Práctica 1: Eficiencia

Alejandro Palencia Blanco
Inma Marín Carballo

Hardware Usado

Esta práctica se ha realizado en un ordenador con estas características:

Memoria	7,7 GiB
Procesador	Intel® Core™ i5-6300HQ CPU @ 2.30GHz x 4
Gráficos	GeForce GTX 950M/PCIe/SSE2
Tipo de SO	Ubuntu Linux 16.04 64bits
Disco	55,2 GB

Compilador

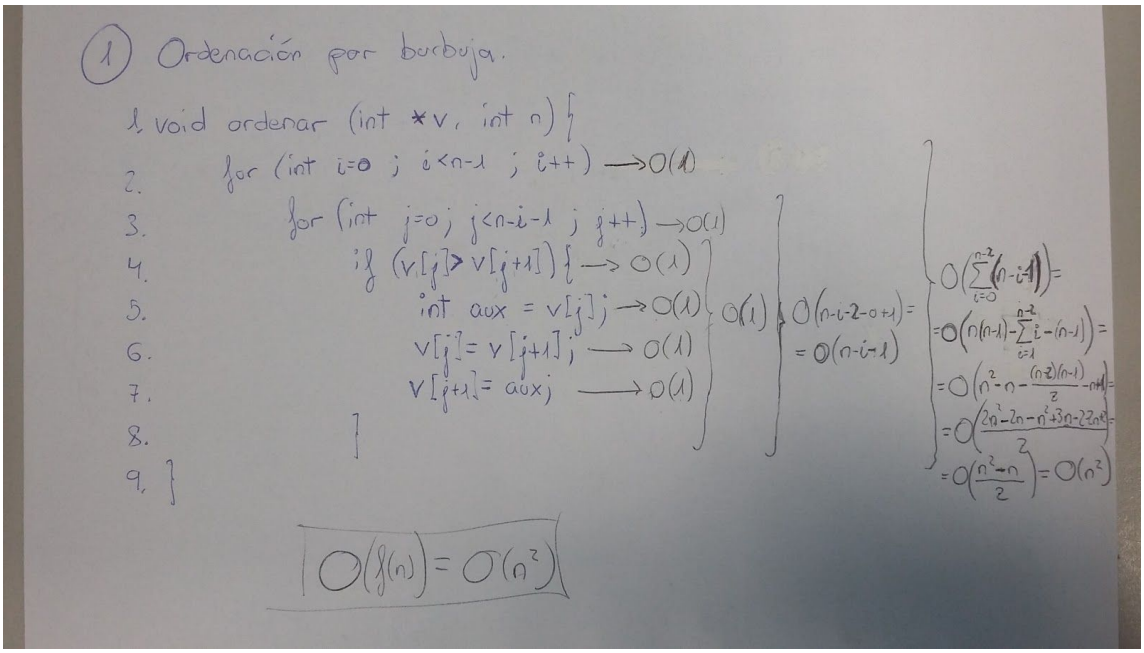
El compilador usado: g++ Ubuntu 5.4.0-6ubuntu1 16.04
Opciones de compilador:

g++ *programa*.cpp -o *programa*
g++ -O3 *programa*.cpp -o *programa*_optimizado

Ejercicio 1

- Eficiencia teórica:

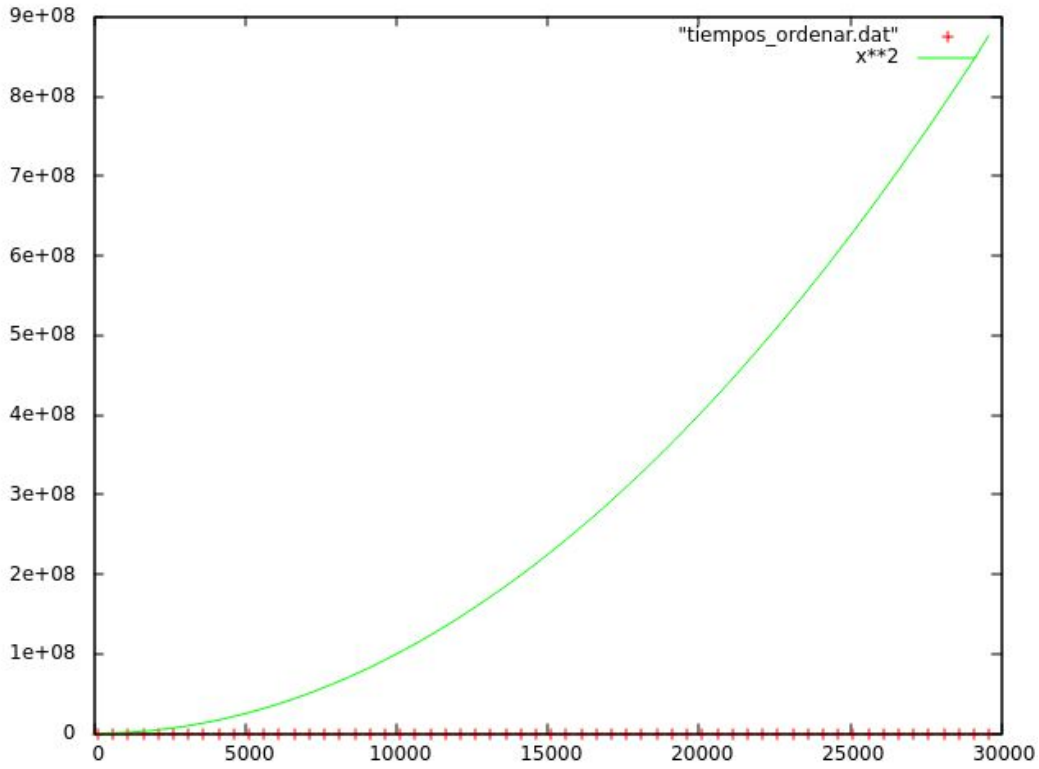
Hemos determinado que la eficiencia teórica de la función es $O(n^2)$



