int cols)
20 {
21     int **producto_mat = new int *[rows];
22     for (int i = 0; i < rows; ++i)
23         producto_mat[i] = new int[cols];
24     int suma, producto;
25     for (int y = 0; y < rows; y++)
26     {
27         for (int z = 0; z < cols; z++)
28         {
29             suma = 0;
30             for (int k = 0; k < rows; k++)
31             {
32                 producto = m1[y][k] * m2[k][z];
33                 suma = suma + producto;
34             }
35             producto_mat[y][z] = suma;
36         }
37     }
38 }
```

La eficiencia teórica tiene un orden  $O(n^3)$ .

## Código fuente: Ejercicio7.cpp

```
1 #include <iostream>
2 #include <stdlib.h>
3 #include <time.h>
4
5 using namespace std;
6
7 void generarArreglo(int **matriz, int rows, int cols, int valor)
8 {
9     srand(time(NULL));
10    for (int i = 0; i < rows; i++)
11    {
12        for (int j = 0; j < cols; j++)
13        {
14            matriz[i][j] = rand() % valor;
15        }
16    }
17 }
18
19 void multiplicar(int **m1, int **m2, int rows, int cols)
20 {
21     int **producto_mat = new int *[rows];
22     for (int i = 0; i < rows; ++i)
23         producto_mat[i] = new int[cols];
24     int suma, producto;
25     for (int y = 0; y < rows; y++)
26     {
27         for (int z = 0; z < cols; z++)
28         {
29             suma = 0;
30             for (int k = 0; k < rows; k++)
31             {
32                 producto = m1[y][k] * m2[k][z];
33                 suma = suma + producto;
34             }
35             producto_mat[y][z] = suma;
36         }
37     }
38 }
39
40 void sintaxis()
41 {
42     cerr << "Sintaxis:" << endl;
43     cerr << " TAM: Tamaño del vector (>0)" << endl;
44     cerr << " VMAX: Valor máximo (>0)" << endl;
45     cerr << "Genera un vector de TAM números aleatorios en [0,VMAX[" << endl;
46     exit(EXIT_FAILURE);
47 }
48
49 int main(int argc, char *argv[])
50 {
51     if (argc != 3) // Lectura de parámetros
52         sintaxis();
53     int tam = atoi(argv[1]); // Tamaño de las matrices
54     int vmax = atoi(argv[2]); // Valor máximo
55
56     if (tam <= 0 || vmax <= 0)
57
58         sintaxis();
59     int rows, cols;
60     rows = cols = tam;
61
62     int **matriz1 = new int *[rows];
63     for (int i = 0; i < rows; ++i)
64         matriz1[i] = new int[cols];
65     generarArreglo(matriz1, rows, cols, vmax);
66
67     int **matriz2 = new int *[rows];
68     for (int i = 0; i < rows; ++i)
69         matriz2[i] = new int[cols];
70     generarArreglo(matriz2, rows, cols, vmax);
71
72     clock_t tini; // Anotamos el tiempo de inicio
73     tini = clock();
74
75     multiplicar(matriz1, matriz2, rows, cols);
```

