Práctica 1

Emilio José Hoyo Medina
Stefan Parvanov

8 de octubre de 2017

1. Condiciones de ejecución

Los ejercicios de esta práctica se han ejecutado en ordenadores con las siguientes prestaciones:

- Ejercicios 1 a 4:
  Sistema Operativo: Mac OS 10.12.6

  CPU: Intel Core i5 @1.6GHz
  Memoria: 8GB ddr3 @1600Mhz
  Disco duro: 128gb SSD

- Ejercicios 5 a 8:
  Sistema Operativo: Ubuntu 17.04

  CPU: Intel Core i7 @2.5GHz x 4
  Memoria: 15.6GB
  Disco duro: 1 TB HDD

2. Opciones de compilación

Para todos los ejercicios se ha utilizado el siguiente formato de compilación:

```
g++ <nombrefichero>.cpp -o <nombrefichero>
```

Excepto para el 6, que se ha utilizado el siguiente:

```
g++ -O3 <nombrefichero>.cpp -o <nombrefichero>
```

3. Ejercicio 1 : Ordenación de la burbuja

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               int aux = v[j];
6               v[j] = v[j+1];
7               v[j+1] = aux;
8                 }
9       }
```

Calcule la eficiencia teórica de este algoritmo. A continuación replique el experimento que se ha hecho antes (búsqueda lineal) con este nuevo código. Debe:

- Crear un fichero ordenacion.cpp con el programa completo para realizar una ejecución del algoritmo.
- Crear un script ejecuciones_ordenacion.csh en C-Shell que permite ejecutar varias veces el programa anterior y generar un fichero con los datos obtenidos.
- Usar gnuplot para dibujar los datos obtenidos en el apartado previo

Los datos deben contener tiempos de ejecución para tamaños del vector 100, 600, 1100, ...,30000. Pruebe a dibujar superpuestas la función con la eficiencia teórica y la empírica. ¿Qué sucede?

Empezamos viendo que el bucle interior se ejecuta $\frac{(n-1)(n)}{2}$ veces. En cada iteración se ejecutan las 4 OE de la condición del if (incremento, dos accesos a memoria y una comparación) y en el interior del bloque if siempre se ejecutarán 7 OE. El bucle exterior se ejecuta $(n-1)$ veces y el interior se ejecuta $\frac{(n-1)(n)}{2}$ y en cada bucle exterior se ejecutan las 3OE de la condicion del bucle(decremento, comparación, incremento) más 3 OE de la inicialización del bucle interior(asignación, resta, comparacion). Idénticamente en el bucle interior se ejecutan 3 OE del propio bucle y 3 OE del cuerpo. Además añadimos las 3 OE de la inicialización del bucle exterior:

$$T(n) = 3 + (3+3)(n-1) + \frac{(3+3+7)(n-1)(n)}{2} = \frac{13n^2 - n - 6}{2} \tag{1}$$

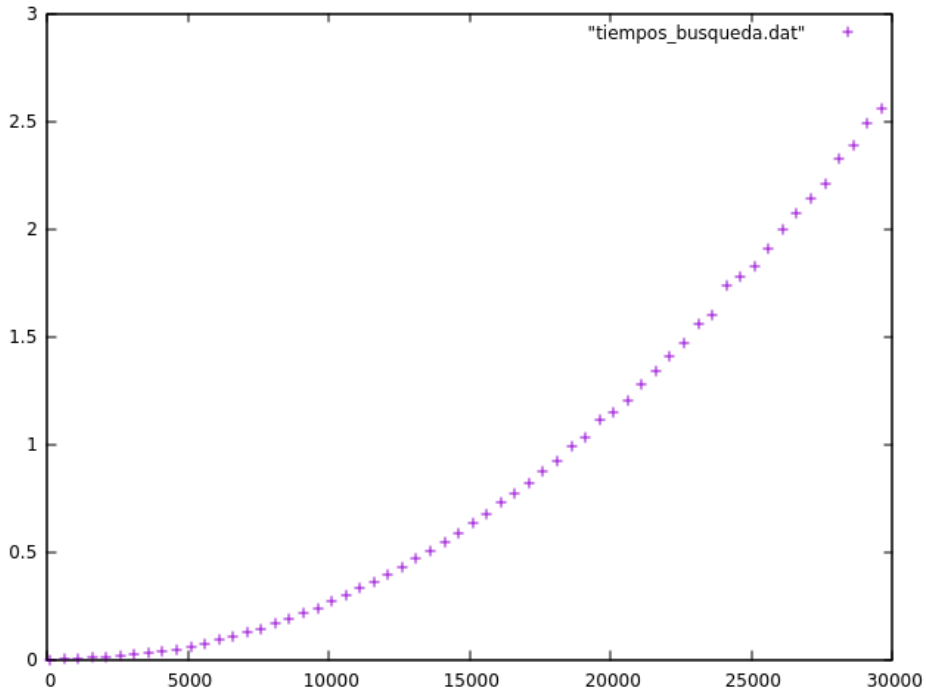
Nos tenemos que fijar que tenemos que dividir el número de operaciones realizadas por la frecuencia del procesador para obtener la función de tiempo.

$$T(n) = \frac{13n^2 - n - 6}{2 * 1{,}6 * 10^9} = \frac{0{,}4}{10^9}n^2 - \frac{0{,}31}{10^9}n - \frac{1{,}88}{10^9} \tag{2}$$

$$T(n) \in O(n^2) \tag{3}$$

Representamos los datos empíricos mediante gnuplot:

Figura 1: Eficiencia empírica



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

Replique el experimento de ajuste por regresión a los resultados obtenidos en el ejercicio 1 que calculaba la eficiencia del algoritmo de ordenación de la burbuja. Para ello considere que f(x) es de la forma ax2+bx+c

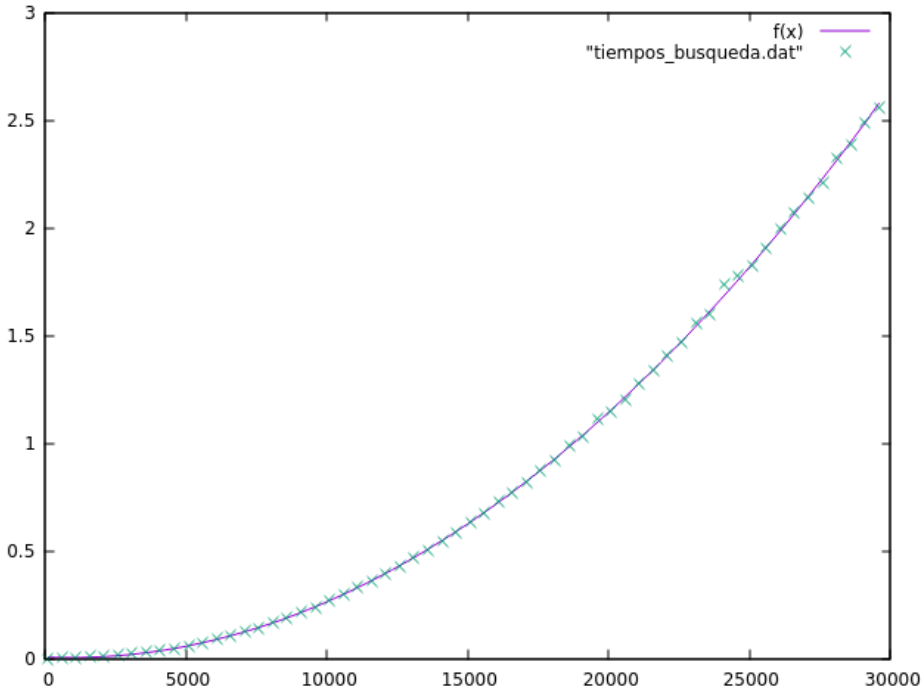
```
1  gnuplot> f(x) = a*x**2 + b*x + c
2  gnuplot> fit f(x) 'tiempos_ordenacion.dat' via a,b,c
3
4  ...............................................................
5
6  After 12 iterations the fit converged.
7  final sum of squares of residuals : 0.00377746
