Practica 1

UNIVERSIDAD DE GRANADA

Jaime Álvarez Orgaz
Grupo D1

Index

Hardware

```
parallels@jaime:~$ gcc --version
gcc (Ubuntu 7.4.0-1ubuntu1~18.04.1) 7.4.0
Copyright (C) 2017 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

```
parallels@jaime:~$ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 18.04.3 LTS
Release:        18.04
Codename:       bionic
```

```
parallels@jaime:/home$ cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 61
model name      : Intel(R) Core(TM) i5-5257U CPU @ 2.70GHz
stepping        : 4
microcode       : 0x2d
cpu MHz         : 2700.000
cache size      : 3072 KB
physical id     : 0
siblings        : 2
core id         : 0
cpu cores       : 2
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 20
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
pat pse36 clflush mmx fxsr sse sse2 ss ht syscall nx rdtscp lm constant_tsc nopl
 xtopology nonstop_tsc cpuid tsc_known_freq pni pclmulqdq ssse3 fma cx16 pcid ss
e4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand hyp
ervisor lahf_lm abm 3dnowprefetch invpcid_single pti fsgsbase tsc_adjust bmi1 av
x2 smep bmi2 invpcid rdseed adx smap xsaveopt dtherm arat pln pts
bugs            : cpu_meltdown spectre_v1 spectre_v2 spec_store_bypass l1tf mds
swapgs
bogomips        : 5400.00
clflush size    : 64
cache_alignment : 64
address sizes   : 36 bits physical, 48 bits virtual
power management:
```

```
processor       : 1
vendor_id       : GenuineIntel
cpu family      : 6
model           : 61
model name      : Intel(R) Core(TM) i5-5257U CPU @ 2.70GHz
stepping        : 4
microcode       : 0x2d
cpu MHz         : 2700.000
cache size      : 3072 KB
physical id     : 0
siblings        : 2
core id         : 1
cpu cores       : 2
apicid          : 1
initial apicid  : 1
fpu             : yes
fpu_exception   : yes
cpuid level     : 20
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
pat pse36 clflush mmx fxsr sse sse2 ss ht syscall nx rdtscp lm constant_tsc nopl
 xtopology nonstop_tsc cpuid tsc_known_freq pni pclmulqdq ssse3 fma cx16 pcid ss
e4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand hyp
ervisor lahf_lm abm 3dnowprefetch invpcid_single pti fsgsbase tsc_adjust bmi1 av
x2 smep bmi2 invpcid rdseed adx smap xsaveopt dtherm arat pln pts
bugs            : cpu_meltdown spectre_v1 spectre_v2 spec_store_bypass l1tf mds
swapgs
bogomips        : 5400.00
clflush size    : 64
cache_alignment : 64
address sizes   : 36 bits physical, 48 bits virtual
power management:
```

Ejercicio 1

El siguiente código realiza la ordenación mediante el algoritmo de la burbuja:

```
1  void ordenar(int *v, int n) {
2    for (int i=0; i<n-1; i++)
3      for (int j=0; j<n-i-1; j++)
4        if (v[j]>v[j+1]) {
5           swap(v[j],v[j+1]);
6           //alternatva al swap
7          /*int aux = v[j];//incluir algorithm
8          v[j] = v[j+1];
9          v[j+1] = aux;*/
10       }
11 }
```

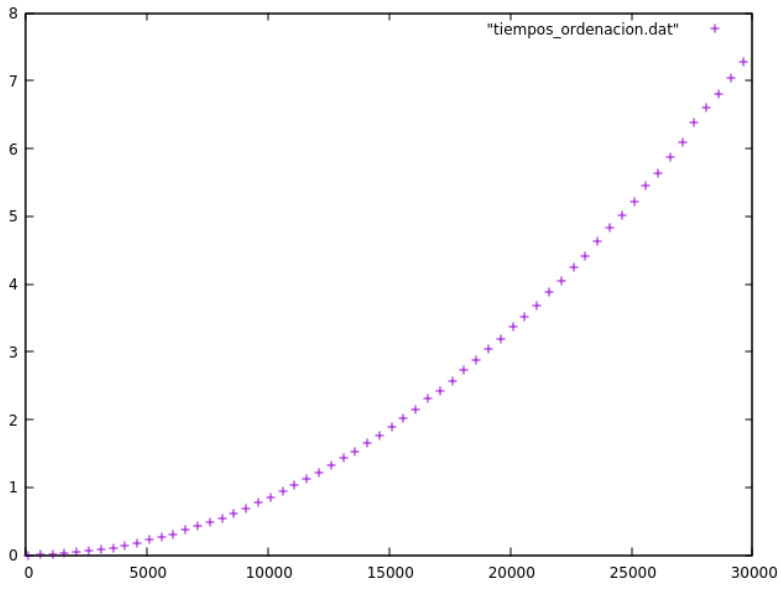
Línea 3. Hay 4 OE: una asignación, una resta, una comparación y un incremento.
