Jose Miguel Hernández García
2º C C3

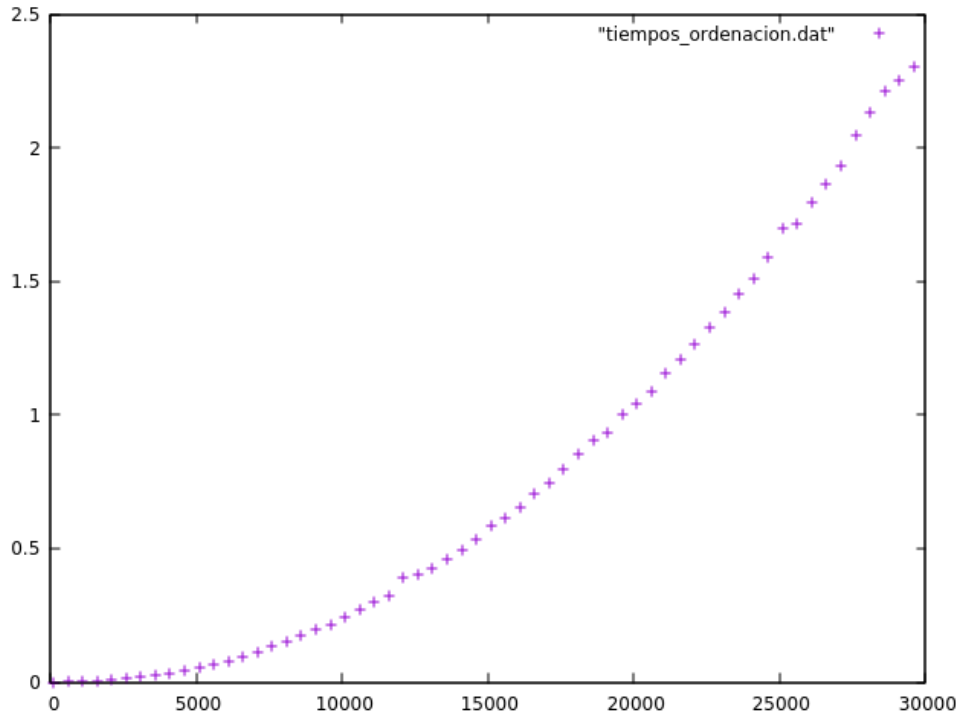
El equipo utilizado para esta práctica ha sido:

MSI GE70 2PE Apache Pro
Intel(R) Core(TM) i7-4710HQ CPU @ 2.50GHz
Size: 3395MHz
Capacity: 3500MHz
Width: 64 bits
Clock: 33MHz
8GB RAM
SO: Ubuntu 16.04 LTS 64B
Compilador: g++ (Ubuntu 5.4.0-6ubuntu1~16.04.5) 5.4.0 20160609
Para compilar todos los ejercicios se ha utilizado la orden: **g++ programa.cpp -o ejecutable -std=c++11** excepto en el ejercicio 6 que hemos añadido la opción -O3 para agregarle optimización como se pedía.

EJERCICIO 1

```
#include <iostream>
#include <ctime> // Recursos para medir tiempos
#include <cstdlib> // Para generación de números aleatorios

using namespace std;

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;
            }
}

void sintaxis(){
    cerr << "Sintaxis: " << endl;
    cerr << "  TAM:  Tamanio 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) // Lectura de parámetros
        sintaxis();

    int tam = atoi(argv[1]);  // Tamanio del vector
    int vmax = atoi(argv[2]); // Valor máximo

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

    // Generamos el vector aleatorio
    int * v = new int[tam];  // Reserva de memoria
    srand(time(0));          // Inicializamos el generador de números aleatorios

    for(int i = 0; i < tam; i++) // Recorrer vector
        v[i] = rand() % vmax;    // Generar aleatorio [0, VMAX[

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

    ordenar(v, tam); // Ordenamos el vector

    clock_t tfin; // Anotamos el tiempo de finalización
    tfin = clock();

    // Mostramos resultados (Tam del vector y tiempo de ejecución en seg.)
    cout << tam << "\t" << (tfin-tini) / (double) CLOCKS_PER_SEC << endl;

    delete[] v; // Liberamos memoria dinámica
}
```

Tras compilar el programa y ejecutar el script que ejecuta dicho programa. Hemos obtenido el fichero tiempos_ordenacion.dat. Y tras recurrir a Gnuplot para representar la gráfica de los tiempos de ejecución respecto al tamaño del vector a ordenar, hemos obtenido la siguiente gráfica:



Por otro lado, si estudiamos de forma teórica la eficiencia del algoritmo de ordenación

```
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;
            }
}
```

Vemos que su eficiencia teórica es:

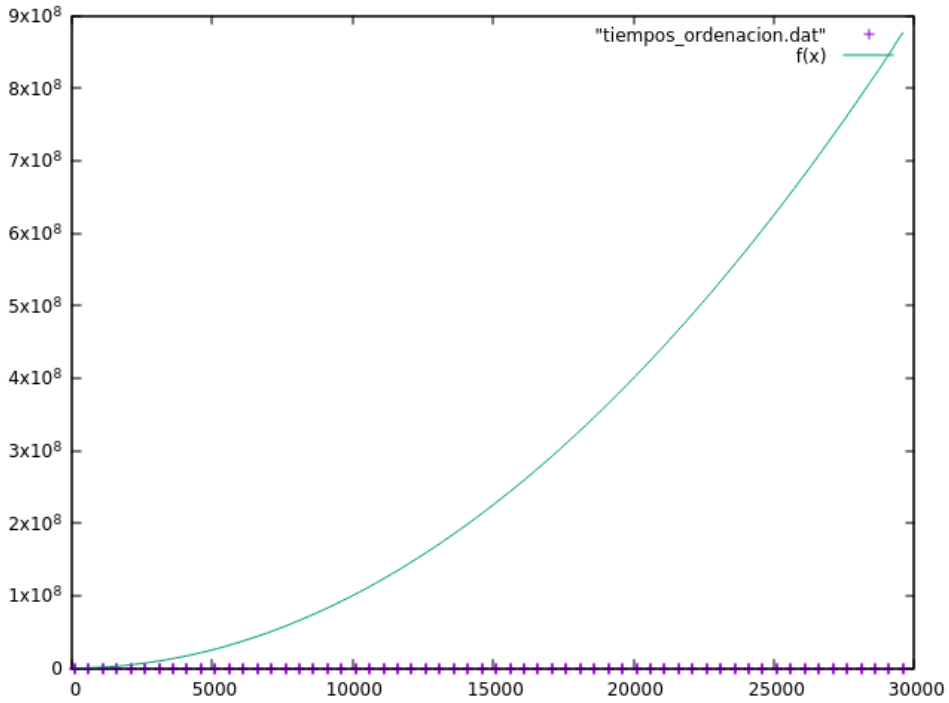
Linea 2: Bucle for: $\sum(1 + 1 + 1) = \sum(3) = O(n)$
Linea 3: Bucle for; $\sum(1 + 1 + 1) = \sum(3) = O(n)$
Linea 5: Asignación; $1 = O(1)$
Linea 6: Asignación; $1 = O(1)$
Linea 7: Asignación; $1 = O(1)$

Finalmente = $\sum(3 + \sum(3 + 1 + 1 + 1)) = n * n = O(n^2)$

(Ambas sumatorias van desde i, j=0 hasta n-1 y n-i-1 respectivamente)

Por tanto podemos ver que la eficiencia teórica del algoritmo, en el peor de los casos es $O(n^2)$.

Tras superponer ambas gráficas (la de eficiencia empírica y teórica), y utilizando $f(x) = a*x^2 + b*x + c$, obtenemos la siguiente gráfica:



Es posible apreciar que, dado que la eficiencia teórica es n^2, al elevar los valores 100, 600, …, 30000 al cuadrado, vemos que la gráfica crece hasta valores muy grandes que, comparados con los que hemos obtenido en la eficiencia empírica, hacen que esta parezca una recta en y = 0 ya que sus valores son demasiado pequeños en comparación a los de la eficiencia teórica.

