Práctica Eficiencia

Francisco Coca Cruz - 53911747T
Adrián Sánchez Cerrillo - 76655183R

2º Ingeniería Informática – Grupo B1

Tarea 1:

Primero calculamos la eficiencia teórica:

Queremos calcular la eficiencia teórica del algoritmo siguiente:

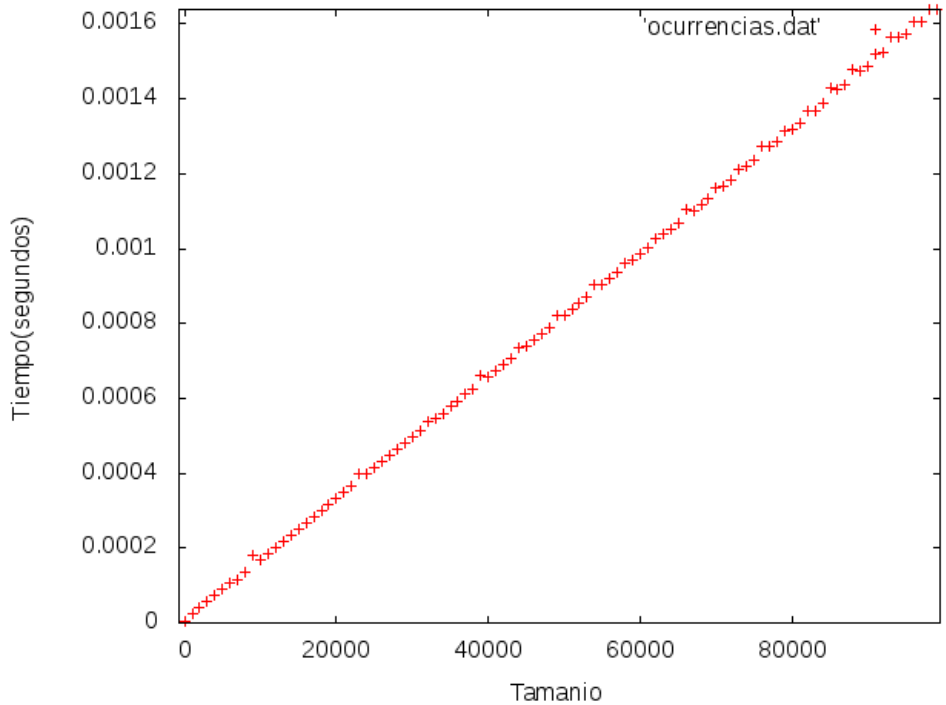
```
int contar_hasta( vector<string> & V, int ini, int fin, string & s) {
  int cuantos = 0;
  for (int i=ini; i< fin ; i++)
    if (V[i]==s) {
      cuantos ++;
    }
  return cuantos;
}
```

Calculamos su eficiencia en función de "*n=fin-ini*"; tal y como podemos observar las operaciones ejecutadas fuera del bucle 'for' son constantes, con eficiencia O(1), al igual que con el condicional 'if', luego el orden de tiempo de ejecución T(n) será:

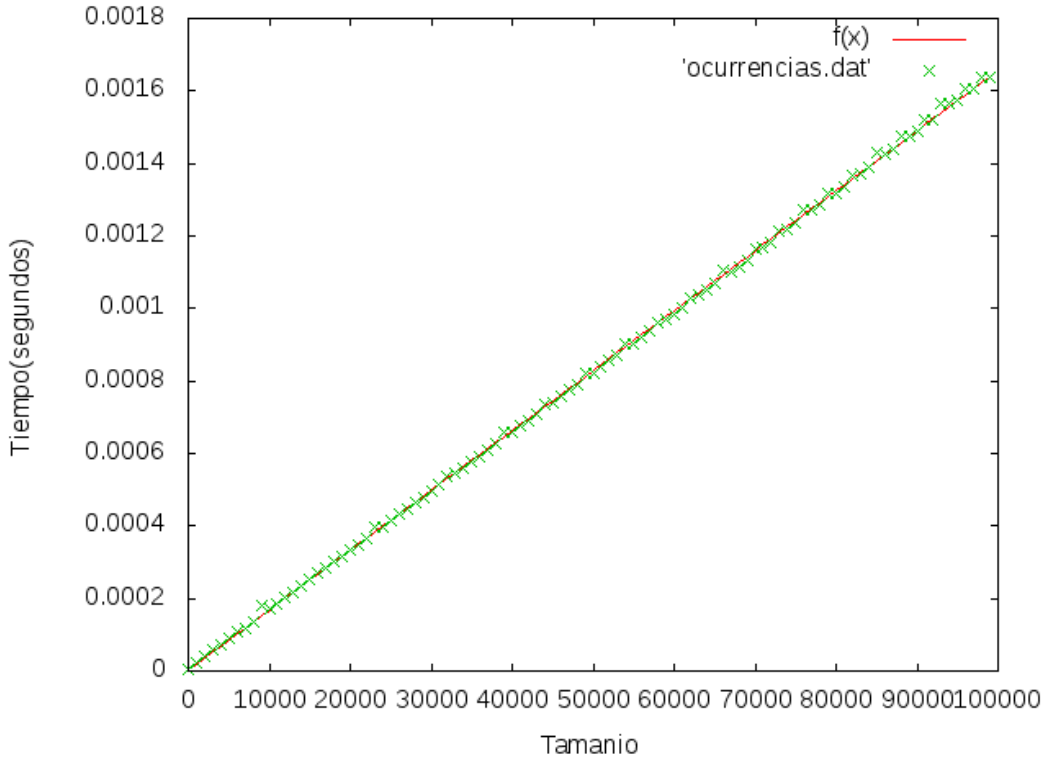
$1 + \sum_{i=1}^{n} 1 = 1 + n$ que pertenece a **O(n)**.