Línea 4. Hay 5 OE: igual que la línea anterior, pero se realizan dos restas.
Línea 5. Hay 4 OE: dos accesos a un vector, una suma y una comparación.
Línea 6. Hay 2 OE: asignación y acceso al vector.
Línea 7. Hay 4 OE: dos accesos, suma y asignación.
Línea 8. Hay 3 OE: acceso, suma y asignación.

$$T(n) = 1+\sum_{i=0}^{n-2}\left(4+1+\sum_{j=0}^{n-i-2}18\right) = 1+\sum_{i=0}^{n-2}(5+18(n-i-1)) = 1+\sum_{i=0}^{n-2}18n-\sum_{i=0}^{n-2}18i-\sum_{i=0}^{n-2}13 =$$

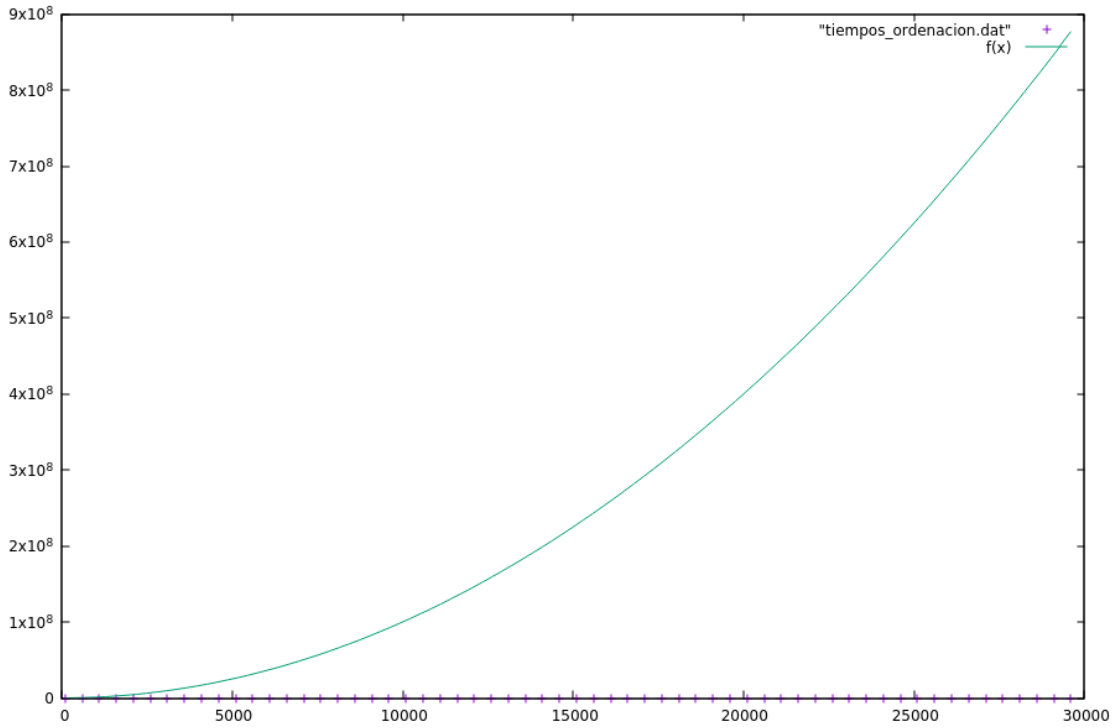
$$= 1+18n(n-1)-18\left(\frac{0+(n-2)}{2}\cdot(n-1)\right)-13(n-1) = 9n^2-\frac{111}{2}n-\frac{21}{2}.$$

Por tanto, afirmamos que T(n) ∈ O(n²), y el algoritmo de ordenación burbuja es de orden de eficiencia O(n²).