- *ejecuciones_ordenacion.sh* :

```
#!/bin/bash

inicio=100
fin=30000
incremento=500
ejecutable="ordenacion"
salida="tiempos_ordenar.dat"

i=$inicio
echo > $salida
while [ $i -lt $fin ]
do
  echo "Ejecución tam = " $i
  echo `./$ejecutable $i 10000` >> $salida
  i=$[$i+$incremento]
done
```

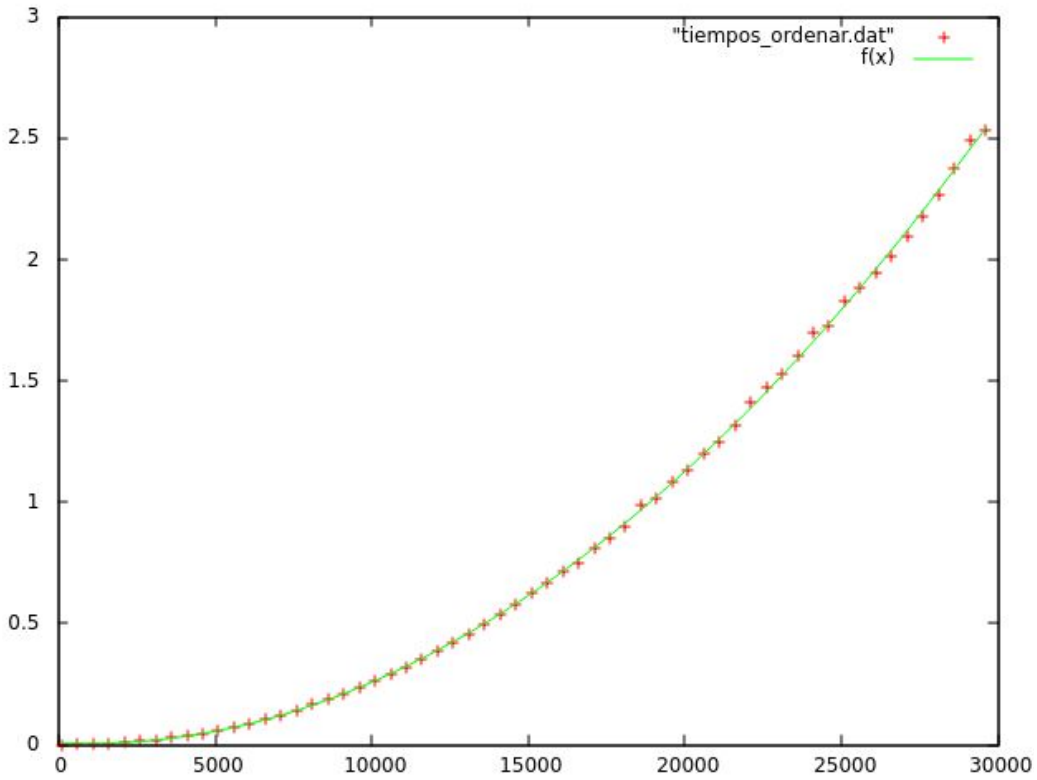
- Plot de las funciones de eficiencia teórica y empírica:



Al dibujar las dos funciones en la misma gráfica, comprobamos que la eficiencia teórica tiene una escala muy superior a la empírica. Esto ocurre porque no hemos hallado la función exacta que nos da la eficiencia teórica, sino su clase (que es n^2).

Ejercicio 2

Tras el reajuste por regresión:



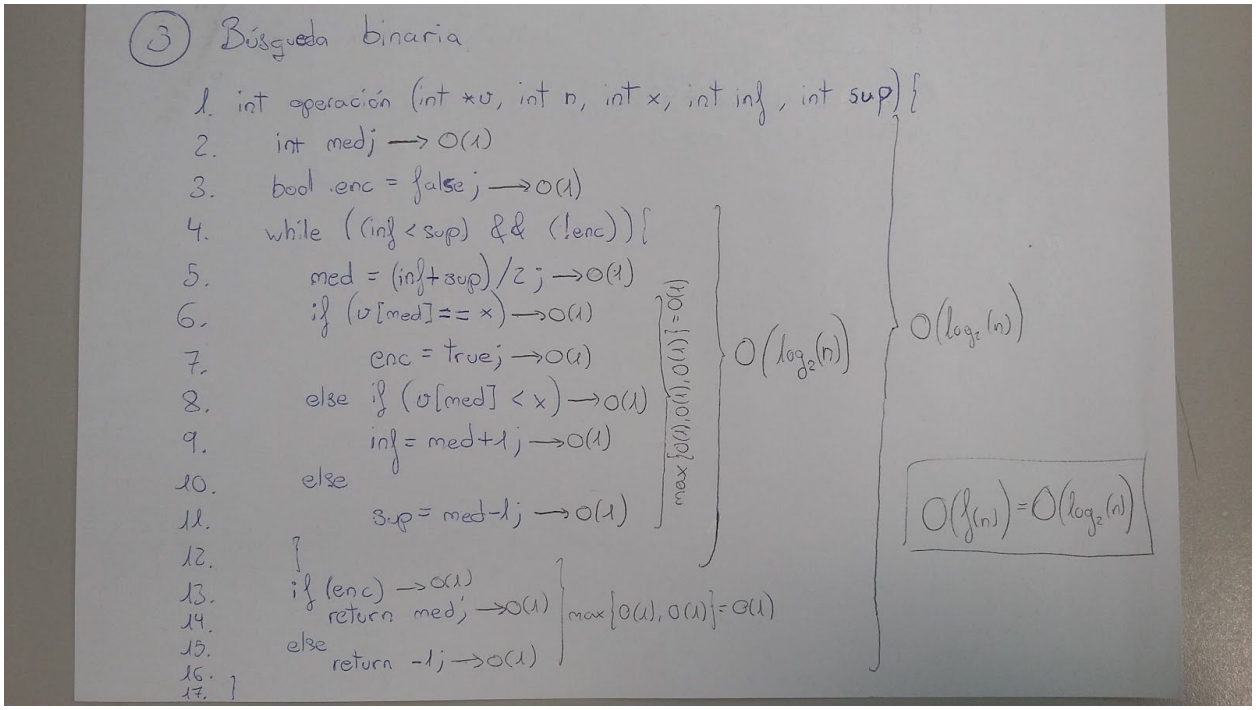
Podemos comprobar que la eficiencia empírica y teórica coinciden.

