
UNIVERSIDAD
DE GRANADA

# Estudio de la Eficiencia

**Estructura de Datos**

**José Miguel Aguado Coca**
**Alejandro Martín Casas**

# ESPECIFICACIONES EN COMÚN PARA TODOS LOS EJERCICIOS

**HARDWARE:**

- 13th Gen Intel(R) Core(TM) i7-1355U
- 16 gb ram

**SO:**

- Linux Ubuntu

**COMPILADOR:**

-g++ fichero.cpp -o ejecutable

# EJERCICIO 1

### 1) Cálculo de la eficiencia teórica

```
void ordenar(int *v, int n) {                                   //Total: O(n²)

        for (int i=0; i<n-1; i++){                              //O(n)

                for (int j=0; j<n-i-1; j++){                    //O(n)

                        if (v[j]>v[j+1]) {              //O(1)

                                int aux = v[j];         //O(1)
                                v[j] = v[j+1];          //O(1)
                                v[j+1] = aux;           //O(1)

                        }
                }
        }
}
```

- Eficiencia Teórica : O(n) * O(n) = O(n²)

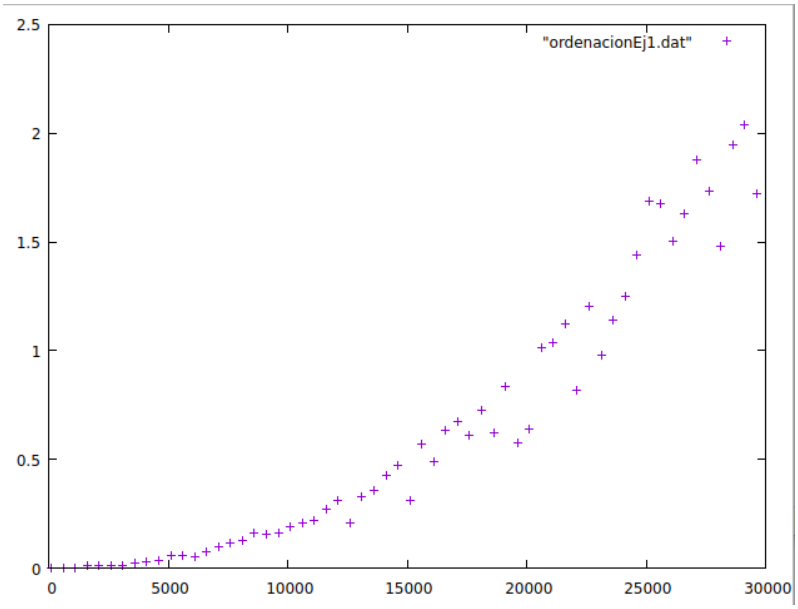
### 2) Ordenacion.cpp

```cpp
#include <iostream>
#include <cstdlib>  // Para rand() y srand()
#include <ctime>    // Para time()
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;
                        }
                }
        }
}

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

        srand(time(NULL));
        int n = atoi(argv[1]);
        int *v = new int[n];

        for (int i = 0; i < n; i++) {
                v[i] = rand() % 1000;
        }

        //INICIO MEDIDA TIEMPO

        clock_t tini;
        tini=clock();

        ordenar(v, n);

        //FIN MEDIDA TIEMPO

        clock_t tfin;
        tfin = clock();

        cout << n << "\t" << (tfin-tini)/(double)CLOCKS_PER_SEC << endl;
        delete[] v;

}
```

**3) ejecuciones_ordenacion.csh**

```
#!/bin/csh
@ inicio = 100
@ fin = 30000
@ incremento = 500

set ejecutable = ordenacion
set salida = ordenacionEj1.dat

@ i = $inicio
echo > $salida
while ( $i <= $fin )
  echo Ejecución tam = $i
  echo `./{$ejecutable} $i 10000` >> $salida
  @ i += $incremento
end
```

**4) Dibujo con gnuplot**




**4) ¿Qué sucede al dibujar la eficiencia empírica y la función teórica?**

Solo se dibuja la función $x^2$, mientras que la de los datos no. Esto se debe a los valores pequeños que toman los tiempos en el eje Y.