Pasemos ahora a calcular la eficiencia empírica:
Las siguientes mediciones han sido realizadas con un ordenador con las siguientes características:
Memoria: 8 GB DDR4
Procesador: Intel® Core™ i5 6300-HQ CPU @2.3 GHz x 4
SO: Linux Mint 18 Cinnamon 64 bits

Resultados de la ejecución



Ajustamos nuestra eficiencia teórica de orden **O(n)** a los datos obtenidos. Y concluimos que el ajuste es Y= 2.25418e-06 + X*1.6536e-08.



Tarea 2:

Eficiencia teórica:

- **Versión 1**

```
void contar_frecuencias_V1( vector<string> & libro, int ini, int fin, vector<string> &pal, vector<int> &
frec ){
        int cuantas;
        for (int i = ini; i<fin; i++)
        {
                cuantas = contar_hasta(libro,ini,fin,libro[i]);
                pal.push_back(libro[i]);
                frec.push_back(cuantas);
        }
}
```

Consideramos n = fin-ini. Hemos calculado la eficiencia de la función "*contar_hasta(..)*", siendo O(n) su eficiencia (en este caso las 'n' son la misma).
Tambien sabemos que la eficiencia de "*push_back(..)*" es **O(1)**. Por tanto el tiempo de ejecución T(n) viene dado por:

$\sum_{i=1}^{n}(n+1) = 1 + n + n^2$ que pertenece a **O(n²)**

→ **Pregunta:** ¿Proporciona la salida correcta?
El algoritmo no es correcto ya que cuando aparece una palabra por segunda vez, vuelve a crear una entrada de esta misma palabra en "pal" con su frecuencia en "frec", es decir, está repitiendo información.

- **Versión 2**

Primero analizamos la eficiencia de buscar:

```
int buscar( vector<string> & V, string & s) {
        bool enc= false;
        int pos = POS_NULA;
        for (int i=0; i< V.size() && !enc; i++){
                if (V[i]==s) {
                        enc = true;
                        pos = i;
                }
        }
        return pos;
}
```

Las instrucciones de dentro del for y de fuera tienen orden de ejecución constante. Por tanto, el tiempo de ejecución T(n) viene dado por:

$1 + \sum_{i=1}^{n} 1$ que es claramente orden **O(n)**, donde aquí 'n' es el tamaño del vector V(*pal)*.

Volvemos a la función:

```
void contar_frecuencias_V2( vector<string> & libro, int ini, int fin,
                            vector<string> &pal, vector<int> & frec ){
  int pos;
  for (int i = ini; i<fin; i++){

    pos = buscar(pal, libro[i]);
    if (pos==POS_NULA) {
      pal.push_back(libro[i]);    // Analisis amortizado O(1)
      frec.push_back(1);    // Analisis amortizado O(1)
    }
    else {
      frec[pos]++;
    }

  }
}
```

Vemos que el peor caso posible sería que en cada iteración libro[i] no estuviera en “pal”, con lo que pal aumentaria su tamaño en 1 unidad cada iteración, es decir la iteración i tendría tamaño “i”. Por tanto, el tiempo de ejecución va a ser:
que es de orden **O(n²)**.

- **Versión 3**

```
void contar_frecuencias_V3( vector<string> & libro, int ini, int fin,
                  vector<string> &pal, vector<int> & frec ){
 vector<string>::iterator pos;
 for (int i = ini; i<fin; i++){
   pos = lower_bound(pal.begin(), pal.end(), libro[i]);  // O(log (n) ), n tama del vector  Busqueda
                                                         // Binaria
   if ((pos==pal.end()) || (*pos!=libro[i])) {
      frec.insert(frec.begin() + (pos-pal.begin()), 1);   //O(n)
      pal.insert(pos, libro[i]);          //O (n)
      }
   else {
      frec[pos-pal.begin()]++; // O(1)
      }
   }
}
```

Nos ponemos en el peor caso posible, esto será cuando en cada iteración la nueva palabra no haya sido ya leida(no aparece en pal), por lo tanto el tamaño de pal aumenta en uno en cada iteración, es decir, en la iteración i tendrá un tamaño i. Como el tiempo de ejecución de lower_bound será del orden **O(log(tamaño de pal))** y dentro del if lo más grande tiene frecuencia **O(n)** siendo n el tamaño del vector (Nótese que tamaño de pal = tamaño de frec); llegamos a que el orden de eficiencia de nuestro algoritmo viene dado por

$$1 + \sum_{i=1}^{n} (i + log(i) + 1) = 1 + n(2 + n + log(n) + 1)/2 = 1 + (n^2 + nlog(n) + 3)/2$$