Ejercicio 3

- Qué hace el algoritmo:

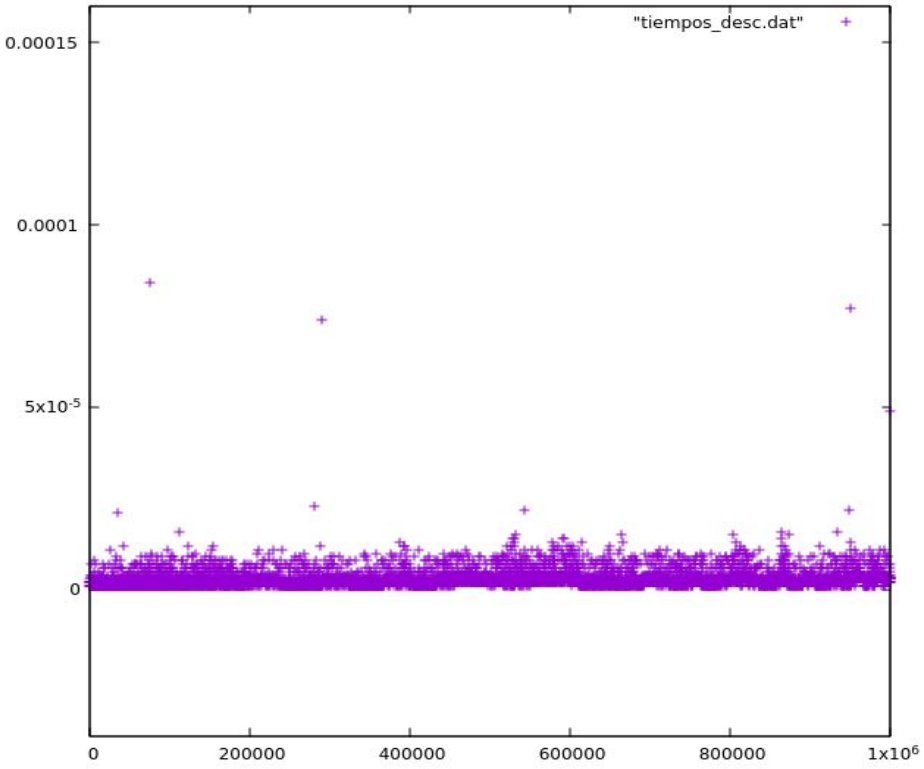
El algoritmo es una búsqueda binaria. El procedimiento necesita cinco parámetros: un vector ya ordenado, el número de componentes del vector (aunque no se use en el algoritmo), un valor a buscar y dos índices (uno inferior y otro superior, que acotarán el vector de forma que búsqueda se realizará entre las componentes cuyos índices estén entre estos dos parámetros).
El algoritmo devuelve el valor de *med* (la posición que contiene el valor buscado) o -1 en el caso de que no exista tal valor en el vector.
Las precondiciones son: *v* tiene que estar ordenado de forma creciente y *0<=inf<sup<n*. Es necesario que *inf* sea estrictamente menor que *sup*, ya que se podría dar un caso extremo en el que *x=v[inf]* con *inf=sup* pero no se entraría en el bucle while, por lo que el algoritmo devolvería -1 (no encontrado).
El funcionamiento del algoritmo es sencillo: se trabaja con dos variables locales: *med*, que almacenará la posición del valor buscado, y *enc*, que inicializada a false, indica si tal posición ha sido ya encontrada. Mediante un bucle while y una encadenación de if-else, en cada iteración se consigue reducir el número de posiciones candidatas a contener el valora la mitad. La variable *med* guarda la posición que se encuentra a mitad de camino entre *inf* y *sup*, es decir, *(sup-inf)/2*. Si *v[med]* es menor al valor buscado, descartamos la mitad inferior del vector (podemos hacerlo gracias a que *v* está ordenado) posicionando *inf* en *med+1*; si es mayor, descartaremos la superior posicionando *sup* en *med-1*, sin olvidarnos de posicionar med en la mitad del nuevo rango en ambos casos. Cuando se encuentre el valor buscado (*v[med]==x*), *enc* pasará a true y saldremos del bucle. En el caso de que inf y sup se lleguen a encontrar, significará que no existe el valor buscado, por lo que se saldrá del bucle con *enc=false*. Esta última variable le indicará a la función si ha de devolver *med* o -1.
El hecho de que en cada iteración se descarte la mitad del rango de búsqueda hace que la función que nos da la eficiencia sea logarítmica (como se verá en la eficiencia teórica). Esto hace de la búsqueda binaria un algoritmo de búsqueda muy eficiente frente a otros como la secuencial. Sin embargo, requiere que el vector esté ordenado, por lo que será necesario aplicar un algoritmo que lo ordene previamente.