Al analizar la eficiencia empírica del algoritmo, obtenemos la siguiente gráfica:



Al representar superpuestas la función de la eficiencia teórica y la empírica, obtenemos lo siguiente:



Podemos observar que la gráfica de *“tiempos_ordenacion.dat”* es una constante en 0 debido a que comparada con la gráfica de la eficiencia teórica, los valores son tan pequeños que parece que no cambia. Solucionaremos esto en el ejercicio 2.

Ejercicio 2

A continuación ajustaremos la eficiencia teórica usando la función

$f(x) = ax^2+bx+c$

Para ello introducimos los siguientes comandos en la terminal

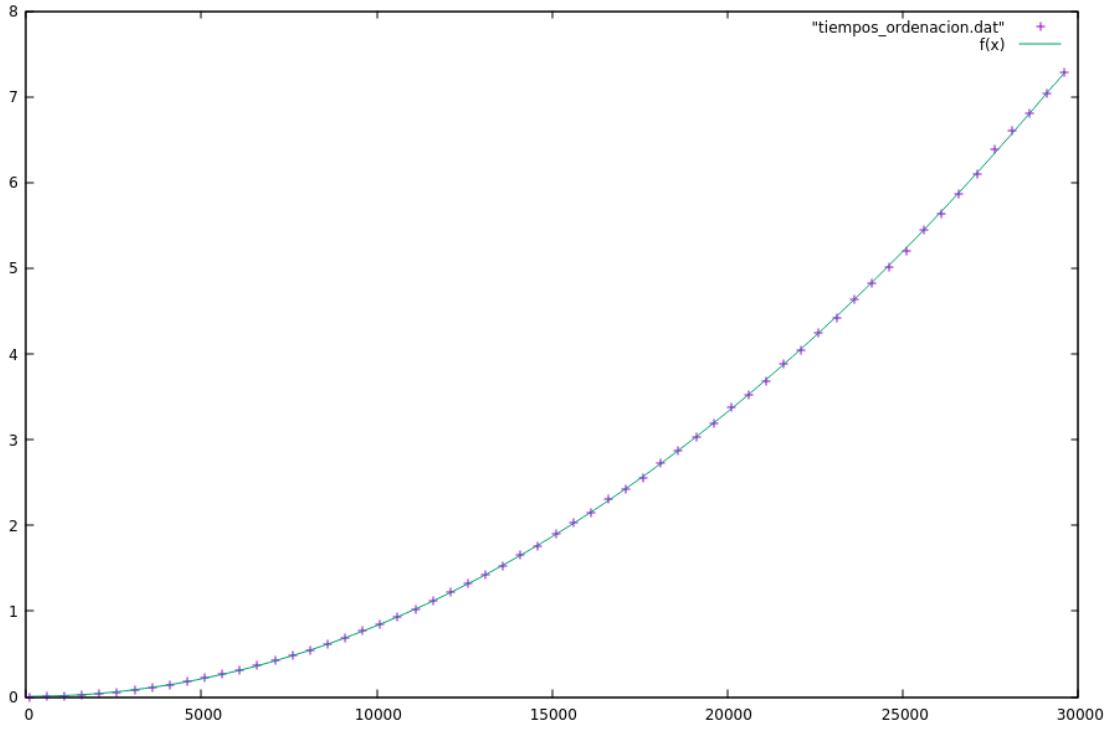
f(x) = a*(x*x) + b*x + c

fit f(x) “tiempos_ordenacion.dat” via a, b, c

y como resultado obtendremos los siguiente valores para a, b, c:

```
Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a                 = 8.34235e-09    +/- 2.379e-11    (0.2852%)
b                 = -8.90801e-07   +/- 7.302e-07    (81.97%)
c                 = 0.00560431     +/- 0.004692     (83.72%)
```

Una vez ajustada, volvemos a pintar la gráfica y observamos que ambas se superponen



Ejercicio 3

A) El algoritmo del fichero "*ejercicio_desc.cpp"* se trata del algoritmo de búsqueda binaria, el cual realiza la búsqueda de un elemento dentro de un vector ordenado.

