UNIVERSIDAD
DE GRANADA

2ºA - Grupo A1

Grado en Ingeniería Informática

Estructura de Datos
Práctica 1 - Eficiencia

Autor:
Jose Luis Gallego Peña

15 de octubre de 2018

Índice

1. Introducción

Hardware:

- Procesador: Intel(R) Core(TM) i7-6700HQ CPU @ 2.60GHz
- RAM: 16GB DDR4
- Disco duro: 1TB
- Gráficos: Intel Corporation HD Graphics 530

Sistema operativo: Ubuntu 18.04 LTS

Compilador: GCC g++

2. Ejercicio 1: Ordenación de la burbuja

Se va a calcular la eficiencia de la ordenación mediante el algoritmo de la burbuja, de forma tanto teórica como empírica. El código en C++ de este algoritmo es el siguiente:

Código 1: Algoritmo de ordenación 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         int aux = v[j];
6         v[j] = v[j+1];
7         v[j+1] = aux;
8       }
9 }
```

2.1. Eficiencia teórica

Si el vector que se le pasa al algoritmo está ordenado de forma inversa, entonces tendrá que realizar un número igual de intercambios entre los términos del vector, ya que en cada comparación los términos estarán desordenados. Por tanto, este es el caso más desfavorable: se realizan los mismos intercambios que comparaciones (es el caso que maś tiempo consume).

Analizando el código podemos ver que las líneas 4, 5, 6 y 7 tienen un orden de eficiencia de O(1). El bucle for de la línea 3 tiene un orden de eficiencia de O(n), así como el bucle for de la línea 2 tiene también un orden de O(n). El conjunto de esos dos bucles nos da un orden de $O(n^2)$.

Por tanto concluimos que el algoritmo de ordenación de la burbuja tiene un orden de eficiencia de $O(n^2)$.

2.2. Eficiencia empírica

Se ha creado el fichero ordenacion.cpp con la implementación del algoritmo para así ejecutarlo.

Código 2: ordenacion.cpp

```
#include <iostream>
#include <ctime>        // Recursos para medir tiempos
#include <cstdlib>      // Para generacion de numeros pseudoaleatorios
#include <algorithm>    // std::sort

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;
      }
}

void sintaxis(){

  cerr << "Sintaxis:" << endl;
  cerr << "  TAM: Tamanio del vector (>0)" << endl;
  cerr << "  VMAX: Valor maximo (>0)" << endl;
  cerr << "Se genera un vector de tamanio TAM con elementos aleatorios en [0,VMAX["
       << endl;
  exit(EXIT_FAILURE);

}

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

  // Lectura de parametros
  if (argc!=3)
    sintaxis();
  int tam=atoi(argv[1]);      // Tamanio del vector
  int vmax=atoi(argv[2]);     // Valor maximo
  if (tam<=0 || vmax<=0)
    sintaxis();

  // Generacion del vector aleatorio
  int *v=new int[tam];        // Reserva de memoria
  srand(time(0));             // Inicializacion del generador de numeros 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();

```