Y por tanto hemos obtenido que tiene eficiencia **O(n²)**

- **Versión 4**

```
void contar_frecuencias_V4( vector<string> & libro, int ini, int fin,
                            vector<string> &pal, vector<int> & frec ){

  map<string,int> M;
  for (int i = ini; i<fin; i++)
    M[libro[i]]++;        // O( log(n) )

  map<string,int>::iterator it;
  for (it = M.begin(); it!= M.end(); ++ it){ // Bucle O(k log k) siendo k el numero de palabras
                                             //  distintas
    pal.push_back( (*it).first );
    frec.push_back( (*it).second );
  }
}
```

El peor caso posible vuelve a ser que no se repita ninguna palabra. Nos encontramos un primer bucle en el que en cada iteración nos cuesta log(n) acceder, si estamos en el peor caso M irá aumentando en cada iteración y al igual que antes tendremos que tendrá longitud i en la iteración i, además sabemos que nos cuesta log(i) la iteración i. El segundo bucle nos dicen su orden de eficiencia, que como hemos supuesto que no se repite ninguna palabra k=n. Con esto vemos que el orden de eficiencia viene dado por:

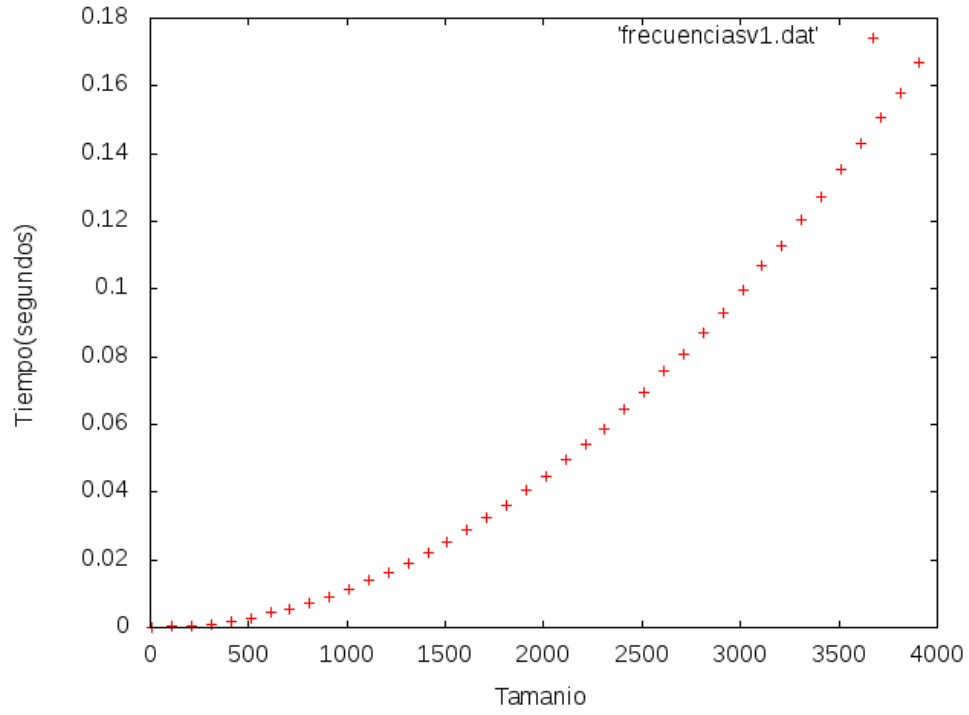
$$1 + \sum_{i=1}^{n}(1 + log(i)) + nlog(n) = 1 + n * (1 + 1 + log(n))/2 + nlog(n)$$

Hemos obtenido que tiene eficiencia **O(n*log(n))**

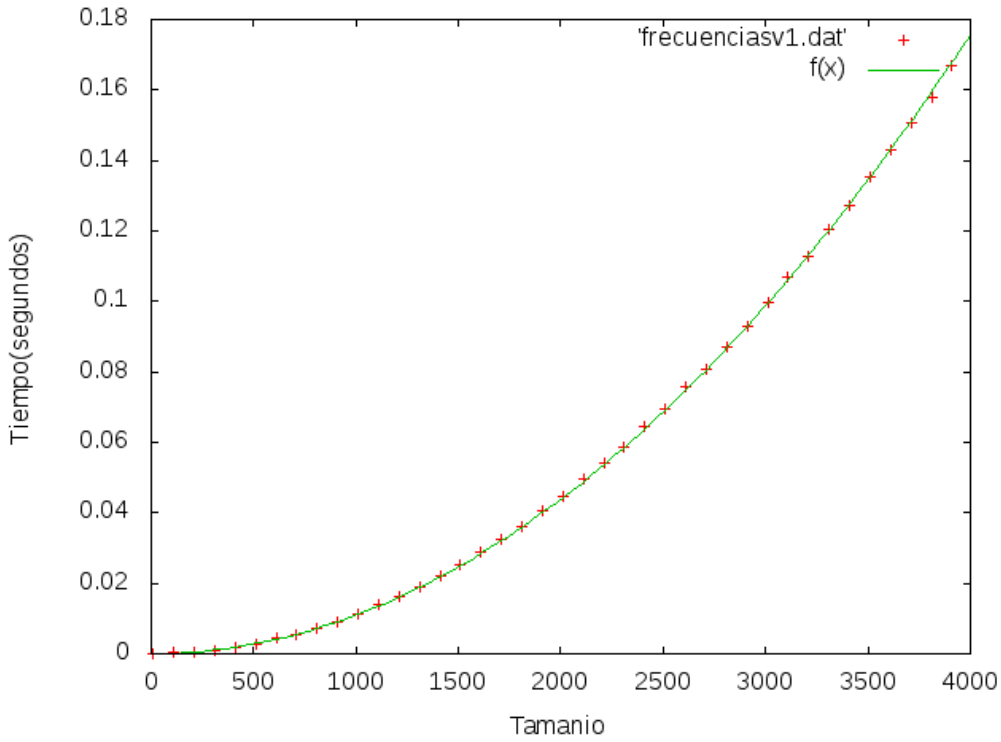
Eficiencia empírica:

- **Versión 1**

Resultados de la ejecución:

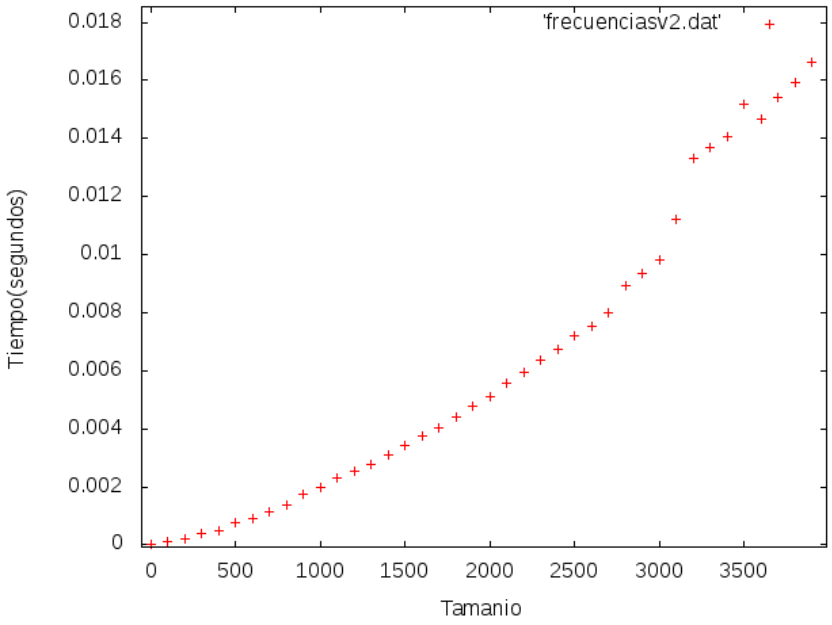


Ajustamos nuestra eficiencia teórica de orden $O(n^2)$ a los datos obtenidos. Y concluimos que el ajuste es Y= **1.09772e-08**$*X^2$



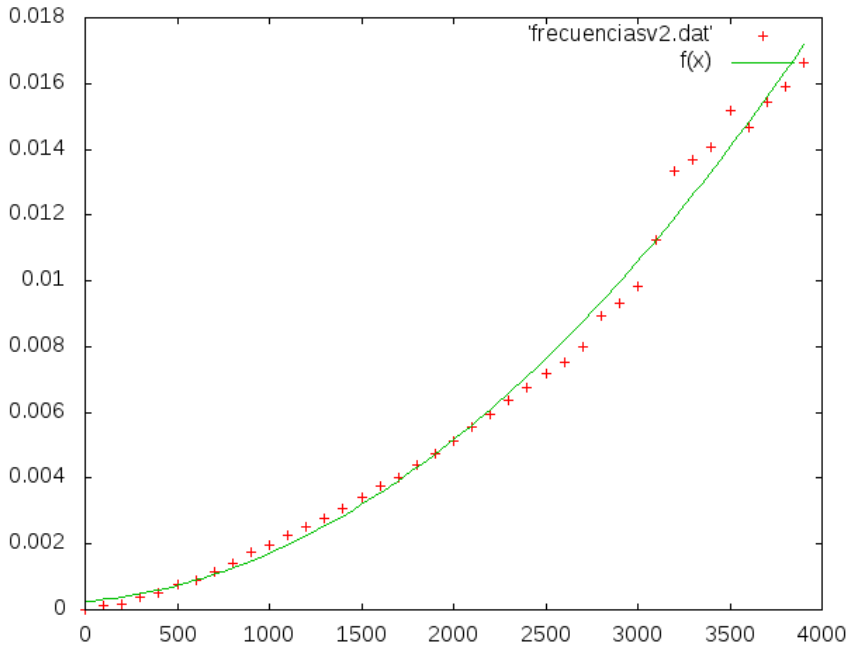
- **Versión 2**

Resultados de la ejecución:



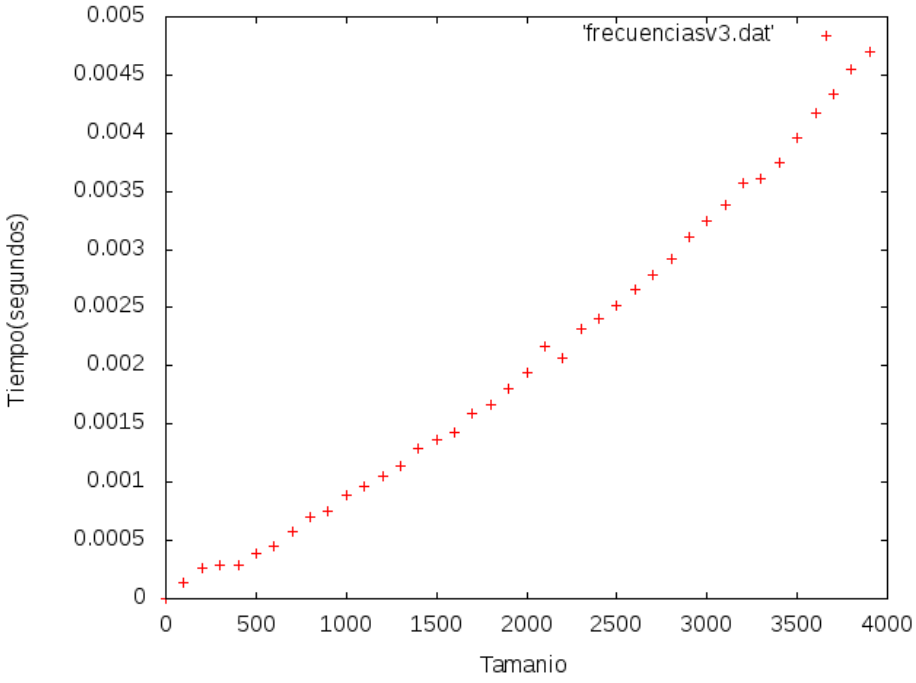
Ajustamos nuestra eficiencia teórica de orden $O(n^2)$ a los datos obtenidos. Y concluimos que el ajuste es $Y = a*X^2 + b*X + c$

a = 9.88972e-10
b = 4.82454e-07
c = 0.000251167



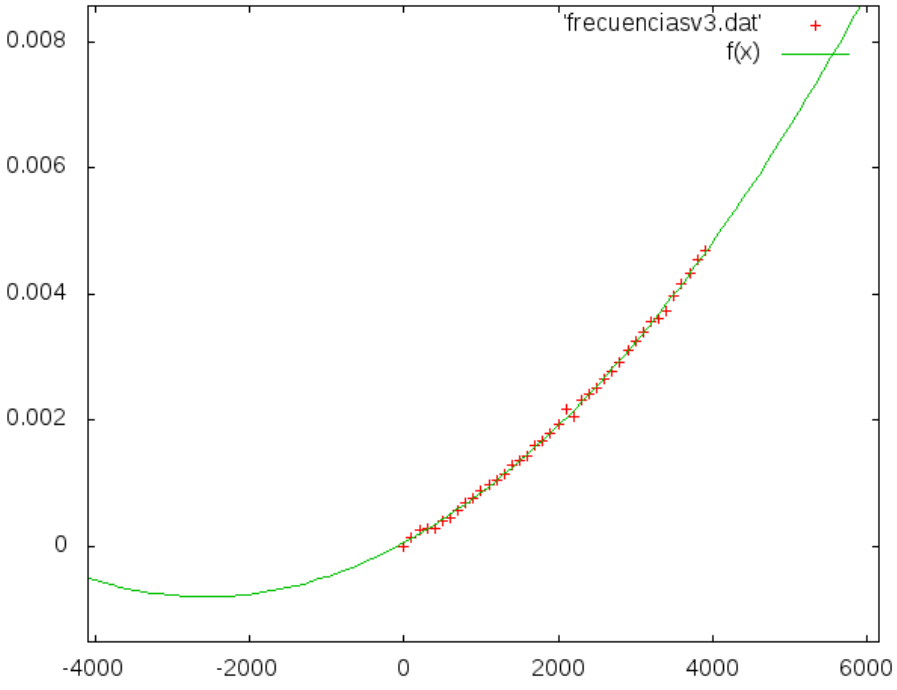
- **Versión 3**

Resultados de la ejecución:



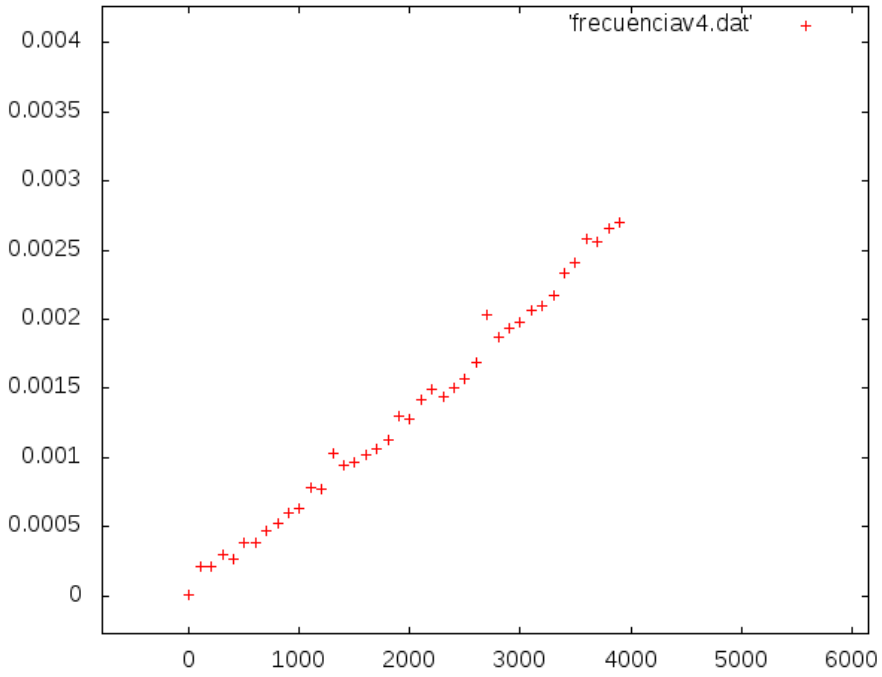
Ajustamos
nuestra eficiencia teórica de orden O(n²) a los datos obtenidos. Y concluimos que el ajuste es Y= a*X² + b*X + c

