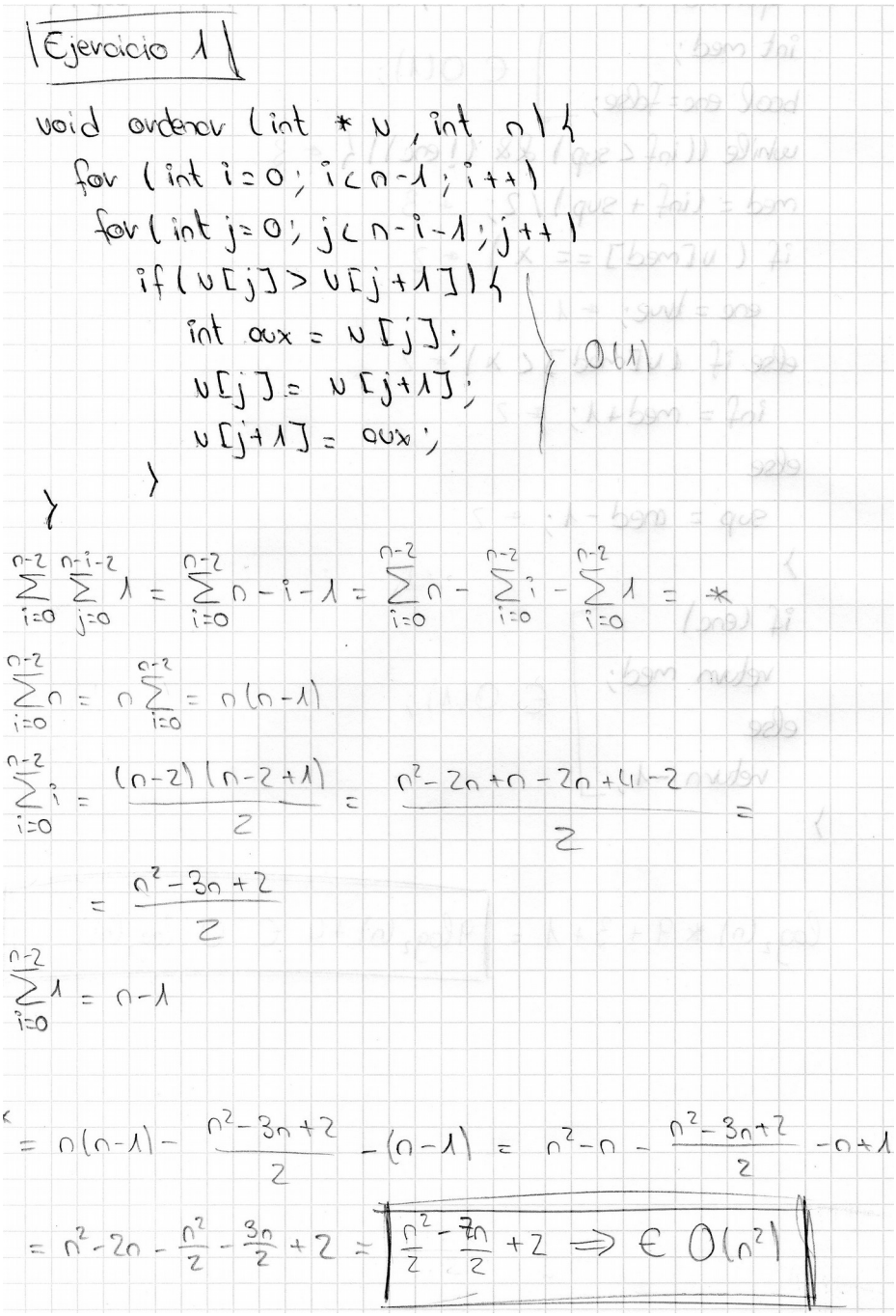
Estructura de Datos – Practica 1 (Eficiencia)

Alumno: Juan Carlos Ruiz Garcia
Grupo: C2
Caracteristicas:

- **SO:** Linux Mint 18 (64-bits)
- **CPU:** Intel(R) Core(TM) i5-4210U CPU @ 1.70GHz
- **RAM:** 8 GB 1600 MHz DDR3

