# PRÁCTICA 1: EFICIENCIA

## ÍNDICE:

**0.Características hardware y SO**

Procesador Intel core i5-8250U de 8ª generación con una velocidad de 1.6GHz y de 3.4GHz de turbo con 4 núcleos en el procesador.
8GB de RAM, 128GB SSD y 1000GB HDD. Además de utilizar la versión 18.04.3 de Ubuntu.

```
arturo@arturo-VirtualBox:~/Escritorio$ lscpu
Arquitectura:                        x86_64
modo(s) de operación de las CPUs:    32-bit, 64-bit
Orden de los bytes:                  Little Endian
CPU(s):                              4
Lista de la(s) CPU(s) en línea:      0-3
Hilo(s) de procesamiento por núcleo: 1
Núcleo(s) por «socket»:              4
«Socket(s)»                          1
Modo(s) NUMA:                        1
ID de fabricante:                    GenuineIntel
Familia de CPU:                      6
Modelo:                              142
Nombre del modelo:                   Intel(R) Core(TM) i5-8250U CPU @
1.60GHz
Revisión:                            10
CPU MHz:                             1800.002
BogoMIPS:                            3600.00
Fabricante del hipervisor:           KVM
Tipo de virtualización:              lleno
Caché L1d:                           32K
Caché L1i:                           32K
Caché L2:                            256K
Caché L3:                            6144K
```

```
arturo@arturo-VirtualBox:~/Escritorio$ cat /etc/issue
Ubuntu 18.04.3 LTS \n \l
```

**1. Ejercicio 1: Ordenación algoritmo burbuja:**

**Código fuente:**

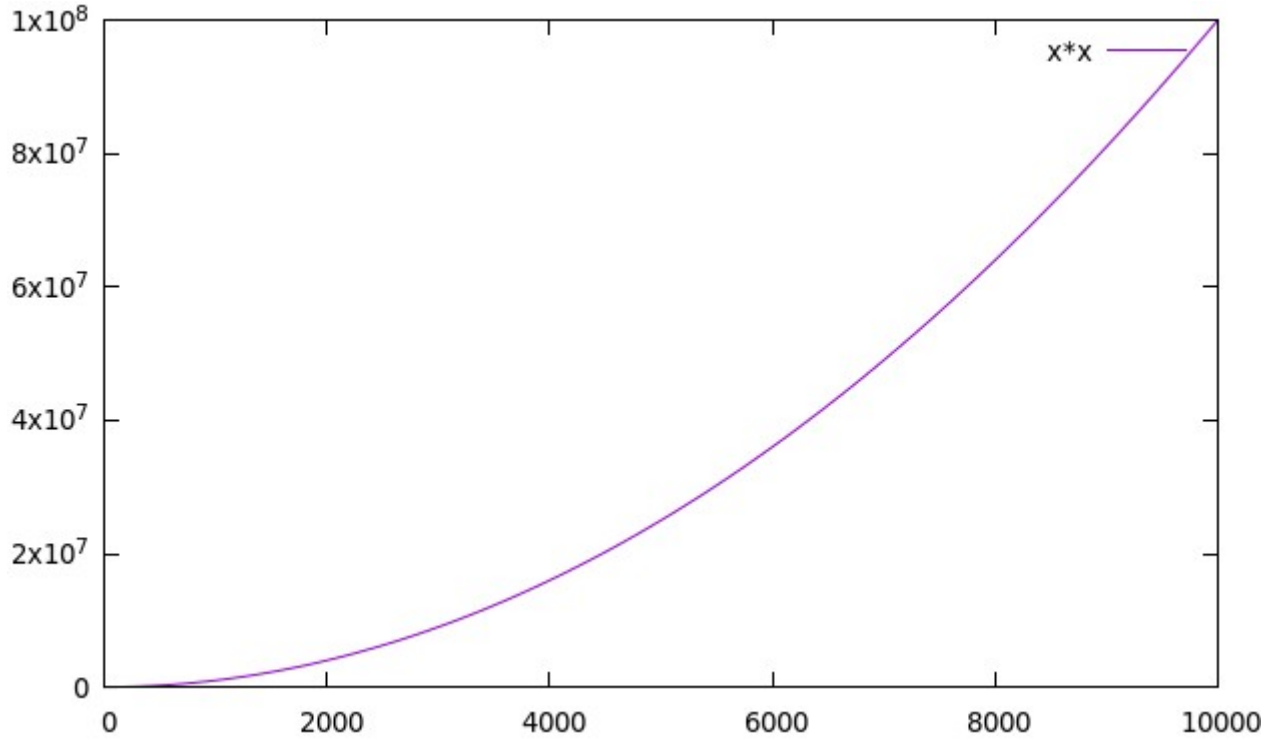
```
#include <iostream>
#include <chrono>
#include <ctime>
#include <cstdlib>

using namespace std;
using namespace std::chrono;


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]);
    }
  }
}

}

void sintaxis() {
  cerr << "Sintaxis:" << endl;
  cerr << " TAM: Tamaño del vector (>0)" << endl;
  cerr << " VMAX: Valor máximo (>0)" << endl;
  cerr << "Genera un vector de TAM números aleatorios en [0,VMAX[" << endl;
  exit(EXIT_FAILURE);
}
int main(int argc, char * argv[]) {
  if (argc!=3)
    sintaxis();

int tam=atoi(argv[1]);
int vmax=atoi(argv[2]);

  if (tam<=0 || vmax<=0)
    sintaxis();

int *v=new int[tam];

srand(time(0));

  for (int i=0; i<tam; i++)
    v[i] = rand() % vmax;

high_resolution_clock::time_point start, end;

duration<double> tiempo_transcurrido;

start = high_resolution_clock::now();
ordenar(v, tam);
end=high_resolution_clock::now();

tiempo_transcurrido = duration_cast<duration<double> >(end - start);

cout << tam << "\t" << tiempo_transcurrido.count() << endl;

delete[] v;
}
```

### 1.1 Eficiencia teórica

El tiempo de ejecución en el peor de los casos será el siguiente:

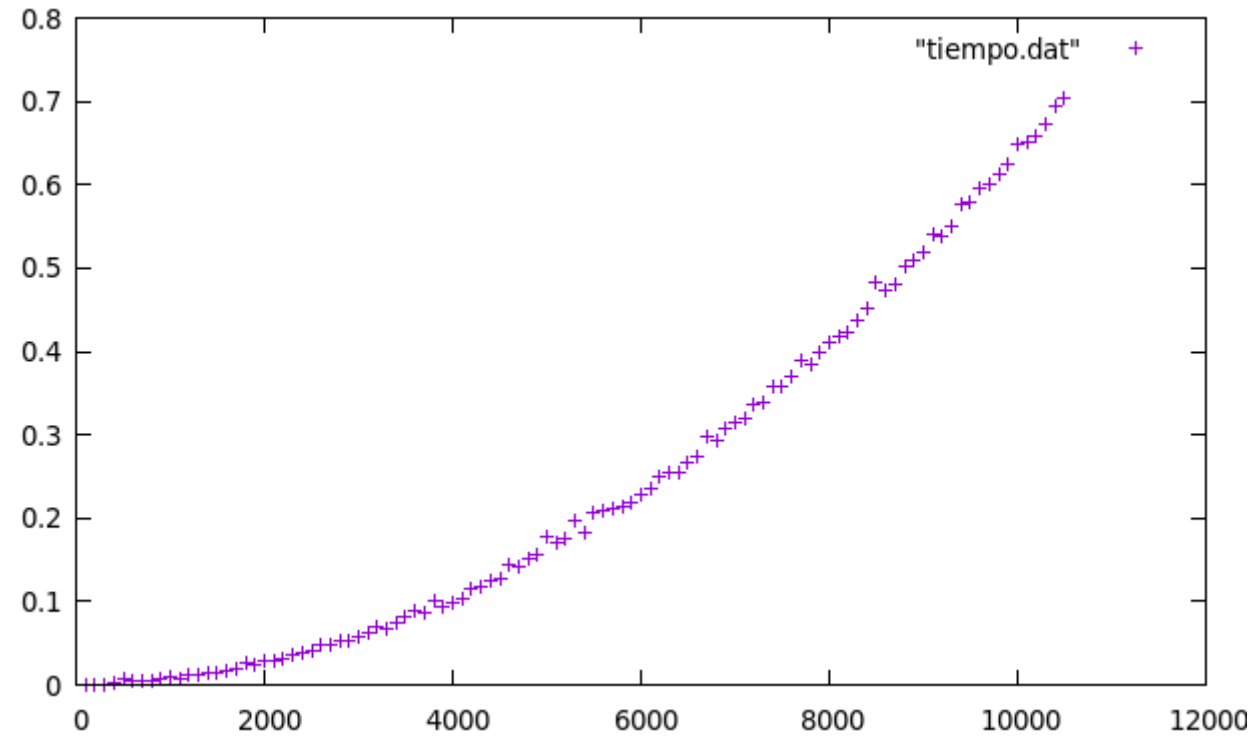
$4+7(n-1)+16n(n-1)-8(n-1)(n-2)-16(n-1) = 4+(n-1)(7+16n-8(n-2)-16)$
$4+(n-1)(8n+7)=$ **$8n^2-n-3$**

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

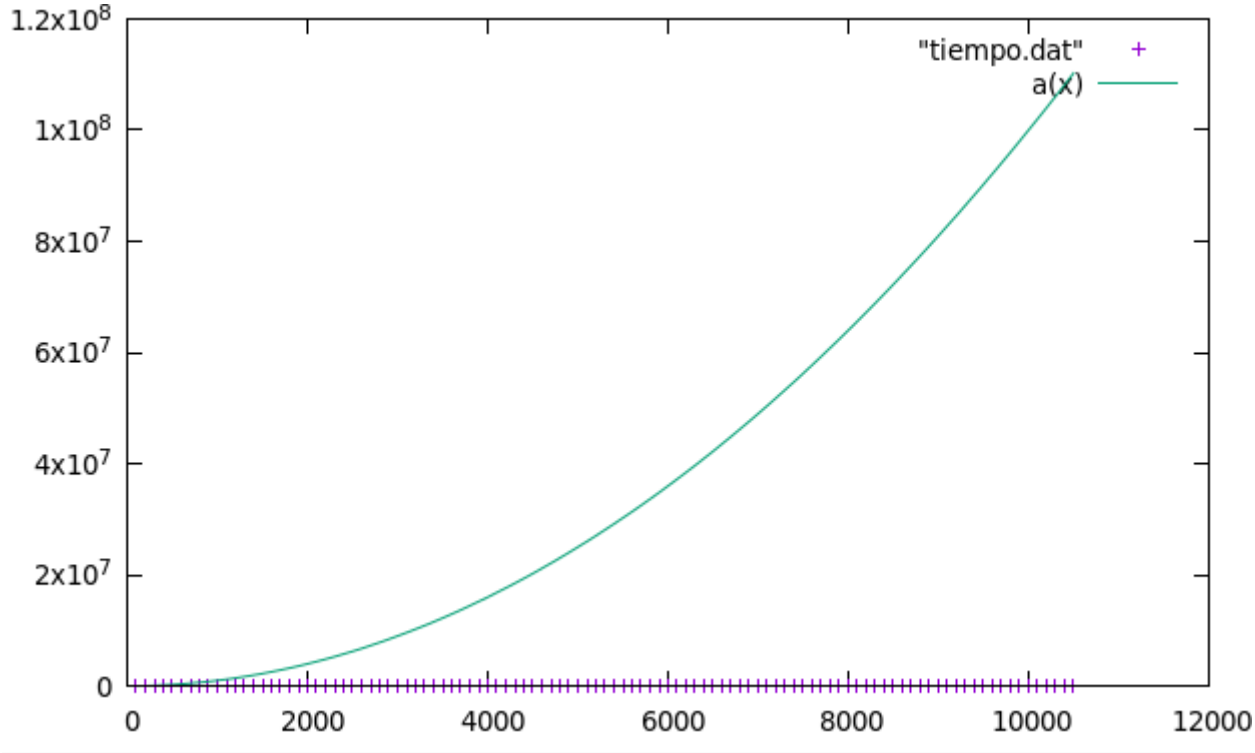



### 1.2 Eficiencia empírica

Tras ejecutar el archivo *script.csh* y dibujar el resultado obtenemos lo siguiente:



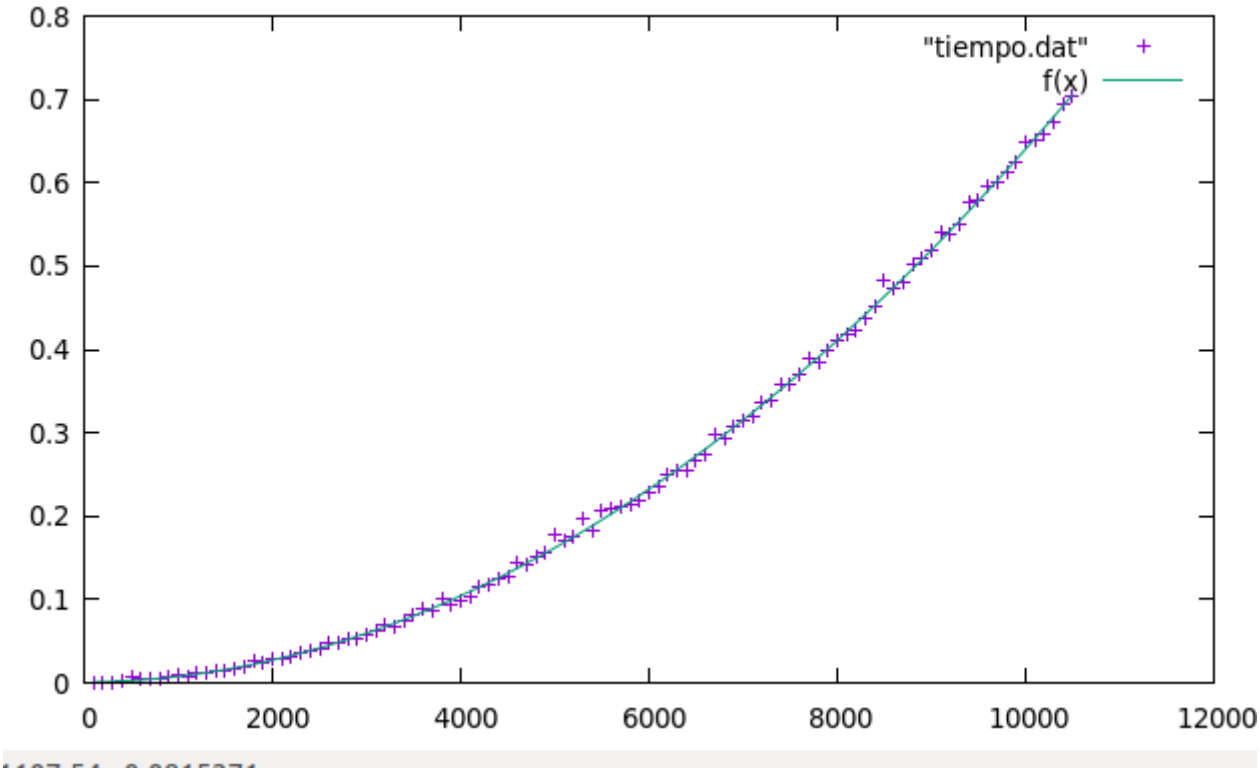
Al tratar de mostrar ambas gráficas, observamos que sí es posible visualizarlas a la vez pero que no se asemejan:



**2. Ejercicio 2: Ajuste ordenación algoritmo burbuja:**

Tras realizar el ajuste, tenemos que los valores de las constantes son:

a=6.33956e-09
b=5.28138e-07
c=0.001152017



## 3. Ejercicio 3: Problemas de precisión:

**Código fuente del ejercicio 3:**

```
int operacion(int *v, int n, int x, int inf, int sup) {
  int med;
  bool enc=false;
  while ((inf<sup) && (!enc)) {
    med = (inf+sup)/2;
    if (v[med]==x)
      enc = true;
    else if (v[med] < x)
      inf = med+1;
    else
      sup = med-1;
  }
  if (enc)
    return med;
  else
    return -1;
}
```

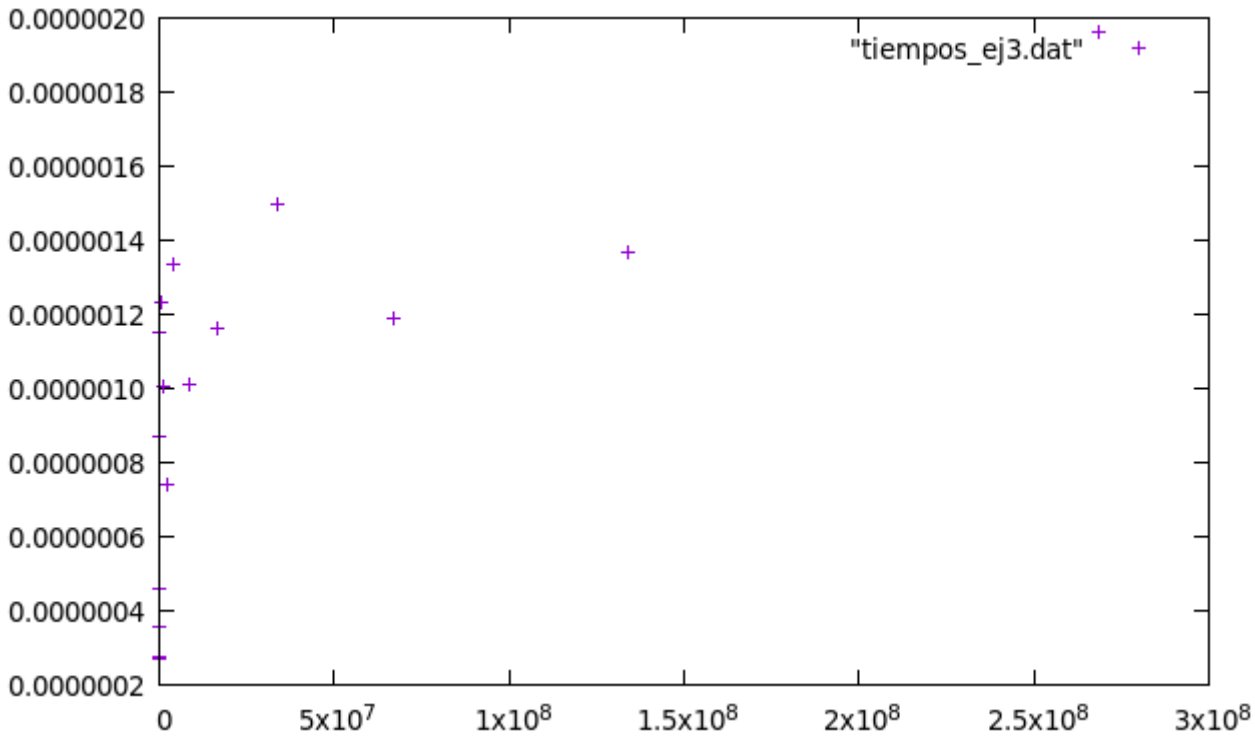
### 3.1 Eficiencia teórica:

8+7(log(n)+1) = 7log(n)+15

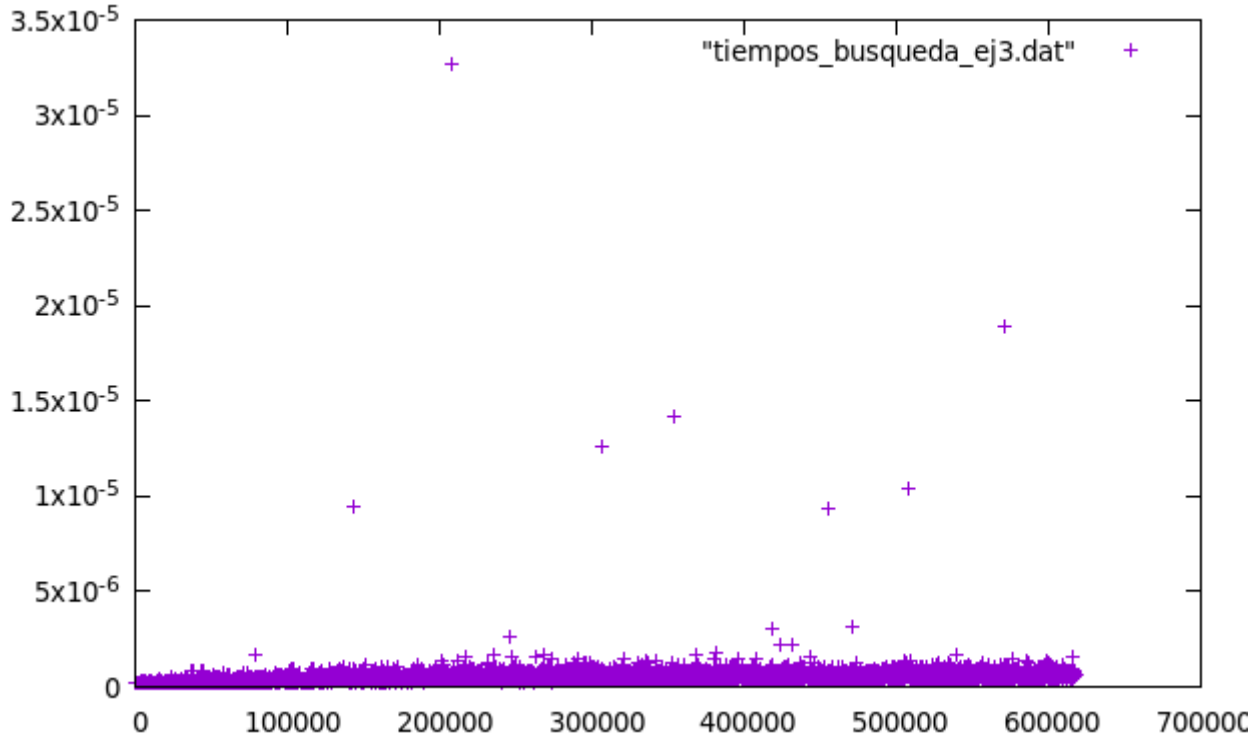
La eficiencia teórica tiene un orden de O(log(n)).

