Estructura de Datos (2016-2017)
Subgrupo C2
Grado en Ingeniería Informática
Universidad de Granada

Práctica 1: Eficiencia

Mario Rodríguez Ruiz

16 de octubre de 2016

Índice

Índice de figuras

Índice de tablas

1. Información del ordenador

Portatil ASUS X75A

1.1. Hardware

```
[mario@manjario ~]$ cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 58
model name      : Intel(R) Pentium(R) CPU 2020M @ 2.40GHz
stepping        : 9
microcode       : 0x12
cpu MHz         : 1346.812
cache size      : 2048 KB
```

Figura 1.1: Información del CPU.

```
[mario@manjario ~]$ sudo dmidecode --type memory
# dmidecode 3.0
Getting SMBIOS data from sysfs.
SMBIOS 2.7 present.

Handle 0x000B, DMI type 16, 23 bytes
Physical Memory Array
        Location: System Board Or Motherboard
        Use: System Memory
        Error Correction Type: None
        Maximum Capacity: 32 GB
        Error Information Handle: Not Provided
        Number Of Devices: 4

Handle 0x000D, DMI type 17, 34 bytes
Memory Device
        Array Handle: 0x000B
        Error Information Handle: Not Provided
        Total Width: 64 bits
        Data Width: 64 bits
        Size: 4096 MB
        Form Factor: SODIMM
        Set: None
        Locator: ChannelA-DIMM0
        Bank Locator: BANK 0
        Type: DDR3
        Type Detail: Synchronous
        Speed: 1600 MHz
        Manufacturer: Micron
        Serial Number: 00000000
        Asset Tag: 9876543210
        Part Number: 8JTF51264HZ-1G6D 1
        Rank: 1
        Configured Clock Speed: 1600 MHz
```

Figura 1.2: Módulo 1 de memoria RAM.

```
Handle 0x0010, DMI type 17, 34 bytes
Memory Device
        Array Handle: 0x000B
        Error Information Handle: Not Provided
        Total Width: 64 bits
        Data Width: 64 bits
        Size: 4096 MB
        Form Factor: SODIMM
        Set: None
        Locator: ChannelB-DIMM0
        Bank Locator: BANK 2
        Type: DDR3
        Type Detail: Synchronous
        Speed: 1600 MHz
        Manufacturer: Kingston
        Serial Number: 021CC11A
        Asset Tag: 9876543210
        Part Number: 9905469-135.A00LF
        Rank: 1
        Configured Clock Speed: 1600 MHz
```

Figura 1.3: Módulo 2 de memoria RAM.

1.2. Sistema operativo

```
[mario@manjario ~]$ lsb_release -a
LSB Version:    n/a
Distributor ID: ManjaroLinux
Description:    Manjaro Linux
Release:        16.08
Codename:       Ellada
```

Figura 1.4: Información del SO.

1.3. Compilador

```
[mario@manjario ~]$ gcc --version
gcc (GCC) 6.2.1 20160830
Copyright (C) 2016 Free Software Foundation, Inc.
Esto es software libre; vea el código para las condiciones de copia.  NO hay
garantía; ni siquiera para MERCANTIBILIDAD o IDONEIDAD PARA UN PROPÓSITO EN
PARTICULAR
```

Figura 1.5: Información del compilador.

2. Ejercicio 1: Ordenación de la burbuja

2.1. Cálculo de la eficiencia teórica y empírica de/con 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         int aux = v[j];
6         v[j] = v[j+1];
7         v[j+1] = aux;
8       }
9 }
```

- Linea 2: 3 OE (asignación, decremento, comparación) + 1 OE
- Linea 3: 4 OE (asignación, 2 decrementos, comparación) + 1 OE
- Linea 4: 4 OE (2 accesos al elemento v[j], incremento, comparación)
- Linea 5: 2 OE (acceso al elemento v[j], asignación)
- Linea 6: 4 OE (2 accesos al elemento v[j], incremento, asignación)
- Linea 7: 3 OE (acceso al elemento v[j], incremento, asignación)

Bucle de la j

$$Tj(n) = \sum_{j=0}^{n-i-1} 17 = 17n - 17i \tag{2.1}$$

Bucle de la i

$$Ti(n) = \sum_{i=0}^{n-1}(1 + 17n - 17i) = \sum_{i=0}^{n-1} 1 + \sum_{i=0}^{n-1} 17n - \sum_{i=0}^{n-1} 17i = n + 17n^2 + 17\frac{n(n+1)}{2} \tag{2.2}$$

$$Ti(n) = \frac{1}{2}(51n^2 + 19n) \tag{2.3}$$

Por lo que el orden de eficiencia de la función es de $O(n^2)$ (2.4)

2.2. Experimento con el algoritmo de la burbuja

```
1  #include <iostream>
2  #include <ctime>    // Recursos para medir tiempos
3  #include <cstdlib>  // Para generación de números
       pseudoaleatorios
4
5  using namespace std;
6
7  void ordenar(int *v, int n) {
8      for (int i=0; i<n-1; i++)
9          for (int j=0; j<n-i-1; j++)
10             if (v[j]>v[j+1]) {
11                 int aux = v[j];
12                 v[j] = v[j+1];
13                 v[j+1] = aux;
14             }
15 }
16
17 void sintaxis()
18 {
19   cerr << "Sintaxis:" << endl;
20   cerr << "  TAM: Tamaño del vector (>0)" << endl;
21   cerr << "  VMAX: Valor máximo (>0)" << endl;
22   cerr << "Se genera un vector de tamaño TAM con elementos
       aleatorios en [0,VMAX[" << endl;
23   exit(EXIT_FAILURE);
24 }
25
26
27 int main(int argc, char * argv[])
28 {
29   // Lectura de parámetros
30   if (argc!=3)
31     sintaxis();
32   int tam=atoi(argv[1]);      // Tamaño del vector
33   int vmax=atoi(argv[2]);     // Valor máximo
34   if (tam<=0 || vmax<=0)
35     sintaxis();
36
37   // Generación del vector aleatorio
38   int *v=new int[tam];        // Reserva de memoria
```