- Eficiencia teórica:



- Eficiencia empírica:

Después de ejecutar el script sin modificar el algoritmo, obtuvimos los siguientes datos:



Para solucionar el problema, modificamos las siguientes lineas de código para que así ctime pudiese calcular los datos con mayor precisión:

```
...

// Algoritmo a evaluar
for (int rep=1; rep<=10000; ++rep)
  operacion(v,tam,tam+rep,0,tam-1);

…


// Mostramos resultados
cout << tam << "\t" << (tfin-tini)/(10000*(double)CLOCKS_PER_SEC) << endl;

...
```

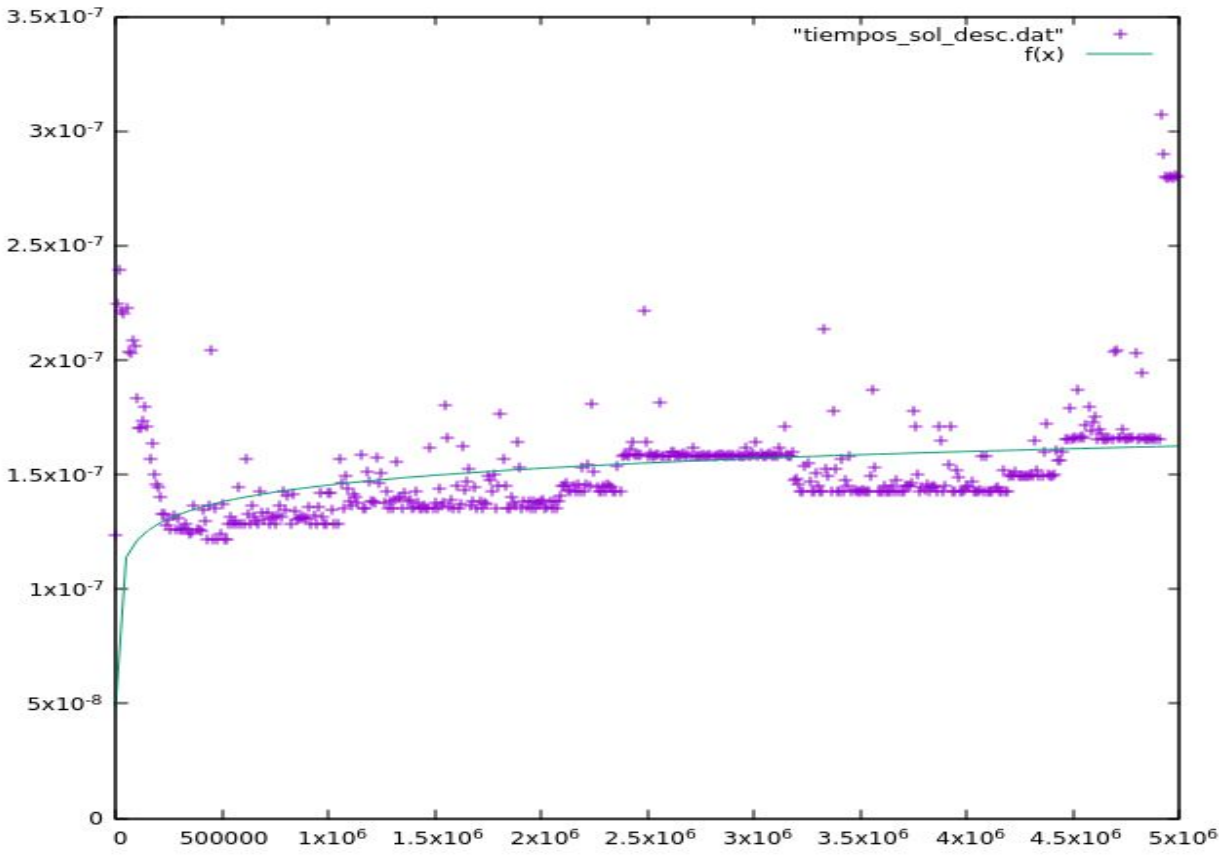
El script usado para el cálculo de los datos correctos es el siguiente:

```
#!/bin/bash
inicio=100
fin=5000000
incremento=10000
ejecutable="solucionado_desc"
salida="tiempos_sol_desc.dat"

i=$inicio
echo > $salida
while [ $i -lt $fin ]
do
  echo "Ejecución tam = " $i
  echo `./$ejecutable $i` >> $salida
  i=$[$i + $incremento]
done
```

Reajuste:

```
After 4 iterations the fit converged.
final sum of squares of residuals : 3.35602e-13
rel. change during last iteration : -2.24317e-11

degrees of freedom    (FIT_NDF)                        : 498
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 2.59596e-08
variance of residuals (reduced chisquare) = WSSR/ndf   : 6.739e-16

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a                  = 7.33327e-09   +/- 7.535e-10  (10.27%)
b                  = 0.933148      +/- 1.38       (147.8%)

correlation matrix of the fit parameters:
                   a      b
a                  1.000
b                  -0.997 1.000
```

Plot de los datos correctos con su reajuste:



Ejercicio 4

Modificamos el código de las siguientes maneras para obtener

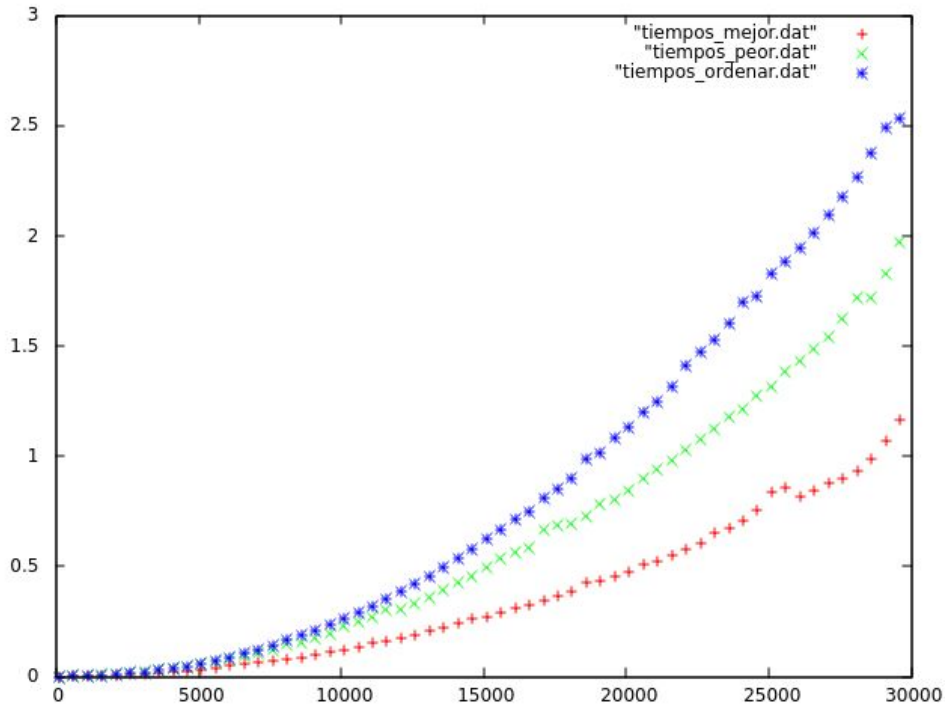
- mejor código posible:

```
for (int i=0; i<tam; i++)          // Recorrer vector
        v[i] = i;                  // Generar vector ordenado
```

