Estructura de Datos

Práctica 1. Eficiencia de algoritmos

Elena Cantero Molina

Informe de la eficiencia

Hardware: Intel® Core™ i5-5200U CPU @ 2.20GHz × 4, 8GB de RAM
Sistema operativo: Ubuntu 16.04 de 64 bits
Compilador utilizado: g++ (Ubuntu 5.4.0-6ubuntu1~16.04.5) 5.4.0

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?

Solución.

Comencemos calculando la eficiencia teórica de este algoritmo, en el peor caso. Para ello, veamos el coste en operaciones elementales (OE) de cada línea:

- Línea 2: 5 OE (asignación, resta, comparación, incremento(incremento + asignación)).

- Línea 3: 6 OE (igual que la línea anterior pero con una resta más).
- Línea 4: 4 OE (2 acceso a vector, suma, comparación).
- Línea 5: 3 OE (declaración, asignación, acceso).
- Línea 6: 4 OE (2 accesos, suma, asignación).
- Línea 7: 3 OE (acceso, suma y asignación).

Entonces, la eficiencia del algoritmo es:

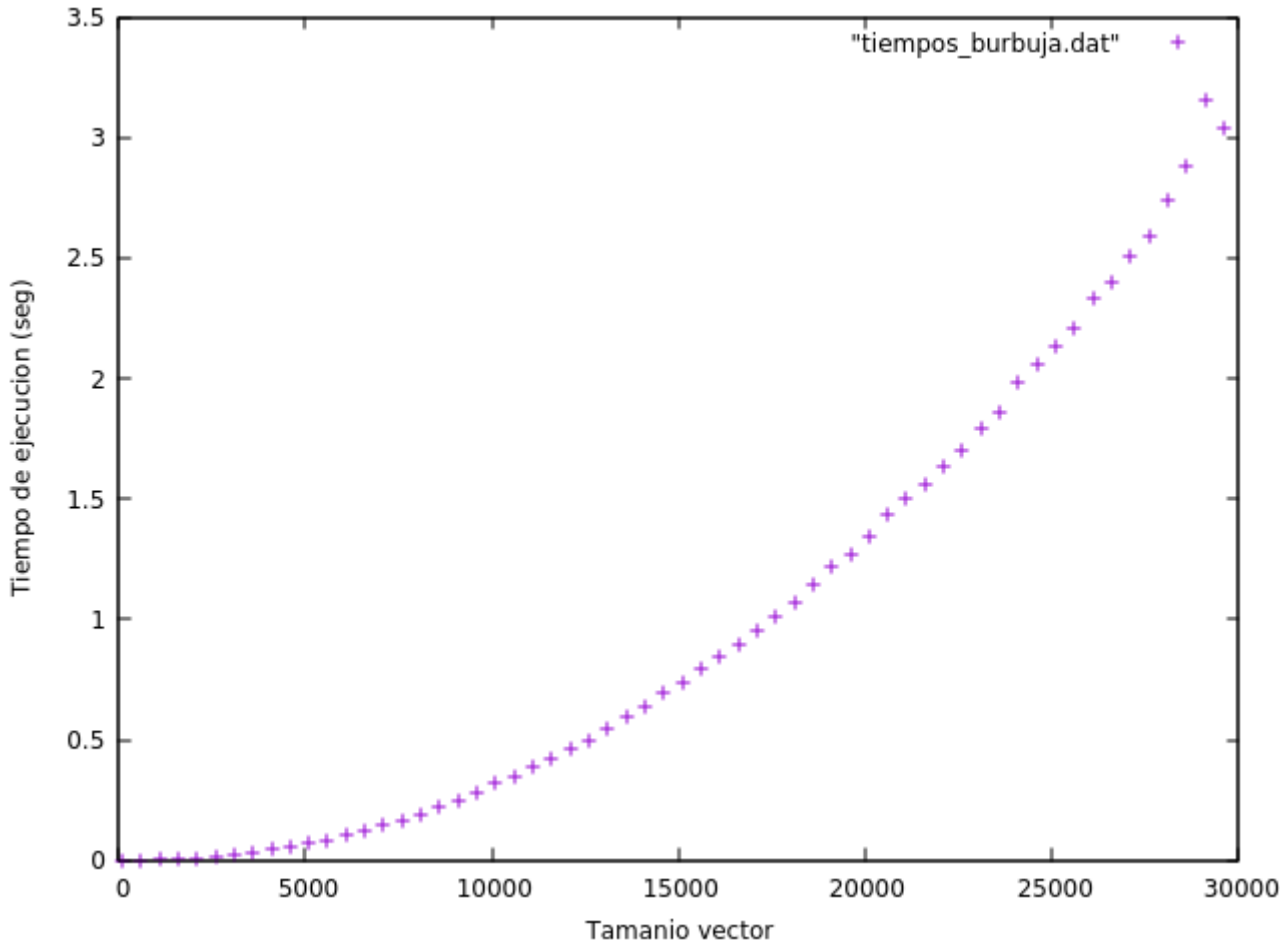
$$T(n)=\sum_{i=0}^{n-2}\sum_{j=0}^{n-i-2}1=\sum_{i=0}^{n-2}n-\sum_{i=0}^{n-2}i-\sum_{i=0}^{n-2}1=n(n-1)-\frac{(n-2)(n-1)}{2}-n+1=n^2-2n+1$$
$$-\frac{n^2-3n+2}{2}=\frac{2n^2-4n+2-n^2+3n-2}{2}=\frac{n^2-n}{2}$$

Por tanto, afirmamos que $T(n)\in O(n^2)$.
Ahora creamos un programa de prueba para analizar la eficiencia empírica, haciendo uso de la biblioteca *ctime* del lenguaje C++:

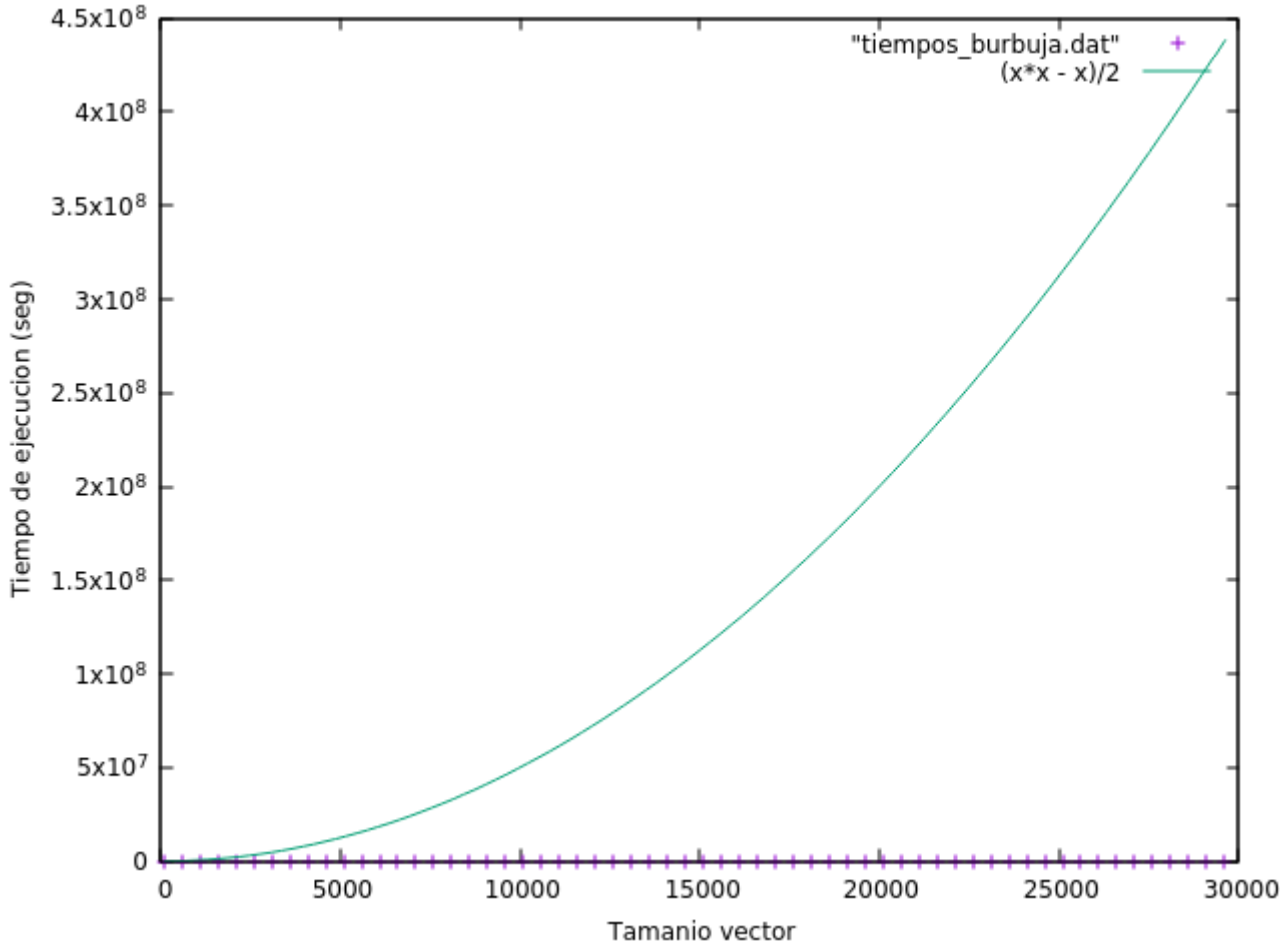
```
#include <iostream>
#include <ctime>    // Recursos para medir tiempos
#include <cstdlib>  // Para generación de números pseudoaleatorios
using namespace std;

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();
  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
}
```

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



Si representamos superpuestas la función de eficiencia teórica y la empírica obtenemos lo siguiente:



Observamos que, aunque la curva de la eficiencia teórica tiene la misma forma que la nube de puntos obtenida experimentalmente, ambas curvas no coinciden al representarlas superpuestas. Esto se debe a que las constantes no coinciden, pues el costo de una operación elemental, e incluso el tiempo total de ejecución del algoritmo, dependen de las condiciones particulares de la máquina en la que se ejecuta.

Ejercicio 2. Ajuste de la ordenación 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 $ax^2 + bx + c$

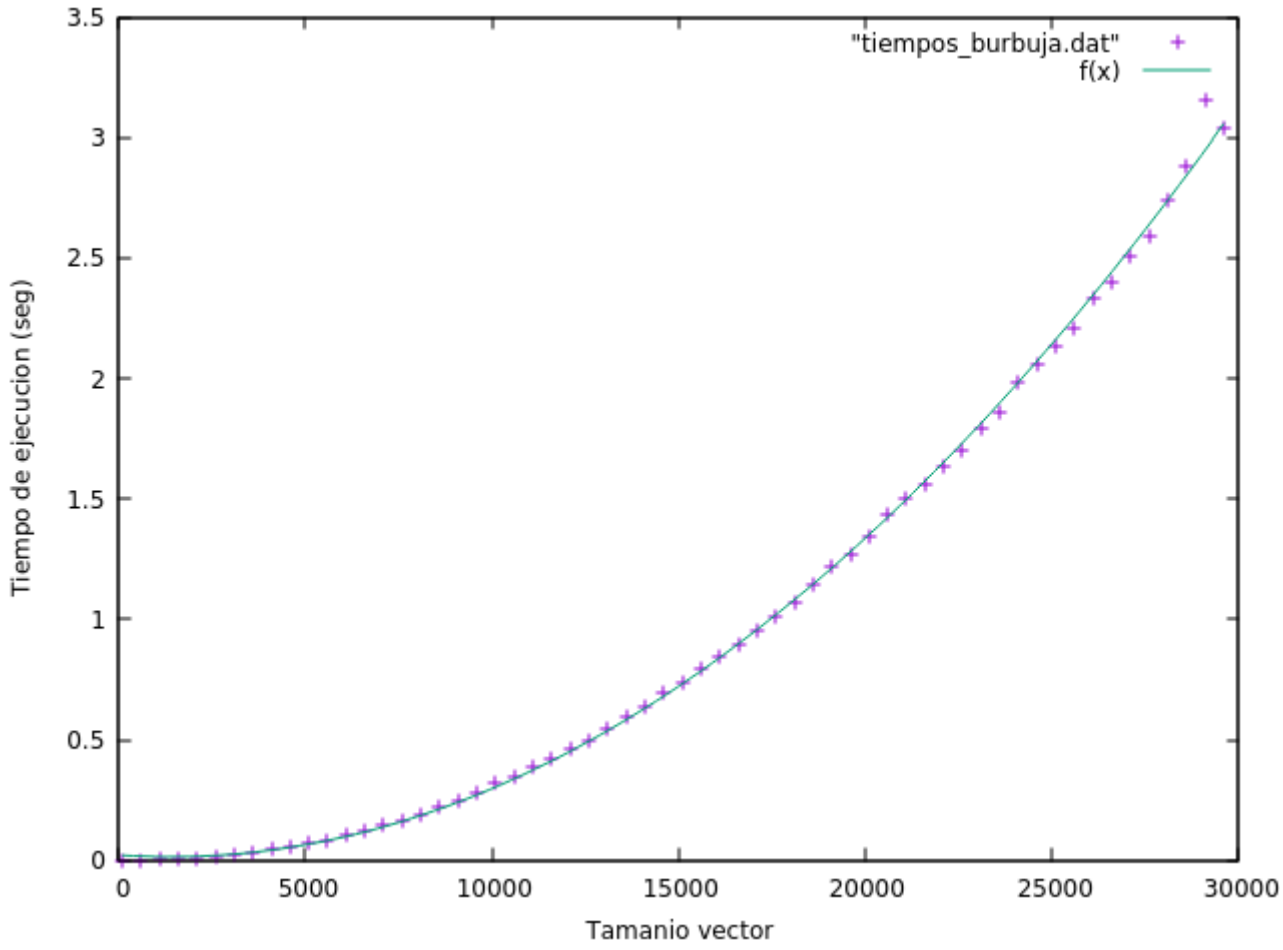
Solución.