### 3.2 Eficiencia empírica:

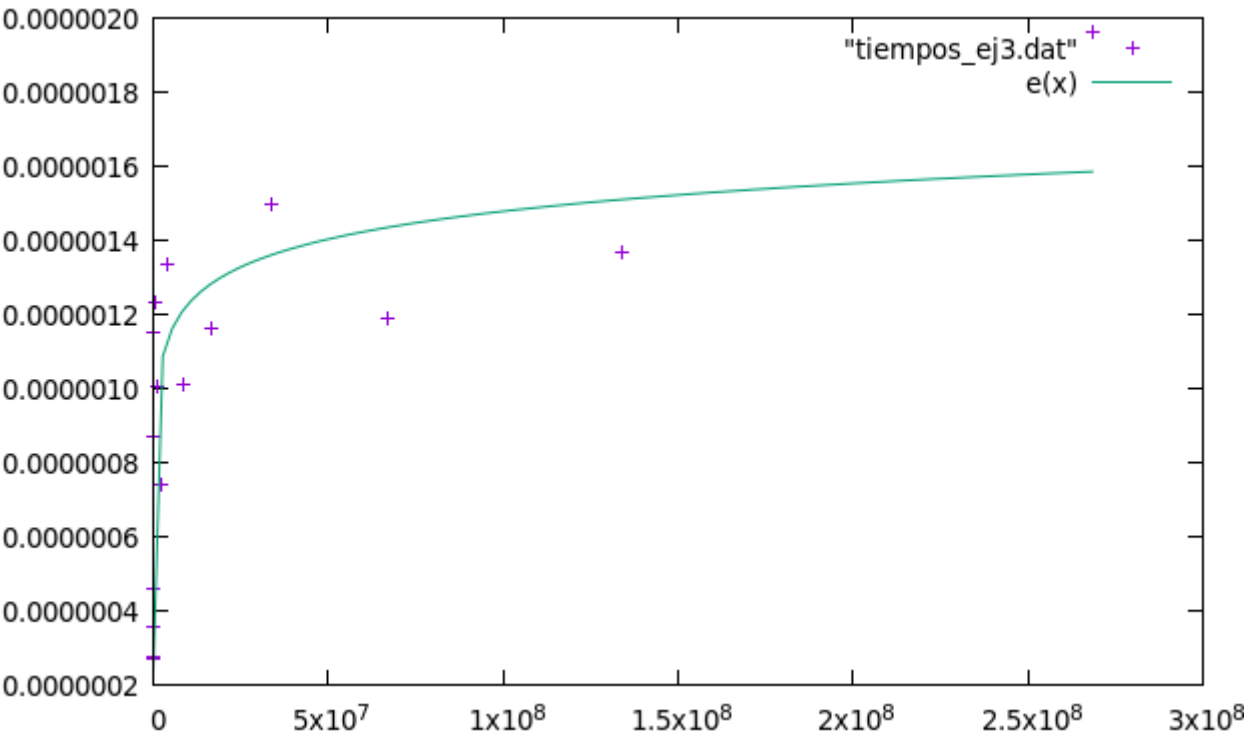
#### 3.2.1 Eficiencia empírica con chrono:



### 3.2.2 Tiempos búsqueda con chorno:

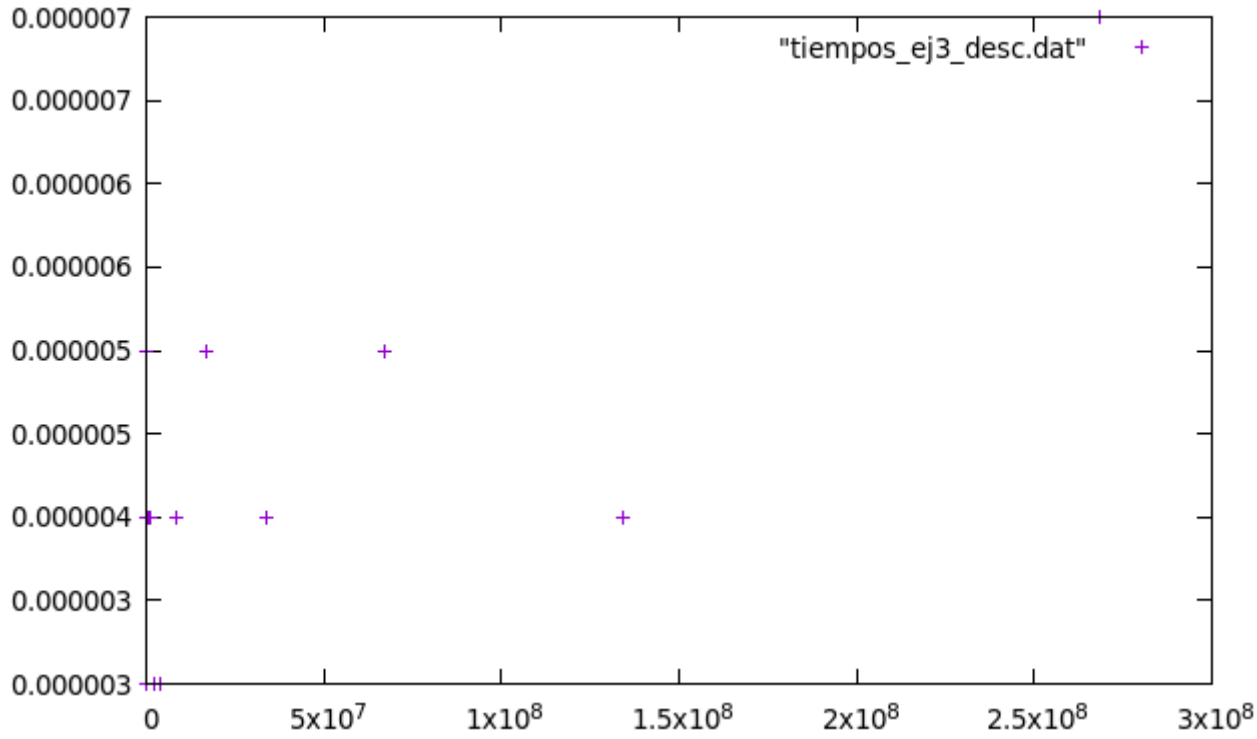



### 3.2.3 Ajuste con chrono:

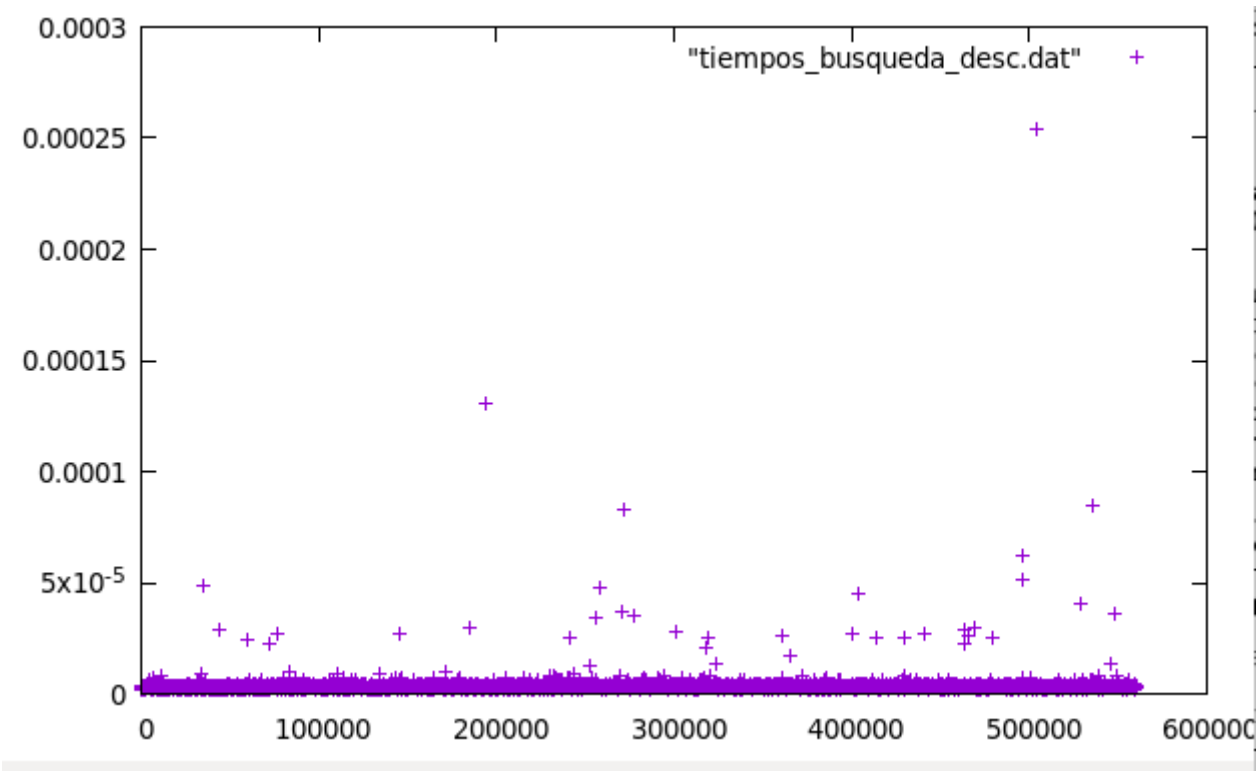



```
a=1.08526e-07
b=-5.20637e-07
```

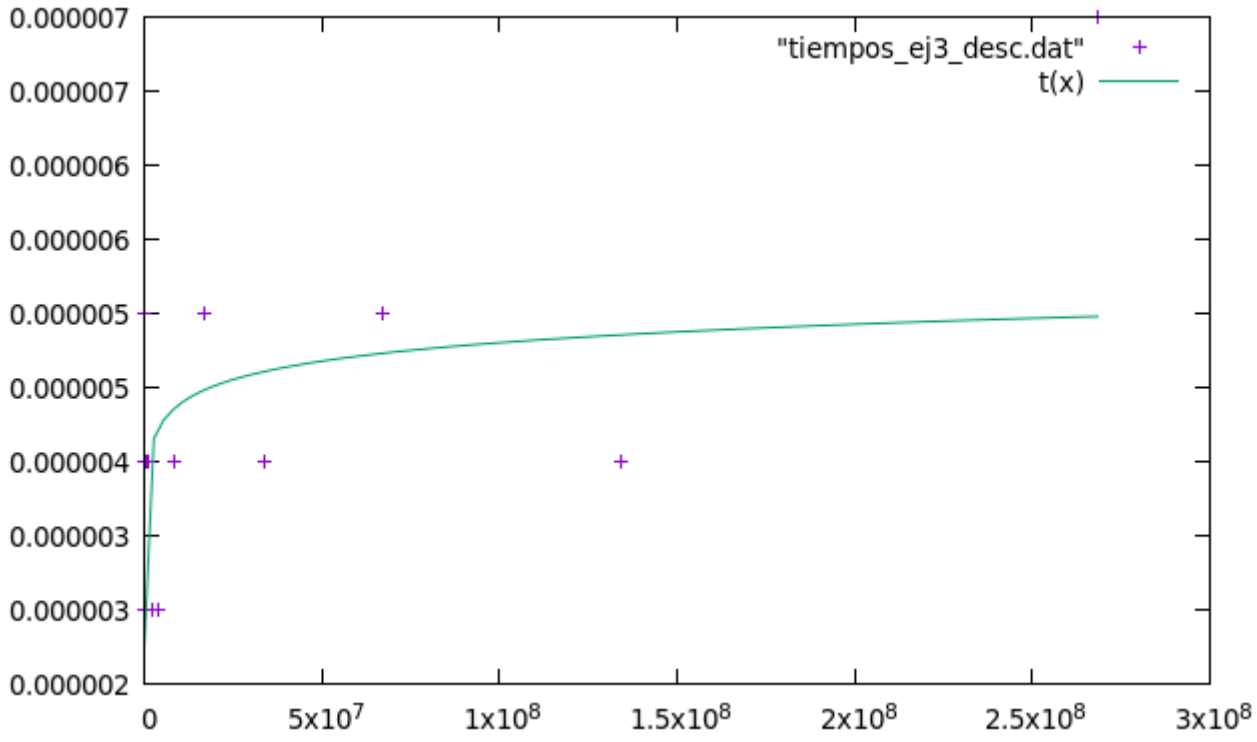
### 3.2.4 Eficiencia empírica con clock():




