PRACTICA 1 DE ESTRUCTURA DE DATOS:
EFICIENCIA DE ALGORITMOS

Realizada por : *Juan Manuel López Castro*
Ignacio Pineda Mochón

Curso y grupo: *2º B // Ingeniería Informática*

EJERCICIO 1: *OCURRENCIAS*

Ana lisis de eficiencia de un programa que realiza la búsqueda de una palabra en un texto. En este caso, se realizan 20 búsquedas en un archivo de texto; en cada búsqueda, el numero de datos es mayor por lo que el tiempo de búsqueda es también mayor.

Codigo Fuente:

….

```
/* Contar ocurrencias de un string en el vector V
 V: vector sobre el que queremos buscar
 ini: primera posición desde la que buscar
 fin: posición siguiente a la ultima para buscar (desde V[0] hasta V[fin-1])
 s: palabra a buscar
 return la posición en la que se encuentra la palabra, POS_NULA en caso contrario
 */

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;
}



int main() {

        vector<string> Q;
        int pos;

        high_resolution_clock::time_point start,end;
        duration<double> tiempo_transcurrido;

        int contador =0;

        lee_fichero("quijote.txt", Q);

         ////////////////////////////////////////////////
        // CONTAR OCURRENCIAS PALABRA
         // Si los tiempos son muy grandes se itera sobre un numero elevado de iteraciones y se
         // calcula en tiempo realizado en media para una de ellas (en el ejemplo son 2000)
         ////////////////////////////////////////////////

        int max_iteraciones = 2000;
        for (int fin = 100; fin < 100000 ; fin+= 5000){
```

```
        string b="hidalgo";

        start = high_resolution_clock::now();
        for (int iteraciones = 0; iteraciones < max_iteraciones; iteraciones++)  // Numero de
        iteraciones se debe ajustar a cada caso
        pos = contar_hasta(Q, 0,fin, b);
        end= high_resolution_clock::now();

        tiempo_transcurrido  = duration_cast<duration<double> >(end - start);
        cout << fin << " " << tiempo_transcurrido.count() /( (double) max_iteraciones )<< endl;
        }
        return 0;
}
```

Para compilar el .cpp, se ha utilizado el compilador gcc y la siguiente instrucción:

```
g++ ocurrencias.cpp -o ocurrencias
```

A continuación, se ha ejecutado el programa, redirigiendo la salida a un archivo para poder utilizar posteriormente los datos.

```
./ocurrencias > ocurrencias.dat

cat ocurrencias.dat

100 1.26544e-06
5100 5.73105e-05
10100 0.000113269
15100 0.000169074
20100 0.000232352
25100 0.000279995
…...
```

En este archivo se recogen los tiempos de búsqueda (derecha) para un determinado numero de datos (izquierda).

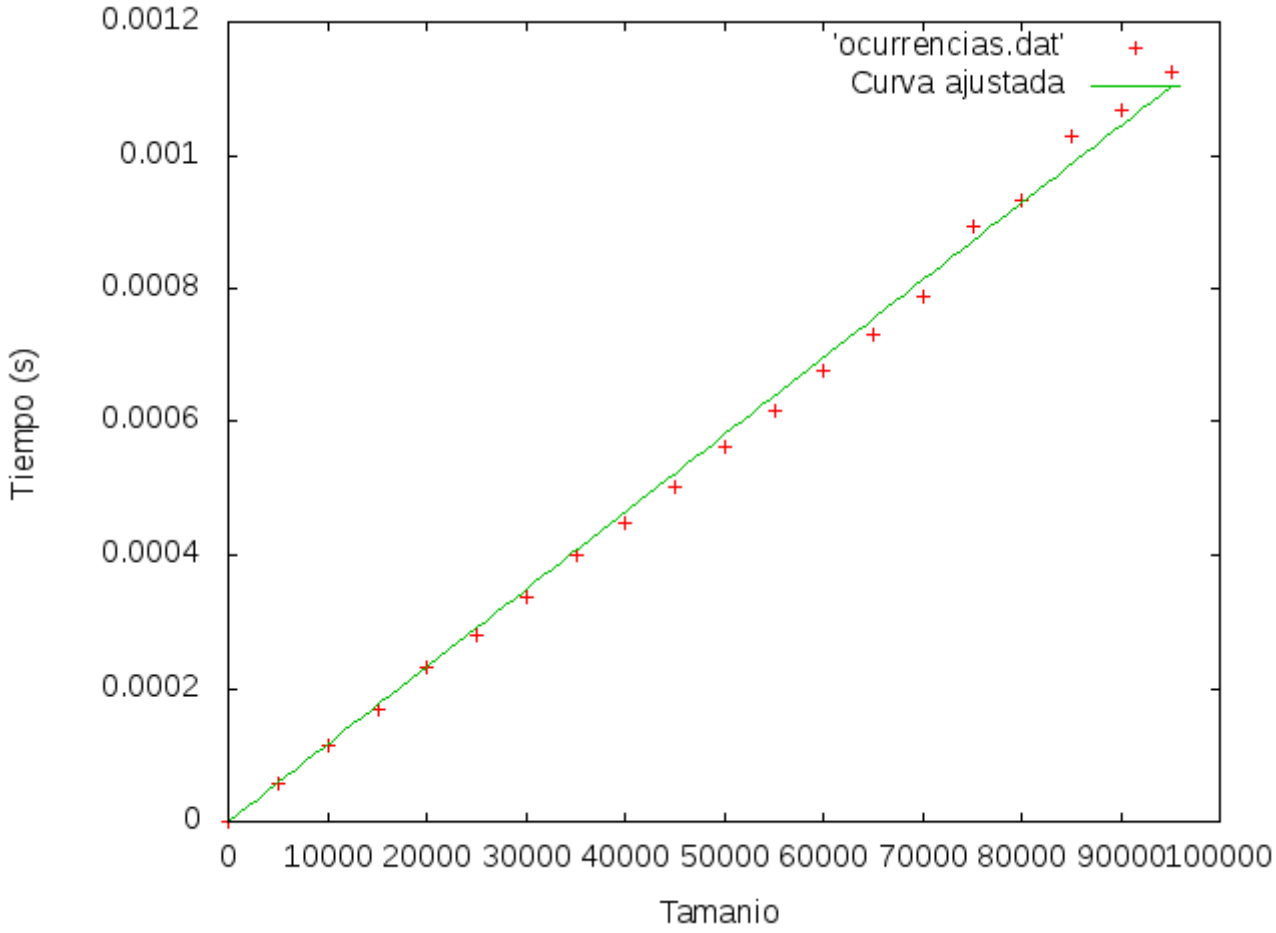
Análisis de eficiencia:

En primer lugar se ha calculado teóricamente el orden de eficiencia,O(n), del algoritmo. El algoritmo empleado para la búsqueda es el implementado en la función contar_hasta:

```
int contar_hasta( vector<string> & V, int ini, int fin, string & s) {
  int cuantos = 0;                          // O(1)
  for (int i=ini; i< fin ; i++)             // O(n)
    if (V[i]==s) {                          // O(1)
      cuantos ++;                           // O(1)
    }
  return cuantos;
}
```

Concluimos que el algoritmo tiene el orden de eficiencia O(n) (bucle for), pues las demas instrucciones tienen un orden O(1). Ahora vamos a comprobarlo experimentalmente.

Utilizando los datos del archivo ocurrencias.dat, procedemos a realizar una regresión lineal. Ejecutamos gnuplot, y realizamos el ajuste con 'f(x) = a * x' y los datos almacenados en 'ocurrencias.dat'.



Los cálculos son correctos pues, como se puede comprobar, la variación entre la curva y los datos es mínima.

$$T(n) = 1.16022e\text{-}8 * n$$

EJERCICIO 2: *FRECUENCIAS*

Análisis de 4 algoritmos diferentes que cuentan la frecuencia de aparición de cada palabra en un libro. Los 4 algoritmos recibían el mismo fichero y cada uno tardaba un tiempo diferente en hacerlo.

Algoritmo 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);
    }
}
```

La función contar_hasta es O(n). Al estar dentro de un bucle se llama n veces y el O de la función contar_frecuencia_V1 es O(n^2).

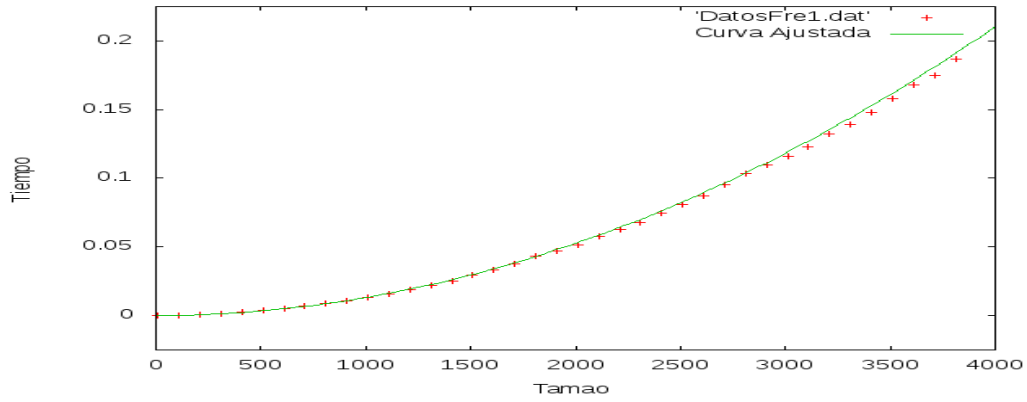
Compilamos y ejecutamos el programa guardando su tiempo en un fichero:

./frecuencias > DatosFre1.dat

El contenido de DatosFre1.dat sera:

N	Tiempo
10	1.7828e-05
110	0.000196408
210	0.00062904
310	0.00137966
410	0.00240595
510	0.00369544
610	0.00527721
710	0.00707629
810	0.00889235
910	0.0109363
1010	0.0131453
…..	….....

Haciendo el análisis híbrido y la regresión obtenemos la siguiente gráfica:



Podemos ver que los datos son correctos ya que la variación entre la curva ajustada y la curva con los datos recogidos es mínima.

$T(n)=1.31448e-08 * n^2$

Algoritmo 2:

```
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]++;
       }

    }
}
```

La función buscar es O(n) y como esta dentro de un bucle for se ejecuta n veces por lo que el O final de la función también sera $O(n^2)$.

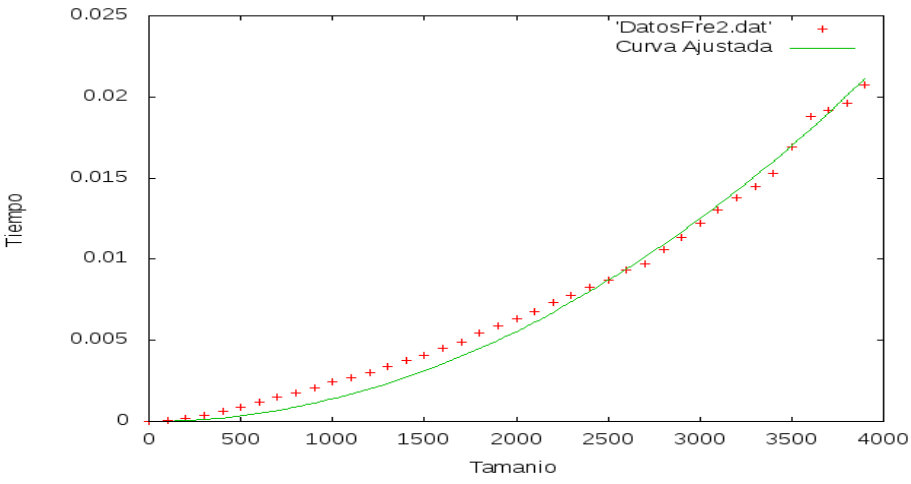
Compilamos y ejecutamos el programa guardando su tiempo en un fichero:

./frecuencias > DatosFre2.dat

El contenido de DatosFre2.dat sera:

N	Tiempo
0	9.3e-08
100	5.924e-05
200	0.00018334
300	0.000372513
400	0.000614689
500	0.000879445
600	0.00116523
700	0.00147376
800	0.00177003
900	0.00209594
1000	0.00243355
……	…….

Haciendo el análisis híbrido y la regresión obtenemos la siguiente gráfica:



Podemos ver que los datos son correctos ya que la variación entre la curva ajustada y la curva con los datos recogidos es mínima.

$T(n)=1.38899e-09 * n^2$

Algoitmo 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)
       }

    }
}
```

En este algoritmo el if() seria O(n) que es mayor que O(log(n)) por lo que nos ponemos en el peor caso y cogemos O(n). Al estar dentro de un bucle for y repetirse n veces el O final de la función es $O(n^2)$.

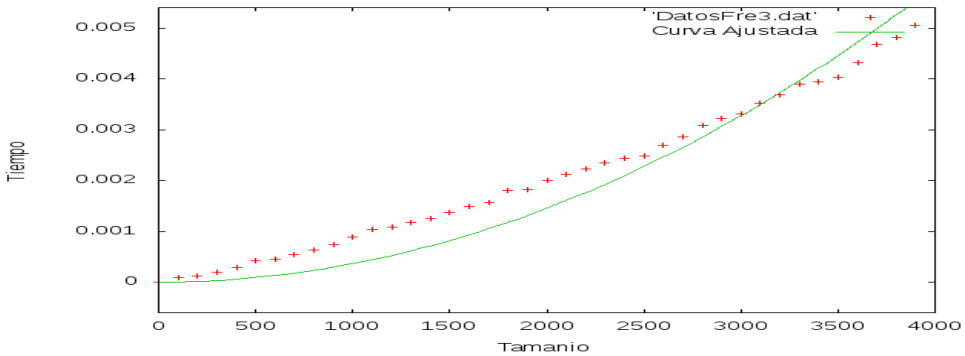
Compilamos y ejecutamos el programa guardando su tiempo en un fichero:

./frecuencias > DatosFre3.dat

El contenido de DatosFre3.dat sera:

N	Tiempo
0	1.54e-07
100	8.679e-05
200	0.00012927
300	0.00020414
400	0.000285148
500	0.000427257
600	0.000454203
700	0.000548047
800	0.000640394
900	0.000742321
1000	0.000891523
…..	……..

Haciendo el análisis híbrido y la regresión obtenemos la siguiente gráfica:



$T(n)=3.63718e\text{-}10 * n^2$

Algoritmo 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 );
     }
}
```

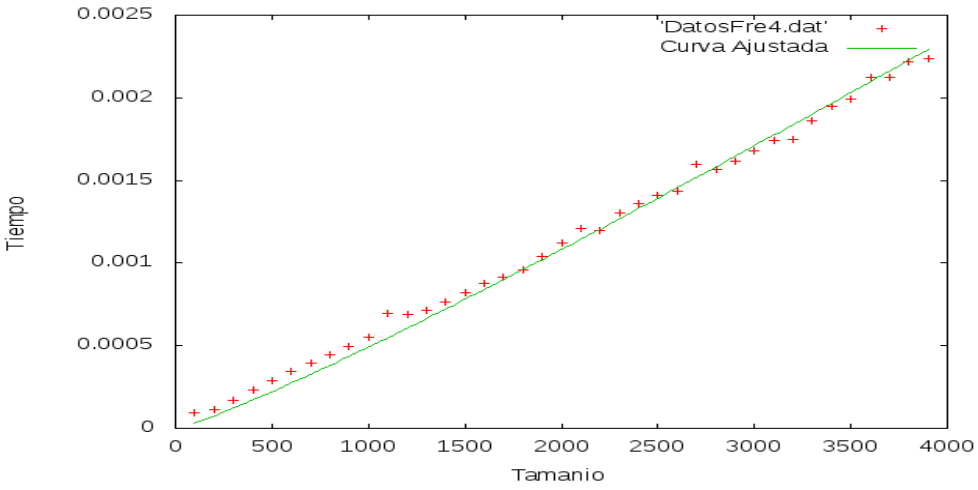
En este algoritmo la peor situación seria que se ejecutase entero el primer for , por lo que el O final de la función sera O(n*log(n)).
Compilamos y ejecutamos el programa guardando su tiempo en un fichero:

./frecuencias > DatosFre4.dat

El contenido de DatosFre4.dat sera:

N	Tiempo	
100	9.2403e-05	//De aquí quitamos el valor 0 por que no existe log(0).
200	0.000114014	
300	0.000167114	
400	0.00023335	
500	0.000285104	
600	0.000345584	
700	0.000393059	
800	0.000447311	
900	0.000497914	
1000	0.000553504	
1100	0.000695859	
…..	….....	

Haciendo el análisis híbrido y la regresión obtenemos la siguiente gráfica:



Podemos ver que los datos son correctos ya que la variación entre la curva ajustada y la curva con los datos recogidos es mínima.
T(n)=7.11867e-08 * n * log(n)

EJERCICIO 3: *BUSQUEDA 'BURBUJA'*

Análisis de eficiencia del algoritmo de ordenación “burbuja”. Para ello, se han ordenado parte de los datos de los archivos quijote.txt y lema.txt (se intentaron ordenar los archivos completos pero, tras una hora de ejecución, tan solo se habían realizado varias ordenaciones y, previniendo el tiempo total que supondría, decidimos reducir el tamaño del archivo a ordenar).

Código Fuente (y algoritmo):

…..

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

int main(int argc, char * argv[]){
          int tamanio;
          vector<char> Dicc;
          vector<char> Q;
          high_resolution_clock::time_point start,end;
          duration<double> tiempo_transcurrido;


          lee_fichero("lema.txt", Dicc);
          lee_fichero("quijote.txt", Q);
          int tama1=100;
          int tama2=100;

          for ( ; tama1<Q.size();tama1+=1000){
          start =  high_resolution_clock::now();;

          burbuja(Q,0,tama1);

          end=  high_resolution_clock::now();;


          tiempo_transcurrido  = duration_cast<duration<double> >(end-start);;
          cout << tama1 << " " << tiempo_transcurrido.count() << endl;

          }

          cout << "\n";
```

```
for ( ; tama2<Dicc.size();tama2+=100){
start =  high_resolution_clock::now();;

burbuja(Dicc,0,tama2);

end=high_resolution_clock::now();;


tiempo_transcurrido  = duration_cast<duration<double> >(end-start);;
cout << tama2 << " " << tiempo_transcurrido.count() << endl;

}


return 0;

}
```