Para ello, se pasan como parámetros el inicio (inf) y el final (sup) de dicho vector. La función devuelve la posición donde se ha encontrado el elemento, o −1 si no estaba en el vector. Para el correcto funcionamiento del programa el vector debe estar ordenado.

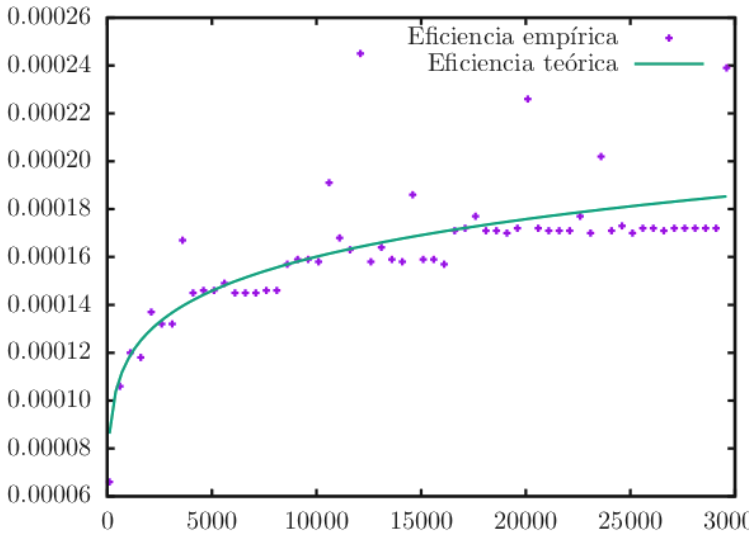
En resumen, el procedimiento se basa en dividir el vector original por la mitad, y si no está
ahí el elemento buscado, nos quedamos únicamente con el sub-vector donde se puede encontrar dicho elemento (recordemos que el vector está ordenado). Repitiendo el proceso, acabaremos encontrando el elemento, o descubriendo que no está en el vector, cuando ya no se puedan hacer más divisiones.

B) Para el cálculo de la eficiencia teórica de la búsqueda binaria, nos basamos en el hecho de que se divide sucesivamente en dos un vector de tamaño n. En el caso peor, el proceso continuará hasta que no se puedan hacer más divisiones del vector.

Como el resto de operaciones son elementales (O(1)), concluímos que la eficiencia del algoritmo es logarítmica, es decir, $T(n) \in O(\log(n))$.

C) El algoritmo es tan rápido que es imposible apreciar cambios en el tiempo de ejecución para vectores de tamaño mayor, lo que nos lleva a obtener una gráfica prácticamente horizontal. Por tanto, para analizar empíricamente la eficiencia de este algoritmo debemos ejecutarlo más de una vez por cada tamaño del vector.

En nuestro caso lo hemos ejecutado 1000 veces por cada tamaño, obteniendo la siguiente gráfica:



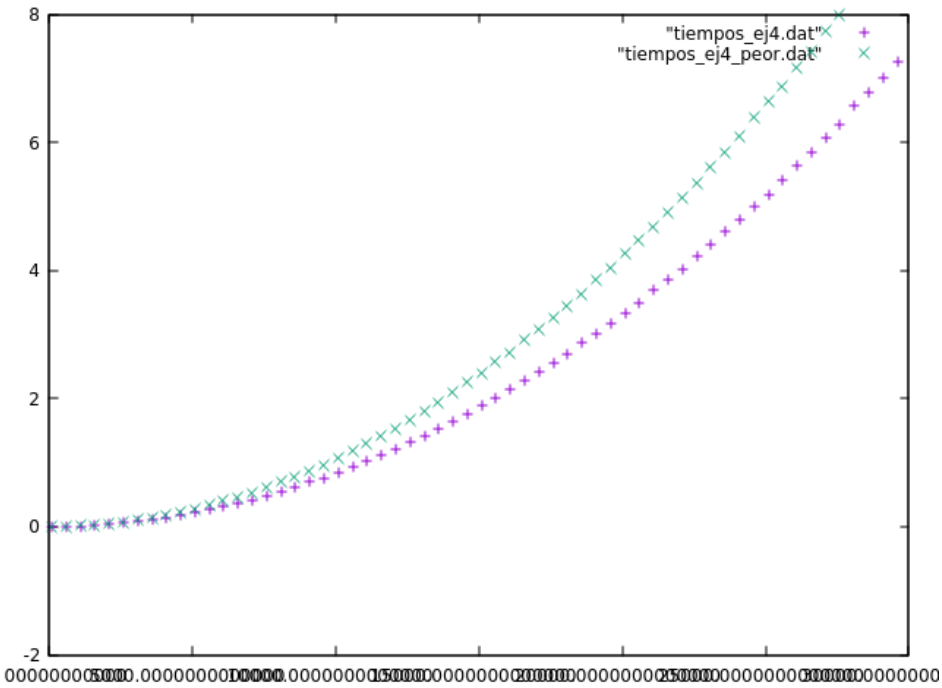
Ejercicio 4

La eficiencia teórica se corresponde con la calculada en el ejercicio 1.