### 3.2.5 Tiempos búsqueda con clock():



### 3.2.6 Ajuste con clock:




a=1.79609e-07
b=1.49452e-07

**4. Ejercicio 4: Dependencia de la implementación:**

**Código fuente:**

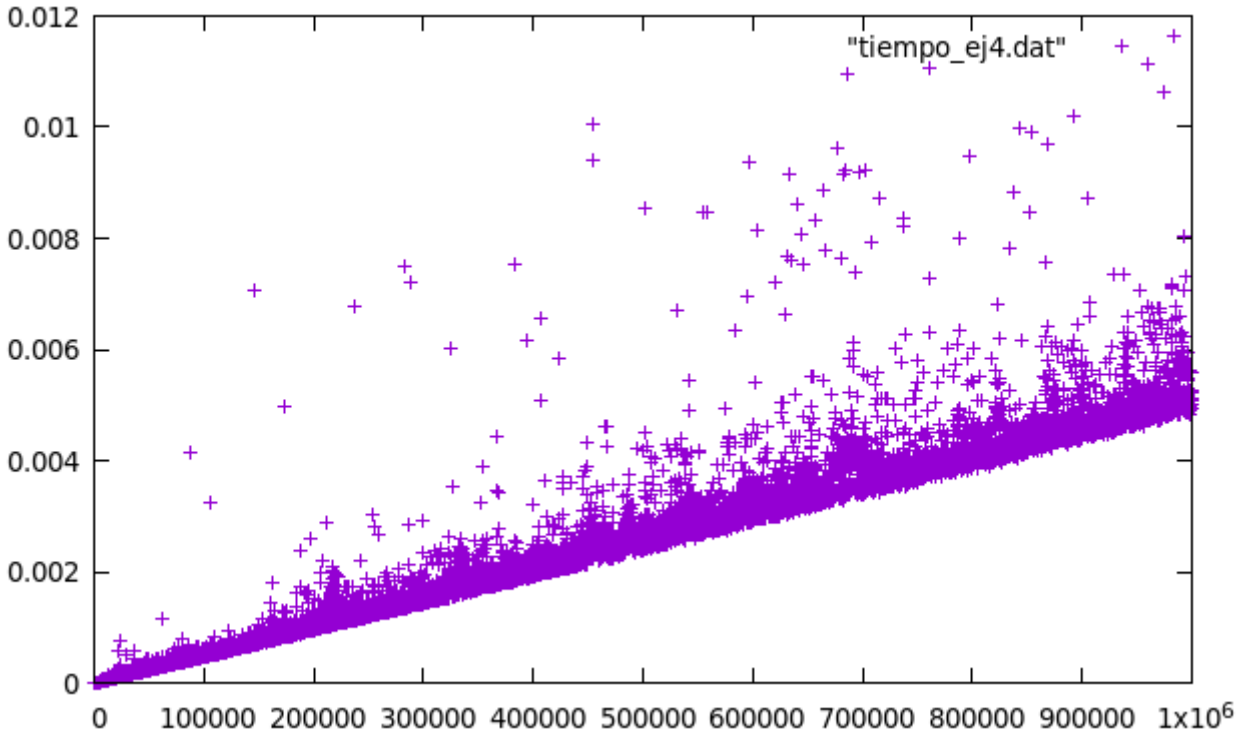
```
#include <iostream>
#include <chrono>
#include <ctime>
#include <cstdlib>

using namespace std;
using namespace std::chrono;


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]);
        }
      }
}
void sintaxis() {
  cerr << "Sintaxis:" << endl;
  cerr << " TAM: Tamaño del vector (>0)" << endl;
  cerr << " VMAX: Valor máximo (>0)" << endl;
  cerr << "Genera un vector de TAM números aleatorios en [0,VMAX[" << endl;
  exit(EXIT_FAILURE);
}
int main(int argc, char * argv[]) {
  if (argc!=3)
    sintaxis();

int tam=atoi(argv[1]);
int vmax=atoi(argv[2]);

  if (tam<=0 || vmax<=0)
    sintaxis();

int *v=new int[tam];

srand(time(0));
//Vector ordenado, para el mejor caso
  for (int i=0; i<tam; i++)
    v[i] = i-1;

high_resolution_clock::time_point start, end;

duration<double> tiempo_transcurrido;

start = high_resolution_clock::now();
ordenar(v, tam);
end=high_resolution_clock::now();

tiempo_transcurrido = duration_cast<duration<double> >(end - start);

cout << tam << "\t" << tiempo_transcurrido.count() << endl;

delete[] v;
}
```

**4.1 Eficiencia teórica:**

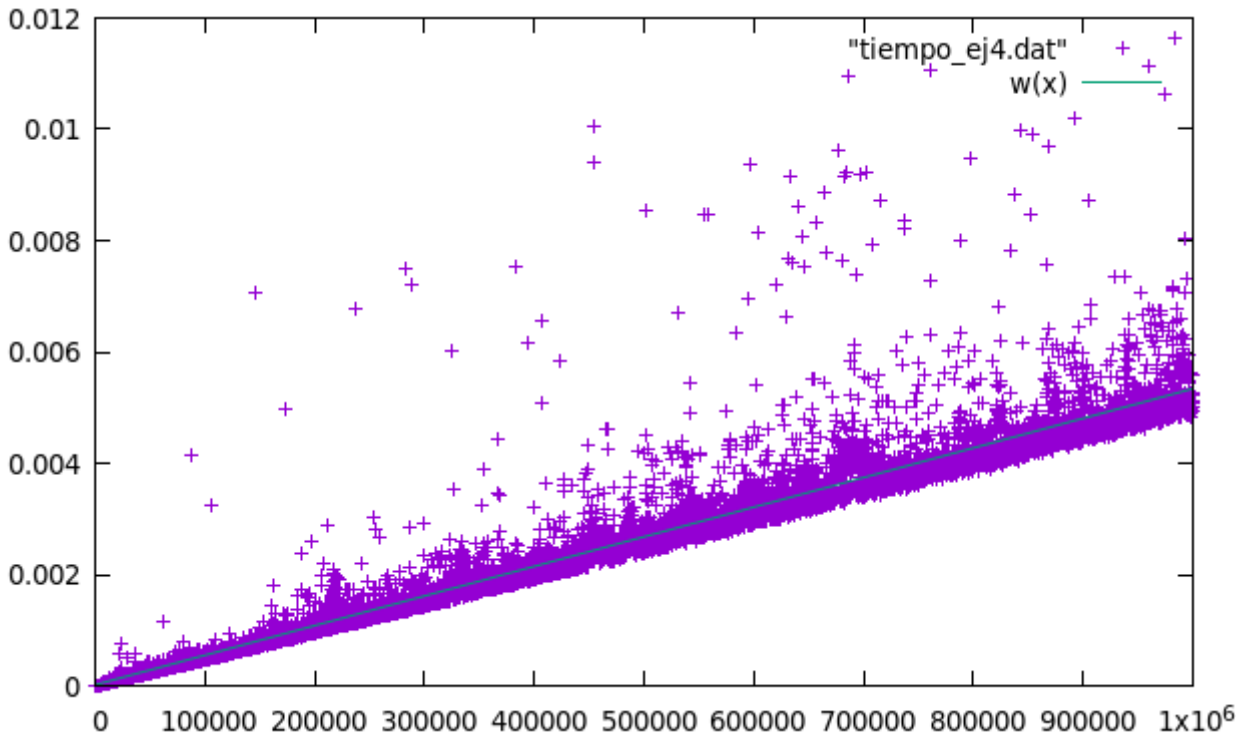
15+6(n−1) = 15+6n−6 = 6n+9

Tiene una eficiencia teórica con un orden de O(n).

**4.2 Eficiencia empírica:**




**4.3 Ajuste (w(x)):**



**5. Ejercicio 5: Mejor y pero caso:**

**-Eficiencia teórica:**

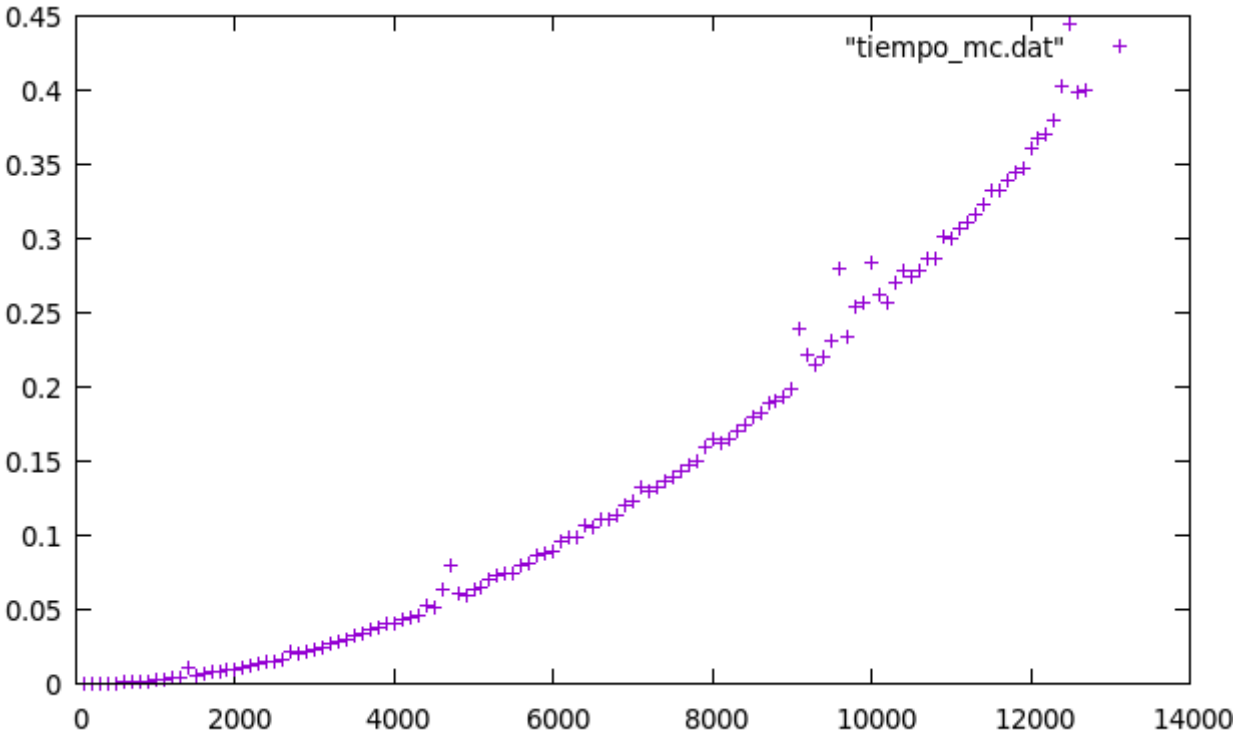
En ambos casos tiene un orden de O(n) (ejercicio 1).

