Práctica 0

Condiciones de ejecución

Alumna: Inés Jiménez Díaz

Hardware: intel i7-11700F 2.5Ghz ROCKET,32gb RAM, tarjeta gráfica MSI GTX 1630

SO: Ubuntu 22.04.1 LTS

Compilador: g++

Opciones de compilación: -g -o -03

Ejercicio 1

En este ejercicio comprobaremos la eficiencia teórica y la eficiencia empírica del siguiente algoritmo:

```
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;
      }
    }
  }
```

Análisis de la eficiencia teórica en el peor caso

Línea 2: Hay 4 OE: Una asignación, una resta, una comparación y un incremento.

Línea 3: Hay 5 OE: igual que en la línea anterior, pero se realizan dos restas.

Línea 4: Hay 4 OE: Dos accesos a un vector, una suma y una comparación.

Línea 5: Hay 2 OE: Asignación y acceso al vector.

Línea 6: Hay 4 OE: Dos accesos, suma y asignación.

Línea 7: Hay 3 OE: Acceso, suma y asignación.

Por ello, la eficiencia del algoritmo es:

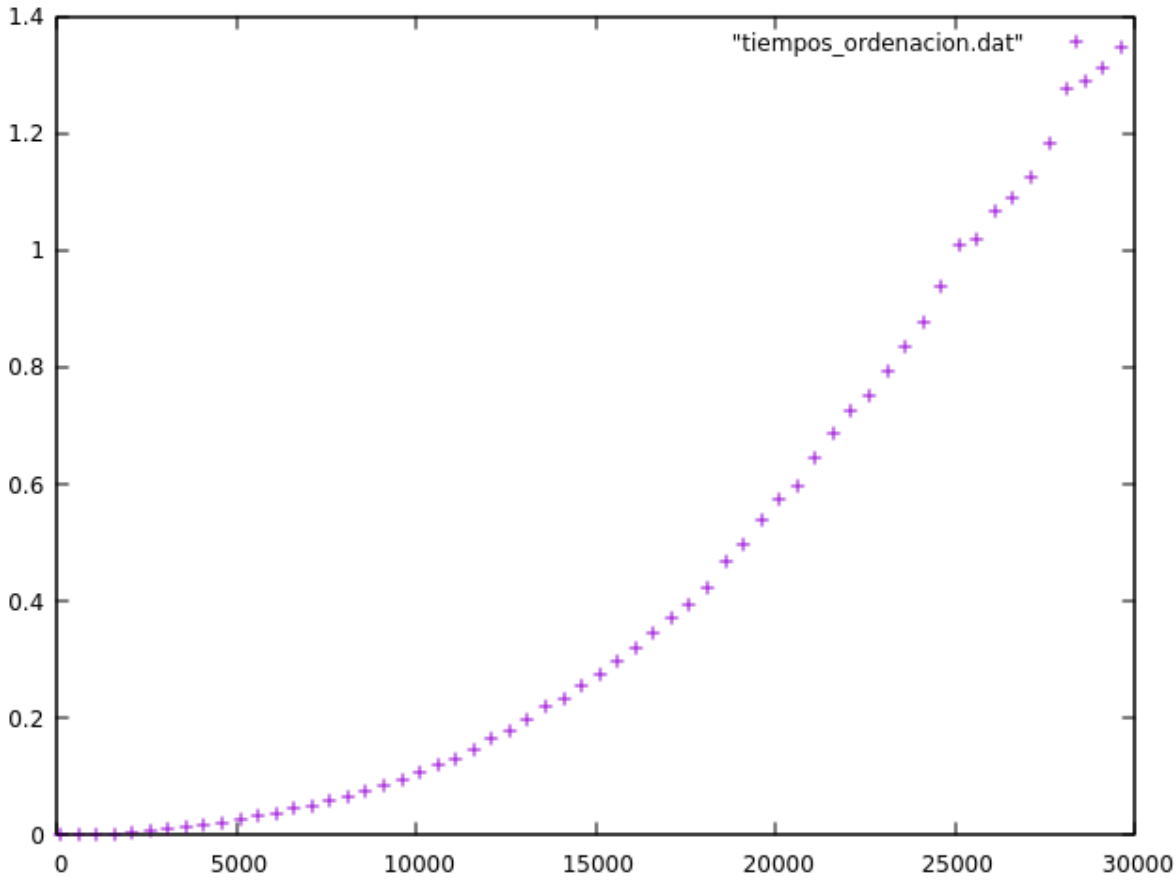
$$T(n) = \sum_{i=0}^{n-2} \left(4 + \sum_{j=0}^{n-i-2} 4 + 9\right) = \sum_{i=0}^{n-2} (4 + 13(n-i-1)) = \frac{13n^2-5n-8}{2}$$

Por tanto, afirmamos que $T(n) \in O(n^2)$, y el algoritmo de ordenación de burbuja es de orden de eficencia O(n²).

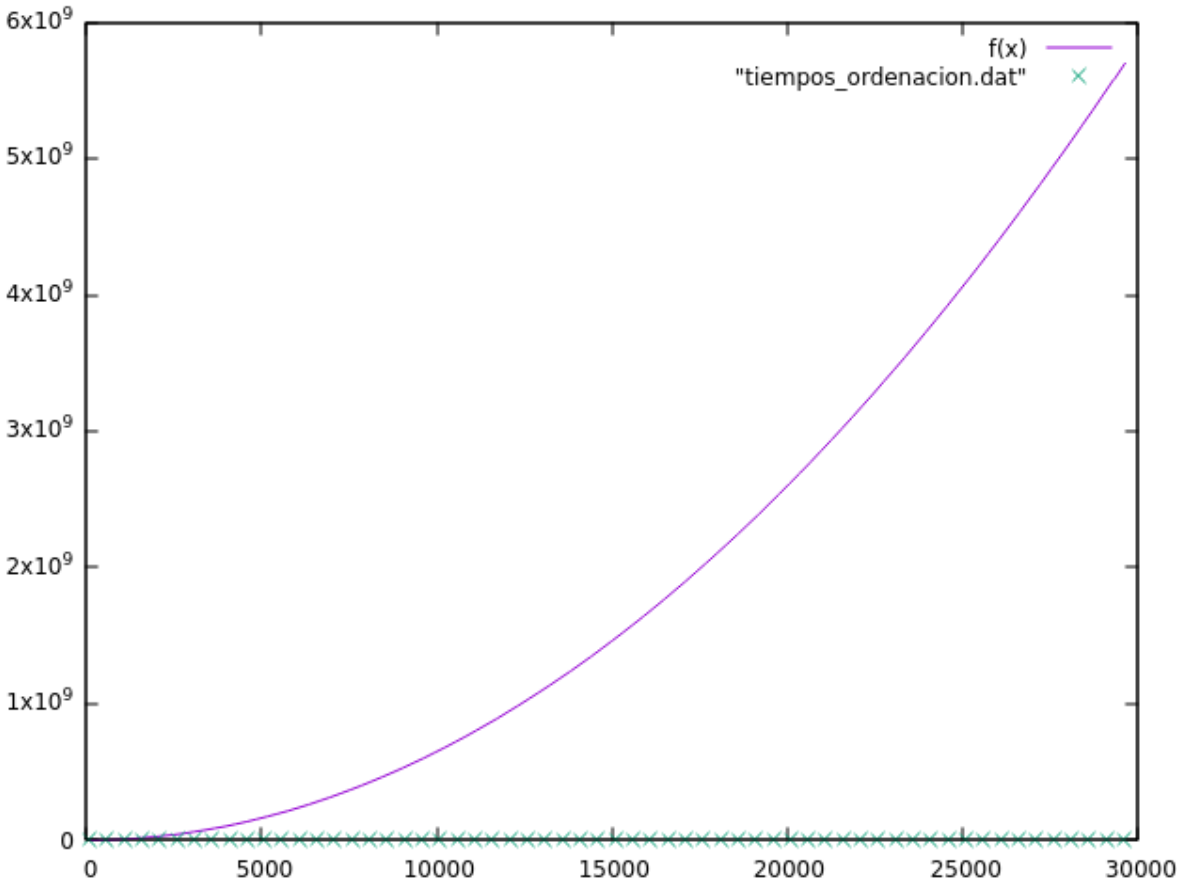
Creamos un código para analizar la eficencia empírica, haciendouso de la biblioteca ctime de c++:

```
#include <iostream>
#include <ctime>
#include <cstdlib>
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: 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[]){
  if(argc != 2)
    sintaxis();
  //lectura de los parámetros
  int tam = atoi(argv[1]);
  //Genera el vector aleatorio y reservar memoria:
  int * v=new int[tam];
  //Inicialización generador numeros pseudoaleatorios
  srand(time(0));
  for(int i = 0; i < tam; i++){
    //generar numeros aleatorios entre [0,vmax[
    v[i]=rand();
  }
  //Anotamos tiempo de inicio
  clock_t inicio;
  inicio = clock();

  ordenar(v,tam);

  //Anotamos el tiempo final
  clock_t fin;
  fin = clock();
  //Mostramos resultados.
  cout << tam << "\n" << (fin-inicio)/(double)CLOCKS_PER_SEC << endl;
  delete [] v;
}
```