- peor código posible:

```
for (int i=0; i<tam; i++)          // Recorrer vector
        v[i] = tam-i;              // Generar vector ordenado de forma inversa
```

Tras ejecutar el script correspondiente a cada uno y comparar los datos obtenidos en una gráfica, obtenemos:



Sorprendentemente, obtenemos que la eficiencia del caso más desfavorable, donde el vector a ordenar está en orden inverso, es mejor que la eficiencia del vector aleatorio.

Ejercicio 5

- Eficiencia teórica

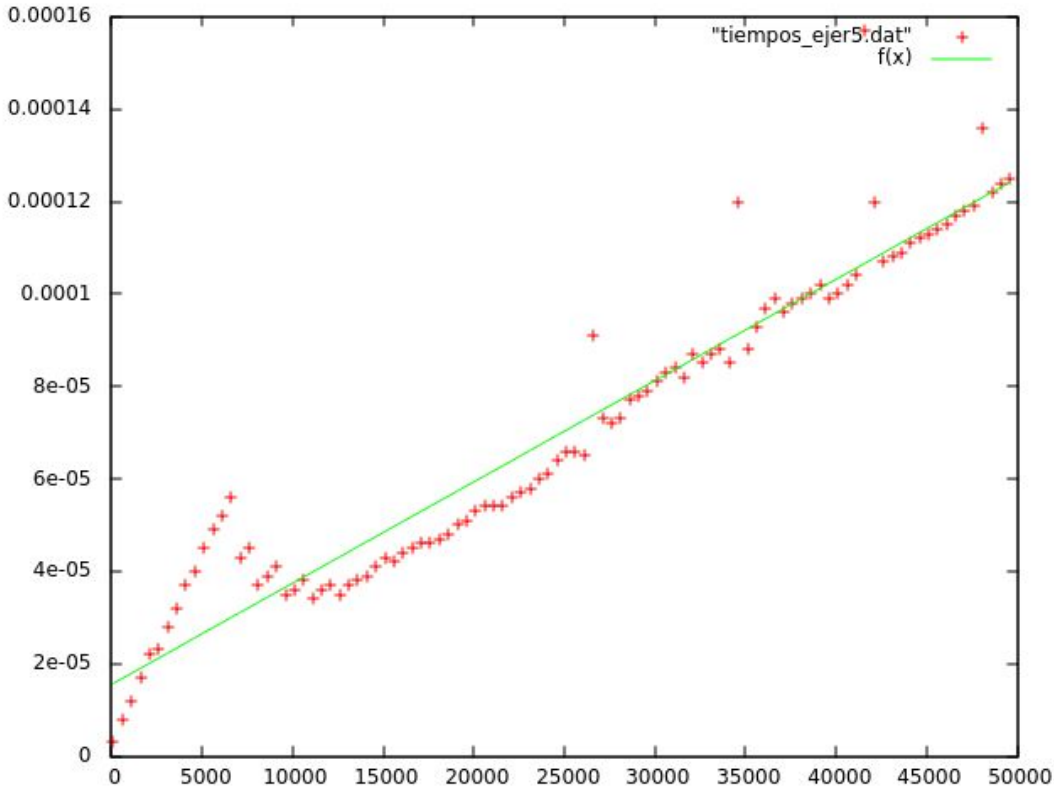
Si suponemos que el vector está ordenado, se entrará tanto en el primer bucle for (cambio=true al iniciar el algoritmo) como en el segundo, además de asignar cambio=false. En el segundo bucle se realizarán n-i-2 iteraciones y en ninguna de ellas se cumplirá la condición del if (al estar el vector ordenado, ningún elemento de índice menor a otro guardará un valor mayor). Por tanto, cambio se mantiene en false y al realizar la segunda iteración del primer bucle, la condición no se cumple y termina. Así que la función que mide la eficiencia empírica en el mejor caso es de clase n, es decir, lineal.

- Eficiencia empírica

Realizamos un reajuste por regresión para comparar la eficiencia teórica y la empírica:

```
After 8 iterations the fit converged.
final sum of squares of residuals : 8.89837e-09
rel. change during last iteration : -3.56095e-08

degrees of freedom    (FIT_NDF)                       : 98
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 9.52889e-06
variance of residuals (reduced chisquare) = WSSR/ndf   : 9.07997e-11

Final set of parameters            Asymptotic Standard Error
=======================            ==========================

a               = 2.19204e-09      +/- 6.602e-11    (3.012%)
b               = 1.54677e-05      +/- 1.897e-06    (12.27%)


correlation matrix of the fit parameters:

               a      b
a               1.000
b              -0.865  1.000
```

Y representamos las gráficas:



Ejercicio 6

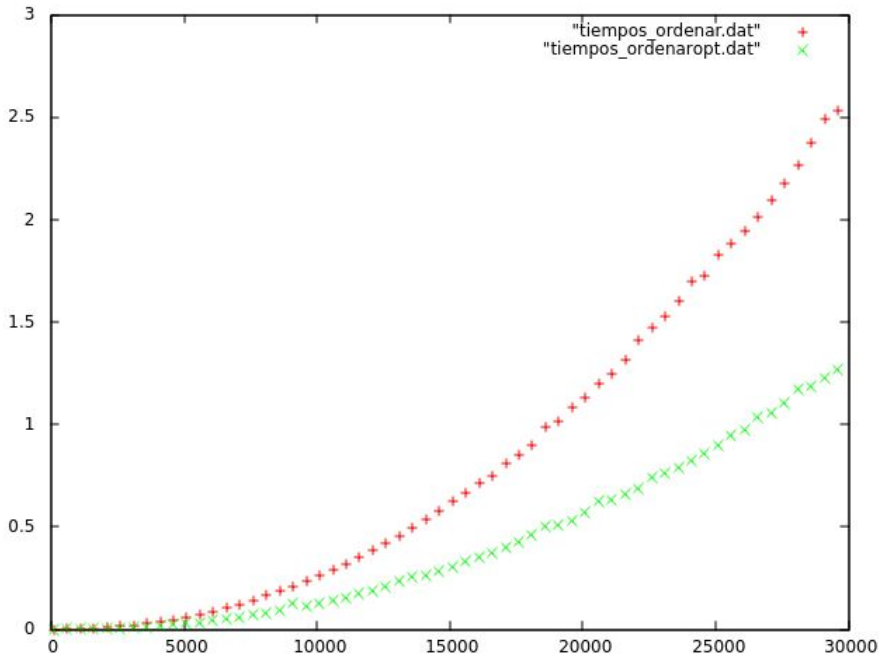
Realizamos una optimización del programa ordenación:

g++ -O3 ordenacion.cpp -o ordenacion_optimizado

Tras esta obtener este ejecutable y reescribir el scrip de ejecuciones:

```
#!/bin/bash

inicio=100
fin=30000
incremento=500
ejecutable="ordenacion_optimizado"
salida="tiempos_ordenaropt.dat"

i=$inicio
echo > $salida
while [ $i -lt $fin ]
do
  echo "Ejecución tam = " $i
  echo `./$ejecutable $i 10000` >> $salida
  i=$[$i+$incremento]
done
```

Ejecutamos el scrip y comparamos los datos del programa *ordenacion* y *ordenacion_opt,* y, evidentemente vemos que la eficiencia del programa optimizado es mucho mayor:



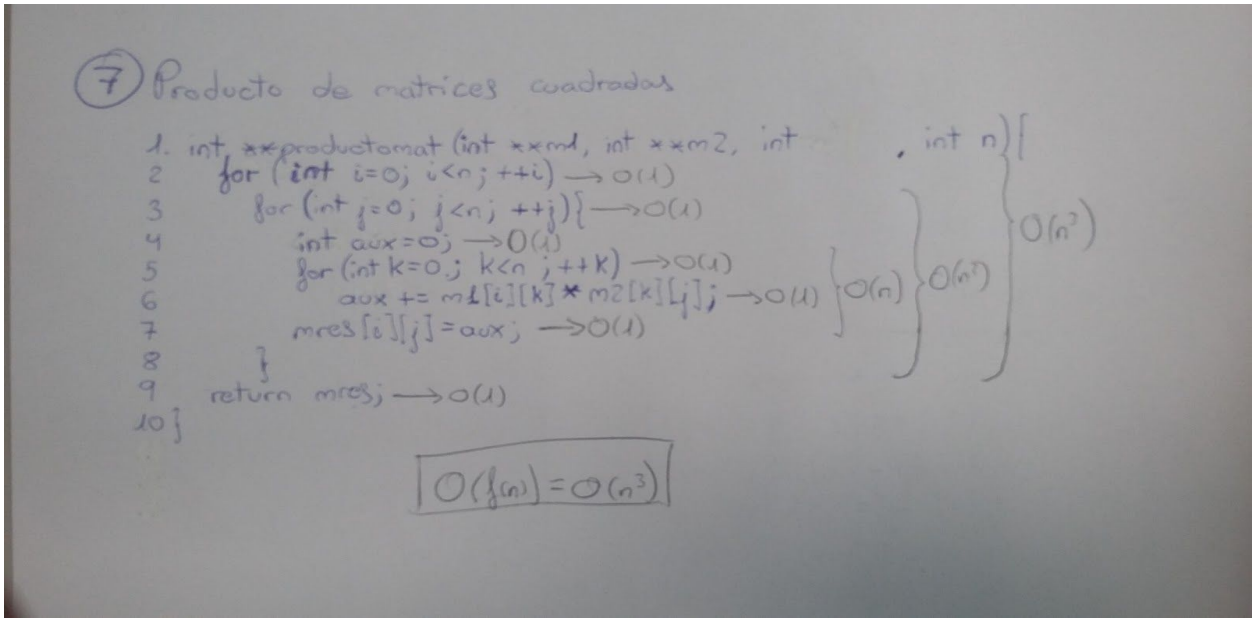
Ejercicio 7

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

using namespace std;


// Realiza el producto de dos matrices cuadradas, ambas de dimensión n*n
int **productomat(int **m1, int **m2, int **mres, int n){
  for (int i=0; i<n; ++i)                    // Recorre m1 por filas
    for (int j=0; j<n; ++j){                 // Recorre m2 por columnas
      int aux=0;
      for (int k=0; k<n; ++k)                // Obtiene un elemento de mres
        aux+=m1[i][k]*m2[k][j];
      mres[i][j]=aux;
    }

  return mres;
}


void sintaxis()
{
  cerr << "Sintaxis:" << endl;
  cerr << "  TAM: Tamaño de las matrices (>0)" << endl;
  cerr << "  VMAX: Valor máximo (>0)" << endl;
  cerr << "Se generan dos matrices 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 **m1=new int*[tam];         // Reserva de memoria para la matriz 1
m1[0]=new int[tam*tam];
for (int i=1; i<tam; ++i)
  m1[i]=m1[i-1]+tam;

int **m2=new int*[tam];         // Reserva de memoria para la matriz 2
m2[0]=new int[tam*tam];
for (int i=1; i<tam; ++i)
  m2[i]=m2[i-1]+tam;

int **mres=new int*[tam];       // Reserva de memoria para la matriz mres
mres[0]=new int[tam*tam];
for (int i=1; i<tam; ++i)
  mres[i]=mres[i-1]+tam;

srand(time(0));          // Inicialización del generador de números pseudoaleatorios
for (int i=0; i<tam; i++)  // Recorrer matriz 1
  for (int j=0; j<tam; j++){
    m1[i][j] = rand() % vmax;  // Generar aleatorio [0,vmax[ en m1
    m2[i][j] = rand() % vmax;  // Generar aleatorio [0,vmax[ en m2
  }