**5.1 Mejor caso:**

**Código fuente:**

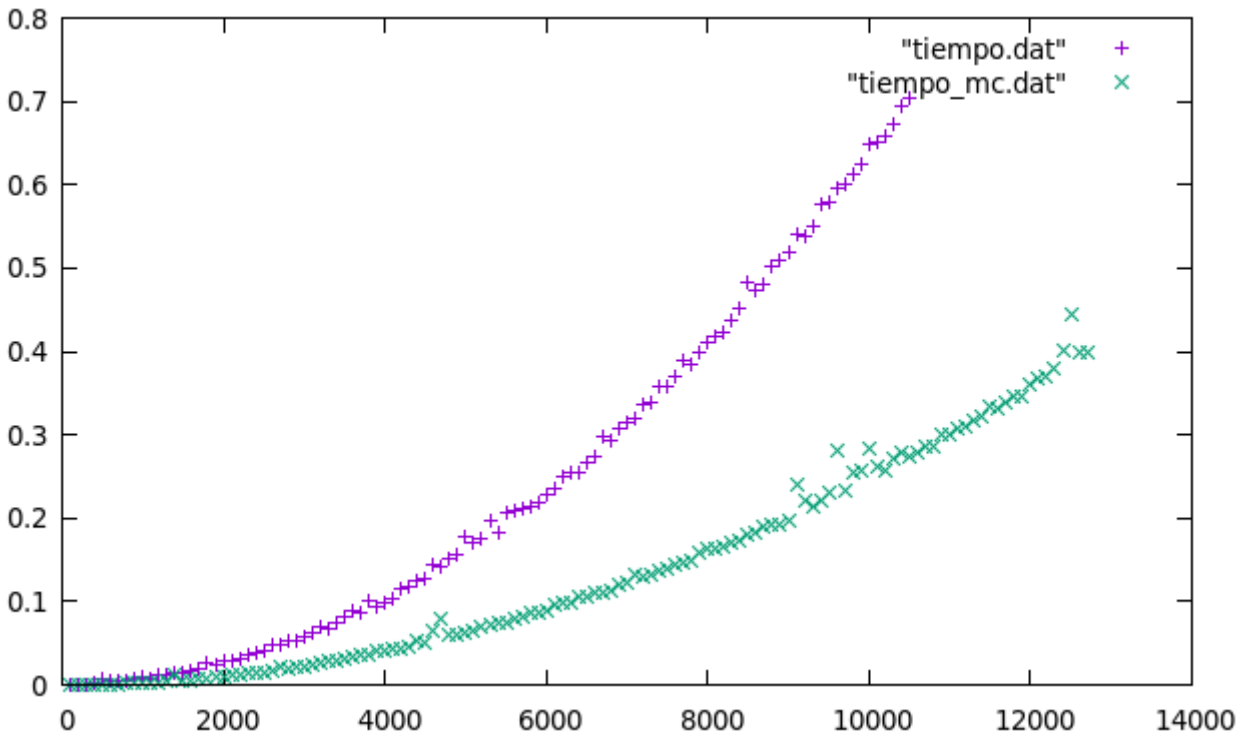
```
#include <iostream>
#include <chrono>
#include <ctime>
#include <cstdlib>

using namespace std;
using namespace std::chrono;

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]);
    }
  }
}
}
void sintaxis() {
  cerr << "Sintaxis:" << endl;
  cerr << " TAM: Tamaño del vector (>0)" << endl;
  cerr << " VMAX: Valor máximo (>0)" << endl;
  cerr << "Genera un vector de TAM números aleatorios en [0,VMAX[" << endl;
  exit(EXIT_FAILURE);
}
int main(int argc, char * argv[]) {
  if (argc!=3)
    sintaxis();
int tam=atoi(argv[1]);
int vmax=atoi(argv[2]);
  if (tam<=0 || vmax<=0)
    sintaxis();

int *v=new int[tam];

srand(time(0));
//Vector ordenado
  for (int i=0; i<tam; i++)
    v[i] = i-1;

high_resolution_clock::time_point start, end;

duration<double> tiempo_transcurrido;

start = high_resolution_clock::now();
ordenar(v, tam);
end=high_resolution_clock::now();
tiempo_transcurrido = duration_cast<duration<double> >(end - start);
cout << tam << "\t" << tiempo_transcurrido.count() << endl;
delete[] v;
}
```

### 5.1.1 Eficiencia empírica:




### 5.1.2 Comparación:

Comparación con el algortimo burbuja del ejercicio 1:

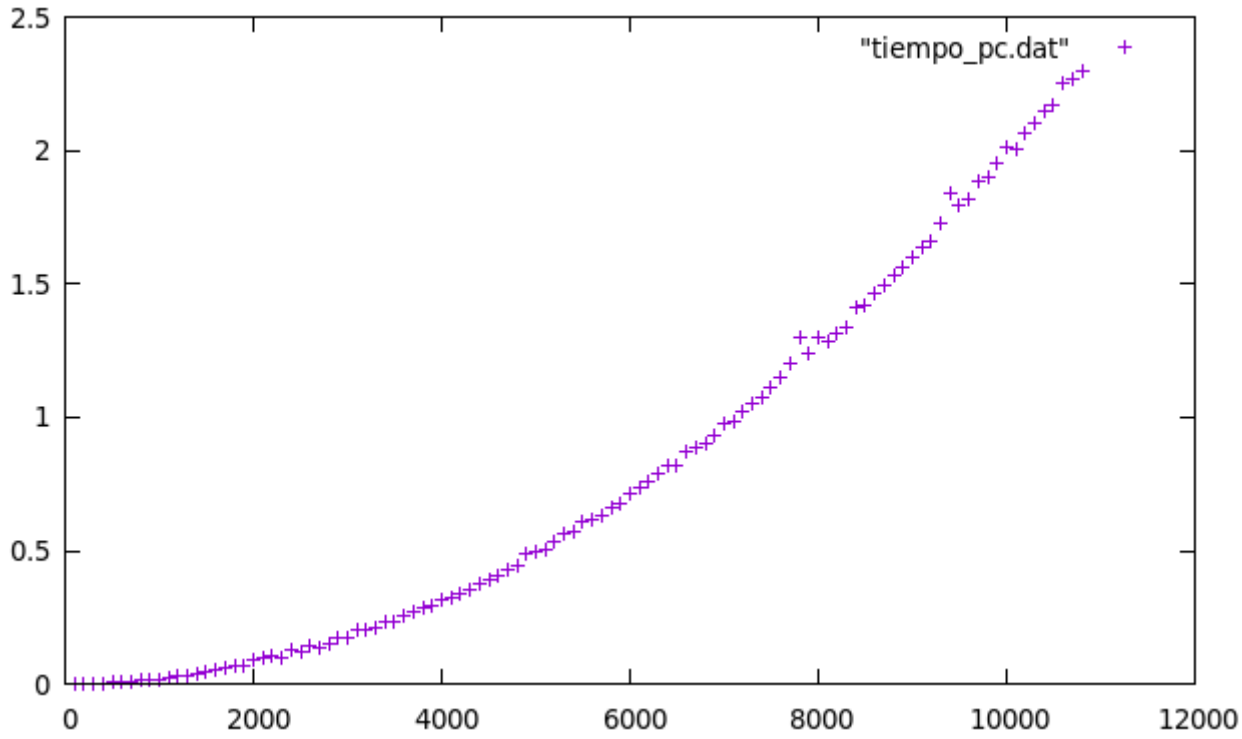


**5.2 Peor caso:**

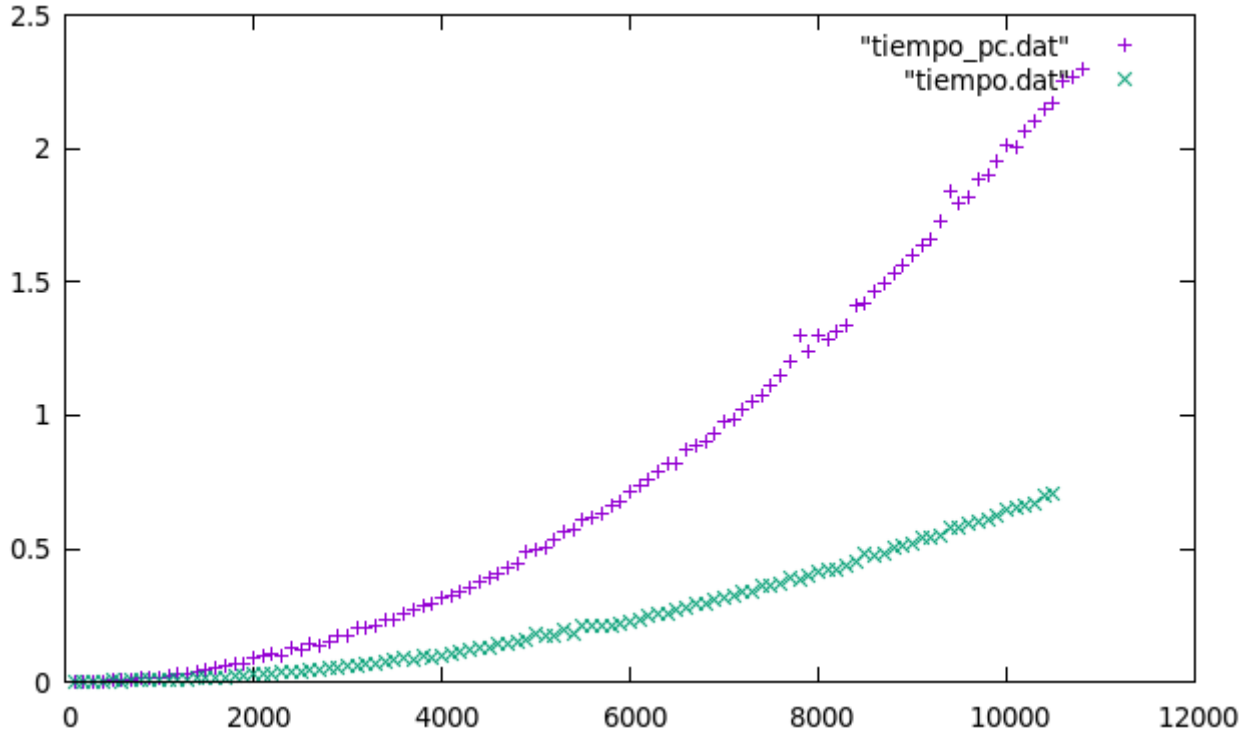
**Código fuente:**

```
#include <iostream>
#include <chrono>
#include <ctime>
#include <cstdlib>

using namespace std;
using namespace std::chrono;

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]);
    }
  }
}
}
void sintaxis() {
  cerr << "Sintaxis:" << endl;
  cerr << " TAM: Tamaño del vector (>0)" << endl;
  cerr << " VMAX: Valor máximo (>0)" << endl;
  cerr << "Genera un vector de TAM números aleatorios en [0,VMAX[" << endl;
  exit(EXIT_FAILURE);
}

int main(int argc, char * argv[]) {
  if (argc!=3)
    sintaxis();

int tam=atoi(argv[1]);
int vmax=atoi(argv[2]);

  if (tam<=0 || vmax<=0)
    sintaxis();

int *v=new int[tam];

srand(time(0));
//Vector ordenado inversamente
int cont=tam-1;
  for (int i=0; i<tam; i++){
    v[i] = cont;
    cont--;
  }

high_resolution_clock::time_point start, end;

duration<double> tiempo_transcurrido;

start = high_resolution_clock::now();
ordenar(v, tam);
end=high_resolution_clock::now();

tiempo_transcurrido = duration_cast<duration<double> >(end - start);
cout << tam << "\t" << tiempo_transcurrido.count() << endl;
delete[] v;
}
```