# EJERCICIO 2

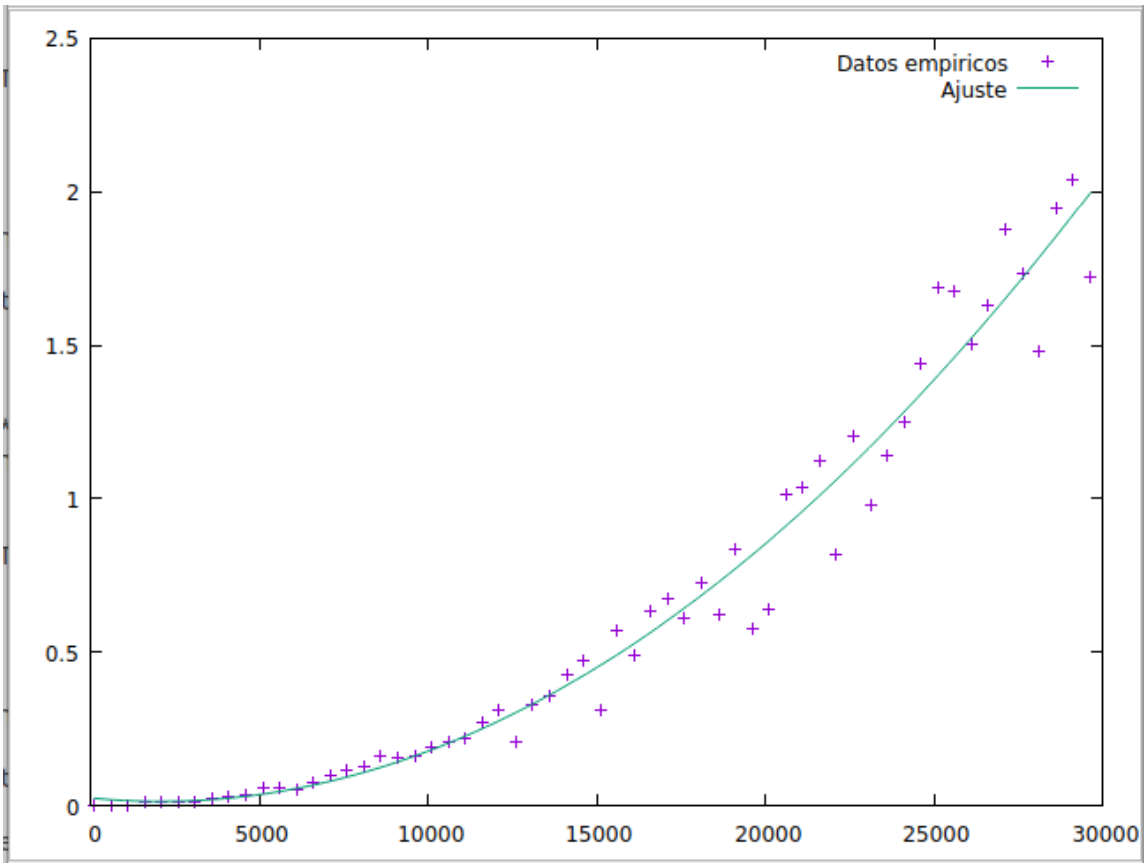
### 1) Ajuste por regresión algoritmo de la burbuja

```
Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a               = 2.61571e-09      +/- 2.138e-10    (8.176%)
b               = -1.0963e-05      +/- 6.564e-06    (59.87%)
c               = 0.0271609        +/- 0.04217      (155.3%)

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

gnuplot> f(x) = a * x**2 + b * x + c

gnuplot> fit f(x) 'ordenacionEj1.dat' using 1:2 via a,b,c

gnuplot> plot 'ordenacionEj1.dat' using 1:2 with points title 'Datos empiricos', f(x) title 'Ajuste'



# EJERCICIO 3

**1) ¿Qué hace este algoritmo?**

- Este algoritmo realiza una búsqueda binaria sobre un vector dinámico, dónde su tamaño se le pasa como argumento, y se rellena con números aleatorios. Se busca el valor tam+1.

**2) Cálculo de la eficiencia teórica**

- En el main, todo será O(1) menos el bucle for que recorre el vector y la llamada a operación.

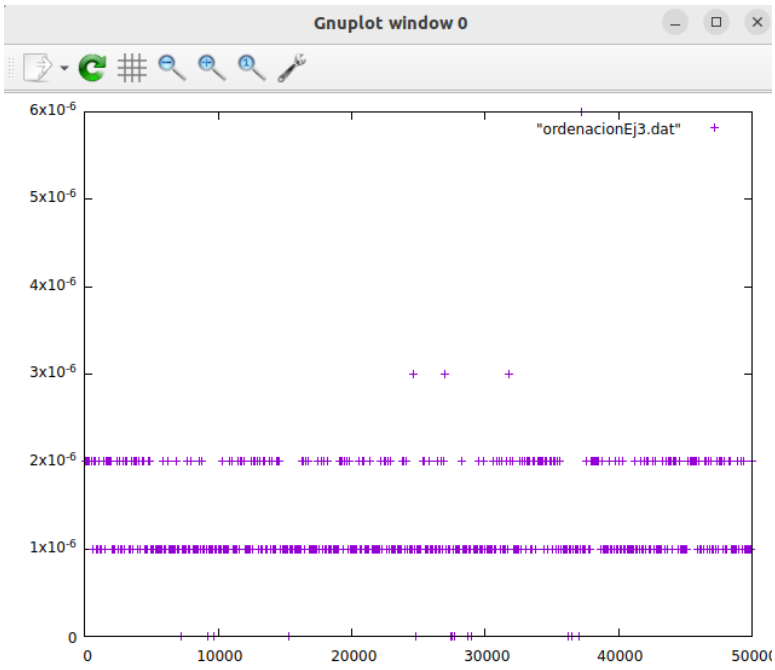
```
for (int i=0; i<tam; i++)       //O(n)
  v[i] = rand() % tam;          //O(1)

operacion(v,tam,tam+1,0,tam-1);
```

- Analizamos esta función para ver su eficiencia.

```
int operacion(int *v, int n, int x, int inf, int sup) {

        int med;        //O(1)
        bool enc=false;         //O(1)

        while ((inf<sup) && (!enc)) {       //En el peor de los casos, este bucle se realiza log n
                                            //O(log n)

                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;

}
```

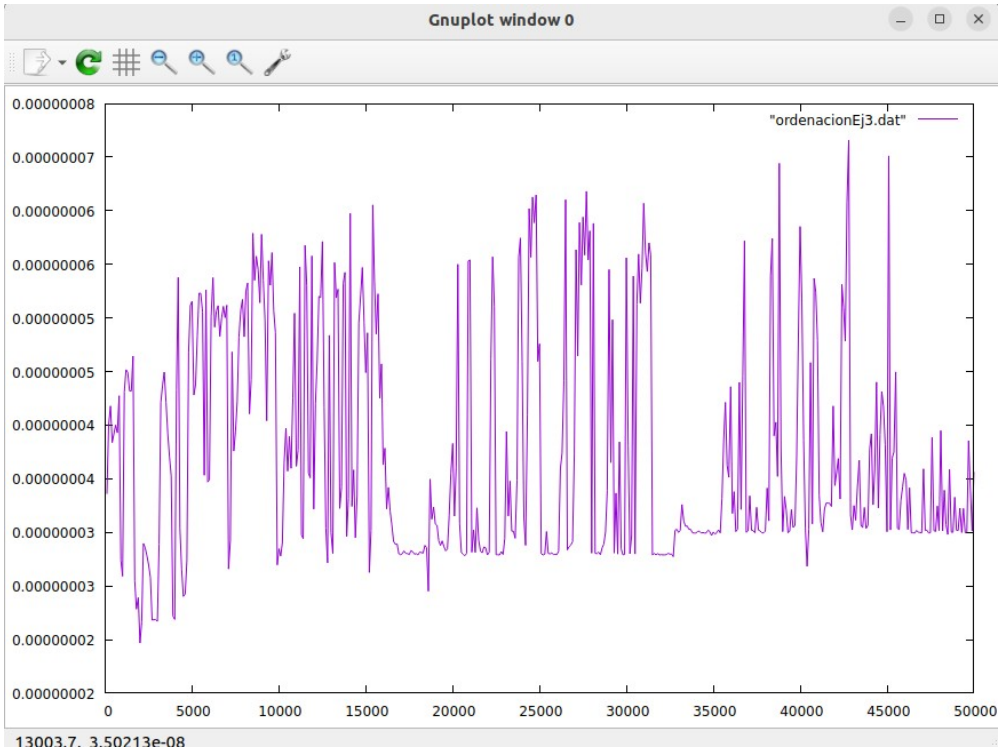
**3) Cálculo de la eficiencia empírica**