8  rel. change during last iteration : -1.08711e-10
9
10 degrees of freedom    (FIT_NDF)                        : 57
11 rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 0.00814071
12 variance of residuals (reduced chisquare) = WSSR/ndf   : 6.62712e-05
13
14 Final set of parameters            Asymptotic Standard Error
15 ======================            ==========================
16 a                = 1.38647e-09     +/- 1.568e-11    (1.131%)
17 b                = 8.02816e-07     +/- 4.812e-07    (59.94%)
18 c                = 0.00179818      +/- 0.003092     (171.9%)
19
20 correlation matrix of the fit parameters:
21                      a      b      c
22 a                  1.000
23 b                 -0.968  1.000
24 c                  0.738 -0.861  1.000
```

Por tanto mediante gnuplot hemos obtenido las siguiente función de ejecución:

$$T(n) = \frac{1{,}39x^2}{10^9} + \frac{8{,}03x}{10^7} + 0{,}0018 \tag{4}$$

Representando ahora conjuntamente los datos obtenidos y la función ajustada:

Figura 2: Eficiencia empírica



5. Ejercicio 3 : Problemas de precisión

Junto con este guión se le ha suministrado un fichero ejercicio_desc.cpp. En él se ha implementado un algoritmo. Se pide que:

- Explique qué hace este algoritmo.
- Calcule su eficiencia teórica.
- Calcule su eficiencia empírica.

Si visualiza la eficiencia empírica debería notar algo anormal. Explíquelo y proponga una solución. Compruebe que su solución es correcta. Una vez resuelto el problema realice la regresioón para ajustar la curva teórica a la empírica.

Se trata de una búsqueda binaria. Lo que hace es buscar un valor dentro de un vector subdividiendo el vector en dos partes hasta encontrarlo. Esto solo se puede aplicar a vectores ordenados, ya que el algoritmo se basa en que si el valor no esta entre las cotas de un intervalo de valores del vector no puede estar en esas posiciones del vector.

Para calcular la eficiencia teórica empezamos fijandonos que el bucle principal se ejecuta $log_2(n)$ veces. Esto es asi porque podemos dividir un número entre 2 como mucho $log_2(n)$ veces. Antes de comenzar el bucle se ejecuta 4 OE (la asignación en linea 1, las comparaciones del inicio del bucle y el AND lógico). Además dentro del bucle siempre se ejecutaran las 3 OE de la condicion y 3 OE del calculo del índice medio. Aplicando la regla de la suma en los bloques condicionales obtenemos que el máximo son 4 OE (acceso a memoria, comparación, incremento y asignación) en la segunda sentencia else if. En total en el cuerpo del bucle se ejecutaran como mucho 10 OE. Al terminar el bucle se ejecutan 2 OE más. En total tenemos:

$$T(n) = 4 + (3 + 3 + 4)log_2(n) + 2 = 6 + 10log_2(n) \tag{5}$$

Dividimos por la frecuencia del procesador para hallar el valor en segundos:

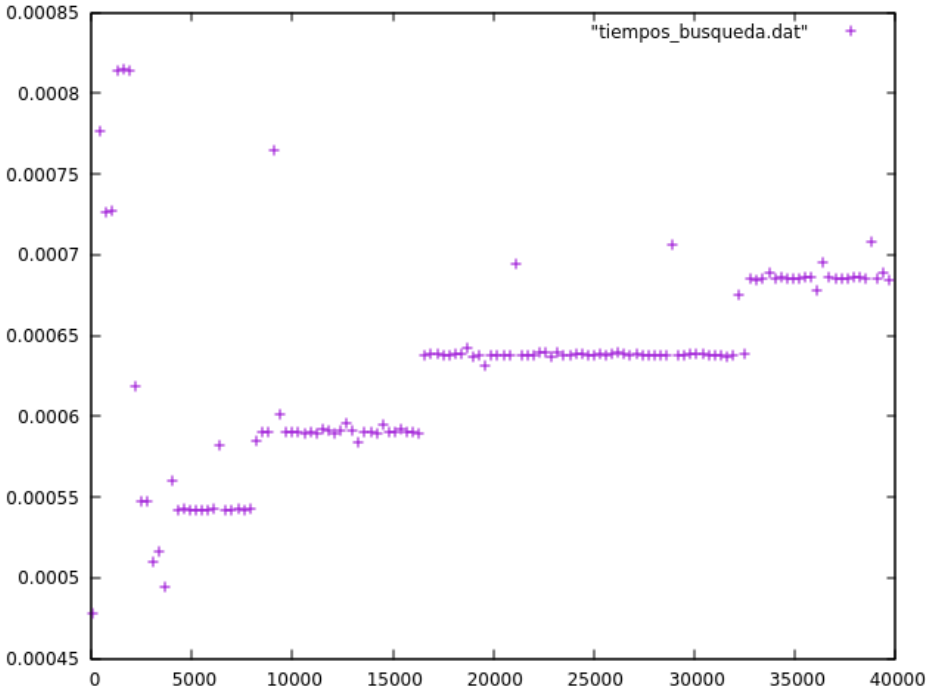
$$T(n) = \frac{6 + 10log_2(n)}{1{,}6 * 10^9} \tag{6}$$

Pero al usar un script para ejecutar el programa para tamaños del vector iguales a 100, 400, 700, ..., 40000 sale que todos los valores de tiempo de ejecución son 0 o muy cercanos a 0. Este error es debido a que para ejecutar el algoritmo de búsqueda binaria sobre un vector este debe estar ordenado, pero aún así el tiempo de ejecución es tan pequeño que es necesario ejecutar el algoritmo de búsqueda muchas veces para obtener un tiempo significativo mediante un bucle for. Es decir el programa final quedaría tal que así:

```
1   ordenar(v,tam);
2
3   clock_t tini;     // Anotamos el tiempo de inicio
4   tini=clock();
5
6   // Algoritmo a evaluar
7   for(int i=0;i<10000;i++){
8   operacion(v,tam,tam+1,0,tam-1);
9   }
10  clock_t tfin;     // Anotamos el tiempo de finalización
11  tfin=clock();
```

Ahora si ejecutamos el programa mediante un script para los tamaños del vector antes mencionados y mediante gnuplot obtenemos la siguiente gráfica de la eficiencia empírica:

Figura 3: Eficiencia empírica



Usando fit de gnuplot para una función del tipo:

$$f(x) = a * log_2 x + b$$

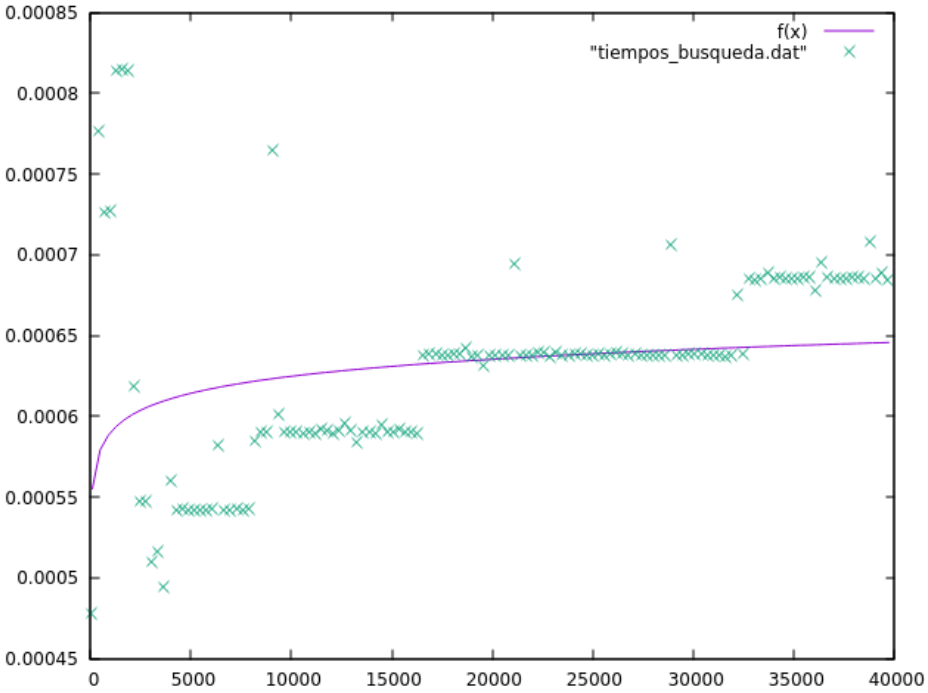
obtenemos los siguientes coeficientes:

```
Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a               = 1.05374e-05      +/- 3.484e-06    (33.06%)
b               = 0.000484692      +/- 4.845e-05    (9.996%)

correlation matrix of the fit parameters:
                a      b
a               1.000
b              -0.995  1.000
```

Y representando conjuntamente los datos y la función obtenemos la siguiente gráfica:

Figura 4: Eficiencia empírica



6. Ejercicio 4 : Mejor y peor caso

Retome el ejercicio de ordenación mediante el algoritmo de la burbuja. Debe modificar el código que genera los datos de entrada para situarnos en dos escenarios diferentes:

- El mejor caso posible. Para este algoritmo, si la entrada es un vector que ya está ordenado el tiempo de cómputo es menor ya que no tiene que intercambiar ningún elemento
- El peor caso posible. Si la entrada es un vector ordenado en orden inverso estaremos en la peor situación posible ya que en cada iteración del bucle interno hay que hacer un intercambio.

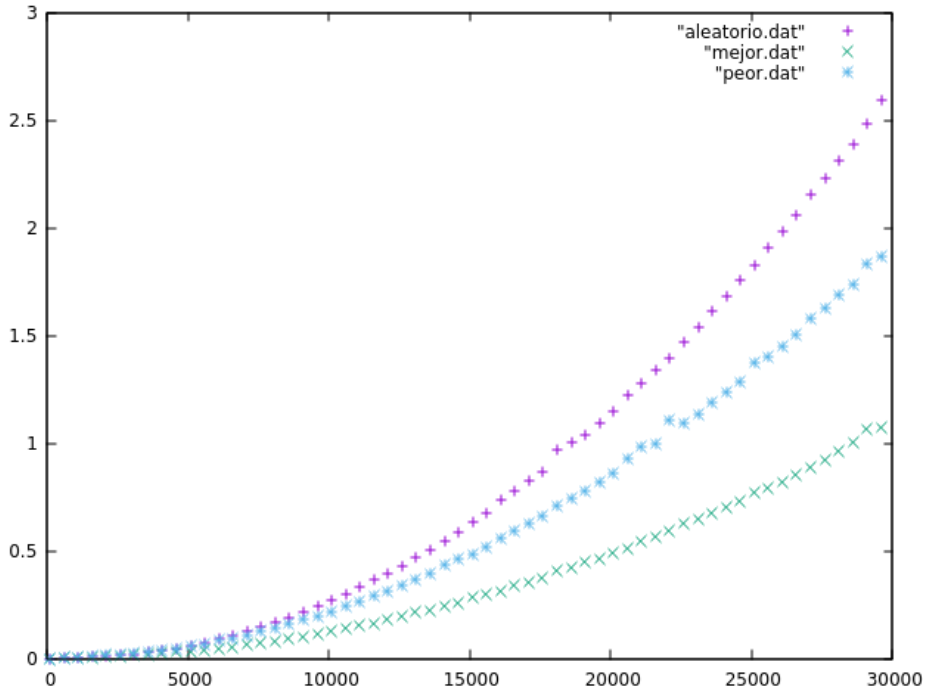
Calcule la eficiencia empírica en ambos escenarios y compárela con el resultado del ejercicio 1.

Sabemos que en general el tiempo que tarda el procesador en comparar dos enteros no depende de su valor y por tanto nos sirve con crear el vector de entrada más simple. Es decir, en el mejor caso el vector [0,1,2,3 ... n] y en el peor caso [n,n-1, ... ,3,2,1,0]. Además es trivial calcular la eficiencia teórica en el mejor caso, quitando las operaciones de dentro del bloque if obteniendo así:

$$T(n) = 3 + (3+3)(n-1) + \frac{(3+3+)(n-1)(n)}{2} = 3n^2 + 3n - 6 \tag{7}$$

Y la eficiencia teórica en el peor caso es la ya calculada en el ejercicio 1 (suponiendo que el bloque if se ejecuta siempre). Ejecutando la función en el mejor y peor caso obtenemos la siguiente gráfica de tiempos de ejecución de gnuplot:

Figura 5: Diferentes eficiencias



En el caso del peor caso (vector ordenado al revés) el tiempo de ejecución es menor que en el vector aleatorio gracias al predictor de saltos. Este mecanismo de la CPU permite al procesador tomar decisiones en el caso de un salto condicional basándose en las ejecuciones anteriores del mismo salto. Como en el nuestro caso en el bucle if siempre se incumple la condición el procesador tardará menos tiempo en ejecutar el salto al bucle, que es una operación bastante costosa, tomando siempre el caso de que no se cumpla. En el caso de que se cumpliera la condición habría una penalización y tardará más, pero en nuestro caso eso nunca pasa.

7. Ejercicio 5 : Dependencia de la implementación

Considere esta otra 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          }
```

En ella 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. Considere ahora la situación del mejor caso posible en la que el vector de entrada ya está ordenado. ¿Cuál sería la eficiencia teórica en ese mejor caso? Muestre la gráfica con la eficiencia empírica y compruebe si se ajusta a la previsión.

Empecemos calculando la eficiencia teórica:

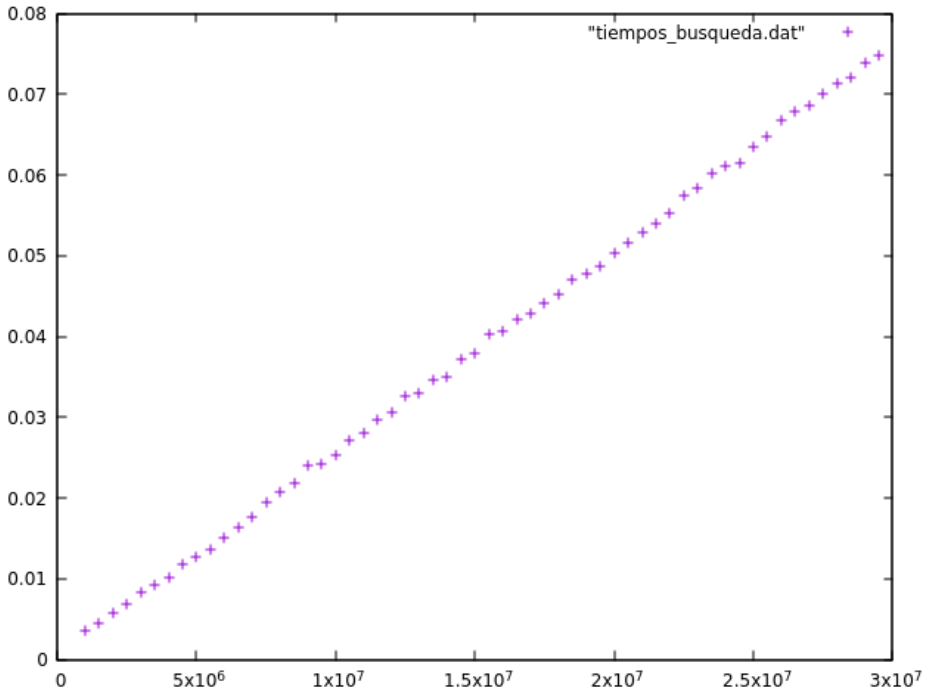
- Línea 2: 1OE (asignación).
- Línea 3: 1OE (asignación) + 3OE (resta, comparación, operación &&) + 1OE (incremento).
- Línea 4: 1OE (asignación).
- Línea 5: 1OE (asignación) + 3OE (resta*2, comparación) + 1OE (incremento).
- Línea 6: 3OE (acceso al elemento [j] y [j+1], comparación).
- Línea 7: 1OE (asignación).
- Línea 8: 2OE (acceso al elemento [j], asignación).
- Línea 9: 3OE (acceso al elemento [j] y [j+1], asignación).
- Línea 10: 2OE (acceso al elemento [j+1], asignación).

Obtenemos el siguiente tiempo de ejecución en el mejor de los casos (con el número de línea entre paréntesis):

$$T(n) = 1OE(2) + 4OE(3) + 1OE(4) + 4OE(5) + ((3OE(6) + 1OE(5) + 3OE(5)) * (n-1)) \\ + 1OE(3) + 3OE(3) = 14 + 7n - 7 = 7n + 7$$

La gráfica obtenida con gnuplot es la siguiente, con tiempos de ejecución para tamaños del vector tomados de 500000 en 500000 empezando en n=1000000 y acabando en n=30000000:

Figura 6: Eficiencia en el mejor caso



Como podemos ver si se ajuta a la teórica pues la mayoría de valores de la gráfica se agrupan formando una recta.

8. Ejercicio 6 : Influencia del proceso de compilación

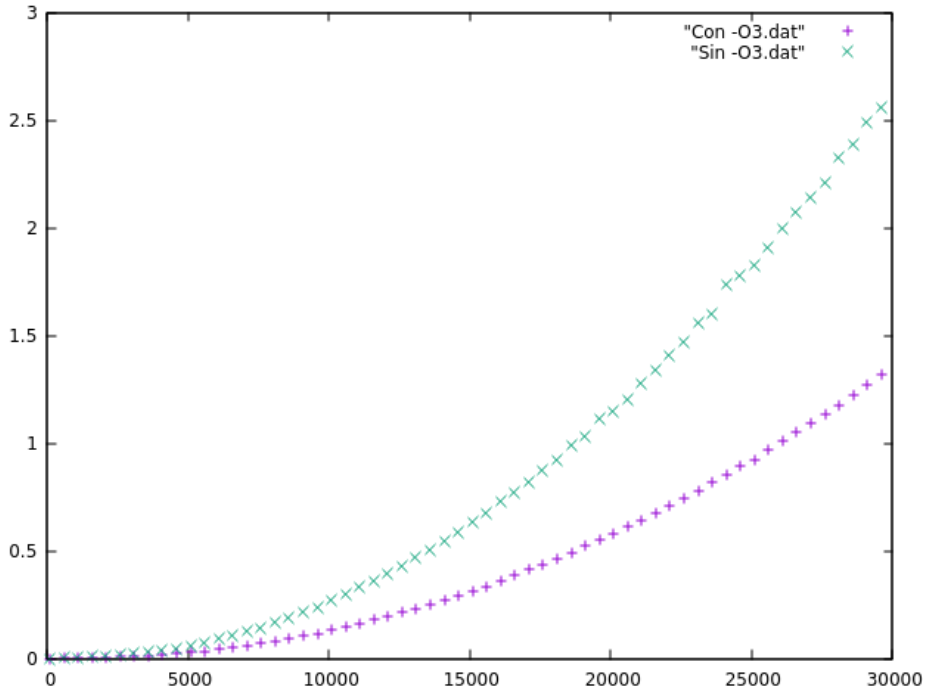
Retome el ejercicio de ordenación mediante el algoritmo de la burbuja. Ahora replique dicho ejercicio pero previamente deberá compilar el programa indicándole al compilador que optimice el código. Esto se consigue así:

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

Compare las curvas de eficiencia empírica para ver cómo mejora esto la eficiencia del programa.

La diferencia en eficiencia cuando se utiliza la opcion -O3 se puede ver claramente en la siguiente gráfica donde se han representado los datos obtenidos en ambos casos:

Figura 7:



Se han tomado tiempos de ejecución para tamaños del vector 100, 600, 1100, ..., 30000. Podemos ver claramente como el tiempo de ejecución se reduce considerablemente.

9. Ejercicio 7 : Multiplicación matricial

Implemente un programa que realice la multiplicación de dos matrices bidimensionales. Realice un análisis completo de la eficiencia tal y como ha hecho en ejercicios anteriores de este guión.