a = 1.30503e-10
b = 6.69356e-07
c = 5.52284e-05



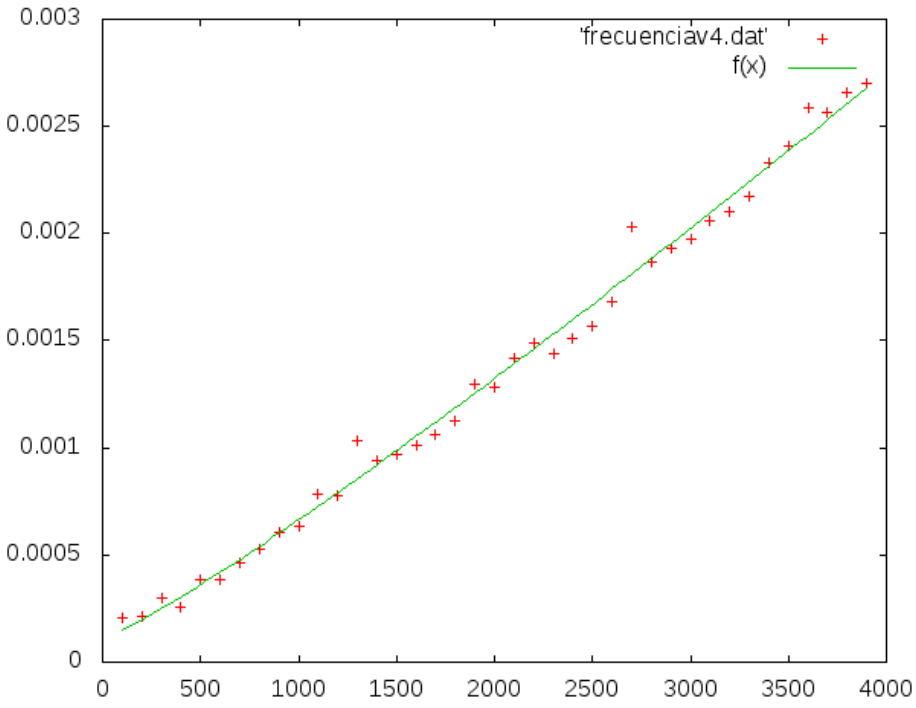
- **Versión 4**

Resultados de la ejecución:



Ajustamos nuestra eficiencia teórica de orden O(n*log(n)) a los datos obtenidos. Y concluimos que el ajuste es Y= a*X*log(X)+b

a = 7.95626e-08
b = 0.000114547



Tarea 3:

- **Algoritmo Burbuja**

Eficiencia Teórica

```
void burbuja(vector<string> & T, int inicial, int final) {
  int i, j;
  string aux;
  for (i = inicial; i < final - 1; i++)
    for (j = final - 1; j > i; j--)
        if (T[j] < T[j-1]) {
          aux = T[j];
          T[j]= T[j-1];
          T[j-1] = aux;
        }
}
```

Si tomamos n=final-inicial, teniendo en cuenta que todas las operaciones que aparecen tienen coste constante, tenemos que el orden de eficiencia viene dado por

$$1+\sum_{i=1}^{n}\sum_{j=1}^{i}1 = 1+\sum_{i=1}^{n} i = 1+n(1+n)/2 = 1+(n^2+n)/2$$