```
#!/bin/csh
@ inicio = 100
@ fin = 50000
@ incremento = 100

set ejecutable = ejercicio_desc
set salida = ordenacionEj3.dat

@ i = $inicio
echo > $salida
while ( $i <= $fin )
  echo Ejecución tam = $i
  echo `./{$ejecutable} $i` >> $salida
  @ i += $incremento
end
```



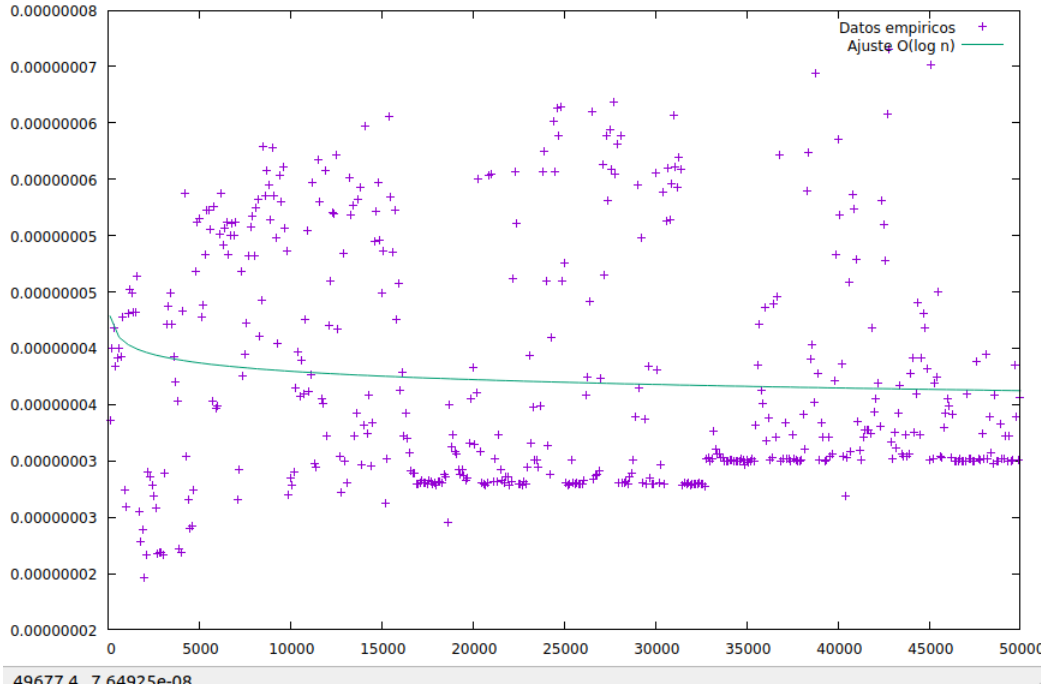

- Al visualizar la eficiencia empírica vemos algo anormal. El algoritmo tarda demasiado poco, por lo que no se puede medir bien. Para solucionarlo, he usado chrono con high resolution, y he ejecutado el algoritmo muchas veces, y lo he dividido el tiempo total en ese número de veces, para sacar el promedio. Esto con un bucle for.

```
const int NUM_REPETICIONES = 100000;

// Usamos std::chrono para medir el tiempo de ejecución
auto tini = chrono::high_resolution_clock::now();

// Algoritmo a evaluar (repetido varias veces)
for (int i = 0; i < NUM_REPETICIONES; i++) {
  operacion(v, tam, tam + 1, 0, tam - 1);
}

auto tfin = chrono::high_resolution_clock::now();

// Calculamos la duración total y mostramos el tiempo promedio por operación
chrono::duration<double> duracion = tfin - tini;
cout << tam << "\t" << (duracion.count() / NUM_REPETICIONES) << endl;
```




## 4) Regresión

```
*******************************************************************************
Sun Sep 29 11:47:00 2024


FIT:    data read from 'ordenacionEj3.dat' using 1:2
        format = x:z
        #datapoints = 500
        residuals are weighted equally (unit weight)

function used for fitting: f(x)
        f(x) = a * log(x) + b
fitted parameters initialized with current variable values

iter      chisq       delta/lim  lambda   a             b
   0 5.9096244415e+04   0.00e+00  7.02e+00    1.000000e+00   1.000000e+00
   6 5.3677061560e-14   0.00e+00  7.02e-06   -1.070357e-09   5.281119e-08

After 6 iterations the fit converged.
final sum of squares of residuals : 5.36771e-14
rel. change during last iteration : 0

degrees of freedom    (FIT_NDF)                      : 498
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 1.0382e-08
variance of residuals (reduced chisquare) = WSSR/ndf   : 1.07785e-16