```
49    ordenar(v,tam);
50
51    clock_t tfin;     // Anotamos el tiempo de finalizacion
52    tfin=clock();
53
54    // Mostramos resultados
55    cout << tam << "\t" << (tfin-tini)/(double)CLOCKS_PER_SEC << endl;
56
57    delete [] v;      // Liberamos memoria dinamica
58
59  }
```

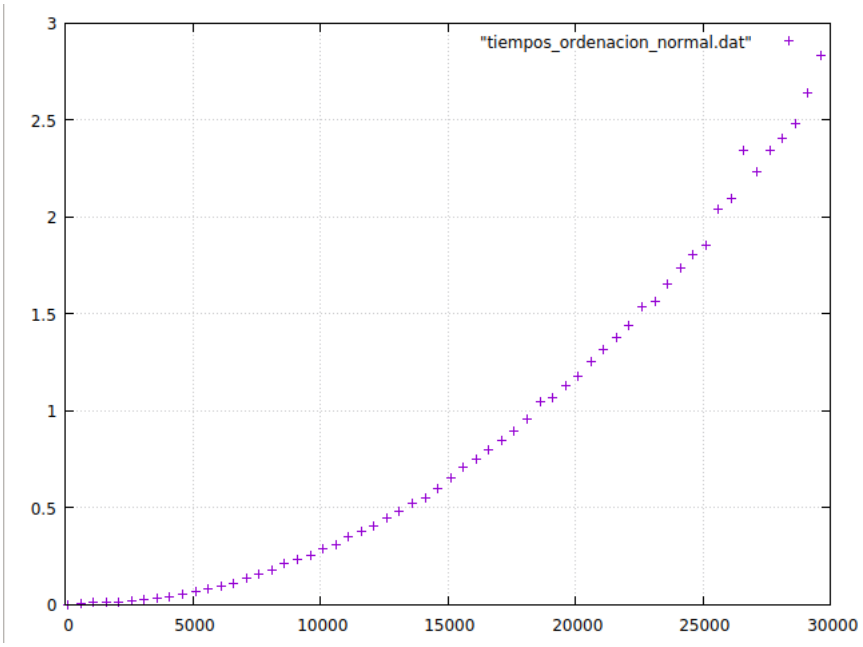
También se ha creado el script ejecuciones_ordenacion.csh para ejecutar varias veces el programa y obtener un archivo con los datos obtenidos.

Código 3: ejecuciones_ordenacion.csh

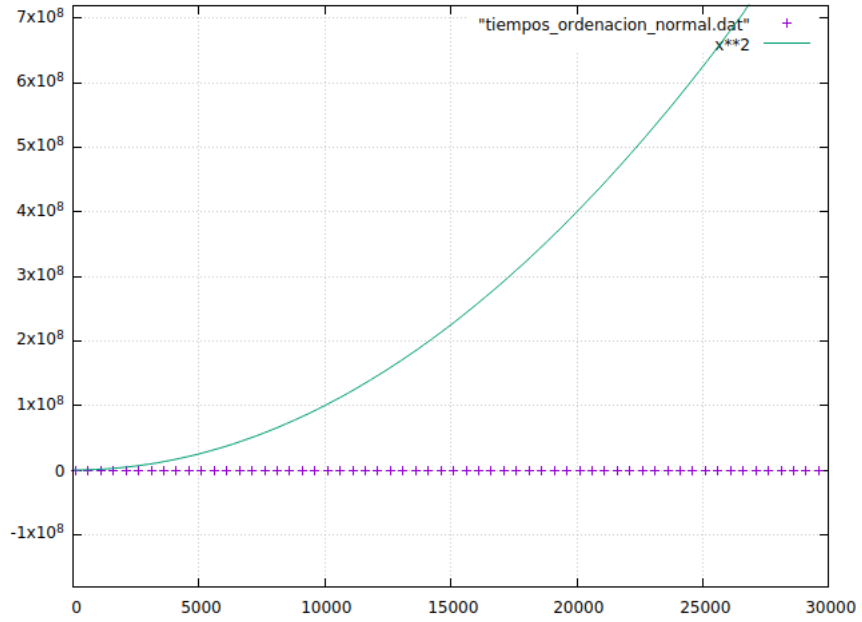
```
1  #!/bin/csh
2  @ inicio = 100
3  @ fin = 1000000
4  @ incremento = 500
5  set ejecutable = ordenacion
6  set salida = tiempos_ordenacion.dat
7
8  @ i = $inicio
9  echo > $salida
10 while ( $i <= $fin )
11   echo Ejecucion tam = $i
12   echo `./{$ejecutable} $i 10000` >> $salida
13   @ i += $incremento
14 end
```

Se llama al script, que ejecuta el algoritmo de ordenación con valores de tamaño del vector que van desde 100 hasta 30.000, en incrementos de 500. Se obtiene un fichero de datos llamado tiempos_ordenacion.dat que contiene esos tiempos de ejecución que se han ido calculando según el tamaño del vector.

A continuación representamos los datos en una gráfica usando gnuplot, y podemos observar como la gráfica tiene forma de parábola (orden n^2).



Superponiendo la función con la eficiencia teórica y la empírica nos queda una gráfica algo más extraña, en la que los datos empíricos dejan de ser una parábola y son una línea recta, ya que hay muchos menos puntos representados.



3. Ejercicio 2: Ajuste en la ordenación de la burbuja

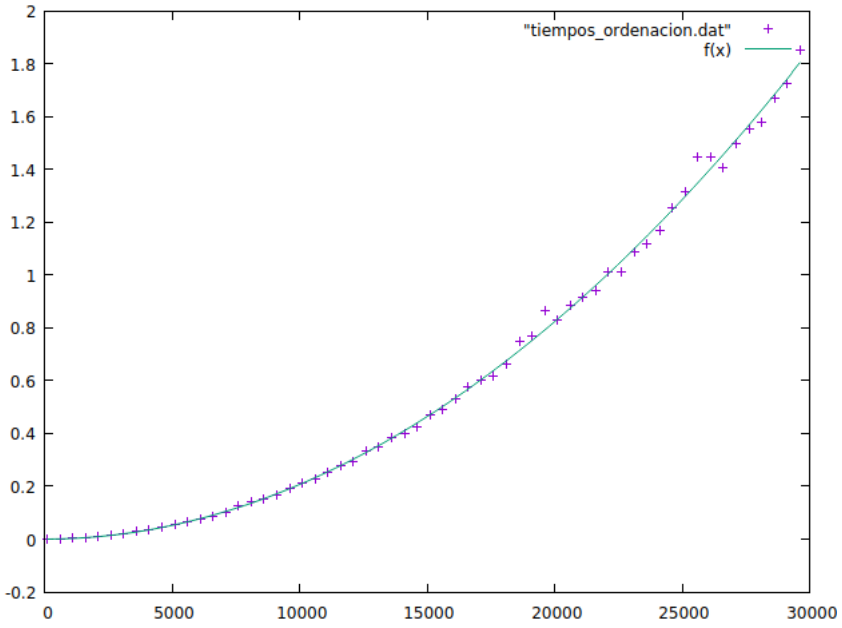
A continuación vamos a ajustar una función de regresión a los datos obtenidos anteriormente de las ejecuciones del algoritmo de la burbuja. En este caso, al ser de orden n^2, la forma es una pará-

bola, por tanto consideraremos a la función de regresión f(x) de la forma $ax^2 + bx + c$.

A partir de los datos calculados en el apartado anterior, y usando la orden fit de gnuplot, obtenemos los valores de los parámetros y representamos la gráfica.

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

- a = 2.05141e-09
- b = 1.75292e-07
- c = -0.000178031



4. Ejercicio 3: Problemas de precisión

Se tiene el fichero ejercicio_desc.cpp con un algoritmo desconocido que se explicará a continuación.

Código 4: ejercicio_desc.cpp

```
1
2  #include <iostream>
3  #include <ctime>    // Recursos para medir tiempos
4  #include <cstdlib>  // Para generacion de numeros pseudoaleatorios
5
6  using namespace std;
7
8  int operacion(int *v, int n, int x, int inf, int sup) {
9    int med;
```