```

76     clock_t tfin; // Anotamos el tiempo de finalización
77     tfin = clock(); // Mostramos resultados (Tamaño del vector y tiempo de
78         // ejecución en seg.)
79     cout << tam << "\t" << (tfin - tini) / (double)CLOCKS_PER_SEC << endl;
80     return 0;
81 }

```

### Ejercicios\_ej7.csh

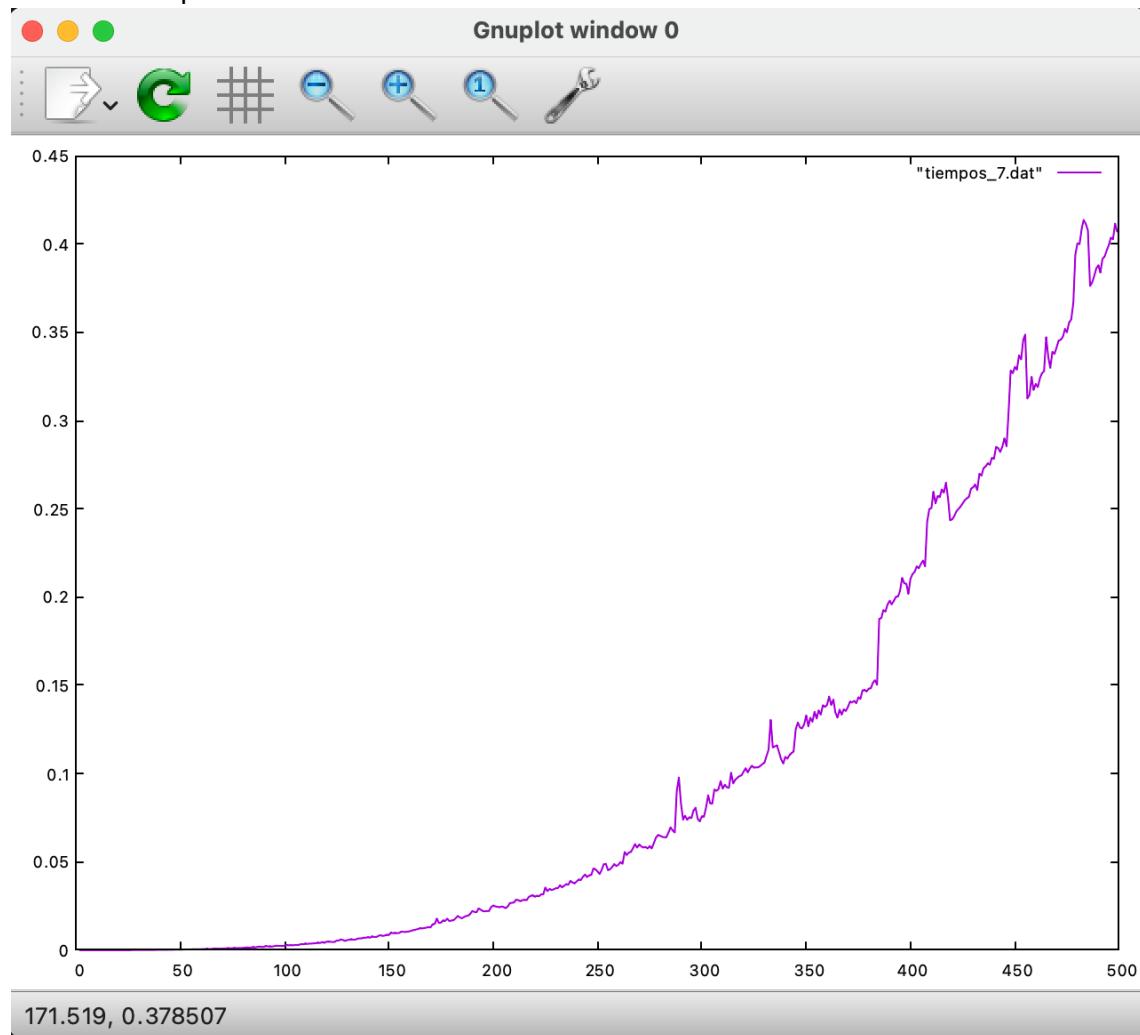
```

1 #!/bin/csh
2 @ inicio = 2
3 @ fin = 500
4 @ incremento = 1
5
6 set ejecutable = ejercicio7
7 set salida = tiempos_7.dat
8 @ i = $inicio
9
10 echo > $salida
11
12 while ($i < $fin)
13     echo Ejecución tam = $i
14     echo `./{$ejecutable} $i 100` >> $salida
15     @ i += $incremento
16 end