```
39    srand(time(0));             // Inicialización del generador
         de números pseudoaleatorios
40    for (int i=0; i<tam; i++)   // Recorrer vector
41      v[i] = rand() % vmax;     // Generar aleatorio [0,vmax[
42
43    clock_t tini;     // Anotamos el tiempo de inicio
44    tini=clock();
45
46    ordenar(v, tam) ;
47
48    clock_t tfin;     // Anotamos el tiempo de finalización
49    tfin=clock();
50
51    // Mostramos resultados
52    cout << tam << "\t" << (tfin-tini)/(double)CLOCKS_PER_SEC
         << endl;
53
54    delete [] v;      // Liberamos memoria dinámica
55  }
```

ejercicio1/ordenacion.cpp

```
1  #!/bin/csh
2  @ inicio = 100
3  @ fin = 30000
4  @ incremento = 100
5  set ejecutable = ordenacion
6  set salida = tiempos_ordenacion.dat
7
8  @ i = $inicio
9  echo > $salida
10 while ( $i <= $fin )
11   echo Ejecución tam = $i
12   echo `./{$ejecutable} $i 10000` >> $salida
13   @ i += $incremento
14 end
```

ejercicio1/ejecuciones_ordenacion.csh

Los resultados, una vez analizados y procesados por gnuplot son los mostrados en la Figura 2.1. Como se puede comprobar, el gráfico muestra un resultado en forma parabólica, coincidiendo con el estudio teórico.

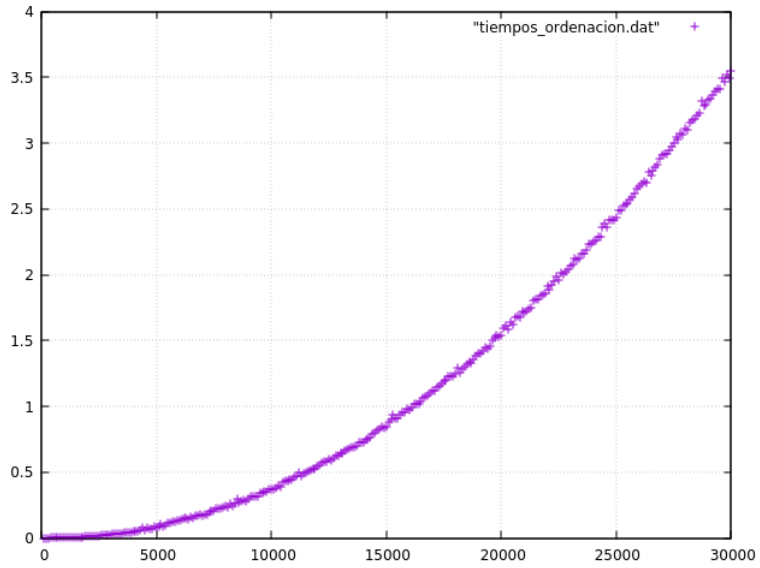

Figura 2.1: Datos dibujados con Gnuplot

Lo que sucede al superponer ambas eficiencias es que los resultados de ambos coinciden completamente tal y como se puede comprobar en la Figura 2.2.

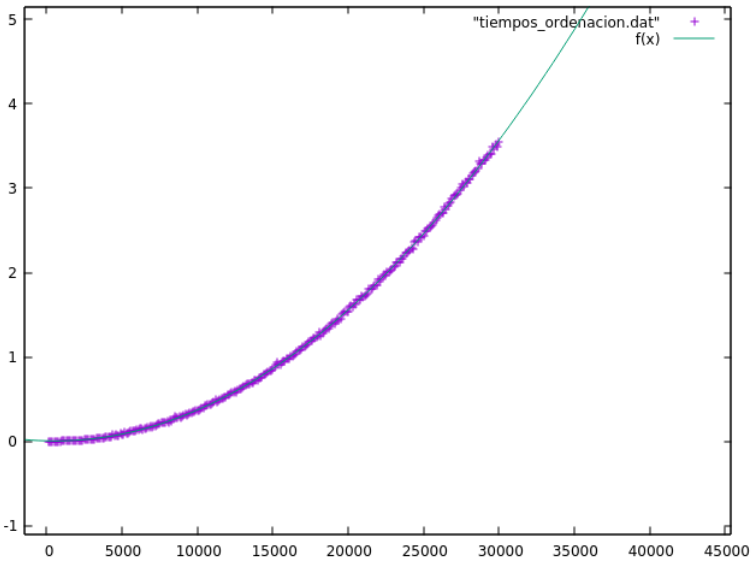

Figura 2.2: Superposición eficiencia teórica y empírica

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

```
Final set of parameters            Asymptotic Standard Error
======================             ==========================
a               = 4.10446e-09      +/- 1.11e-11     (0.2704%)
b               = -4.5291e-06      +/- 3.449e-07    (7.616%)
c               = 0.00557817       +/- 0.002248     (40.3%)

