PRÁCTICA I : EFICIENCIA

- **AUTORES:**

  Daniel Monjas Miguélez
  Manuel Horacio Torres Cañero

- **DATOS TÉNICOS**

  Procesador: Intel® Core™ i5-8300H CPU @2.30 GHz 4 cores; 8 threads.
  Memoria RAM: 8 GB.
  Sistema Operativo: Ubuntu 19.04
  Se ha utilizado el compilador g++ con la opción -o. (En el ejercicio 6 se añada -O3).
  Para realizar las gráficas se ha utilizado la herramienta gnuplot.

- **EJERCICIOS**

Ejercicio 1:

Archivo .cpp utilizado:

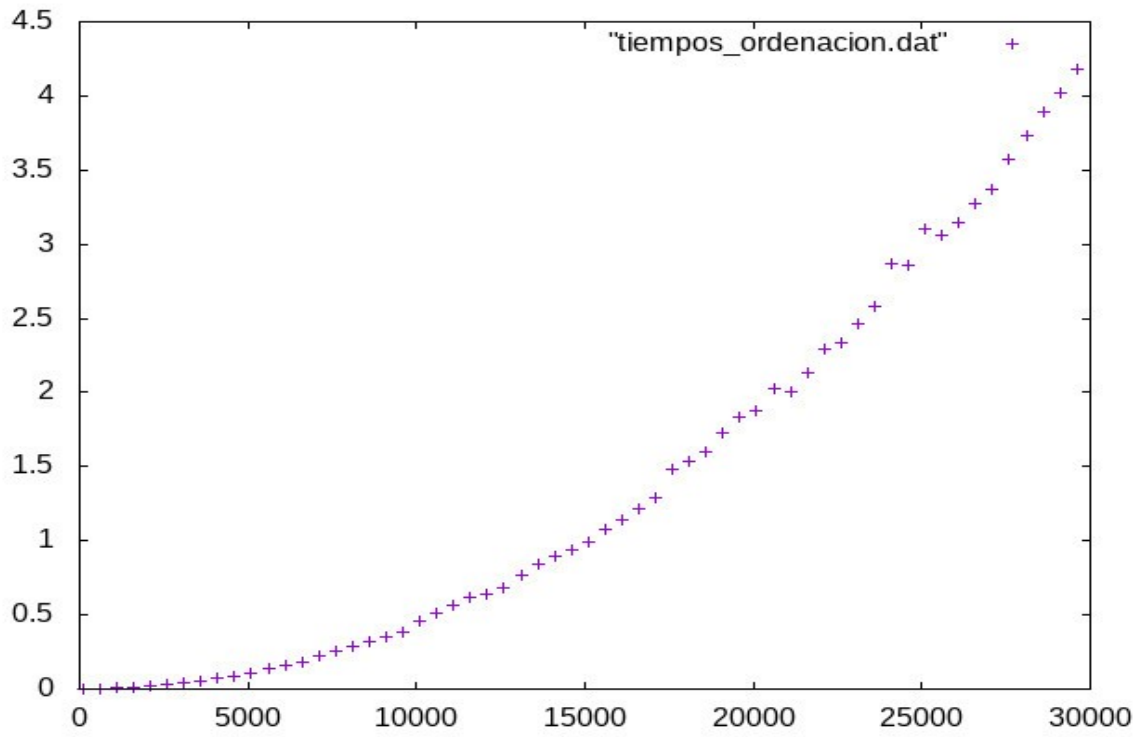
```
#include <iostream>
#include <ctime>    // Recursos para medir tiempos
#include <cstdlib>  // Para generación de números pseudoaleatorios

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]) {
                                swap(v[j],v[j+1]);
                        }
        }


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


  // 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] = rand() % vmax;    // Generar aleatorio [0,vmax[

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


  ordenar(v,tam); // de esta forma forzamos el peor caso

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

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

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

Script utilizado:

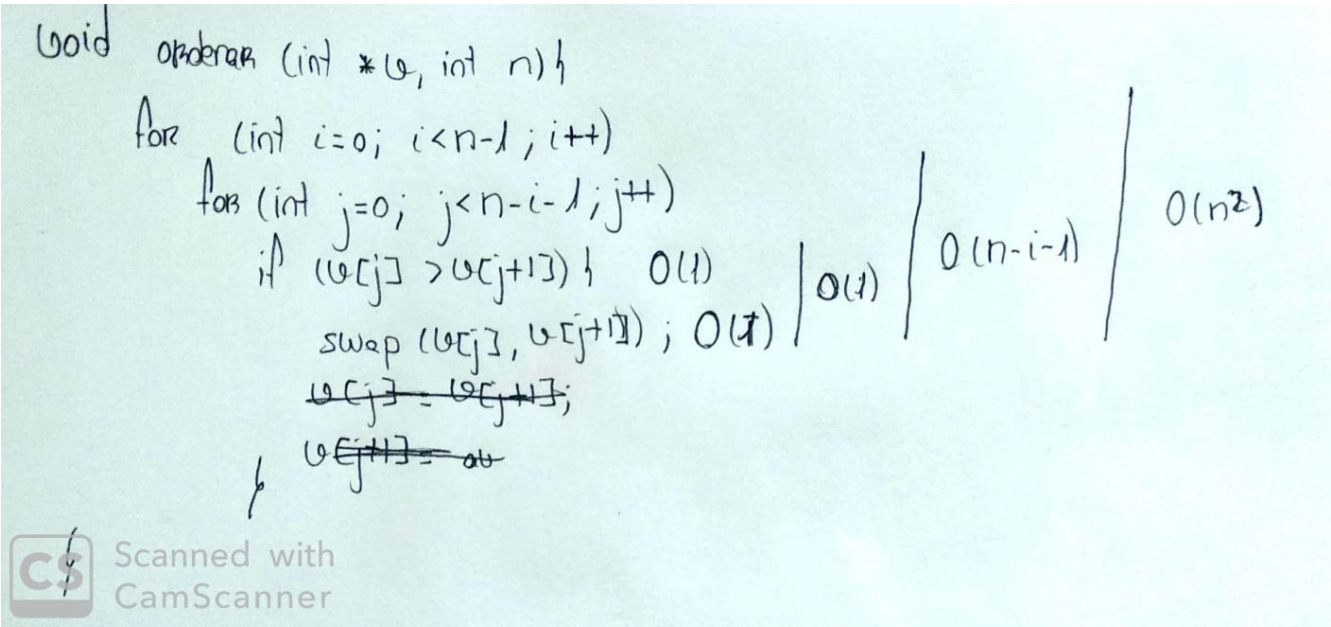
```
#!/bin/csh
@ inicio = 100
@ fin = 30000
@ incremento = 500
set ejecutable = ord_burbuja
set salida = tiempos_ordenacion.dat

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

Ejecutando el script, hemos obtenido un archivo tipo .dat, el cual nos ha servido para realizar nuestra gráfica y así ver la eficiencia empírica

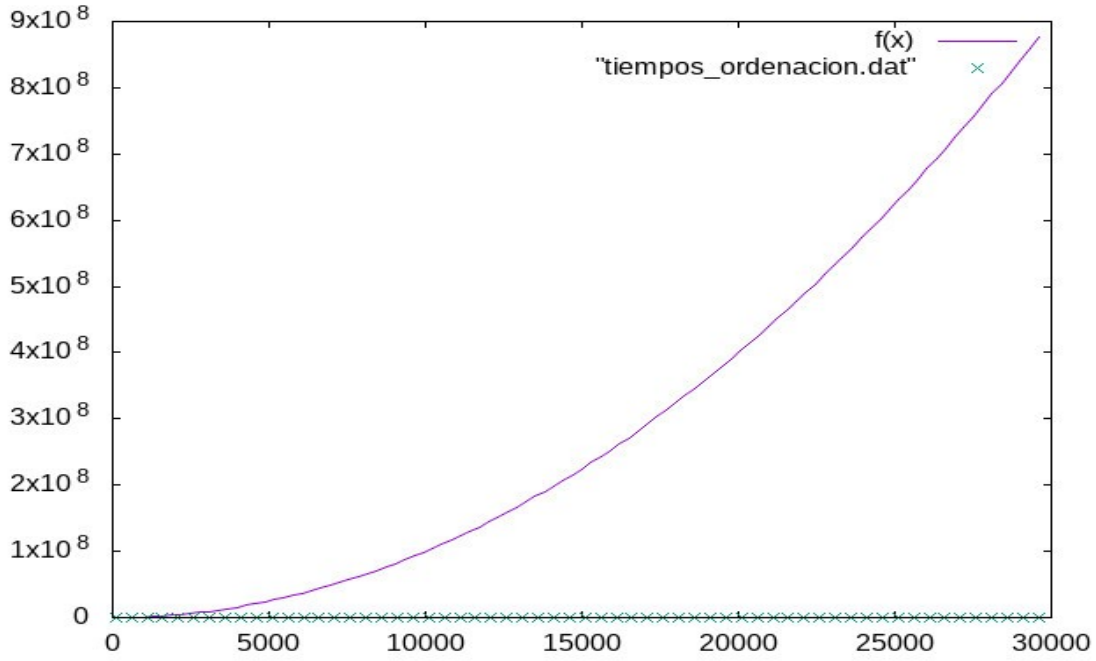
Gráfica de tiempos:



Hemos procedido a calcular la eficiencia teórica, obteniendo como resultado que el algoritmo utilizado sigue una eficiencia de orden $O(n^2)$.



Si comparamos la gráfica de la eficiencia teórica con la gráfica de la eficiencia empírica, obtenemos el siguiente resultado:

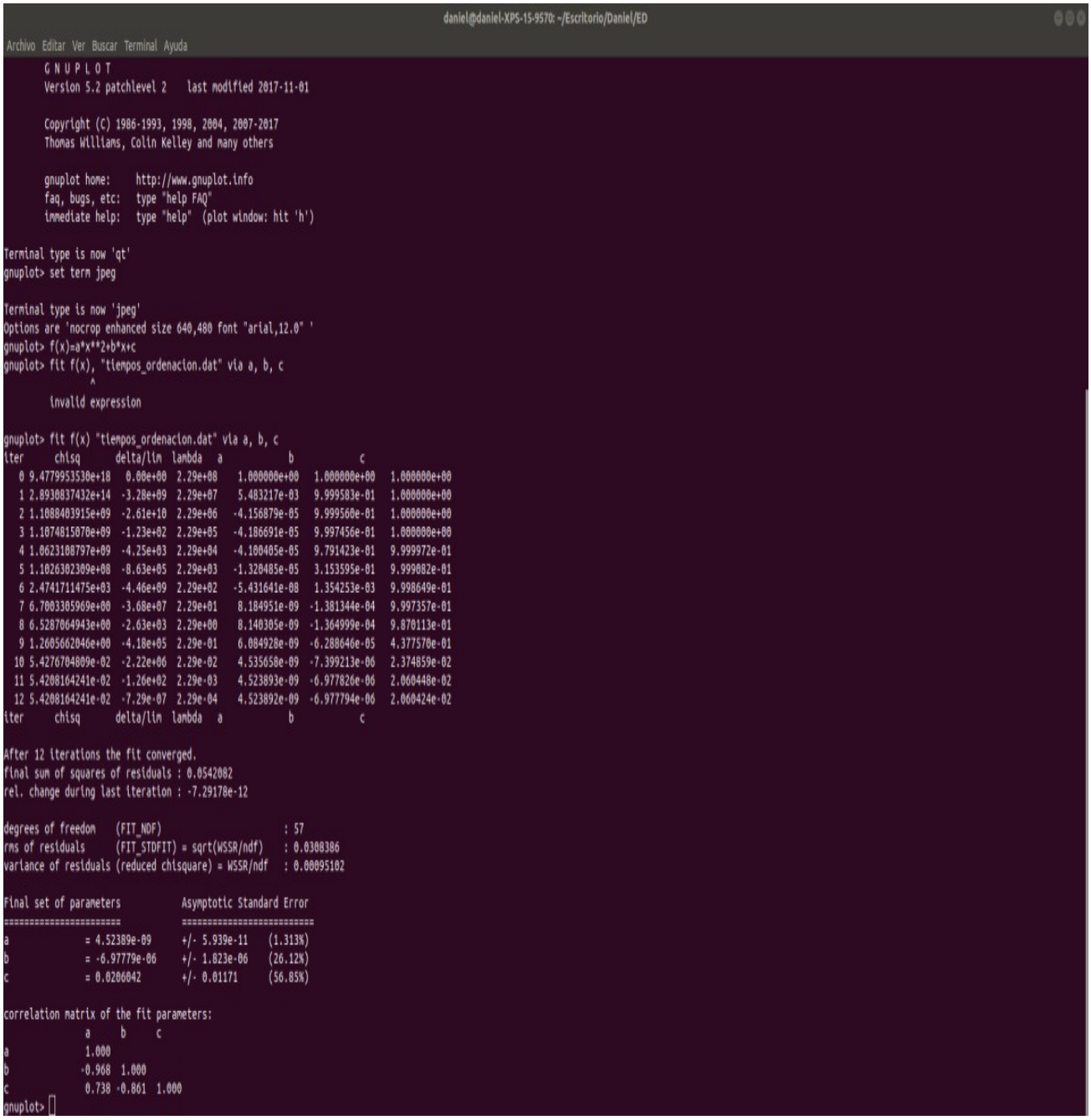


Vemos que las gráficas son muy distintas y esto es debido a que aunque $O(n^2)$, el resultado empírico se ajusta a una función cuadrática de tipo $ax^2 + bx + c$, cuyos coeficientes a,b y c pueden ser muy distintos a los del resultado teórico.

Dificultad:3

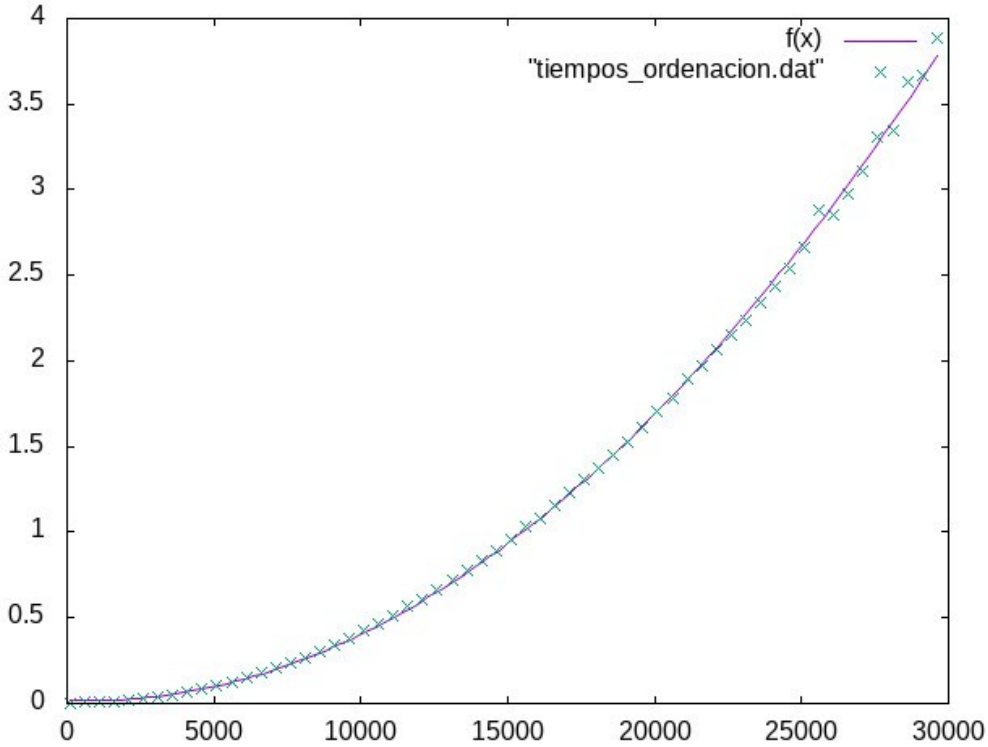
Ejercicio 2

Para realizar este, ejercicio, nos hemos ayudado de la herramienta gnuplot, realizando los siguientes comandos:



```
daniel@daniel-XPS-15-9570: ~/Escritorio/Daniel/ED
Archivo  Editar  Ver  Buscar  Terminal  Ayuda
        G N U P L O T
        Version 5.2 patchlevel 2    last modified 2017-11-01

        Copyright (C) 1986-1993, 1998, 2004, 2007-2017
        Thomas Williams, Colin Kelley and many others

        gnuplot home:     http://www.gnuplot.info
        faq, bugs, etc:   type "help FAQ"
        immediate help:   type "help"  (plot window: hit 'h')

Terminal type is now 'qt'
gnuplot> set tern jpeg

Terminal type is now 'jpeg'
Options are 'nocrop enhanced size 640,480 font "arial,12.0" '
gnuplot> f(x)=a*x**2+b*x+c
gnuplot> fit f(x), "tiempos_ordenacion.dat" via a, b, c
                ^
         invalid expression

gnuplot> fit f(x) "tiempos_ordenacion.dat" via a, b, c
iter      chisq       delta/lim  lambda   a             b             c
   0 9.4779953530e+18   0.00e+00  2.29e+08    1.000000e+00   1.000000e+00   1.000000e+00
   1 2.8938837432e+14  -3.28e+09  2.29e+07    5.483217e-03   9.999583e-01   1.000000e+00
   2 1.1088403915e+09  -2.61e+10  2.29e+06   -4.156879e-05   9.999560e-01   1.000000e+00
   3 1.1074815078e+09  -1.23e+02  2.29e+05   -4.186691e-05   9.997456e-01   1.000000e+00
   4 1.0623108797e+09  -4.25e+03  2.29e+04   -4.100405e-05   9.791423e-01   9.999972e-01
   5 1.1026302309e+08  -8.63e+05  2.29e+03   -1.320485e-05   3.153595e-01   9.999082e-01
   6 2.4741711475e+03  -4.46e+09  2.29e+02   -5.431641e-08   1.354253e-03   9.998649e-01
   7 6.7003305969e+00  -3.68e+07  2.29e+01    8.184951e-09  -1.381344e-04   9.997357e-01
   8 6.5287064943e+00  -2.63e+03  2.29e+00    8.140305e-09  -1.364999e-04   9.870113e-01
   9 1.2605662046e+00  -4.18e+05  2.29e-01    6.084928e-09  -6.288646e-05   4.377570e-01
  10 5.4276704809e-02  -2.22e+06  2.29e-02    4.535658e-09  -7.399213e-06   2.374859e-02
  11 5.4208164241e-02  -1.26e+02  2.29e-03    4.523893e-09  -6.977826e-06   2.060448e-02
  12 5.4208164241e-02  -7.29e-07  2.29e-04    4.523892e-09  -6.977794e-06   2.060424e-02
iter      chisq       delta/lim  lambda   a             b             c

After 12 iterations the fit converged.
final sum of squares of residuals : 0.0542082
rel. change during last iteration : -7.29178e-12

degrees of freedom    (FIT_NDF)                        : 57
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 0.0308386
variance of residuals (reduced chisquare) = WSSR/ndf   : 0.00095102

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a               = 4.52389e-09      +/- 5.939e-11    (1.313%)
b               = -6.97779e-06     +/- 1.823e-06    (26.12%)
c               = 0.0206042        +/- 0.01171      (56.85%)

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>
```

Ahora comparamos la gráfica ajustada con nuestra f(x) obtenida:



Vemos ahora que el resultado empírico se ajusta perfectamente a su curva de regresión.

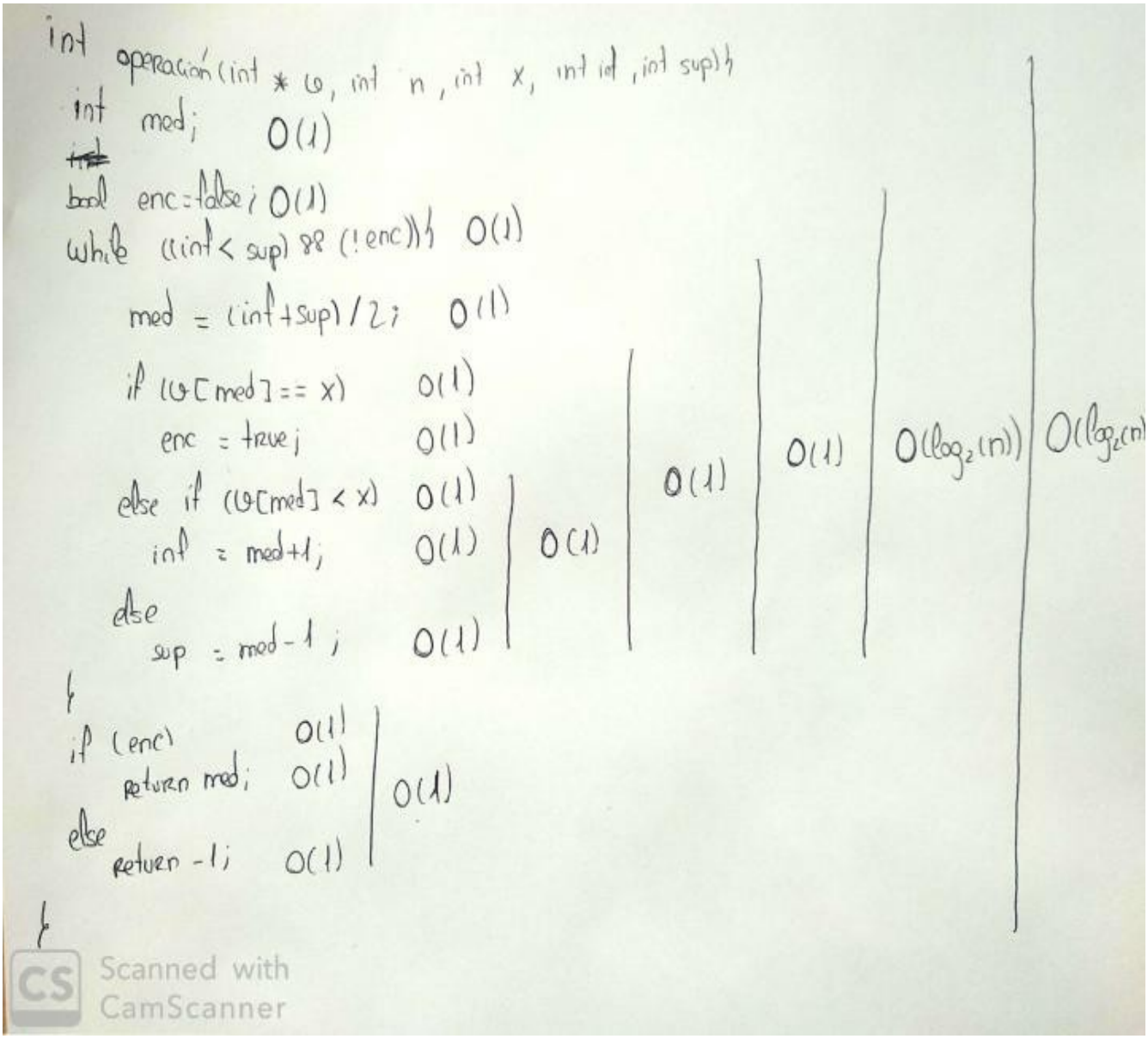
Dificultad:3

Ejercicio 3:

Explicación del algoritmo:

Se trata de un algoritmo de búsqueda, en el cual para que funcione correctamente, nuestros elementos del array tienen que estar previamente ordenados. Mientras que el supremo sea mayor que el ínfimo y no se haya encontrado nuestro dato, se va comprobando si el valor medio del array es el que queremos buscar. Si es nuestro dato, devuelve la posición en la que se encuentra nuestro dato a buscar. Si no encuentra el dato hay dos opciones: si el dato a buscar es menor que el dato que se encuentra en la posición media, igualamos el ínfimo a la posición media + 1; si es es mayor o igual, igualamos el supremo a media – 1. Después de estos dos posibles casos se vuelve a repetir el algoritmo.

Cálculo de la eficiencia teórica:



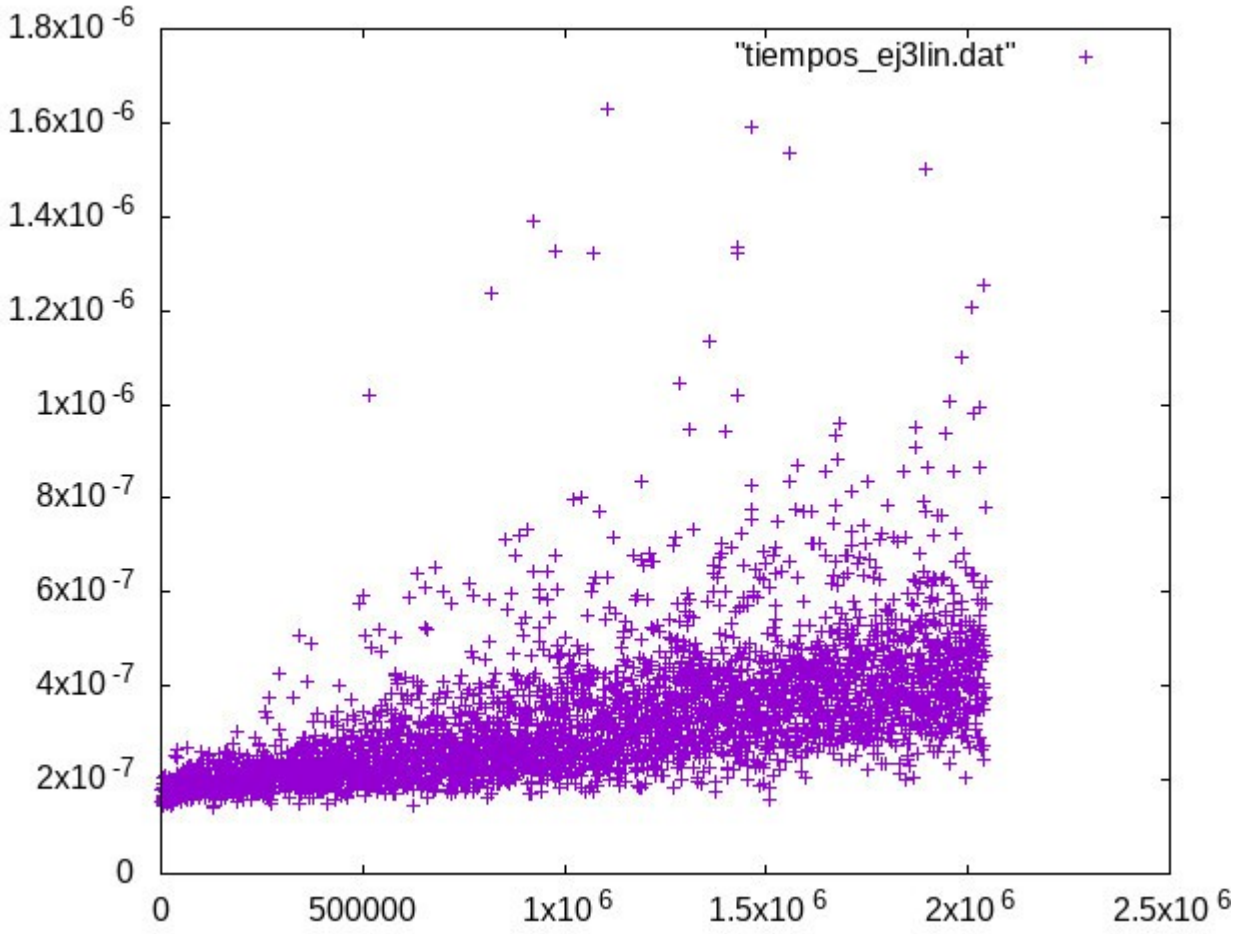
Para el calculo de la eficiencia teórica hemos estudiado el código de la función línea a línea, como se ve en la imagen. Poniendo especial atención en el código del bucle while hemos utilizado la expresion O(condicion)+O(iteraciones)*(O(cuerpo)+O(condicion)). Como se ve las lineas del cuerpo son todas de eficiencia O(1) (hemos ajustado según lo explicado en teoría, es decir, O(7) lo tratamos como O(1) y así con los distintos tipos de eficiencia), con lo que nos queda conocer las iteraciones del bucle. Como el bucle lo que hace es ir dividiendo el array en segmentos de la forma que el nuevo segmento es de mitad longitud que el anterior, esto nos queda que en el peor de los casos podemos dividir el segmento tantas veces como exponente tenga la potencia de dos inmediatamente inferior, o igual, que la longitud del segmento inicial. Es decir, que la eficiencia es O(log2(n)), con esto conocemos la eficiencia del bucle while. Para los if else tomamos siempre el de peor eficiencia pero como en este caso tienen la

misma es O(1). Finalmente sumamos la eficiencia del bucle while, los if else y el resto de lineas y lo ajustamos teniendo que la eficiencia es O(log2(n)).

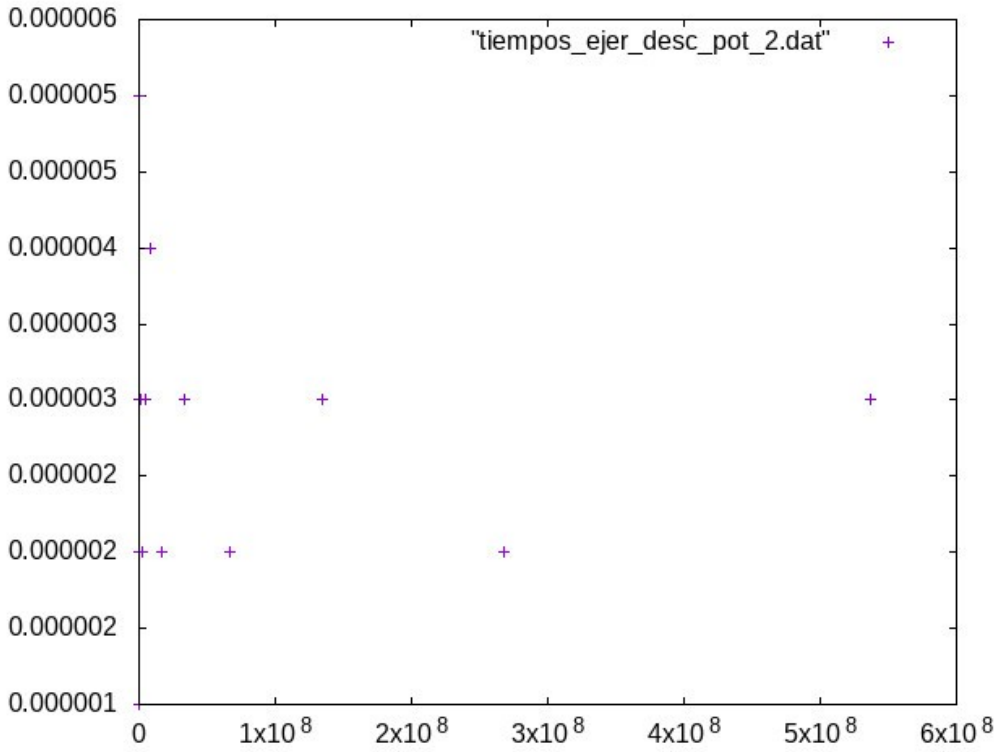
Cálculo de la eficiencia empírica:

Tenemos a nuestra disposición dos programas que utilizan el algoritmo anteriormente comentado: ejercicio3.cpp y ejercicio_desc.cpp. El primero utiliza la librería chrono que nos sirve para medir el tiempo y el segundo utiliza la librería ctime para el mismo cometido.