```
10    bool enc=false;
11    while ((inf<sup) && (!enc)) {
12      med = (inf+sup)/2;
13      if (v[med]==x)
14        enc = true;
15      else if (v[med] < x)
16        inf = med+1;
17      else
18        sup = med-1;
19    }
20    if (enc)
21      return med;
22    else
23      return -1;
24  }
25
26  void sintaxis()
27  {
28    cerr << "Sintaxis:" << endl;
29    cerr << "  TAM: Tamanio del vector (>0)" << endl;
30    cerr << "Se genera un vector de tamanio TAM con elementos aleatorios" << endl;
31    exit(EXIT_FAILURE);
32  }
33
34  int main(int argc, char * argv[])
35  {
36    // Lectura de parametros
37    if (argc!=2)
38      sintaxis();
39    int tam=atoi(argv[1]);      // Tamanio del vector
40    if (tam<=0)
41      sintaxis();
42
43    // Generacion del vector aleatorio
44    int *v=new int[tam];        // Reserva de memoria
45    srand(time(0));             // Inicializacion del generador de numeros pseudoaleatorios
46    for (int i=0; i<tam; i++)   // Recorrer vector
47      v[i] = rand() %tam;
48
49    clock_t tini;     // Anotamos el tiempo de inicio
50    tini=clock();
51
52    // Algoritmo a evaluar
53    operacion(v,tam,tam+1,0,tam-1);
54
55    clock_t tfin;     // Anotamos el tiempo de finalizacion
56    tfin=clock();
57
58    // Mostramos resultados
59    cout << tam << "\t" << (tfin-tini)/(double)CLOCKS_PER_SEC << endl;
60
61    delete [] v;      // Liberamos memoria dinamica
62  }
```

El programa exige que se le pase por línea de comandos un único parámetro, el tamaño de un vector. Genera un vector aleatorio de ese tamaño, y a continuación ejecuta el algoritmo llamado *operacion*, midiendo el tiempo que tarda en ejecutarse por completo y mostrándolo por pantalla.

El algoritmo es el de la búsqueda binaria. Pide como argumentos el vector, el tamaño del vector, un valor a buscar, una cota inferior (0, el principio del vector) y una cota superior (tamaño-1, el final del vector). Busca a lo largo del vector (entre las dos cotas) un valor x. Para ello en cada iteración va acortando el vector, buscando si está en la mitad de los dos valores acotados; si el valor de la mitad es menor que el buscado, se sigue buscando de la misma forma por la parte derecha del vector, si no, se sigue buscando por la parte izquierda del vector.

4.1. Eficiencia teórica

El peor caso en el que este algoritmo podría actuar sería sobre un vector en el que no existe el valor buscado.

Teniendo un vector de tamaño N, en cada iteración del algoritmo el tamaño del vector se parte por la mitad, es decir, $\frac{N}{2}$. Consecutivamente se obtendrán tamaños $\frac{N}{4}$, $\frac{N}{8}$, $\frac{N}{16}$... hasta un número n de veces. Por tanto la última iteración, en la que ya no se podrá subdividir más el vector, sería $\frac{N}{2^n} = 1$, ya que si no se puede subdividir más, solo hay un elemento en el vector.

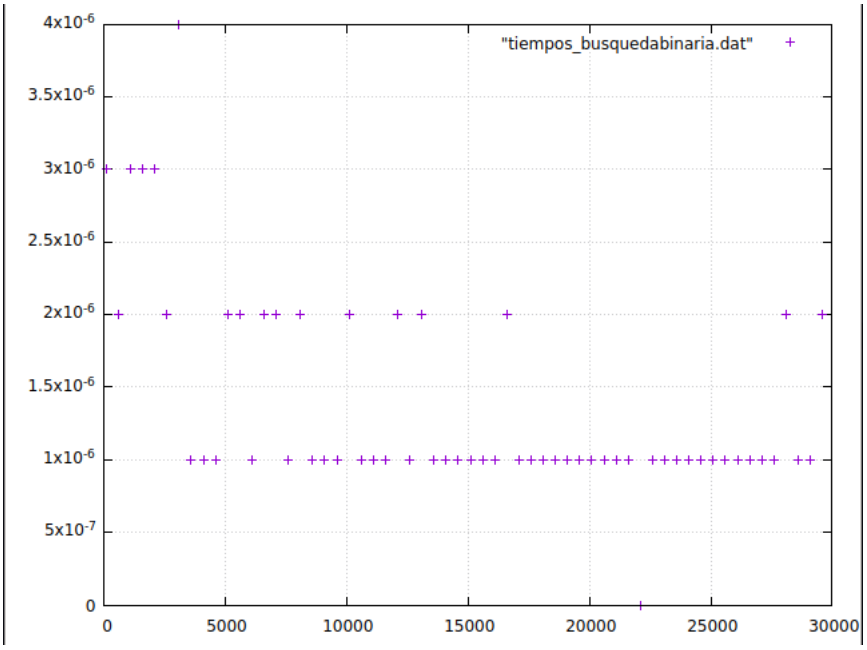
Desarrollando la igualdad matemáticamente tenemos lo siguiente:

$$\frac{N}{2^n} = 1$$
$$N = 2^n$$
$$\log_2(N) = log_2(2^n)$$
$$\log_2(N) = n * log_2(2)$$
$$\log_2(N) = n$$

Por tanto tenemos que el orden de eficiencia de la búsqueda binaria es O($\log_2(n)$).

4.2. Eficiencia empírica

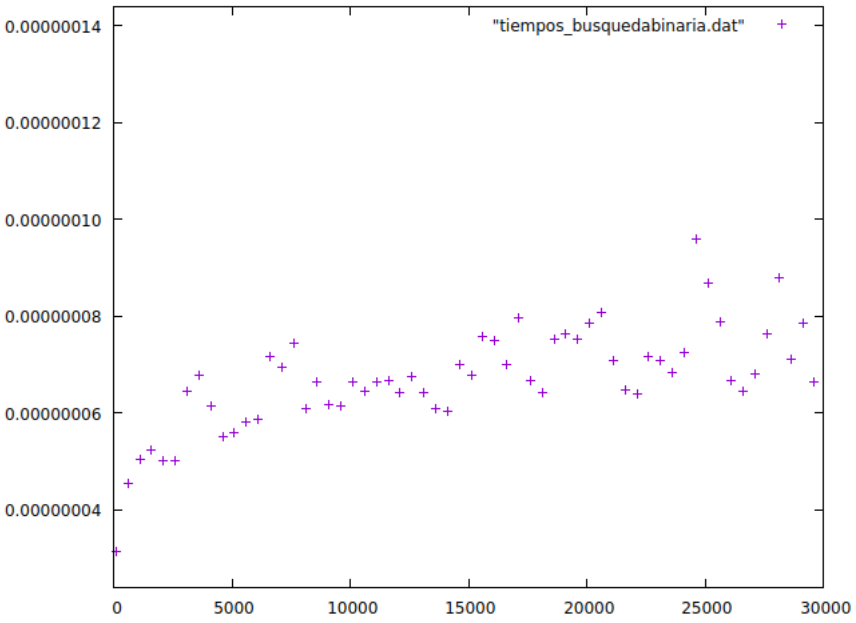
De la misma manera que se ha realizado anteriormente, se ejecuta el algoritmo con valores de tamaño del vector que van desde 100 hasta 30.000, en incrementos de 500. Se obtiene un fichero de datos con los tiempos de ejecución calculados en cada ejecución del programa. Representando los datos:



Podemos observar que la gráfica es extraña, no da unos valores lógicos debido a que el algoritmo es muy rápido y los recursos de ctime no tienen tanta precisión como sería deseable. Para solucionar esto meteremos el algoritmo en un bucle for y dividiremos el tiempo total de ejecución por el número de ejecuciones realizadas.

Código 5: Instrucciones añadidas