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>
```

Figura 3.1: Cálculo de a, b y c por gnuplot

Replicando el experimento anterior, esta vez antes hemos obtenido los valores de a, b y c por medio de gnuplot para que, a continuación, como se ve en la Figura 3.2 superponer ambas eficiencias.

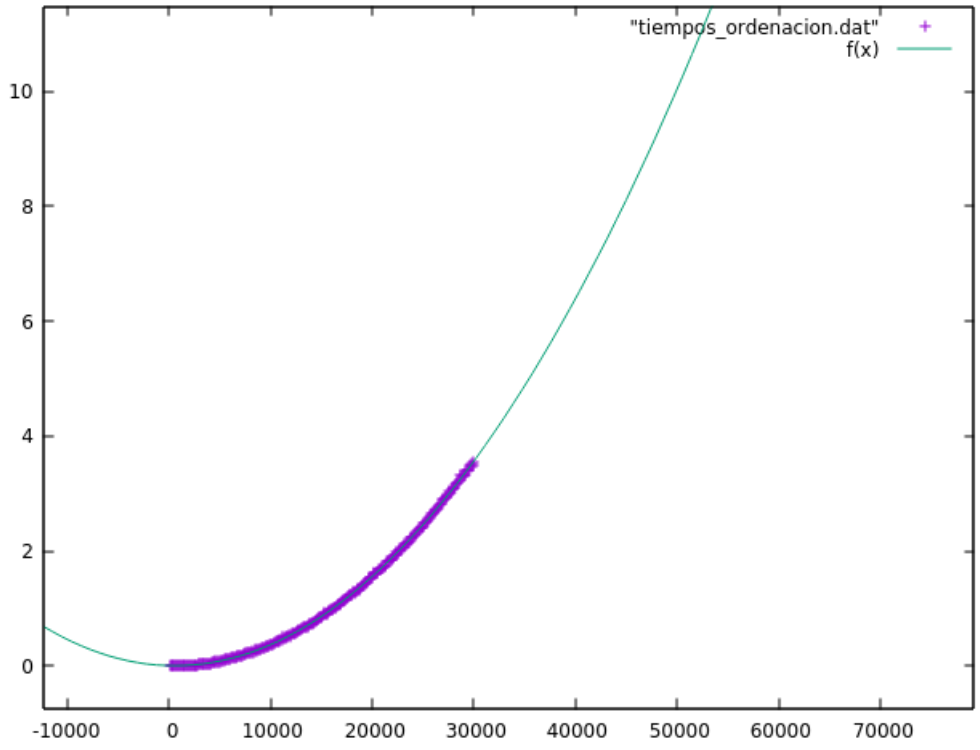

Figura 3.2: Superposición eficiencia teórica y empírica

4. Ejercicio 3: Problemas de precisión

4.1. Qué hace el algoritmo

El algoritmo es un común de búsqueda binaria. Este, para su correcto funcionamiento, requiere un vector ordenado en el que busca un valor concreto.
La característica fundamental de éste es que en cada iteracción va descartando la mitad del vector en la que no se encuentra el valor especificado. Su funcionamiento es el siguiente: compara el valor especificado con el valor del centro del vector; si es mayor el del centro, el valor está en la parte izquierda, en caso contrario estará en la derecha. De esta forma en cada iteración se elimina la mitad del vector restante.

4.2. Eficiencia teórica

Para este estudio solo es necesario hacer énfasis en la condición del bucle ya que el resto de instrucciones son de orden constante, por lo que se pueden obviar.
En cuanto a la condición, se tiene que

```
while ((inf<sup) && (!enc))
```

siendo el peor de los casos **(inf<sup)**, lo que produce que `inf==sup`, es decur, sólamente es necesario quedarse con un elemento.
Teniendo un vector de `n elementos`, en la primera iteración se obtendría n, pero en las siguientes iteraciones se tendría `n/2, n/4, n/8, n/16`... así sucesivamente hasta llegar a la iteración `i-esima`. En esta última se obtendrá que $\mathbf{n/2i <= 1}$ elementos.
Por tanto, el número de iteraciones obtenida a raiz de la fórmula obtenida es:

$$\mathbf{n <= 2i}, \text{ que en forma logarítmica será: } \mathbf{log2\ n <= i}$$

Es decir, el bucle se recorrerá en el peor de los casos i veces, siendo i el logaritmo en base 2 de n.
Finalmente, se concluye que el algoritmo tiene un orden de complejidad **O(log n)**.

4.3. Eficiencia empírica

Un problema que puede ocasionar un algoritmo tan eficiente como éste es la dificultad para medir tiempos tan bajos. Una solución podría ser la utilización de vectores muy grandes, pero aún así el algoritmo lo solucionaría de forma idéntica. Lo que además ocurre es que para tamaños pequeños el tiempo aumenta muy rápido y, para tamaños grandes muy despacio. Por ello es que se tengan que hacer esos incrementos siempre en función del tamaño.
Para realizar una medición que solucione los problemas anteriores se proponen dos cambios:

1. Uso de la clase **high_resolution_clock** de la biblioteca <**chrono**>

   Ésta permite hacer mediciones en **nanosegundos**.