Veamos que podemos ajustar, mediante gnuplot, los puntos obtenidos al analizar la eficiencia empírica en una función de la forma $f(n) = an^2 + bn + c$.

En efecto, obtenemos que la función en cuestión es:

$f(n) = 3{,}82269\,e^{-9}\,n^2 - 1{,}06888\,e^{-5}\,n + 0{,}0218994$

Observamos que las constantes obtenidas son mucho más pequeñas que las calculadas en la eficiencia teórica. La representación gráfica es la siguiente:



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:

1. Explique qué hace este algoritmo.
2. Calcule su eficiencia teórica.
3. 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 regresión para ajustar la curva teórica a la empírica.

Solución

1. Explicación

El algoritmo en cuestión es la **búsqueda binaria**, donde, dado un vector *v* de *n* elementos enteros, se busca el entero *x* en él. Para ello, se pasan como parámetros el inicio(*inf*) y el final(*sup*) de dicho vector. La función devuelve la posición donde se ha encontrado el elemento, o -1 si no se encontraba en el vector.

Para el correcto funcionamiento del algoritmo, es obligatorio que el vector que se pasa, esté ordenado, pues el procedimiento es el siguiente:

a) Se establece la posición *med = (inf + sup)/2*.

b) Se comprueba si el elemento x está en la posición *med*.

c) Si está, hemos acabado. En caso contrario, se comprueba si el elemento *v[med]* es mayor o menor que *x*.
- Si *v[med]< x, inf = med + 1*.
- Si *v[med]> x*, *sup = med - 1*.

d) repetimos este proceso hasta encontrar el elemento, o concluir que no está en el vector.

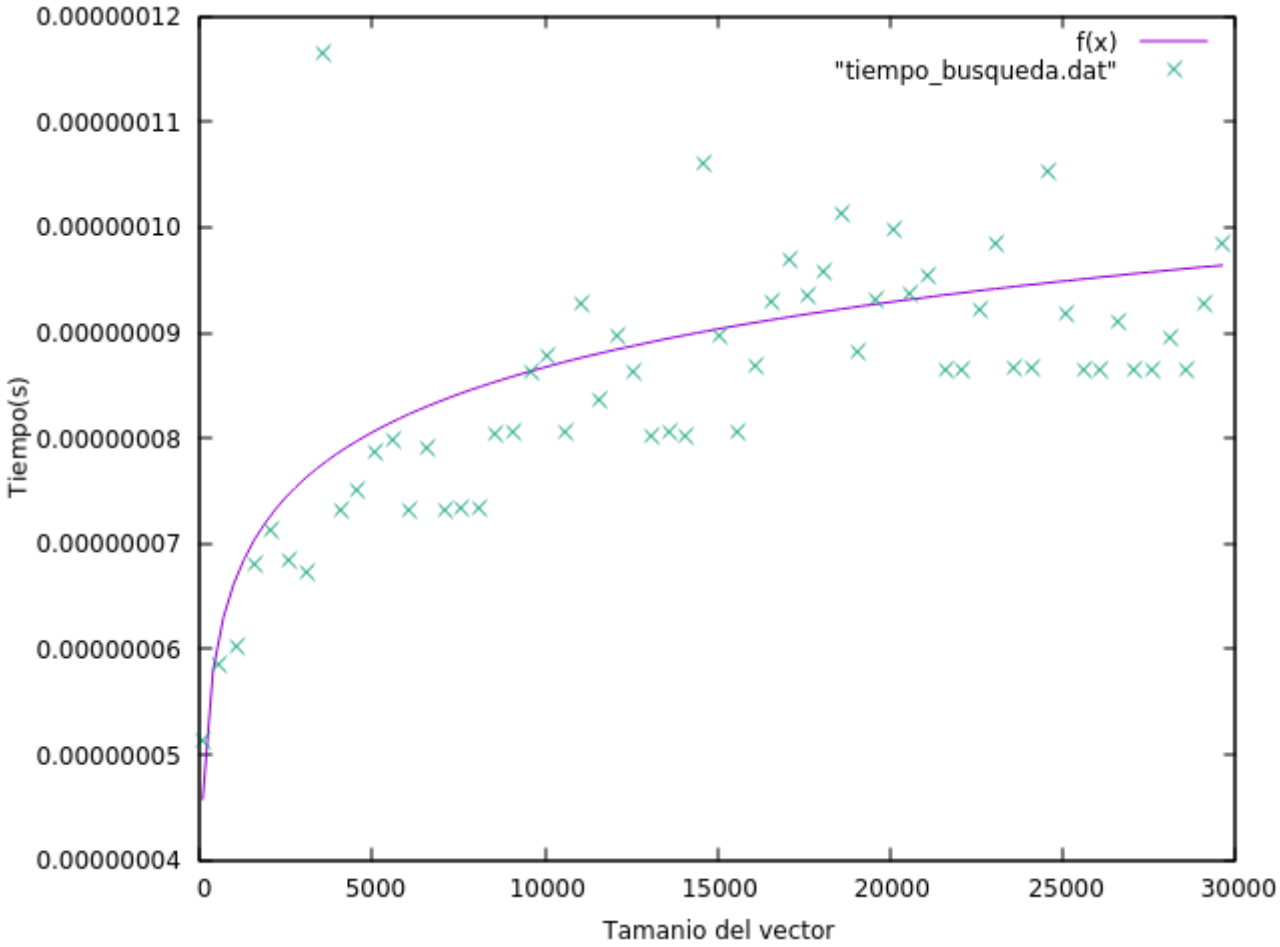
2. Eficiencia Teórica

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

Por definición, el número máximo de veces que se puede dividir por la mitad un vector de tamaño *n* es *log_2(n)* veces. como el resto de operaciones son elementales *O(1)*, concluimos que la eficiencia del algoritmo es logarítmica, es decir, $T(n) \in O(\log(n))$.