```
1
2  // Algoritmo a evaluar
3  for (int i = 0 ; i < 100000 ; i++){
4    operacion(v,tam,tam+1,0,tam-1);
5  }
6
7  ...
8
9  // Mostramos resultados
10 cout << tam << "\t" << (tfin-tini)/(double)CLOCKS_PER_SEC/100000 << endl;
```



Y vemos que efectivamente la gráfica, aunque tenga los puntos muy dispersos, ya se asemeja más a la eficiencia teórica, del orden O($\log_2(n)$).

4.3. Ajuste de la curva teórica a la empírica

5. Ejercicio 4: Mejor y peor caso

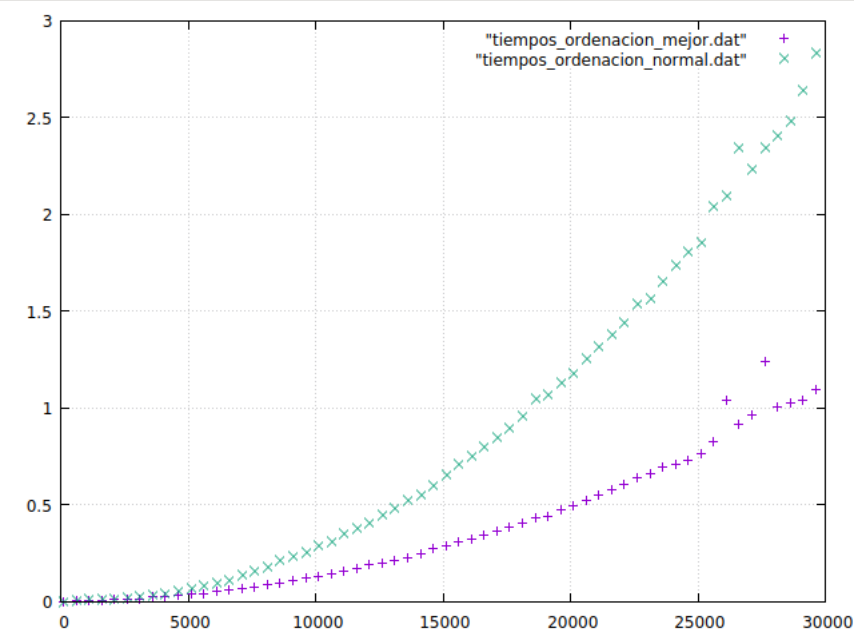
5.1. Mejor caso

El mejor caso en el que se podría ejecutar la burbuja es el caso en el que el vector esté ordenado por completo, ya que no tiene que intercambiar ningún elemento. Para forzar este caso en el programa se añade el siguiente código antes de llamar al algoritmo:

Código 6: Mejor caso caso

```
1     sort(v, v + tam);
```

Y nos queda la siguiente gráfica, en la que podemos observar como la gráfica es más rápida que la del ejercicio 1 ya que emplea menos tiempo y realiza las ejecuciones a partir del tamaño 5000 con medio segundo de ventaja aproximadamente (realiza una curva ascendente menos pronunciada, es decir, no emplea tanto tiempo).



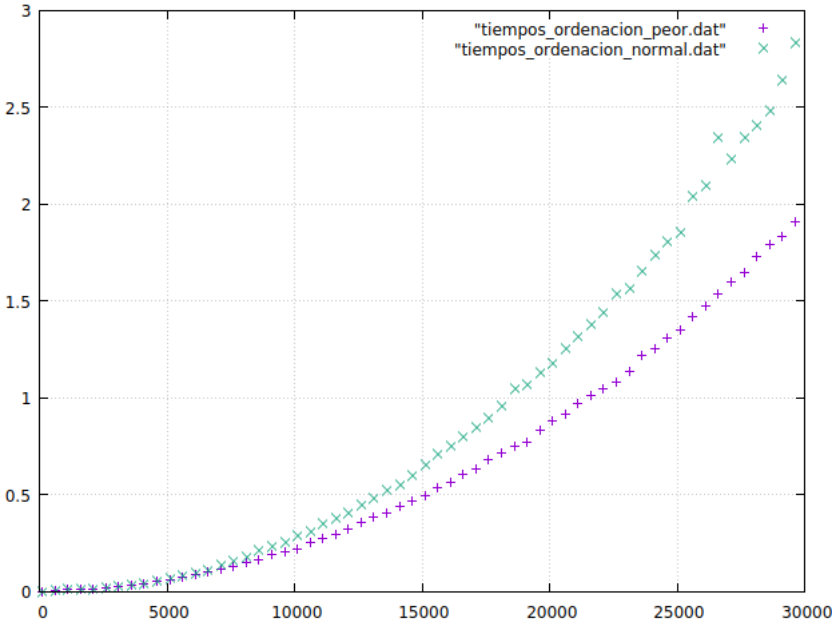
5.2. Peor caso

El peor caso en el que se podría ejecutar la burbuja es, como hemos dicho anteriormente, el caso en el que el vector esté ordenado por completo de forma inversa ya que en cada iteración hay que hacer un intercambio. Para forzar este caso en el programa se añade el siguiente código antes de llamar al algoritmo:

Código 7: Peor caso

```
1     sort(v, v + tam);
2     reverse(v, v + tam);
```

Y nos queda la siguiente gráfica, en la que podemos observar como la curva del peor caso emplea más tiempo que la del mejor caso, pero no llega a tardar más que la del caso normal.



6. Ejercicio 5: Dependencia de la implementación

Se añade una nueva implementación del algoritmo de la burbuja en la que se ha introducido una variable que permite saber si, en una de las iteraciones del bucle externo no se ha modificado el vector. Si esto ocurre significa que ya está ordenado y no hay que continuar.

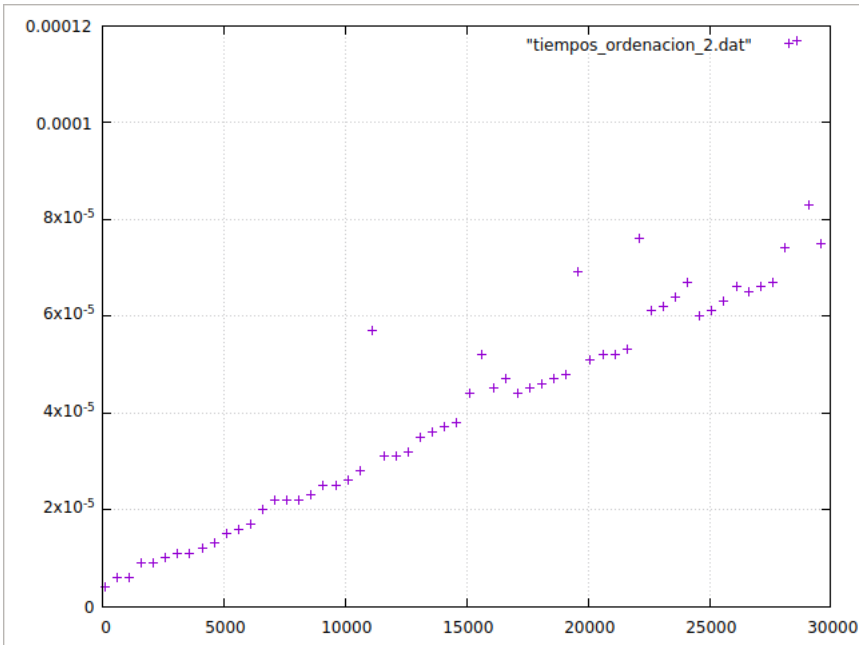
Código 8: Nueva implementación del algoritmo de la burbuja