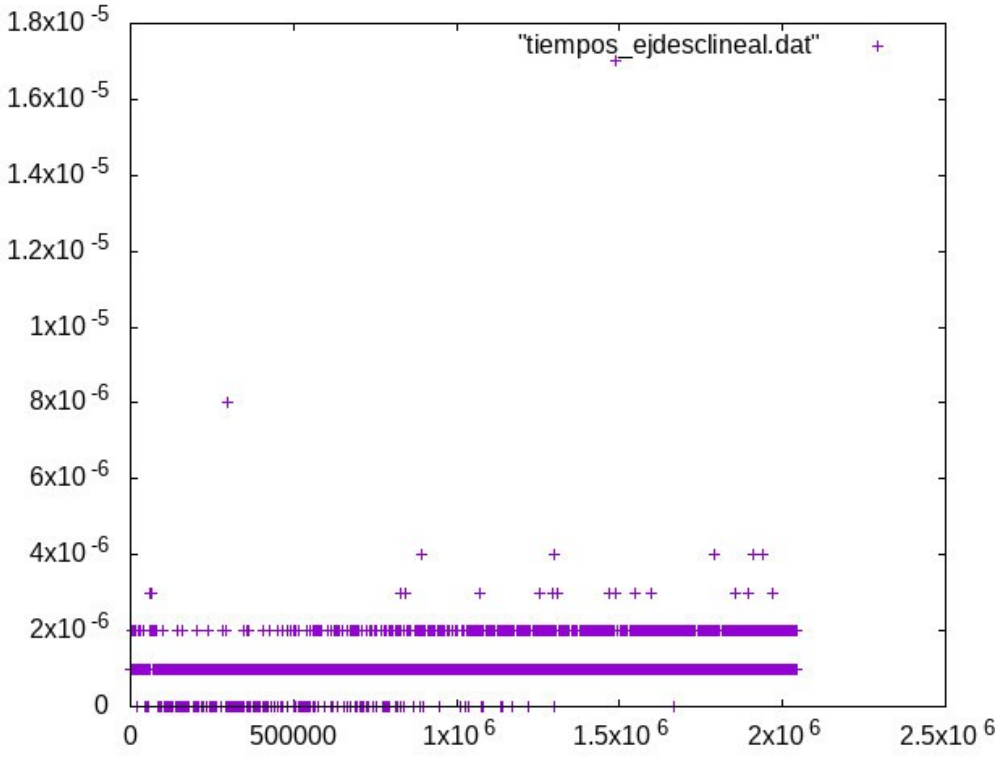
El guion nos dice que los incrementos para comprobar la eficiencia empírica tienen que ser potencias de dos y para ver su necesidad, vamos a comparar los programas ejecutados con un script con incrementos de potencias de 2 y otro con incrementos lineales;



2. Figura: ejercicio3.cpp con incremento lineal



2. Figura: ejercicio_desc.cpp con incrementos potenciales

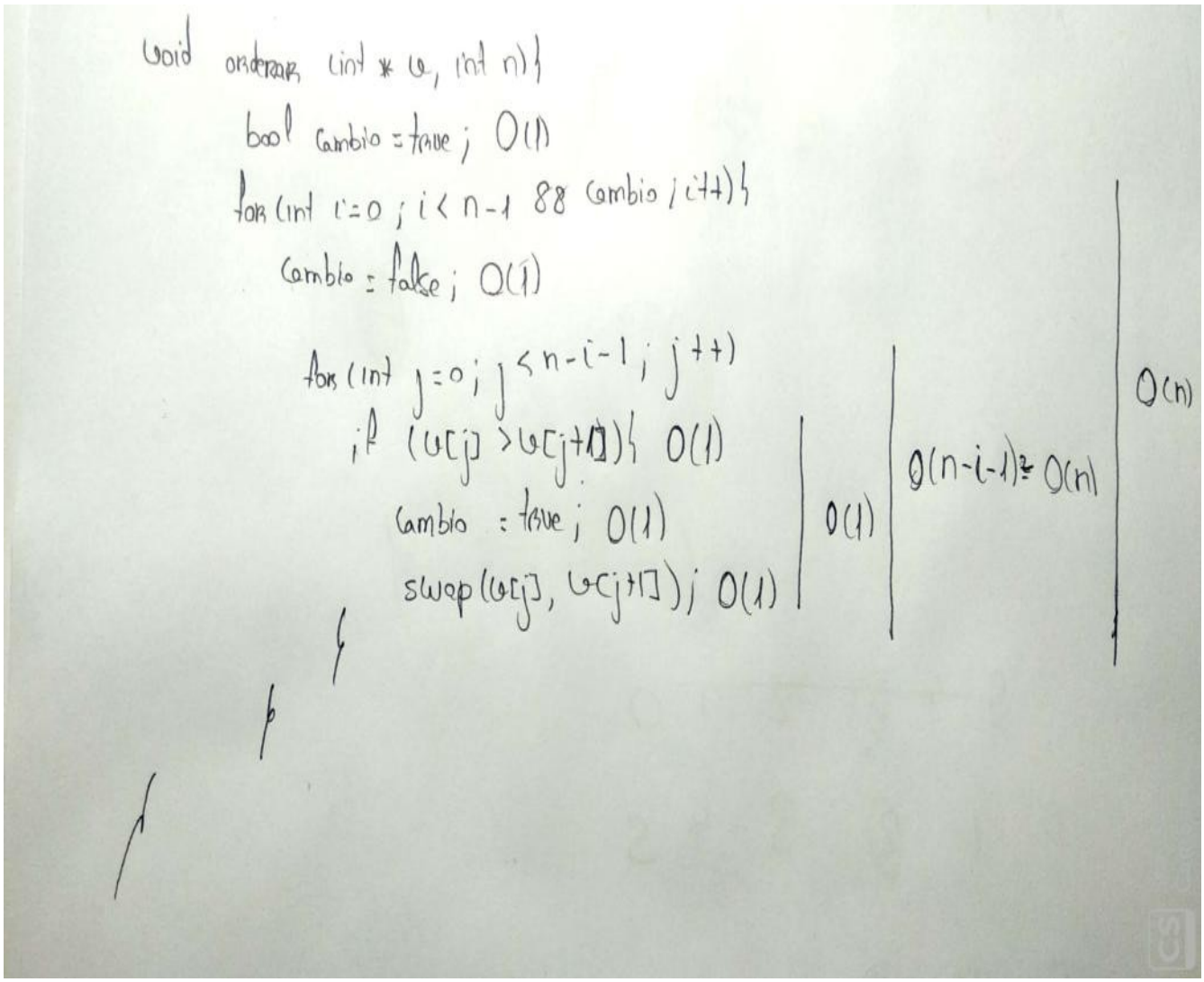


3. Figura: ejercicio_desc.cpp con incremento lineal

Vemos que si realizamos un incremento lineal, la gráfica no sale como debería y esto se debe a que

a partir de número muy altos O(log2(n)) tiene parte entera igual para numeros con tamaños muy distintos. Por ejemplo, si trabajamos entre 2^20 y 2^21 veremos que la parte entera de los tamaños comprendidos entre 1.048.576 y 2.097.152 son la misma cambiando solo un poco los decimales. Luego si trabajamos con un incremento lineal del tamaños del problema distintos tamaños pueden tener un tiempo de ejecución muy parecido, como se muestra en la imagen.
Dificultad:7 (en el enunciado no se explica que hacer respecto a que el vector no esté ordenado).

Ejercicio 4



Para calcular la eficiencia teórica de este algoritmo en el mejor caso, procedemos a realizar los siguientes razonamientos:

Obteniendo que el algoritmo sigue una eficiencia de orden O(n).

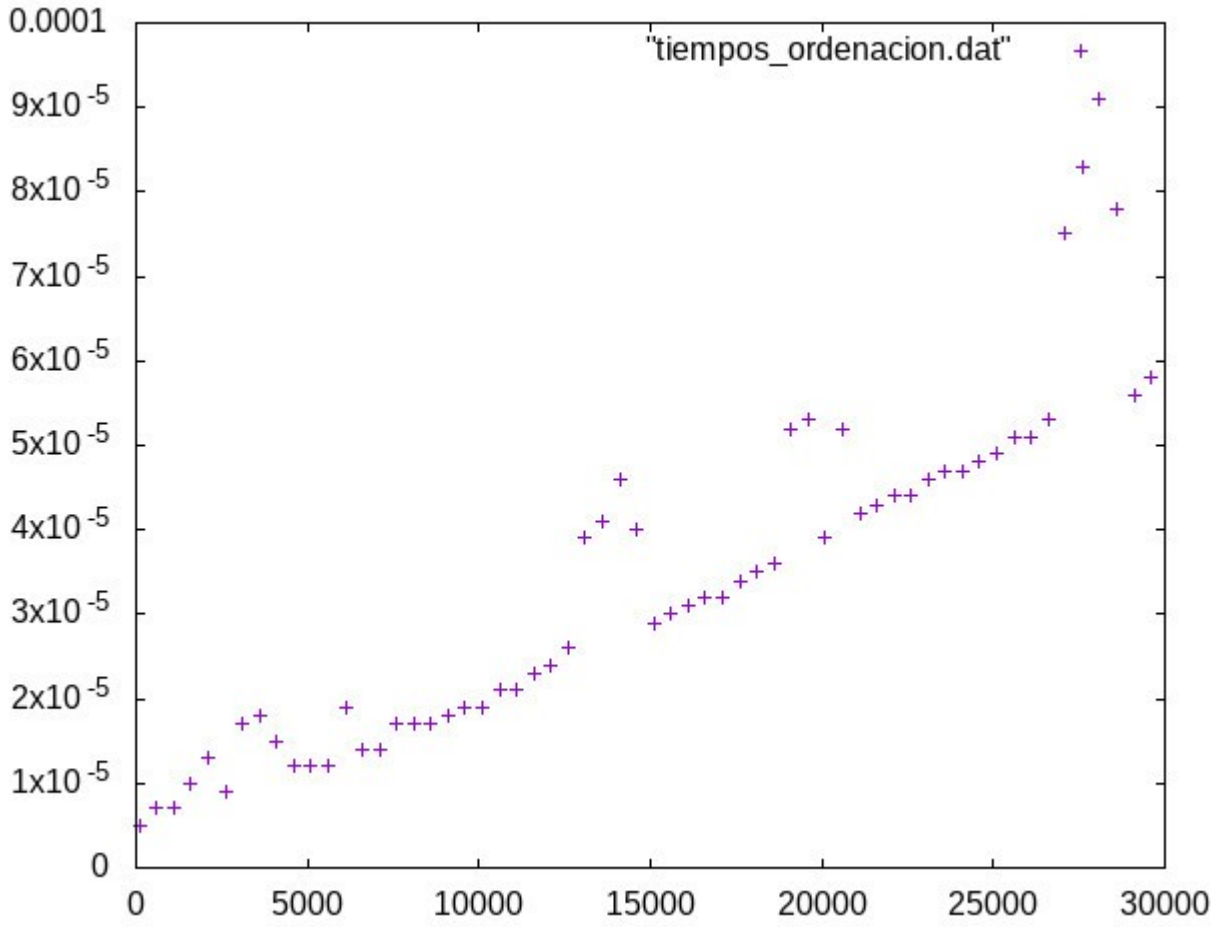
Al tratarse del mejor caso el primer bucle de los dos anidados realiza una única iteración. El segundo bucle realizará N-1 iteraciones. Como se tratan de operaciones simples como asignación, comparación e incremento estas tienen eficiencia O(1) (la función swap también son asignaciones asique tiene eficiencia O(1)). Llegamos a que O(for)=O(asignacion)+O(condicion)+O(iteraciones)*(O(cuerpo) +O(condicion)+O(incremento)), con esta expresión tenemos que el segundo bucle tiene eficiencia O(N-

1) que ajustado es O(N), luego como el primer bucle tiene O(1) para la eficiencia de las iteraciones, la condición y el incremento, y O(N) para el cuerpo aplicando la expresión llegamos a que los dos bucles anidados tiene eficiencia O(N), finalmente le sumamos la eficiencia de la primera asignación en la línea 2 y los ajustamos llegando a una eficiencia total del código de O(N).

