ESPECIFICACIONES TÉCNICAS:

CPU: Intel(R) Core(TM) i5-5200U CPU @ 2.20GHz
RAM: 4GB
SO: Ubuntu 14.04
Compilador: g++

EJERCICIO 1 y 2:

Para resolver este ejercicio he creado un archivo ordenacion.cpp y un script ejecuciones_ordenacion.sh para ejecutar varias veces el ejecutable. Lo primero que he realizado es modificar el código original para incluir el algoritmo de ordenación en vez de realizar el de búsqueda, luego lo he compilado con:

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

También he modificado el script que nos proporciona la practica para que se ejecute 30000 veces con tamaños cada vez 100 veces más mayor que el anterior, es decir, de 100 en 100 desde 100 hasta 30000. ADJUNTO AMBOS ARCHIVOS.

Lo primero que voy a realizar es calcular la eficiencia teoria del 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;
                  }
}
```

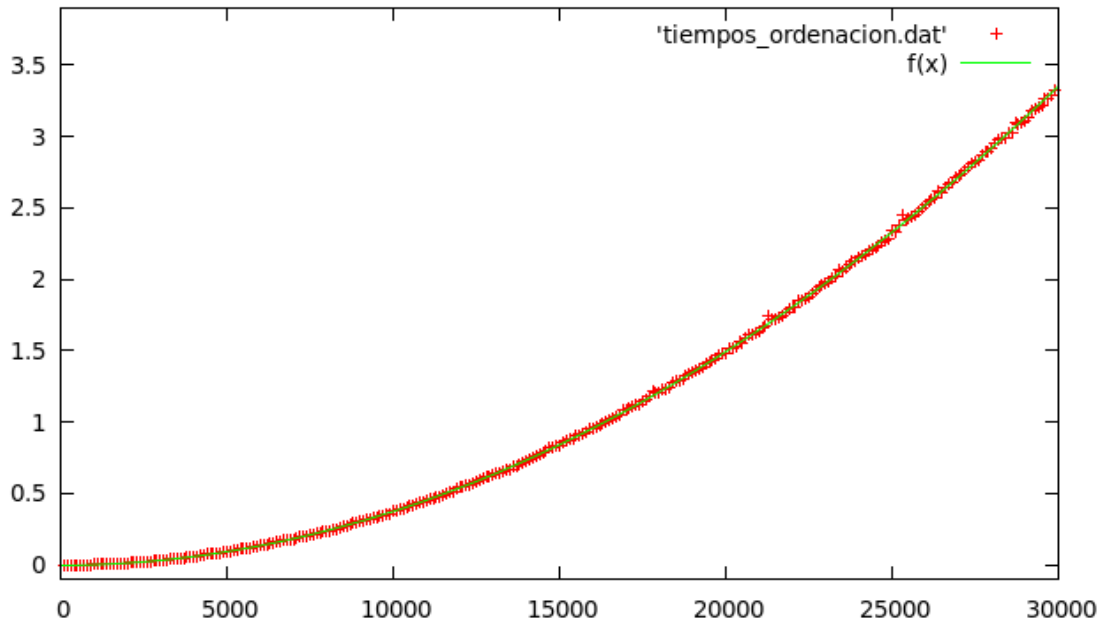
El orden de este algoritmo es $O(n^2)$ ya que en el peor de los casos se ejecutaria el primer bucle n veces, luego el orden de este es n^2. Ahora ejecutamos el script que nos va a proporcionar los datos necesarios para dibujar su función, para ellos ejecutamos el script con ./ejecuciones_ordenacion.sh

Ahora como sabemos que nuestra función se parece a $f(x)=ax^2+bx+c$ pues tenemos que buscar que valores de a, b y c se parecen más a nuestra función dada por los datos recopilados, para ellos realizamos el siguiente procedimiento para encontrar a,b y c:

1. f(x)=a*x**2+b*x+c
2. fit f(x) “tiempos_ordenacion.dat” via a,b,c
3. plot “tiempos_ordenacion.dat”, f(x)

Con el punto 3 vamos a ver representadas la función que nos proporciona los datos recopilados y la función o curva teórica. Dejo el procedimiento en capturas.

```
gnuplot> f(x)=a*x**2+b*x+c
gnuplot> fit f(x) 'tiempos_ordenacion.dat' via a,b,c


 Iteration 0
 WSSR        : 0.0232574         delta(WSSR)/WSSR   : 0
 delta(WSSR) : 0                 limit for stopping : 1e-05
 lambda    : 2.31798e+08

initial set of free parameter values

a               = 3.70067e-09
b               = 8.18447e-07
c               = -0.0028975
************
The maximum lambda = 1.000000e+20 was exceeded. Fit stopped.
final sum of squares of residuals : 0.0232574
rel. change during last iteration : 0

degrees of freedom    (FIT_NDF)                        : 296
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 0.0088641
variance of residuals (reduced chisquare) = WSSR/ndf   : 7.85724e-05