Peor Caso:

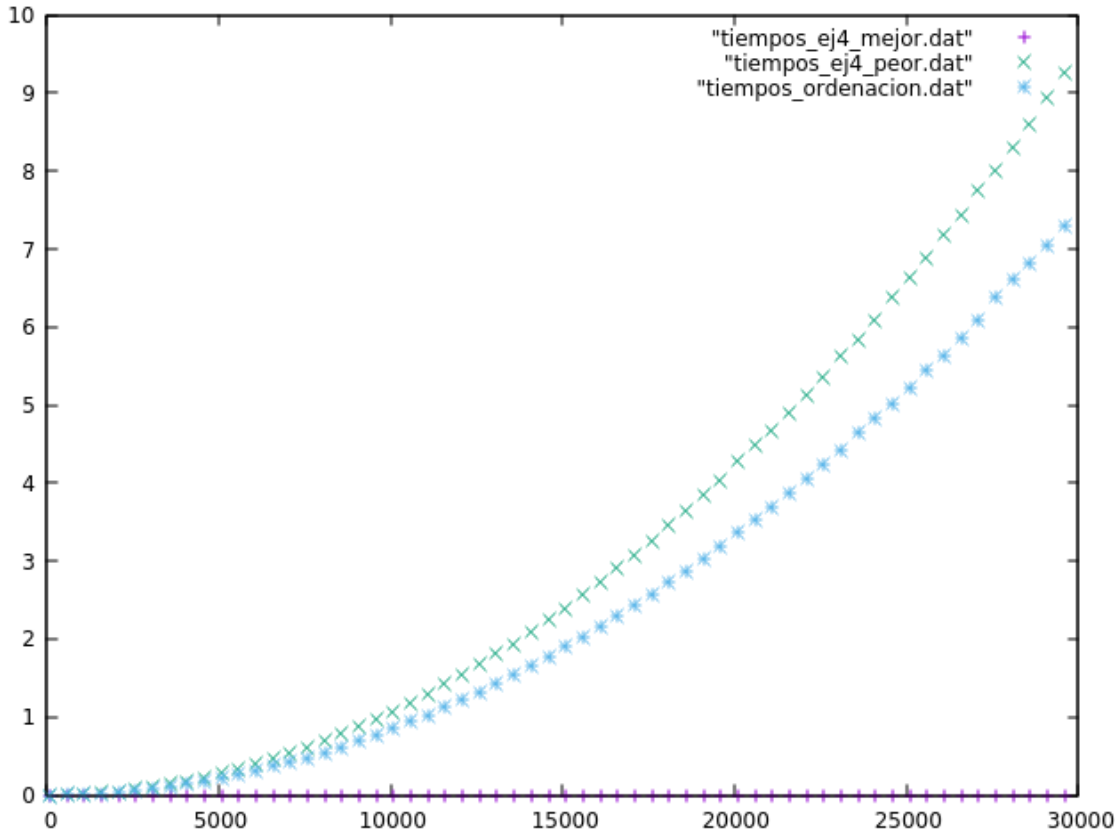
```
// Generación del vector
int *v=new int[tam];        // Reserva de memoria
v[0] = vmax;
for (int i=0; i<tam; i++)   // Recorrer vector
  v[i] = v[i-1] - 1;
```

Comparación



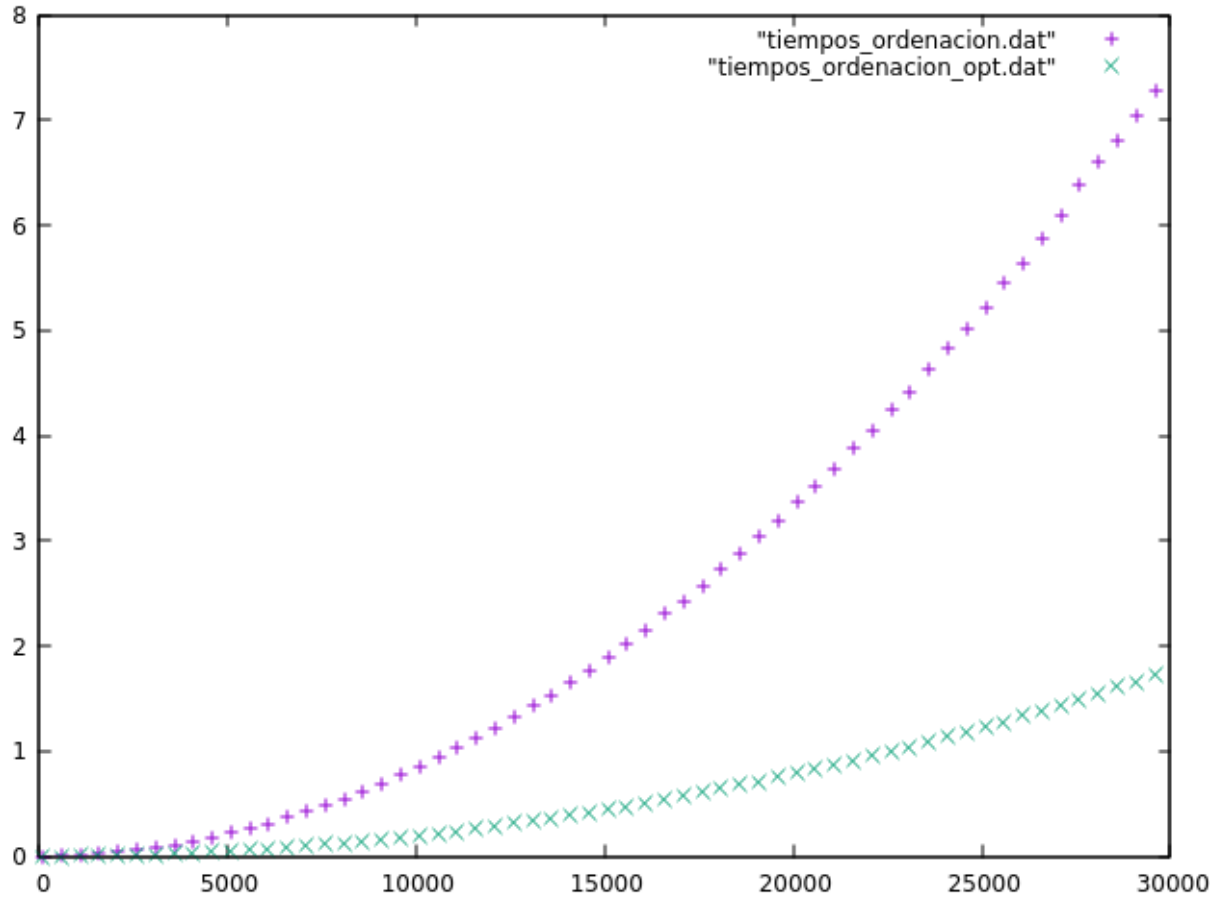
Ejercicio 5

Comparación con el ejercicio 1



Ejercicio 6

Como vemos en la siguiente gráfica, el tiempo de ordenación en el programa compilado con la línea proporcionada, es menor que el tiempo de ordenación con el programa compilado de forma convencional.