Para compilar el .cpp, se ha utilizado el compilador gcc y la siguiente instrucción:

```
g++ BurbujaEficiencia2.cpp -o  BurbujaEficiencia2 -std=c++11
```

A continuación, se ha ejecutado el programa, redirigiendo la salida a un archivo para poder utilizar posteriormente los datos.

```
./ BurbujaEficiencia2 > Burbuja.dat
```

(Se ha divido el archivo Burbuja.dat en 2, uno para los datos de quijote.txt y otro para los de lema.txt)

```
cat Burbuja.dat

100 6.1224e-05
1100 0.00725373
2100 0.0258914
3100 0.0567718
4100 0.100762
5100 0.157322
6100 0.228337
….

cat Burbuja2.dat

100 7.9548e-05
200 0.000205783
300 0.000483595
400 0.000852844
500 0.00142227
600 0.00188635
700 0.00283693
….
```

En estos archivo se recogen los tiempos de búsqueda (derecha) para un determinado numero de datos (izquierda).

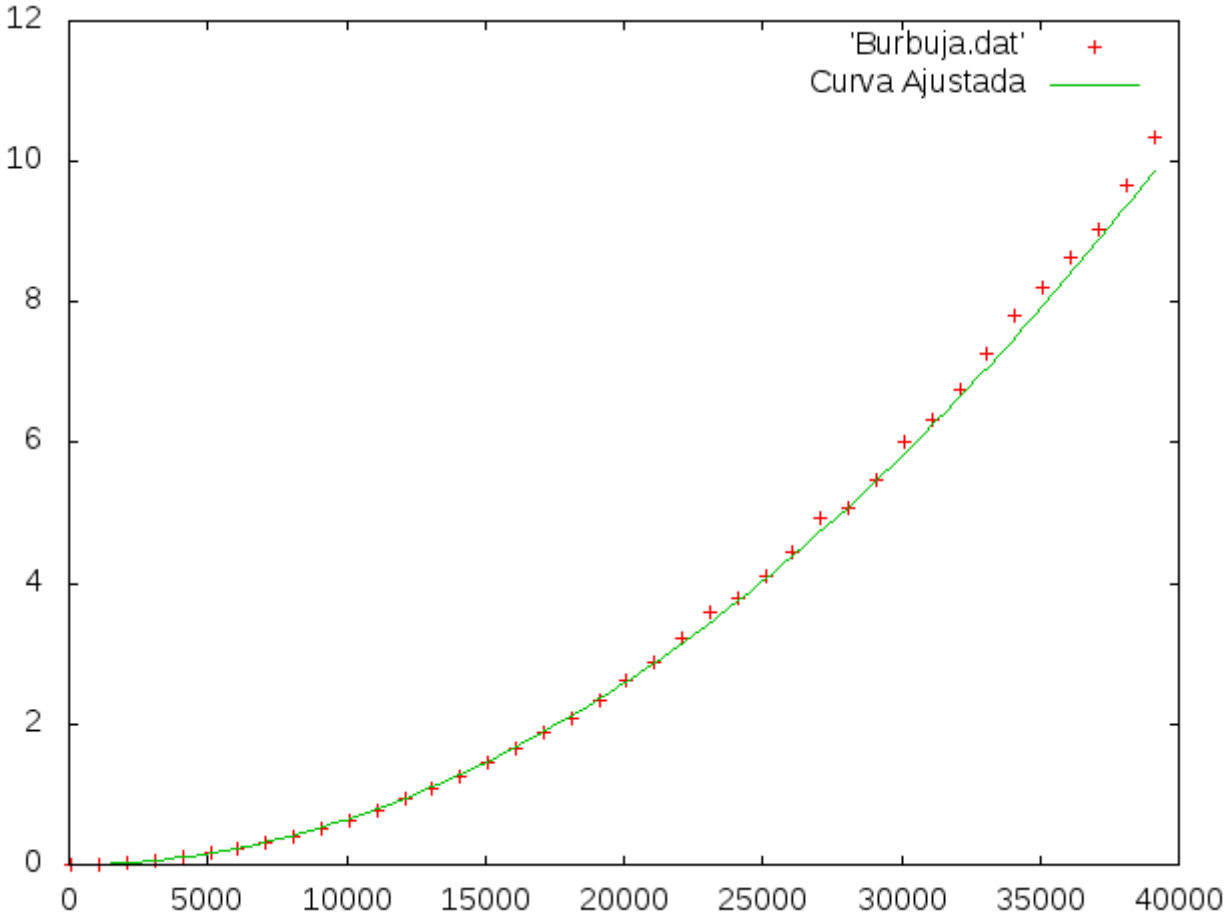
Análisis de eficiencia:

En primer lugar se ha calculado teóricamente el orden de eficiencia,O(n), del algoritmo de la burbuja:

```
void burbuja(vector<char> & T, int inicial, int final) {

        int i, j;
        char aux;
        vector<char> V=T;

        for (i = inicial; i < final - 1; i++)           // O(n)
                for (j = final - 1; j > i; j--)         // O(n)
                        if (V[j] < V[j-1]) {
                                aux = V[j];             // O(1)
                                V[j] = V[j-1];          // O(1)
                                V[j-1] = aux;           // O(1)
                        }
}
```

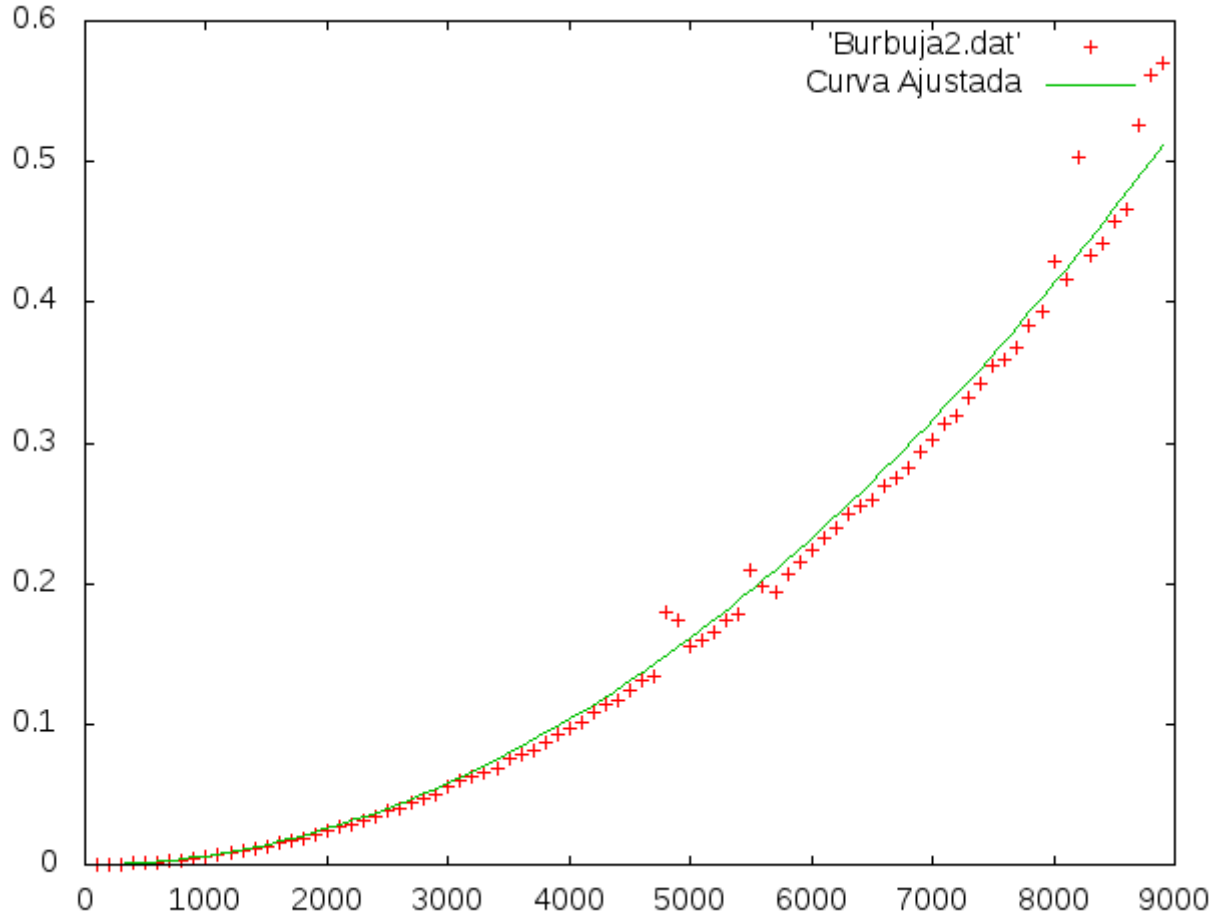
Tras realizar los cálculos con las sumatorias, comprobamos que el algoritmo tiene un orden de eficiencia $O(n^2)$. Ahora vamos a comprobarlo experimentalmente.