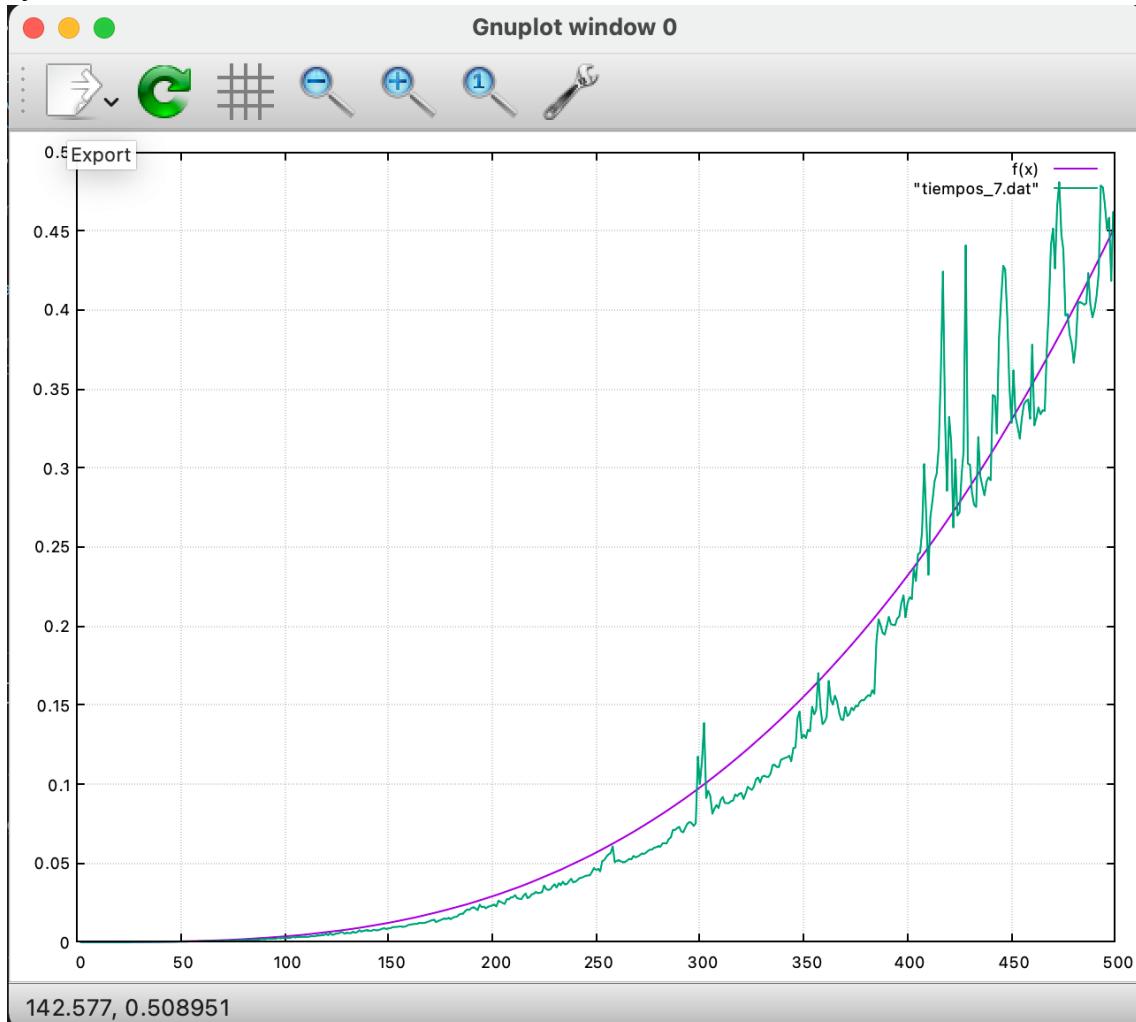
```

Grafica obtenido con gnuplot:

Eficiencia empírica:



### Ajuste con la eficiencia teórica:



```

gnuplot> f(x)=a*x**3
gnuplot> fit f(x) "tiempos_7.dat" via a
iter      chisq      delta/lim   lambda   a
  0 2.7500712881e-01  0.00e+00  1.71e-01  3.625077e-09
  1 2.7500712881e-01  0.00e+00  1.71e-02  3.625077e-09
iter      chisq      delta/lim   lambda   a