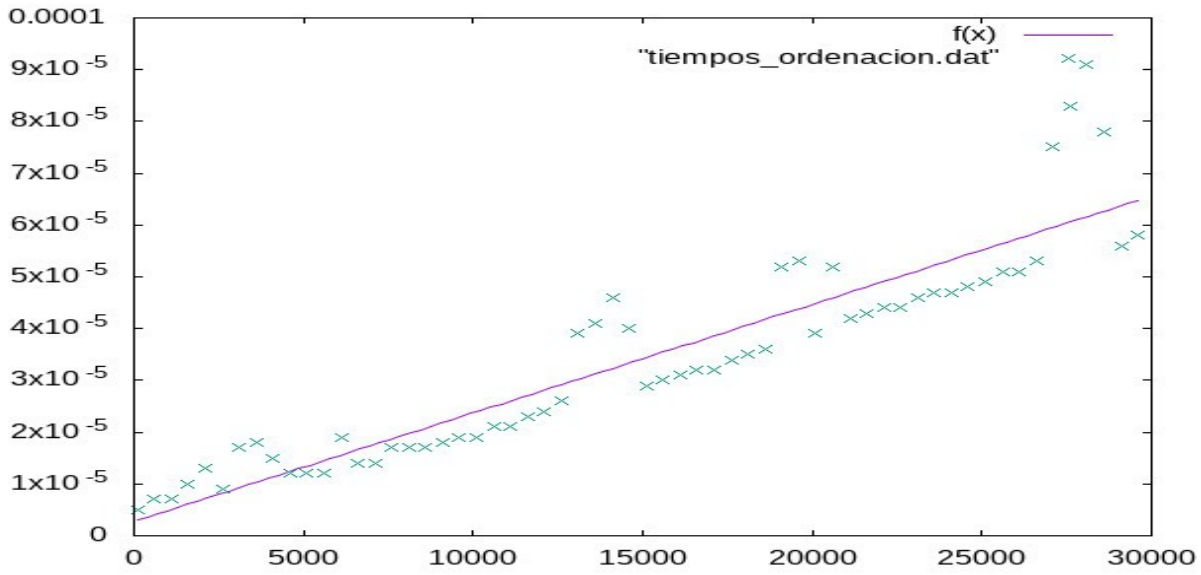
Procedemos a calcular la eficiencia empírica. Para ello, hemos ejecutado el siguiente código:

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

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;
swap (v[j],v[j+1]); //incluir algorithm
}
}
}



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


  // Generación del vector aleatorio
  int *v=new int[tam];       // Reserva de memoria

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

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


  ordenar(v,tam); // de esta forma forzamos el peor caso

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

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

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

Hemos aprovechado el script del ejercicio 1, para así obtener un archivo .dat que nos indica el tamaño del vector utilizado y el tiempo que tarda en ejecutarse el algoritmo con ese tamaño. Esto nos sirve para realizar una gráfica y así poder ver la eficiencia empírica.
La gráfica es la siguiente:



Comparamos con nuestra eficiencia teórica. Para ello, ajustamos nuestros datos empíricos a una gráfica de tipo f(x) = ax + b. Así, obtenemos la siguiente comparación:



Dificultad:4

Ejercicio 5

El algoritmo a comprobar es el algoritmo de la burbuja dado en el ejercicio 1 pero en este caso vamos a comprobar el mejor caso y el peor caso.

Para calcular el mejor caso, hemos hecho que los vectores se generen de manera ordenada. Para ello hemos asignado a cada posición del vector el valor del iterador en cada momento. El código quedaría así:

```
#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]) {
swap(v[j],v[j+1]);
//alternatva al swap
/*int aux = v[j];//incluir algorithm
v[j] = v[j+1];
v[j+1] = aux;*/
}
}


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


  // Generación del vector aleatorio
  int *v=new int[tam];       // Reserva de memoria

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

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


  ordenar(v,tam); // de esta forma forzamos el peor caso

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

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

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

Para calcular el peor caso, hemos hecho que el vector se genere con enteros ordenados de manera inversa. El código resultante es el siguiente:

```
#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]) {
                                swap(v[j],v[j+1]);
                        }
}


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


  // Generación del vector aleatorio
  int *v=new int[tam];       // Reserva de memoria

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

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


  ordenar(v,tam); // de esta forma forzamos el peor caso

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

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

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

Para ejecutar nuestro código aprovechamos el script del ejercicio 1, obteniendo así un archivo .dat para cada código cuyo contenido es el tiempo que tarda en ejecutarse el algoritmo con vectores de distintos tamaños. Estos archivos son utilizados para crear unas gráficas y así ver su rendimiento empírico. Las gráficas son las siguientes:

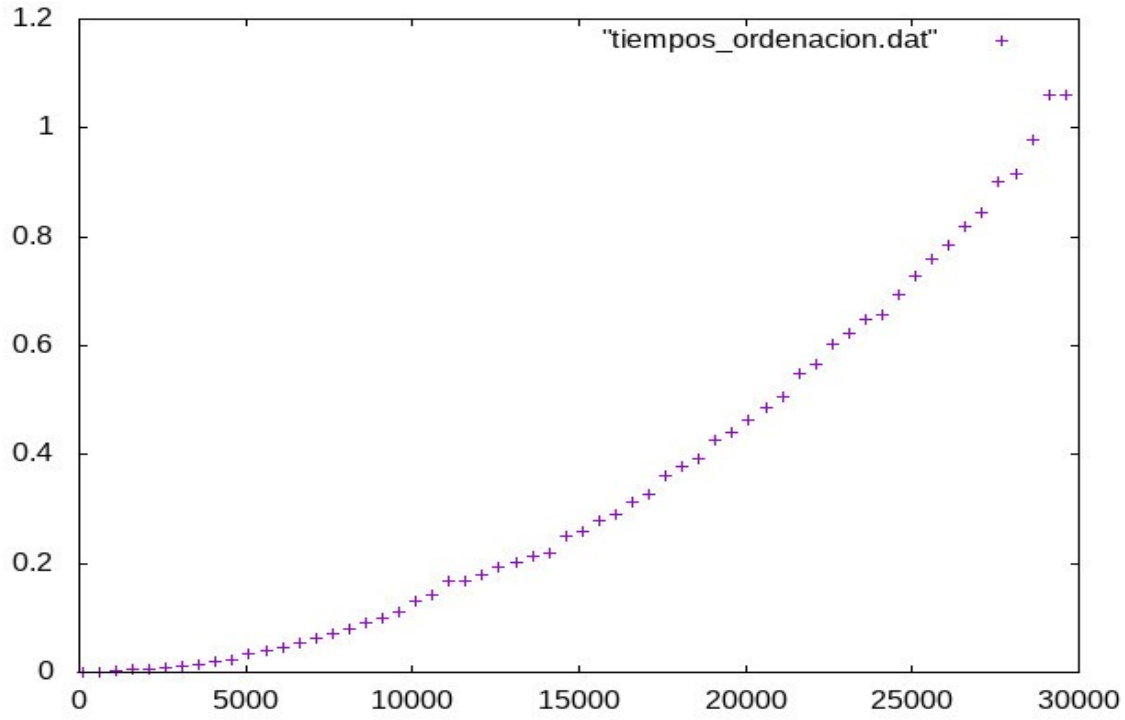
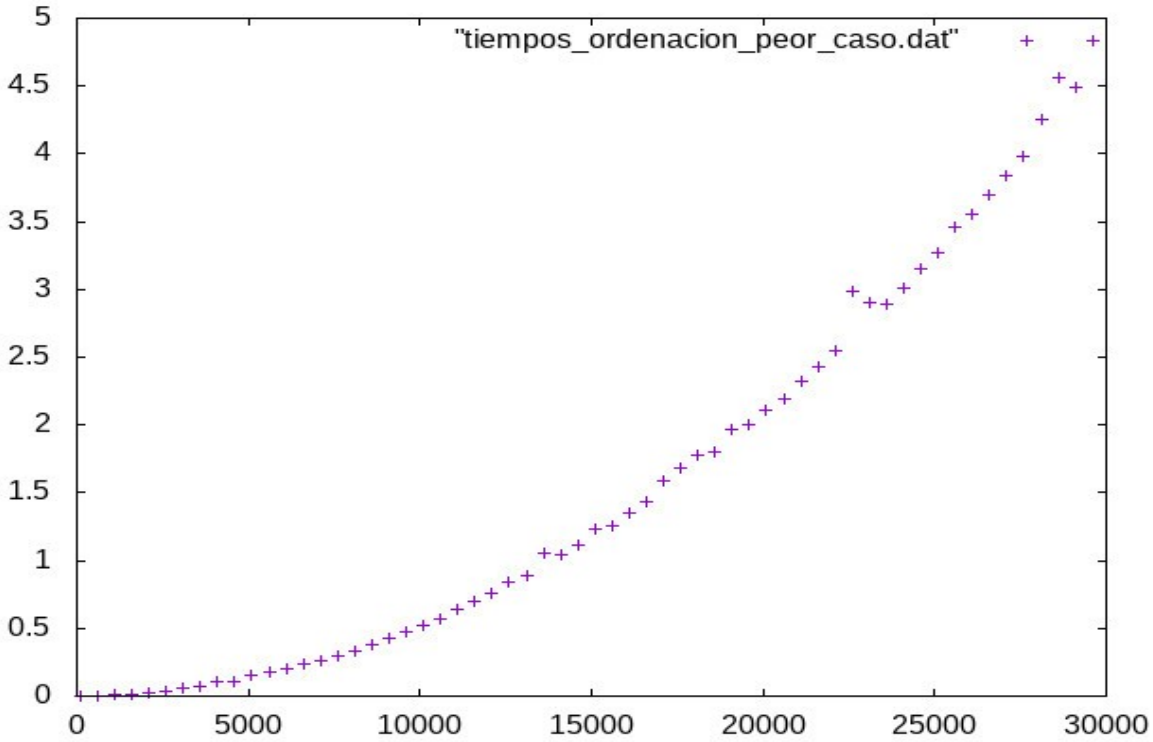
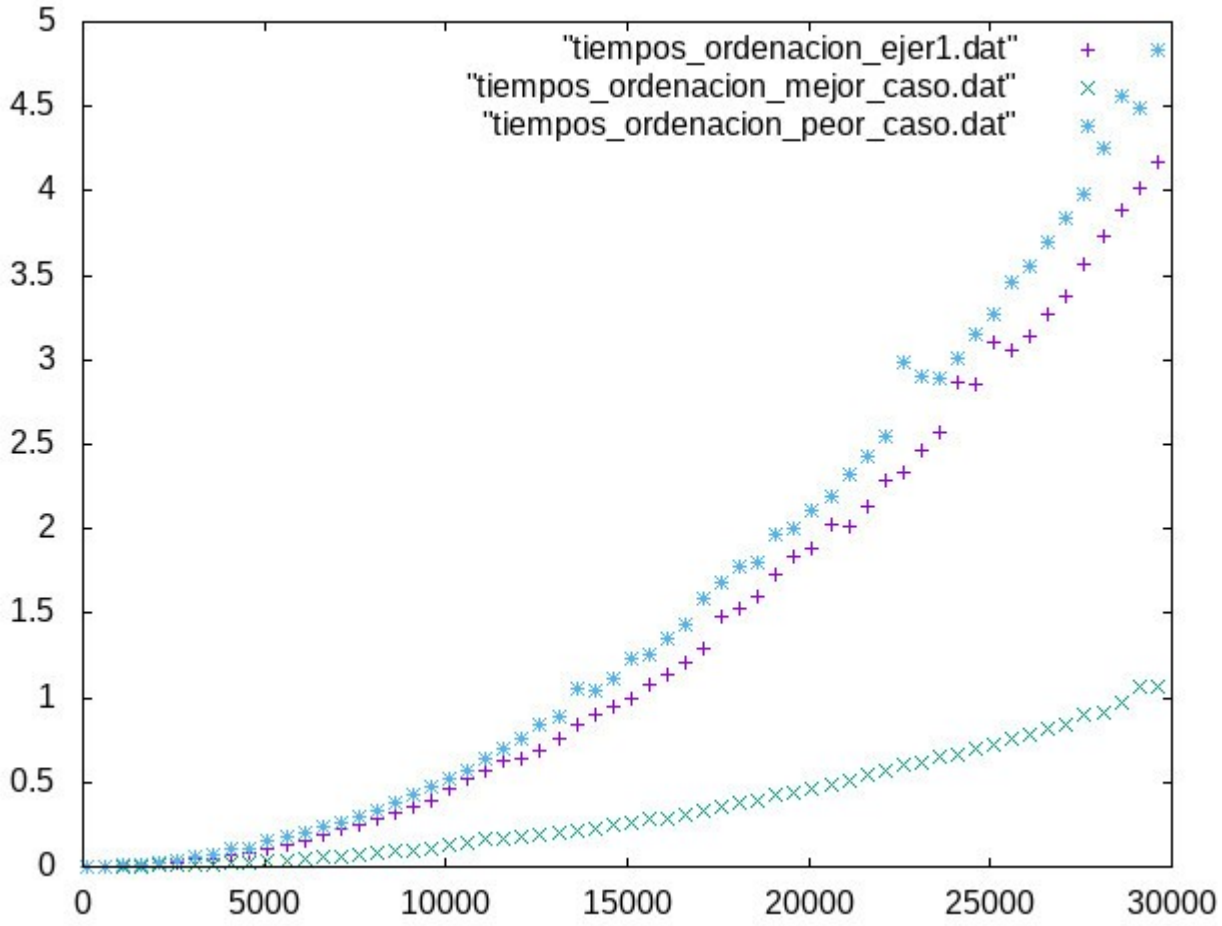