Utilizando los datos de los archivos Burbuja.dat y Burbuja2.dat, procedemos a realizar una regresion lineal. Ejecutamos gnuplot, y realizamos el ajuste con 'f(x) = a * x * x' y los datos almacenados en 'Burbuja.dat' y 'Burbuja2.dat' .



Los cálculos son correctos pues, como se puede comprobar, la variación entre la curva y los datos es mínima.

$$T(n) = 6.62099e\text{-}9 * n^2$$



Los cálculos son correctos pues, como se puede comprobar, la variación entre la curva y los datos es mínima.

$$T(n) = 6.45524e\text{-}9 * n^2$$

EJERCICIO 3: *SELECCION*

Análisis de eficiencia del algoritmo de selección. Para ello, se han ordenado parte de los datos de los archivos quijote.txt y lema.txt.

Código Fuente (y algoritmo):

….

```
void seleccion(vector<char> & T, int final) {

  vector<char> aux = T;
  int i, j, min, t;

  for (i = 0; i < final - 1; i++) {

    min = i;
```

```
    for (j = i + 1; j < final; j++)
      if (aux[j] < aux[min])
        min = j;

        t = aux[min];
        aux[min] = aux[i];
        aux[i] = t;

    }
}

int main(){

  vector<char> Q;

  high_resolution_clock::time_point tantes,tdespues;
  duration<double> tiempo_transcurrido;

  lee_fichero("lema.txt", Q);
  int tama = 100;

  for (; tama < Q.size(); tama +=100){
  tantes = high_resolution_clock::now();
  seleccion (Q, tama);
  tdespues = high_resolution_clock::now();

  tiempo_transcurrido  = duration_cast<duration<double> >(tdespues - tantes);
  cout << tama << " " << tiempo_transcurrido.count() << endl;

  }

  Q.clear();

  // SEGUNDA PARTE

  vector<char> D;

  lee_fichero("quijote.txt", D);
  tama = 100;

  for (; tama < D.size(); tama +=1000){
  tantes = high_resolution_clock::now();
  seleccion (D, tama);
  tdespues = high_resolution_clock::now();

  tiempo_transcurrido  = duration_cast<duration<double> >(tdespues - tantes);
  cout << tama << " " << tiempo_transcurrido.count() << endl;

  }

  D.clear();
```

```
  return 0;

}
```

Para compilar el .cpp, se ha utilizado el compilador gcc y la siguiente instrucción:

```
g++ SeleccionEficiencia.cpp -o  BurbujaEficiencia -std=c++11
```

A continuación, se ha ejecutado el programa, redirigiendo la salida a un archivo para poder utilizar posteriormente los datos.

```
./SeleccionEficiencia > Seleccion.dat
```

(Se ha divido el archivo Seleccion.dat en 2, uno para los datos de quijote.txt y otro para los de lema.txt)

```
cat Seleccion.dat

100 5.7476e-05
200 0.000152906
300 0.000308288
400 0.000429143
500 0.000689195
600 0.00109341
700 0.0013208
….

cat Seleccion2.dat

100 5.5945e-05
1100 0.00342208
2100 0.0125646
3100 0.0264901
4100 0.0453706
5100 0.0703916
6100 0.100568
….
```

En estos archivo se recogen los tiempos de búsqueda (derecha) para un determinado numero de datos (izquierda).

Analisis de eficiencia:

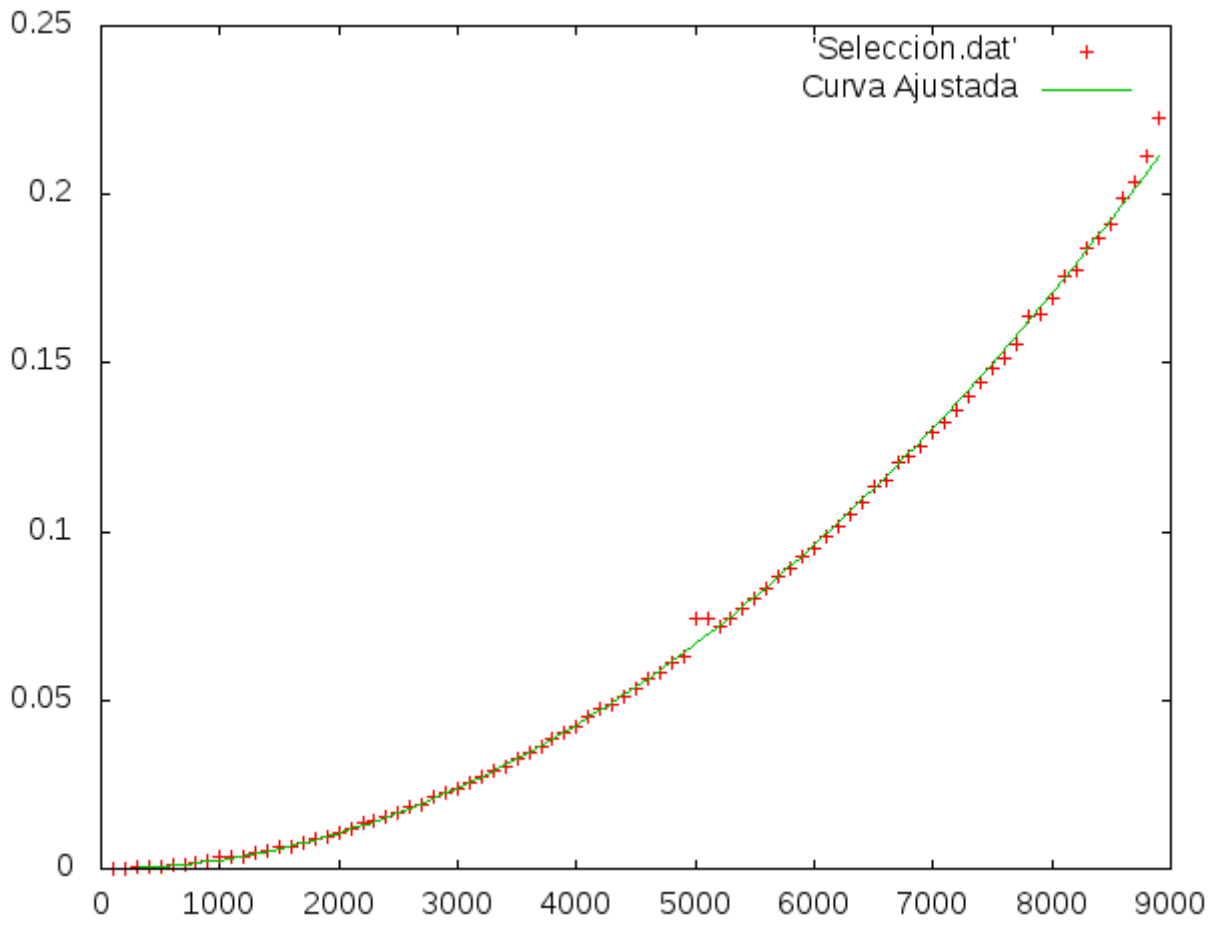
En primer lugar se ha calculado teóricamente el orden de eficiencia, O(n), del algoritmo de ordenación por selección:

```
void seleccion(vector<char> & T, int final) {

    vector<char> aux = T;
    int i, j, min, t;
```

```
        for (i = 0; i < final - 1; i++) {           // O(n)

        min = i;

        for (j = i + 1; j < final; j++)             // O(n)
                if (aux[j] < aux[min])
                        min = j;

                t = aux[min];
                aux[min] = aux[i];
                aux[i] = t;

        }
}
```