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

a               = 3.70067e-09      +/- 7.693e-12    (0.2079%)
b               = 8.18447e-07      +/- 2.383e-07    (29.12%)
c               = -0.0028975       +/- 0.001548     (53.43%)


correlation matrix of the fit parameters:

               a      b      c
a               1.000
b              -0.968  1.000
c               0.748 -0.868  1.000
gnuplot> plot 'tiempos_ordenacion.dat', f(x)
```



DESTACAR: He realizado directamente el procedimiento con ax^2+bx+c por que al realizarlo directamente con la ordenación de n^2 no se ajustaba correctamente a la curva que nos daban los dato proporcionados al ejecutar el script, así que para ajustar al máximo esa curva hemos tenido que usar ax^2+bx+c para que se ajuste a la perfección

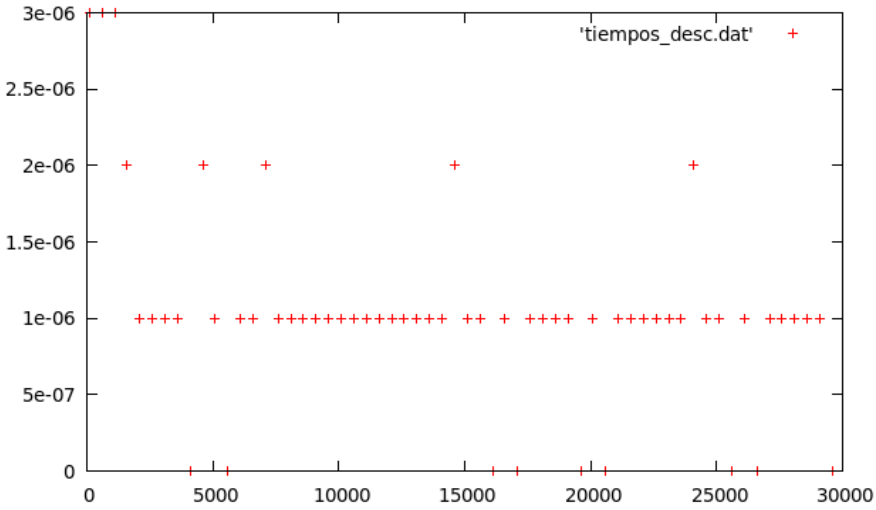
EJERCICIO 3:

El algoritmo que se representa es un algoritmo de búsqueda binaria o logarítmica, se trata de un algoritmo que compara el valor que buscamos con el elemento del centro del vector, si no son iguales, la mitad en la cual el valor no puede estar es eliminada y la búsqueda continuaría con la mitad que queda restante y así sucesivamente hasta que se encuentre el valor o que en el peor caso no se encuentre ningún valor.

Al realizar la eficiencia teórica del algoritmo nos damos cuenta que tanto la declaración de variables, el if-else final y todo lo de dentro del bucle es de O(1), luego solo nos faltaría saber que orden tiene el bucle while, no damos cuenta que tiene una operación de comparación, otra comprueba si es una variable verdadera o falsa y una operación &&, las tres operaciones son también de O(1), el problema viene cuando nos damos cuenta que en el peor de los casos el algoritmo se va a ir ejecutando tantas veces como se pueda dividir el vector por la mitad, es decir, se divide el vector por la mitad se comprueba todo lo que hemos explicado antes (funcionamiento del algoritmo) y si no se encuentra el valor se vuelve a dividir la parte que nos queda y así sucesivamente, luego esto es de orden logarítmico, concretamente tiene O(log2n). Luego finalizamos concretando que el orden de este algoritmo es O(log2n).

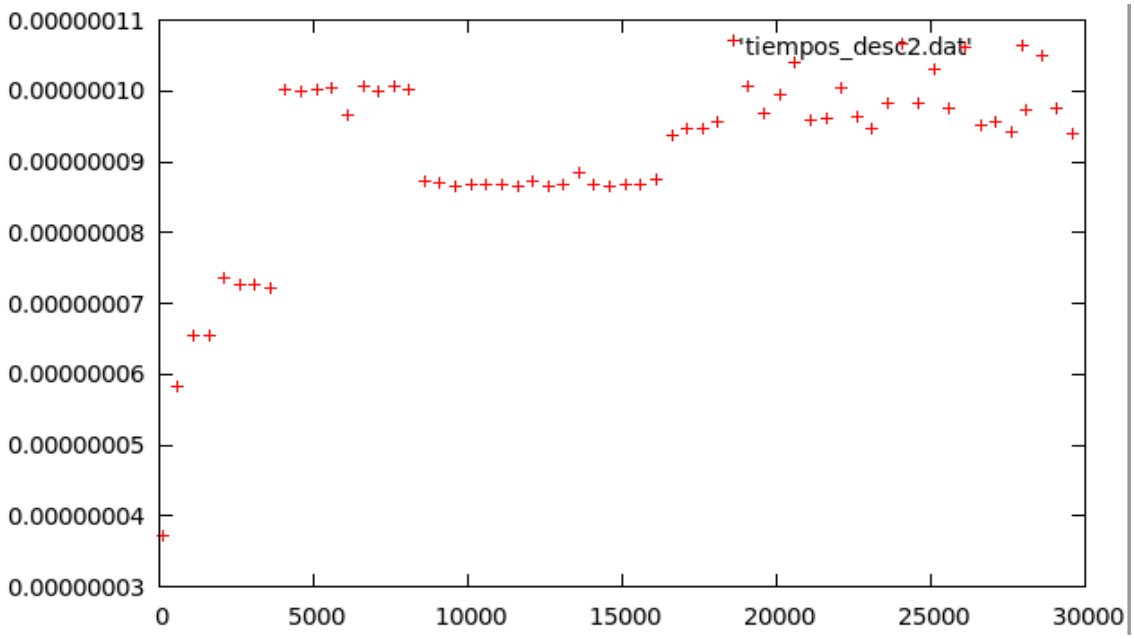
Para calcular la eficiencia empírica voy a usar el script que usé anteriormente, pero modificándolo para usar este algoritmo (ADJUNTO), al ejecutarlo nos produce la siguiente gráfica.