```
1  void ordenar(int *v, int n) {
2    bool cambio=true;
3      for (int i=0; i<n-1 && cambio; i++) {
4        cambio=false;
5        for (int j=0; j<n-i-1; j++)
6          if (v[j]>v[j+1]) {
7            cambio=true;
8            int aux = v[j];
9            v[j] = v[j+1];
10           v[j+1] = aux;
11         }
12   }
13 }
```

Considerando ahora el mejor caso posible para este algoritmo, que el vector de entrada ya esté ordenado, la eficiencia teórica sería del orden constante O(1) ya que en este caso no se intercambia ningún elemento y por tanto no realiza ninguna iteración el bucle. Además, las variables bool se aseguran de no iterar en el bucle de forma inútil, es decir, sabiendo que el vector ya está ordenado.

Calculando la eficiencia empírica de esta implementación tenemos la siguiente gráfica, en la que podemos observar que el algoritmo se ejecuta de forma extremadamente rápida, y se adapta a la

función lineal de orden constante que hemos descrito en la eficiencia teórica. Por lo tanto, podemos concluir que la eficiencia de un algoritmo depende mucho de su implementación.

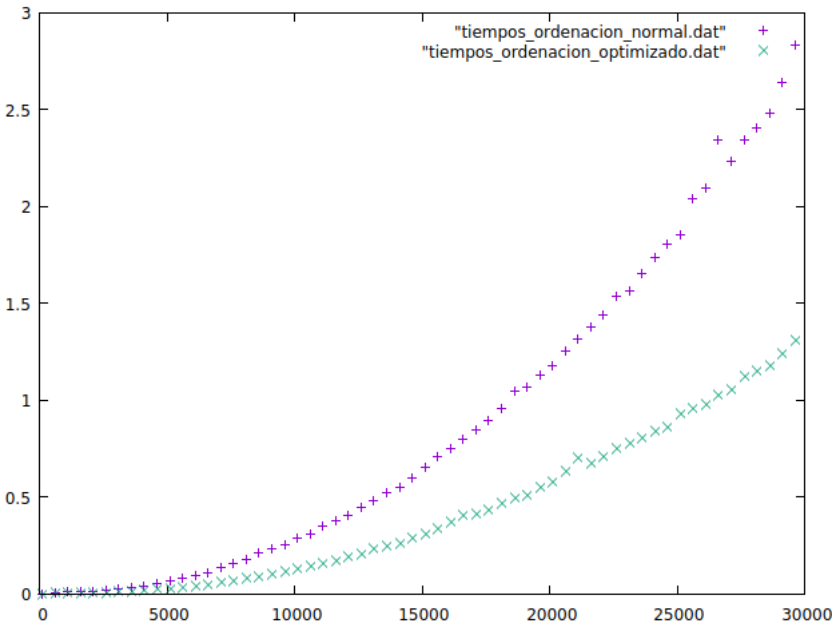


7. Ejercicio 6: Influencia del proceso de compilación

Compilamos el programa que realiza el algoritmo de ordenación de la burbuja, pero ahora indicándole al compilador que optimice el código con la siguiente orden:

```
1 $ g++ -O3 ordenacion.cpp -o ordenacion_optimizado
```

En la gráfica tenemos las curvas de eficiencia empírica del algoritmo normal y optimizado. Podemos observar como la ejecución del algoritmo con las opciones de compilación optimizadas afecta notablemente a la curva, puesto que ahora se realizan las ejecuciones en menos tiempo que en el algoritmo normal. Esto se aprecia mejor en tamaños mayores, donde la función normal se dispara hacia arriba (aumenta mucho el tiempo), mientras que la optimizada sube de forma más constante.



8. Ejercicio 7: Multiplicación matricial

Se ha implementado el siguiente programa en C++, que realiza la multiplicación de dos matrices bidimensionales.

Código 9: multiplicacion_matricial.cpp

```
#include <iostream>
#include <ctime>        // Recursos para medir tiempos
#include <cstdlib>      // Para generacion de nmeros pseudoaleatorios
using namespace std;

void multiplicar(int ** m, int ** m1, int ** m2, int tam){

   for(int i = 0 ; i < tam ; i++)
      for(int j = 0 ; j < tam ; j++)
         for(int k = 0 ; k < tam ; k++)
            m[i][j] += m1[i][k] * m2[k][j];

}

void sintaxis(){

   cerr << "Sintaxis:" << endl;
   cerr << "  TAM: Tamanio de filas y columnas de la matriz (>0)" << endl;
   cerr << "  VMAX: Valor maximo (>0)" << endl;
   cerr << "Se genera una matriz de tamanio TAMxTAM con elementos aleatorios"
        << endl;
   exit(EXIT_FAILURE);

```