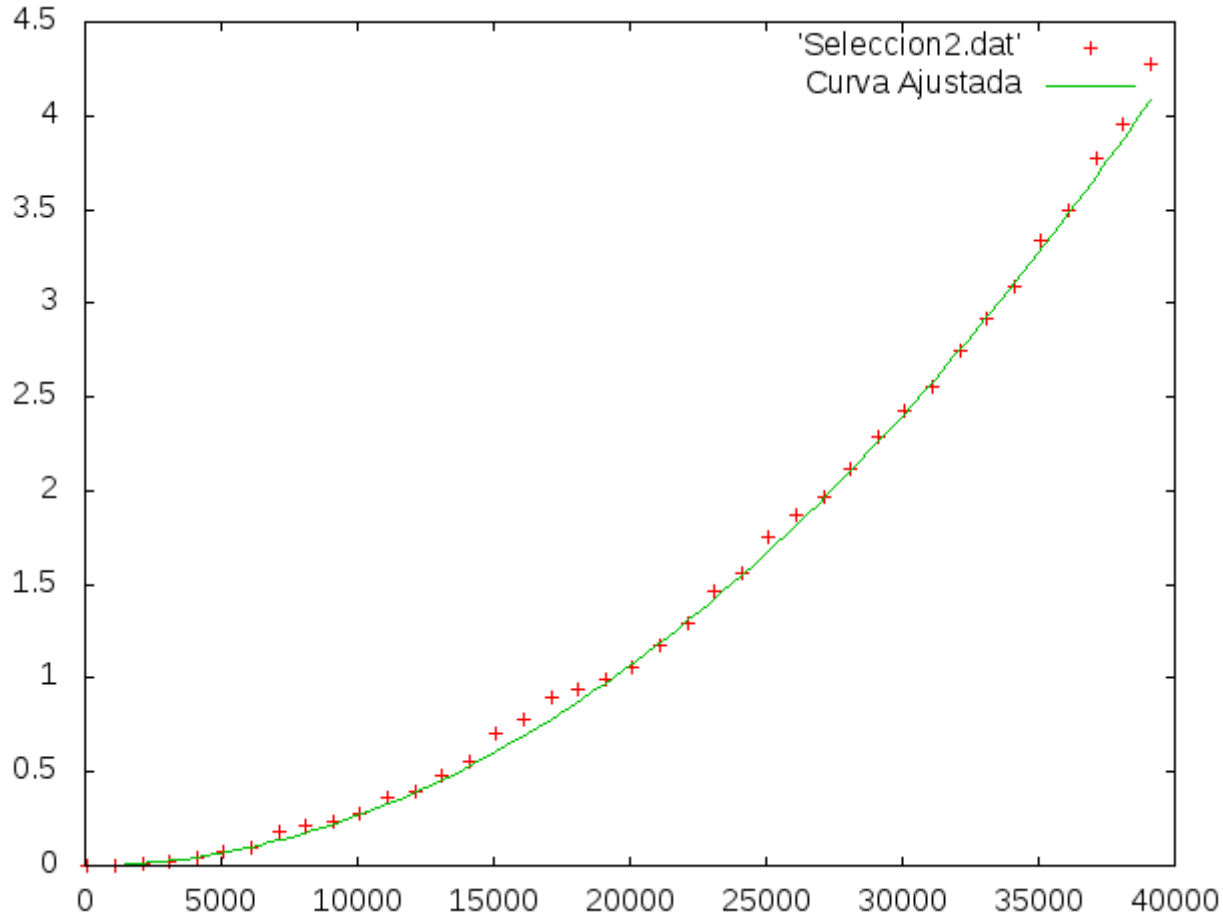
Tras realizar los cálculos con las sumatorias, comprobamos que el algoritmo tiene un orden de eficiencia $O(n^2)$. Ahora vamos a comprobarlo experimentalmente.