```
#!/bin/bash
inicio=100
fin=30000
incremento=500
ejecutable="ejercicio_desc"
salida="tiempos_desc.dat"

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



Como podemos observar, a excepción de esos valores sueltos en 2e-06, todos los valores están en una misma linea totalmente alineados, pero vemos que para valores mayores del tamaño del vector el tiempo sigue siendo igual, cerca de 0. Esto se debe a que el algoritmo es de O(log2n), entonces el crecimiento es muy lento y el tamaño del problema también lo es, luego el sistema es capaz de realizar los cálculos muy rápidamente y por lo tanto que no podamos conseguir sacar el verdadero tiempo de ejecución del algoritmo (ejercicio_desc.cpp).

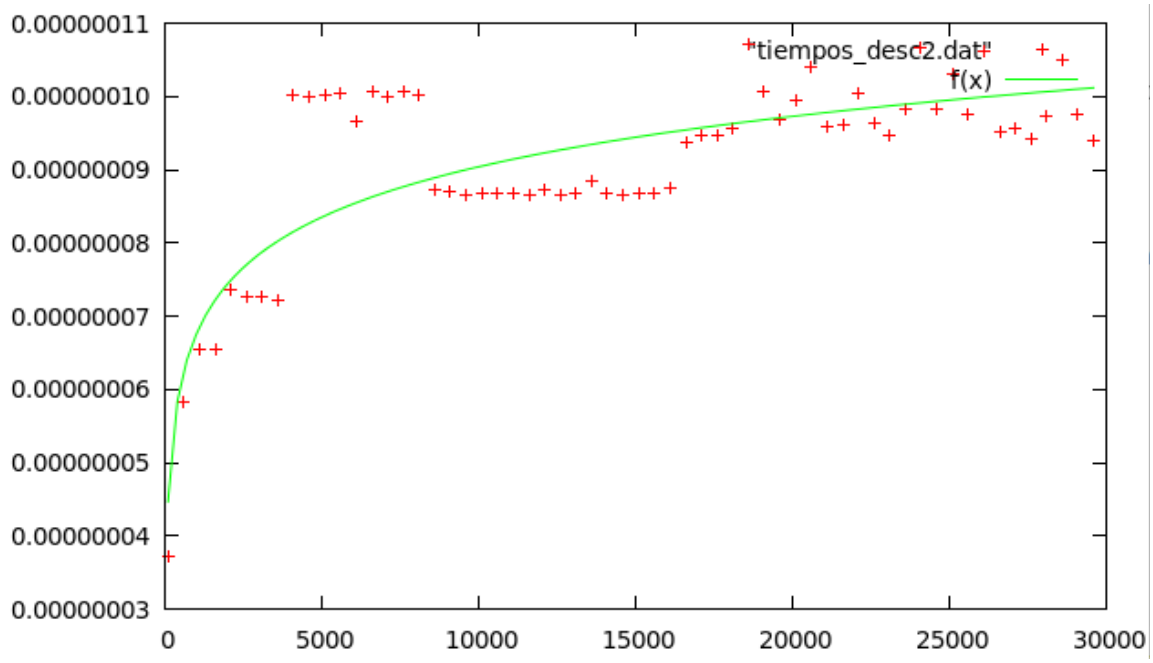
Para solucionar este problema, vamos a ejecutar muchas veces el algoritmo de búsqueda y luego de ejecutar varias veces el algoritmo lo que tenemos que hacer es dividir el tiempo total entre las veces que hemos ejecutado el algoritmo, para ellos realizaremos modificaciones en el código .cpp donde usamos el algoritmo (ejercicio_desc2.cpp). El resultado de ejecutarlo es el siguiente:



Ahora voy a calcular la curva que se ajusta a los datos recopilados para poder hacemos a la idea a que orden pertenece, para ellos vamos a realizar el siguiente procedimiento:

1. f(x)=a*log(b*x)
2. fit f(x) “tiempos_desc2.dat” via a,b
3. plot “tiempos_desc2.dat”, f(x)

```
gnuplot> f(x)=a*log(b*x)
gnuplot> fit f(x) "tiempos_desc2.dat" via a,b


 Iteration 0
 WSSR        : 3.24392e-15       delta(WSSR)/WSSR   : 0
 delta(WSSR) : 0                 limit for stopping : 1e-05
 lambda   : 6.5271

initial set of free parameter values

a               = 9.93207e-09
b               = 0.897428
/

 Iteration 1
 WSSR        : 3.24392e-15       delta(WSSR)/WSSR   : -1.21591e-16
 delta(WSSR) : -3.9443e-31       limit for stopping : 1e-05
 lambda   : 0.65271

resultant parameter values

a               = 9.93207e-09
b               = 0.897428

After 1 iterations the fit converged.
final sum of squares of residuals : 3.24392e-15
rel. change during last iteration : -1.21591e-16

degrees of freedom    (FIT_NDF)                        : 58
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 7.47861e-09
variance of residuals (reduced chisquare) = WSSR/ndf   : 5.59297e-17

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

a               = 9.93207e-09      +/- 9.148e-10    (9.21%)
b               = 0.897428         +/- 0.7633       (85.06%)


correlation matrix of the fit parameters:

               a      b
a               1.000
b              -0.993  1.000
gnuplot> plot "tiempos_desc2.dat", f(x)
```