After 1 iterations the fit converged.
final sum of squares of residuals : 0.275007
rel. change during last iteration : 0

degrees of freedom      (FIT_NDF)            : 497
rms of residuals        (FIT_STDFIT) = sqrt(WSSR/ndf)    : 0.0235231
variance of residuals (reduced chisquare) = WSSR/ndf   : 0.000553334

Final set of parameters            Asymptotic Standard Error
=====      =====
a      = 3.62508e-09      +/- 2.234e-11    (0.6164%)
gnuplot> plot f(x) w l,"tiempos_7.dat" w l

```

## Ejercicio 8: Ordenación por mezcla(nivel de dificultad: 7)

Estudie el código del algoritmo recursivo disponible en el fichero mergesort.cpp. En él, se integran 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.

Eficiencia de teoría del Inserción:

```
103 inline static void insercion(int T[], int num_elem)
104 {
105     insercion_lims(T, 0, num_elem);
106 }
107
108 static void insercion_lims(int T[], int inicial, int final)
109 {
110     int i, j;
111     int aux;
112     for (i = inicial + 1; i < final; i++)
113     {
114         j = i;
115         while ((T[j] < T[j - 1]) && (j > 0))
116         {
117             aux = T[j];
118             T[j] = T[j - 1];
119             T[j - 1] = aux;
120             j--;
121         };
122     };
123 }
```

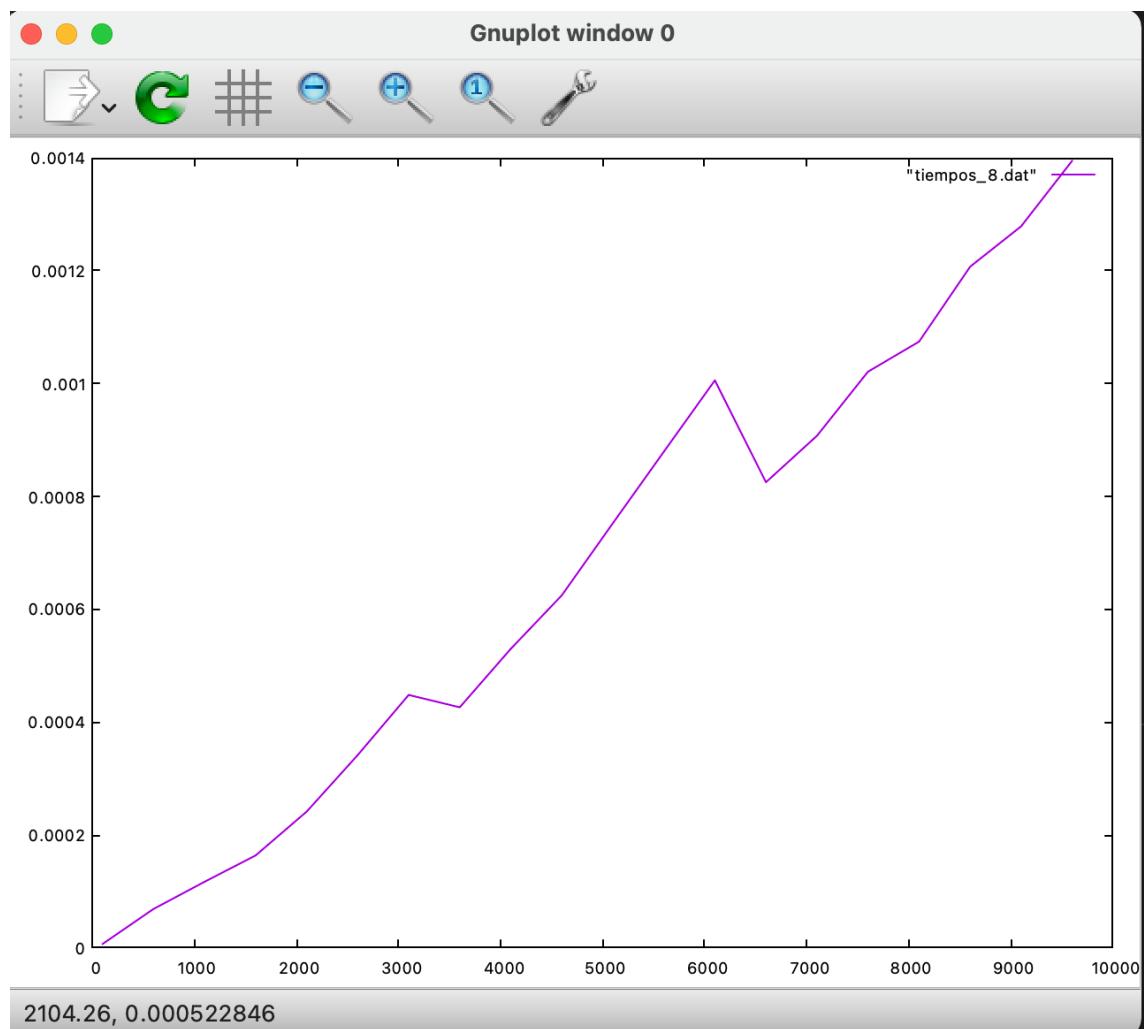
$$\begin{aligned} & 1 + 3 + \sum_{i=1}^n \left( 1 + \sum_{j=1}^0 (2 + 4 + 3 + 1) + 2 \right) = \\ & = 4 + \sum_{i=1}^n (3 + \sum_{j=1}^0 10) = 4 + \sum_{i=1}^n (3 + 10 \sum_{j=1}^0 1) = \\ & = 4 + \sum_{i=1}^n (3 + 10n) = 4 + \sum_{i=1}^n 3 + \sum_{i=1}^n 10n = \\ & = 4 + 3n + 10n^2 \end{aligned}$$

$O(n^2)$

### ejercicio8.csh