```cpp
#include <iostream>
#include <ctime>    // Recursos para medir tiempos
#include <chrono>   // Medir tiempos en nanosegundos.
#include <cstdlib>  // Números pseudoaleatorios

using namespace std;

int operacion(int *v, int n, int x, int inf, int sup) {
  int med;
  bool enc=false;
  while ((inf<sup) && (!enc)) {
    med = (inf+sup)/2;
    if (v[med]==x)
      enc = true;
    else if (v[med] < x)
      inf = med+1;
    else
      sup = med-1;
  }
  if (enc)
    return med;
  else
    return -1;
}

void sintaxis()
{
  cerr << "Sintaxis:" << endl;
  cerr << "  TAM: Tamaño del vector (>0)" << endl;
  cerr << "Se genera un vector de tamaño TAM con
     elementos aleatorios" << endl;
  exit(EXIT_FAILURE);
}

int main(int argc, char * argv[])
{
  // Lectura de parámetros
  if (argc!=2)
    sintaxis();
  int tam=atoi(argv[1]);     // Tamaño del vector
  if (tam<=0)
    sintaxis();

```

```
43   // Generación del vector aleatorio
44   int *v=new int[tam];        // Reserva de memoria
45   srand(time(0));             // Inicialización del
        generador de números pseudoaleatorios
46   for (int i=0; i<tam; i++)  // Recorrer vector
47     v[i] = rand() % tam;
48
49   // Anotamos el tiempo de inicio
50   auto tini = chrono::high_resolution_clock::now();
51   // Algoritmo a evaluar
52   operacion(v,tam,tam+1,0,tam-1);
53   // Anotamos el tiempo de finalización
54   auto tfin = chrono::high_resolution_clock::now();
55
56   // Calculamos el tiempo transcurrido y mostramos
        resultados
57   cout << tam << "\t" << chrono::duration_cast < chrono::
        nanoseconds>(tfin - tini).count() << endl;
58
59   delete [] v;      // Liberamos memoria dinámica
60 }
```

ejercicio3/ejercicio_desc.cpp

2. Ejecución mediante un script que incrementa los tamaños lentamente al principio y brúscamente, al superar el crecimiento exponencial de la curva.

```
1  #!/bin/csh
2  @ inicio = 10
3  @ fin = 599990000
4  @ incremento1 = 10000
5  @ incremento2 = 10000000
6  @ i = $inicio
7  echo > tiempos_desc.dat
8  while ( $i <= $fin )
9    echo Ejecución tam = $i
10   echo `./ejercicio_desc $i` >> tiempos_desc.dat
11   if ($i<1000000) then
12     @ i += $incremento1
13   else
14     @ i += $incremento2
15   endif
16 end
```

ejercicio3/ejecuciones_desc.csh

4.4. Ajuste de la eficiencia teórica

Para calcular a y b (Figura 4.1) en este apartado se ejecuta lo siguiente en **gnuplot**:
f(x) = a*log(x) + b

```
Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a               = 13.1733          +/- 0.3254       (2.47%)
b               = -18.216          +/- 5.048        (27.71%)

correlation matrix of the fit parameters:
                a      b
a               1.000
b              -0.976  1.000
gnuplot>
```

Figura 4.1: Cálculo de a y b

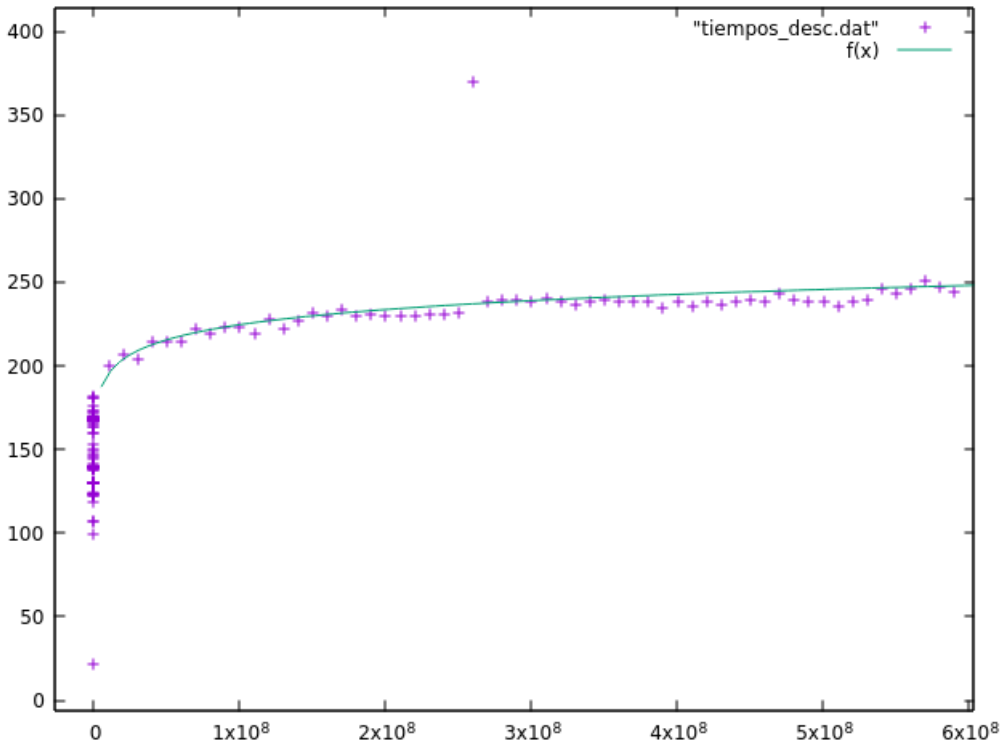

Figura 4.2: Regresión para ajustar la curva teórica a la empírica.

5. Ejercicio 4: Mejor y peor caso

Modificación del código para crear una situación de dos posibles casos.

5.1. El mejor caso posible

Este caso es en el que el vector ya se encuentre ordenado. Para ello sólamente hay que modificar en el código la parte donde se rellena el vector después de su obligatoria reserva de espacio.