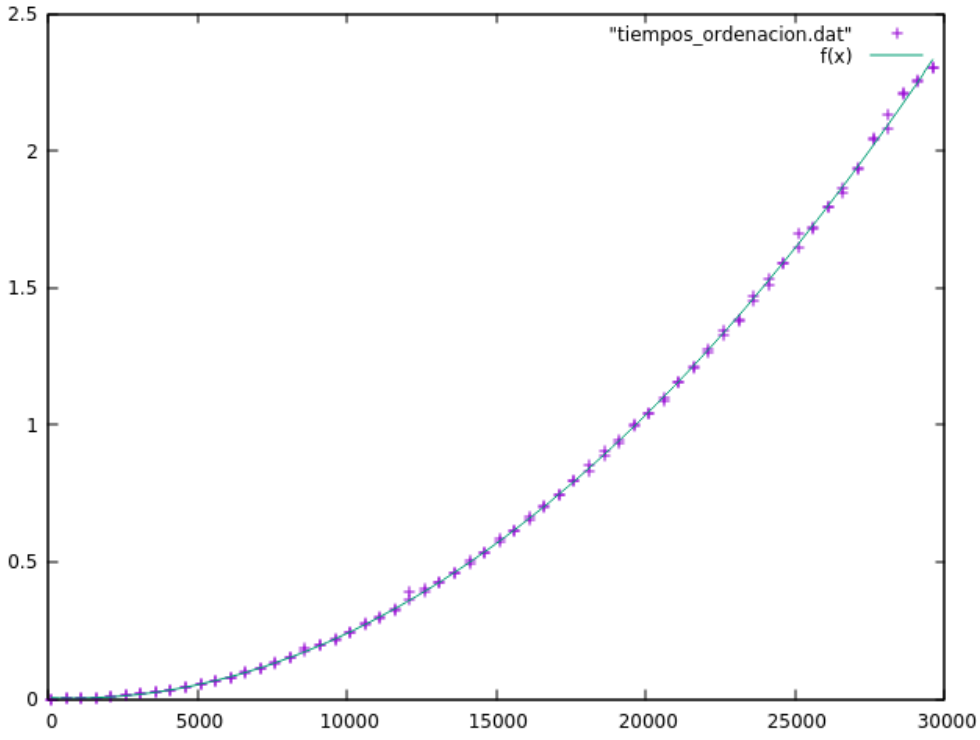
EJERCICIO 2

Tras superponer las dos gráficas(la de eficiencia empírica y la de f(x) = ax2 + bx+ c) hemos obtenido los siguientes resultados junto con la gráfica adjunta:

Final set of parameters		Asymptotic Standard Error	
==========		==========	
a	= 2.81514e-09	+/- 1.5e-11	(0.5329%)
b	= -4.72115e-06	+/- 4.605e-07	(9.754%)
c	= 0.00500093	+/- 0.002959	(59.17%)

correlation matrix of the fit parameters:

	a	b	c
a	1.000		
b	-0.968	1.000	
c	0.738	-0.861	1.000



Donde podemos apreciar que ambas gráficas (eficiencia teórica y práctica) siguen la misma curva, y se superponen dado que son prácticamente iguales.

EJERCICIO 3

```
#include <iostream>
#include <cstdlib>  // Para generación de números pseudoaleatorios
#include <chrono>   // Recursos para medir tiempos
using namespace std;
using namespace std::chrono;

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;
}
void sintaxis()
{
  cerr << "Sintaxis:" << endl;
  cerr << "  TAM: Tamaño del vector (>0)" << endl;
  cerr << "  VMAX: Valor máximo (>0)" << endl;
  cerr << "Se genera un vector de tamaño TAM con elementos aleatorios en [0,VMAX[" << endl;
  exit(EXIT_FAILURE);
}

int main(int argc, char * argv[])
{
  // Lectura de parámetros
  if (argc!=3)
    sintaxis();
  int tam=atoi(argv[1]);     // Tamaño del vector
  int vmax=atoi(argv[2]);    // Valor máximo
  if (tam<=0 || vmax<=0)
    sintaxis();

  // Generación del vector aleatorio
  int *v=new int[tam];       // Reserva de memoria
  srand(time(0));            // Inicialización del generador de números pseudoaleatorios
  for (int i=0; i<tam; i++)  // Recorrer vector
    v[i] = i;    // Generar aleatorio [0,tam[

 high_resolution_clock::time_point start,//punto de inicio
                                   end; //punto de fin
 duration<double> tiempo_transcurrido;  //objeto para medir la duración de end y start

 start = high_resolution_clock::now(); //iniciamos el punto de inicio

  int x = tam+1;  // Buscamos un valor que no está en el vector
  operacion(v,tam,x,0,tam-1);
 end = high_resolution_clock::now(); //anotamos el punto de de fin

//el tiempo transcurrido es
 tiempo_transcurrido  = (duration_cast<duration<double> >(end - start));

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

  delete [] v;     // Liberamos memoria dinámica
}
```

Algoritmo utilizado:

```
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;
}
```

El algoritmo anterior recibe como parámetros un vector de enteros, el tamaño del vector, un entero x y los extremos inferior y superior de un intervalos.
El algoritmo busca en dicho vector de enteros, el valor x de la siguiente forma:
En primer lugar se calcula el punto medio del intervalo. Posteriormente se compara si el valor que contiene el vector en dicho punto medio es igual a X, en cuyo caso, la variable booleana recibe el valor “true” y se devolverá la posición donde se encuentra dicho valor.
Sin embargo, si estos valores no coinciden(v[med] y x), en caso de que v[med] < x, el extremo inferior del intervalo pasa a ser la posición siguiente a med (med + 1), y volvería a realizarse la búsqueda en dicho intervalo, volviendo a calcular el punto medio del nuevo intervalo. Por otro lado, en caso de que v[med] > x, se realiza el mismo procedimiento pero manteniendo el extremo inferior del intervalo igual, en este caso varía el extremo superior del intervalo que pasaría a ser la posición anterior al punto medio (med – 1).

La eficiencia teórica del algoritmo sería la siguiente:

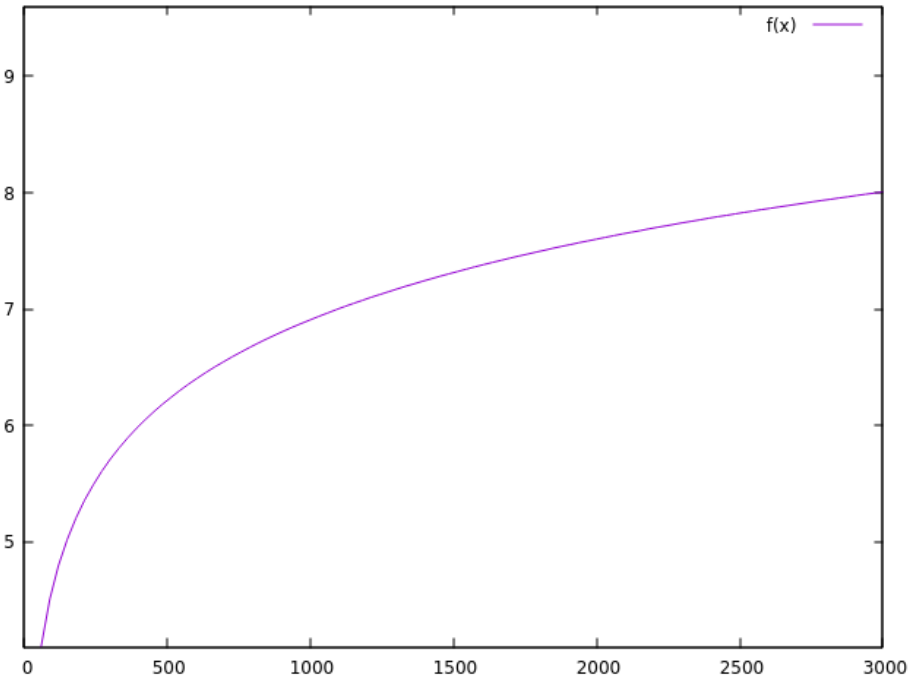
Linea 3: Asignación; 1 = O(1)
Linea 4: Bucle while; $O(\log_2(n))$
Linea 5: Asignación, suma y división; 1 + 1 + 1 = 3 = O(1)
Linea 6: Comparación; 1 = O(1)
Linea 7: Asignación; 1 = O(1)
Linea 8: Comparación; 1 = O(1)
Linea 9: Asignación; 1 = O(1)
Linea 11: Asignación; 1 = O(1)
Linea 14: Devolución; 1 = O(1)
Linea 16: Devolución; 1 = O(1)