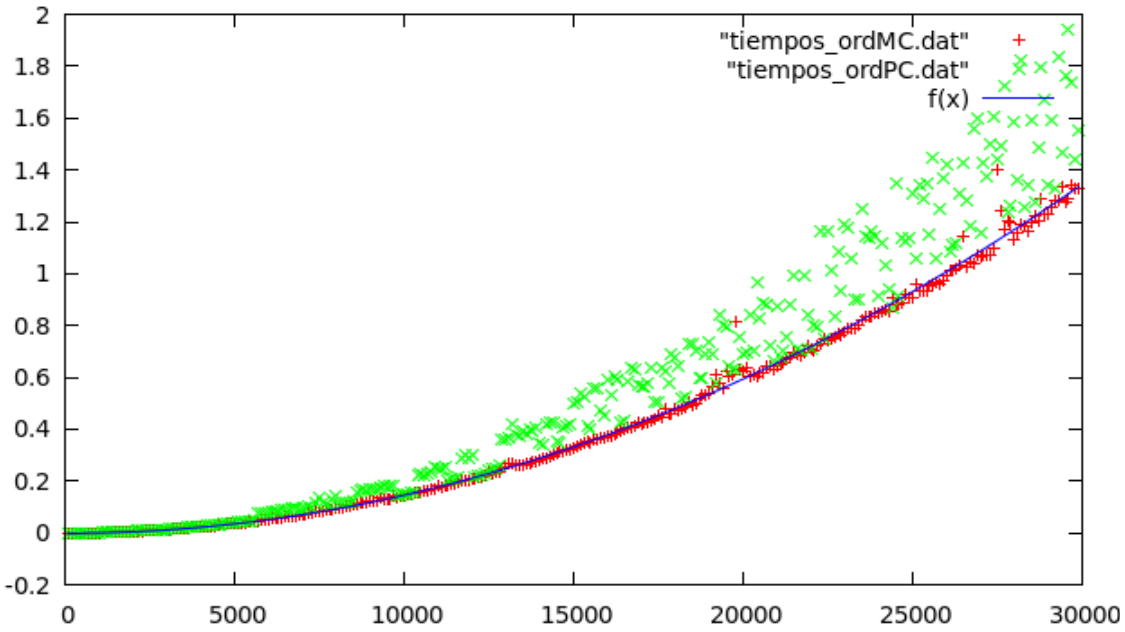
Hemos hecho el ajuste a f(x)=a*log(b*x) por que el orden teórico que hemos calculado ha sido de O(log2n), como observamos se ajusta a los datos relativamente bien.

EJERCICIO 4:

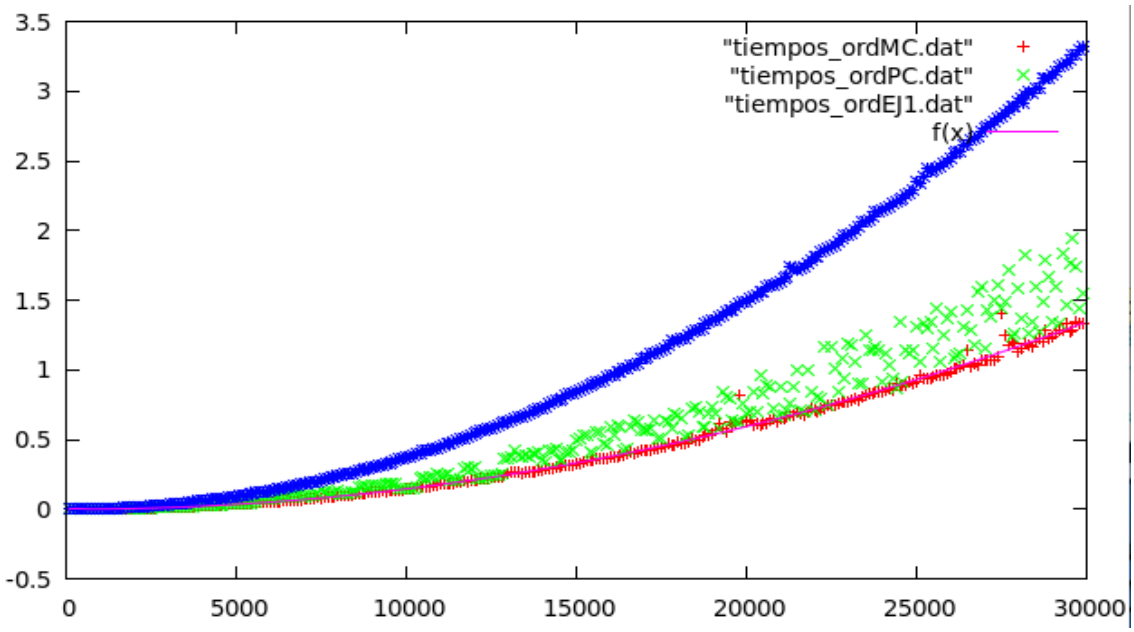
Para realizar este ejercicio y obtener el resultado en los dos escenarios posibles, es decir, en el peor y en el mejor de los casos, tenemos que crear dos algoritmos uno que me ordene el vector para obtener el resultado en el mejor de los casos y otro que me lo desordene por completo para obtener el peor de los casos, para ello he creado estos dos algoritmos:

```
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 desordenar(int *v, int n){
  for(int i=0;i-1;i++)
    for(int j=0;j<n-i-1;j++)
        if(v[j]<v[j+1]){
        int aux=v[j];
        v[j+1]=aux;
      }
}
```