```
24 }
25
26 int main(int argc, char ** argv){
27
28    // Lectura de parametros
29    if (argc != 3)
30       sintaxis();
31    int tam = atoi(argv[1]);      // Tamanio de la matriz
32    int vmax = atoi(argv[2]);     // Valor maximo
33    if (tam <= 0 || vmax <= 0)
34       sintaxis();
35
36    // Declaracion de las matrices dinamicas
37    int ** m = new int * [tam];
38    for (int i = 0 ; i < tam ; i++){
39       m[i] = new int [tam];
40    }
41
42    int ** m1 = new int * [tam];
43    for (int i = 0 ; i < tam ; i++){
44       m1[i] = new int [tam];
45    }
46
47    int ** m2 = new int * [tam];
48    for (int i = 0 ; i < tam ; i++){
49       m2[i] = new int [tam];
50    }
51
52    srand(time(0));    // Inicializacion del generador de numeros pseudoaleatorios
53
54    // Recorrer las matrices
55    for (int i = 0 ; i < tam ; i++){
56       for (int j = 0 ; j < tam ; j++){
57          m[i][j] = 0;
58       }
59    }
60
61    for (int i = 0 ; i < tam ; i++){
62       for (int j = 0 ; j < tam ; j++){
63          m1[i][j] = rand() %vmax;  // Generar aleatorio [0,vmax[
64       }
65    }
66
67    for (int i = 0 ; i < tam ; i++){
68       for (int j = 0 ; j < tam ; j++){
69          m2[i][j] = rand() %vmax;  // Generar aleatorio [0,vmax[
70       }
71    }
72
73    clock_t tini;                  // Anotamos el tiempo de inicio
74    tini=clock();
75
76    multiplicar(m, m1, m2, tam);  // Llamada a la funcion que multiplica las dos matrices
```

```
77
78      clock_t tfin;                   // Anotamos el tiempo de finalizacion
79      tfin=clock();
80
81      // Mostramos resultados
82      cout << tam << "\t" << (tfin-tini)/(double)CLOCKS_PER_SEC << endl;
83
84      // Liberacion de memoria dinamica
85      for (int i = 0 ; i < tam ; i++){
86
87          delete [] m1[i];
88          delete [] m2[i];
89
90      }
91      delete [] m1;
92      delete [] m2;
93
94      return 0;
95
96  }
```

A continuación se calcula la eficiencia de este programa.

8.1. Eficiencia teórica

Como podemos observar en el código siguiente:

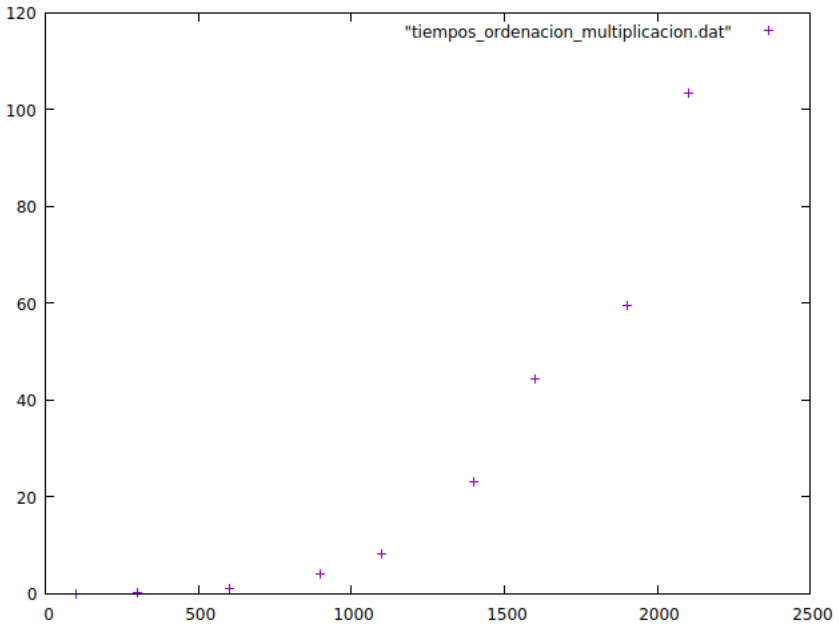
Código 10: función que multiplica dos matrices