Finalmente = $1 + \sum(2+ 3 + 1 + 1 + 1 + 1 + 1 + 1 + 1) = O(\log_2(n))$

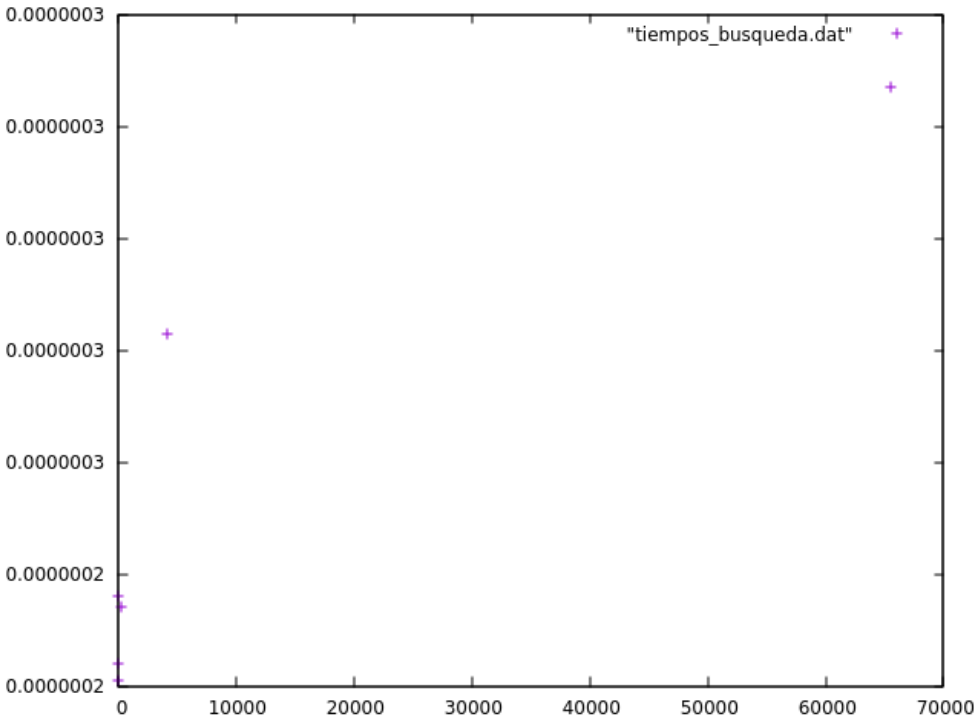
(La sumatoria va desde i= 0 hasta $\log_2(n)$)

Por tanto podemos ver que la eficiencia teórica de dicho algoritmo corresponde a $O(\log_2(n))$

Tras estudiar la eficiencia teórica del algoritmo hemos obtenido la siguiente gráfica:

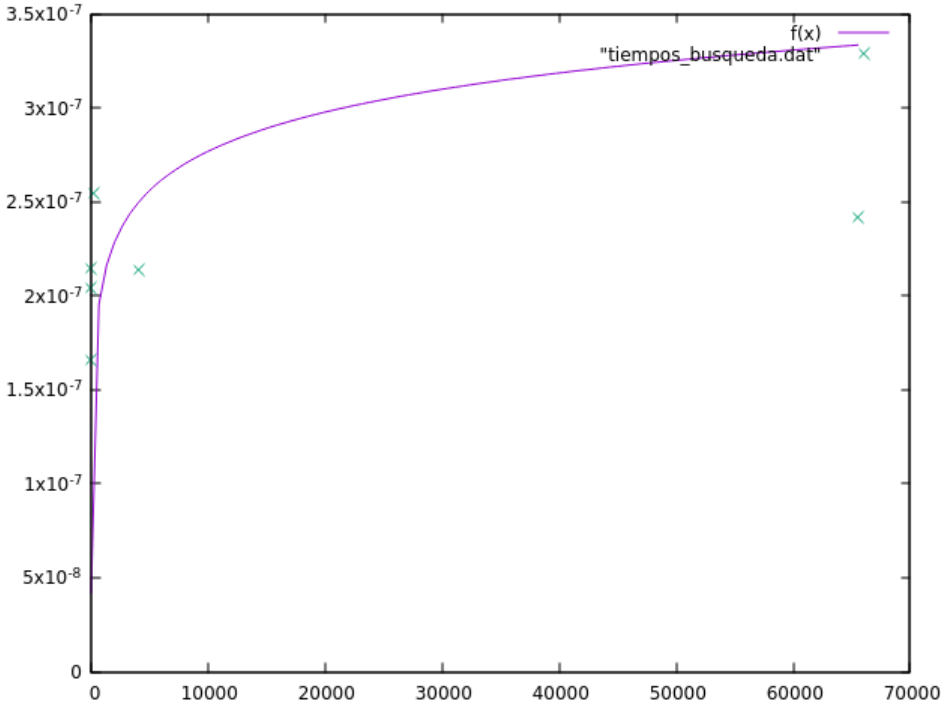


Por otro lado, la gráfica obtenida gracias a la eficiencia empírica es:



El problema que encontramos en este algoritmo es el crecimiento logarítmico del tiempo de ejecución frente al tamaño del vector que se le pasa, lo cual no nos permite comprobar correctamente la eficiencia empírica del programa.
Por tanto, para solucionar dicho problema, lo que podemos hacer es añadirle más tamaños de vector al script que ejecuta dicho programa. Así podremos obtener más puntos que nos hagan ver la gráfica de mejor forma.

Y si superponemos las gráficas de eficiencia teórica y eficiencia empírica, ajustando la teórica a la empírica y utilizando f(x) = a*log(x) + b, obtenemos:

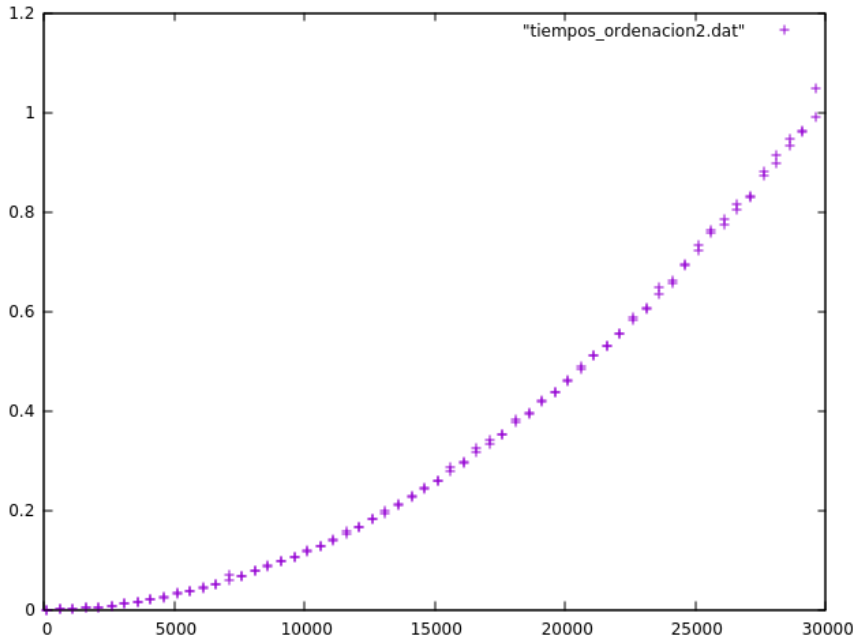


EJERCICIO 4

Ordenación con vector ya ordenado:

```
#include <iostream>
#include <ctime> // Recursos para medir tiempos
#include <cstdlib> // Para generación de números aleatorios

using namespace std;

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;
            }
}

void sintaxis(){
    cerr << "Sintaxis: " << endl;
    cerr << "  TAM:  Tamanio 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) // Lectura de parámetros
        sintaxis();

    int tam = atoi(argv[1]);  // Tamanio del vector
    int vmax = atoi(argv[2]); // Valor máximo

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

    /* Generamos el vector aleatorio
    int * v = new int[tam];  // Reserva de memoria
    srand(time(0));          // Inicializamos el generador de números aleatorios

    for(int i = 0; i < tam; i++) // Recorrer vector
        v[i] = rand() % vmax;    // Generar aleatorio [0, VMAX[
    */

    // Generamos vector ordenado
    int * v = new int[tam];

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

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

    ordenar(v, tam); // Ordenamos el vector

    clock_t tfin; // Anotamos el tiempo de finalización
    tfin = clock();

    // Mostramos resultados (Tam del vector y tiempo de ejecución en seg.)
    cout << tam << "\t" << (tfin-tini) / (double) CLOCKS_PER_SEC << endl;

    delete[] v; // Liberamos memoria dinámica
}
```

Tras realizar la ordenación de un vector que ya está ordenado y comprobar su eficiencia empírica, obtenemos la siguiente gráfica:



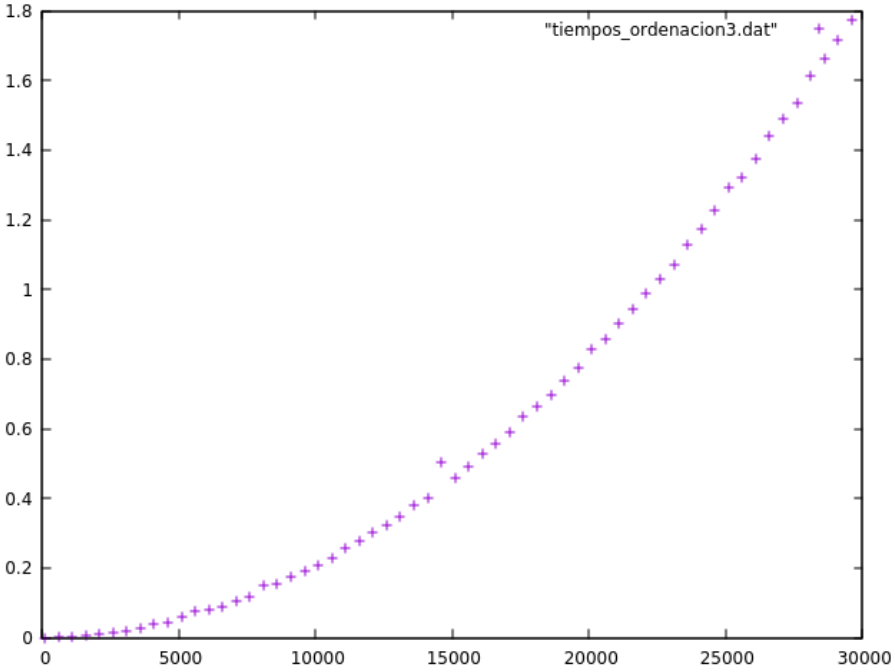
Donde hemos utilizado el siguiente código para general el vector ordenado:

```
int * v = new int[tam];

for(int i = 0; i < tam; i++)
    v[i] = i;
```

Ordenación con un vector ordenado de forma inversa:

```
#include <iostream>
#include <ctime> // Recursos para medir tiempos
#include <cstdlib> // Para generación de números aleatorios

using namespace std;

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;
            }
}

void sintaxis(){
    cerr << "Sintaxis: " << endl;
    cerr << "  TAM:  Tamanio 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) // Lectura de parámetros
        sintaxis();

    int tam = atoi(argv[1]);  // Tamanio del vector
    int vmax = atoi(argv[2]); // Valor máximo

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

    /* Generamos el vector aleatorio
    int * v = new int[tam];  // Reserva de memoria
    srand(time(0));          // Inicializamos el generador de números aleatorios

    for(int i = 0; i < tam; i++) // Recorrer vector
        v[i] = rand() % vmax;    // Generar aleatorio [0, VMAX[
    */

    // Generamos vector ordenado de forma inversa
    int * v = new int[tam];
    int a = tam;
    for(int i = 0; i < tam; i++){
        v[i] = a;
        a--;
    }

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

    ordenar(v, tam); // Ordenamos el vector

    clock_t tfin; // Anotamos el tiempo de finalización
    tfin = clock();

    // Mostramos resultados (Tam del vector y tiempo de ejecución en seg.)
    cout << tam << "\t" << (tfin-tini) / (double) CLOCKS_PER_SEC << endl;

    delete[] v; // Liberamos memoria dinámica
}
```

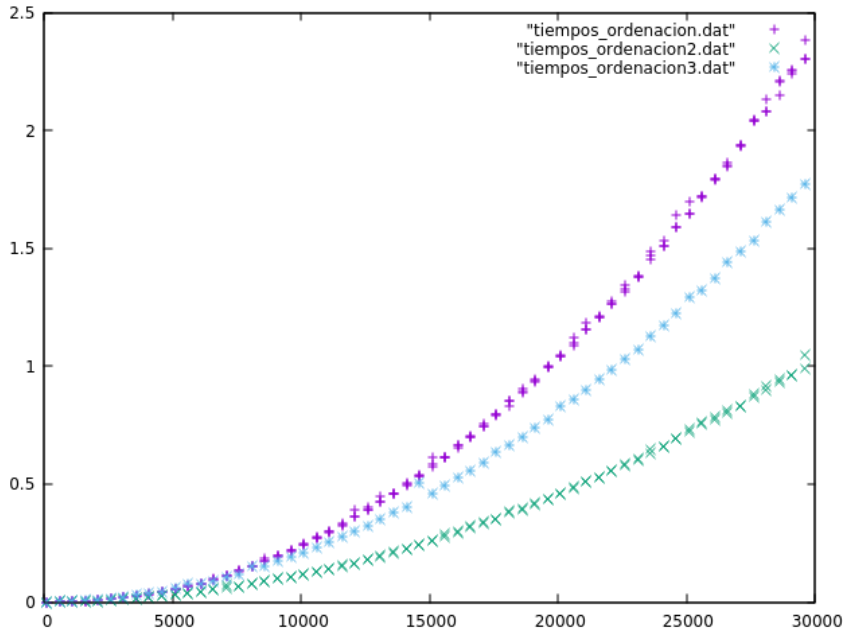
Posteriormente, tras comprobar la eficiencia al ordenar un vector que está ordenado de forma inversa, obtenemos la siguiente gráfica:



Donde hemos utilizado el siguiente código para general el vector ordenado de forma inversa:

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

Finalmente si comparamos ambas gráficas obtenidas, con la gráfica de la eficiencia empírica del ejercicio 1, vemos la siguiente gráfica:



En comparación, podemos apreciar que el que obtiene mejores tiempos de ejecución es el número 2, que es el caso en el que utilizamos un vector ya ordenado. Posteriormente, vemos que el caso en el que utilizamos un vector ordenado de forma inversa obtiene tiempos un poco más lentos, pero el más lento de los tres casos es el que recurre a un vector generado de forma aleatoria.

EJERCICIO 5

```
#include <iostream>
#include <ctime> // Recursos para medir tiempos
#include <cstdlib> // Para generación de números aleatorios

using namespace std;

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;
                int aux = v[j];
                v[j] = v[j+1];
                v[j+1] = aux;
            }
    }
}

void sintaxis(){
    cerr << "Sintaxis: " << endl;
    cerr << "  TAM:  Tamanio 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) // Lectura de parámetros
        sintaxis();

    int tam = atoi(argv[1]);  // Tamanio del vector
    int vmax = atoi(argv[2]); // Valor máximo

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

    // Generamos el vector aleatorio
    int * v = new int[tam];  // Reserva de memoria
    srand(time(0));          // Inicializamos el generador de números aleatorios

    for(int i = 0; i < tam; i++) // Recorrer vector
        v[i] = rand() % vmax;    // Generar aleatorio [0, VMAX[

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

    ordenar(v, tam); // Ordenamos el vector

    clock_t tfin; // Anotamos el tiempo de finalización
    tfin = clock();

    // Mostramos resultados (Tam del vector y tiempo de ejecución en seg.)
    cout << tam << "\t" << (tfin-tini) / (double) CLOCKS_PER_SEC << endl;

    delete[] v; // Liberamos memoria dinámica
}
```

El nuevo algoritmo de ordenación por burbuja es:

```
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;
                int aux = v[j];
                v[j] = v[j+1];
                v[j+1] = aux;
            }
    }
}
```

Y si estudiamos la eficiencia teórica de este algoritmo podemos ver que:

Linea 2: Asignación; O(1)
Linea 3: Bucle for; $\sum(1+1+1+1) = \sum(4) = O(n)$
Linea 4: Asignación; O(1)
Linea 5: Bucle for; $\sum(1+1+1) = \sum(3) = O(n)$
Linea 6: Comparación; O(1)
Linea 7: Asignación; O(1)
Linea 8: Asignación; O(1)
Linea 9: Asignación; O(1)
Linea 10: Asignación; O(1)

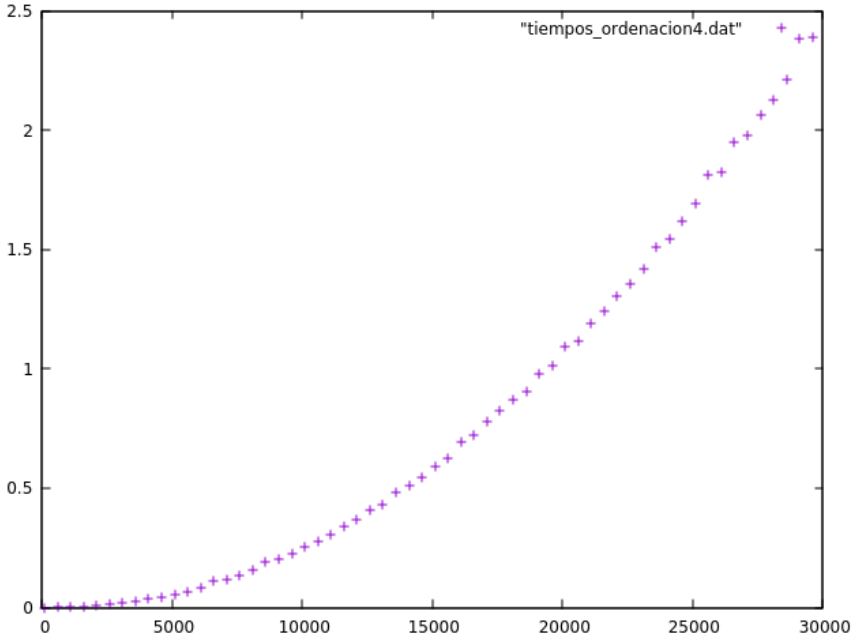
Finalmente = $1 + \sum(4 + 1 + (\sum(3 + 1 + 1 + 1 + 1 + 1)) = 1 + n * n = O(n^2)$

(Ambas sumatorias van desde i, j=0 hasta n-1 y n-i-1 respectivamente)

Por tanto, podemos ver que si realizamos la suma de cada una de las eficiencias obtenidas anteriormente, vemos que la eficiencia de este algoritmo es $O(n^2)$

Sin embargo, en el caso de que el vector que le pasemos al algoritmo ya esté ordenado, la eficiencia teórica que se obtiene es O(n) ya que no tendría que realizar ninguna iteración en el segundo ciclo for.