Lo he añadido a los dos archivos cpp usándolo en los sitios adecuados, es decir, antes de coger el tiempo inicial pues ordeno el vector y ya el tiempo me lo daría en el mejor de los casos y viceversa, en los .cpp que adjunto se puede ver claramente. Finalmente nos damos cuenta que ambas gráfica tanto la del mejor de los casos y la del peor de los casos los tiempos son prácticamente iguales y se ajusta al $O(n^2)$, con esto podemos finalmente decir que el algoritmo de búsqueda burbuja es uno de los algoritmos más ineficientes que existen para la búsqueda.



En la siguiente imagen comparo los tiempos obtenidos en el ejercicio 1, los tiempos del peor y mejor caso y el ajuste al $O(n^2)$, lo que nos da a entender que varían muy poco con respecto al termino medio que obtuvimos en el primer ejercicio, por eso decidimos que el orden de los tres casos es $O(n^2)$.

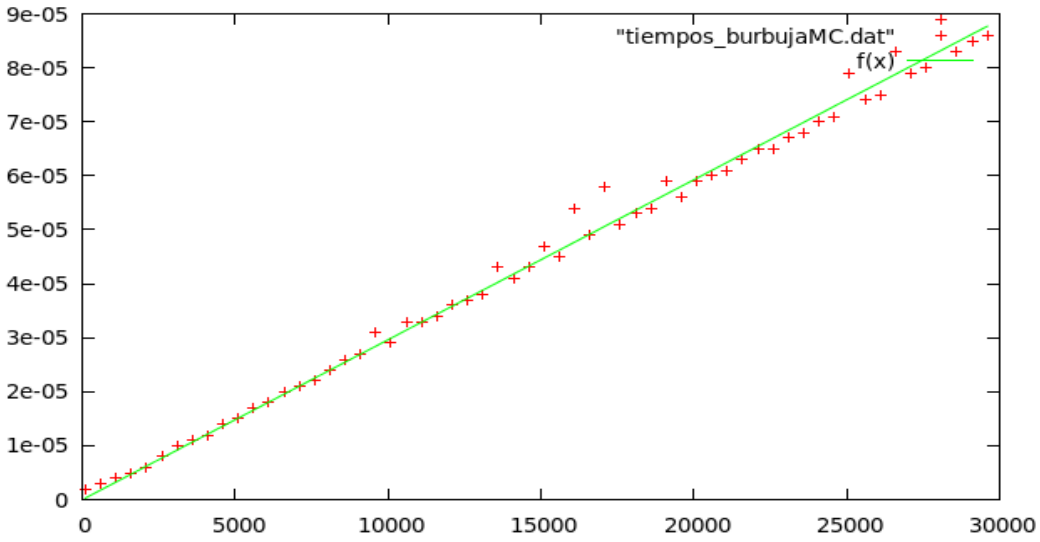


EJERCICIO 5:

Si analizamos la eficiencia teórica que puede tener dicho algoritmo nos vamos a dar cuenta que el primer for solo se va a realizar una vez, debido a que estamos teniendo en cuenta que el vector se encuentra en el mejor de los casos, es decir, ordenador, y que por lo tanto, al estar ordenado el primer for solo se entraría una vez, ya que la condición por la que se termina el bucle, condición booleana, una vez que se entre cambiaría a false y no volvería a entrar. Como ya sabemos que este bucle se entra solo una vez y es de $O(1)$, debido a sus operaciones elementales, vamos a evaluar el bucle de dentro, el interno. El bucle interno se va a realizar una vez,, esto se debe a que va a entrar una vez y como no va a ejecutarse la condición del if y nunca lo va hacer pues el valor de cambio es false siempre por que estamos considerando el vector ordenado, el mejor caso. Entonces podemos deducir que el orden de este bucle es:

Sumatorio(n-i-1) desde 0 hasta 1= (n-0-1)+(n-1-1)=2n-3 → O(n)

Ahora voy a comprobar si es verdad que es esta eficiencia, realizando el procedimiento de manera empírica, como hemos hecho en los casos anteriores:



Como podemos observar hemos realizado la eficiencia teórica correctamente, ya que se corresponde perfectamente al O(n) que hemos calculado con la eficiencia empírica dada por los tiempos que hemos recopilado. El ajuste como siempre lo he realizado como he lo he realizado en los ejercicios anteriores:

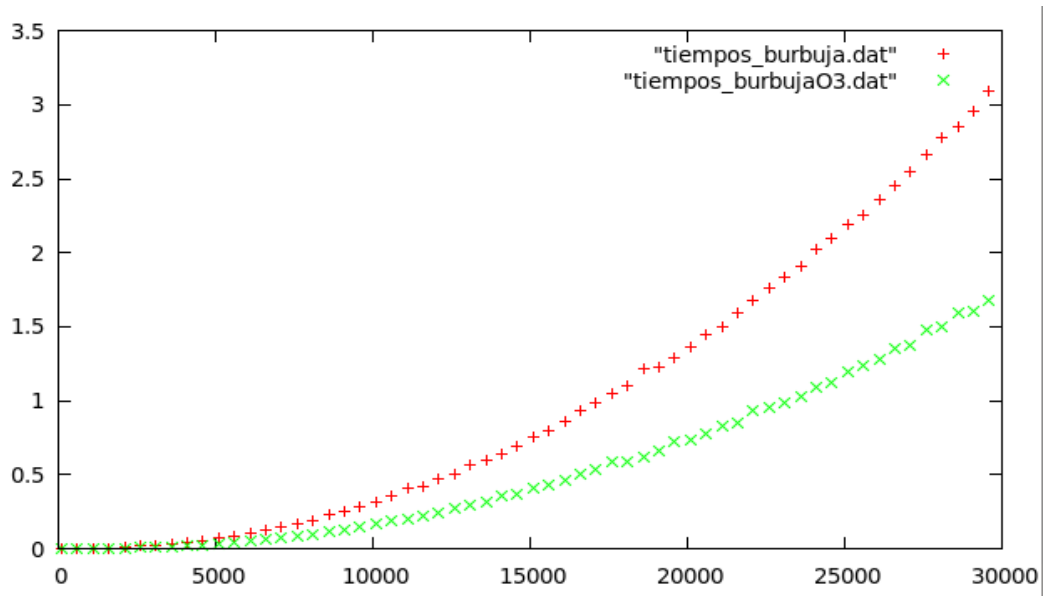
1. f(x)=a*x+b
2. fit f(x) “tiempos_burbujaMC.dat” via a,b
3. plot “tiempos_burbujaMC.dat”, f(x)

EJERCICIO 6:

Para realizar este ejercicio he cogido el codigo del algoritmo de burbuja y he compilado uno sin optimización y otro con optimización O3, de la siguiente manera:

```
g++ burbuja.cpp -o burbuja
g++ -O3 burbuja.cpp -o burbuja_optimizado
```

El primero sin optimización y el segundo con optimización, finalmente he ejecutado ambos algoritmos con sus respectivos script y los he representado en una gráfica ambos dando lugar a que el algoritmo optimizado queden los datos por debajo del no optimizado, luego esto da lugar a que la optimización ayuda a que la eficiencia del algoritmo mejore.



EJERCICIO 7:

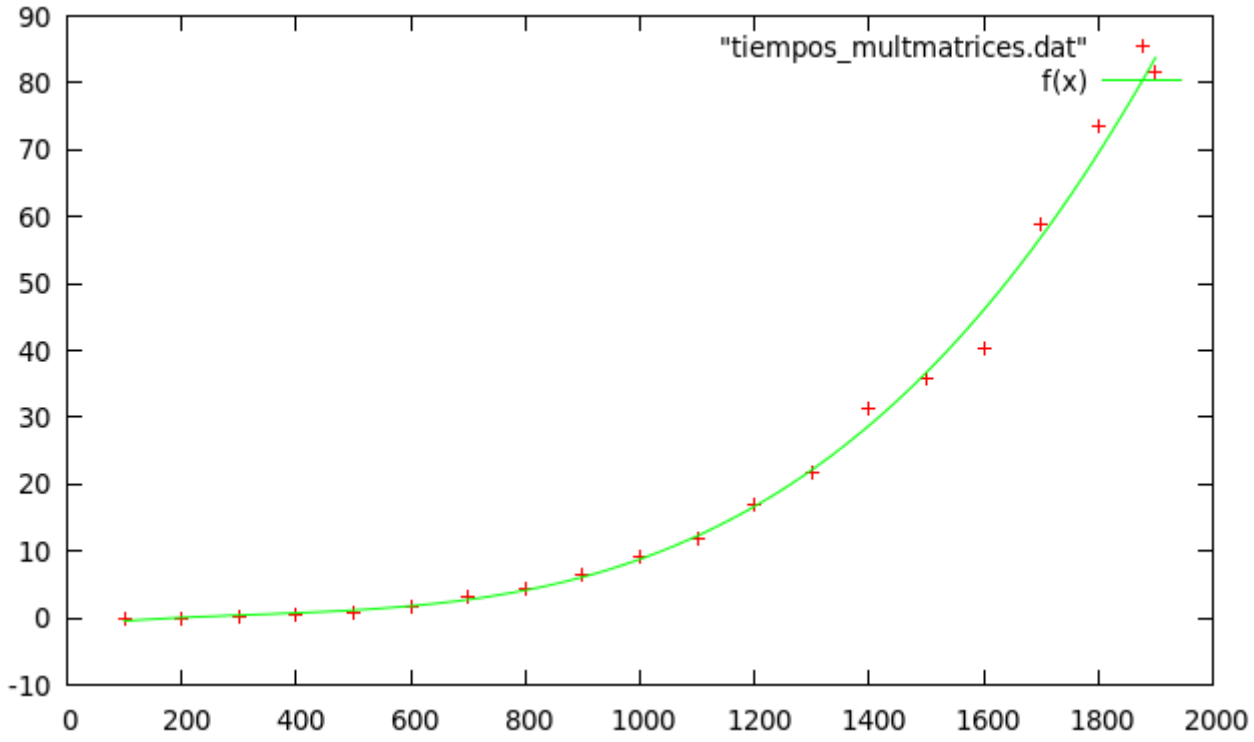
El algoritmos que he usado para multiplicar matrices es para matrices cuadradas para que sea más fácil programarlo, concretamente he usado el siguiente algoritmo:

```
for(int i=0; i<tam; ++i)
   for(int j=0; j<tam; ++j)
       for(int z=0; z<tam; ++z)
           c[i][j] += a[i][z] * b[z][j];
```

Los dos primeros bucles ambos van desde 0 hasta n en el peor de los casos luego aplicamos la regla de la multiplicación y nos saldría que su eficiencia es de orden $O(n^2)$, el ultimo también se produce lo mismo, lo que nos da finalmente un orden de eficiencia de $\mathbf{O(n^3)}$.

