Práctica 1. Estructuras de Datos
Noelia Escalera Mejías. Grupo A3

Ejercicio 1:
Procedemos a calcular la eficiencia teórica del siguiente 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;
              }
}
```

Eficiencia Teórica:

-Bucle for: Se ejecuta n-1 veces

- **Línea 2:** 4 operaciones, Asignación (i = 0), Resta (n − 1), Comparación (i < n − 1) e Incremento (i++). 3 operaciones se ejecutan una vez y otras tres se ejecutan n veces.

-Bucle for (dentro del for anterior): Se ejecuta n-i-1 veces, es una progresión aritmética

- **Línea 3:** 5 operaciones, Asignación (j = 0), Resta (n − i, (n − i) − 1), Comparación (j < n − i − 1) e Incremento (j + +). 4 operaciones se ejecutan una vez y otras 4 se ejecutan n veces.

-If (dentro del for anterior): Se ejecuta siempre, ya que estamos en el peor de los casos

- **Línea 4:** 4 operaciones, Acceso a vectores (v[j], v[j +1]), Suma (j +1), Comparación (v[j] > v[j + 1]).

- **Línea 5:** 2 operaciones, Acceso a vector (v[j]), Asignación (aux = v[j]).

- **Línea 6:** 4 operaciones, Acceso a vectores (v[j], v[j + 1]), Asignación (v[j] = v[j + 1]), Suma (j + 1).

- **Línea 7:** 3 operaciones, Acceso a vector (v[j + 1]), Suma (j + 1), Asignación (v[j + 1] = aux).

Por tanto el tiempo de ejecución en el peor de los casos será:

$$3+\sum_{i=0}^{n-1}\left(3+4+\sum_{j=0}^{n-i-1}(4+4+2+4+3)\right)=3+\sum_{i=0}^{n-1}(7+17(n-i-1))=3+(n-1)\frac{7+17(n-n+3-1)+7+17(n+1)}{2}=¿$$

$$3+(n-1)\frac{7+17*2+7+17n+17}{2}=3+(n-1)\frac{17n+65}{2}=3+\frac{17n^2+65n-17n-65}{2}=\frac{17n^2+48n-59}{2}$$

Luego podemos decir que tenemos una eficiencia de O(n^2).

Eficiencia empírica

Para calcular la eficiencia empírica hemos usado los siguientes ficheros fuente:

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

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: Tamaño del vector (>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[])
{
  // Lectura de parámetros
  if (argc!=3)
    sintaxis();
  int tam=atoi(argv[1]);     // Tamaño del vector
  int vmax=atoi(argv[2]);    // Valor máximo
  if (tam<=0 || vmax<=0)
    sintaxis();

  // Generación del vector aleatorio
  int *v=new int[tam];       // Reserva de memoria
  srand(time(0));            // Inicialización del generador de números 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();

  int x = vmax+1;
  ordenar(v,tam);

 clock_t tfin;    // Anotamos el tiempo de finalización
  tfin=clock();
```

```
 // Mostramos resultados
  cout << tam << "\t" << (tfin-tini)/(double)CLOCKS_PER_SEC << endl;

  delete [] v;      // Liberamos memoria dinámica
}
```

fichero: ordenacion.cpp

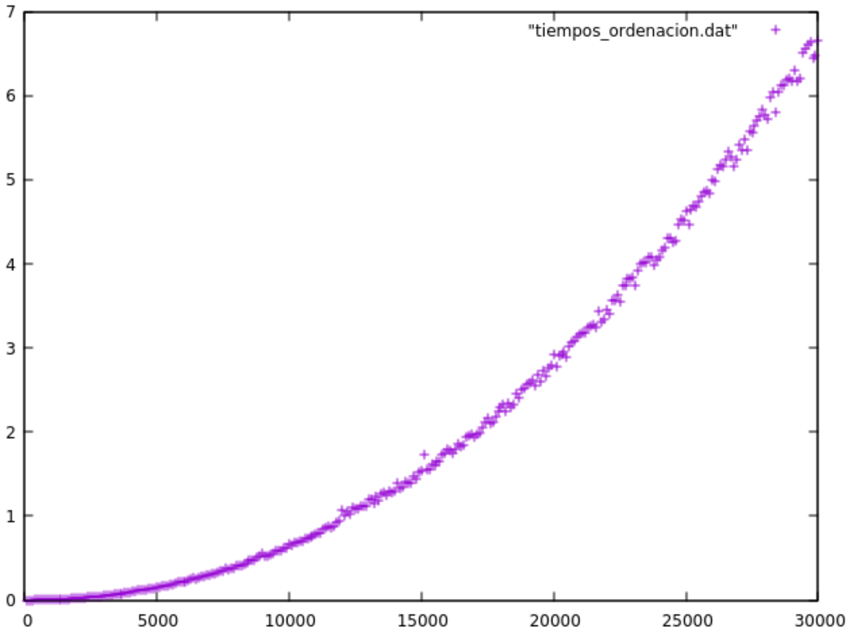
```
#!/bin/csh
@ inicio = 100
@ fin = 30000
@ incremento = 100
set ejecutable = ordenacion
set salida = tiempos_ordenacion.dat

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

fichero: ejecuciones_ordenacion.csh

Hemos compilado el el programa ordenación.cpp de la siguiente manera:

```
g++ busqueda_lineal.cpp -o busqueda_lineal
```

Tras ejecutar el script ejecuciones_ordenacion.csh, procedemos a dibujar la gráfica de los tiempos con gnuplot:

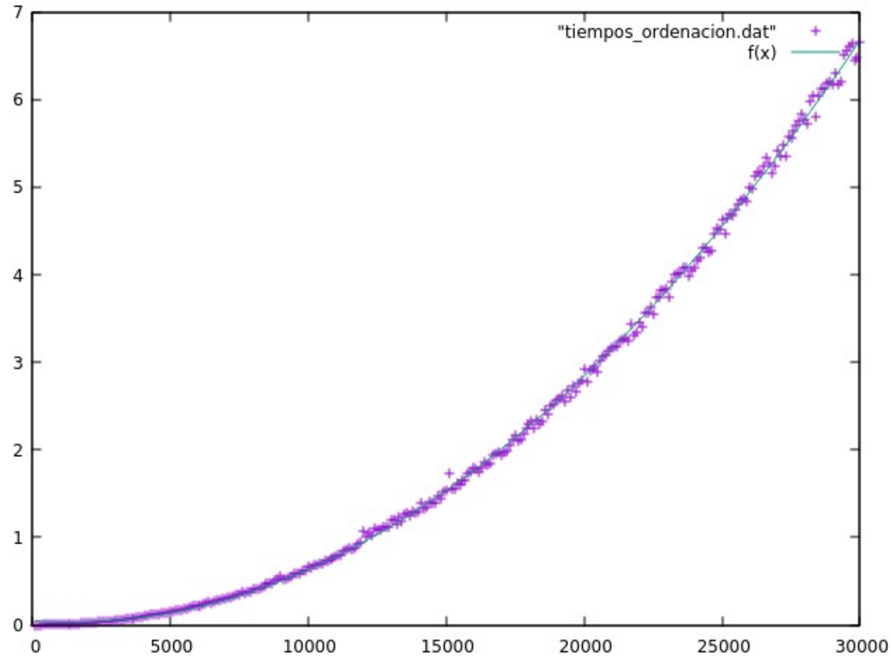


Ejercicio 2:

A continuación, vamos a obtener el ajuste de regresión para el algoritmo anterior. Para realizar este ajuste, supondremos que $f(x)=ax^2+bx+c$. f(x) es la función a la que queremos ajustar nuestros tiempos. Según gnuplot, los valores más adecuados para a, b y c son:

$$a=8.11499e-09$$
$$b=-2.26893e-05$$
$$c=0.0486703$$

Si dibujamos superpuestas f(x) y la función de tiempos_ordenación.dat, nos queda lo siguiente:



Ejercicio 3:

El algoritmo del que se nos pide ahora calcular la eficiencia se trata de una búsqueda binaria. Consiste en, dado un vector y un valor, desplazarse a la mitad del vector. Dependiendo de si el valor de la mitad del vector es mayor o menor al buscado, se buscará en la mitad posterior o anterior del vector (obviamente, si es igual el valor se ha encontrado). La mitad correspondiente del vector se dividirá de nuevo por la mitad y se repetirá el proceso hasta que se encuentre el valor o no haya más posiciones en las que buscar. El código del algoritmo es el siguiente:

```
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;
}
```

Eficiencia teórica

Cada iteración, el vector tendrá un tamaño la mitad de pequeño que la iteración anterior. Es decir, que para un número k de iteraciones tendremos:

$$\frac{n_{inicial}}{2^k}<1\,; n_{inicial}<2^k; \ln(n_{inicial})<\ln(2^k)\,; \ln(n_{inicial})<k*\ln(2)\,; \log_b=\frac{\ln(x)}{\ln(b)}$$

Eficiencia empírica

Hemos compilado el programa ejercicio_desc.cpp de la siguiente forma:

```
g++ ejercicio_desc.cpp -o ejercicio_desc
```

El código fuente que usaremos será el siguiente:

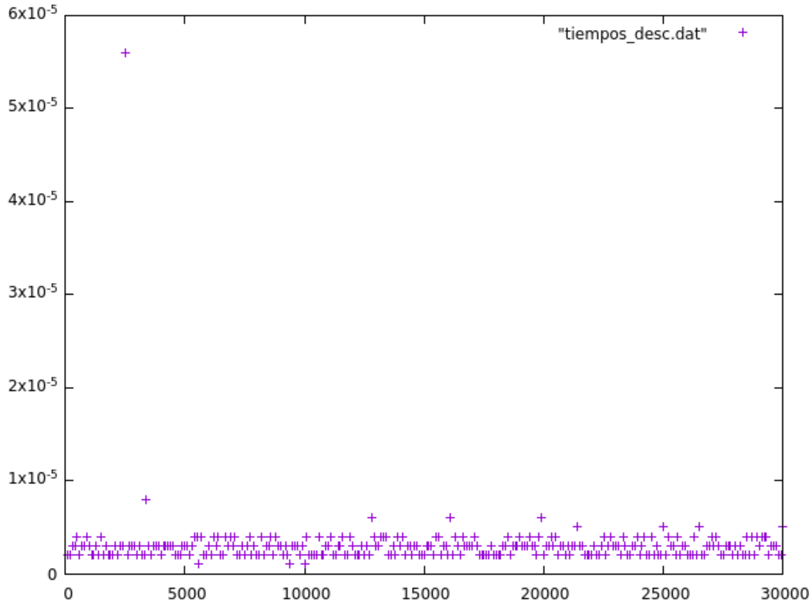
```
#!/bin/csh
@ inicio = 100
@ fin = 30000
@ incremento = 100
set ejecutable = ejercicio_desc
set salida = tiempos_desc.dat