```
1 // Generación del vector ORDENADO
2 int *v=new int[tam];        // Reserva de memoria
3 for (int i=0; i<tam; i++)   // Recorrer vector
4   v[i] = i;     // Generar vector ya ordenado
```

– **Eficiencia empírica**

```
Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a               = 1.62132e-09      +/- 1.05e-11     (0.6473%)
b               = 2.72897e-06      +/- 3.262e-07    (11.95%)
c               = -0.00713629      +/- 0.002126     (29.79%)

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>
```

Figura 5.1: Cálculo de los parámetros.

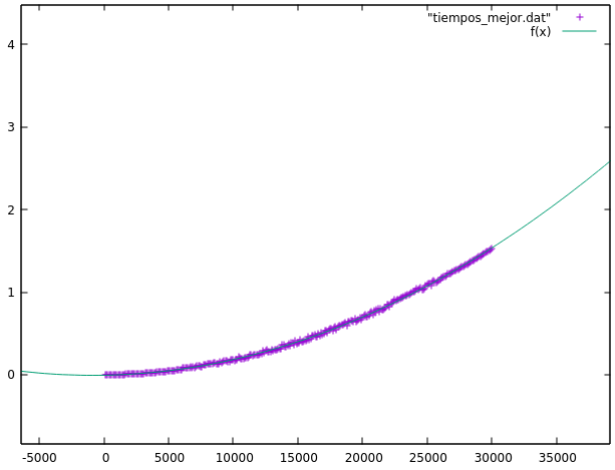

Figura 5.2: Gráfica de la eficiencia empírica

5.2. El peor caso posible

Este caso es en el que el vector ya se encuentre ordenado pero de manera inversa, es decir, el tamaño de los valores del vector irán de más a menos. Para ello sólamente hay que modificar en el código la parte donde se rellena el vector después de su obligatoria reserva de espacio.

```
1 // Generación del vector ordenado del revés
2 int *v=new int[tam];        // Reserva de memoria
3 // Recorrer vector
4 for (int i=tam-1, j = 0 ; i >= 0 ; i--, j++)
5   v[i] = j;     // Generar vector ya ordenado
```

– **Eficiencia empírica**

```
Final set of parameters            Asymptotic Standard Error
======================             ==========================
a               = 3.04566e-09      +/- 1.365e-11    (0.4483%)
b               = 3.95462e-06      +/- 4.244e-07    (10.73%)
c               = -0.00780301      +/- 0.002766     (35.45%)

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>
```

Figura 5.3: Cálculo de los parámetros.

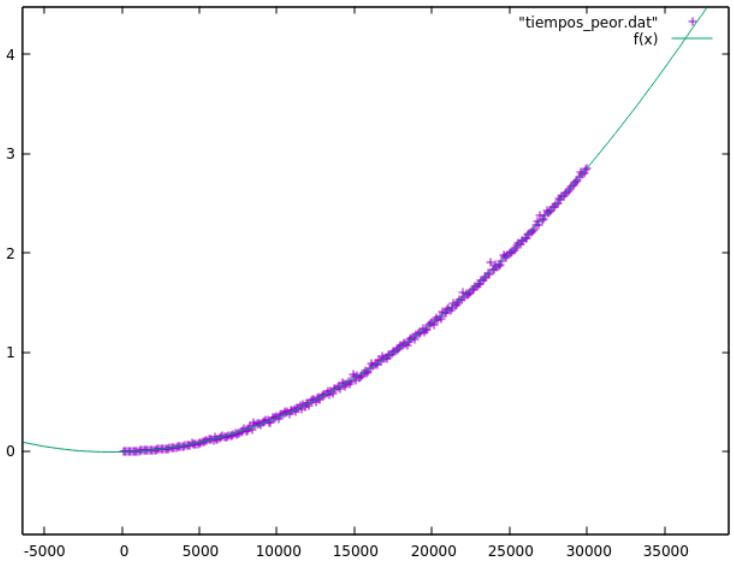

Figura 5.4: Gráfica de la eficiencia empírica

5.3. Comparación de resultados

Una vez obtenidos todos los resultados empíricos de la eficiencia se puede apreciar en la Figura 5.5, como era de esperar, que los tiempos con los resultados más eficientes son las del mejor caso y los menos, las del peor caso.

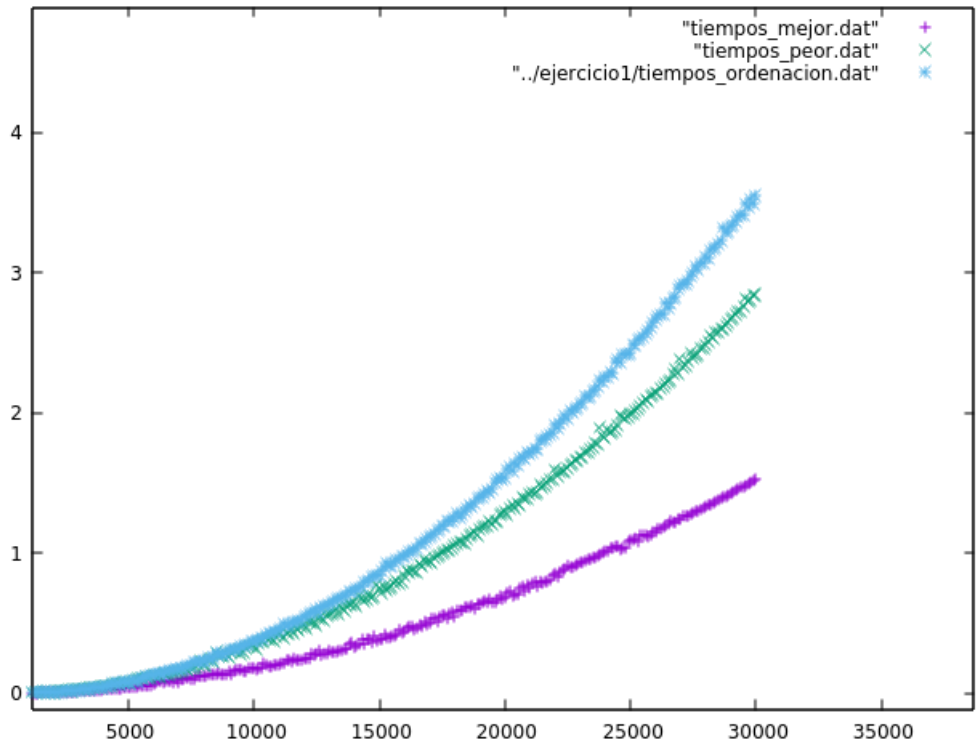

Figura 5.5: Comparación de resultados.

6. Ejercicio 5: Dependencia de la implementación

6.1. Cálculo de la eficiencia teórica del nuevo 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              /*Aquí nunca llegará con el bucle ordenado en el
                   mejor caso*/
8              }
9      }
10 }
```