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a               = -1.07036e-09     +/- 4.762e-10    (44.49%)
b               = 5.28112e-08      +/- 4.703e-09    (8.906%)

correlation matrix of the fit parameters:
                a      b
a               1.000
b              -0.995  1.000
```



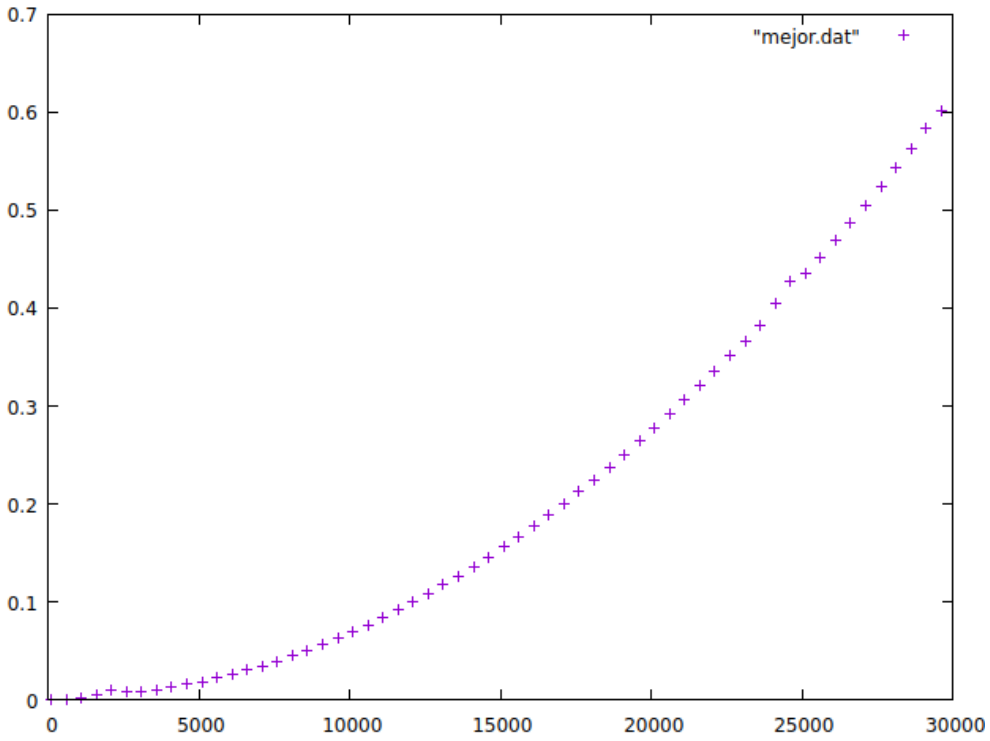

No funciona del todo bien.

# EJERCICIO 4

**1) Mejor caso (vector ordenado)**

```
// Mejor caso: vector ordenado
for (int i = 0; i < n; i++) {
    v[i] = i;
}
```

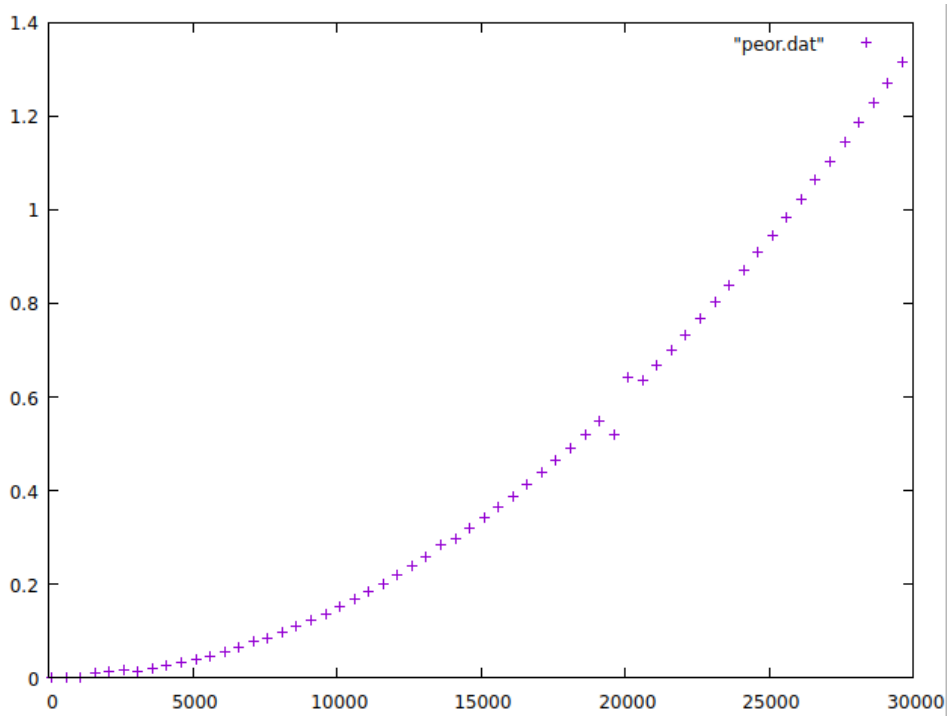
```
#!/bin/csh
@ inicio = 100
@ fin = 30000
@ incremento = 500

set ejecutable = mejor
set salida = mejor.dat

@ i = $inicio
echo > $salida
while ( $i <= $fin )
  echo Ejecución tam = $i
  echo `./{$ejecutable} $i` >> $salida
  @ i += $incremento
end
```



**2) Peor caso (vector inverso)**