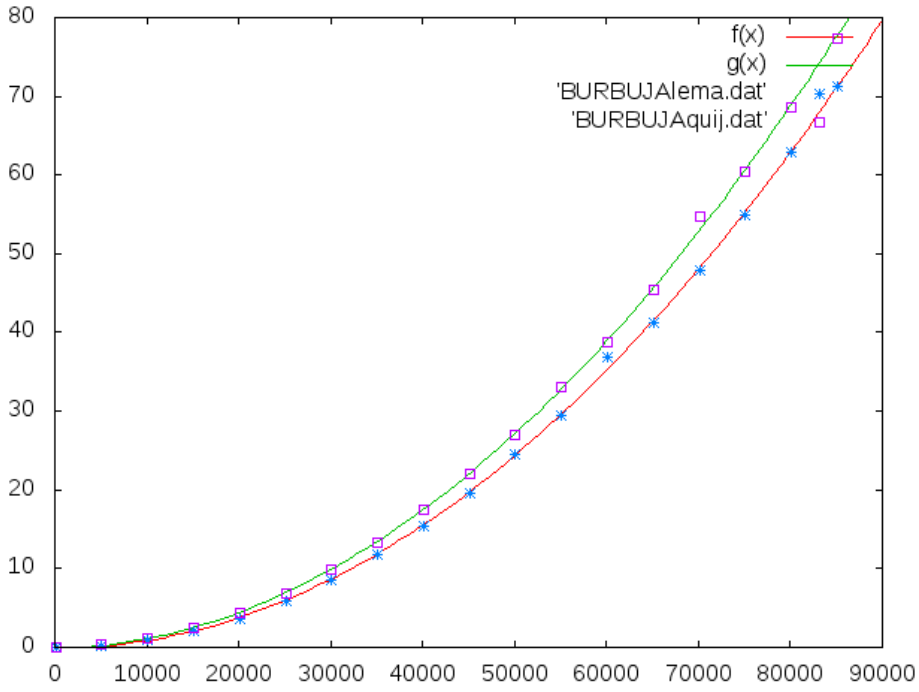
Así pues tiene orden **O(n²)**

Eficiencia Empírica

Ajuste para Y = a*x² + b*x + c

lema.txt f(x)
a = 9.94361e-09
b = -8.70272e-06
c = -0.0722014

quijote.txt g(x)
a = 1.09048e-08
b = -8.70255e-06
c = -0.0722014



- **Algoritmo Insercción**

Eficiencia Teórica

```
void insercion(vector<string> & T, int inicial, int final) {
  int i, j;
  string aux;

  for (i = inicial; i < final - 1; i++){
    aux = T[i];
    j = i-1;
    while((T[j]>aux) && (j>=0)){
          T[j+1]=T[j];
          j--;
    }
    T[j+1]=aux;
  }
}
```

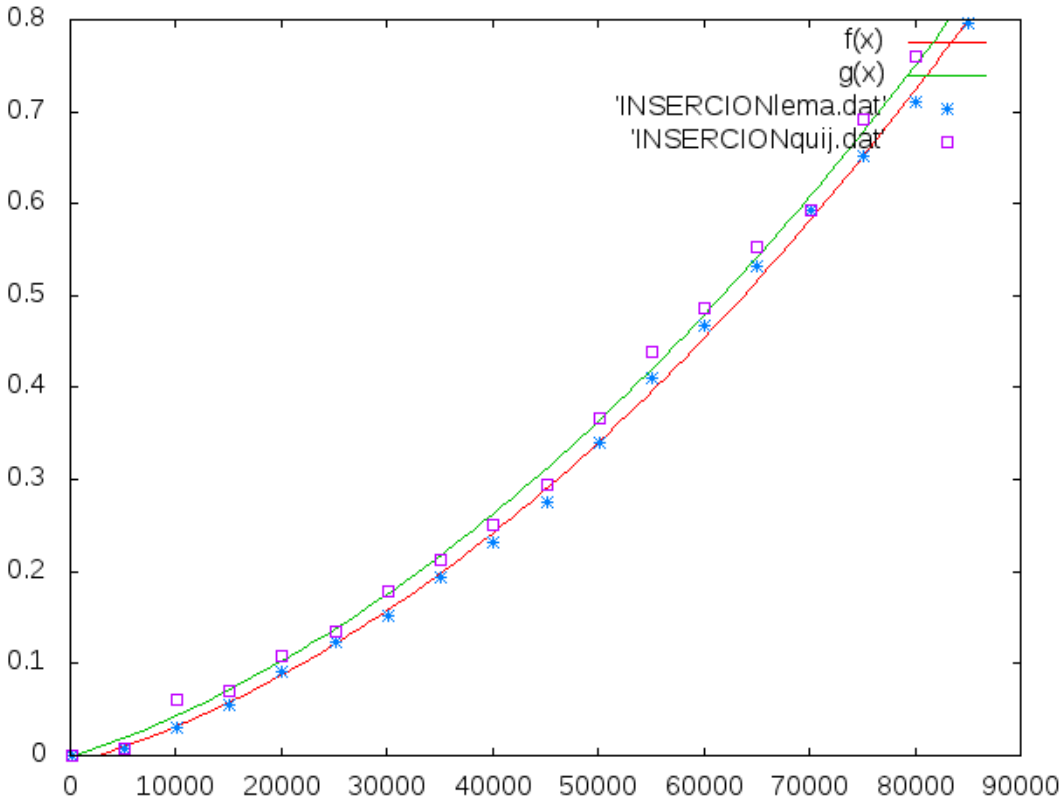
A la hora de estudiar la eficiencia sería igual que el algoritmo burbuja, por tanto también tiene orden de eficiencia **O(n^2)**

Eficiencia Empírica

Ajuste para Y = a*x^2 + b*x + c

lema.txt f(x)
a = 7.20843e-11
b = 3.39825e-06
c = -0.009477

quijote.txt g(x)
a = 6.97366e-11
b = 3.82942e-06
c = -0.00218384



- **Algoritmo Selección**

Eficiencia Teórica

```
void insercion(vector<string> & T, int inicial, int final) {
  int i, j;
  string aux;

  for (i = inicial; i < final - 1; i++){
    aux = T[i];
    j = i-1;
    while((T[j]>aux) && (j>=0)){
          T[j+1]=T[j];
          j--;
    }
    T[j+1]=aux;
  }
}
```