```
1  #!/bin/csh
2  @ inicio = 100
3  @ fin = 10000
4  @ incremento = 500
5  set ejecutable = ejercicio8
6  set salida = tiempos_8.dat
7
8  @ i = $inicio
9  echo > $salida
10 while ( $i <= $fin )
11   echo Ejecución tam = $i
12   echo `./{$ejecutable} $i` >> $salida
13   @ i += $incremento
14 end
```

Grafica de Inserción:



### Ajuste con la eficiencia teórica:

```

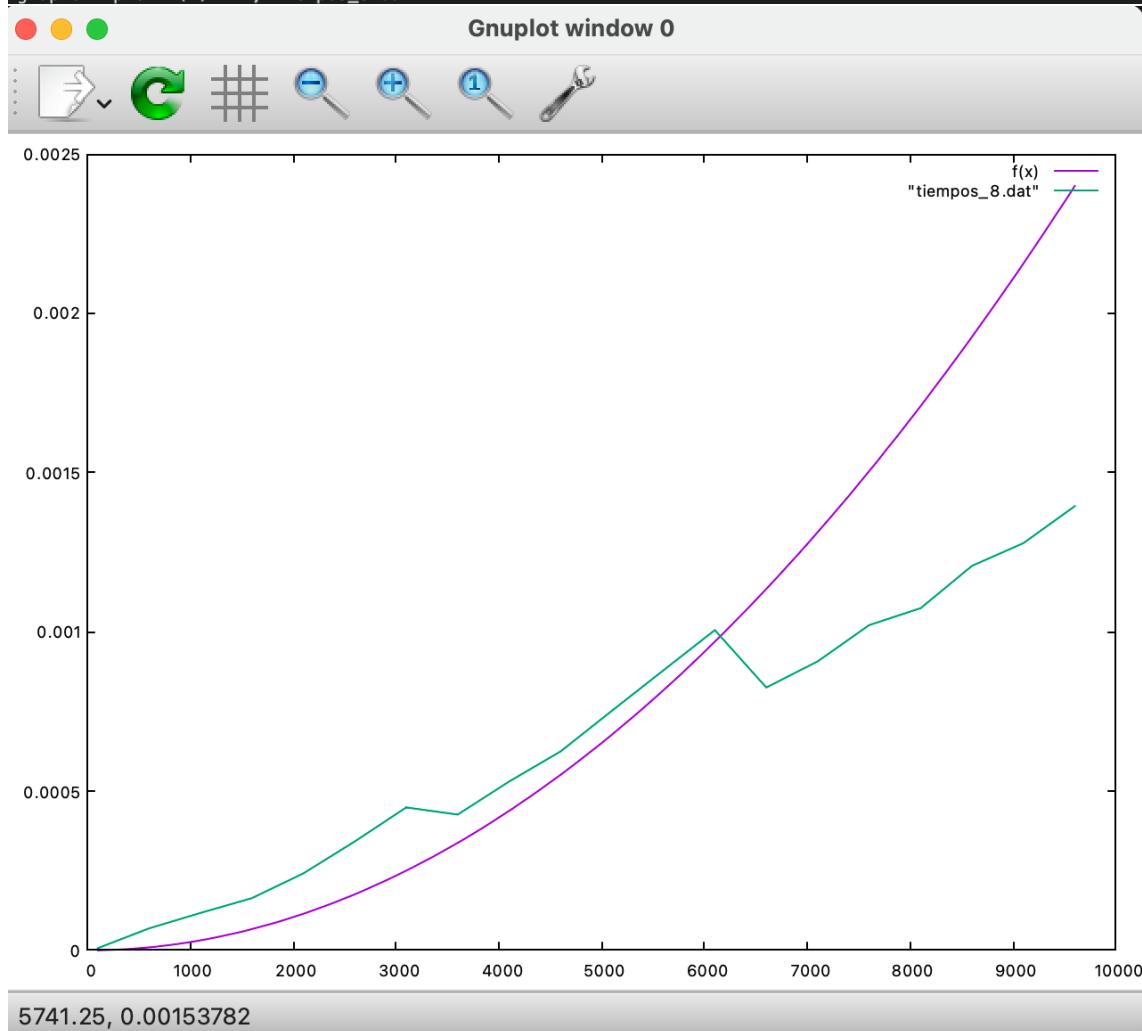
gnuplot> f(x)=a*x**2
gnuplot> fit f(x) "tiempos_8.dat" via a
iter   chisq    delta/lim  lambda a
 0 5.9273047062e+04  0.00e+00  6.51e+01  3.141530e-06
 1 2.6343576495e+02 -2.24e+07  6.51e+00  2.094596e-07
 2 1.3444554500e-04 -1.96e+11  6.51e-01  1.755315e-10
 3 2.3134574290e-07 -5.80e+07  6.51e-02  2.604396e-11
 4 2.3134573606e-07 -2.96e-03  6.51e-03  2.604290e-11
iter   chisq    delta/lim  lambda a

After 4 iterations the fit converged.
final sum of squares of residuals : 2.31346e-07
rel. change during last iteration : -2.95988e-08

degrees of freedom      (FIT_NDF)          : 13
rms of residuals       (FIT_STDFIT) = sqrt(WSSR/ndf)   : 0.000133401
variance of residuals (reduced chisquare) = WSSR/ndf   : 1.77958e-08