Al analizar la eficiencia empírica del algoritmo, obtenemos la siguiente gráfica:



Si representamos superpuestas la función de la eficiencia teórica y la empírica, obtenemos lo siguiente:



Observamos que ambas curvas no coinciden al representarlas superpuestas. Esto se debe a que las constantes no coinciden, pues el costo de una operación elemental dependen de las condiciones de la máquina en la que se ejecuta.

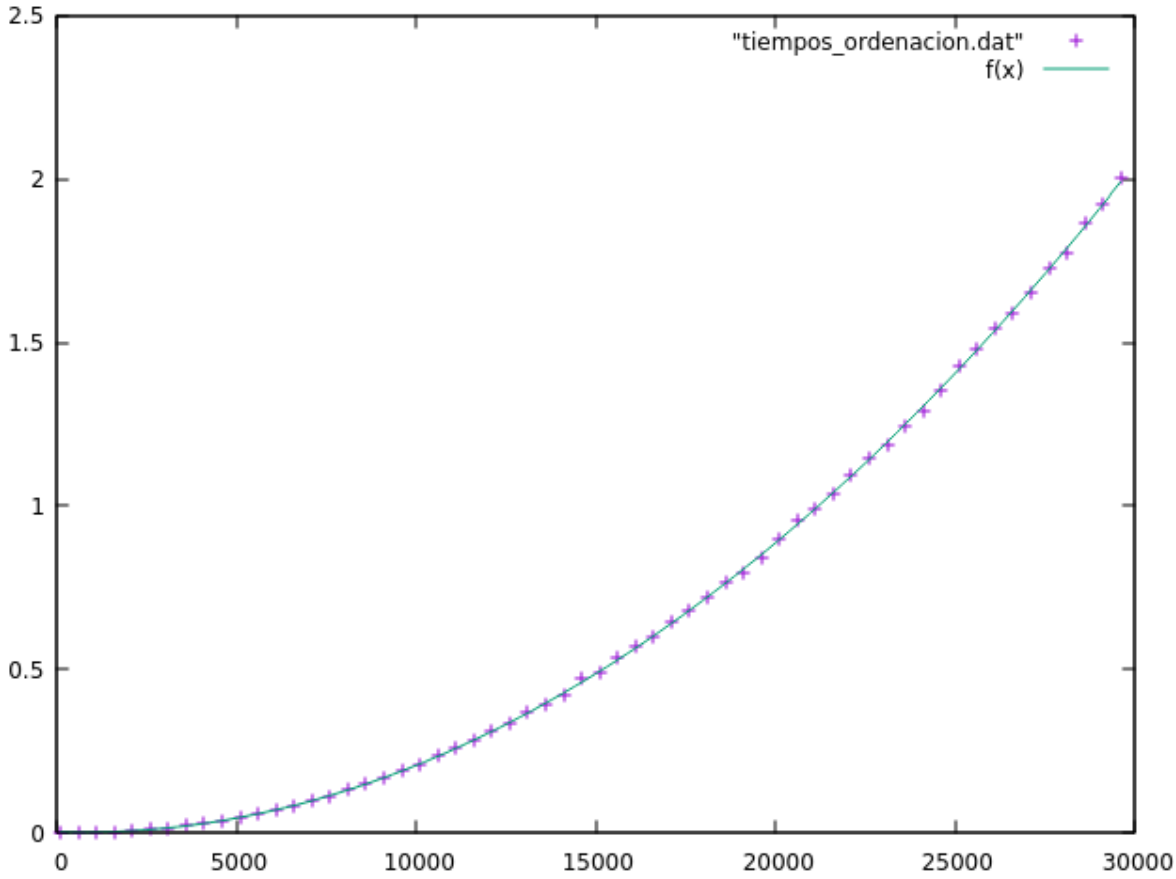
Dificultad: 4/10

Ejercicio 2

Mediante gnuplot podemos ajusar los puntos obtenidos al analizar la eficiencia empírica en una función de la forma f(n)=an²+bn+c. La función que obtenemos es:

f(n)=2.38528e^{-09}x² - 3.54123e-06x + 0.00439071.

Obtenemos que las consantes obtenidas son mucho más pequeñas que las calculadas para la eficencia teórica. La representación gráfica es la siguiente:



Dificultad: 5/10

Ejercicios 3

¿Qué hace este algoritmo?

Este algoritmo es una búsqueda binaria, donde dado un vector v de n elementos enteros, busca el entero x en el. Para ello, se pasan como parámetros el inicio y el final de dicho vector. La función devuelve la posición donde se ha encontrado el elemento, o -1 si no estaba en el vector.

Es imprescindible que el vector esté ordenado ya que el procedimiento es el siguiente:

1. Se establece la mitad= $\frac{inf+sup}{2}$
2. Se comprueba si el elemento x está en la mosición mitad.
3. Si está hemos acabado. SI no, se comprueba si el elemento v[mitad] es mayor o menos que x.
    a. Si v[mitad]<x, actualizamos inf = mitad+1.
    b. Si v[mitad]>x, actualizamos sup = mitad-1.