### 5.2.1 Eficiencia empírica:

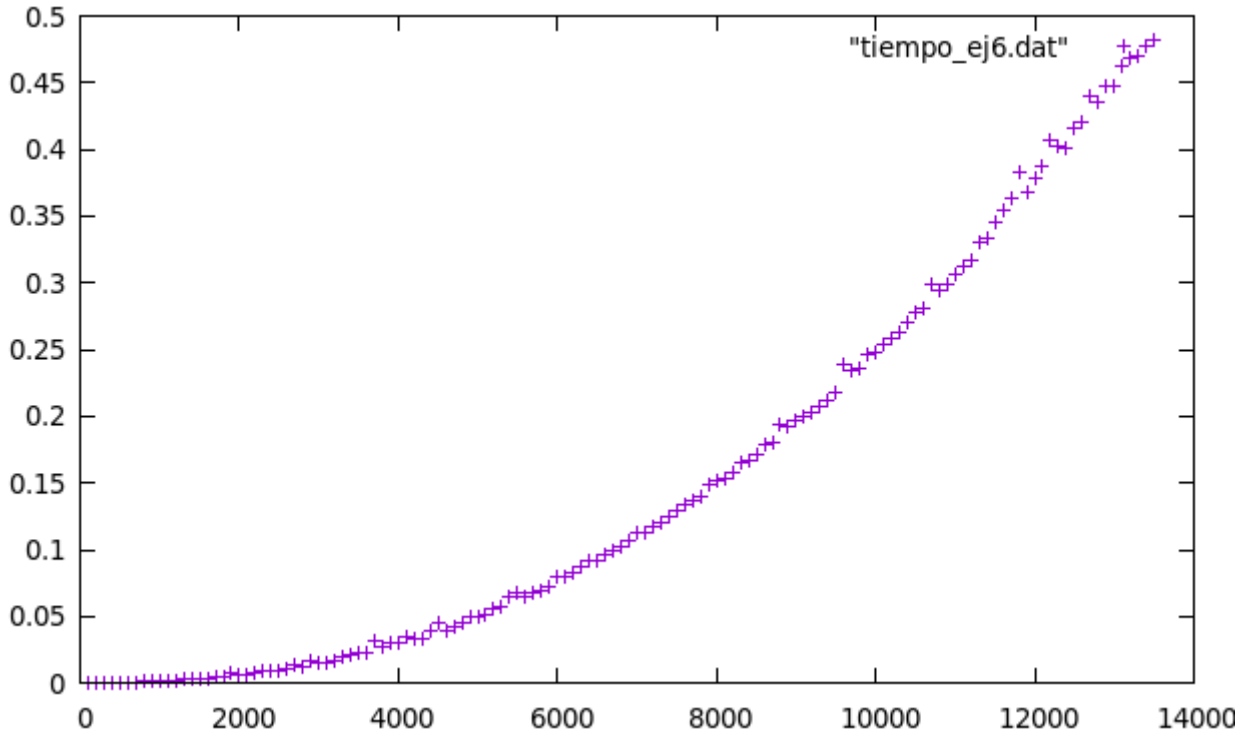



### 5.2.2 Comparación:

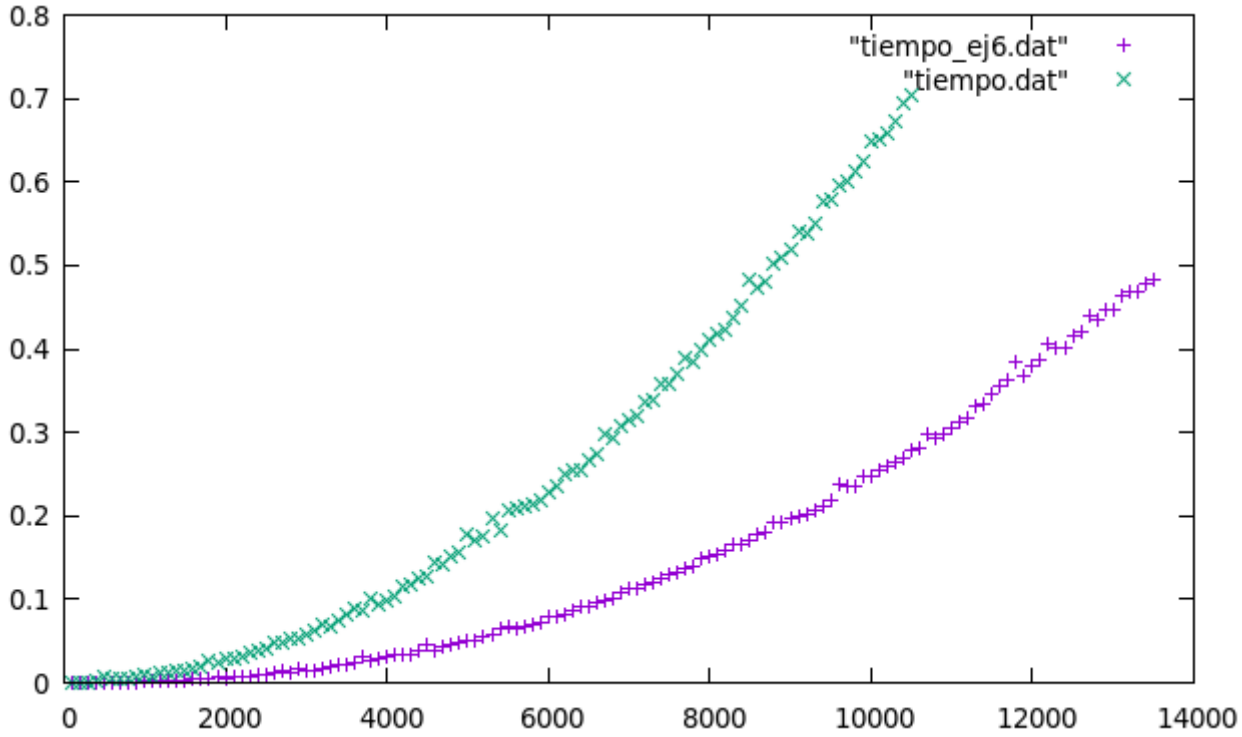


## 6. Ejercicio 6: Influencia del proceso de compilación:

### 6.1 Eficiencia empírica:




### 6.2 Comparación:



**7. Ejercicio 7: Multiplicación Matricial:**

**Código fuente:**

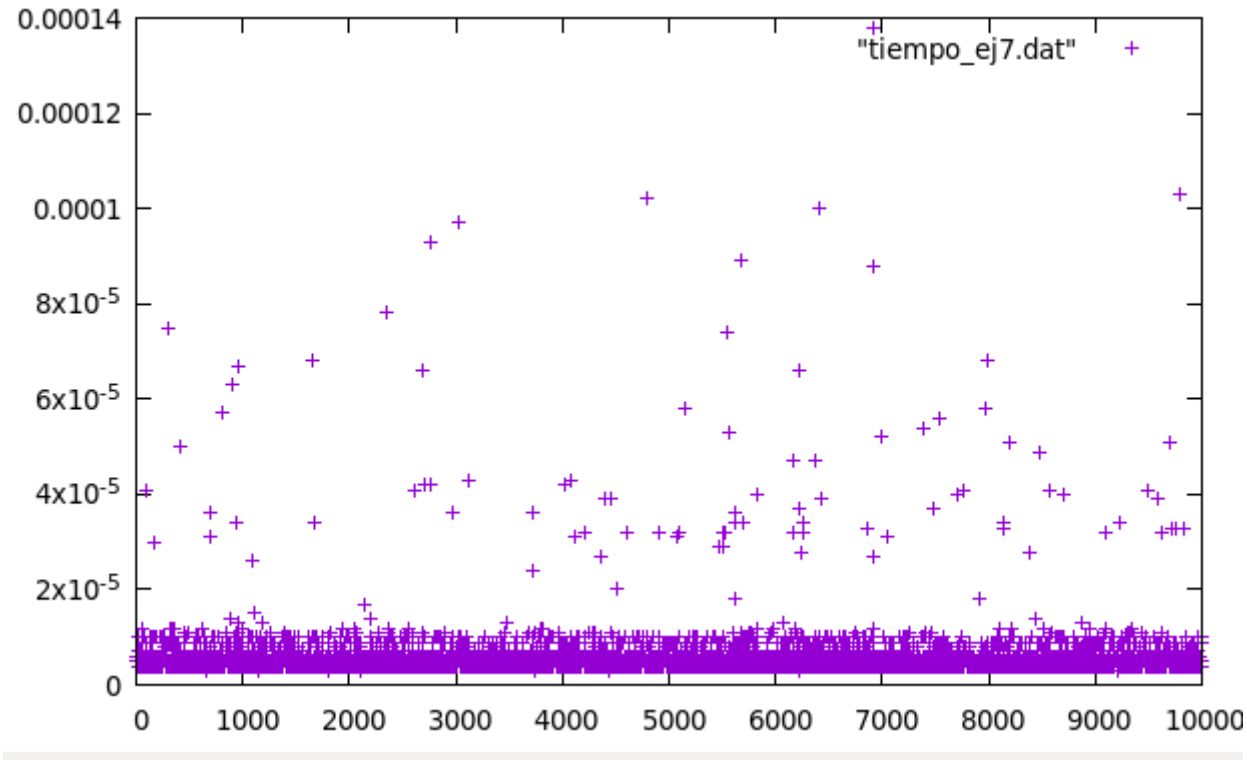
```
#include <iostream>
#include <ctime>    // Recursos para medir tiempos

using namespace std;

void Multiplicar(int matriz[3][3], int matrizz[3][3]) {

    int product[3][3] = {{0, 0, 0}, {0, 0, 0}, {0, 0, 0}};

    for (int fila = 0; fila < 3; fila++) {
        for (int col = 0; col < 3; col++) {
            for (int i = 0; i < 2; i++) {
                product[fila][col] += matriz[fila][i] * matrizz[i][col];
            }
        }
    }
}
int main() {

    clock_t tini;    // Anotamos el tiempo de inicio
    tini=clock();

    int matriz [3][3], matrizz[3][3];

    for (int fila = 0; fila < 3; fila++)  {
        for (int columna = 0; columna < 3; columna++)  {
            matriz[fila][columna] = rand() % 100;
            matrizz[fila][columna] = rand() % 100;
        }
    }

    Multiplicar(matriz, matrizz);
    clock_t tfin;    // Anotamos el tiempo de finalización
    tfin=clock();
    // Mostramos resultados
    cout << (tfin-tini)/(double)CLOCKS_PER_SEC << endl;
}
```

**7.1 Eficiencia teórica:**

$6+6n+3n^2-3n+11(n^3-n^2)/2 = 3n^2+3n+6+11n^3/2-11n^2/2= 5.5n^3-2.5n^2+3n+6$

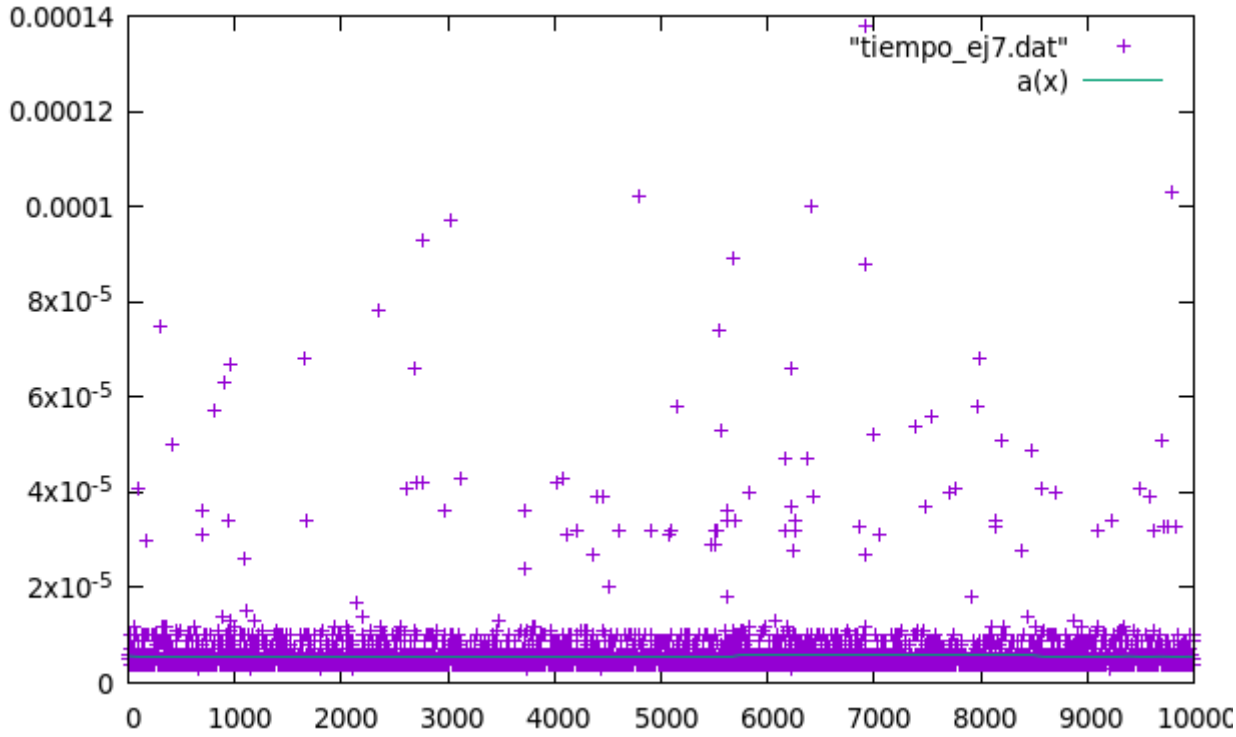
Tiene un eficiencia con orden $O(n^3)$.