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

productomat(m1,m2,mres,tam); // Calculamos el producto de matrices

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

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

delete [] m1[0];         // Liberamos memoria dinámica
delete [] m1;
delete [] m2[0];
delete [] m2;
```

```
  delete [] mres[0];
  delete [] mres;
}
```

- Eficiencia teórica



- Eficiencia empírica

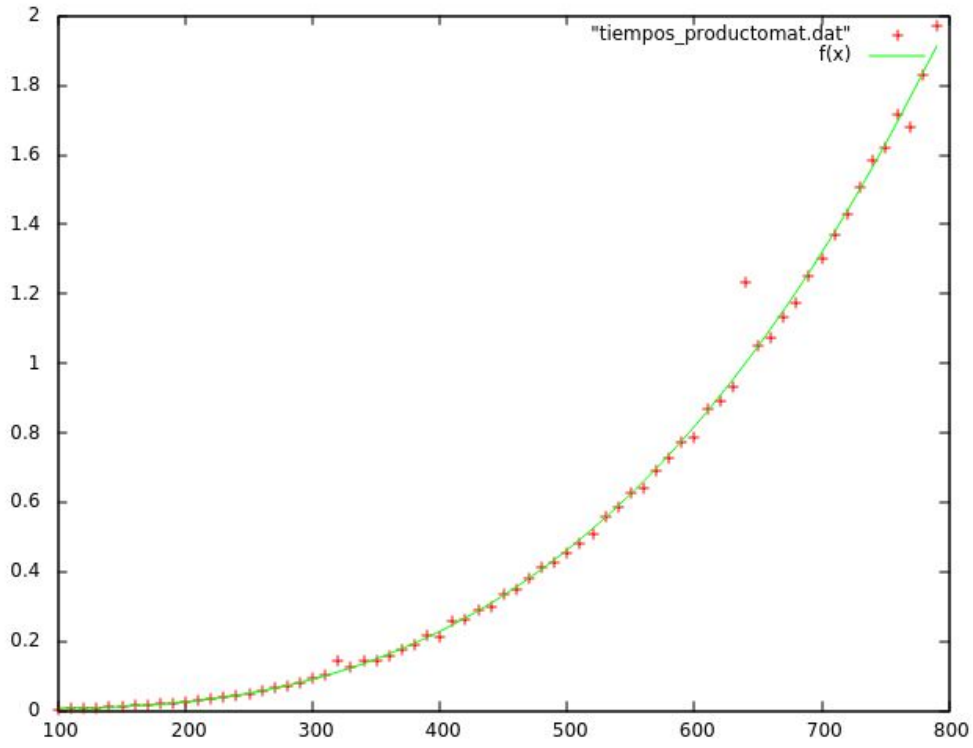
Script:

```
#!/bin/bash
inicio=10
fin=460
incremento=10
ejecutable="productomat"
salida="tiempos_mat.dat"

i=$inicio
echo > $salida
while [ $i -lt $fin ]
do
  echo "Ejecución tam = " $i
  echo `./$ejecutable $i 10` >> $salida
  i=$[$i + $incremento]
done
```

Reajuste:

```
After 11 iterations the fit converged.
final sum of squares of residuals : 0.00952847
rel. change during last iteration : -2.5571e-10

degrees of freedom    (FIT_NDF)                        : 41
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 0.0152447
variance of residuals (reduced chisquare) = WSSR/ndf   : 0.000232402

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a                  = 6.47899e-09   +/- 1.324e-09  (20.44%)
b                  = -1.05316e-06  +/- 9.261e-07  (87.93%)
c                  = 0.000148351   +/- 0.0001844  (124.3%)
d                  = -0.00314908   +/- 0.009904   (314.5%)