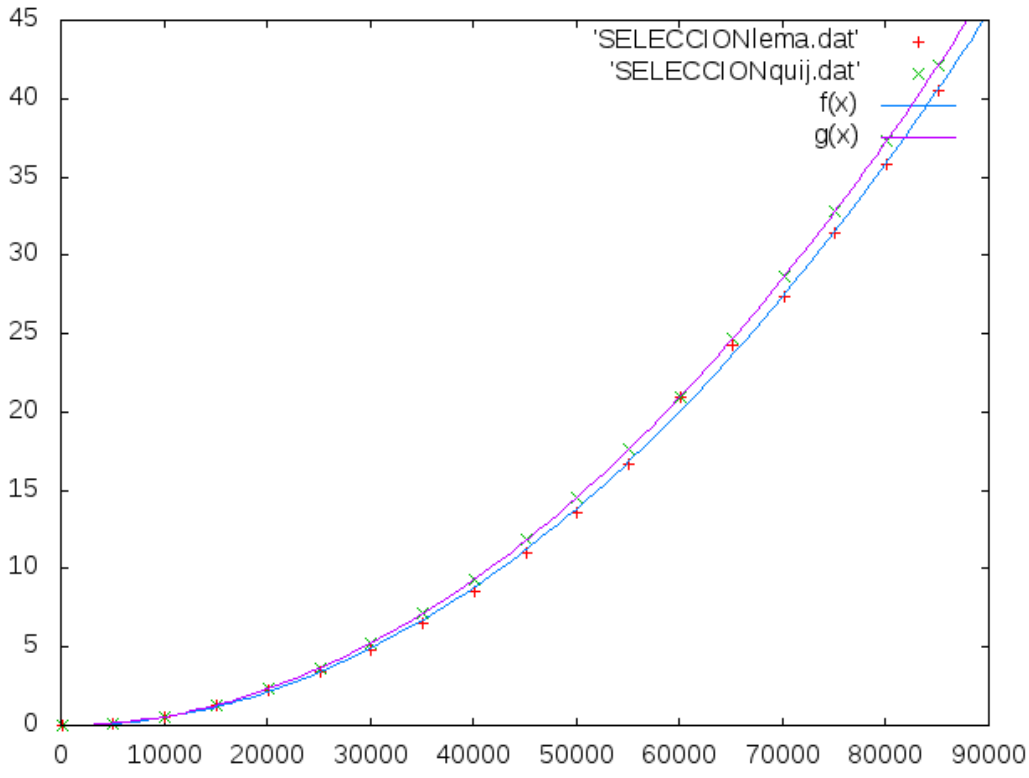
A la hora de estudiar la eficiencia sería igual que el algoritmo burbuja, por tanto también tiene orden de eficiencia **O(n^2)**

Eficiencia Empírica

Ajuste para Y = a*x^2 + b*x + c

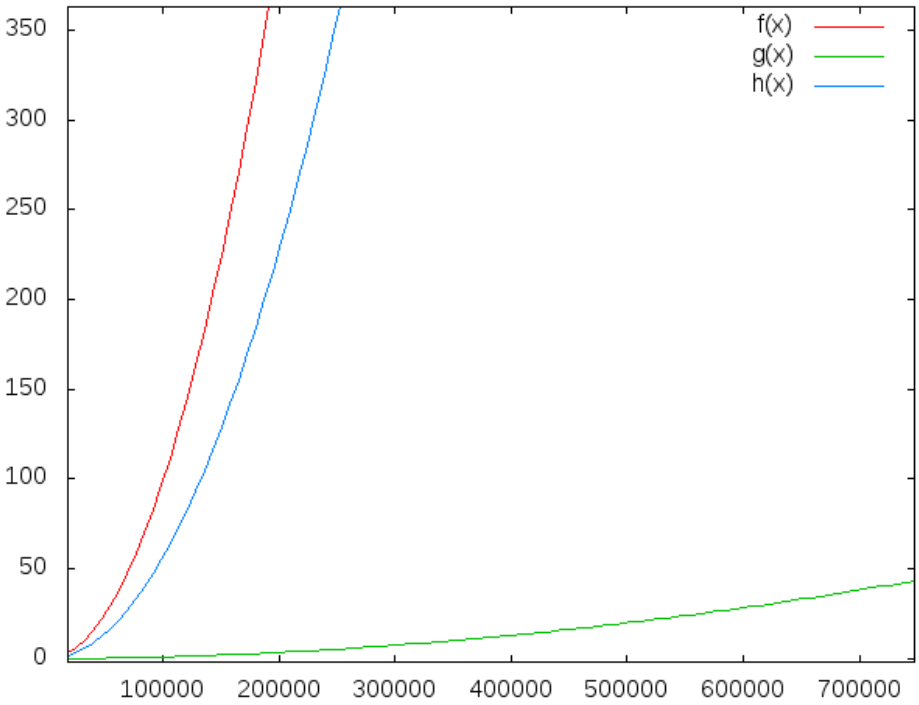
lema.txt f(x)
a = 5.7204e-09
b = -9.75185e-06
c = 0.0255561

quijote.txt g(x)
a = 5.82659e-09
b = -7.45202e-07
c = -0.00618913



Comparación de los distintos algoritmos para “lema.txt”

Burbuja → f(x)
Inserción → g(x)
Selección → h(x)



Comparación de los distintos algoritmos para “quijote.txt”

Burbuja → f(x)
Inserción → g(x)
Selección → h(x)