4. Repetimos el proceso hasta encontrar el elemento o concluir que no está en el vector.

Cálculo eficiencia teórica

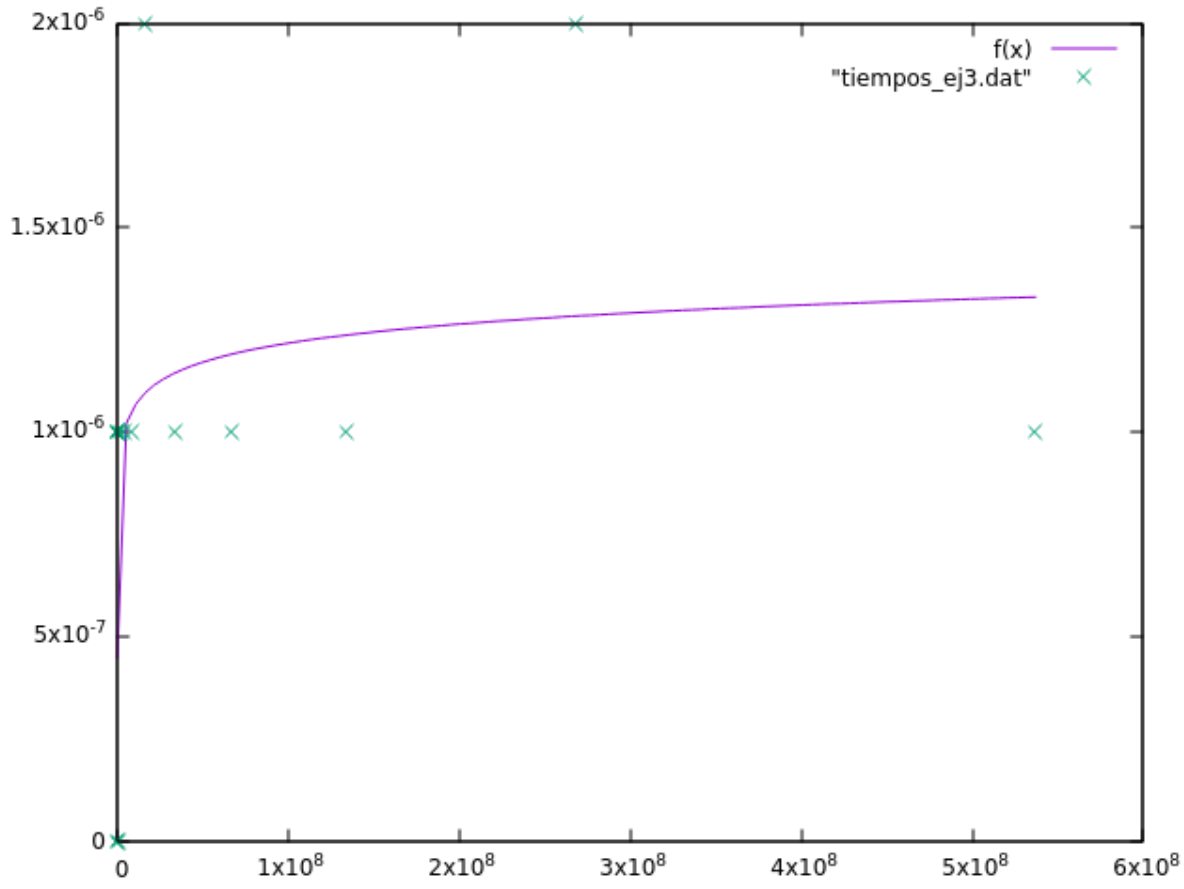
Para el cálculo de la eficiencia total de la búsqueda binaria, nos basamos en que se divide sucesivamente en dos un vector de tamaño n. En el peor caso, el proceso sigue hasta que no se puedan hacer más divisores del vector.

Como el resto de operaciones son elementales (O(1)), concluimos que la eficiencia es logarítmica.

Cálculo eficiencia empírica

Algunas ejecuciones tardan 0 segundos, suponemos que es porque el elemento se encontraba en mitad del vector.

Para calcular la eficiencia empírica se ha ejecutado el archivo un gran número de veces. Los datos los vemos reflejados en la siguiente gráfica:



Vemos que hay una gran diferencia entre la eficencia empírica y la teórica. La fórmula de la eficiencia teórica es $6.7172 * 10^{-08} log(x) - 2.01529^{-08}$.

Dificultad: 6.5/10

Ejercicios 4

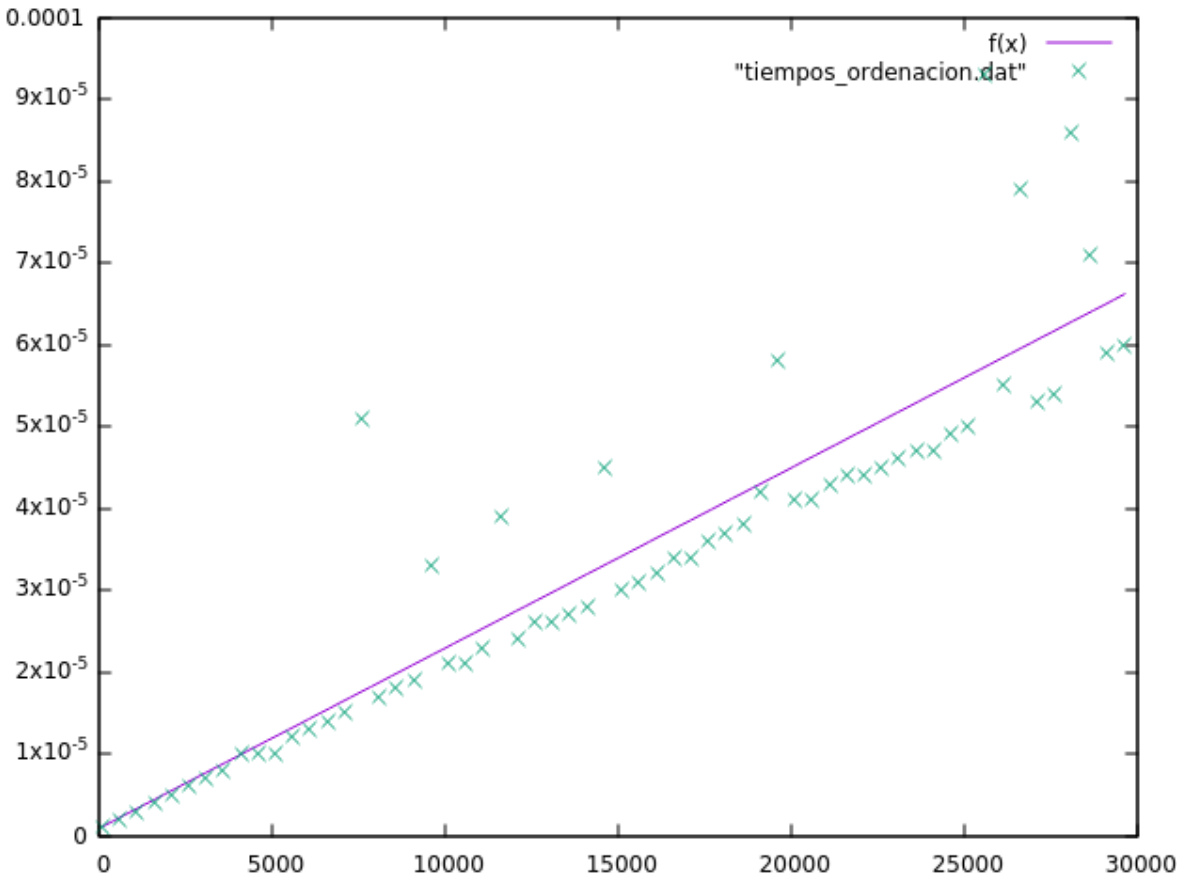
Utilizando el algoritmo:

```
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]);
    }
  }

}
```