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

fichero: ejecuciones_desc.csh

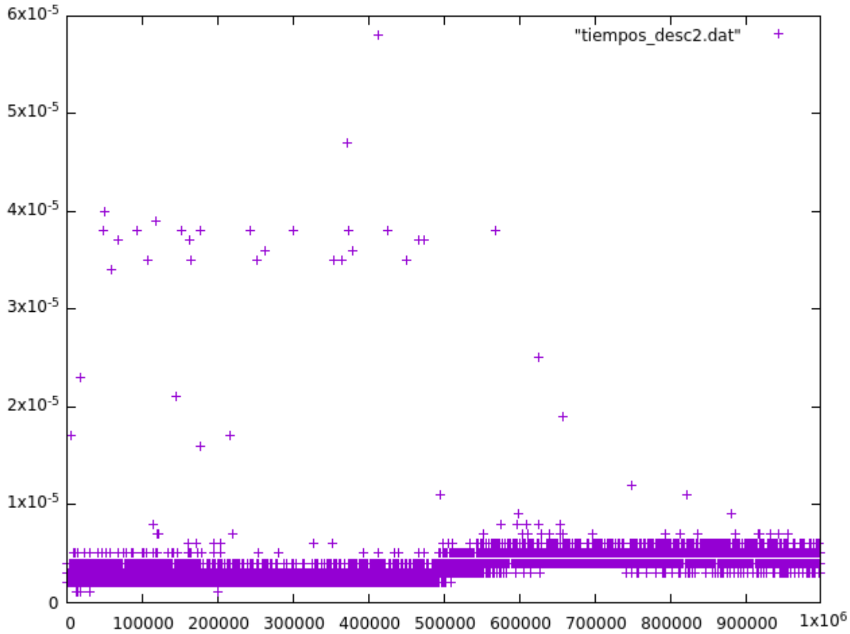
```cpp
#include <iostream>
#include <ctime>    // Recursos para medir tiempos
#include <cstdlib>  // Para generación de números pseudoaleatorios

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();

  // Generación del vector aleatorio
  int *v=new int[tam];       // Reserva de memoria
  srand(time(0));            // Inicialización del generador de números pseudoaleatorios
  for (int i=0; i<tam; i++)  // Recorrer vector
    v[i] = rand() % tam;

  clock_t tini;    // Anotamos el tiempo de inicio
  tini=clock();

  // Algoritmo a evaluar
  operacion(v,tam,tam+1,0,tam-1);

  clock_t tfin;    // Anotamos el tiempo de finalización
  tfin=clock();

  // Mostramos resultados
  cout << tam << "\t" << (tfin-tini)/(double)CLOCKS_PER_SEC << endl;

  delete [] v;     // Liberamos memoria dinámica
}
```

fichero: ejercicio_desc.cpp

Tras ejecutar el script ejecuciones_desc.csh, hemos obtenido la siguiente gráfica con gnuplot:



En esta gráfica no podemos apreciar la tendencia logarítmica, se asemeja más a una gráfica lineal. Para poder apreciar mejor la tendencia logarítmica, hemos subido el número de ejecuciones del algoritmo (hemos cambiado el 30000 por un 1000000 en ejecuciones_desc.csh).

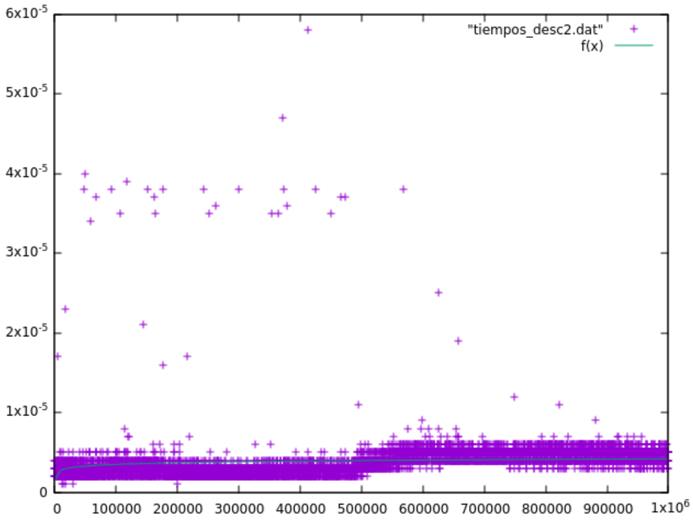


Ahora ya se puede apreciar (al menos ligeramente) la tendencia logarítmica. Realicemos el ajuste, vamos a ajustar la gráfica a f(x)=alog(bx). Nos da los siguientes valores de a y b:

$$a = 3.02384e\text{-}07$$

$$b = 0.955181$$

Y si dibujamos las gráficas superpuestas nos queda lo siguiente:



Ejercicio 4:

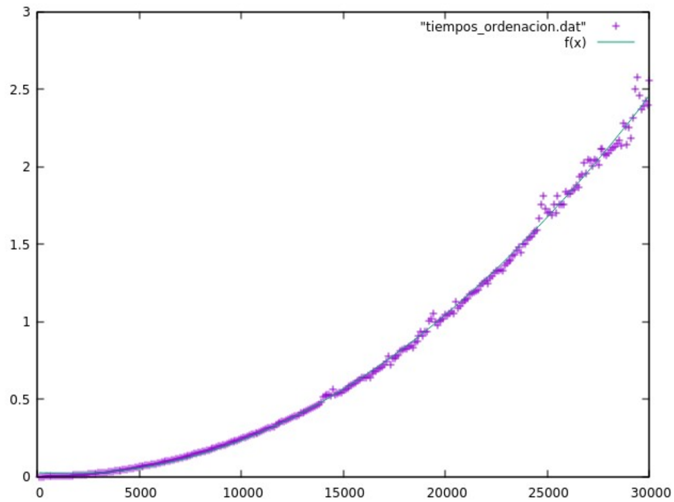
Hemos modificado el código de ordenacion.cpp de manera que el vector esté ya ordenado (mejor caso posible). Para ello, hemos llenado el vector de la siguiente manera:

```
for (int i=0; i<tam; i++)
    v[i] = i;
```