Por otro lado, podemos ver la gráfica de eficiencia empírica obtenida:

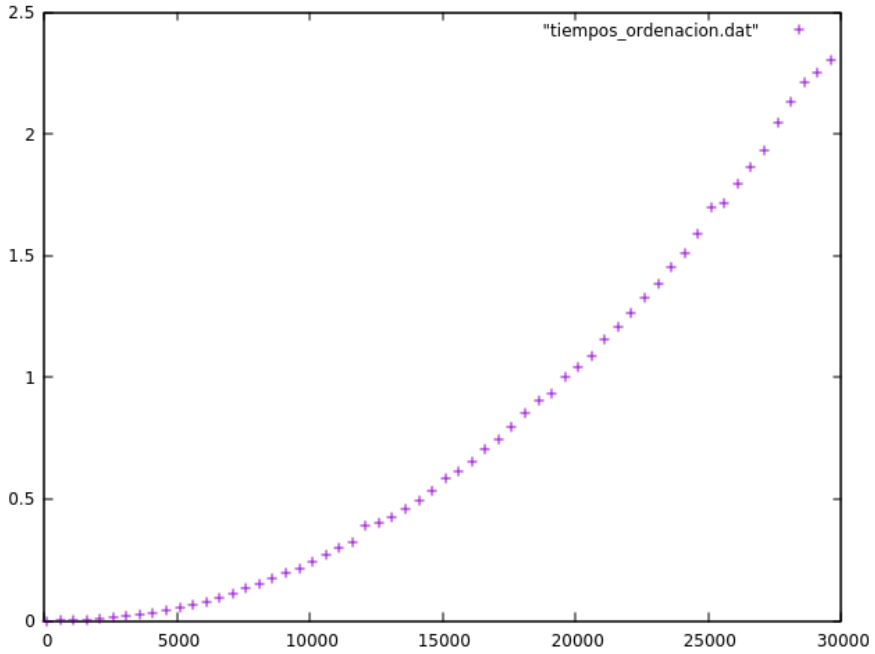


EJERCICIO 6

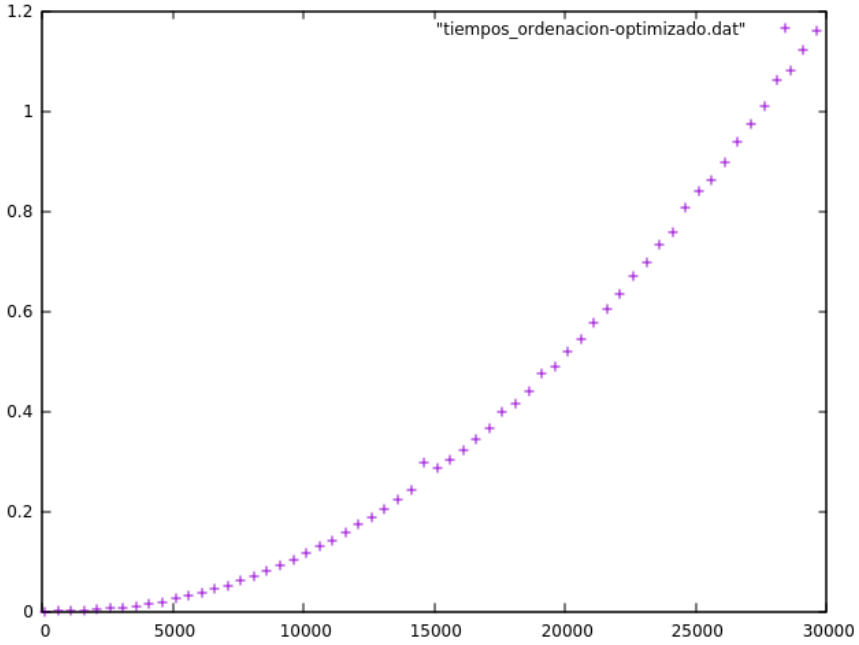
El código del ejercicio 6 ha sido el mismo que el ejercicio 1, la única diferencia es que a la hora de compilar hemos utilizado la orden:

g++ -O3 ordenacion.cpp -o ordenacion_optimizado

La curva de eficiencia empírica del ejercicio de ordenación sin optimización es la siguiente:



Y tras realizar la compilación con optimización y obtener la gráfica resultante de los tiempos, obtenemos:



Donde es posible apreciar una gráfica igual a la del programa sin optimización, sin embargo, si nos fijamos en el eje de ordenadas, vemos que los tiempos de ejecución que se representan llegan hasta 1.2, la mitad que en la primera gráfica, donde el tiempo de ejecución más alto es 2.5, por lo que vemos que con optimización los tiempos se han reducido a la mitad, manteniéndose la misma proporción tiempo de ejecución - tamaño de vector.

EJERCICIO 7

```
#include <iostream>
#include <stdlib.h> // Incluye srand() y rand()
#include <time.h>

using namespace std;

float ** multiplica_matrices(int ** m1, int ** m2, int tam){

    // Creamos la matriz resultado
    float ** resultado = new float * [tam];

    for(int i = 0; i < tam; i++)
        resultado[i] = new float [tam];

    // Multiplicamos las matrices
    for(int i = 0; i < tam; i++)
        for(int j = 0; j < tam; j++)
            for(int k = 0; k < tam; k++)
                resultado[i][j] += m1[i][k] * m2[k][j];

    return resultado;
}

void sintaxis()
{
  cerr << "Sintaxis:" << endl;
  cerr << "  TAM: Tamaño de la matriz (>0)" << endl;
  cerr << "  MAX: Valor maximo de la matriz (>=0)" << endl;
  cerr << "Se genera una matriz de tamaño TAM con elementos aleatorios" << endl;
  exit(EXIT_FAILURE);
}

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

    // Lectura de parámetros
    if (argc!=3)
      sintaxis();
    int tam = atoi(argv[1]);     // Tamaño de las matrices
    int max = atoi(argv[2]);
    if (tam<=0)
      sintaxis();

    // Generación de dos matrices aleatorias
    int ** m1 = new int * [tam];
    int ** m2 = new int * [tam];

    for(int i = 0; i < tam; i++)
        m1[i] = new int [tam];

    for(int i = 0; i < tam; i++)
        m2[i] = new int [tam];

    // Rellenamos las dos matrices con números aleatorios
    srand(time(0));

    // m1
    for(int i = 0; i < tam; i++)
        for(int j = 0; j < tam; j++)
            m1[i][j] = rand() % max;

    // m2
    for(int i = 0; i < tam; i++)
        for(int j = 0; j < tam; j++)
            m2[i][j] = rand() % max;

    // Generamos la matriz resultante
    float ** resultado = new float * [tam];

    for(int i = 0; i < tam; i++)
        resultado[i] = new float [tam];

    clock_t t_ini; // Anotamos tiempo de inicio
    t_ini = clock();

    resultado = multiplica_matrices(m1, m2, tam); // Multiplicamos las matrices

    clock_t t_fin; // Anotamos tiempo de finalización
```

```
    t_fin = clock();

    // Mostramos resultados (Tam de la matriz y tiempo de ejecución en seg)
    cout << tam << "\t" << (t_fin - t_ini) / (double) CLOCKS_PER_SEC << endl;

    return 0;
}
```

En primer lugar, estudiamos la eficiencia teórica de el algoritmo que hemos utilizado:

```
float ** multiplica_matrices(int ** m1, int ** m2, int tam){
    float ** resultado = new float * [tam];

    for(int i = 0; i < tam; i++)
        resultado[i] = new float [tam];

    for(int i = 0; i < tam; i++)
        for(int j = 0; j < tam; j++)
            for(int k = 0; k < tam; k++)
                resultado[i][j] += m1[i][k] * m2[k][j];

    return resultado;
}
```