Figura 4: MEJOR CASO



5. Figura: Peor caso

Ahora procedemos a realizar una gráfica que muestra estas dos últimas gráficas comparadas con la gráfica del ejercicio 1:

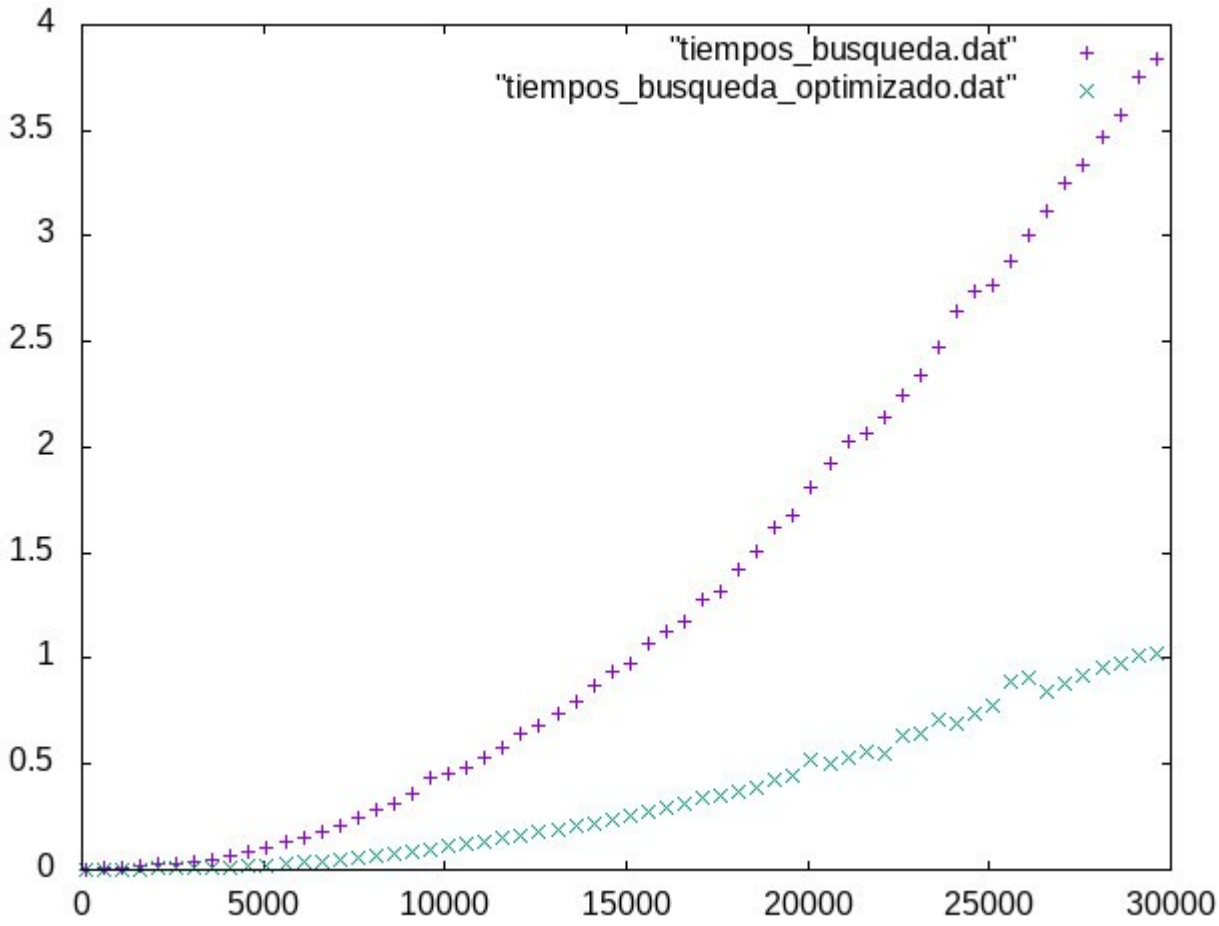


Ejercicio 6

Nos pide retomar el algoritmo de la burbuja pero en este caso nos pide compilar el programa de tal forma que optimice el código. Para ello, ejecutamos el comando siguiente en nuestra terminal:

```
g++ -O3 ordenacion.cpp -o ordenacion_optimizado
```

Procedemos a comparar las curvas de eficiencia para así poder ver gráficamente la mejora en rendimiento del programa:



6. Figura: Programa normal vs optimizado

Dificultad:2

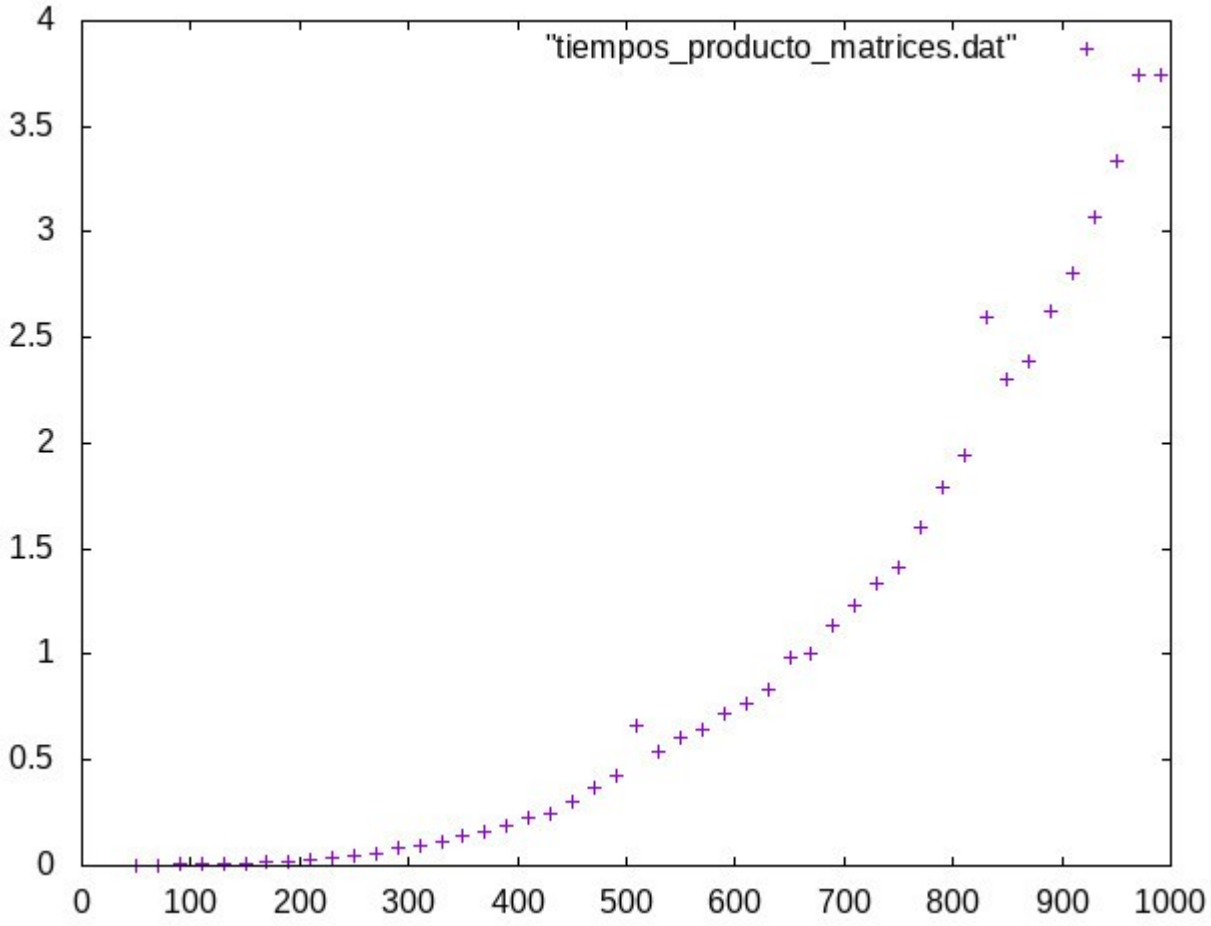
Ejercicio 7

Para poder realizar este ejercicio, hemos implementado un programa en el que se pide el número de filas y columnas de dos matrices de tal forma que se puedan multiplicar. El programa rellena las matrices de forma aleatoria con un determinado umbral (el cual hemos decidido que sea 200). El programa es el siguiente:

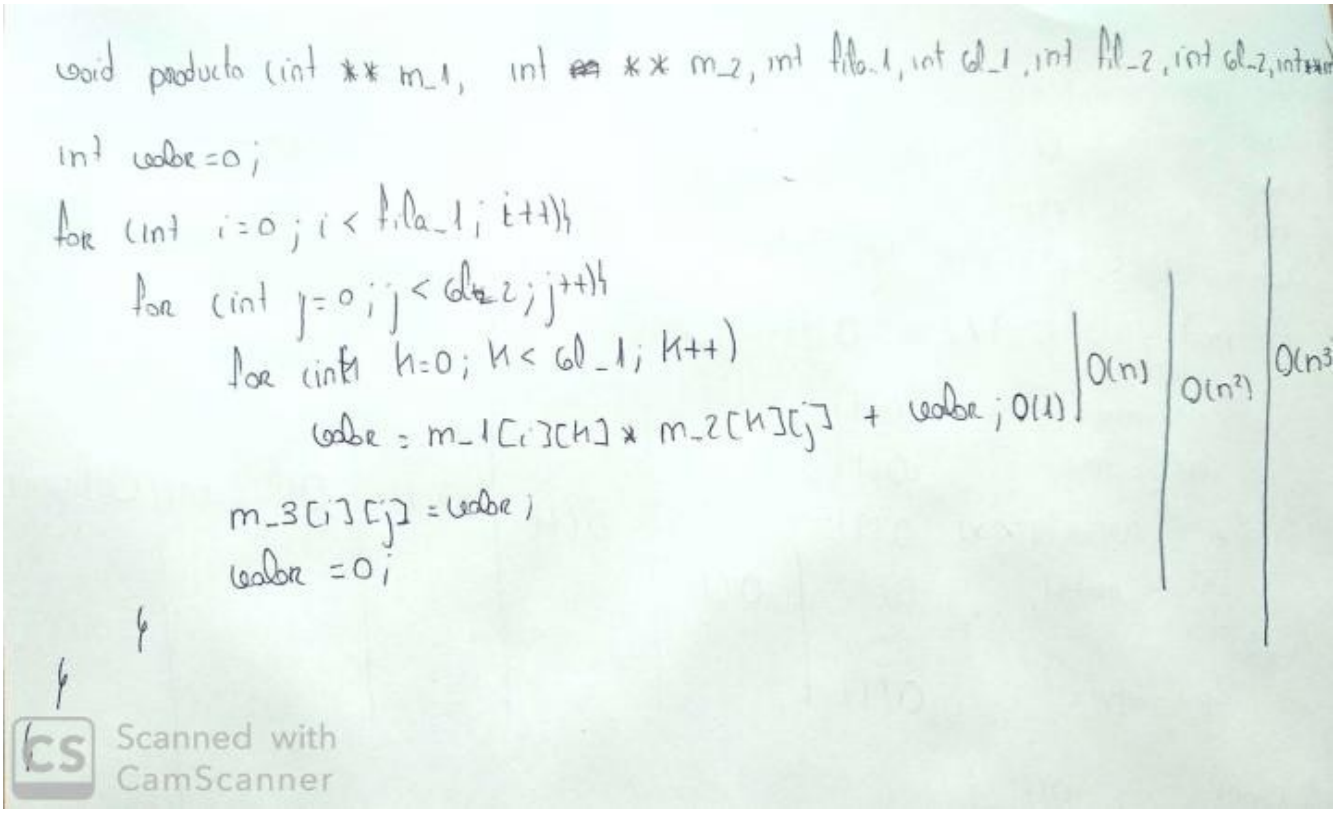
```cpp
#include <iostream>
#include <ctime>
#include <cstdlib>
#include <chrono>

using namespace std;
using namespace std::chrono;

void sintaxis() {
 cerr << "Sintaxis:" << endl;
 cerr << " FIL1:numero de filas de la primera matriz" << endl;
 cerr << " COL1:numero de columnas de la primera matriz" << endl;
 cerr << " FIL2:numero de filas de la segunda matriz" << endl;
 cerr << " COL2:numero de columnas de la segunda matriz" << endl;
 cerr << " VMAX: Valor máximo (>0)" << endl;
 cerr << " Genera dos matrices de dimension FIL1xCOL1 y FIL2xCOL2" << endl;
 exit(EXIT_FAILURE);
}

void producto(int **m_1, int **m_2,int filas_1,int columna_1, int filas_2, int columna_2,int **m_3){

int valor=0;

for(int i=0 ; i < filas_1;i++){
        for(int j=0; j < columna_2; j++){
                for(int k = 0; k < columna_1; k++)
                        valor = m_1[i][k]*m_2[k][j] + valor;

                m_3[i][j] = valor;
                valor = 0;
        }
}
}

int main (int argc, char *argv[]){
if(argc!=6)
sintaxis();
int fil_1=atoi(argv[1]);
int col_1=atoi(argv[2]);
int fil_2=atoi(argv[3]);
int col_2=atoi(argv[4]);
int vmax=atoi(argv[5]);

if(vmax<=0 || col_1!=fil_2)
sintaxis();

int **matriz_1=new int *[fil_1];
int **matriz_2=new int *[fil_2];
int **matriz_resultado=new int *[fil_1];

for(int i=0; i < fil_1; i++)
        matriz_1[i]=new int[col_1];

for(int i=0; i < fil_2;i++)
         for(int j=0; j < col_2; j++)
                 matriz_2[i][j]= rand%vmax();

high_resolution_clock::time_point start,
end;
duration<double> tiempo_transcurrido;
start = high_resolution_clock::now();
producto(matriz_1, matriz_2, fil_1, col_1, fil_2,col_2,matriz_resultado);
end=high_resolution_clock::now();
tiempo_transcurrido= duration_cast<duration<double> >(end - start);

for(int i=0; i < fil_1; i++){
         delete [] matriz_1[i];
         delete [] matriz_resultado[i];
}

for(int i=0; i < fil_2; i++)
         delete [] matriz_2[i];

cout << fil_1 << "\t" << tiempo_transcurrido.count() << endl;

delete [] matriz_1;
delete [] matriz_2;
delete [] matriz_resultado;
}
```

Para su ejecución, hemos hecho un script el cual ejecutaba el programa poniendo datos el tamaño de las filas y columnas de las dos matrices. Hemos elegido que esas matrices sean cuadradas. Se empiezan multiplicando matrices de tamaño 50x50 hasta llegar a matrices de tamaño 1000x1000, incrementando en 20 el tamaño en cada ejecución.

Como resultado, obtenemos esta gráfica:

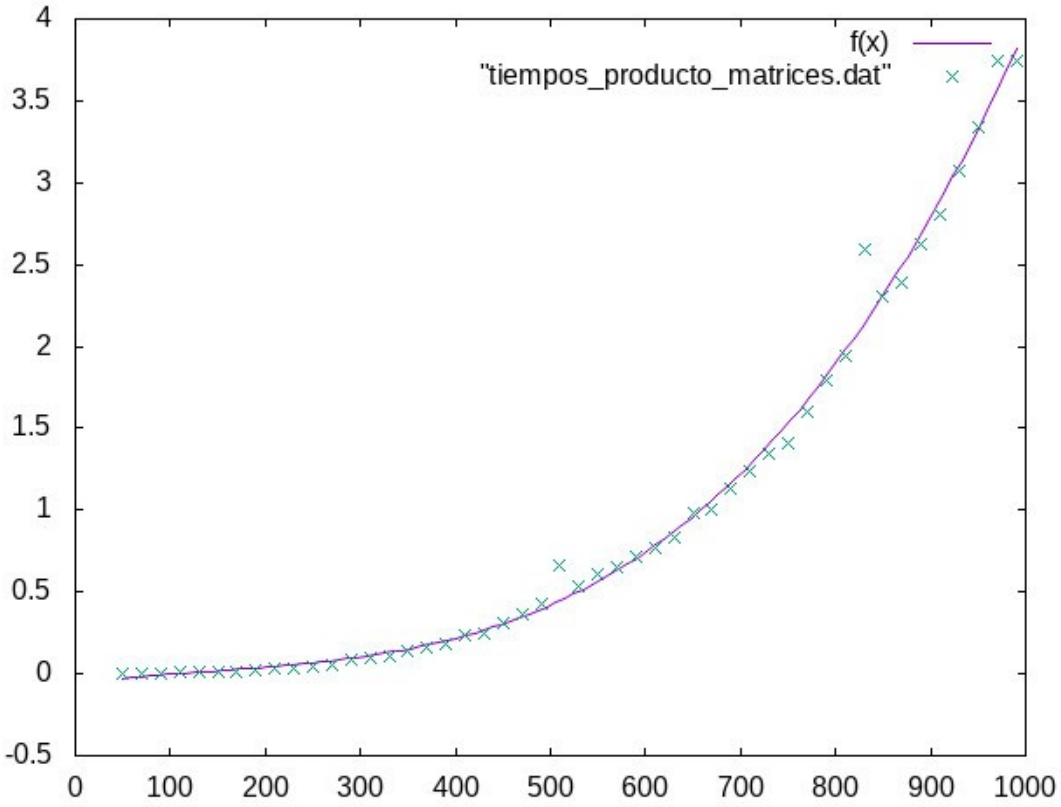


Posteriormente, hemos procedido a calcular la eficiencia teórica, dándonos como resultado que nuestro algoritmo es $O(n^3)$.



Ahora, para así verlo de manera mas clara, hemos ajustado nuestra gráfica correspondiente a la eficiencia teórica a una curva de tipo $f(x) = ax^3 + bx^2 + cx + d$, dándonos como resultado esta imagen:

Como se puede ver en el código la función utiliza tres bucle for anidados, que siempre recorren todas su posiciones. Para estudiar la eficiencia teórica suponemos que hacemos el producto de matrices cuadradas NxN. Las asignaciones, productos, sumas y acceso a índices tienen eficiencia O(1), asi como las condiciones y el incremento de los bucles for. Estudiaremos desde el bucle for más bajo al más alto usando la expresión O(for)=O(asignacion)+O(condicion)+O(iteraciones)*(O(cuerpo)+O(condicion) +O(incremento)). Ya sabemos que la condicion, la asignación y el incremento tiene eficiencia O(1), y si nos fijamos en el cuerpo se trata de eficiencia O(1) (está ajustado), luego como recorre todas las posiciones tendra N iteraciones y por tanto eficiencia O(N), este primer bucles está contenido en otra que al igual que este tiene todo eficiencia O(1), menos las iteraciones y el cuerpo que tienen eficiencia O(N) luego por la expresión de arriba la eficiencia de este segundo bucle es $O(N^2)$. Por último, los dos bucles anteriores están contenidos en un tercero, el cual tiene condicion, incremento y asignación O(1), tiene N iteraciones y su cuerpo tiene eficiencia $O(N^2)$ luego por la expresión de arriba se trata de una eficiencia $O(N^3)$. Como en el resto del código sólo hay una asignación, llegamos a que la eficiencia final es la suma de ambas que ajustada sería $O(N^3)$.(Imagen del proceso arriba).