Ejercicio 7

El programa creado tiene el siguiente algoritmo para el cálculo de la multiplicación de matrices:

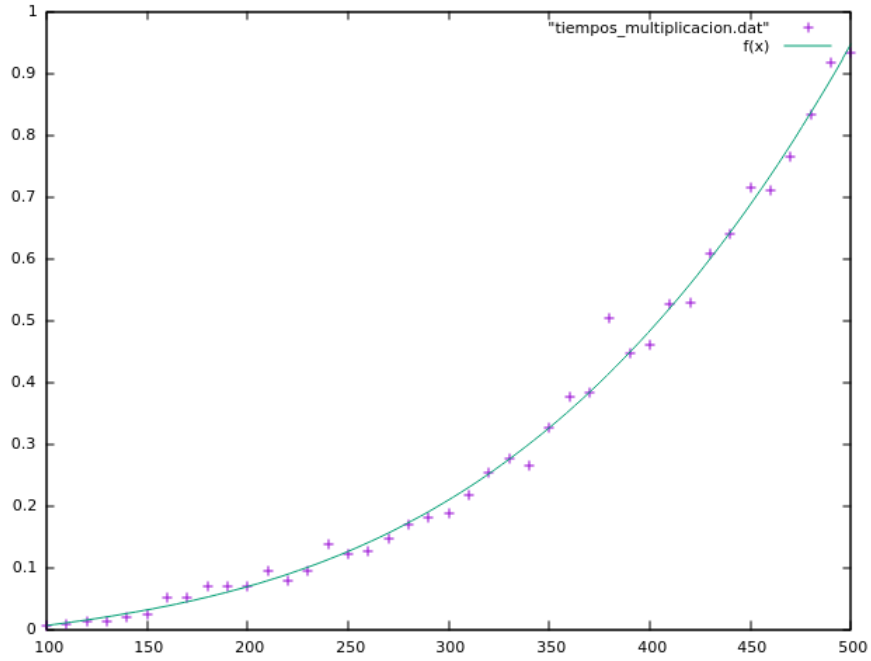
```
void multiplicacion(int **v_m, int **v1, int **v2, int n) {
        for (int i=0; i<n; i++){
            for (int j=0; j<n; j++){
                for (int k=0; k<n; k++){
                    v_m[i][j] += v1[i][k]*v2[k][j];
                }
            }
        }
    }
```

La eficiencia teórica de este algoritmo, teniendo en cuenta el número de multiplicaciones realizadas en el peor de los casos, es de orden **O(n) = n³.** Ahora vamos a pasar a comprobar el algoritmo de forma empírica.

Lo primero que haremos va a ser modificar el script porque si no el tiempo que tarda es muy grande (en mi caso voy a poner los tamaños de la matriz 100, 110, 120, … , 500). Una vez modificado lo ejecutamos y obtenemos los tiempos. Ahora realizaremos la regresión para ajustar la curva teórica a la empírica utilizando la fórmula *f(x) = a*x**3 + b*x**2 + c*x + d. El resultado es:*

a = 9.34251e-09
b = -1.73615e-06
c = 0.000497637
d = -0.0358442