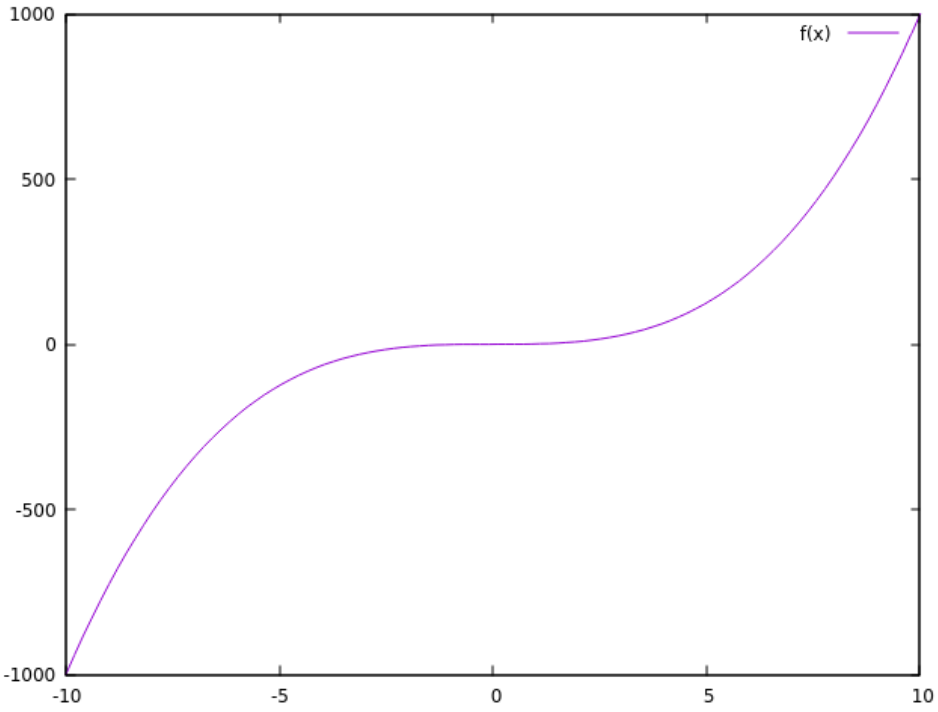
Linea 2: Asignación; O(1)
Linea 4: Bucle for; $\sum(1+1+1) = O(n)$
Linea 5: Asignación; O(1)
Linea 7: Bucle for; $\sum(1+1+1) = O(n)$
Linea 8: Bucle for; $\sum(1+1+1) = O(n)$
Linea 9: Bucle for; $\sum(1+1+1) = O(n)$
Linea 10: Asignación, suma y producto; 1+1+1 = 3 = O(1)
Linea 12: Devolución; O(1)

Finalmente = $1 + \sum(3+1) + \sum(3 + \sum(3 + \sum(3 + 3))) + 1 = 1 + n + n^3 + 1 = O(n^3)$

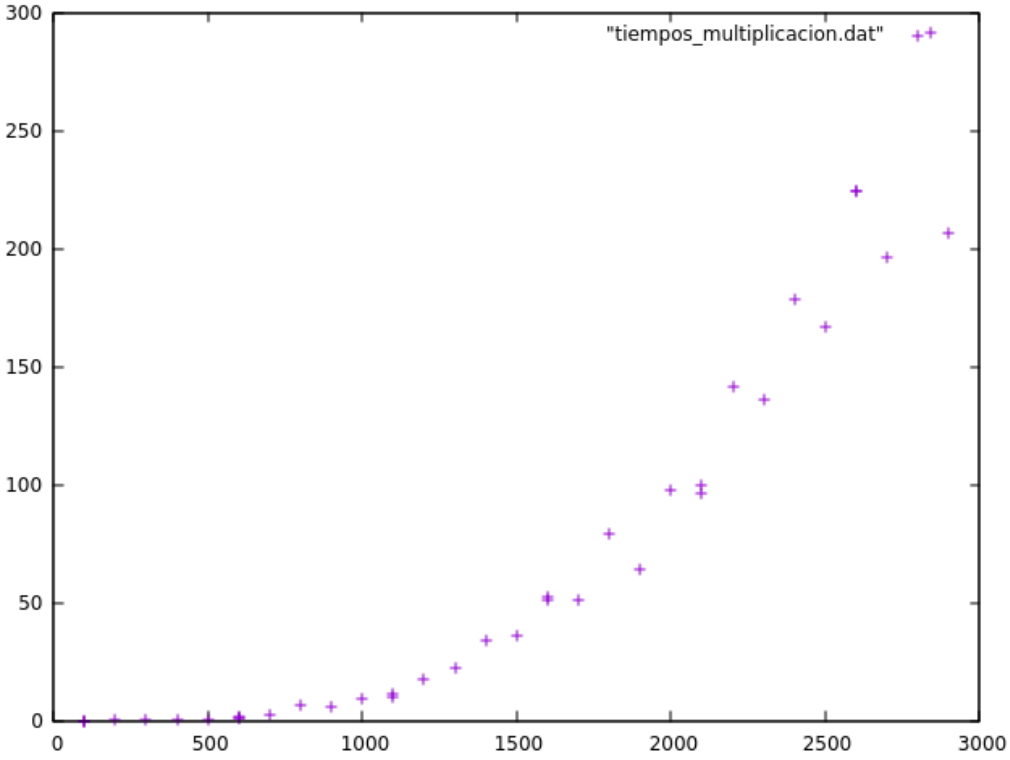
(Las sumatorias van desde i, j, k = 0 hasta n)

Por tanto la eficiencia de nuestro algoritmo es $O(n^3)$

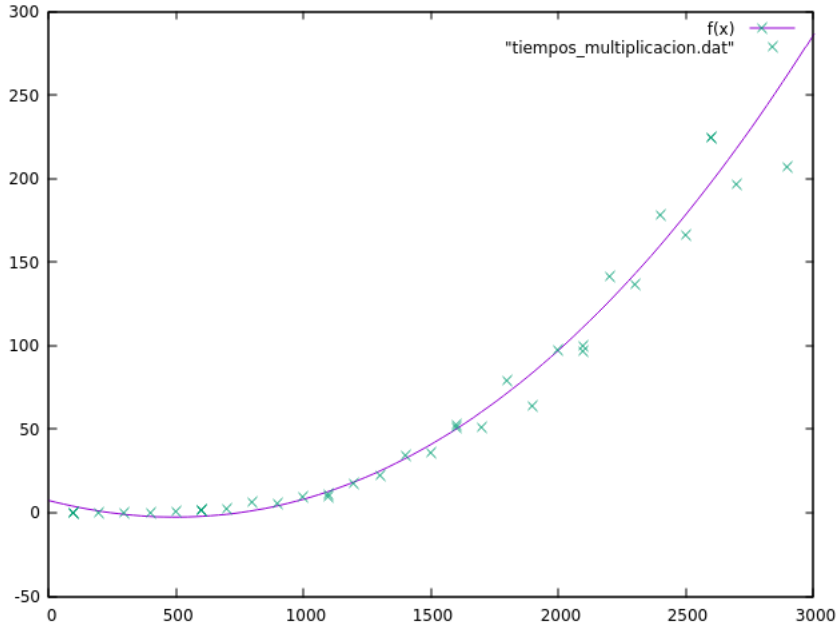
Si pintamos la gráfica de la eficiencia teórica obtendremos:



Y tras realizar la gráfica de eficiencia empírica obtenemos:



Y si realizamos el ajuste de las dos gráficas, con $f(x) = a*x^3 + b*x^2 + c*x + d$, obtenemos:



Donde podemos ver que la eficiencia teórica y empírica siguen la misma curva hasta mas o menos el tamaño de vector 2000, momento en el cual los puntos de la gráfica empiezan a dispersarse poco a poco, siguiendo, eso sí, la misma curva de eficiencia teórica.