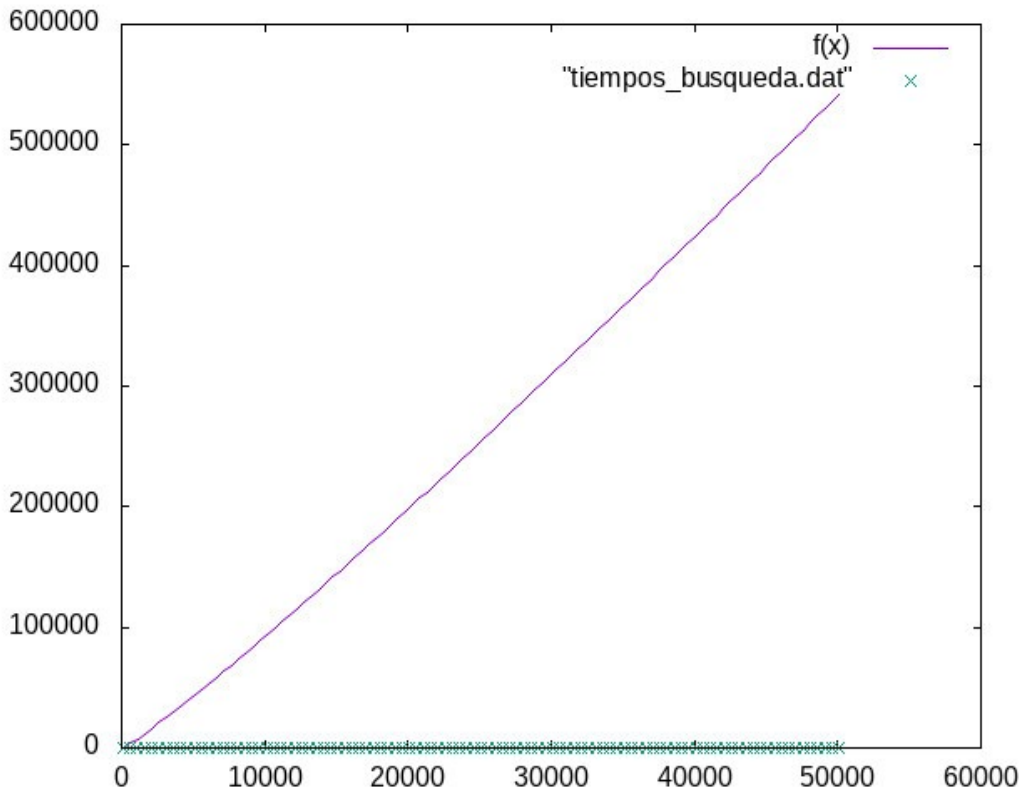
Dificultad:8

Ejercicio 8

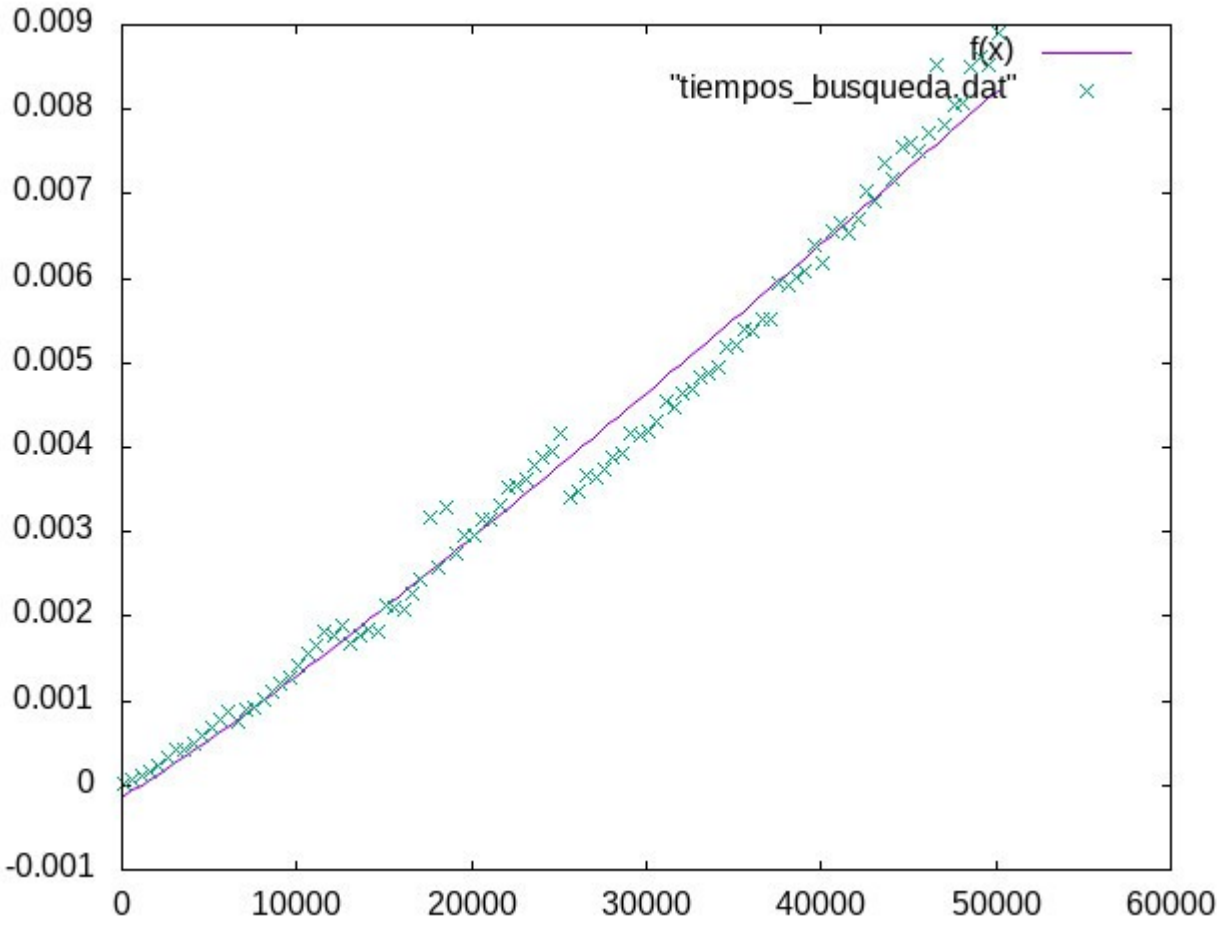
Para la realización de este ejercicio, hemos compilado el programa mergesort.cpp con el script ejecuciones_blineal.csh modificado de la siguiente manera:

```
#!/bin/csh
@ inicio = 100
@ fin = 10100
@ incremento = 100
set ejecutable = mergesort_modificado
set salida = tiempos_busqueda_modificado.dat

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

Sabemos que la eficiencia teórica de mergesort es O(n*log(n)). Procedemos a realizar una gráfica de su eficiencia empírica y comparamos con un ajuste de su eficiencia teórica.



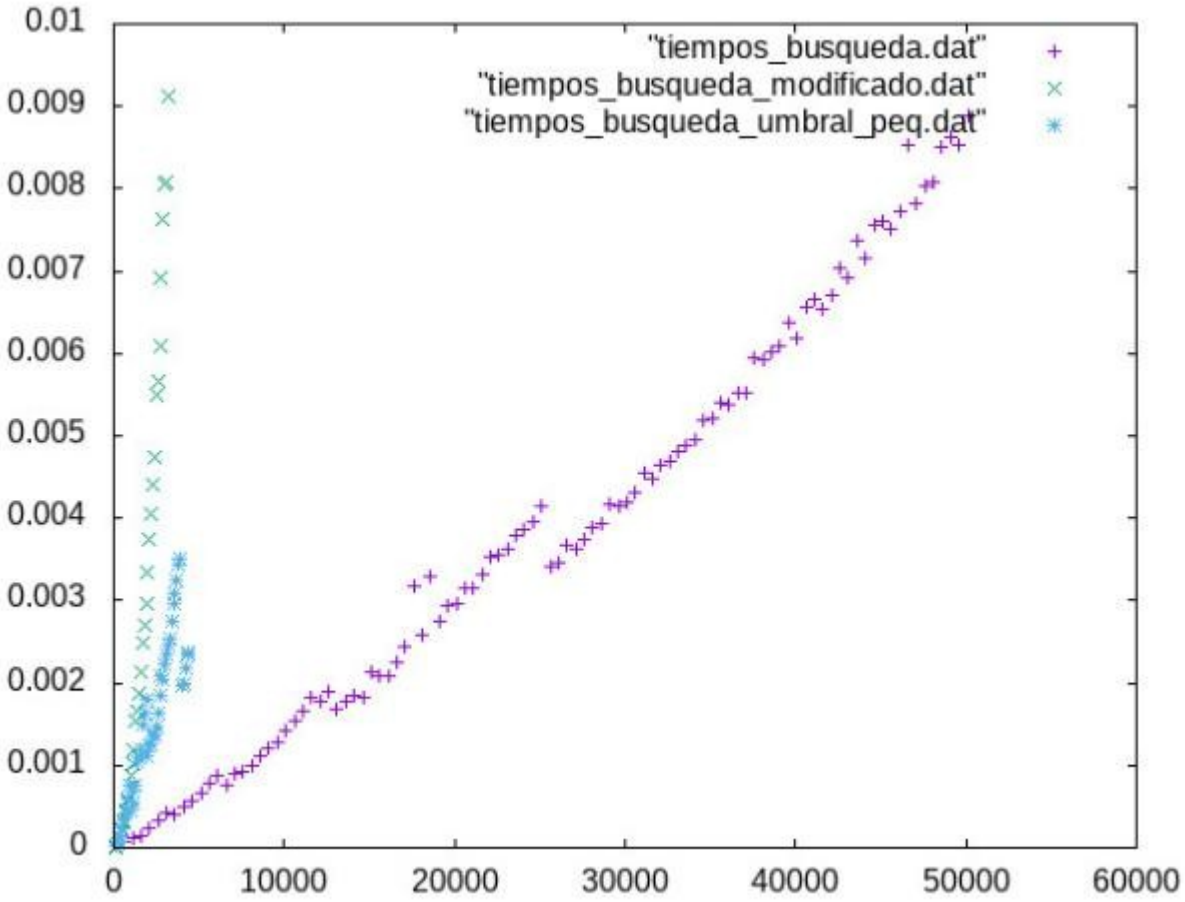
*7. Figura: n*log(n) vs mergesort*



*8. Figura: Regresión tipo n*log(n) vs mergesot*

Vemos como mergesot se ajusta bastante bien a una funcion tipo n*log(n)
En el programa entra en juego una variable denominada UMBRAL_MS la cual hace que nuestro programa utilice antes o después el algoritmo de insercción.

Para ver como varía el programa para distintos valores de UMBRAL_MS, hemos obtenido diferentes archivos de tiempo para distintos valores de UMBRAL_MS, compaŕandolos en una gráfica. Hemos utilizado tres valores distintos, los cuales son: 100, 1000 y 10000. La gráfica comparativa es la siguiente:



9. Figura: tiempos_busqueda.dat --> UMBRAL_MS = 100 ; tiempos_busqueda_modificado.dat --> UMBRAL_MS = 10000 ; tiempos_busqueda_umbral_peq.dat --> UMBRAL_MS = 1000

Podemos ver claramente cómo empeora la eficiencia del programa conforme va aumentando el valor de UMBRAL_MS. Esto es debido a que si aumentamos el valor de UMBRA_MS, hacemos que nuestro programa utilice de manera “mas directa” el algoritmo de inserción, cuya eficiencia teórica es $O(n^2)$, y así conseguimos que nuestro programa tome esta tendencia.

Dificultad: 7