La gráfica resultante para estos valores sería:

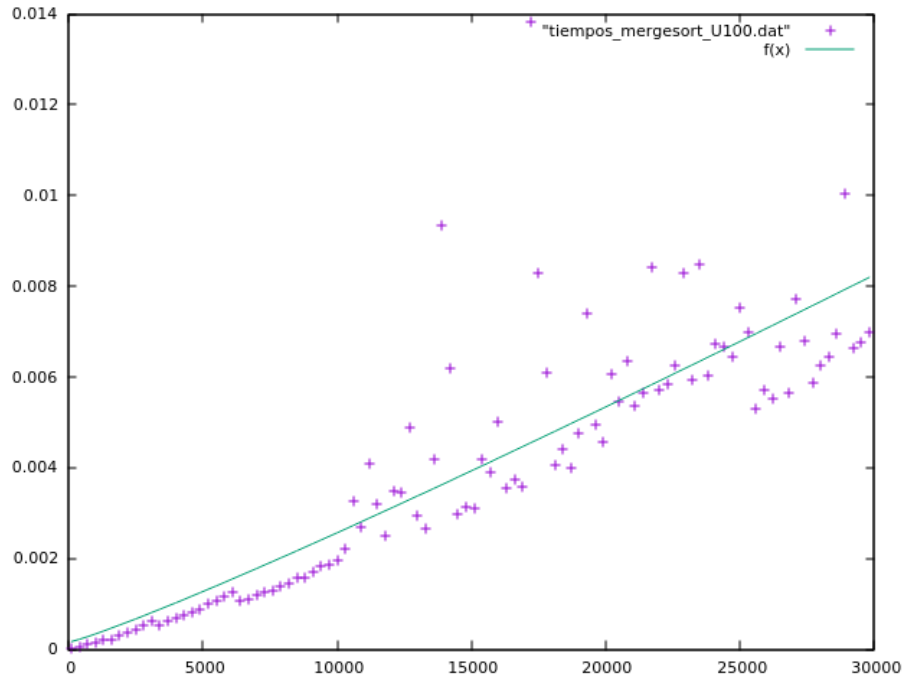


Ejercicio 8

Una vez realizado el análisis de la eficiencia empírica y el ajuste obtenemos los siguientes resultados para la función f(x) = a*(n*log(n)) + b:

a 2.61826e-08
b 0.000154399

La gráfica con el ajuste de regresión es:



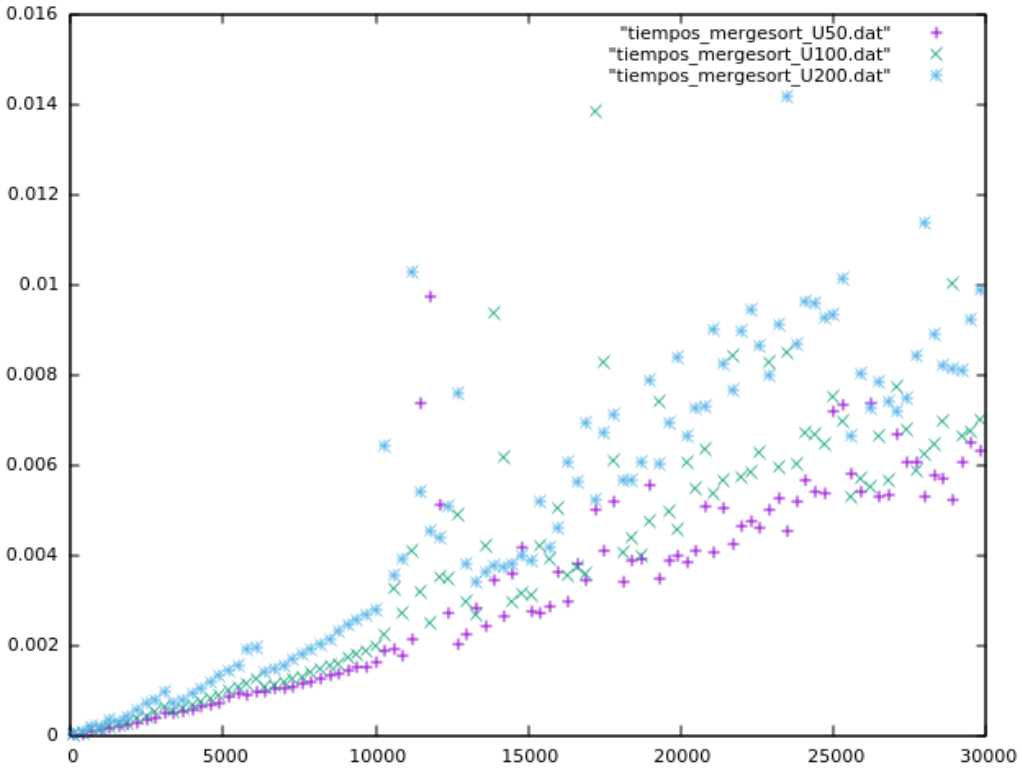
Para el estudio de cómo afecta el parámetro UMBRAL_MS a la eficiencia del algoritmo probamos con distintos valores.

En mi caso lo he realizado con los valores 50, 100 y 200. Gracias a estos datos podemos ver que la eficiencia del algoritmo va disminuyendo a medida que aumentamos el valor del UMBRAL_MS, es decir, que el tiempo que tarda en ejecutarse es mayor.

Esto es debido al “filtro” que hace el programa de la siguiente forma:

```
if (final - inicial < UMBRAL_MS){
        insercion_lims(T, inicial, final);
}
```

A continuación una gráfica con la eficiencia empírica del algoritmo con estos valores:


"tiempos_mergesort_U50.dat"
"tiempos_mergesort_U100.dat"
"tiempos_mergesort_U200.dat"
0.016
0.014
0.012
0.01
0.008
0.006
0.004
0.002
0
0
5000
10000
15000
20000
25000
30000