Ejercicio 1 - *Ordenación de la burbuja*

Eficiencia Teórica



$$\sum_{i=0}^{n-2}\sum_{j=0}^{n-i-2} 1 = \sum_{i=0}^{n-2} n-i-1 = \sum_{i=0}^{n-2} n - \sum_{i=0}^{n-2} i - \sum_{i=0}^{n-2} 1 = *$$

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

$$\sum_{i=0}^{n-2} i = \frac{(n-2)(n-2+1)}{2} = \frac{n^2-2n+n-2n+4-2}{2} =$$

$$= \frac{n^2-3n+2}{2}$$

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

$$= n(n-1) - \frac{n^2-3n+2}{2} - (n-1) = n^2-n-\frac{n^2-3n+2}{2} - n+1$$

$$= n^2-2n-\frac{n^2}{2}-\frac{3n}{2}+2 = \frac{n^2}{2}-\frac{7n}{2}+2 \Rightarrow \in O(n^2)$$

Eficiencia Empírica

ordenacion.cpp

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

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

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

  int x = vmax+1; // Buscamos un valor que no está en el vector
  ordenar(v,tam); // de esta forma forzamos el peor caso

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

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

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

Compilamos el codigo fuente *ordenacion.cpp* y seguidamente creamos el siguiente script en *csh* para calcular la eficiencia empírica en X casos de la función de ordenación.

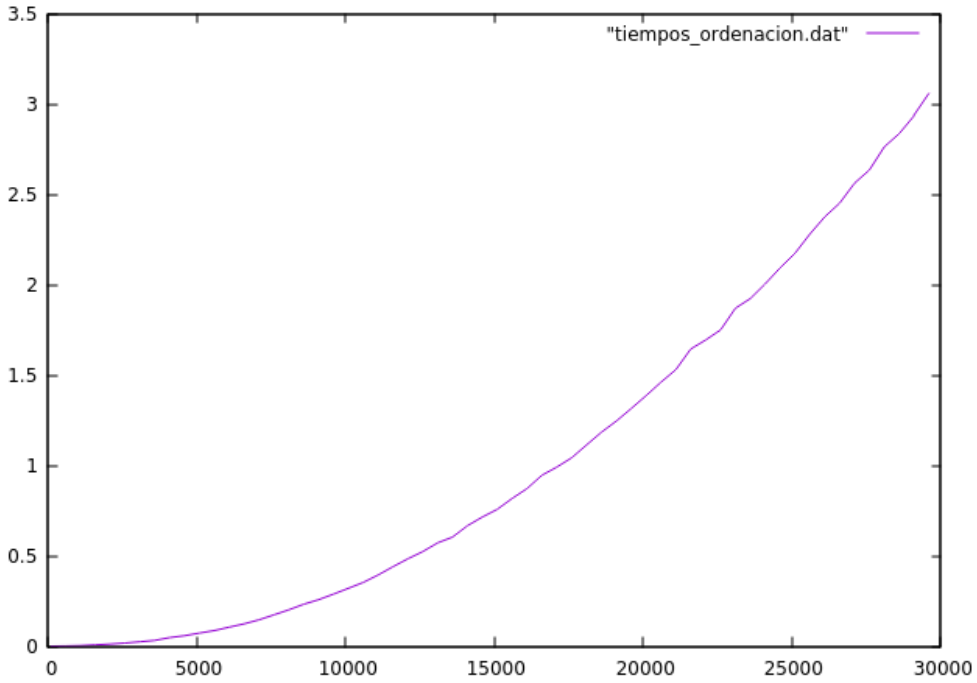
Ejecuciones_ordenacion.csh

```
#!/bin/csh
@ inicio = 100
@ fin = 30000
@ incremento = 500
@ i = $inicio
echo > tiempos_ordenacion.dat
while ( $i <= $fin )
echo Ejecución tam = $i
echo `./ordenacion $i 10000` >> tiempos_ordenacion.dat
@ i += $incremento
end
```

Lanzamos el *gnuplot* desde un terminal y ejecutamos las siguientes instrucciones:

```
plot "tiempos_ordenacion.dat" with lines
```

Esto nos genera la siguiente grafica:

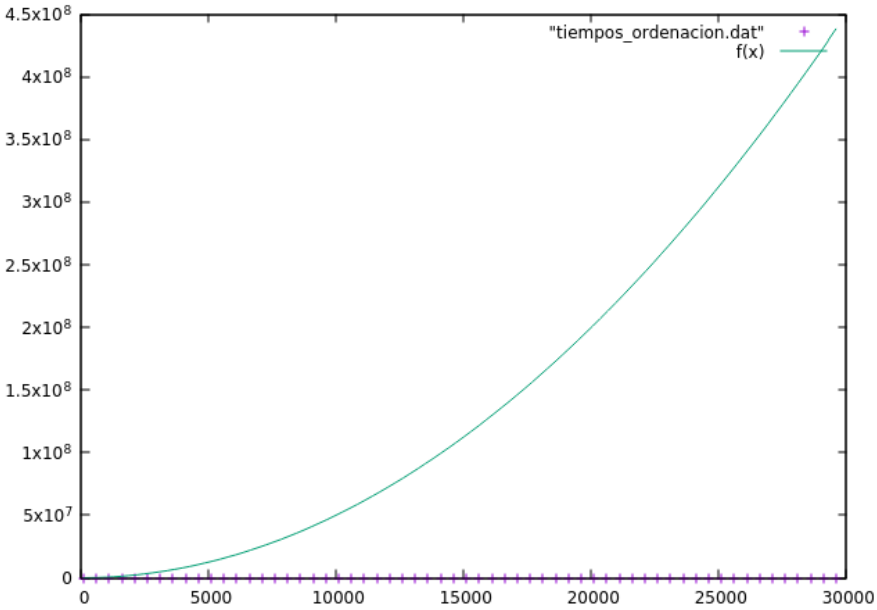


Seguidamente mostramos la eficiencia empírica y la teórica superpuestas en la misma grafica utilizando los siguientes comandos en *gnuplot*.

```
f(x) = (x*x/2) - 7*x/2 + 2
plot "tiempos_ordenacion.dat", f(x)
```

Esto nos muestra la siguiente grafica.

Lo que ocurre es que en



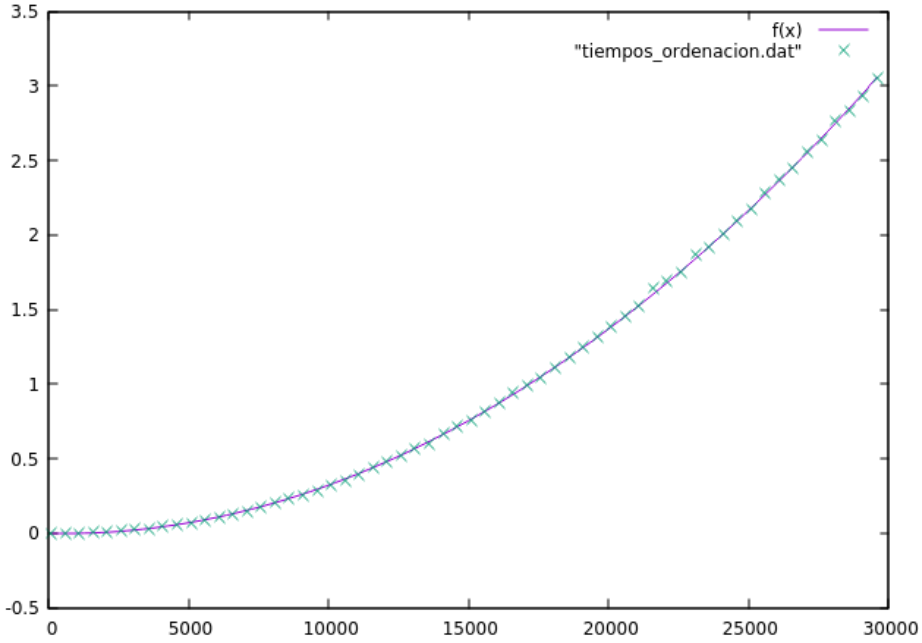
tiempos_ordenacion.dat la grafica se dispara hacia arriba debido a que no hemos ajustado *f(x)* a los datos. Esto se solucionará en el proximo ejercicio.