3. Eficiencia Empírica

Para analizar la eficiencia empírica, no podemos simplemente repetir el proceso que hemos seguido anteriormente. Si lo hiciéramos así, obtendríamos una gráfica casi recta.
Esto se debe a que el algoritmo es demasiado rápido y es imposible apreciar cambios en el tiempo de ejecución para vectores de mayor tamaño. Por tanto, para analizarlo empíricamente, debemos ejecutarlo más de una vez por cada tamaño del vector. En nuestro caso, lo hemos ejecutado 10000 veces por cada tamaño, obteniendo la siguiente gráfica:



Ajustando la nube de puntos a la función logarítmica $f(n) = 8.22027\,\mathrm{e}^{-9}\,(\log(0.925707\,x)) + 9.40202\,\mathrm{e}^{-9}$, podemos observar claramente que la eficiencia empírica describe una gráfica de forma logarítmica.

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:

a) 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.

b) 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.

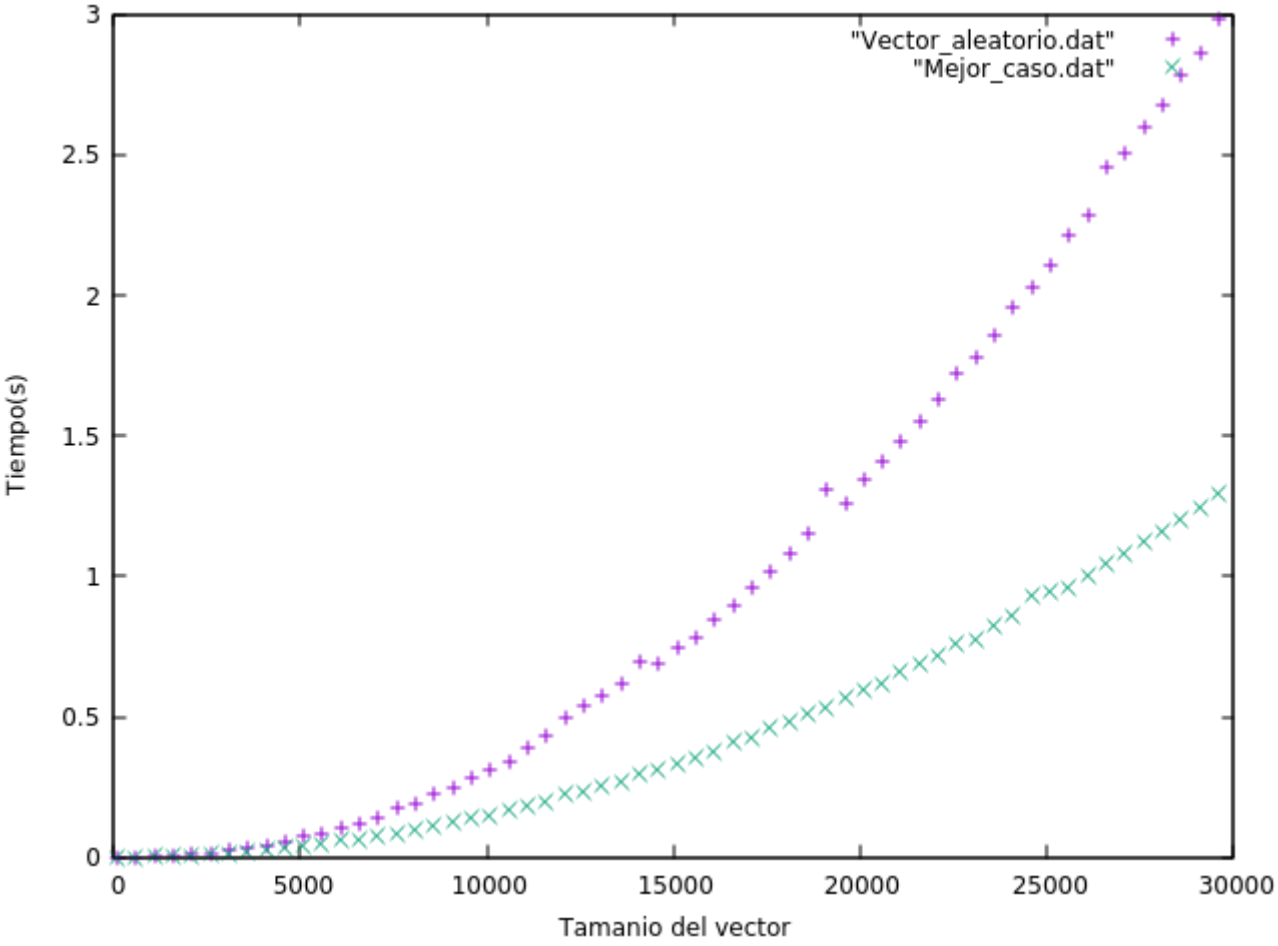
Calcule la eficiencia empírica en ambos escenarios y compárela con el resultado del ejercicio 1.

Solución

a) Para este apartado, construimos un vector ordenado para representar el mejor caso. Usaremos, por ejemplo, el siguiente código, y elegiremos *vmax > tam*:

```
//Generación de un vector ordenado
int *v = new int[tam];          //Reserva de memoria
v[0] = 0;
for(int i = 1; i < tam; i++)    //Recorrer el vector
   v[i] = v[i – 1] + 1;         //Generar vector ordenado
```

La gráfica que representa la eficiencia empírica es la siguiente:

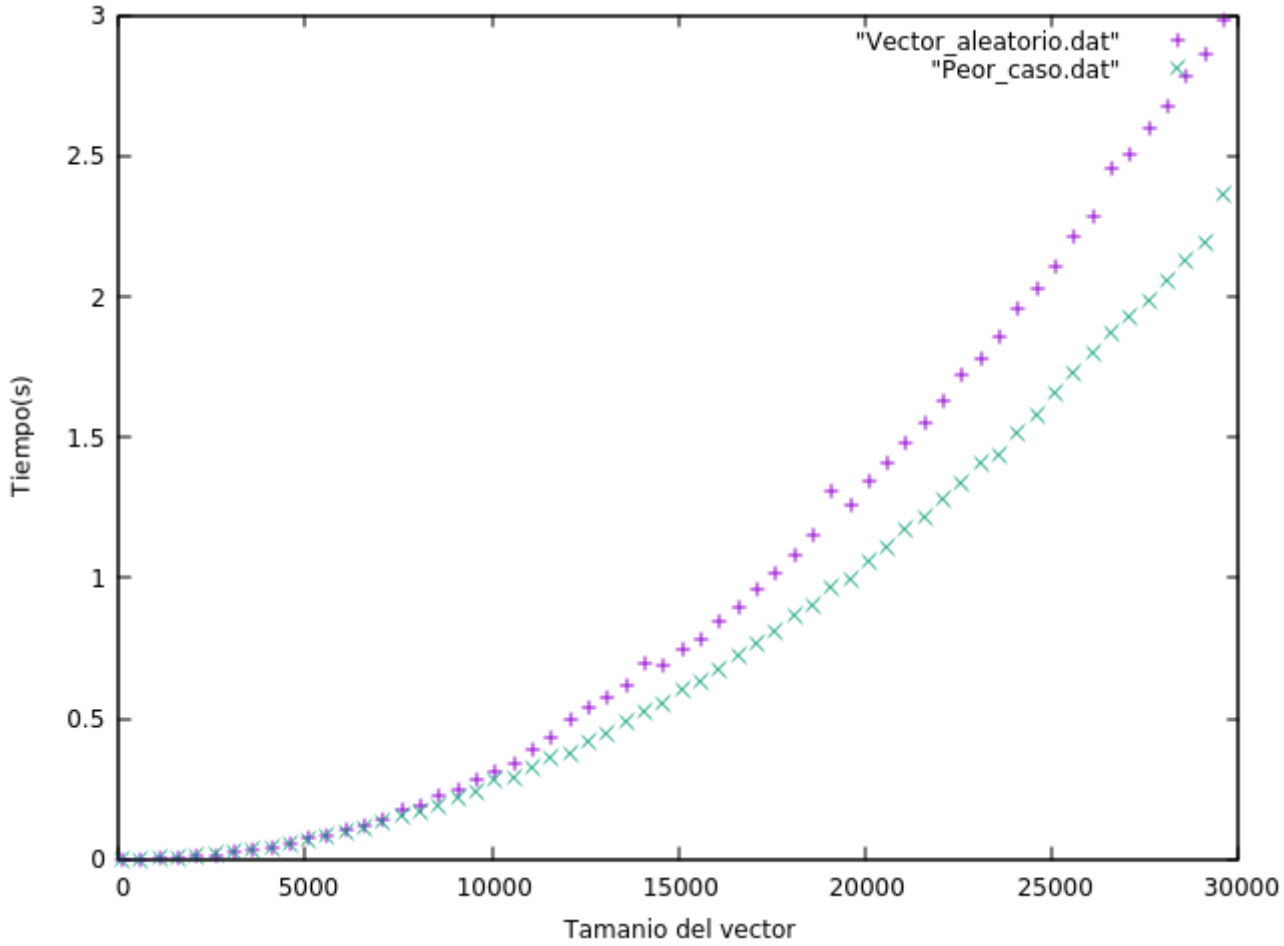


Como se puede observar, el tiempo de ejecución es mucho menor para el mejor caso que para el vector con los elementos generados aleatoriamente.

b) Para este apartado, construimos un vector ordenado en orden inverso, para representar el peor caso. Usaremos el siguiente código:

```
//Generación de un vector ordenado
int *v = new int[tam];          //Reserva de memoria
v[0] = vmax;
for(int i = 1; i < tam; i++)    //Recorrer el vector
   v[i] = v[i – 1] - 1;         //Generar vector ordenado inversamente
```

La gráfica que representa la eficiencia empírica es la siguiente:



Observando esta gráfica nos damos cuenta de algo extraño. A pesar de encontrarnos en el 'caso peor', la ordenación de este vector es más rápida que la del vector aleatorio. Al observar esto, vemos que es más eficiente el vector aleatorio que nuestro 'peor caso'.

Ejercicio 5. Dependencia de la implementación

Considere esta otra implementación 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;
```