```
1  void multiplicar(int ** m, int ** m1, int ** m2, int tam){
2      for(int i = 0 ; i < tam ; i++)
3          for(int j = 0 ; j < tam ; j++)
4              for(int k = 0 ; k < tam ; k++)
5                  m[i][j] += m1[i][k] * m2[k][j];
6  }
```

La función que multiplica las dos matrices bidimensionales consta de tres bucles anidados. La línea 5 es del orden constante O(1) y cada bucle (líneas 2, 3 y 4) es del orden O(n), por tanto la función completa es del orden $O(n^3)$.

8.2. Eficiencia empírica

Se ejecuta el programa de la misma manera que en ejercicios anteriores, con varios tamaños distintos para la matriz. Se representa la gráfica de su eficiencia.

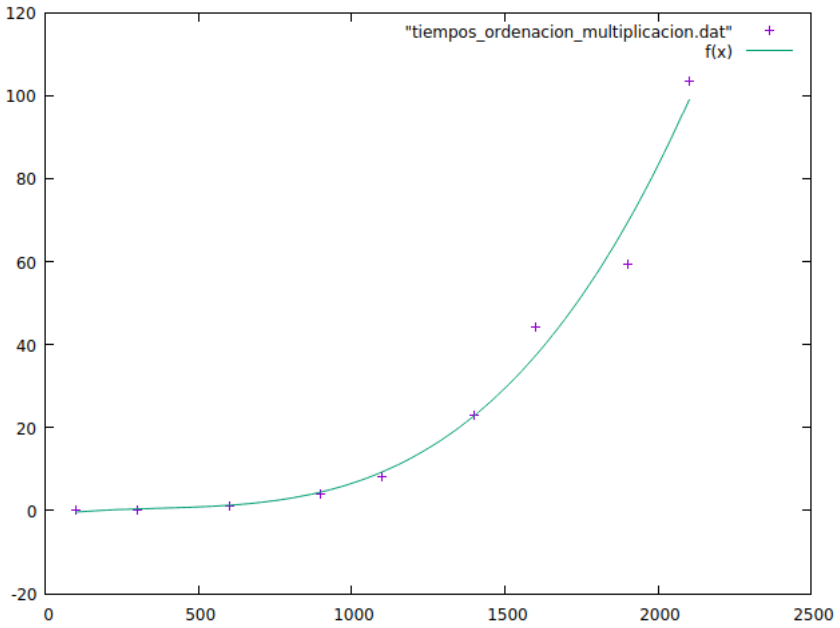


Podemos ver que es una operación con un coste de tiempo muy alto, la gráfica sube de forma muy pronunciada aún aumentando poco el tamaño de la matriz, llegando a tardar más de un minuto para tamaños mayores a 2000.

8.3. Ajuste de la curva teórica a la empírica

A continuación vamos a ajustar una función de regresión a los datos obtenidos anteriormente de las ejecuciones del programa. En este caso, al ser de orden n^3, la forma es una función polinómica de tercer grado, por tanto consideraremos a la función de regresión f(x) de la forma $ax^3 + bx^2 + cx + d$.

A partir de los datos calculados en el apartado anterior, y usando la orden fit de gnuplot, obtenemos los valores de los parámetros y representamos la gráfica.



La gráfica empírica se adapta a la forma de la función de eficiencia teórica.

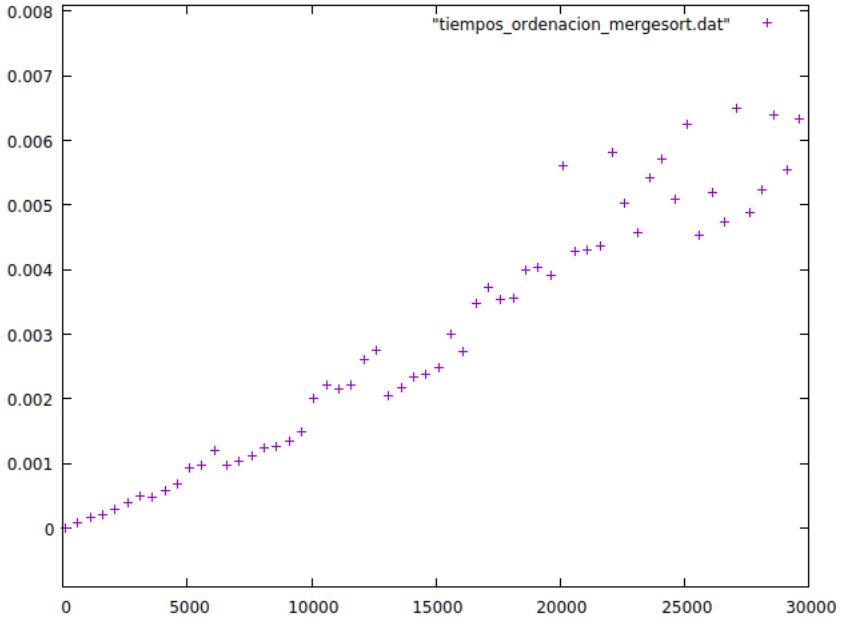
9. Ejercicio 8: Ordenación por mezcla