EJERCICIO 8

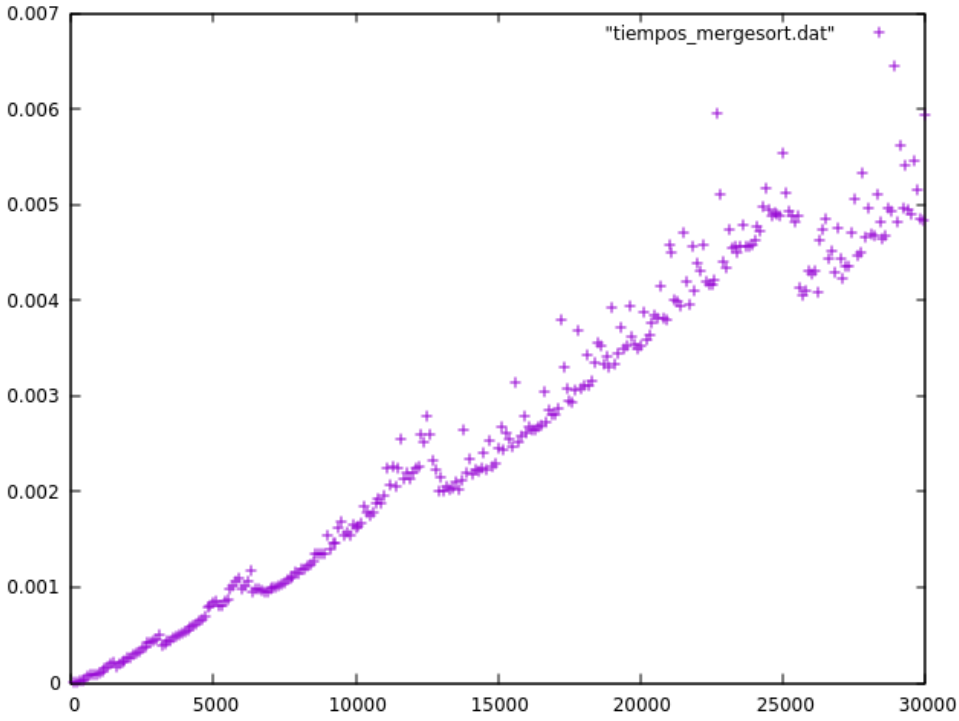
```
/**
   @file Ordenación por mezcla
*/

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

using namespace std;

/* ********************************************************** */
/*  Método de ordenación por mezcla  */

/**
   @brief Ordena un vector por el método de mezcla.

   @param T: vector de elementos. Debe tener num_elem elementos.
             Es MODIFICADO.
   @param num_elem: número 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 método de mezcla.

   @param T: vector de elementos. Tiene un número de elementos
                  mayor o igual a final. Es MODIFICADO.
   @param inicial: Posición que marca el inicio de la parte del
                  vector a ordenar.
   @param final: Posición detrás de la última 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 método de inserción.

   @param T: vector de elementos. Debe tener num_elem elementos.
             Es MODIFICADO.
   @param num_elem: número 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 inserción.
*/
inline static
void insercion(int T[], int num_elem);

/**
   @brief Ordena parte de un vector por el método de inserción.

   @param T: vector de elementos. Tiene un número de elementos
                  mayor o igual a final. Es MODIFICADO.
   @param inicial: Posición que marca el inicio de la parte del
                  vector a ordenar.
   @param final: Posición detrás de la última 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 inserción.
*/
```

```
static void insercion_lims(int T[], int inicial, int final);

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

   @param T: vector de elementos. Tiene un número de elementos
                  mayor o igual a final. Es MODIFICADO.
   @param inicial: Posición que marca el inicio de la parte del
                  vector a escribir.
   @param final: Posición detrás de la ultima de la parte del
                  vector a escribir
                              inicial < final.
   @param U: Vector con los elementos ordenados.
   @param V: Vector con los elementos ordenados.
             El numero 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[]);

/**
   Implementación 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);
```

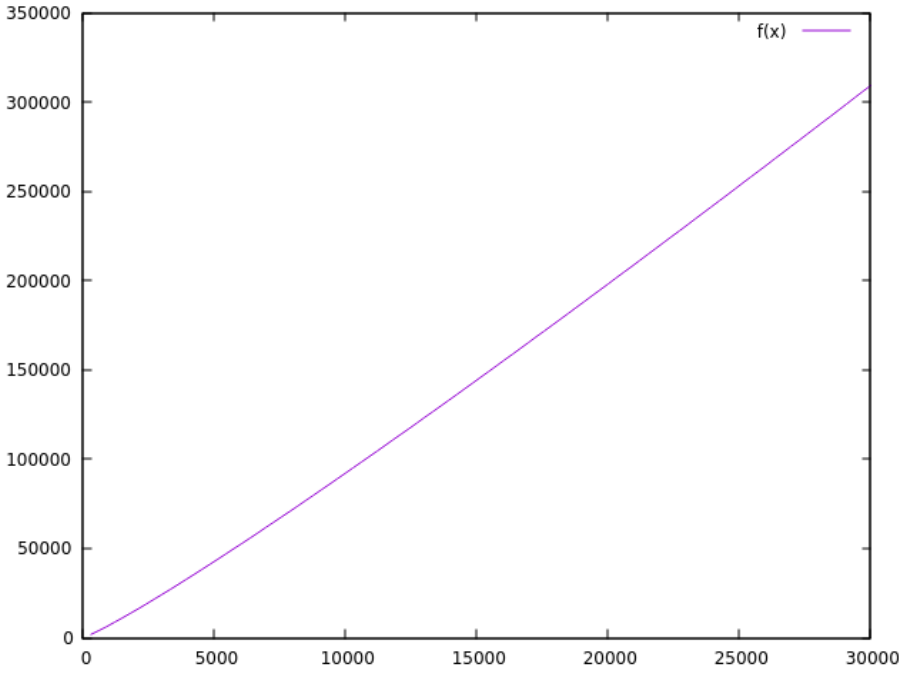
```
      delete [] U;
      delete [] V;
    };
}

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;
};
```

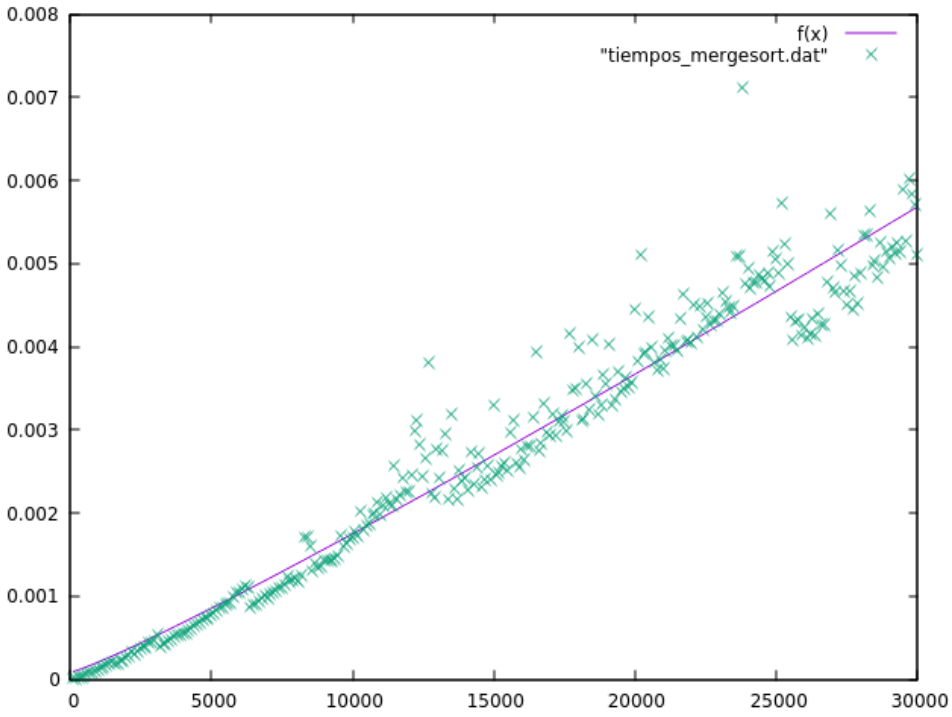
Tras comprobar la eficiencia empírica del algoritmo mergesort hemos obtenido la siguiente gráfica:



Posteriormente, como el propio enunciado dice, la eficiencia teórica del algoritmo es x*log(x), por lo que si representamos la gráfica de eficiencia teórica obtenemos:



Y finalmente, si realizamos el ajuste lineal entre la gráfica de eficiencia empírica y la gráfica de eficiencia teórica, utilizando como f(x) = a*x*log(x) + b, obtenemos:

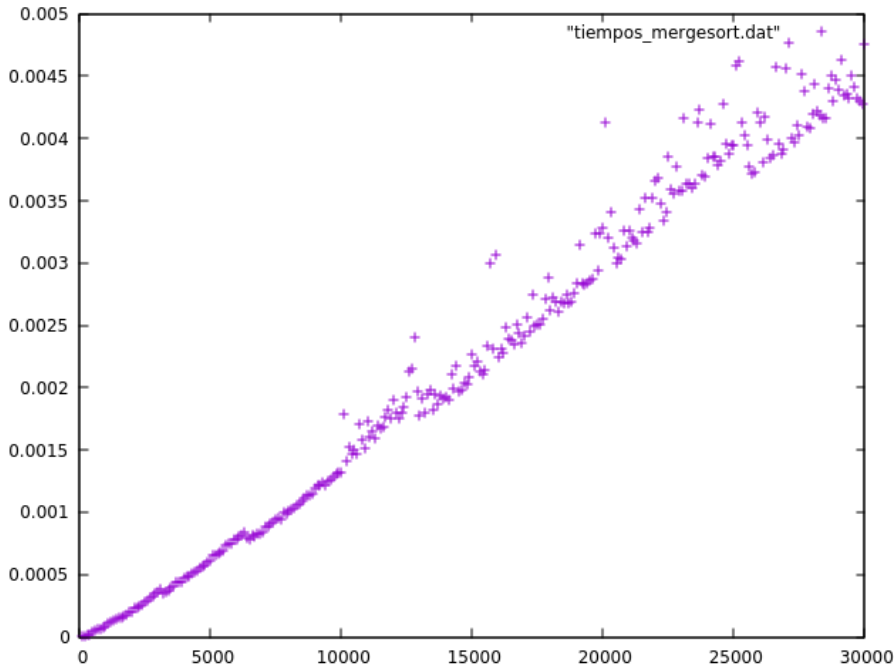


Donde podemos ver que más o menos, la eficiencia empírica sigue el recorrido de la teórica con una ligera dispersión.