- Linea 2: 1 OE (asignación)
- Linea 3: 4 OE (asignación, decremento, comparación, operación &&) + 1 OE
- Linea 4: 1 OE (asignación)
- Linea 5: 4 OE (asignación, 2 decrementos, comparación) + 1 OE
- Linea 6: 4 OE (2 accesos al elemento v[j], incremento, comparación)

Al añadirse la variable booleana **cambio** se produce que, en el mejor de los casos, el orden de eficiencia teórico sea **O(n)**. Dicha variable sería siempre **false**, ya que la comparación que le precede también lo será por lo que el bucle externo se ejecuta una sola vez.
A continuación se realizará el estudio teórico del algoritmo.
Bucle de la j

$$Tj(n) = \sum_{j=0}^{n-i-1} 9 = 9n - 9i \tag{6.1}$$

$$T(n) = 1 + \sum_{i=0}^{1}(6 + 9n - 9i) = 6 + \sum_{i=0}^{1}(9n) - \sum_{i=0}^{1}(9i) = 6 + 9n - 9 = 9n - 3 \tag{6.2}$$

Por lo que se puede confirmar que la eficiencia de la función es de **O(n)**

6.2. Cálculo de la eficiencia empírica del nuevo algoritmo de la burbuja

```
Final set of parameters            Asymptotic Standard Error
======================             ==========================
a               = 3.43556e-09      +/- 2.407e-11    (0.7006%)
b               = 9.78149e-07      +/- 4.18e-07     (42.73%)

correlation matrix of the fit parameters:
                a      b
a               1.000
b              -0.867  1.000
gnuplot>
```

Figura 6.1: Cálculo de los parámetros.

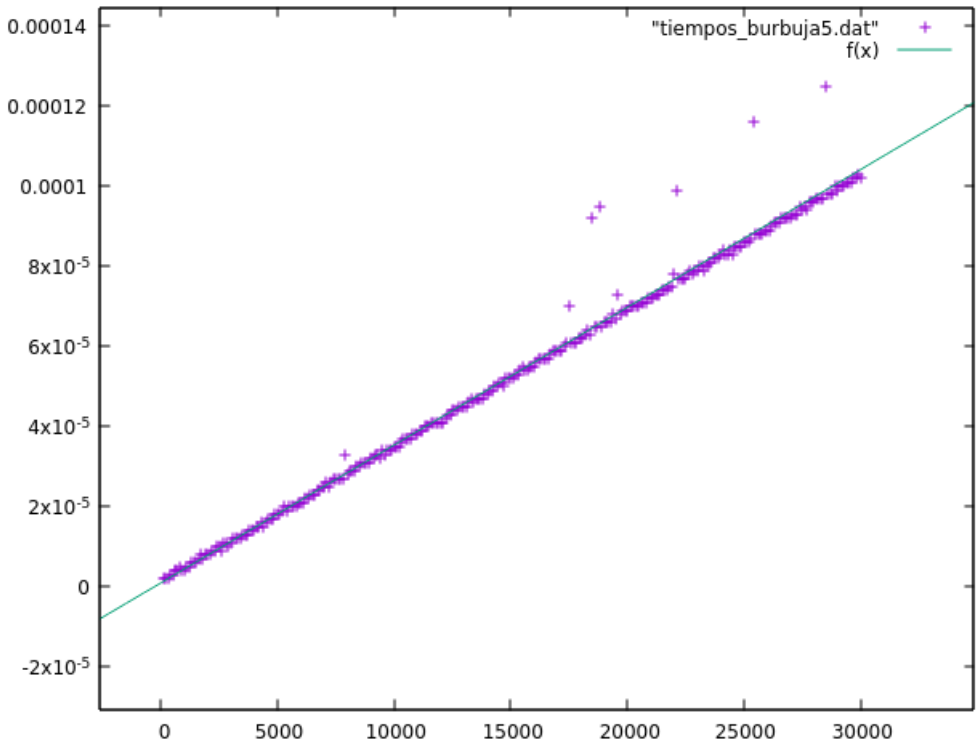

Figura 6.2: Gráfica de la eficiencia empírica.