Si realizamos el ajuste con $f(x)=ax^2+bx+c$ nos quedan los siguientes valores:

$$a = 6.05795e\text{-}09$$
$$b = -7.61309e\text{-}05$$
$$c = 0.258188$$

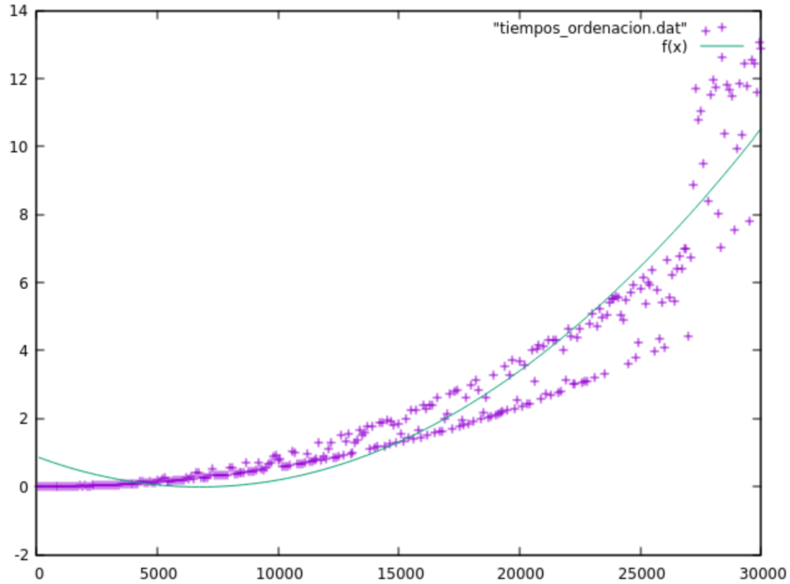
Obtenemos la siguiente gráfica:



Ahora tenemos que modificar el código para que el vector esté ordenado al revés (peor caso posible). Para ello, hemos llenado el vector de esta forma:

```
for (int i=0; i<tam; i++)
   v[tam-i-1] = i;
```

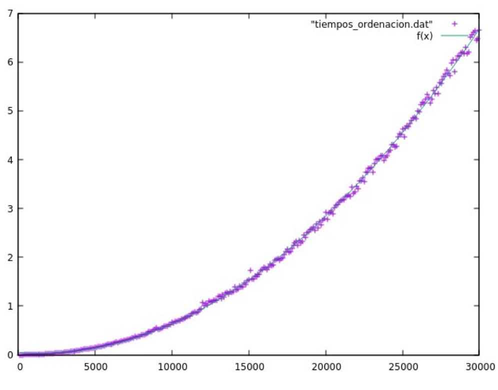
Obtenemos la siguiente gráfica:



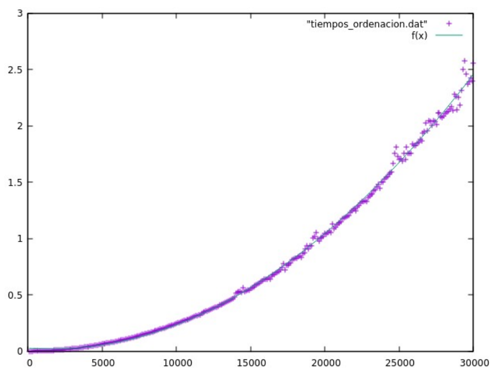
Si hacemos el ajuste con $f(x)=ax^2+bx+c$, nos queda:

$$a = 1.95253e\text{-}08$$
$$b = -0.000264635$$
$$c = 0.886071$$

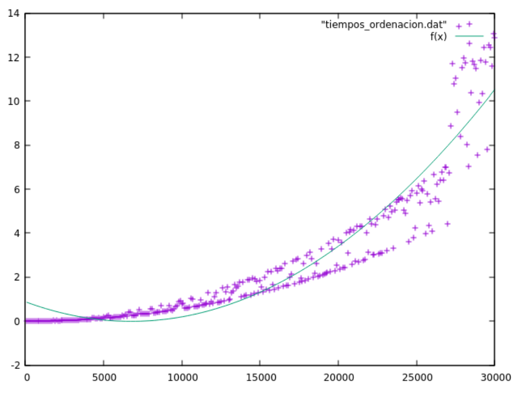
Comparamos las tres gráficas de la ordenación por burbuja:



vector aleatorio



mejor caso posible



peor caso posible

Como vemos, en el caso de vectores aleatorios se llega, como máximo, a un tiempo de 7 por ejecución, en el del mejor caso posible a unos 2’5 y en el del peor caso hasta los 14. Además, en este último caso, la gráfica está mucho más dispersa.

Ejercicio 5

Ahora procedemos a calcular la eficiencia teórica de esta variante del algoritmo de la burbuja:

```
void ordenar(int *v, int n) {
   bool cambio=true;
   for (int i=0; i<n-1 && cambio; i++) {
        cambio=false;
        for (int j=0; j<n-i-1; j++)
                if (v[j]>v[j+1]) {
                        cambio=true;
                        int aux = v[j];
                        v[j] = v[j+1];
                        v[j+1] = aux;
                }
   }
}
```

La vamos a calcular en el mejor caso posible, es decir, que el vector esté ya ordenado.