Utilizando los datos de los archivos Seleccion.dat y Seleccion2.dat, procedemos a realizar una regresión lineal. Ejecutamos gnuplot, y realizamos el ajuste con 'f(x) = a * x * x' y los datos almacenados en 'Seleccion .dat' y 'SeleccionSeleccion 2.dat' .



Los cálculos son correctos pues, como se puede comprobar, la variación entre la curva y los datos es mínima.

$$T(n) = 2.66856e\text{-}9 * n^2$$



Los cálculos son correctos pues, como se puede comprobar, la variación entre la curva y los datos es mínima.

$$T(n) = 2.71152e\text{-}9 * n^2$$

EJERCICIO 3: *BUSQUEDA POR INSERCCION*

Algoritmo de inserción(Con quijote.txt como entrada)

```
void insercion(vector<char> &T, int n) {
        vector<char>aux=T;

        int i, j;
        char valor;
        for (i = 1; i < n; i++) {
                valor = aux[i];
                j = i;
        while ((j > 0) && (aux[j-1] > valor)) {
```

```
            aux[j] = aux[j-1];
            j--;
            }
        aux[j] = valor;
        }
}
```

a)Análisis Teórico:
Realizando la sumatoria obtenemos:

n/2* (n-1)-(n-1) = ((n-2) * (n-1))/2 ----> $O(n^2)$

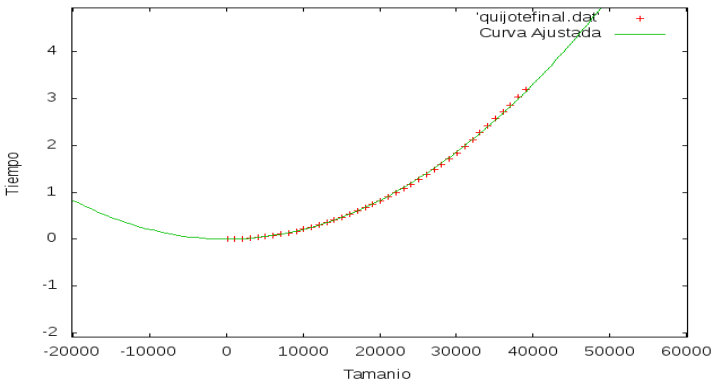
b)Análisis Práctico:
Compilamos y ejecutamos el programa guardando su tiempo en un fichero:

./insercion > quijotefinal.dat

El contenido de quijotefinal.dat sera:

N	Tiempo
100	3.5673e-05
1100	0.00397555
2100	0.0124159
3100	0.0248723
4100	0.0392372
5100	0.0537645
6100	0.0761608
7100	0.103211
8100	0.134327
9100	0.170061
10100	0.211111
…….	…….....

Haciendo el análisis híbrido y la regresión obtenemos la siguiente gráfica:



Podemos ver que los datos son correctos ya que la variación entre la curva ajustada y la curva con los datos recogidos es mínima
T(n)= 2.06361e-09 * n^2

Algoritmo de Inserción (Con lema.txt como entrada)

```
void insercion(vector<char> &T, int n) {
    vector<char>aux=T;

    int i, j;
    char valor;
    for (i = 1; i < n; i++) {
        valor = aux[i];
        j = i;
```

```
        while ((j > 0) && (aux[j-1] > valor)) {
                aux[j] = aux[j-1];
                j--;
                 }
        aux[j] = valor;
        }
}
```

a)Análisis Teórico:
Realizando la sumatoria obtenemos:

$n/2* (n-1)-(n-1) = ((n-2) * (n-1))/2$ ----> $O(n^2)$

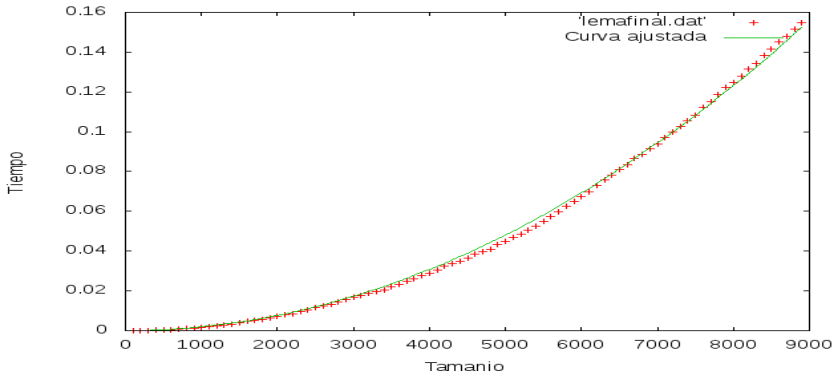
b)Análisis Práctico:
Compilamos y ejecutamos el programa guardando su tiempo en un fichero:

./insercion > lemafinal.dat

El contenido de lemafinal.dat sera:

N	Tiempo
100	3.2762e-05
200	8.1194e-05
300	0.000156217
400	0.000261559
500	0.000427033
600	0.0005962
700	0.000807443
800	0.00107703
900	0.001373
1000	0.00173299
1100	0.00198182
…..	….....

Haciendo el análisis híbrido y la regresión obtenemos la siguiente gráfica:



Podemos ver que los datos son correctos ya que la variación entre la curva ajustada y la curva con los datos recogidos es mínima.
$T(n)= 1.92508e-09 * n^2$