Ahora voy a realizar el mismo procedimiento que he ido realizando hasta ahora, como sabemos que la eficiencia del algoritmo es $O(n^3)$ podemos hacer el ajuste a $f(x)=ax^3+bx^2+cx+d$, luego voy a dejar aquí todo el procedimiento. Destacar que antes de todo he recopilado los datos de tiempo del algoritmo usando un script para ellos, después de ejecutar el script he realizado el siguiente procedimiento.

```
gnuplot> f(x)=a*x**3+b*x**2+c*x+d
gnuplot> fit f(x) "tiempos_multmatrices.dat" via a,b,c,d


 Iteration 0
 WSSR        : 65.2874          delta(WSSR)/WSSR   : 0
 delta(WSSR) : 0                limit for stopping : 1e-05
 lambda   : 1.41633e+09

initial set of free parameter values

a               = 2.01222e-08
b               = -1.97927e-05
c               = 0.00970918
d               = -1.22062
/

 Iteration 1
 WSSR        : 65.2874          delta(WSSR)/WSSR   : -1.74133e-15
 delta(WSSR) : -1.13687e-13     limit for stopping : 1e-05
 lambda   : 1.41633e+08

resultant parameter values

a               = 2.01222e-08
b               = -1.97927e-05
c               = 0.00970918
d               = -1.22062

After 1 iterations the fit converged.
final sum of squares of residuals : 65.2874
rel. change during last iteration : -1.74133e-15

degrees of freedom    (FIT_NDF)                        : 15
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 2.08626
variance of residuals (reduced chisquare) = WSSR/ndf   : 4.35249

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

a               = 2.01222e-08      +/- 3.765e-09    (18.71%)
b               = -1.97927e-05     +/- 1.144e-05    (57.79%)
c               = 0.00970918       +/- 0.009977     (102.8%)
d               = -1.22062         +/- 2.365        (193.7%)


correlation matrix of the fit parameters:

               a      b      c      d
a               1.000
b              -0.988  1.000
c               0.929 -0.974  1.000
d              -0.736  0.810 -0.906  1.000
gnuplot> plot "tiempos_multmatrices.dat", f(x)
```

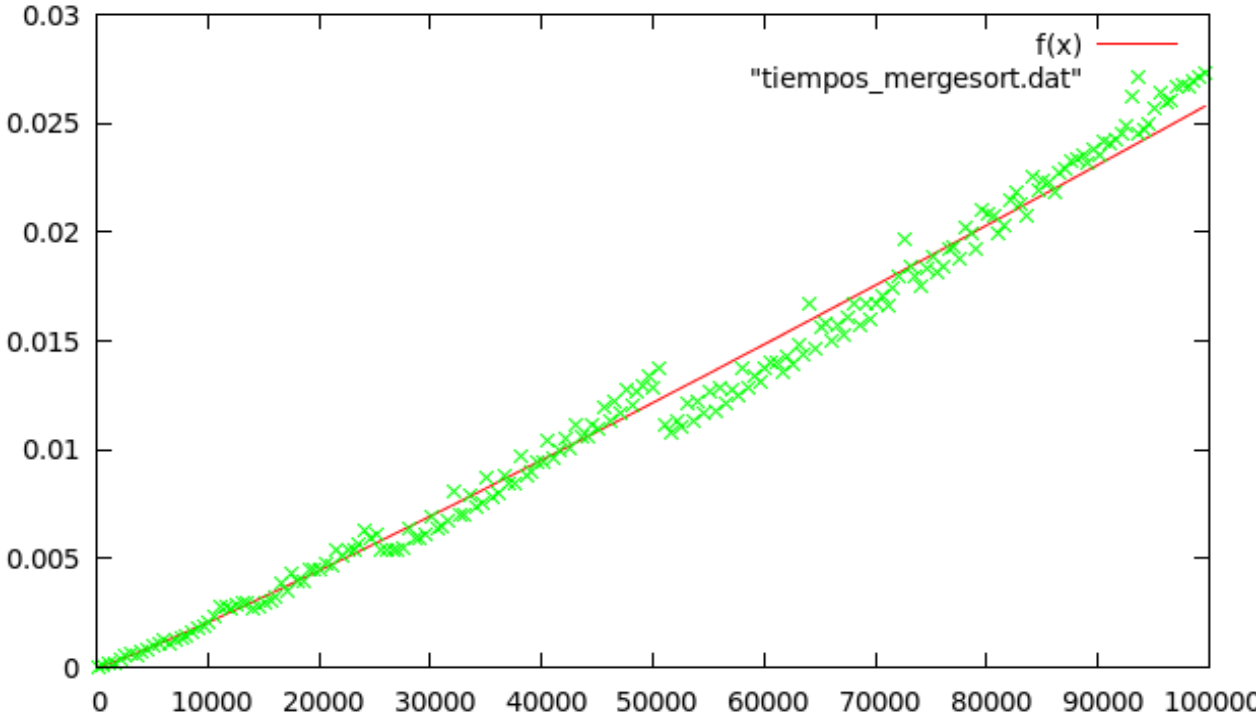


Finalmente como podemos observar se realiza un ajuste perfecto de la función cubica a los datos que hemos recopilado luego podemos deducir que hemos calculado correctamente la eficiencia teórica y que el algoritmo es bastante ineficiente.