Final set of parameters           Asymptotic Standard Error
==================================  =====
a        = 2.60429e-11    +/- 1.721e-12   (6.61%)
gnuplot> plot f(x) w l,"tiempos_8.dat" w l

```



Ejecución de las dos funciones:

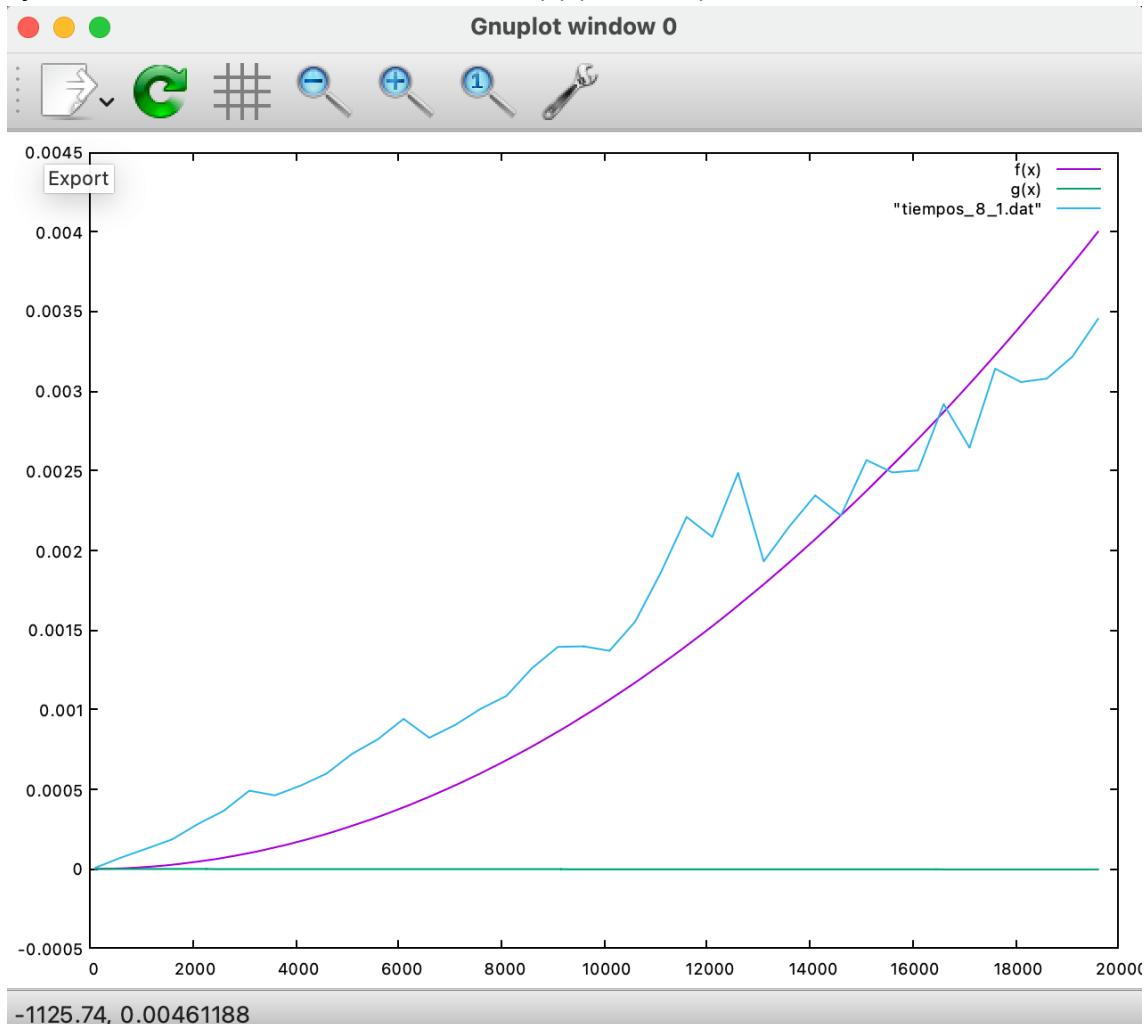
ejercicio8\_1.csh