correlation matrix of the fit parameters:
                a      b      c      d
a               1.000
b              -0.987  1.000
c               0.922 -0.971  1.000
d              -0.694  0.774 -0.884  1.000
```

Plot de los datos y el reajuste:



Ejercicio 8

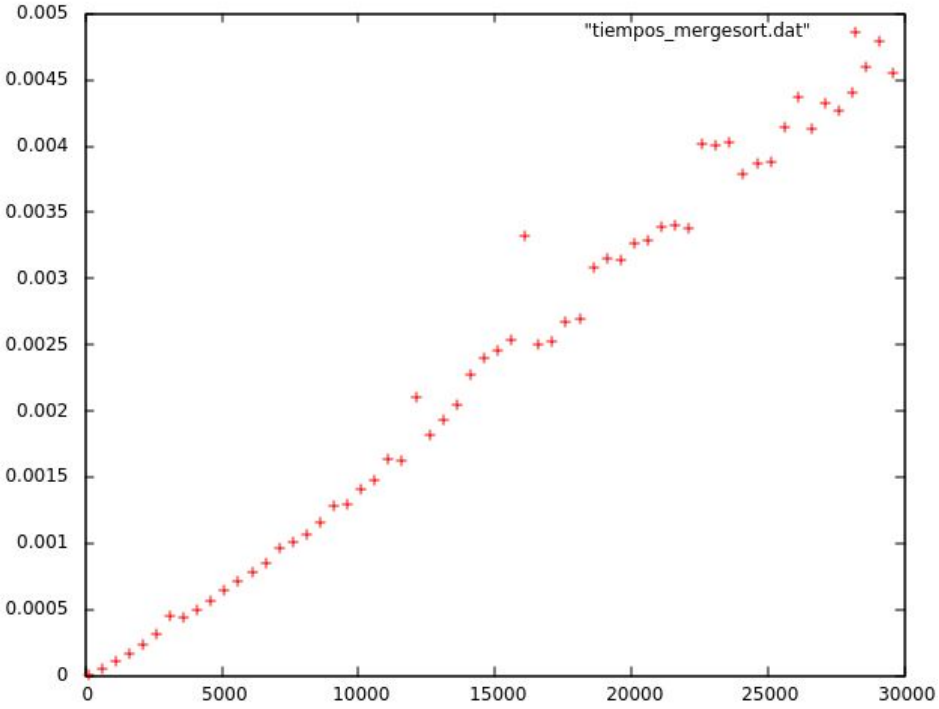
El algoritmo de ordenación por mezcla se basa en la subdivisión del vector que se quiere ordenar en otros de menor longitud recursivamente. Una vez que se tiene un conjunto de vectores cuya longitud no sobrepase un umbral establecido, se ordenan por inserción. Finalmente, los vectores ordenados se fusionan de forma ordenada para llegar al vector completo.

Comprobamos los tiempos con el scrip *ejecuciones_mergesort.sh* :

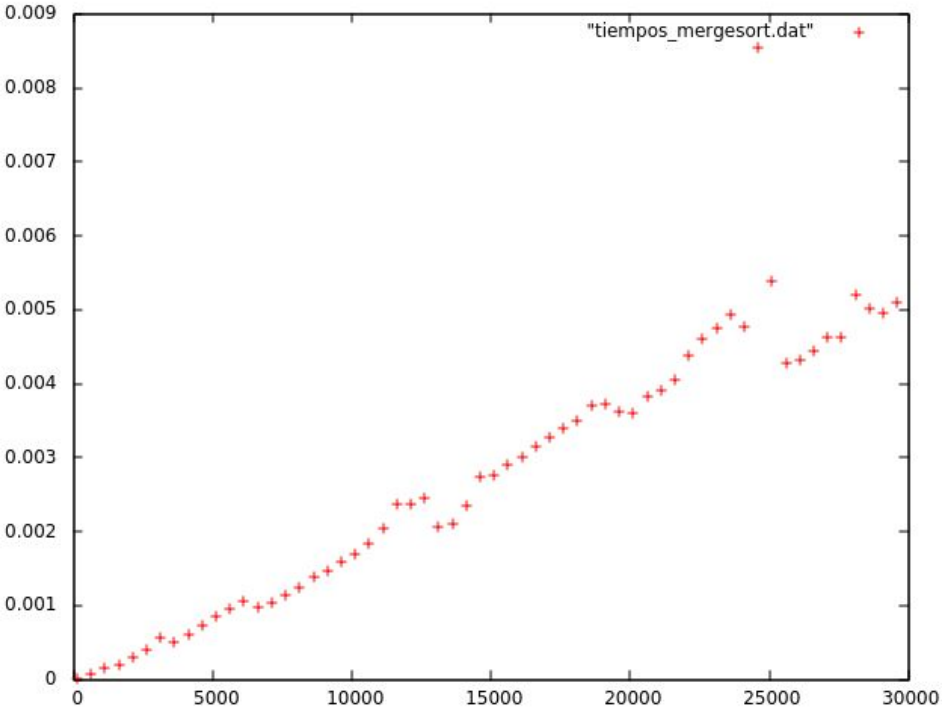
```
#!/bin/bash

inicio=100
fin=30000
incremento=500
ejecutable="mergesort"
salida="tiempos_mergesort.dat"

i=$inicio
echo > $salida
while [ $i -lt $fin ]
do
  echo "Ejecución tam = " $i
  echo `./$ejecutable $i` >> $salida
  i=$[$i+$incremento]
done
```

Al cambiar los valores de MS_UMBRAL, comprobamos que, cuanto más pequeño sea el umbral, más rápido ordena el algoritmo.

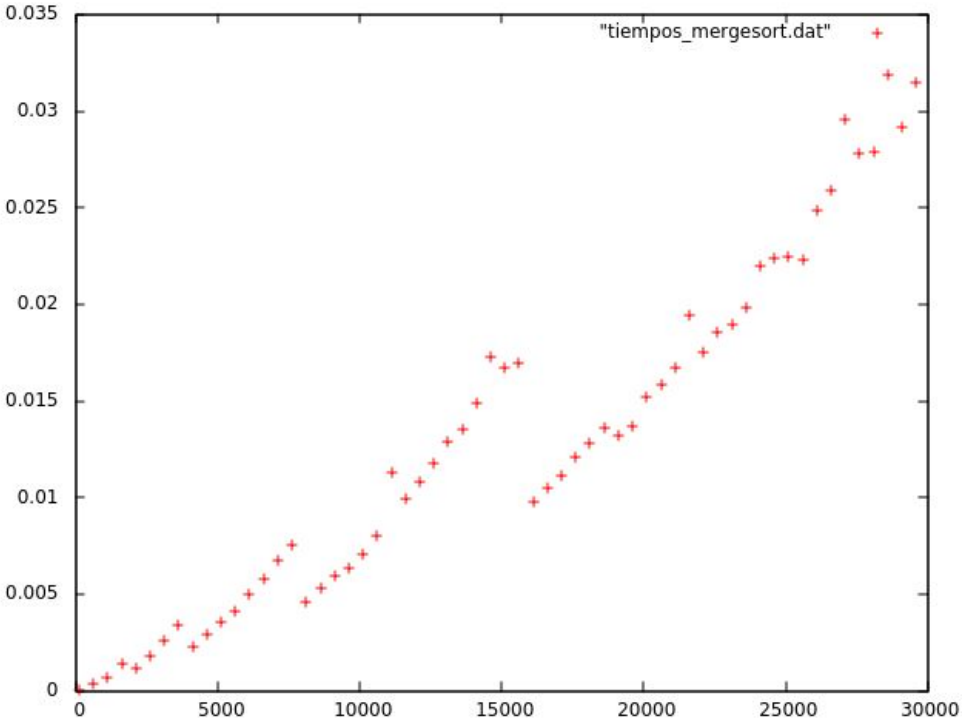
const int UMBRAL_MS = 20;



const int UMBRAL_MS = 100;



const int UMBRAL_MS = 1000;



const int UMBRAL_MS = 10000;