```
            }
          }
        }
```

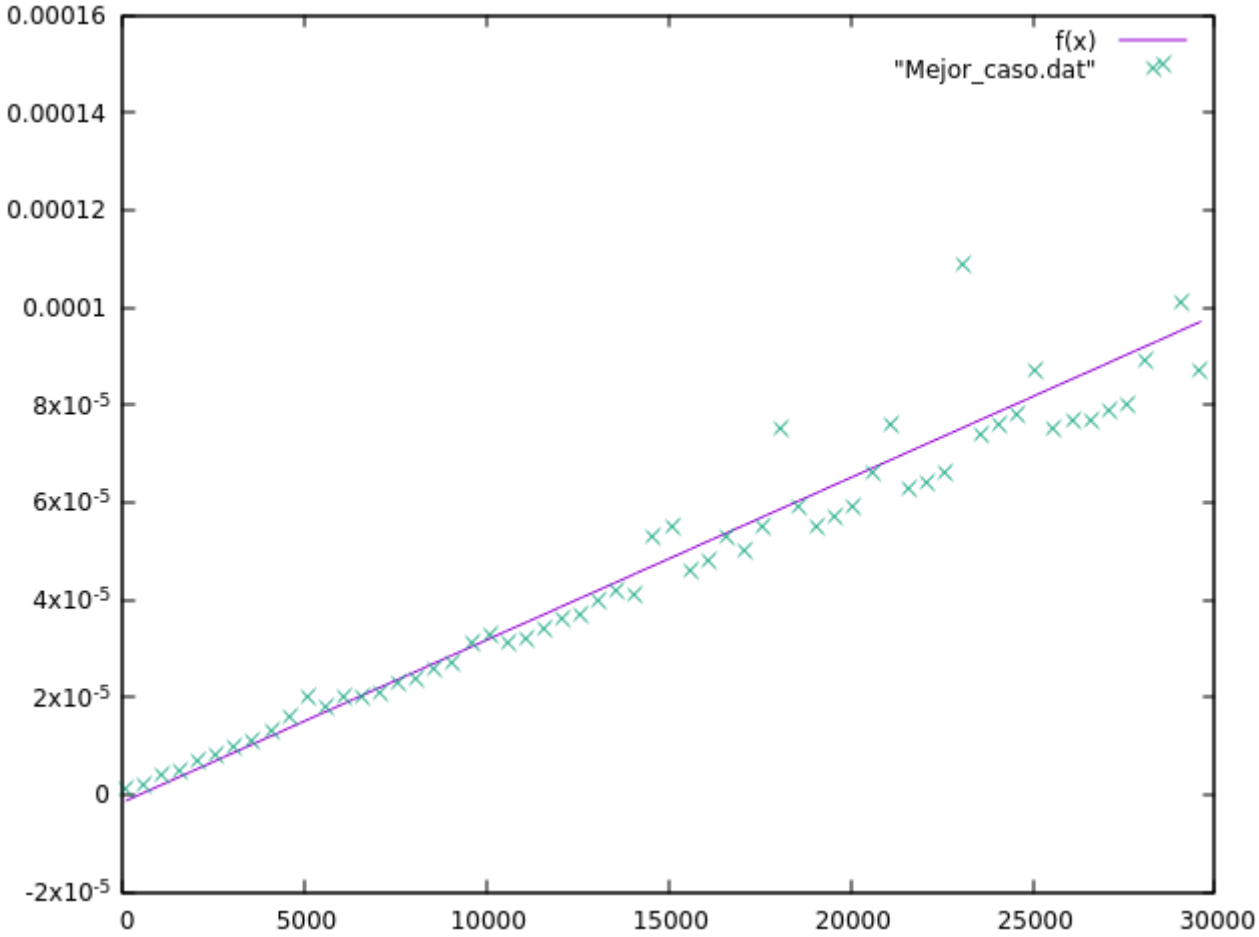
En ella se ha introducido una variable que permite saber si, en una de las iteraciones del bucle externo no se ha modificado el vector. Si esto ocurre significa que ya está ordenado y no hay que continuar.

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.

Solución.

Si realizamos el estudio teórico de la eficiencia en el mejor caso, nos damos cuenta rápidamente de que $T(n) \in O(n)$.

Ahora analizamos la eficiencia empírica del algoritmo en cuestión. Veamos la gráfica asociada:



Como podemos ver, el orden de eficiencia es efectivamente lineal, con una función de ajuste $f(n) = 3.33259\,\mathrm{e}^{-9} n + (-1.62233\,\mathrm{e}^{-6})$.

Ejercicio 6. Influencia del proceso de compilación

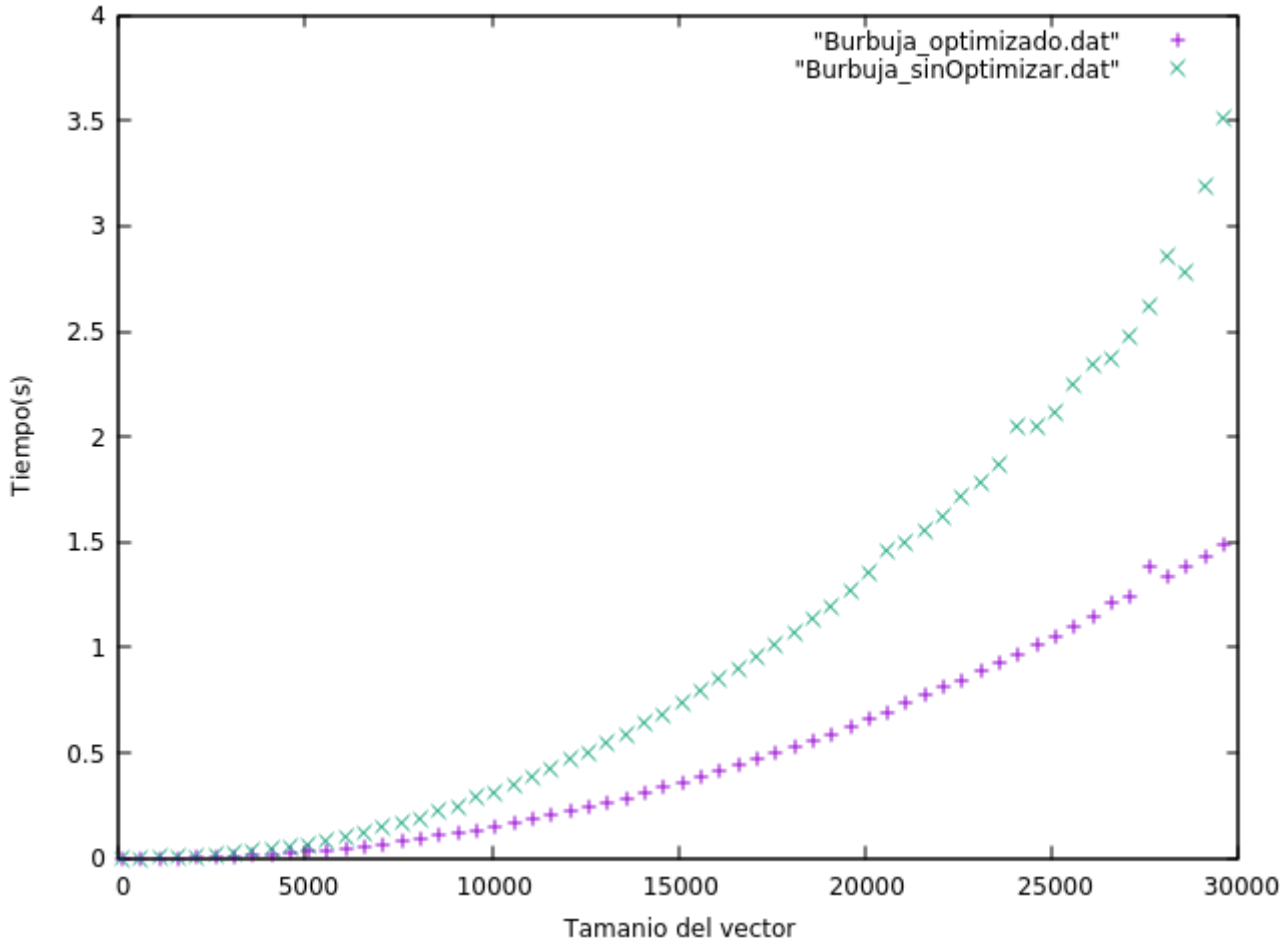
Retome el ejercicio de ordenación mediante el algoritmo de la burbuja. Ahora replique dicho ejercicio pero previamente deberá compilar el programa indicándole al compilador que optimice el código. Esto se consigue así:

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

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

Solución.