Ejercicio 2 - *Ajuste en la ordenación de la burbuja*

Lanzamos *gnuplot* desde una terminal y ejecutamos las siguientes instrucciones.

```
f(x) = a*x**2 + b*x + c
fit f(x) "tiempos_ordenacion.dat" via a,b,c
plot f(x), "tiempos_ordenacion.dat"
```

Con esto conseguimos ajustar *f(x)* a los datos obtenidos en la eficiencia empírica y con ellos la siguiente gráfica.



Ejercicio 3 - *Problemas de precisión*

El algoritmo es la *busqueda binaria*, asumiendo que el vector sobre el cual se ejecuta este algoritmo esta ordenado, se encarga de buscar un dato dentro de el y devolver su posicion.
La finalidad del *ejercicio_desc.cpp* es evaluar la busqueda binaria siempre en el peor de los casos. Ya que siempre busca un número que no existe dentro del vector, partiendo los indices *sup* e *inf* en dos y hacia la derecha.

Eficiciencia Teórica

Ejercicio 2

```
int operación (int * v, int n, int x, int inf, int sup){
    int med;
    bool enc=false;                          ] ∈ O(1);
    while ((inf < sup) && (!enc)){           ← 3
    med = (inf + sup) / 2;                   ← 3
    if ( v[med] == x )                       ← 2
      enc = true;                            ← 1
    else if (v[med] < x)                     ← 2
      inf = med+1;                           ← 2
    else
      sup = med - 1;                         ← 2
    }
    if (enc)
      return med;                            ] ∈ O(1);
    else
      return -1;
}
```

$\log_2(n) * 9 + 3 + 1 = 9\log_2(n) + 4 \in O(\log_2(n))$

Eficiencia Empírica

Al calcular su eficiencia empírica nos hemos dado cuenta de que el algoritmo se ejecuta de forma tan rapida que es imposible apreciar las diferencias de tiempo.

```
50 1e-06
150 2e-06
250 1e-06
350 1e-06
450 3e-06
550 1e-06
650 1e-06
750 1e-06
850 1e-06
950 1e-06
```

Para arreglar esto y poder ver unas diferencias de tiempo que se puedan evaluar, vamos meter en el codigo de *ejercicio_desc.cpp* un retardo. Este retardo repitiar la misma busqueda durante 3000 veces lo que pausará el proceso lo suficiente para poder evaluar el tiempo de ejecución.

```
for (int i = 0; i < 3000; i++) {
  // Algoritmo a evaluar
  operacion(v,tam,tam+1,0,tam-1);
}
```

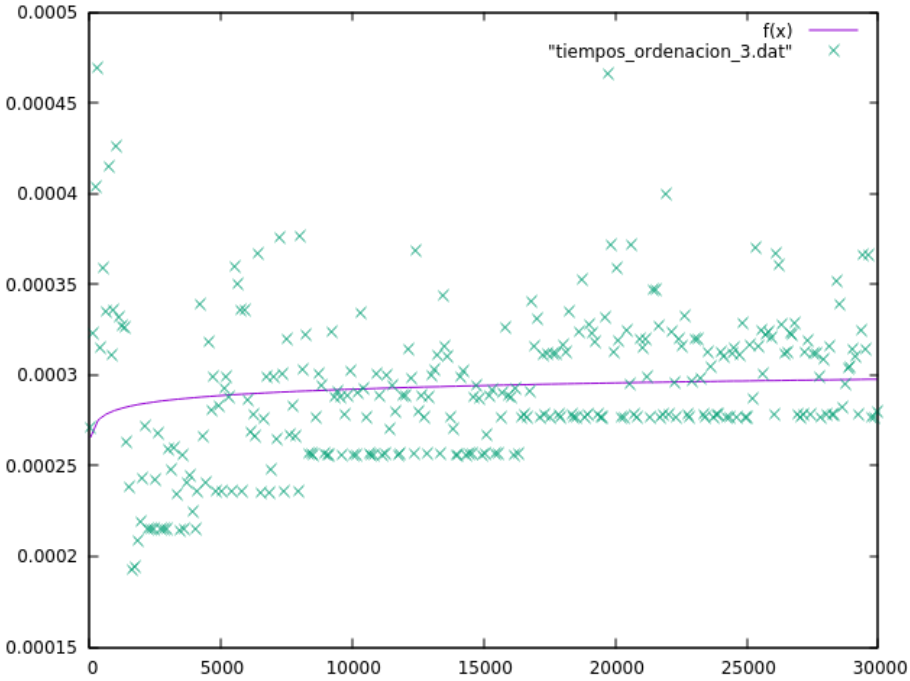
Obtendriamos unos tiempos mas razonables para analizar.

```
50 0.000271
150 0.000323
250 0.000404
350 0.000469
450 0.000315
550 0.000359
650 0.000335
750 0.000415
850 0.000311
950 0.000336
```