-Línea 2: 1 operación (asignación).
-Línea 3: 4 operaciones (asignación, comparación, resta y operación lógica). Las realiza solo una vez, pues luego cambio se pone a false.
-Línea 4: 1 operación (asignación).
-Línea 5: 4 operaciones que se ejecutan una vez (asignación, comparación y dos restas) y 4 operaciones que se ejecutan n-1 veces (comparación, dos restas e incremento).
-Línea 6: 4 operaciones que se ejecutan n-1 veces (2 accesos a vectores, comparación y suma).

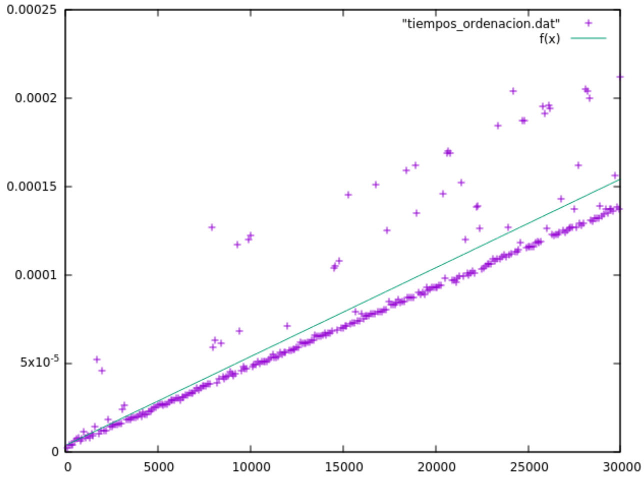
$$1+4+1+4+\sum_{j=0}^{n-1}(4+4)=10+8(n-1)=10+8n-8=8n-2$$

Luego tenemos una eficiencia lineal, *O(n)*.

Valores de ajuste:

a = 5.02089e-09
b =3.40555e-06

Y nos queda la siguiente gráfica:



Ejercicio 6:

Para este ejercicio vamos a usar la primera versión del programa de ordenación por burbuja. Compilamos el código de la siguiente manera:

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

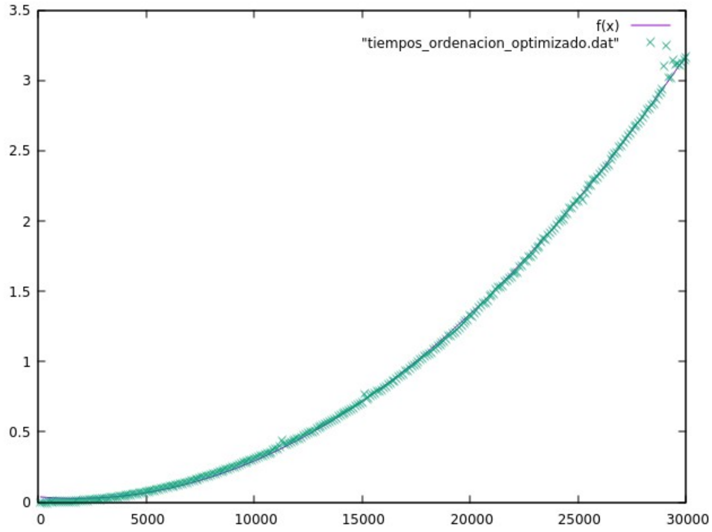
Vamos a usar el siguiente script para las ejecuciones:

```
#!/bin/csh
@ inicio = 100
@ fin = 30000
@ incremento = 100
set ejecutable = ordenacion_optimizado
set salida = tiempos_ordenacion_optimizado.dat

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

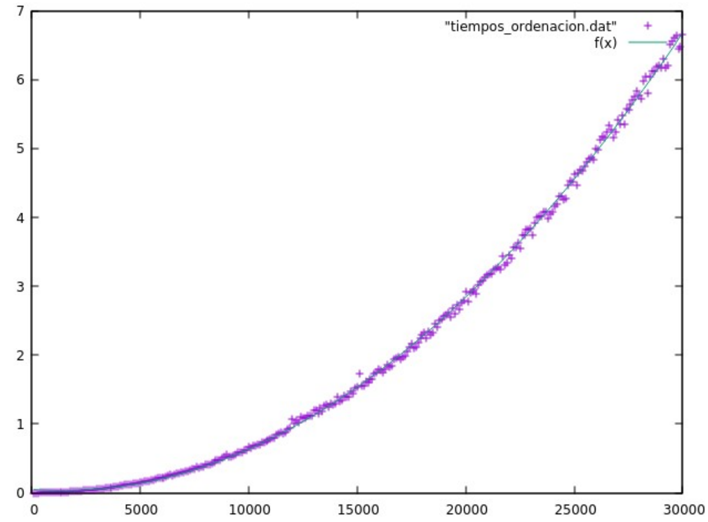
Si ajustamos la gráfica de tiempos a $f(x)=ax^2+bx+c$ nos quedan los siguientes valores de a, b y c:

$$a = 3.96101e\text{-}09$$
$$b = -1.41711e\text{-}05$$
$$c = 0.0383802$$

Vamos a comparar la gráfica de tiempos optimizada con la no optimizada:



gráfica con código optimizado



gráfica normal

Como vemos, en la gráfica optimizada el tiempo máximo de ejecución es de alrededor de 3, mientras que en la no optimizada se eleva a unos 7. Por otro lado, los datos de la gráfica optimizada se adaptan mejor a la gráfica de f(x).

Ejercicio 7:

El algoritmo que hemos usado para multiplicar matrices es el siguiente:

```
for (int i=0; i<n; ++i){
        for (int j=0; j<n; ++j){
                for (int k=0; k<n; ++k){
                        resultado[i][j] += m1[i][k] * m2[k][j];
                }
        }
}
```

Eficiencia teórica:

-Bucle for. Se ejecuta n veces:

· **Línea 1:** 2 operaciones que se ejecutan una vez (asignación y comparación) y 2 que se ejecutan n veces (comparación e incremento).

-Bucle for (dentro del for anterior). Se ejecuta n veces por cada iteración del for anterior:

· **Línea 2:** 2 operaciones que se ejecutan una vez (asignación y comparación) y 2 que se ejecutan n veces (comparación e incremento).

-Bucle for (dentro del for anterior). Se ejecuta n veces por cada iteración del for anterior:

· **Línea 3:** 2 operaciones que se ejecutan una vez (asignación y comparación) y 2 que se ejecutan n veces (comparación e incremento).

· **Línea 4:** 5 operaciones: 3 accesos a matrices, un += y una multiplicación.

$$2+\sum_{i=0}^{n}\left(2+2+\sum_{j=0}^{n}\left(2+2+\sum_{k=0}^{n}(2+5)\right)\right)=2+\sum_{i=0}^{n}\left(4+\sum_{j=0}^{n}(4+7n)\right)=2+\sum_{i=0}^{n}(4+(4+7n)n)=¿$$
$$2+(4+(4+7n)n)n=2+(4+4n+7n^2)n=7n^3+4n^2+4n+2$$

Luego tenemos una eficiencia de *O(n³)*.

Eficiencia empírica:

Hemos usado los siguientes ficheros para calcular la eficiencia empírica:

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

using namespace std;

int main(int argc, char * argv[])
{
  // Lectura de parámetros
  int tam=atoi(argv[1]);     // Tamaño del vector
  int vmax=atoi(argv[2]);    // Valor máximo

  // Generación del vector aleatorio
  int ** m1=new int * [tam];
   int ** m2=new int * [tam];
   int ** resultado=new int * [tam];

   for (int i=0; i<tam; i++){
        m1[i] = new int [tam];
        m2[i] = new int [tam];
        resultado[i] = new int [tam];
   }

  srand(time(0));

  for (int i=0; i<tam; i++){
        for (int j=0; j<tam; j++){
                m1[i][j] = rand()%vmax;
                m2[i][j] = rand()%vmax;
        }
   }

  clock_t tini;    // Anotamos el tiempo de inicio
  tini=clock();

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

  clock_t tfin;    // Anotamos el tiempo de finalización
  tfin=clock();
```

```
  // Mostramos resultados
  cout << tam << "\t" << (tfin-tini)/(double)CLOCKS_PER_SEC << endl;

}
```

fichero: matrices.cpp

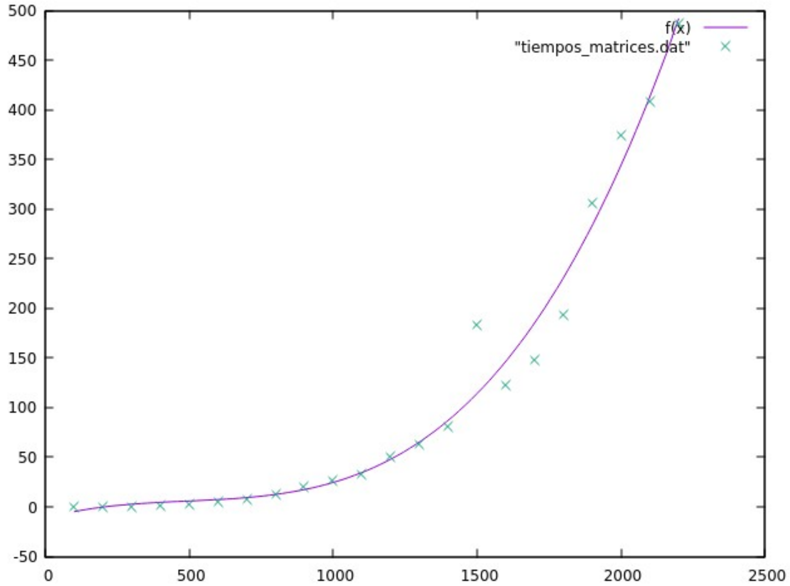
```
#!/bin/csh
@ inicio = 100
@ fin = 2300
@ incremento = 100
set ejecutable = matrices
set salida = tiempos_matrices.dat

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

fichero: ejecuciones_matrices.csh

Si ajustamos la gráfica de tiempos a $f(x)=ax^3+bx^2+cx+d$, nos quedan los siguientes valores:

$$a = 9.29005e\text{-}08$$
$$b = -0.000136846$$
$$c = 0.0800989$$
$$d = -11.5687$$

Y la gráfica nos quedaría así:



Ejercicio 8:

Se nos pide calcular la eficiencia empírica de mergesort. Para ello, usaremos los siguientes ficheros:

```
/**
   @file Ordenaci�n por mezcla
*/


#include <iostream>
#include <ctime>
#include <cstdlib>
#include <climits>
#include <cassert>

using namespace std;



/* ************************************************************ */
/*  Método de ordenaci�n por mezcla  */

/**
   @brief Ordena un vector por el m�todo de mezcla.

   @param T: vector de elementos. Debe tener num_elem elementos.
             Es MODIFICADO.
   @param num_elem: n�mero de elementos. num_elem > 0.

   Cambia el orden de los elementos de T de forma que los dispone
   en sentido creciente de menor a mayor.
   Aplica el algoritmo de mezcla.
*/
inline static
void mergesort(int T[], int num_elem);



/**
   @brief Ordena parte de un vector por el m�todo de mezcla.