Compilando el algoritmo de la burbuja con la opción -O3 conseguimos que el compilador optimice el código de nuestro programa. Así, midiendo el tiempo de ejecución para distintos tamaños del vector y comparándolo con el que obtuvimos en el ejercicio 1, obtenemos la siguiente gráfica:



Podemos observar claramente que el programa con las optimizaciones es más rápido que sin ellas, como era de esperar.

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.

Solución.

Para calcular la eficiencia teórica primero usaremos el siguiente código para ver las operaciones elementales:

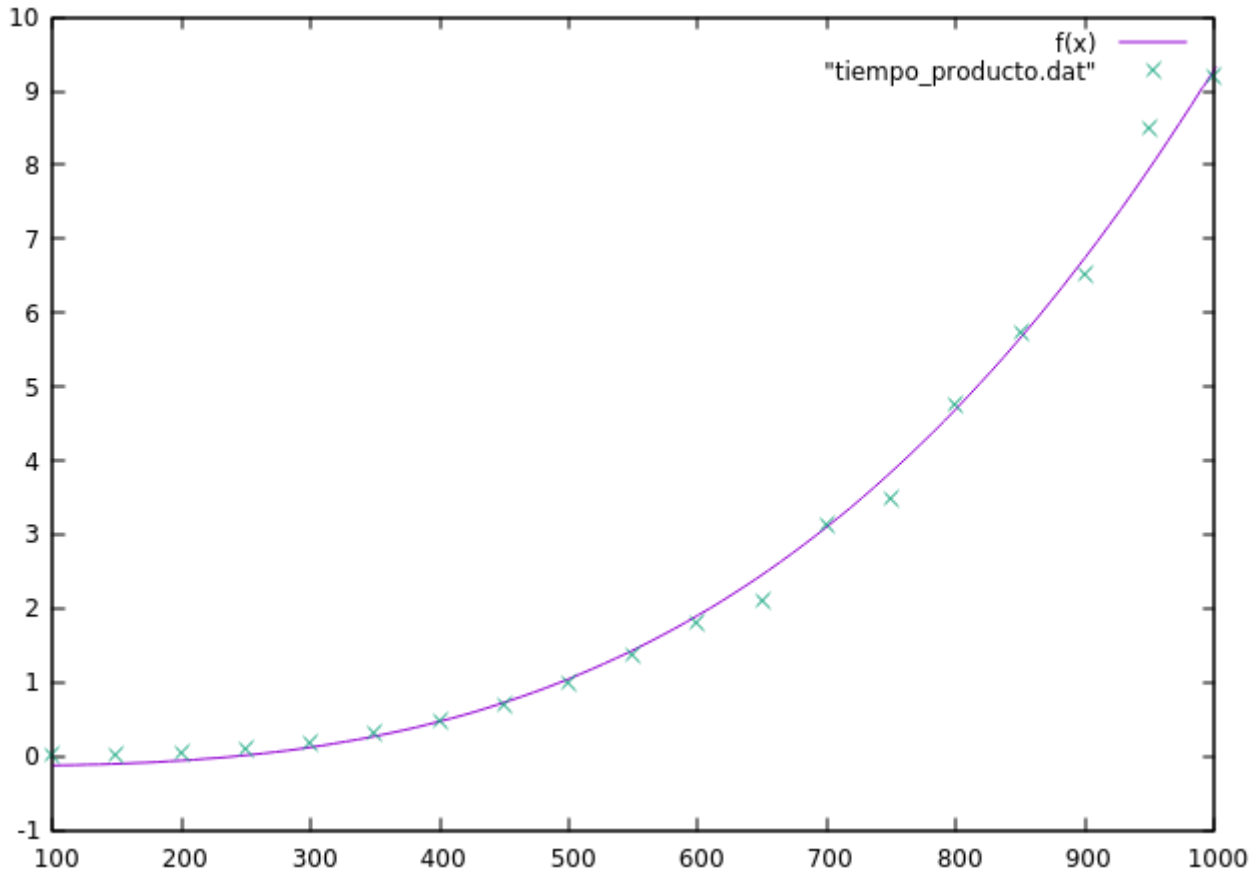
```
1    for (i = 0; i < N; i++)
2      for (j = 0; j < N; j++)
3        for (k = 0; k < N; k++)
4          A[i][j] += B[i][k] * C[k][j];
```

En las líneas 1, 2 y 3 encontramos 3 OE (asignación, comparación e incremento)
Y en la línea 4 nos encontramos con 6 OE (3 accesos a vector, suma, asignación y producto).
Entonces la eficiencia del algoritmo es:

$$T(n) = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} \sum_{k=0}^{N-1} 1 = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} N = N \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} 1 = N^2 \sum_{i=0}^{N-1} 1 = N^3$$

Por lo tanto, $T(n) \in O(n^3)$

Al analizar la eficiencia empírica del algoritmo junto con el ajuste teórico, obtenemos la siguiente gráfica:



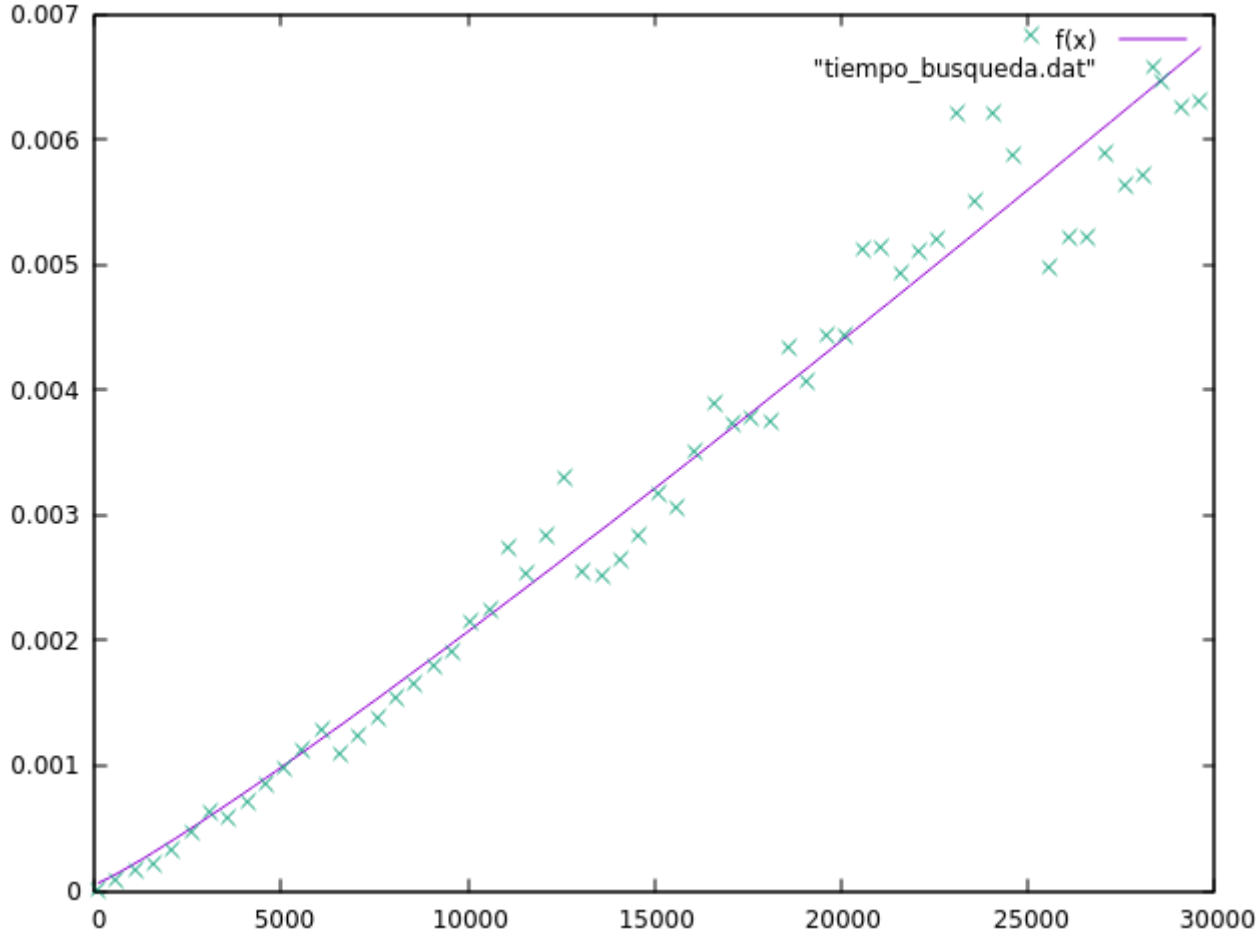
Ajustando la nube de puntos a la función cúbica $f(n) = 9.45222\mathrm{e}^{-9}x^3 - 0.137314$, podemos observar claramente que la eficiencia empírica describe una gráfica de forma cúbica.

Ejercicio 8. Ordenación por mezcla

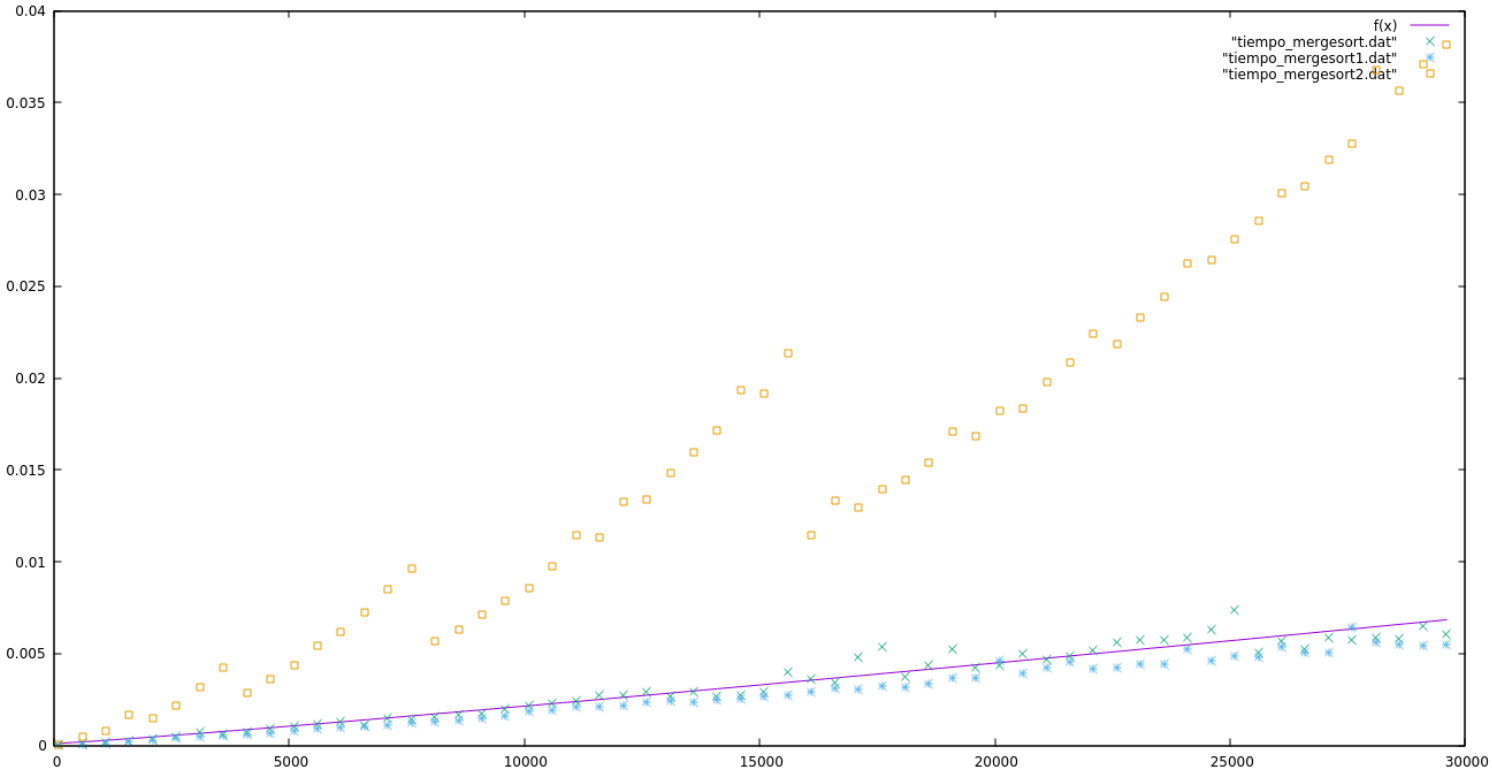
Estudie el código del algoritmo recursivo disponible en el fichero mergesort.cpp. En él, se integran dos algoritmos de ordenación: inserción y mezcla (o mergesort). El parámetro UMBRAL_MS condiciona el tamaño mínimo del vector para utilizar el algoritmo de inserción en vez de seguir aplicando de forma 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.

Solución.

Al realizar la eficiencia empírica del algoritmo mergesort y hacer el ajuste de ambas curvas obtenemos lo siguiente:



Como podemos ver, el orden de eficiencia es casi lineal y logarítmica, con una función de ajuste $f(n) = 2.19074\,\mathrm{e}^{-8}\, n \log(n) + 5.20003\,\mathrm{e}^{-5}$. Esta gráfica está hecha con el dato de UMBRAL_MS sin modificar.
Ahora, veamos como afecta modificar UMBRAL_MS a la gráfica siguiente:



Como se puede observar, al ser UMBRAL_MS más pequeño que el dado, se ajusta bien al ajuste antes dado. Sin embargo, al aumentarlo, se separa del ajuste.