7. ADICIONAL - Ejercicio 6: Influencia del proceso de compilación

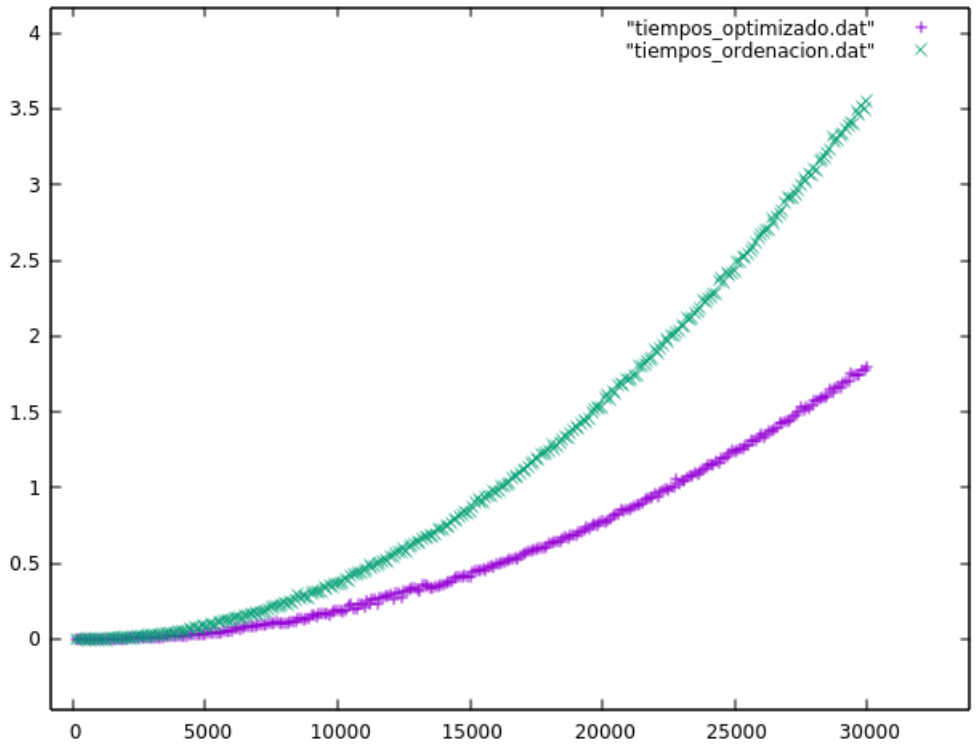

Figura 7.1: Compilación normal VS optimizada.

8. ADICIONAL - Ejercicio 7: Multiplicación matricial

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

using namespace std;

void reservaMemoriaMatriz(int **&M, const int tam){
    M = new int*[tam];
    for(int i = 0; i < tam; i++)
        M[i] = new int[tam] ;
}

// Relleno de matriz con valores aleatorios
void rellenaMatriz(int **&M, const int tam, const int vmax){
    for(int i = 0; i < tam; i++)
        for(int j=0; j < tam; j++)
            M[i][j] = rand() % vmax ;
}

void imprimeMatriz(int **M, const int tam){
    for(int i=0; i < tam; i++){
        for(int j=0; j < tam; j++){
            cout <<  M[i][j] << " ";
        }
        cout << endl;
    }
}

void sintaxis()
{
    cerr << "Sintaxis:" << endl;
    cerr << "  TAM: Tamaño de la matriz (>0)" << endl;
    cerr << "  VMAX: Valor máximo (>0)" << endl;
    cerr << "Se genera un vector de tamaño TAM con elementos aleatorios en [0,VMAX[" << endl;
    exit(EXIT_FAILURE);
}

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

```
40      // Lectura de parámetros
41      if (argc!=3)
42          sintaxis();
43      int tam=atoi(argv[1]);      // Tamaño del vector
44      int vmax=atoi(argv[2]);     // Valor máximo
45      if (tam<=0 || vmax<=0)
46          sintaxis();
47
48      // Generación de las matrices con valores aleatorios
49      int **A ;
50      int **B ;
51      int **R ;
52
53      reservaMemoriaMatriz(A, tam) ;
54      rellenaMatriz(A, tam, vmax) ;
55      reservaMemoriaMatriz(B, tam) ;
56      rellenaMatriz(B, tam, vmax) ;
57      reservaMemoriaMatriz(R, tam) ;
58
59      srand(time(0));             // Inicialización del
            generador de números pseudoaleatorios
60
61      clock_t tini;     // Anotamos el tiempo de inicio
62      tini=clock();
63
64      // Multiplicación de A*B guardada en R
65      for(int i = 0; i < tam; i++)
66          for(int j = 0; j < tam; j++){
67              R[i][j] = 0;
68              for(int k =0 ; k < tam; k++)
69                  R[i][j]= R[i][j]+A[i][k]*B[k][j];
70          }
71
72      clock_t tfin;     // Anotamos el tiempo de finalización
73      tfin=clock();
74
75      //imprimeMatriz(R, tam) ;
76
77      // Mostramos resultados
78      cout << tam << "\t" << (tfin-tini)/(double)CLOCKS_PER_SEC
            << endl;
79
80      // Liberamos memoria dinámica
81      for(int i=0; i < tam; i++){
```

```
82          delete []A[i] ;
83          delete []B[i] ;
84          delete []R[i] ;
85      }
86
87      delete [] A;
88      delete [] B;
89      delete [] R;
90  }
```