EJERCICIO 8:

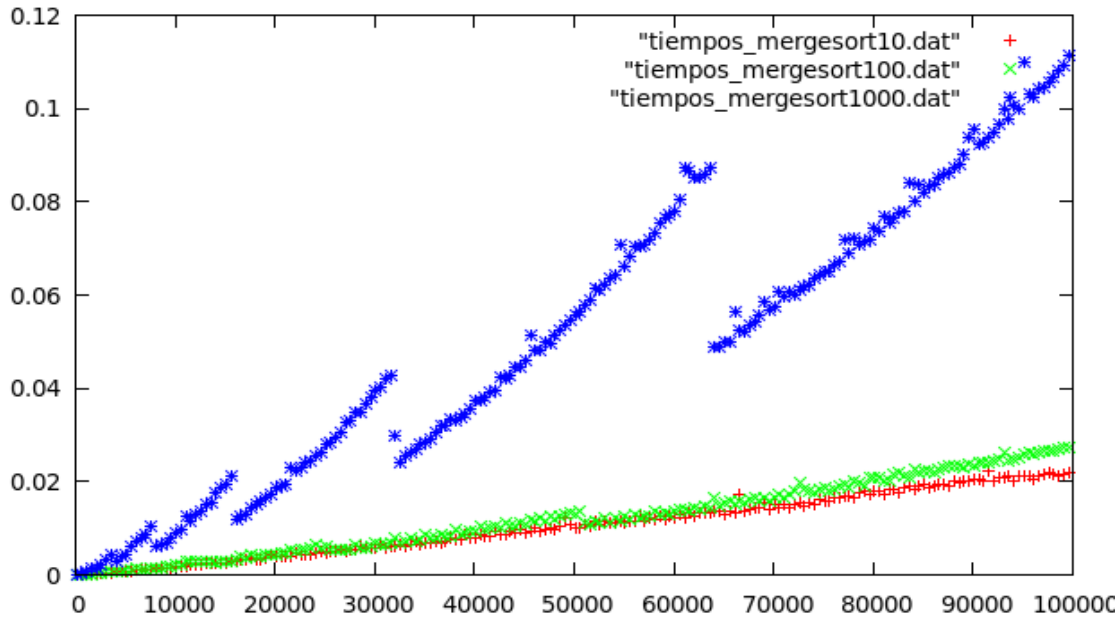
Como siempre he realizado, voy a ejecutar el script para recopilar los datos del algoritmo y luego voy a realizar el ajuste con el orden teórico del algoritmo en este caso **O(nlogn)**. Voy a dejar todo el procedimiento con capturas:

```
gnuplot> f(x)=(a*x)*log(b*x)
gnuplot> fit f(x) "tiempos_mergesort.dat" via a,b


 Iteration 0
 WSSR        : 0.00013785       delta(WSSR)/WSSR   : 0
 delta(WSSR) : 0                limit for stopping : 1e-05
 lambda   : 451760

initial set of free parameter values

a               = 2.26754e-08
b               = 0.911827
*/

 Iteration 1
 WSSR        : 0.00013785       delta(WSSR)/WSSR   : -3.93254e-16
 delta(WSSR) : -5.42101e-20     limit for stopping : 1e-05
 lambda   : 451760

resultant parameter values

a               = 2.26754e-08
b               = 0.911827

After 1 iterations the fit converged.
final sum of squares of residuals : 0.00013785
rel. change during last iteration : -3.93254e-16

degrees of freedom    (FIT_NDF)                        : 198
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 0.000834394
variance of residuals (reduced chisquare) = WSSR/ndf   : 6.96213e-07

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

a               = 2.26754e-08      +/- 3.073e-09    (13.55%)
b               = 0.911827         +/- 1.371        (150.4%)


correlation matrix of the fit parameters:

                a      b
a               1.000
b              -1.000  1.000
gnuplot> plot f(x), "tiempos_mergesort.dat"
```



Como podemos observar el análisis teórico se corresponde de nuevo perfectamente al análisis practico, luego podemos finalizar diciendo que el algoritmo mergesort como bien decía el enunciado tiene un **O(nlogn)**.

Ahora voy a cambiar el valor de UMBRAL_MS como dice el enunciado para ver que es lo que ocurre con la eficiencia del algoritmo si mejora, empeora o no sucede nada. Voy a probar con un valor de UMBRAL_MS de 10, 100 y de 1000 y lo voy a unir todo en una sola gráfica para ver que ocurre:



Finalmente podemos deducir que mientras mayor es el valor del umbral la eficiencia del algoritmo empeora ya que se necesita mayor tamaño del vector para utilizar el algoritmo de inserción en lugar de seguir empleando el mergesort ya que este ultimo es más rapido y nos daria un valor de eficiencia mejor.