Ahora realizaremos la regresión para ajustar la curva teórica a la empírica. Para ello utilizaremos las siguientes instrucciones desde terminal.

```
f(x) = a*log(x)/log(2) + b
fit f(x) "tiempos_ordenacion_3.dat" via a,b
plot f(x), "tiempos_ordenacion_3.dat"
```

Y obtenemos la siguiente gráfica.



Ejercicio 4 - *Mejor y peor caso*

Calculamos la eficiencia empirica del peor de los casos y del mejor.

Mejor caso:

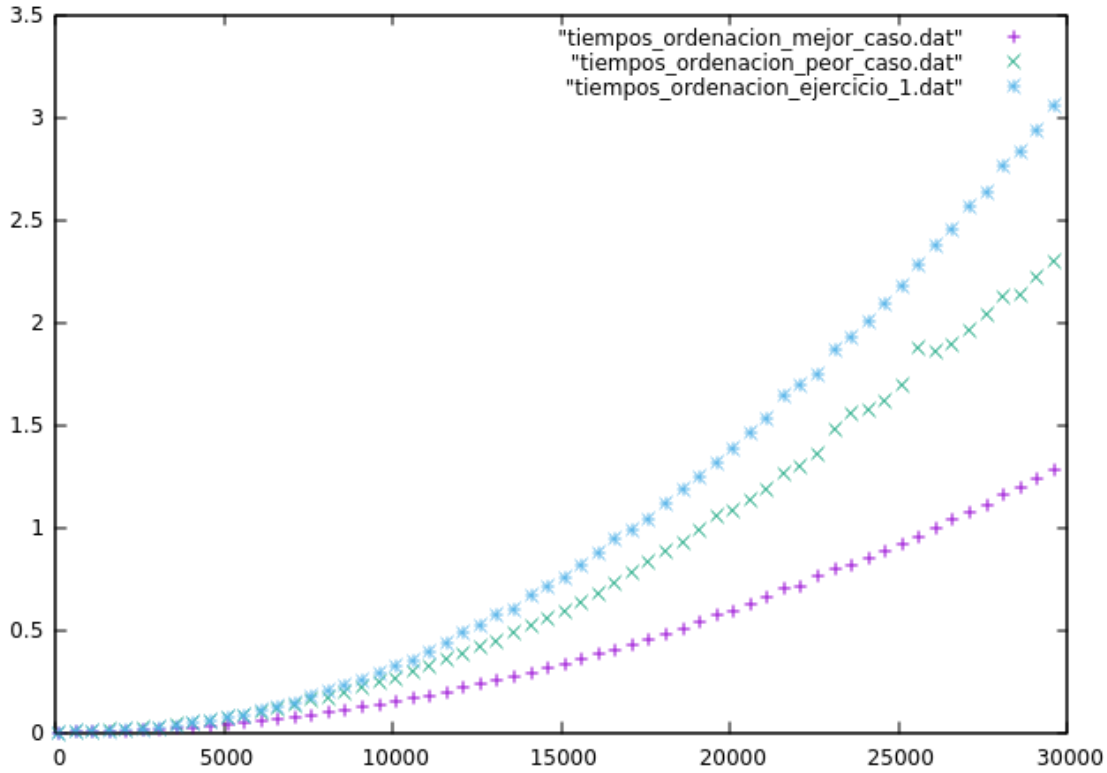
```
for (int i=0; i<tam; i++) // Recorrer vector
  v[i] = i; // Añade elementos al vector desde 0 a tam-1
```

Peor caso:

```
for (int i=tam-1; i>=0; i--) // Recorrer vector
  v[tam-i-1] = i; // Añade elementos al vector desde 0 a tam-1
```

Generamos la grafica de ambos casos superpuestas, y además le añadiremos el caso aleatorio del ejercicio 1. Para ello utlizamos los siguientes comandos en *gnuplot*.