   @param T: vector de elementos. Tiene un n�mero de elementos
                   mayor o igual a final. Es MODIFICADO.
   @param inicial: Posici�n que marca el incio de la parte del
                   vector a ordenar.
   @param final: Posici�n detr�s de la �ltima de la parte del
                   vector a ordenar.
           inicial < final.

   Cambia el orden de los elementos de T entre las posiciones
   inicial y final - 1 de forma que los dispone en sentido creciente
   de menor a mayor.
   Aplica el algoritmo de la mezcla.
*/
static void mergesort_lims(int T[], int inicial, int final);
```

```
/**
   @brief Ordena un vector por el m�todo de inserci�n.

   @param T: vector de elementos. Debe tener num_elem elementos.
             Es MODIFICADO.
   @param num_elem: n�mero de elementos. num_elem > 0.

   Cambia el orden de los elementos de T de forma que los dispone
   en sentido creciente de menor a mayor.
   Aplica el algoritmo de inserci�n.
*/
inline static
void insercion(int T[], int num_elem);


/**
   @brief Ordena parte de un vector por el m�todo de inserci�n.

   @param T: vector de elementos. Tiene un n�mero de elementos
                   mayor o igual a final. Es MODIFICADO.
   @param inicial: Posici�n que marca el incio de la parte del
                   vector a ordenar.
   @param final: Posici�n detr�s de la �ltima de la parte del
                   vector a ordenar.
          inicial < final.

   Cambia el orden de los elementos de T entre las posiciones
   inicial y final - 1 de forma que los dispone en sentido creciente
   de menor a mayor.
   Aplica el algoritmo de la inserci�n.
*/
static void insercion_lims(int T[], int inicial, int final);


/**
   @brief Mezcla dos vectores ordenados sobre otro.

   @param T: vector de elementos. Tiene un n�mero de elementos
                   mayor o igual a final. Es MODIFICADO.
   @param inicial: Posici�n que marca el incio de la parte del
                   vector a escribir.
   @param final: Posici�n detr�s de la �ltima de la parte del
                   vector a escribir
          inicial < final.
   @param U: Vector con los elementos ordenados.
   @param V: Vector con los elementos ordenados.
             El n�mero de elementos de U y V sumados debe coincidir
             con final - inicial.

   En los elementos de T entre las posiciones inicial y final - 1
   pone ordenados en sentido creciente, de menor a mayor, los
   elementos de los vectores U y V.
*/
static void fusion(int T[], int inicial, int final, int U[], int V[]);



/**
   Implementaci�n de las funciones
**/
```

```
inline static void insercion(int T[], int num_elem)
{
  insercion_lims(T, 0, num_elem);
}


static void insercion_lims(int T[], int inicial, int final)
{
  int i, j;
  int aux;
  for (i = inicial + 1; i < final; i++) {
    j = i;
    while ((T[j] < T[j-1]) && (j > 0)) {
      aux = T[j];
      T[j] = T[j-1];
      T[j-1] = aux;
      j--;
    };
  };
}


const int UMBRAL_MS = 100;

void mergesort(int T[], int num_elem)
{
  mergesort_lims(T, 0, num_elem);
}

static void mergesort_lims(int T[], int inicial, int final)
{
  if (final - inicial < UMBRAL_MS)
    {
      insercion_lims(T, inicial, final);
    } else {
      int k = (final - inicial)/2;

      int * U = new int [k - inicial + 1];
      assert(U);
      int l, l2;
      for (l = 0, l2 = inicial; l < k; l++, l2++)
   U[l] = T[l2];
      U[l] = INT_MAX;

      int * V = new int [final - k + 1];
      assert(V);
      for (l = 0, l2 = k; l < final - k; l++, l2++)
   V[l] = T[l2];
      V[l] = INT_MAX;

      mergesort_lims(U, 0, k);
      mergesort_lims(V, 0, final - k);
      fusion(T, inicial, final, U, V);
      delete [] U;
      delete [] V;
    };
}
```

```
static void fusion(int T[], int inicial, int final, int U[], int V[])
{
  int j = 0;
  int k = 0;
  for (int i = inicial; i < final; i++)
    {
      if (U[j] < V[k]) {
   T[i] = U[j];
   j++;
      } else{
   T[i] = V[k];
   k++;
      };
    };
}

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

  if (argc != 2)
    {
      cerr << "Formato " << argv[0] << " <num_elem>" << endl;
      return -1;
    }

  int n = atoi(argv[1]);

  int * T = new int[n];
  assert(T);

  srandom(time(0));

  for (int i = 0; i < n; i++)
    {
      T[i] = random();
    };

  const int TAM_GRANDE = 10000;
  const int NUM_VECES = 1000;

  if (n > TAM_GRANDE)
    {
      clock_t t_antes = clock();

      mergesort(T, n);

      clock_t t_despues = clock();

      cout << n << "  " << ((double)(t_despues - t_antes)) / CLOCKS_PER_SEC
      << endl;
    } else {
      int * U = new int[n];
      assert(U);

      for (int i = 0; i < n; i++)
   U[i] = T[i];

      clock_t t_antes_vacio = clock();
      for (int veces = 0; veces < NUM_VECES; veces++)
   {
     for (int i = 0; i < n; i++)
       U[i] = T[i];
   }
```