El mejor caso es que el vector este ordenado,por lo que utilizamos el algoritmo:

```
for(int i = 0; i < tam; i++){
    //generar vector ordenado
    v[i]=i;
  }
```

Al analizar la eficiencia empírica del algoritmo, obtenemos la siguiente gráfica:



La eficiencia teórica $T(n) \in O(n)$, por lo que $T(n) = x+$

Dificultad: 6/10

Ejercicios 5

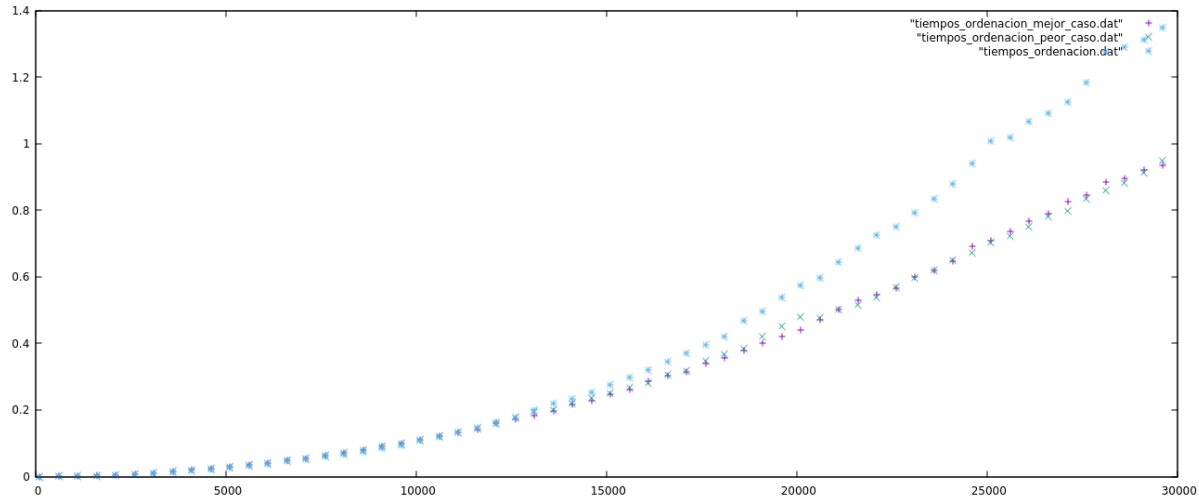
Para el mejor caso hemos hecho el siguiente algoritmo:

```
for(int i = 0; i < tam; i++){
    //generar vector ordenado
    v[i]=i;
  }
```

Para el peor caso hemos hecho el siguiente algoritmo:

```
for(int i = 0; i < tam; i++){
    //generar vector ordenado
    v[tam-i-1]=i;
  }
```

Comparando estas dos eficiencias empíricas y la eficiencia empirica del ejercicio 1, da la siguiente gráfica:



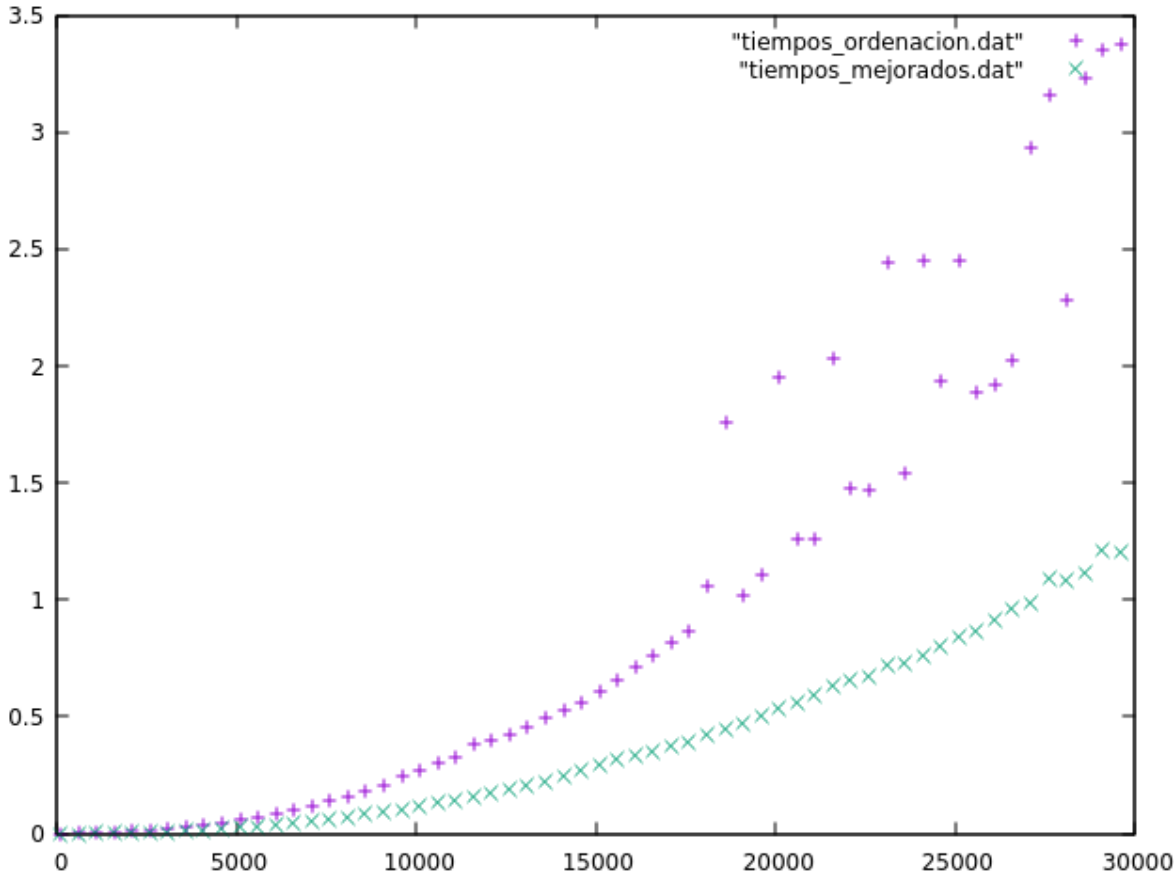
Vemos que la eficiencia empírica en el peor caso sigue siendo más eficiente que la eficiencia empírica del ejercicio 1.

Dificultad:4/10

Ejercicio 6

Hemos calculado la eficacia del programa ordenacion con dos distintas formas de compilación, para:

- tiempos_ordenacion.dat → g++ -g ordenacion.cpp -o ordenado.
- tiempos_mejorados.dat → g++ -03 ordenacion.cpp -o mejorado.



En la gráfica podemos apreciar que el programa compilado con -03 es más eficiente que si lo compilas con -g, además de que se parece más a la eficacia teórica.