Estudio de diferentes valores de UMBRAL_MS

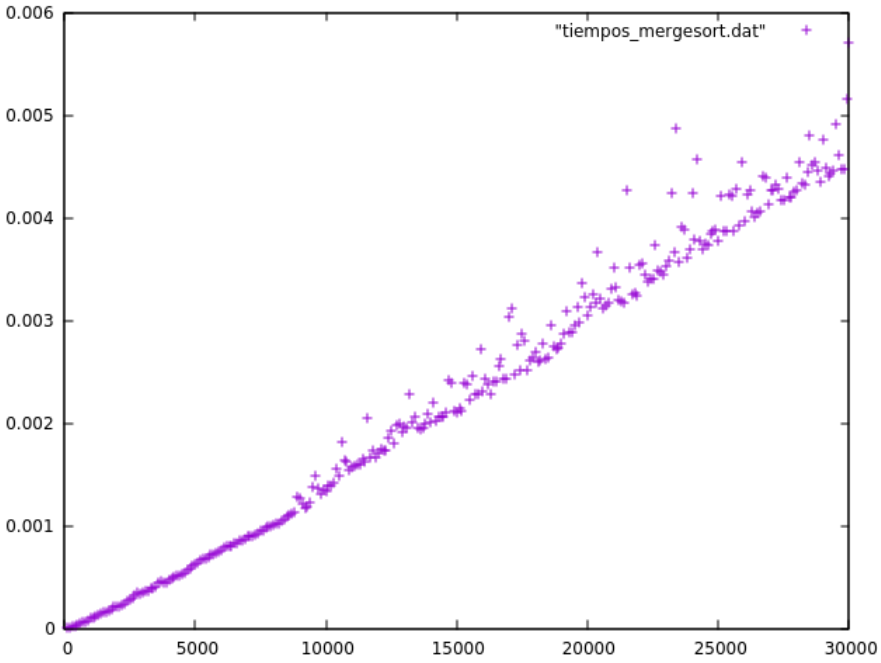
El valor original de UMBRAL_MS con el que hemos realizado el estudio anterior es 100.

A continuación se muestra una gráfica de la eficiencia empírica con un valor de UMBRAL_MS = 50



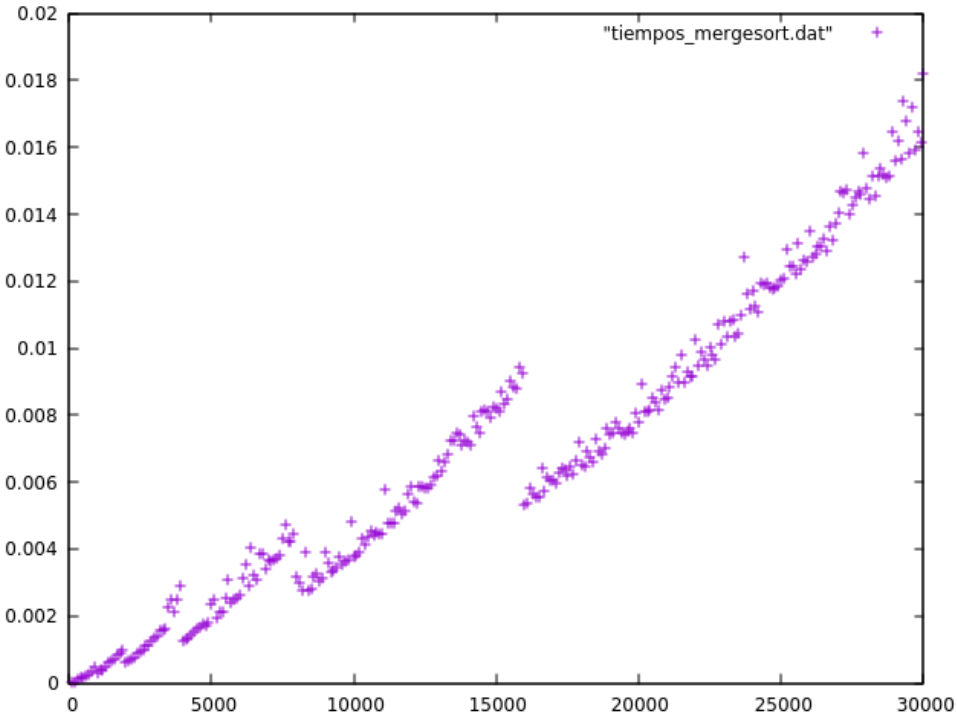
Donde podemos ver que el tiempo se ha reducido de 0.007 a 0.005 respecto a la gráfica obtenida con el valor original

A continuación se muestra una gráfica de la eficiencia empírica con un valor UMBRAL_MS = 10

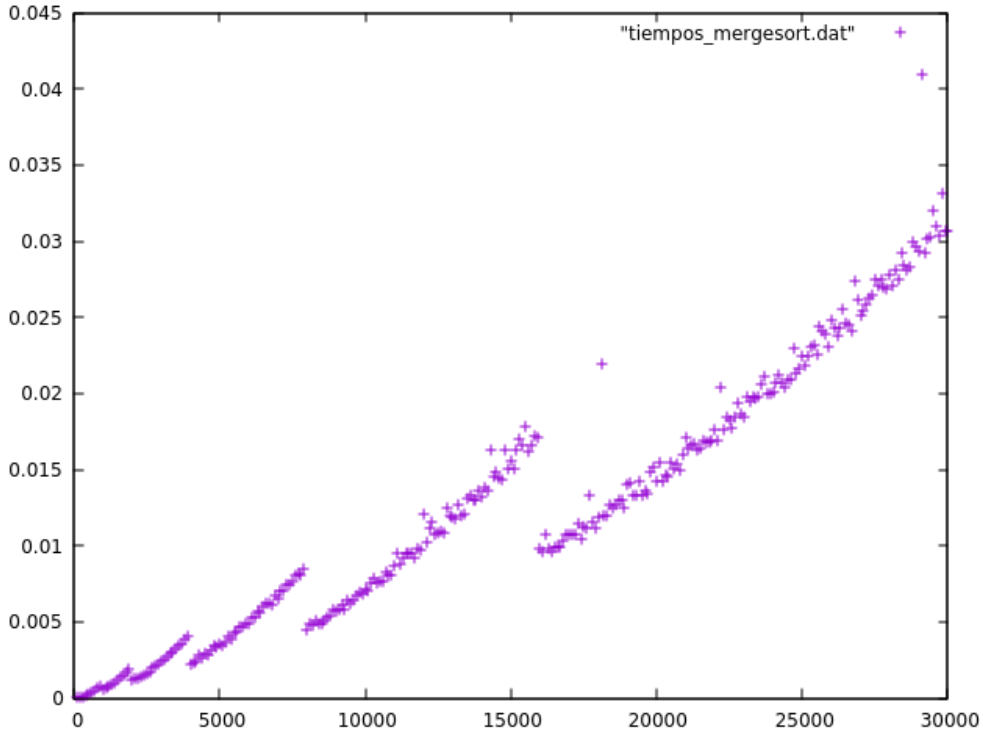


Donde los tiempos también se mantienen por debajo de 0.005

Gráfica de eficiencia empírica con un valor UMBRAL_MS = 500



Gráfica de eficiencia empírica con un valor UMBRAL_MS = 1000



Con esto podemos comprobar que cuanto más aumente el valor de UMBRAL_MS, vemos que la gráfica se fragmenta en varios segmentos que conforme aumenta el valor de UMBRAL_MS estos fragmentos son cada vez más verticales y además el tiempo ha aumentado bastante respecto al resto de ejecuciones con valores más pequeños, mientras que si reducimos el valor de UMBRAL_MS la gráfica es más “uniforme” y los tiempos se reducen.