**7.2 Eficiencia empírica:**




**7.3 Ajuste:**

```
a=-2.51578e-18
b=3.55802e-14
c=-1.18894e-10
d=5.694e-06
```



**8. Ejercicio 8: Mergesort:**

**Código fuente:**

```
/**
   @file Ordenaci▫n por mezcla
*/


#include <iostream>
#include <ctime>
#include <cstdlib>
#include <climits>
#include <cassert>

using namespace std;



/* ********************************************************** */
/*  Método de ordenaci▫n por mezcla  */

/**
   @brief Ordena un vector por el m▫todo de mezcla.

   @param T: vector de elementos. Debe tener num_elem elementos.
             Es MODIFICADO.
   @param num_elem: n▫mero de elementos. num_elem > 0.

   Cambia el orden de los elementos de T de forma que los dispone
   en sentido creciente de menor a mayor.
   Aplica el algoritmo de mezcla.
*/
inline static
void mergesort(int T[], int num_elem);



/**
   @brief Ordena parte de un vector por el m▫todo de mezcla.

   @param T: vector de elementos. Tiene un n▫mero de elementos
                   mayor o igual a final. Es MODIFICADO.
   @param inicial: Posici▫n que marca el incio de la parte del
                   vector a ordenar.
   @param final: Posici▫n detr▫s de la ▫ltima de la parte del
                   vector a ordenar.
               inicial < final.

   Cambia el orden de los elementos de T entre las posiciones
   inicial y final - 1 de forma que los dispone en sentido creciente
   de menor a mayor.
   Aplica el algoritmo de la mezcla.
*/
static void mergesort_lims(int T[], int inicial, int final);


/**
   @brief Ordena un vector por el m▫todo de inserci▫n.

   @param T: vector de elementos. Debe tener num_elem elementos.
```

```
            Es MODIFICADO.
   @param num_elem: n▫mero de elementos. num_elem > 0.

   Cambia el orden de los elementos de T de forma que los dispone
   en sentido creciente de menor a mayor.
   Aplica el algoritmo de inserci▫n.
*/
inline static
void insercion(int T[], int num_elem);


/**
   @brief Ordena parte de un vector por el m▫todo de inserci▫n.

   @param T: vector de elementos. Tiene un n▫mero de elementos
                  mayor o igual a final. Es MODIFICADO.
   @param inicial: Posici▫n que marca el incio de la parte del
                  vector a ordenar.
   @param final: Posici▫n detr▫s de la ▫ltima de la parte del
                  vector a ordenar.
              inicial < final.

   Cambia el orden de los elementos de T entre las posiciones
   inicial y final - 1 de forma que los dispone en sentido creciente
   de menor a mayor.
   Aplica el algoritmo de la inserci▫n.
*/
static void insercion_lims(int T[], int inicial, int final);


/**
   @brief Mezcla dos vectores ordenados sobre otro.

   @param T: vector de elementos. Tiene un n▫mero de elementos
                  mayor o igual a final. Es MODIFICADO.
   @param inicial: Posici▫n que marca el incio de la parte del
                  vector a escribir.
   @param final: Posici▫n detr▫s de la ▫ltima de la parte del
                  vector a escribir
              inicial < final.
   @param U: Vector con los elementos ordenados.
   @param V: Vector con los elementos ordenados.
              El n▫mero de elementos de U y V sumados debe coincidir
              con final - inicial.

   En los elementos de T entre las posiciones inicial y final - 1
   pone ordenados en sentido creciente, de menor a mayor, los
   elementos de los vectores U y V.
*/
static void fusion(int T[], int inicial, int final, int U[], int V[]);



/**
   Implementaci▫n de las funciones
**/


inline static void insercion(int T[], int num_elem)
{
  insercion_lims(T, 0, num_elem);
}
```

```
static void insercion_lims(int T[], int inicial, int final)
{
  int i, j;
  int aux;
  for (i = inicial + 1; i < final; i++) {
    j = i;
    while ((T[j] < T[j-1]) && (j > 0)) {
      aux = T[j];
      T[j] = T[j-1];
      T[j-1] = aux;
      j--;
    };
  };
}


const int UMBRAL_MS = 100;

void mergesort(int T[], int num_elem)
{
  mergesort_lims(T, 0, num_elem);
}

static void mergesort_lims(int T[], int inicial, int final)
{
  if (final - inicial < UMBRAL_MS)
    {
      insercion_lims(T, inicial, final);
    } else {
      int k = (final - inicial)/2;

      int * U = new int [k - inicial + 1];
      assert(U);
      int l, l2;
      for (l = 0, l2 = inicial; l < k; l++, l2++)
      U[l] = T[l2];
      U[l] = INT_MAX;

      int * V = new int [final - k + 1];
      assert(V);
      for (l = 0, l2 = k; l < final - k; l++, l2++)
      V[l] = T[l2];
      V[l] = INT_MAX;

      mergesort_lims(U, 0, k);
      mergesort_lims(V, 0, final - k);
      fusion(T, inicial, final, U, V);
      delete [] U;
      delete [] V;
    };
}


static void fusion(int T[], int inicial, int final, int U[], int V[])
{
  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];
```

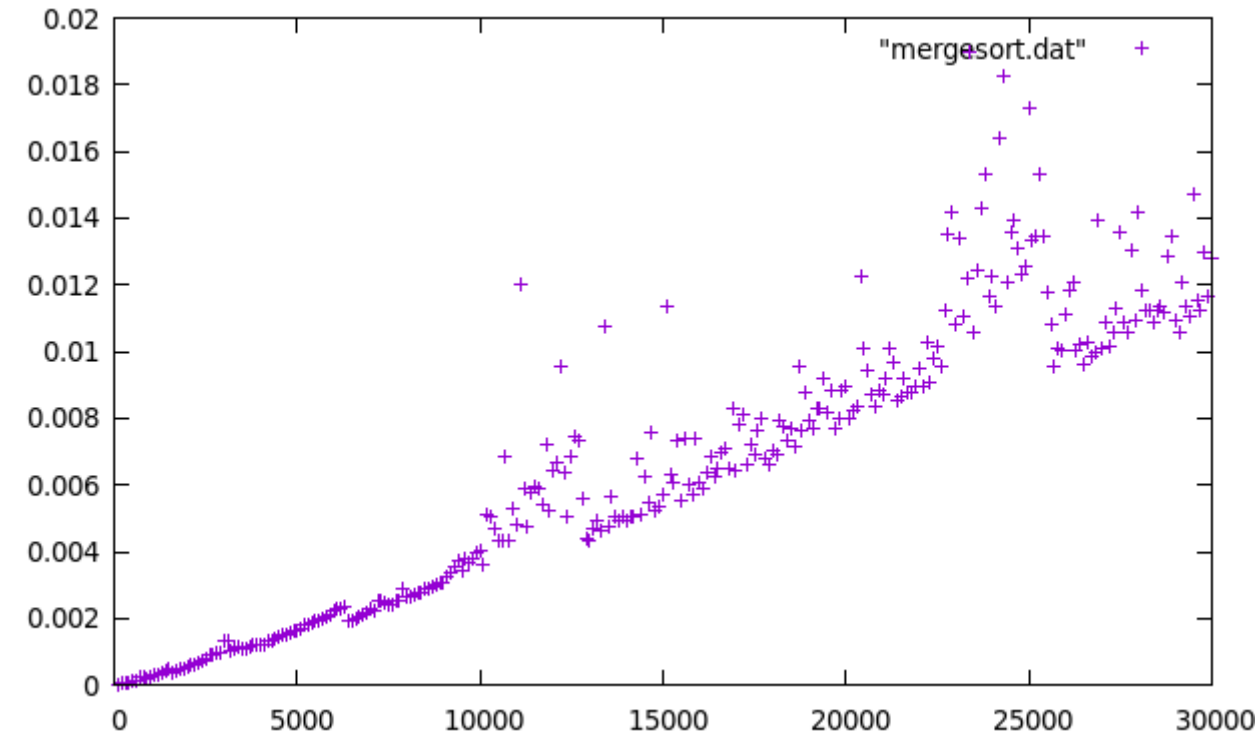
```
      k++;
      };
    };
}




int main(int argc, char * argv[])
{

  if (argc != 2)
    {
      cerr << "Formato " << argv[0] << " <num_elem>" << endl;
      return -1;
    }

  int n = atoi(argv[1]);

  int * T = new int[n];
  assert(T);

  srandom(time(0));

  for (int i = 0; i < n; i++)
    {
      T[i] = random();
    };

  const int TAM_GRANDE = 10000;
  const int NUM_VECES = 1000;

  if (n > TAM_GRANDE)
    {
      clock_t t_antes = clock();

      mergesort(T, n);

      clock_t t_despues = clock();

      cout << n << "  " << ((double)(t_despues - t_antes)) / CLOCKS_PER_SEC
         << endl;
    } else {
      int * U = new int[n];
      assert(U);

      for (int i = 0; i < n; i++)
      U[i] = T[i];

      clock_t t_antes_vacio = clock();
      for (int veces = 0; veces < NUM_VECES; veces++)
      {
        for (int i = 0; i < n; i++)
          U[i] = T[i];
      }
      clock_t t_despues_vacio = clock();

      clock_t t_antes = clock();
      for (int veces = 0; veces < NUM_VECES; veces++)
      {
        for (int i = 0; i < n; i++)
          U[i] = T[i];
        mergesort(U, n);
```

```
      }
      clock_t t_despues = clock();
      cout << n << " \t  "
         << ((double) ((t_despues - t_antes) -
                    (t_despues_vacio - t_antes_vacio))) /
      (CLOCKS_PER_SEC * NUM_VECES)
         << endl;

      delete [] U;
    }


  delete [] T;

  return 0;
};
```