Se tiene el código de un algoritmo recursivo en el fichero mergesort.cpp, donde se integran dos algoritmos de ordenación: inserción y mezcla (mergesort).

9.1. Eficiencia empírica

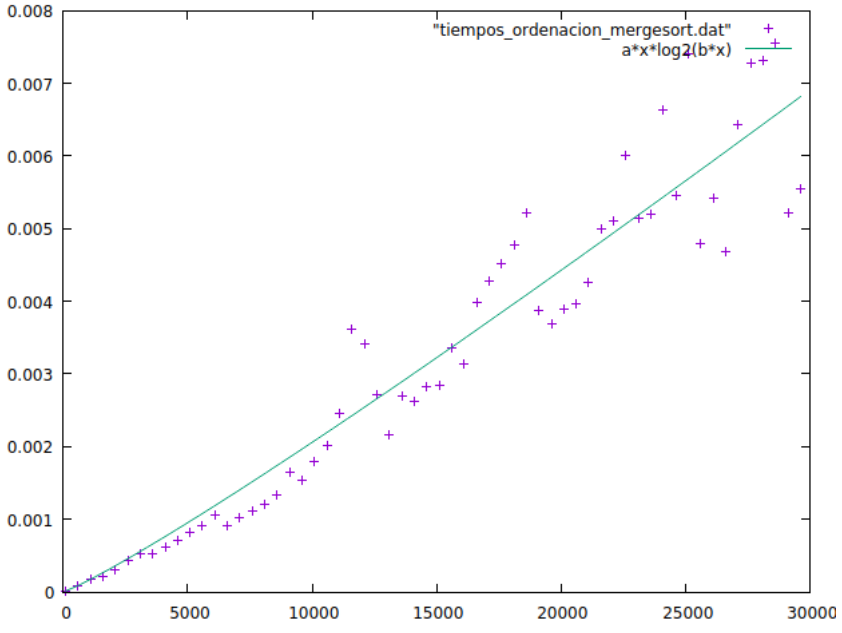
La eficiencia teórica del mergesort es n*log(n). A continuación se estudia la eficiencia empírica como en los ejercicios anteriores, ejecutando el programa varias veces y representando los datos en una gráfica.

Podemos ver como los puntos se dispersan conforme va aumentando el tamaño. Además, es un algoritmo relativamente rápido, que pese a aumentar el tamaño y por tanto el tiempo que tarda en ejecutarse, siguen siendo pocos segundos.



9.2. Ajuste de la curva teórica a la empírica

La gráfica empírica se ajusta a la eficiencia teórica del mergesort que sabemos que es n*log(n).



9.3. Estudio del parámetro UMBRAL_MS

El parámetro UMBRAL_MS condiciona el tamaño mínimo del vector para utilizar el algoritmo de inserción en vez de seguir aplicando de forma recursiva el mergesort. A continuación veremos distintas gráficas de eficiencia empírica para distintos valores de este parámetro.

Conforme aumenta el parámetro se dispersan más los puntos y, como es lógico, aumenta el tiempo que tarda en ejecutarse el algoritmo puesto que el tamaño mínimo de inserción es más grande y se tarda más en dejar de usar de forma recursiva el mergesort.

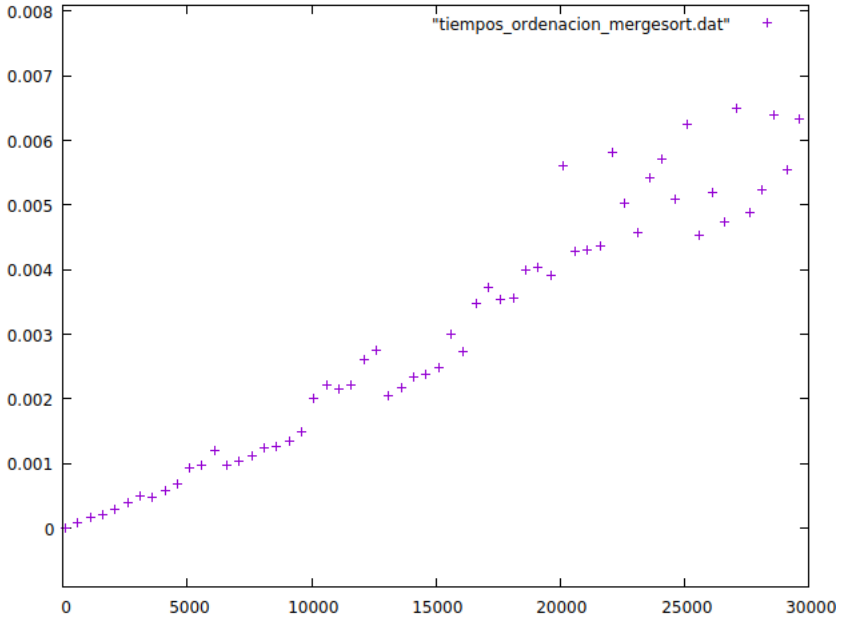


Figura 1: UMBRAL_MS = 100

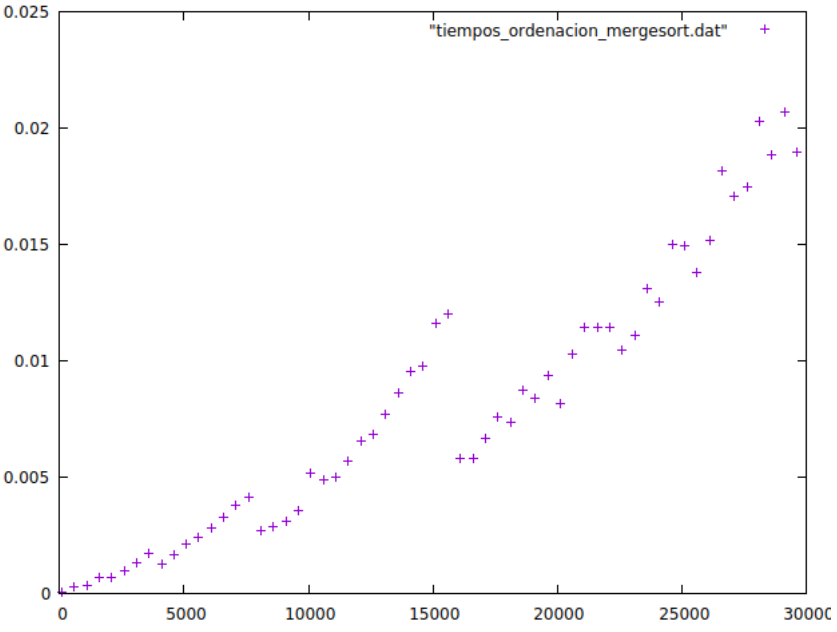


Figura 2: UMBRAL_MS = 500

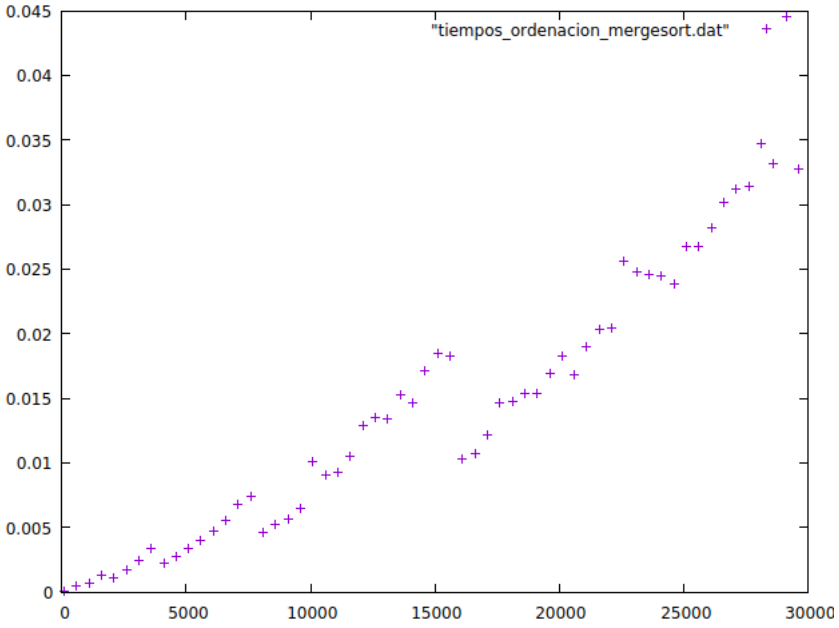


Figura 3: UMBRAL_MS = 1000

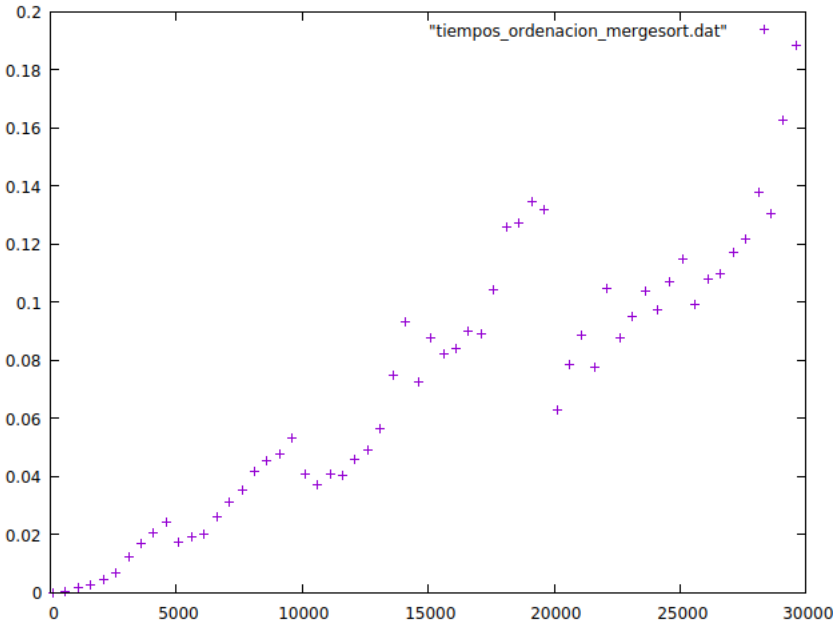


Figura 4: UMBRAL_MS = 5000