```
1  #!/bin/csh
2  @ inicio = 100
3  @ fin = 20000
4  @ incremento = 500
5  set ejecutable = ejercicio8
6  set salida = tiempos_8.dat
7
8  @ i = $inicio
9  echo > $salida
10 while ( $i <= $fin )
11   echo Ejecución tam = $i
12   echo `./{$ejecutable} $i` >> $salida
13   @ i += $incremento
14 end
```

Cambiamos fin a de 10000 a 20000, ya que si leemos el código de mergesort.cpp, antes de tam = 10000 utilizamos el código de Inserción y a partir de dicho tamaño utiliza el mergesort.

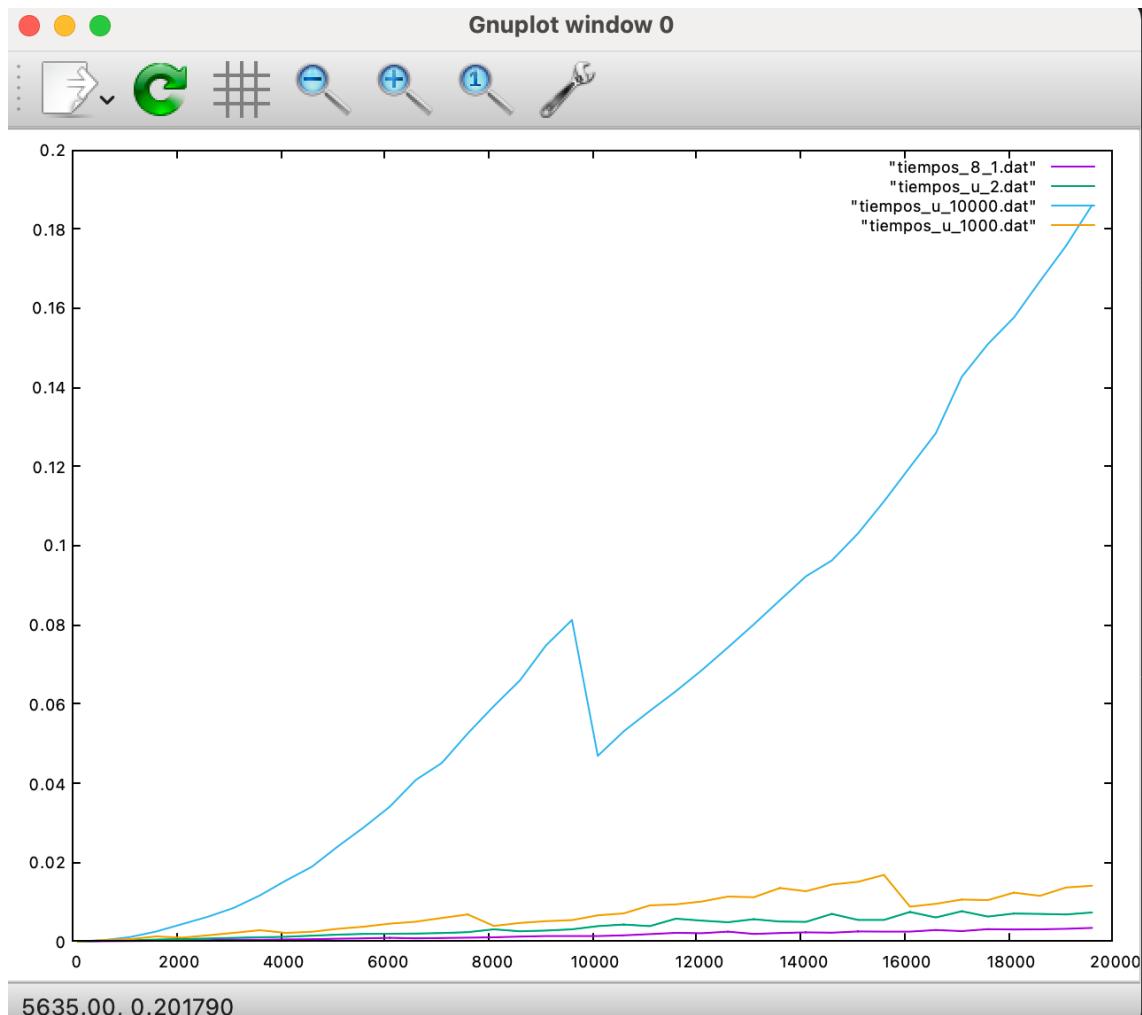
Graficas de ajuste:

Ajuste con eficiencia teórica de Inserción( $f(x)=a*x**2$ )



parámetro UMBRAL\_MS:

para la función “tiempos\_8\_1.dat” UMBRAL\_MS = 100  
para la función “tiempos\_u\_2.dat” UMBRAL\_MS = 2  
para la función “tiempos\_u\_1000.dat” UMBRAL\_MS = 1000  
para la función “tiempos\_u\_10000.dat” UMBRAL\_MS = 10000



Como se ha podido comprobar, conforme se aumentaba el umbral, se incrementaban los tiempos de ejecución.

podemos visualizar cómo la de umbral 10000 crece considerablemente a diferencia de las otras 3.