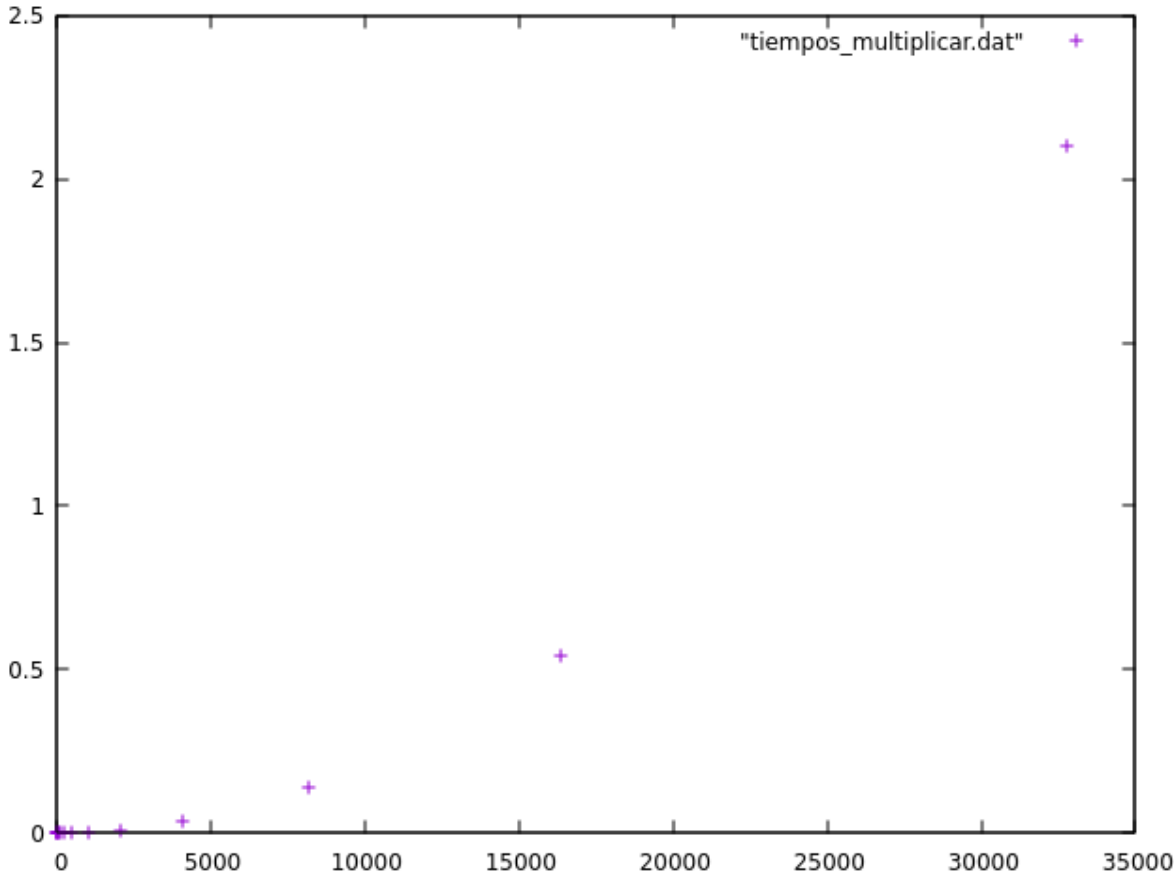
Dificultad: 4/10

Ejercicio 7

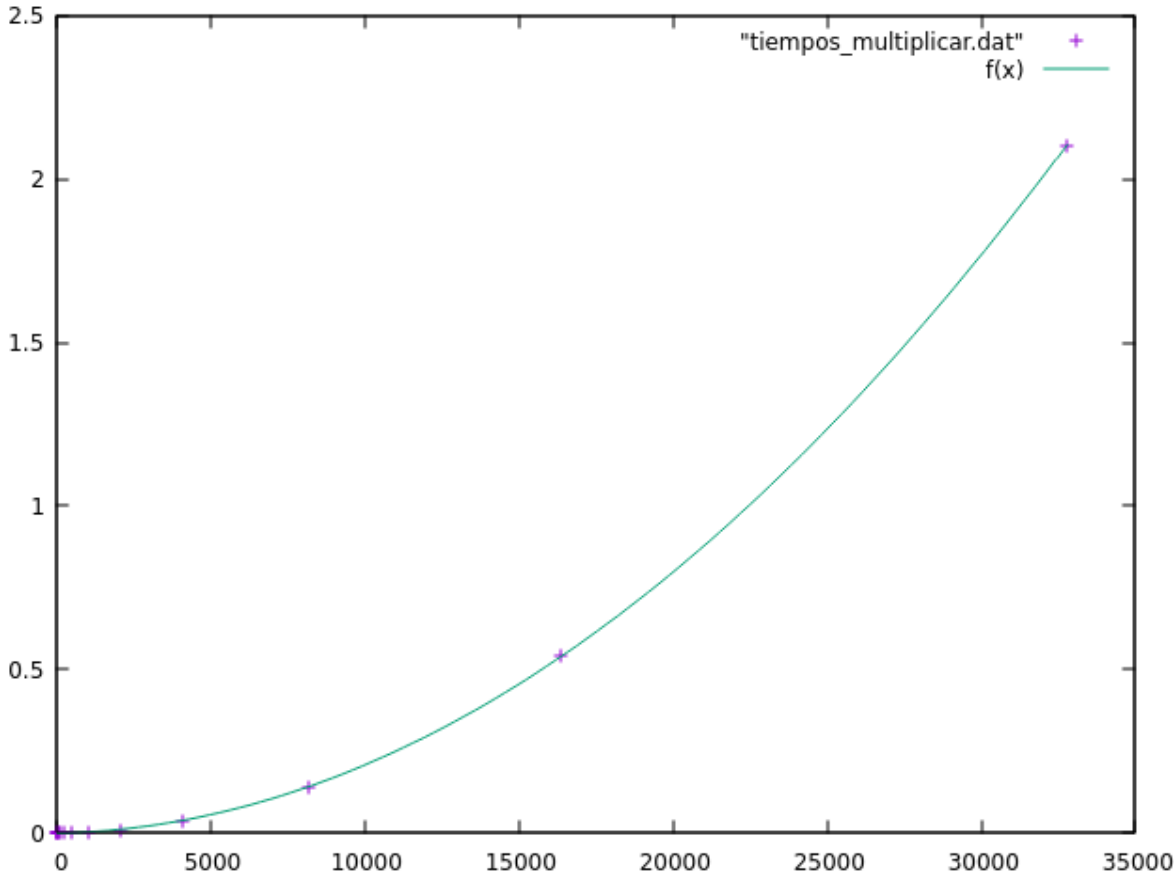
El algoritmo utilizado es;

```
void multiplicar(int **m, int **n,int t) {
  for(int i = 0; i < t; i++){
      for(int j = 0; j < t; j++){
        m[i][j]=m[i][j]*n[i][j];
      }
    }
}
```

Al analizar la eficiencia empírica del algoritmo, obtenemos la siguiente gráfica:



La eficiencia teórica es del orden $O(n^2)$, por lo que la función debe de ser de las formas f(n)=a*x*x+b; $f(n)=ax^2+bx$ o $f(n)=ax^2+bx+c$. En este caso, la que más se ajusta es la de $f(n)=ax^2+bx$, como se puede ver en la siguiente gráfica:

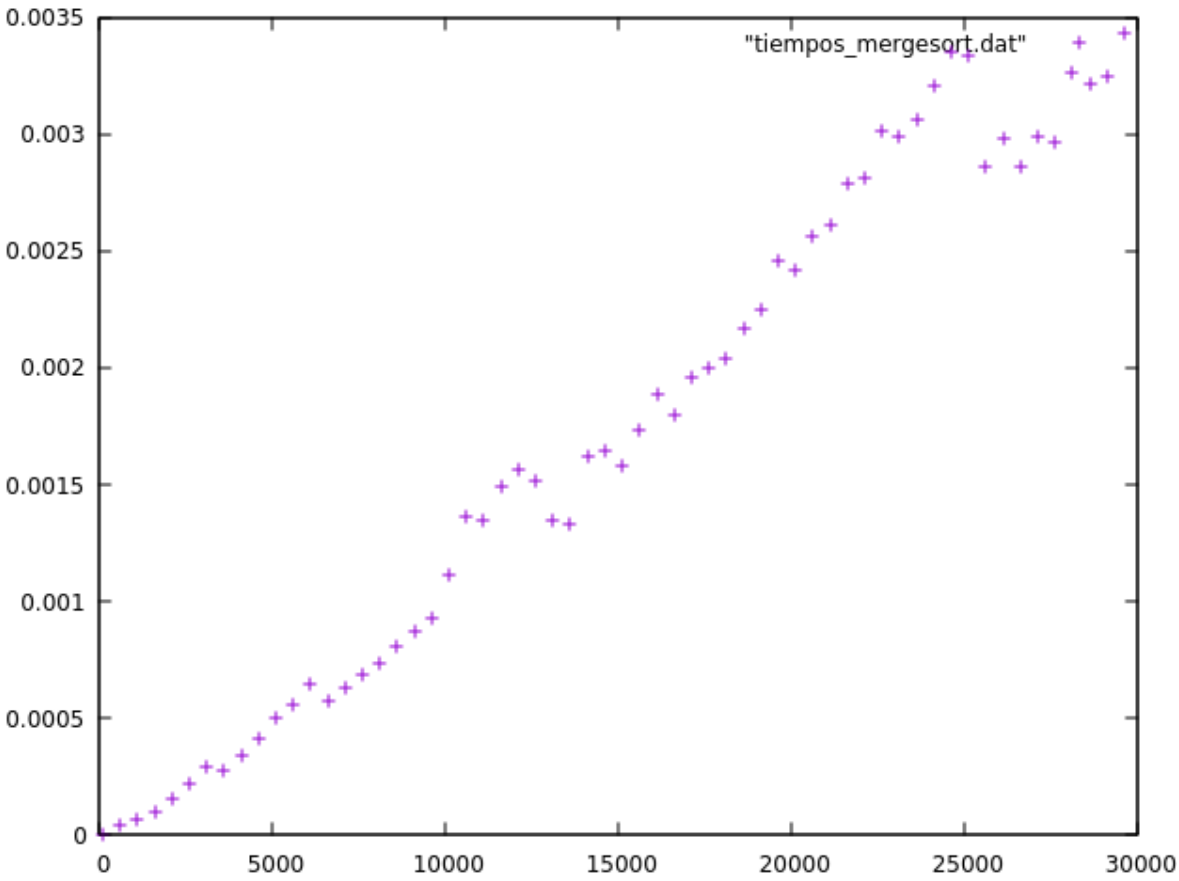


f(x)=ax²+b = 1.90123e-09x²+1.86385e-06x

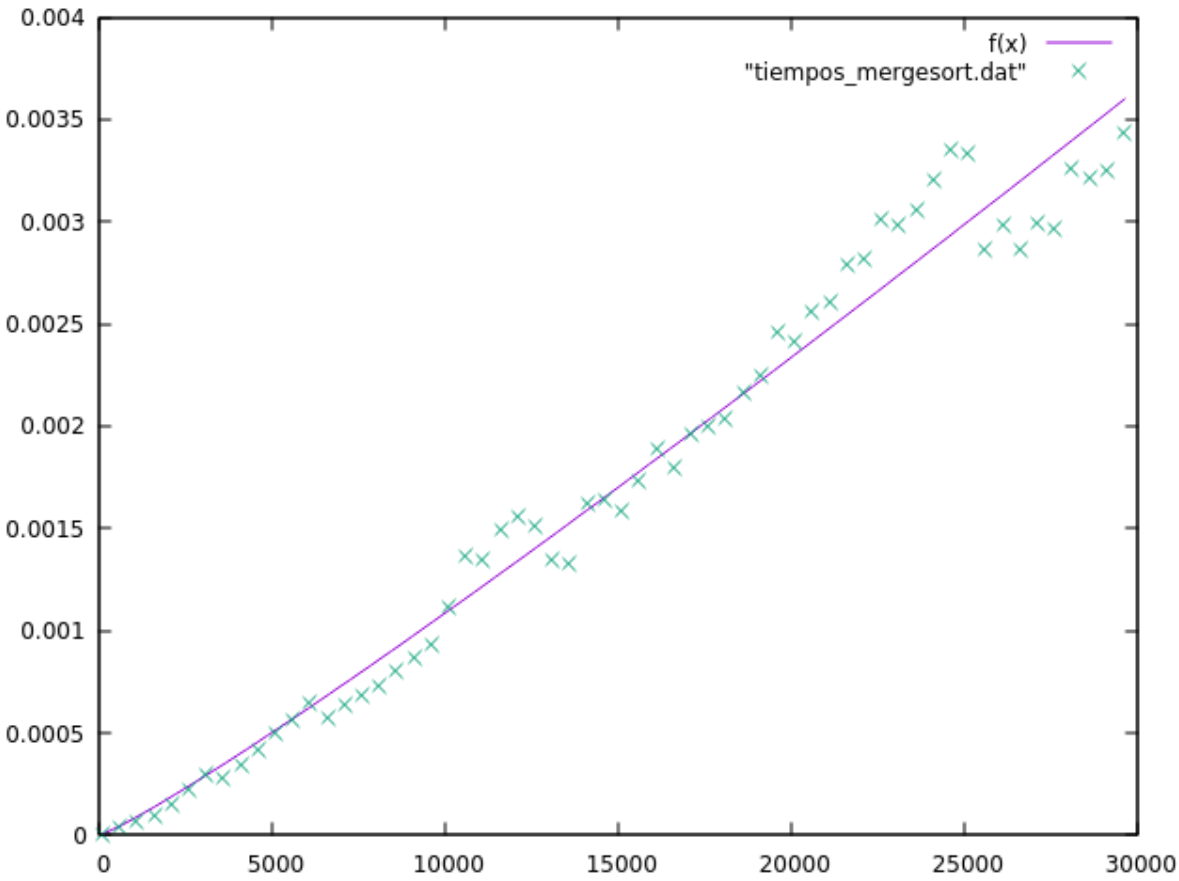
Dificultad:8/10

Ejercicio 8

La eficiencia empírica la hemos conseguido ejecutando el programa muchas veces, obteniendo la siguiente gráfica:

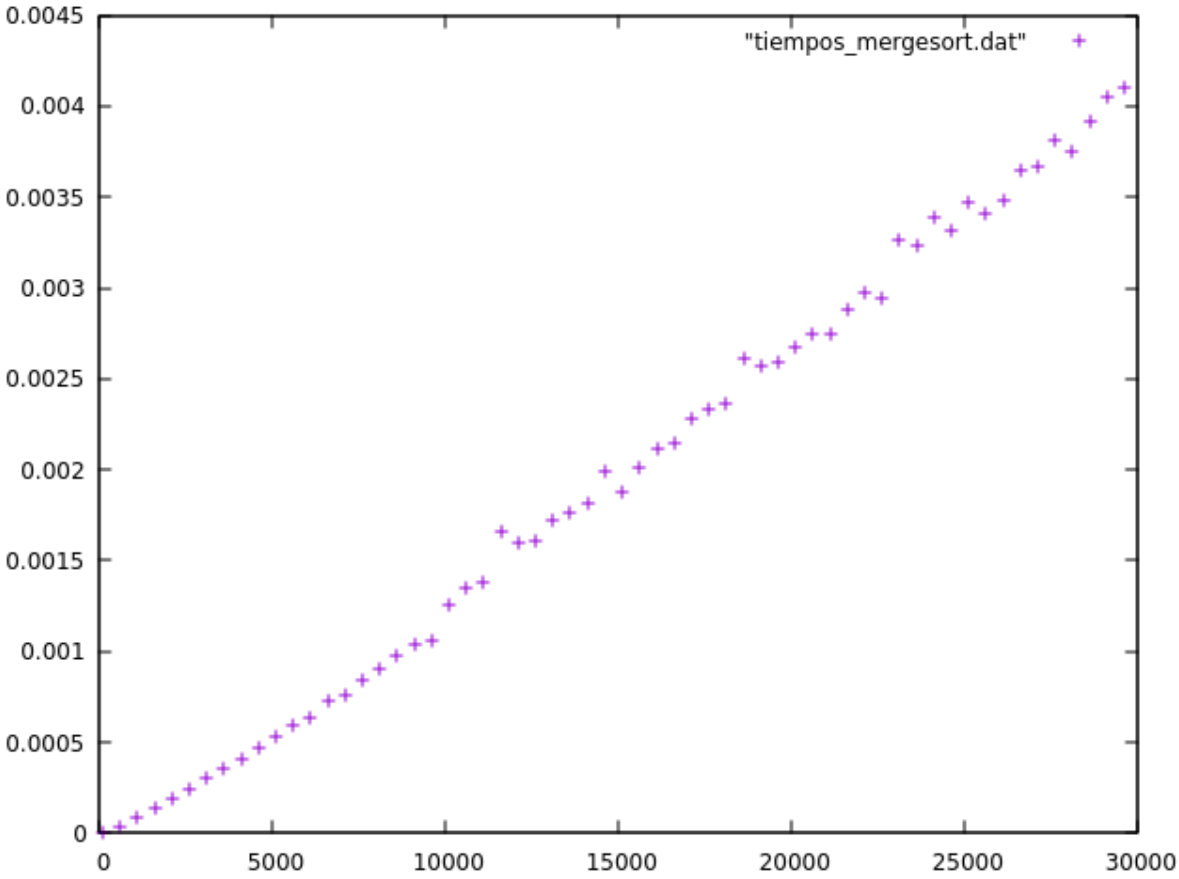


La eficiencia teórica es del orden O(nlog(n)). Ajustando las curvas, la función f(n) será f(n)=1.18069e-08n*log(n), la gráfica queda de la siguiente forma:

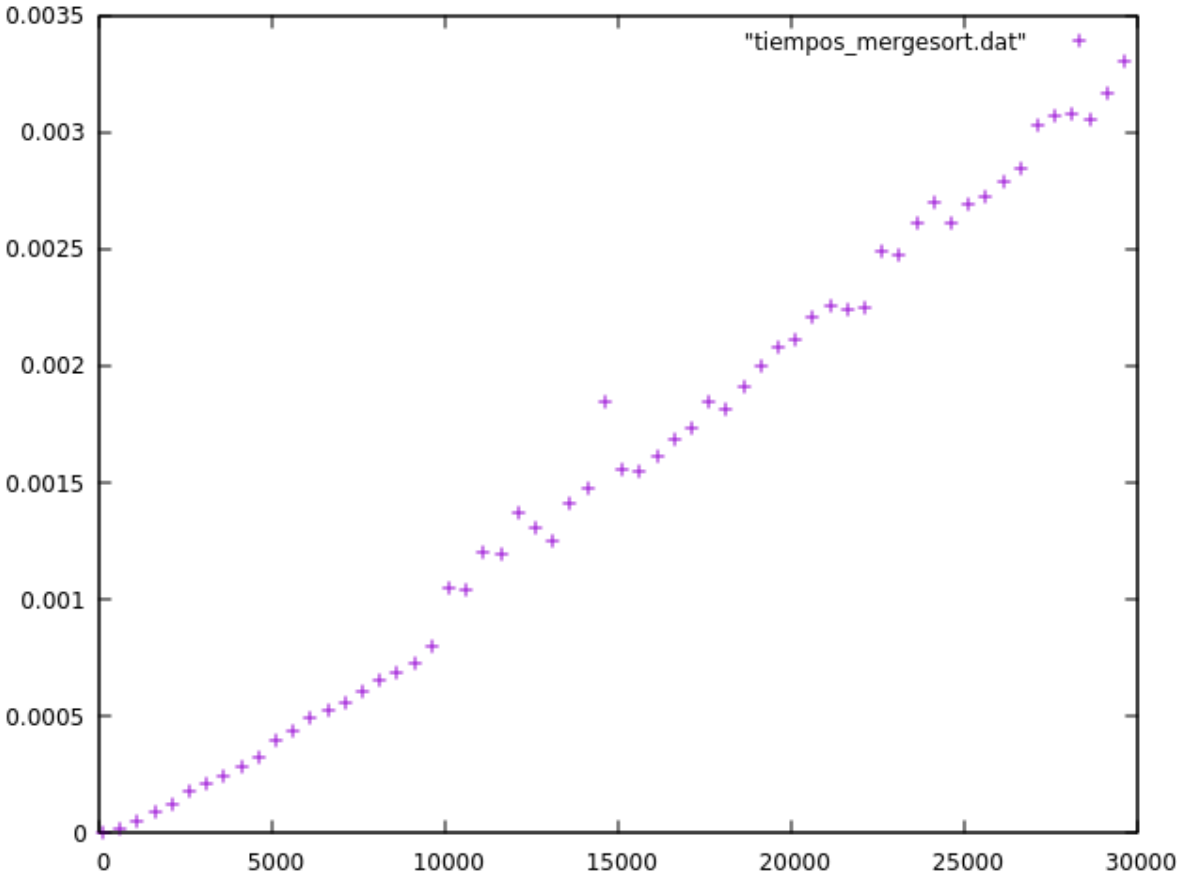


Variaciones de la variable UMBRAL_MS:

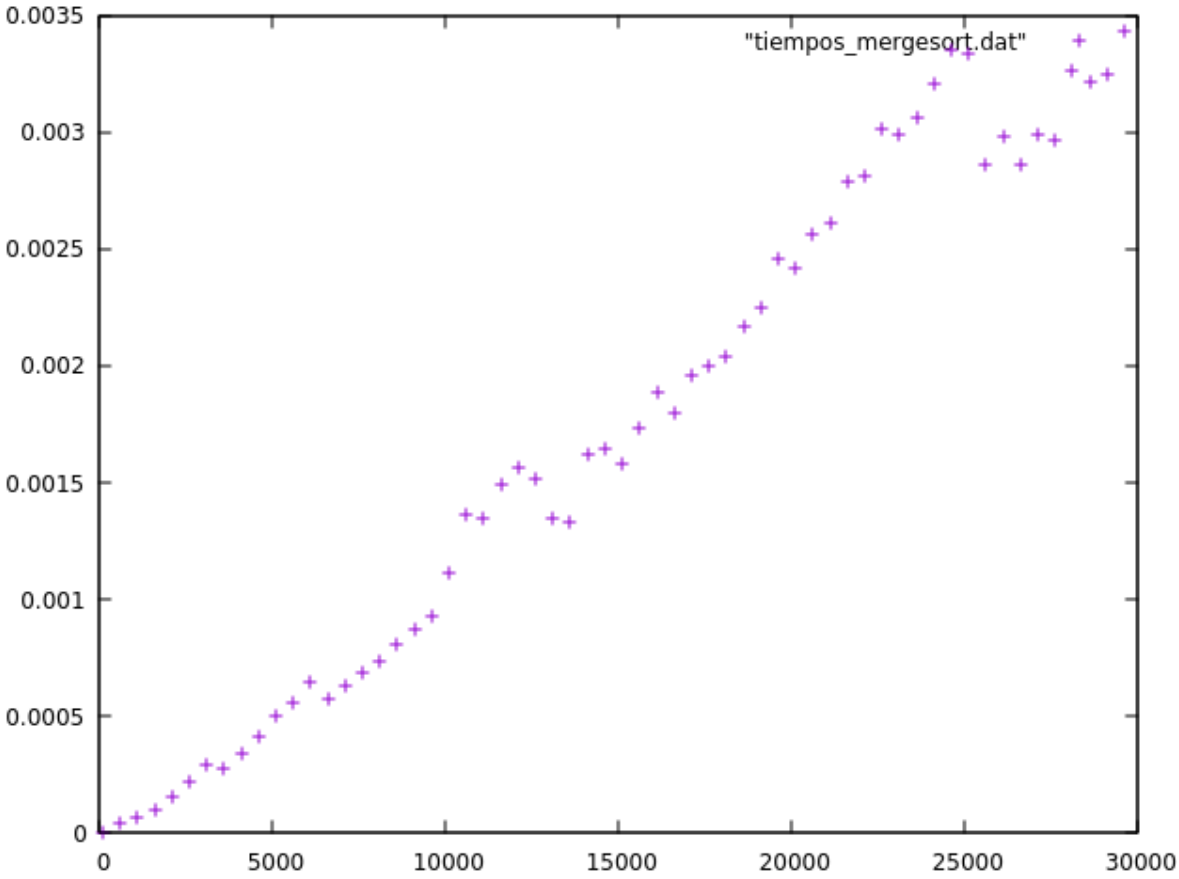
UMBRAL_MS=2


"tiempos_mergesort.dat"
0.0045
0.004
0.0035
0.003
0.0025
0.002
0.0015
0.001
0.0005
0
0
5000
10000
15000
20000
25000
30000

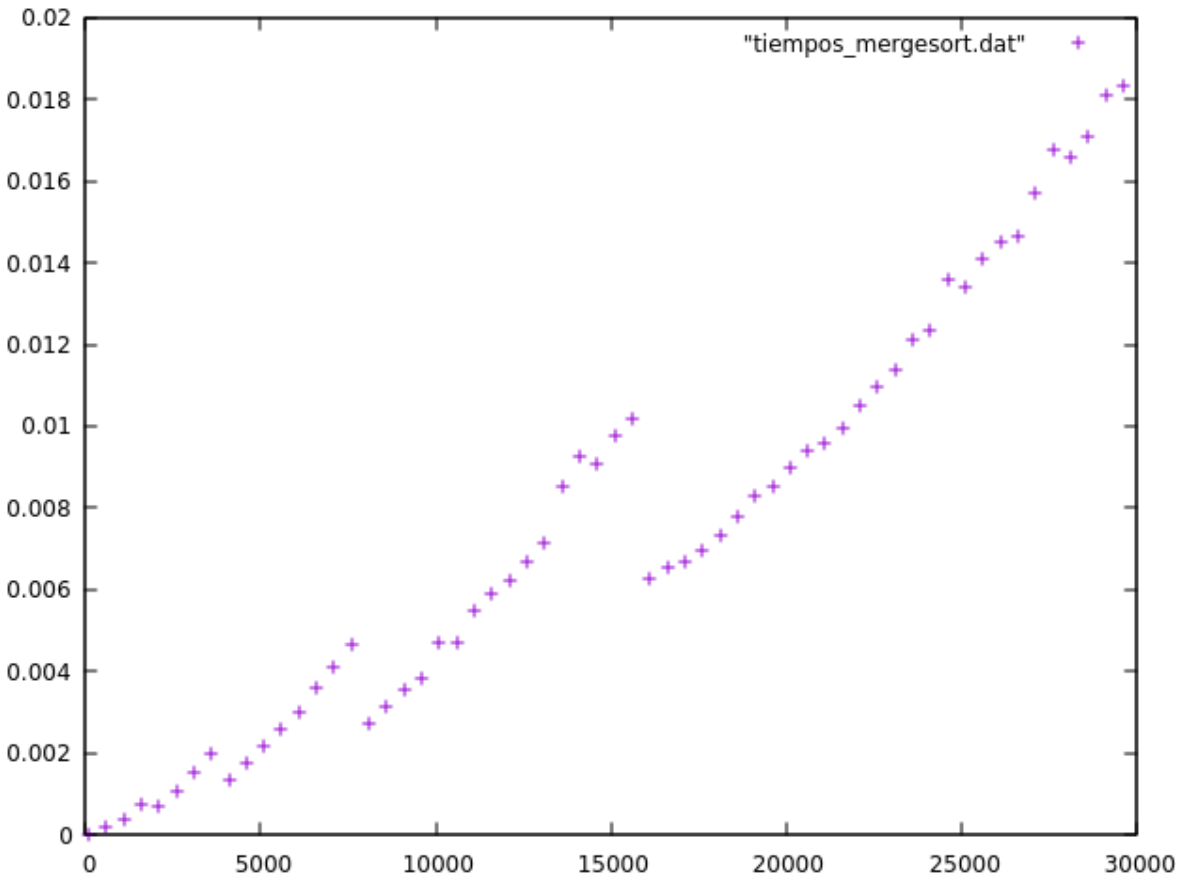
UMBRAL_MS=10



UMBRAL_MS=100



UMBRAL_MS=1000



Vemos que a medida que el valor de UMBRAL_MS va a umentando comienzan a existir más resultados/valores dispersos, en comparación con la eficiencia teórica.

Dificultad: 6/10