El programa a implementar quedaría tal que así:

```
1  #include <iostream>
2  #include <ctime>
3  #include <cstdlib>
4
5  using namespace std;
6
7  void sintaxis()
8  {
9    cerr << ``Sintaxis:'' << endl;
10   cerr << ``TAM:Tamaño de la matriz (>0)'' << endl;
11   cerr << ``VMAX: Valor maximo (>0)'' << endl;
12   cerr << ``Se genera una matriz de tamaño TAM x TAM con elementos
       aleatorios en [0,VMAX[`` << endl;
13   exit(EXIT_FAILURE);
14 }
15
16 int main(int argc, char * argv[])
17 {
18   if (argc!=3)
19     sintaxis();
20   int tam=atoi(argv[1]);
21   int vmax=atoi(argv[2]);
22   if (tam<=0 || vmax<=0)
23     sintaxis();
24
25   // Generación de la matriz aleatoria
26   const int MAXIMO=800;
27   int v[MAXIMO][MAXIMO],u[MAXIMO][MAXIMO], w[MAXIMO][MAXIMO];
28   srand(time(0));
29   for (int i=0; i<tam; i++){
30     for(int j=0;j<tam;j++){
31       v[i][j]=rand() %vmax;
32       u[i][j]=rand() %vmax;
33     }
34   }
35   int suma=0;
36   clock_t tini;
37   tini=clock();
38
39   for(int i=0; i<tam; i++){
40     for(int j=0; j<tam; j++){
41       for(int k=0; k<tam; k++){
42         suma=suma+(v[i][k]*u[k][j]);
43       }
44       w[i][j]=suma;
45       suma=0;
46     }
47   }
48
49   clock_t tfin;
50   tfin=clock();
51
52   cout << tam << "\t" << (tfin-tini)/(double)CLOCKS_PER_SEC << endl;
53 }
```

Vamos a centrarnos en el algoritmo de multiplicación de matrices:

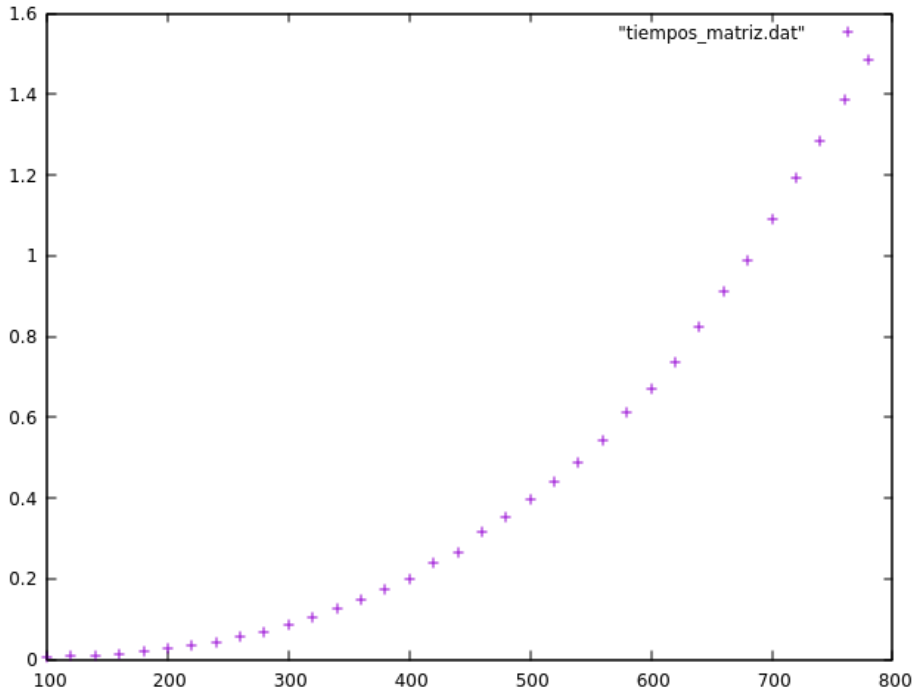
```
1   for(int i=0; i<tam; i++){
2     for(int j=0; j<tam; j++){
3       for(int k=0; k<tam; k++){
4           suma=suma+(v[i][k]*u[k][j]);
5       }
6       w[i][j]=suma;
7       suma=0;
8     }
9   }
```

En cuanto a la eficiencia teórica, al tiempo de ejecución le corresponde la siguiente ecuación (con el número de línea al que corresponde cada conjunto de OEs entre paréntesis), donde n (=tam) es el número de filas y columnas de la matriz:

$$T(n) = 2OE(1)+2OE(2) + 2OE(3) + [[[7OE(4) + 2OE(3)] * n + 3OE(6) + 1OE(7)+ 2OE(2)] * n + 2OE(1)] * n = 9n^3 + 6n^2 + 2n + 6$$

Ahora pasamos a calcular la eficiencia empírica. Compilamos el programa y lo ejecutamos utilizando un script para n= 100, 120, 140, ..., 760, 780, 800, guardando los tiempos de ejecución obtenidos en un archivo tiempos_matriz.dat. Obtenemos con gnuplot la siguiente gráfica:

Figura 8: Eficiencia en el producto de matrices nxn



Hemos de realizar un ajuste con una función de la forma:

$$f(x) = ax^3 + bx^2 + cx + d$$

obtenemos los siguientes coeficientes usando la función fit de gnuplot:

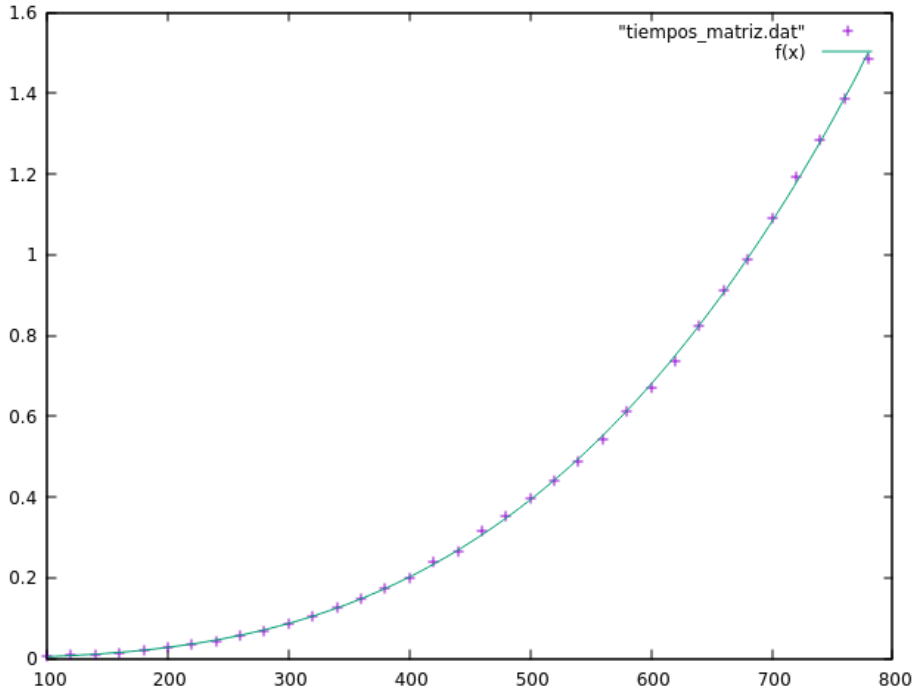
```
Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a               = 3.30848e-09      +/- 1.662e-10    (5.023%)
b               = -1.78549e-07     +/- 2.213e-07    (123.9%)
c               = 5.22417e-05      +/- 8.836e-05    (169.1%)
```

```
d               = -0.00238964      +/- 0.0102       (426.8%)

correlation matrix of the fit parameters:
                a      b      c      d
a               1.000
b              -0.991  1.000
c               0.954 -0.985  1.000
d              -0.862  0.913 -0.965  1.000
```

Ahora representamos conjuntamente f(x) y los datos obtenidos empíricamente y vemos que el ajuste es bastante bueno, es decir, la eficiencia empírica y la teórica se corresponden:

Figura 9: Comparación entre eficiencia empírica y teórica

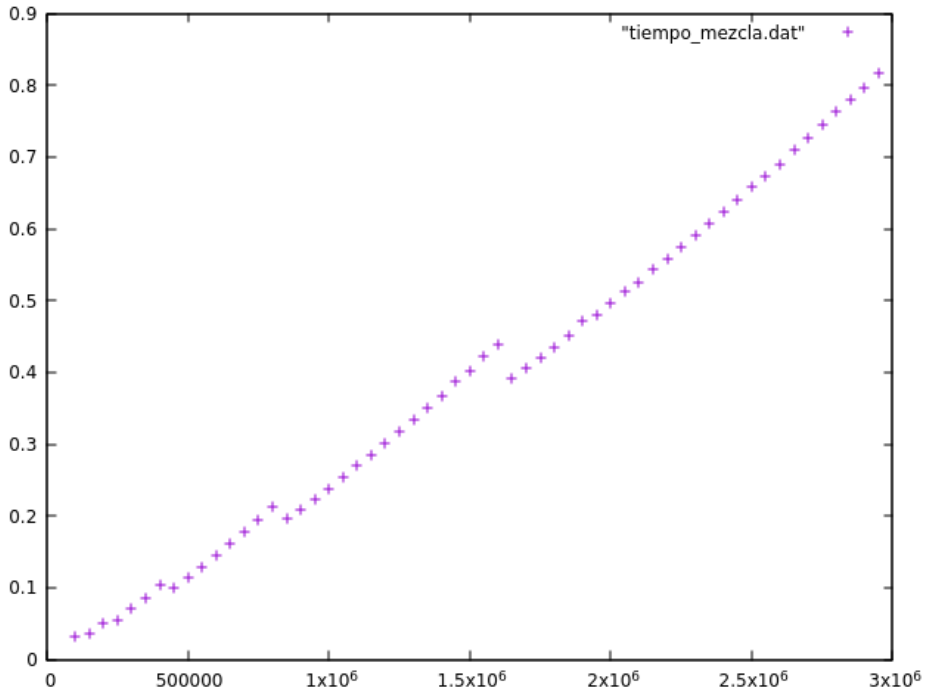


10. Ejercicio 8 : Ordenación por Mezcla

Estudie el código del algoritmo recursivo disponible en el fichero mergesort.cpp. En él, se integran dos algoritmos de ordenación: inserción y mezcla (o mergesort). 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. Como ya habrá estudiado, la eficiencia teórica del mergesort es n log(n). Realice un análisis de la eficiencia empírica y haga el ajuste de ambas curvas. Incluya también, para este caso, un pequeño estudio de cómo afecta el parámetro UMBRAL_MS a la eficiencia del algoritmo. Para ello, pruebe distintos valores del mismo y analice los resultados obtenidos.