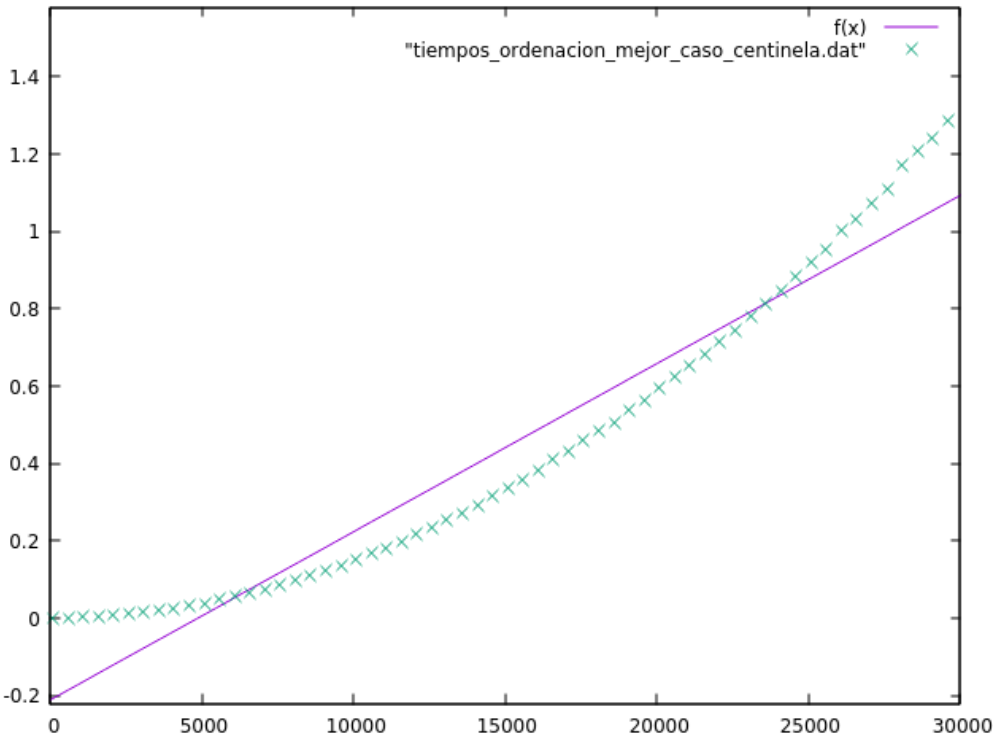
plot "tiempos_ordenacion_mejor_caso.dat", "tiempos_ordenacion_peor_caso.dat", "tiempos_ordenacion_ejercicio_1.dat"



Ejercicio 5 - *Dependencia de la implementación*

En el mejor de los casos la *eficicencia teórica* pasaria de una $O(n^2)$ a O(n) ya que solo se operaría una vez sobre los datos.

Tras calcular la *eficiencia empírica* de ambos escenarios, calculamos la siguiente gráfica.



Para ello hemos utilizado las siguiente instrucciones en *gnuplot*.

```
f(x) = a*x + b
fit f(x) "tiempos_ordenacion_mejor_caso_centinela.dat" via a,b
plot f(x), "tiempos_ordenacion_mejor_caso_centinela.dat"
```

Tras esto podemos darnos cuenta de que la *eficiencia empírica* del ejercicio 5 tiene una progresión parecida a la progresión lineal de O(n).

Ejercicio 6 - *Influencia del proceso de compilación*

Compilando el programa de la forma dicha en el enunciado del ejercicio 6 podemos comprobar que las curvas de la gráfica son distintas y que la eficiencia mejora. Para ello usamos la siguiente instrucción en *gnuplot*.

```
plot "tiempos_ordenacion.dat", "tiempos_ordenacion_optimizado.dat"
```