ejercicio7/ejercicio7_matrices.cpp

8.1. Eficiencia teórica

```
1  for(int i = 0; i < tam; i++)
2      for(int j = 0; j < tam; j++){
3          R[i][j] = 0;
4          for(int k =0 ; k < tam; k++)
5              R[i][j]= R[i][j]+A[i][k]*B[k][j];
6      }
```

- Linea 1: 2 OE (asignación, comparación) + 1 OE
- Linea 2: 2 OE (asignación, comparación) + 1 OE
- Linea 3: 2 OE (acceso al elemento R[i][j], asignación)
- Linea 4: 2 OE (asignación, comparación) + 1 OE
- Linea 5: 7 OE (4 accesos al elementos de las matrices, 2 operaciones, asignación)

Se tienen tres bucles anidados, haciendo cada uno **n iteraciones**. El resto de la función es de orden constante, por lo que puede identificarse cada una como **O(1)**. De esta forma, el tiempo de ejecución en función de n es:

$$T(n) = \sum_{i=0}^{n}\sum_{j=0}^{n}\sum_{k=0}^{n} 1 = \sum_{i=0}^{n}\sum_{j=0}^{n} n = n^3 \tag{8.1}$$

Por lo que se puede confirmar que la eficiencia de la función es de

$$O(n^3) \tag{8.2}$$

8.2. Eficiencia empírica

```
#!/bin/csh
@ inicio = 3
@ fin = 1000
@ incremento = 1
set ejecutable = ejercicio7_matrices
set salida = tiempos_matrices.dat

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

ejercicio7/ejecuciones_matrices.csh

Al ser una función cúbica, como se descubrió anteriormente, se realiza la regresión a partir de los parámetros obtenidos de la funcion

$$f(x) = ax^3 + bx^2 + cx + d \tag{8.3}$$

```
Final set of parameters            Asymptotic Standard Error
======================             ==========================
a               = 1.12276e-08      +/- 5.33e-10     (4.747%)
b               = -1.97302e-06     +/- 4.88e-07     (24.73%)
c               = 0.000287599      +/- 0.0001267    (44.06%)
d               = -0.00945425      +/- 0.008846     (93.56%)

correlation matrix of the fit parameters:
                a      b      c      d
a               1.000
b              -0.986  1.000
c               0.918 -0.969  1.000
d              -0.674  0.756 -0.873  1.000
gnuplot>
```

Figura 8.1: Obtención de parámetros

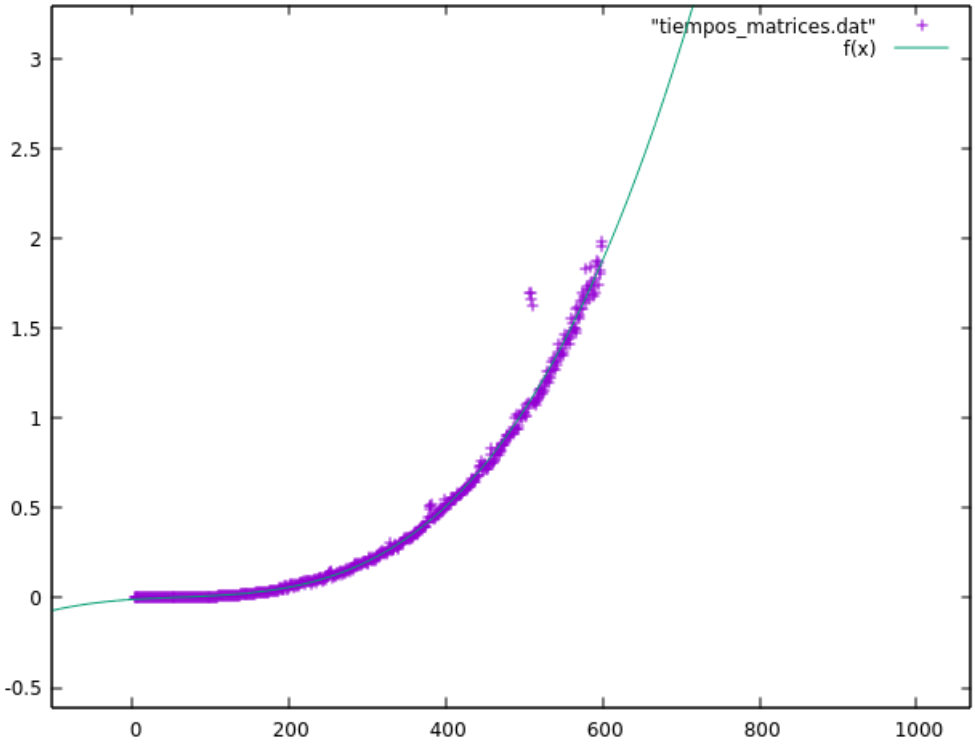

Figura 8.2: Gráfico de la eficiencia empírica