Para medir la eficiencia empírica compilamos el programa mergesort.cpp y lo ejecutamos mediante un script para n = 100000, 150000, 200000, ..., 3000000 guardando en un archivo tiempo_mezcla.dat los tiempos de ejecución obtenidos. Mediante gnuplot obtenemos la siguiente gráfica:

Figura 10: Cálculo eficiencia empírica algoritmo de mezcla-inserción



Ajustamos con la función fit de gnuplot los datos con una función de la forma:

$$f(x) = axlog(x) + b$$

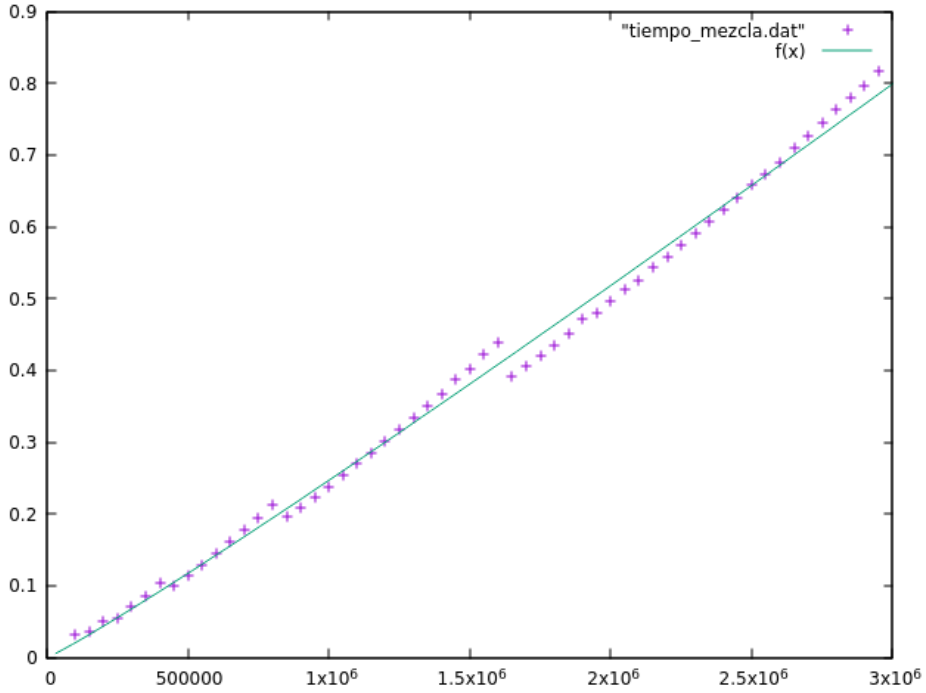
obteniendo los siguientes valores para a y b:

```
Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a               = 1.78633e-08      +/- 1.703e-10    (0.9534%)
b               = -0.000175894     +/- 0.004319     (2455%)

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

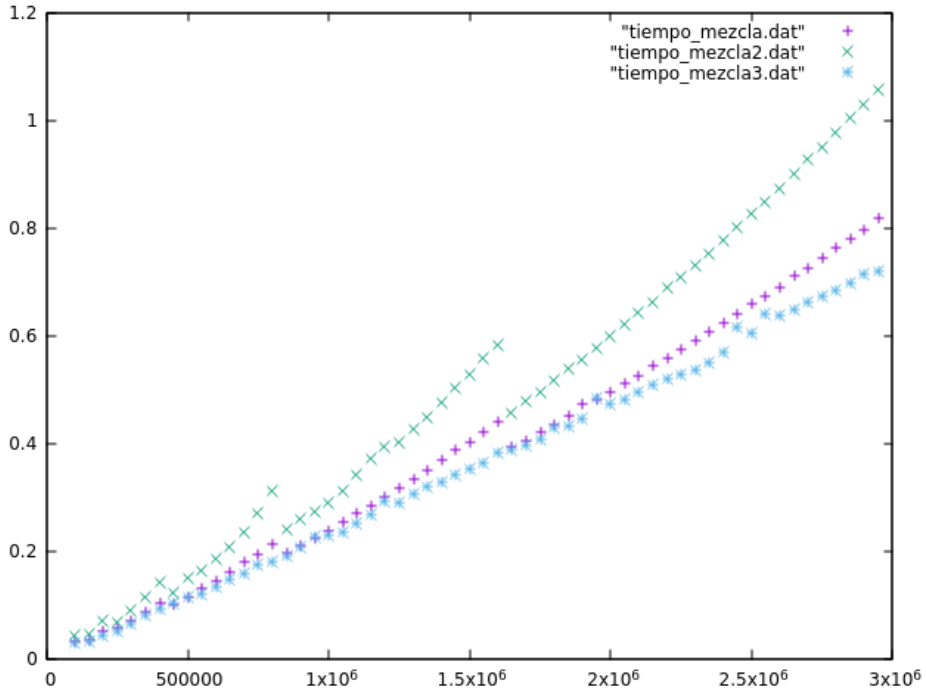
Y obteniendose la siguiente gráfica al representar los datos empíricos y la función ajustada conjuntamente:

Figura 11:



Finalmente, repetimos el cálculo de la eficiencia empírica cambiando el valor del parámetro UMBRAL_MS. Primero lo establecemos en 200, y almacenamos los datos en un archivo tiempo_mezcla2.dat. Luego lo establecemos en 10 y almacenamos los datos en tiempo_mezcla3.dat. Al representar los tres conjuntos de datos obtenidos se obtiene la siguiente gráfica:

Figura 12: Comparación al cambiar UMBRAL_MS



Los datos de color azul corresponden al valor 10 del parámetro, y los de color verde al valor 200. El valor inicial del parámetro era 100, y por tanto podemos concluir que al aumentar el valor del parámetro UMBRAL_MS el tiempo de ejecución aumenta y por tanto el programa es menos eficiente, y que al disminuir el valor del parámetro disminuye el tiempo de ejecución y el programa es más eficiente.