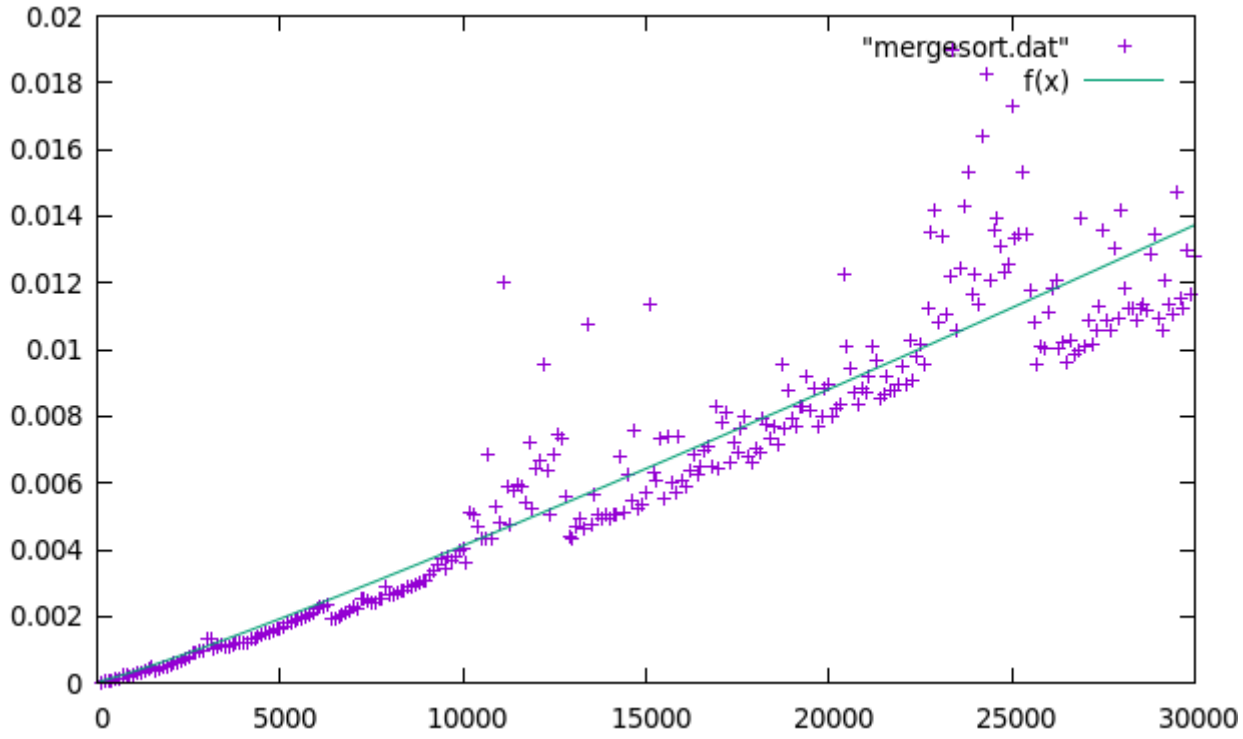
**8.1 Eficiencia teórica:**

El algoritmo tiene una eficiencia del orden de O(n*log(n)).

**8.2 Eficiencia empírica:**



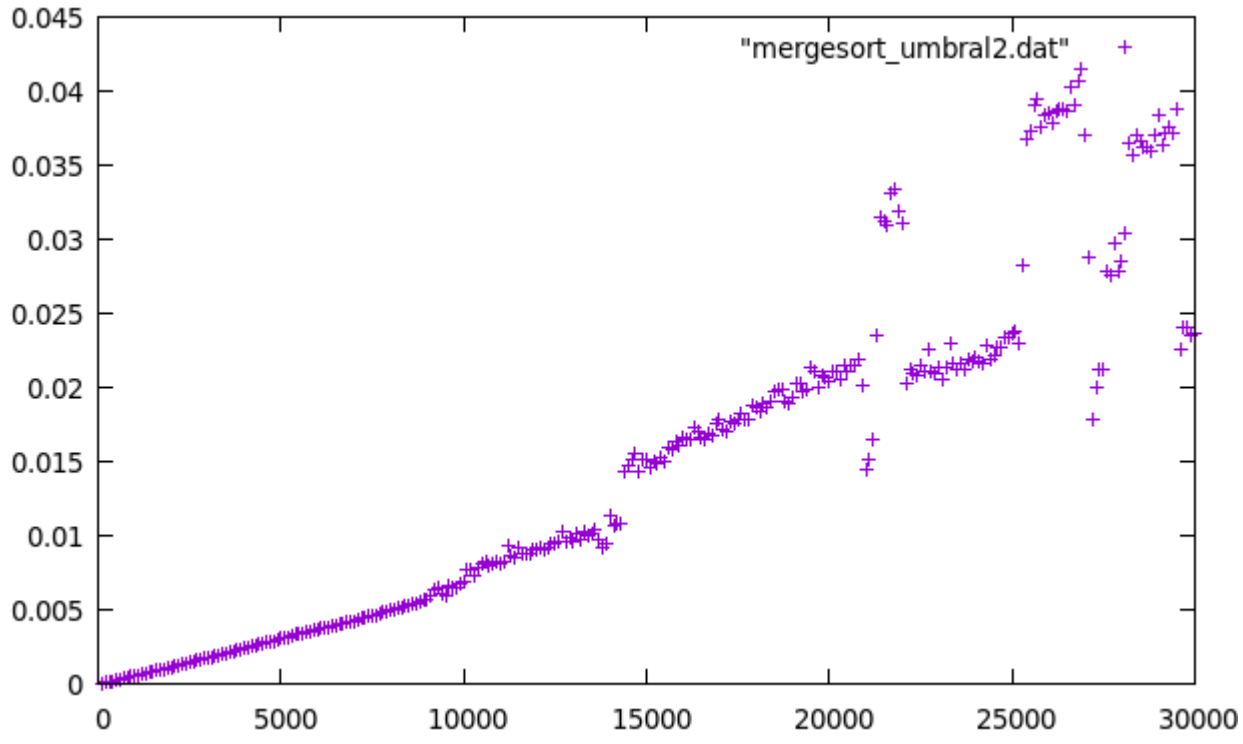
**8.3 Ajuste de mergesort con n*log(n):**



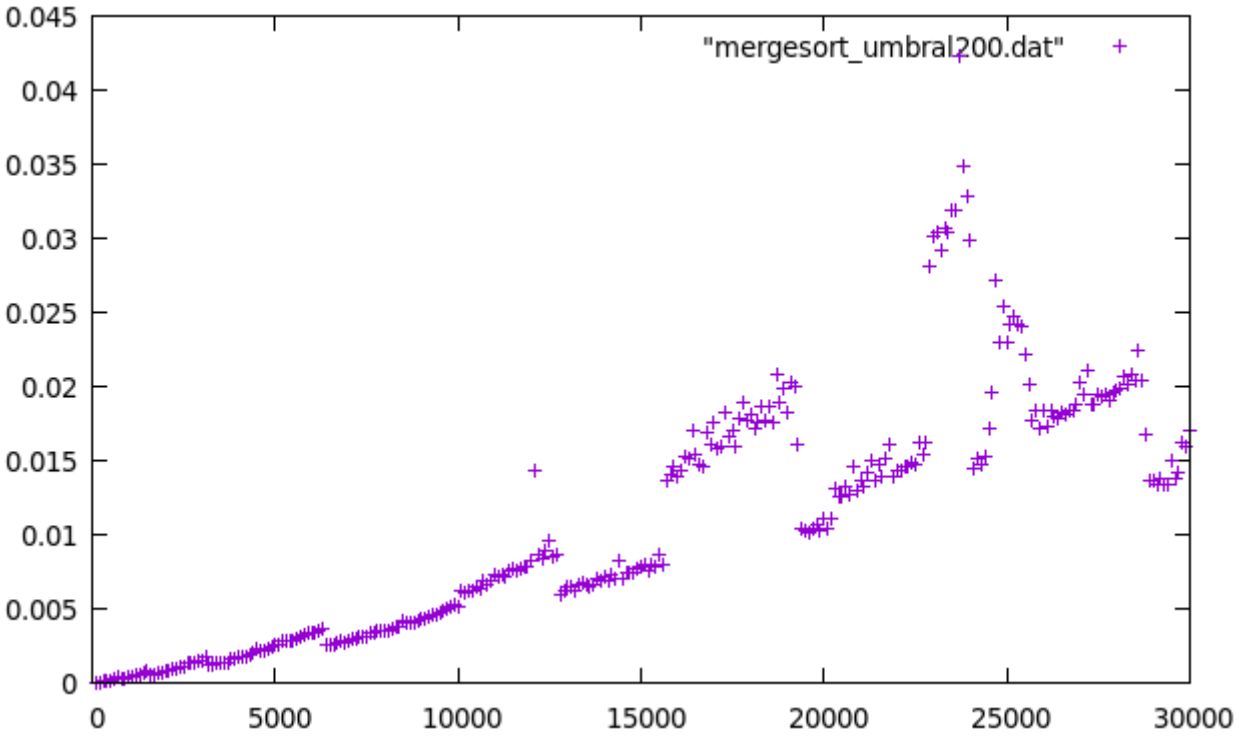

**8.4 Estudio umbral:**

Cuanto mayor es el parámetro del umbral, mayor es el tiempo que tarda en realizar la ordenación. Veamos ahora las gráficas con umbrales 2, 200 y 500 así como la comparación entre ellas y la original (umbral 100):

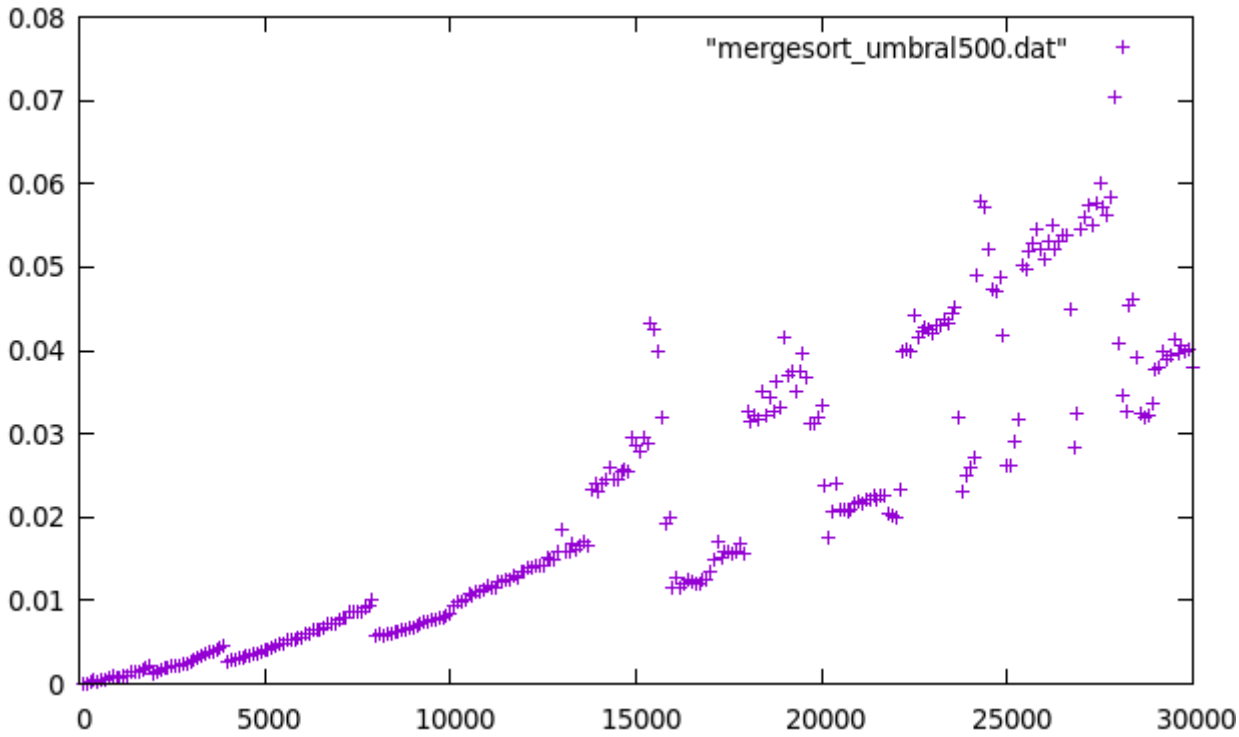
**-Umbral 2:**



**-Umbral 200:**




**-Umbral 500:**



**-Comparación entre los diferentes umbrales:**