```
   clock_t t_despues_vacio = clock();

      clock_t t_antes = clock();
      for (int veces = 0; veces < NUM_VECES; veces++)
   {
     for (int i = 0; i < n; i++)
       U[i] = T[i];
     mergesort(U, n);
   }
      clock_t t_despues = clock();
      cout << n << " \t  "
      << ((double) ((t_despues - t_antes) -
                (t_despues_vacio - t_antes_vacio))) /
   (CLOCKS_PER_SEC * NUM_VECES)
      << endl;

      delete [] U;
    }


  delete [] T;

  return 0;
};
```

fichero: mergesort.cpp

```
#!/bin/csh
@ inicio = 100
@ fin = 30000
@ incremento = 100
set ejecutable = mergesort
set salida = tiempos_mergesort.dat

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

fichero: ejecuciones_mergesort.csh

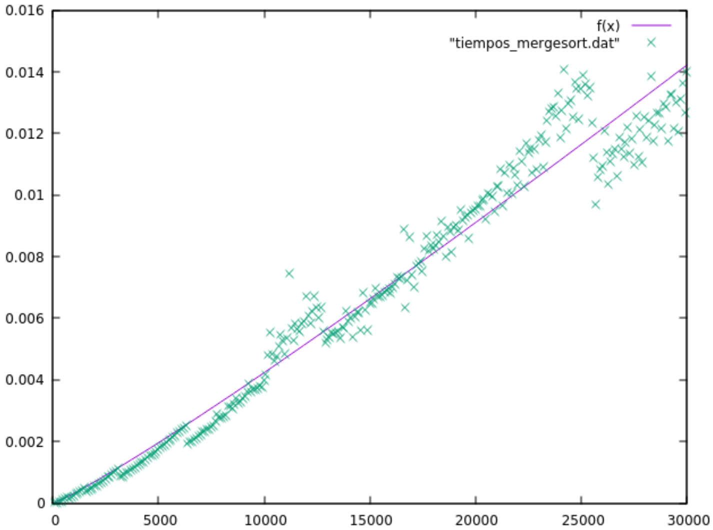
Compilamos el código de la siguiente manera:

```
g++ mergesort.cpp -o mergesort
```

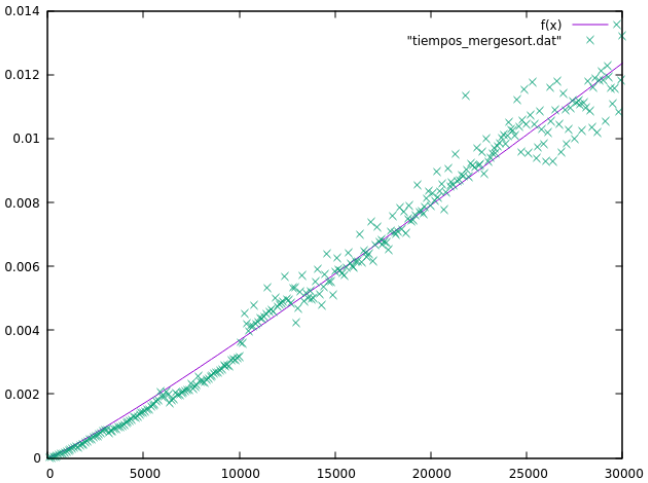
El ajuste de gráficas habrá que hacerlo a f(x)=axlog(bx). Nos quedan los siguientes valores:

a = 4.64592e-08
b = 0.901722

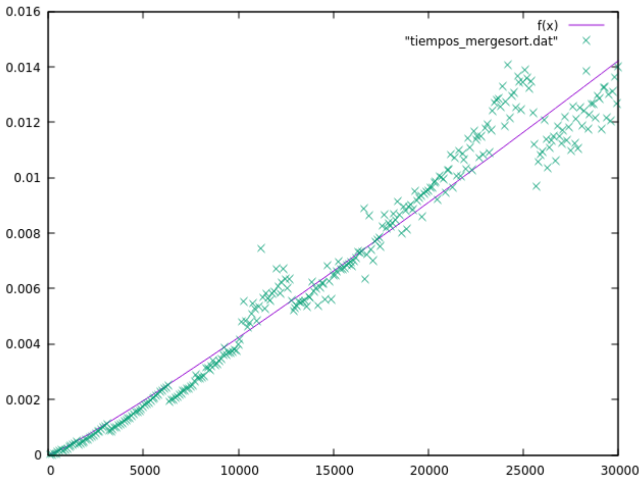
Y si pintamos la gráfica nos queda:



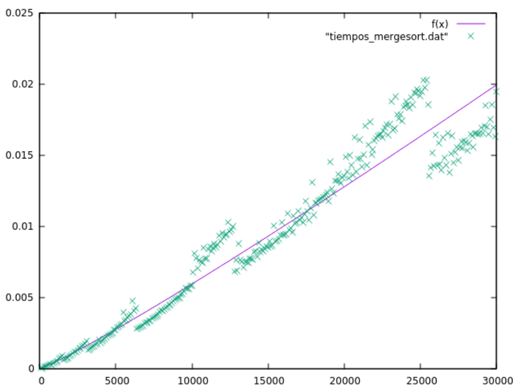
Para ver como afecta UMBRAL_MS a la eficiencia del algoritmo, hemos hecho distintas ejecuciones con distintos valores de UMBRAL_MS y hemos dibujado las gráficas correspondientes:



UMBRAL_MS = 50



UMBRAL_MS = 100



UMBRAL_MS = 200

Conforme subimos el UMBRAL_MS, aumenta el tiempo de ejecución (0.014, 0.016, 0.025), además la gráfica se vuelve más dispersa.

Equipo usado para el informe:

Sistema operativo: Ubuntu 18.04.1 LTS x86_64 (Xubuntu)
Host: HP Laptop 15-bw0xx
Kernel: 4.15.0-34-generic
CPU: AMD E2-9000e RADEON R2 2C+2G
GPU: AMD Device 98e4
RAM: 4 GB