```
// Peor caso: vector ordenado en orden inverso
for (int i = 0; i < n; i++) {
    v[i] = n - i;
}
```

- El fichero csh es igual que en el mejor caso




**3) Comparación de tiempos**

- Con 30000, el mejor caso tarda 0.6, mientras que el peor tarda 1.4. Tarda el doble. En el caso de aleatorios tarda algo más, 2 segundos, pero porque es aleatorio.

# EJERCICIO 5

### 1) Eficiencia teórica en el mejor de los casos

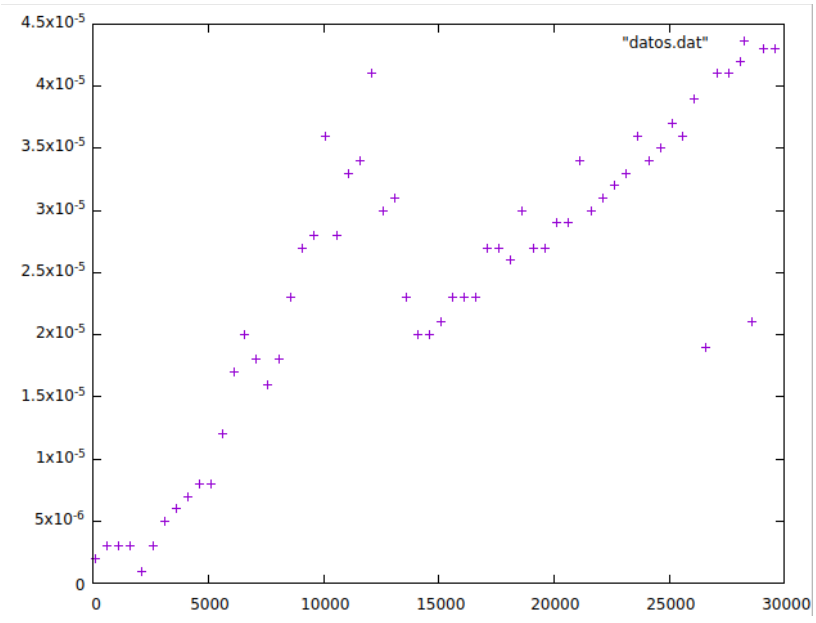
- En el peor de los casos, la eficiencia será de $O(n^2)$ al ser un bucle for anidado. Vamos a ver su eficiencia en el mejor de los casos (vector ordenado al principio).

```
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;
                        }
                }
        }
}
```

- En la iteración 0 del primer bucle, se recorre el vector con el bucle interno n-1 veces. Como el vector ya está ordenado, solo n-1. Por lo tanto, eficiencia O(n).

### 2) Eficiencia empírica

- Se ha usado un csh como en ejemplos anteriores.



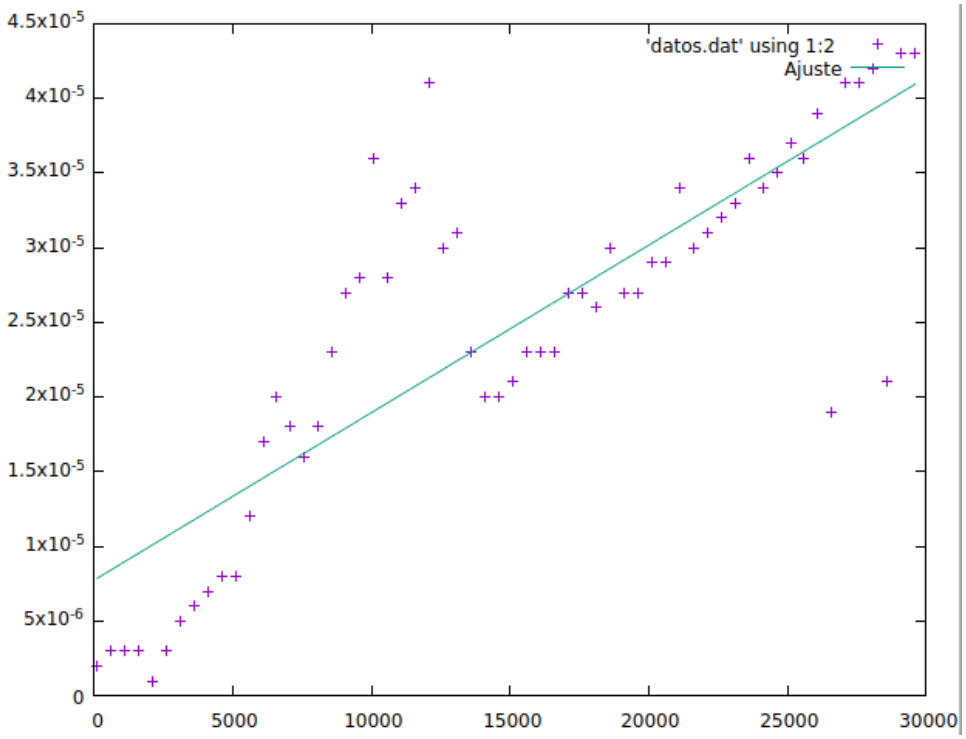
**3) Ajuste curva**

gnuplot> f(x) = a*x + b

gnuplot> fit f(x) 'datos.dat' using 1:2 via a, b

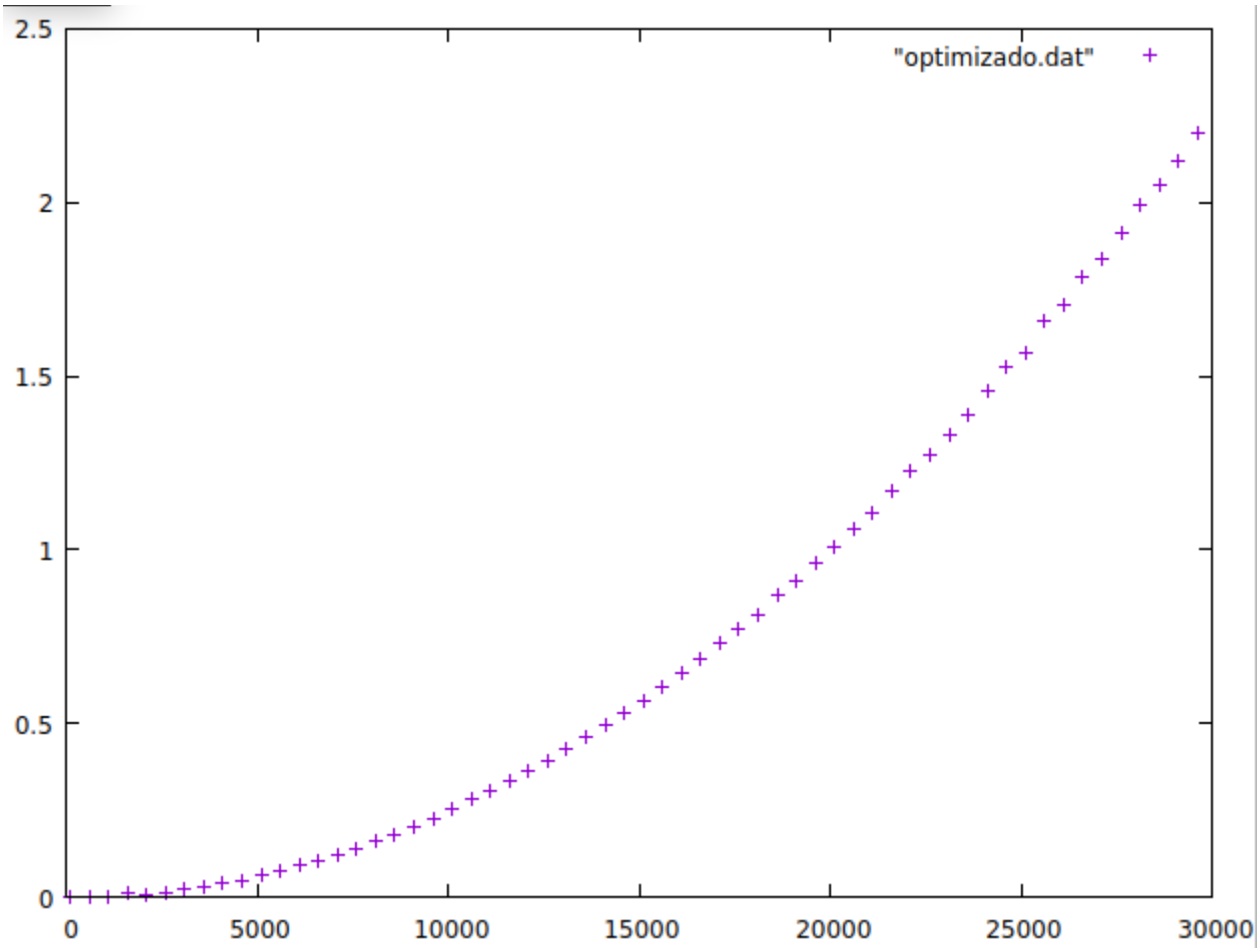
```
Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a                = 1.12081e-09     +/- 1.027e-10    (9.161%)
b                = 7.73928e-06     +/- 1.765e-06    (22.81%)

correlation matrix of the fit parameters:
                 a      b
a                1.000
b               -0.864  1.000
```

gnuplot> plot 'datos.dat' using 1:2 with points, f(x) title 'Ajuste'



# EJERCICIO 6

**1) Replicación con -O3**




- Comparando con la primera gráfica, se ve una pequeña mejoría de 0.25 aprox en 30000.

# EJERCICIO 7

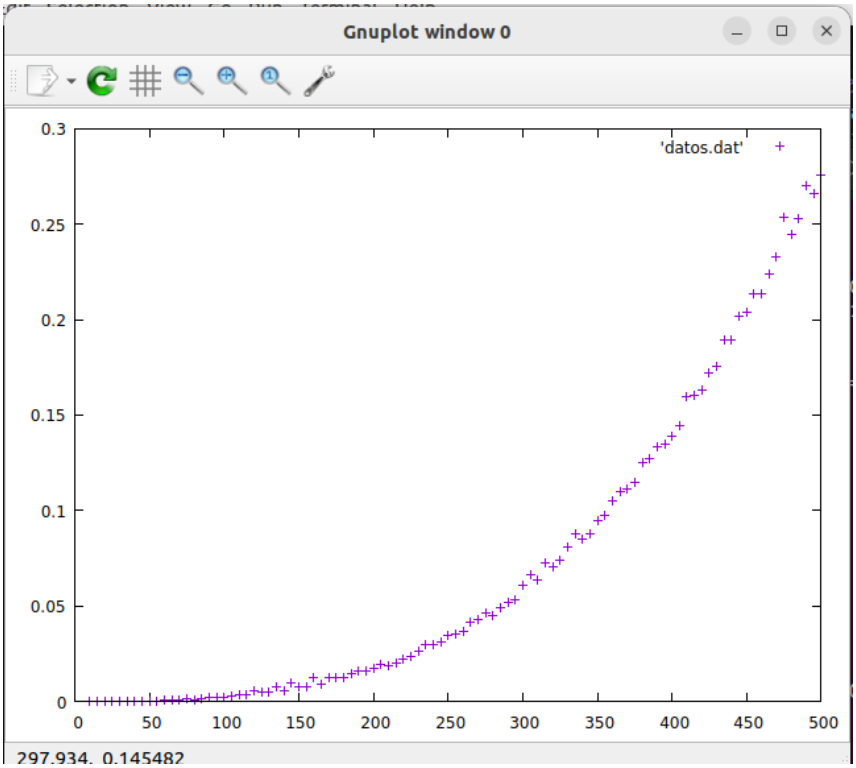
## 1) Código multiplicación

```cpp
void multiplicarMatrices(int **A, int **B, int **C, int tam) {
    for (int i = 0; i < tam; i++) { //En el peor de los casos nuestro for haría tam iteraciones por lo tanto O(n)
        for (int j = 0; j < tam; j++) { // En el peor de los casos nuestro for haría tam iteraciones por lo tanto O(n)
            C[i][j] = 0; // Al ser una asignación simple es O(1)
            for (int k = 0; k < tam; k++) { // En el peor de los casos nuestro for haría tam iteraciones por lo tanto O(n)
                C[i][j] += A[i][k] * B[k][j]; // Al ser una asignación simple es O(1)
            }
        }
    }
} // Al estar los tres bucles anidados y gracias a la ley del producto obtenemos que la eficiencia es O(n^3)

int main(int argc, char *argv[]) {
    int tam = atoi(argv[1]);

    int **A = new int*[tam];
    int **B = new int*[tam];
    int **C = new int*[tam];

    for (int i = 0; i < tam; i++) {
        A[i] = new int[tam];
        C[i] = new int[tam];
    }

    for (int i = 0; i < tam; i++) {
        B[i] = new int[tam];
    }


    for (int i = 0; i < tam; i++) {
        for (int j = 0; j < tam; j++) {
            A[i][j] = i + j;
        }
    }
    for (int i = 0; i < tam; i++) {
        for (int j = 0; j < tam; j++) {
            B[i][j] = i * j;
        }
    }

    clock_t tini = clock();
    multiplicarMatrices(A, B, C, tam);
    clock_t tfin = clock();

    double tiempo = double(tfin - tini) / CLOCKS_PER_SEC;
    cout  << tam << "\t" << tiempo << endl;

    for (int i = 0; i < tam; i++) {
        delete[] A[i];
        delete[] C[i];
    }
    for (int i = 0; i < tam; i++) {
        delete[] B[i];
    }
    delete[] A;
    delete[] B;
    delete[] C;
}
```

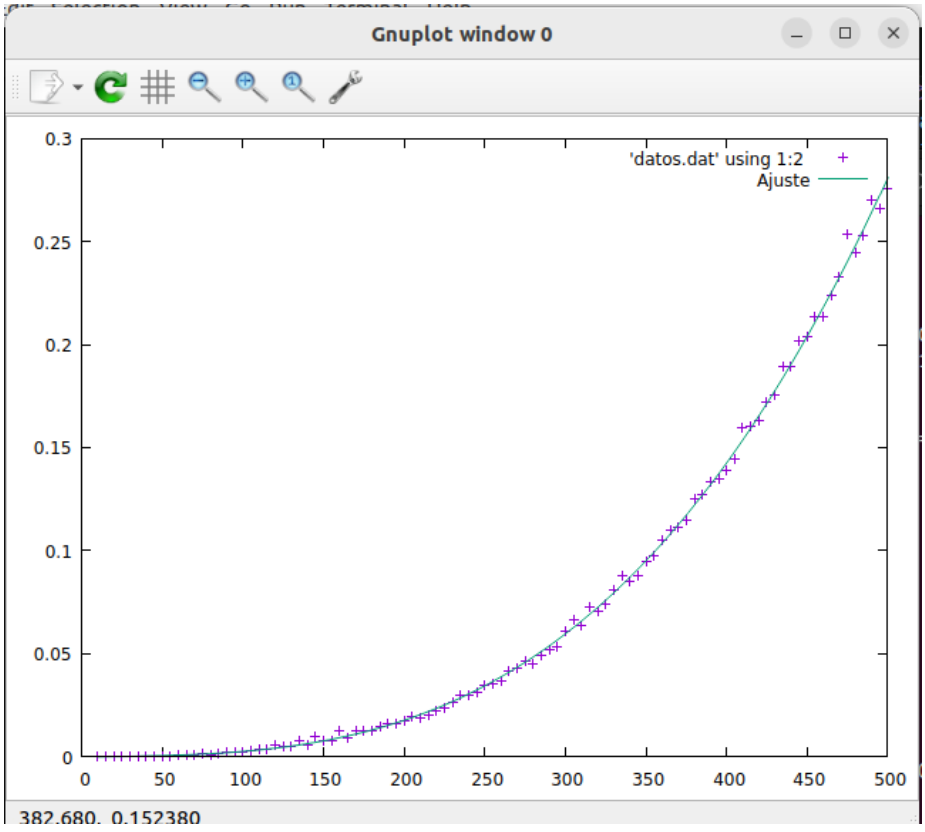
## 2) Cálculo eficiencia teórica

```cpp
void multiplicarMatrices(int **A, int **B, int **C, int tam) {
    for (int i = 0; i < tam; i++) { //En el peor de los casos nuestro for haría tam iteraciones por lo tanto O(n)
        for (int j = 0; j < tam; j++) { // En el peor de los casos nuestro for haría tam iteraciones por lo tanto O(n)
            C[i][j] = 0; // Al ser una asignación simple es O(1)
            for (int k = 0; k < tam; k++) { // En el peor de los casos nuestro for haría tam iteraciones por lo tanto O(n)
                C[i][j] += A[i][k] * B[k][j]; // Al ser una asignación simple es O(1)
            }
        }
    }
} // Al estar los tres bucles anidados y gracias a la ley del producto obtenemos que la eficiencia es O(n^3)
```

## 3) Cálculo eficiencia empírica



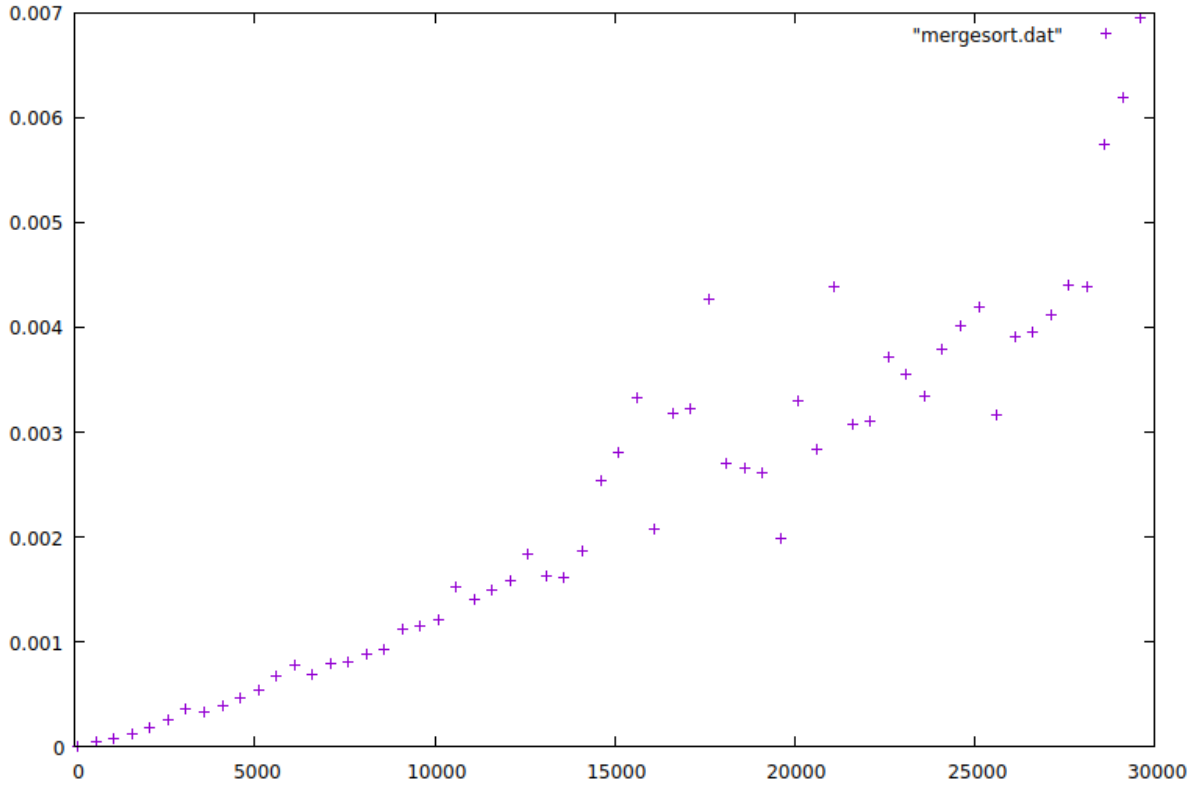

## 4) Ajuste curva



# EJERCICIO 8

### 1) Eficiencia Empírica

- El fichero csh es igual al usado en anteriores casos (burbuja).



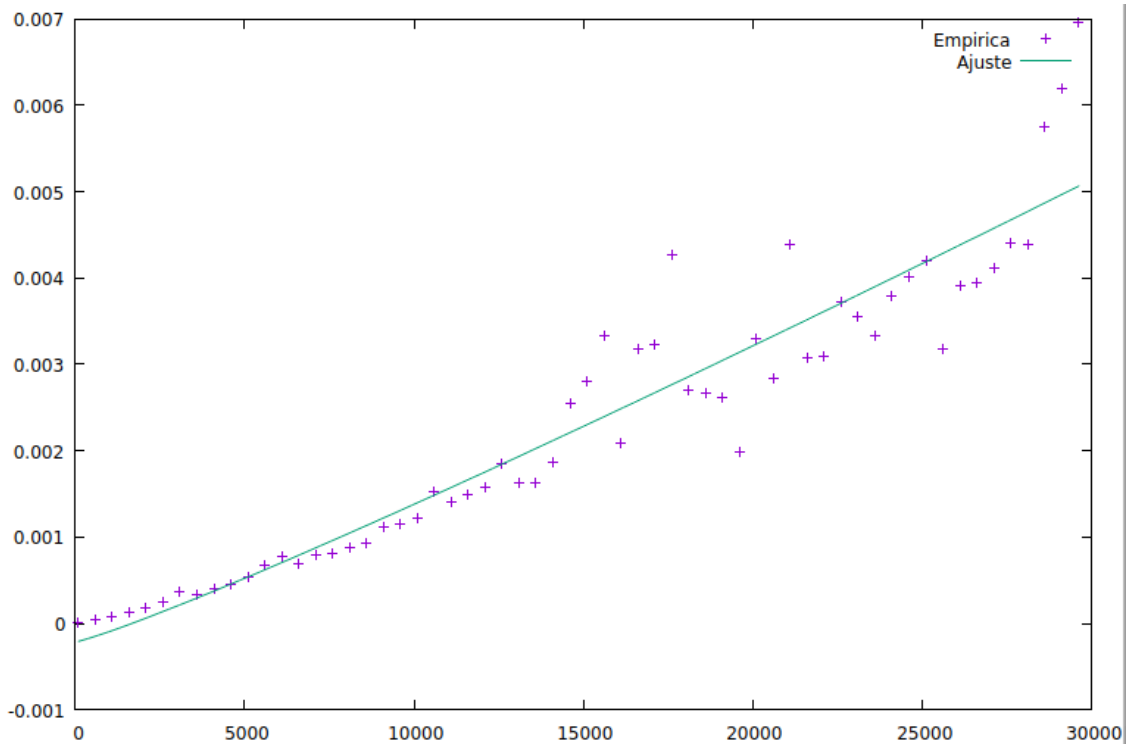

### 2) Ajuste Regresión

gnuplot> f(x) = a * x * log(x) + b

gnuplot> fit f(x) 'mergesort.dat' using 1:2 via a, b

```
Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a               = 1.7312e-08       +/- 7.668e-10    (4.429%)
b               = -0.000214663     +/- 0.0001315    (61.26%)

correlation matrix of the fit parameters:
                a      b
a               1.000
b              -0.849  1.000
```

gnuplot> plot 'mergesort.dat' using 1:2 with points title 'Empirica', f(x) title 'Ajuste'




**3) ¿Cómo afecta UMBRAL_MS?**

Pruebo con 50, 100, 150, 200

Caso1, 50)

1000 5.6193e-05

Caso2, 100)

1000 7.9156e-05

Caso3, 150)

1000 0.000125396

Caso4, 200)

1000 0.000117045

- EL TIEMPO VA A MÁS CONFORME AUMENTA UMBRAL_MS según este pequeño estudio.