

Alumnas:

- **Alba Casillas Rodríguez.**

Componentes del ordenador:

- SO: Ubuntu 18.0.4
- CPU: IntelCore i7-7500U
- Velocidad del reloj 2.90GHz
- Memoria RAM 12.0 GB

- **Paula Cumbreiras Torrente .**

Componentes del ordenador:

- SO: Ubuntu 18.0.4.1 LTS
- CPU: IntelCore i5-7300HQ
- Velocidad del reloj 2.50GHz
- Memoria RAM 7.7 GiB

Grupo A1.

PRÁCTICA 1 ESTRUCTURAS DE DATOS

Ejercicio 1: Calcule la eficiencia teórica de este algoritmo.

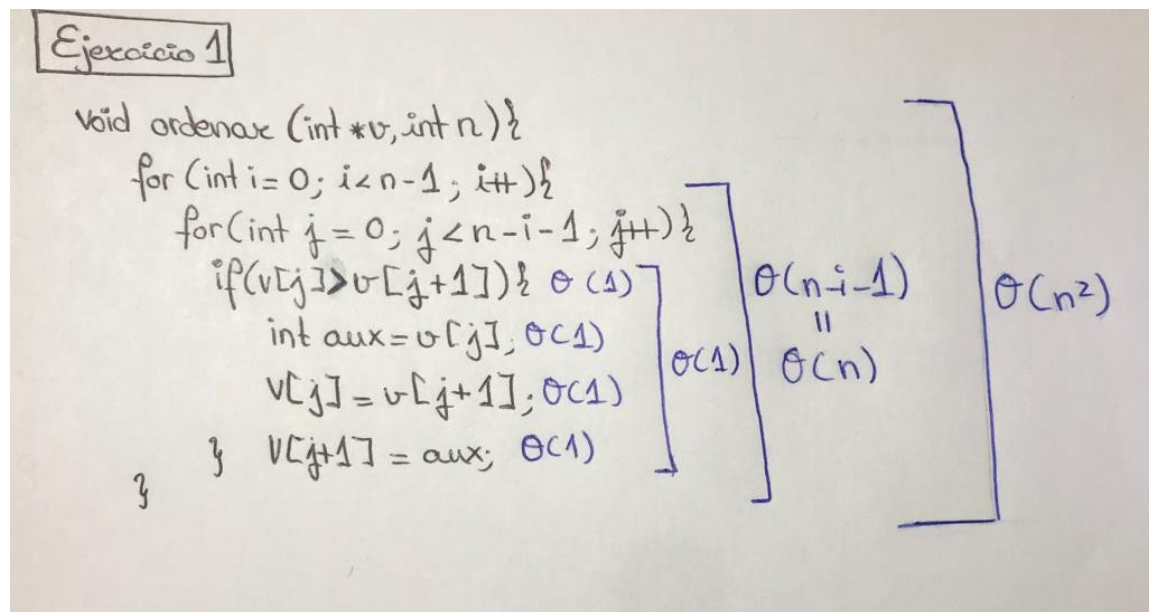
A continuación replique el experimento que se ha hecho antes (búsqueda lineal) con este nuevo código. Debe:

- 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?

Calculamos la eficiencia teórica del algoritmo, llegando a la conclusión de que tiene un Orden de n al cuadrado ($O(n^2)$).



Realizamos y compilamos este algoritmo en un programa llamado “ordenacion.cpp” para calcular la eficiencia empírica.

```
#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; // Buscamos un valor que no esté en el vector
        ordenar(v,tam); // de esta forma forzamos el peor caso

        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
    }
}
```

Generamos un script al que llamaremos “ejecucion_ordenacion.csh” donde calculamos los tiempos de ejecución para tamaños del vector 100, 600, 1100,...,30000; es decir, aumentamos el tamaño en 500.

```
#!/bin/csh
@ inicio = 100
@ fin = 30000
@ incremento = 500
@ i = $inicio
echo > tiempos_ejecucion.dat
while ( $i <= $fin )
echo Ejecucion tam = $i
echo `./ordenacion $i 10000` >> tiempos_ejecucion.dat
@ i += $incremento
end
```

Para su compilación, ha sido necesario proporcionar los permisos necesarios ejecutando en la terminal la orden:

chmod a+x ejecucion_ordenacion.csh

Los resultados se guardan en un fichero llamado “tiempo_ordenacion.dat”.

Secuencia escrita en la terminal:

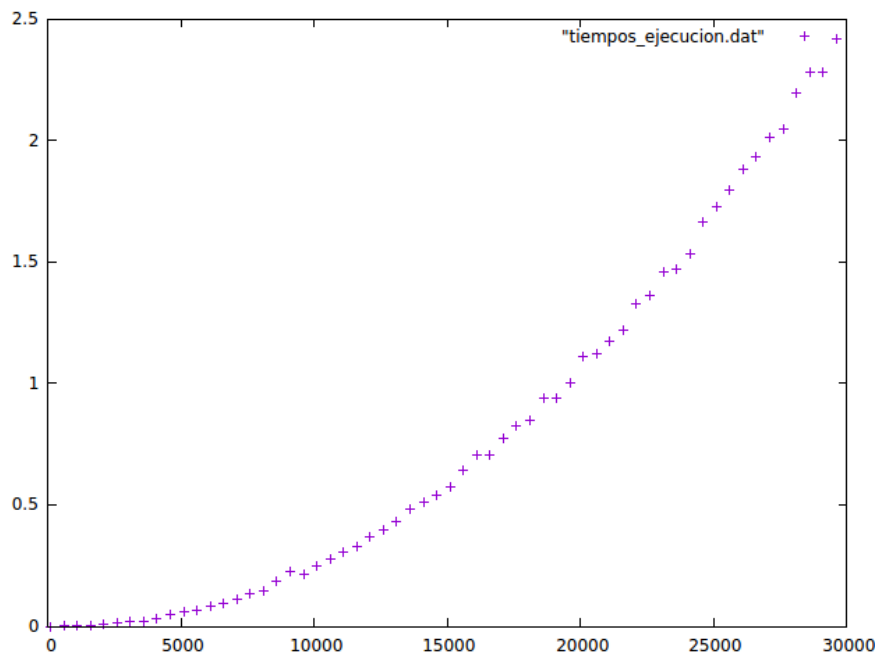
g++ ordenacion.cpp -o ordenacion

./ejecucion_ordenacion.csh

Lanzamos gnuplot desde la terminal e introducimos la siguiente secuencia:

plot “tiempos_ejecucion.dat”

Generando la siguiente gráfica:



Ejercicio 2: 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 .

Dentro de gnuplot definimos una función ($f(x)$) de la forma ax^2+bx+c , la más ideal para un algoritmo de $O(n^2)$.

Secuencia en la terminal:

$f(x) = ax^2+bx+c$
fit $f(x)$ "tiempos_ejecucion.dat" via a,b,c

Para comparar la eficiencia empírica con la teórica de forma en que ambas se distingan en la gráfica, introducimos la secuencia:

Plot $f(x)$, "tiempo_ordenacion.dat" with lines lw 4

Y con ello ajustamos $f(x)$ a los datos obtenidos en la eficiencia empírica.

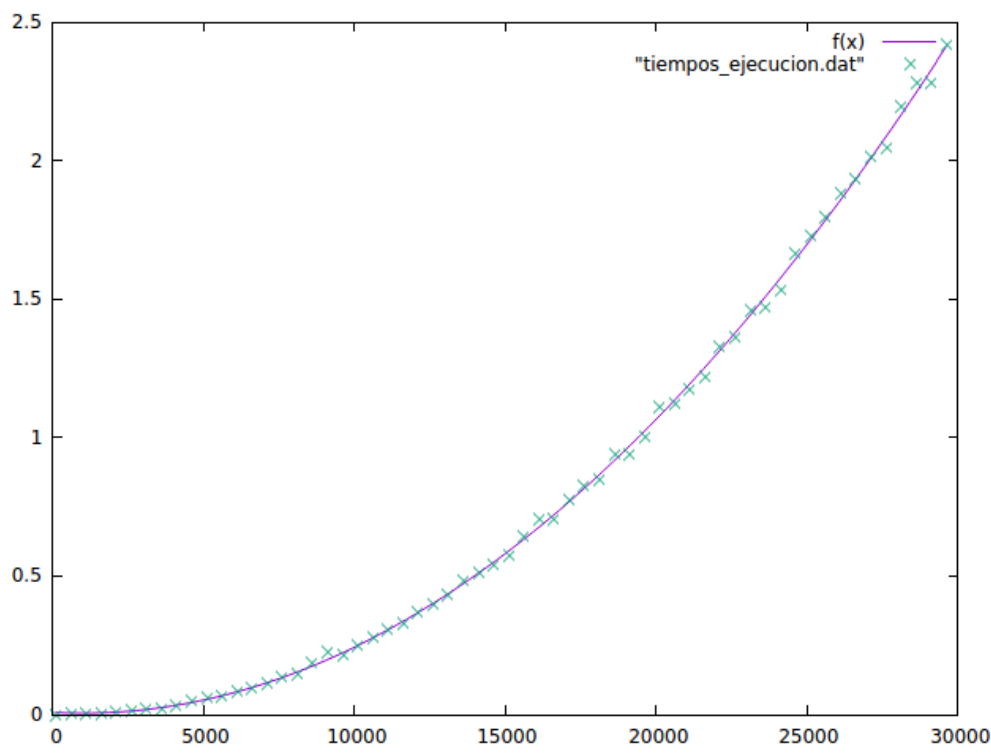
```
gnuplot> fit f(x) "tiempos_ejecucion.dat" via a,b,c
iter   chisq      delta/lin  lambda   a              b              c
0  9.4779953826e+18   0.00e+00  2.29e+08  1.000000e+00  1.000000e+00  1.000000e+00
1  2.8930837523e+14  -3.28e+09  2.29e+07  5.483216e-03  9.999583e-01  1.000000e+00
2  1.1088417473e+09  -2.61e+10  2.29e+06  -4.157035e-05  9.999560e-01  1.000000e+00
3  1.1074828622e+09  -1.23e+02  2.29e+05  -4.186847e-05  9.997456e-01  1.000000e+00
4  1.0623121796e+09  -4.25e+03  2.29e+04  -4.100561e-05  9.791423e-01  9.999972e-01
5  1.1026315814e+08  -8.63e+05  2.29e+03  -1.320639e-05  3.153591e-01  9.999082e-01
6  2.4743095398e+03  -4.46e+09  2.29e+02  -5.585359e-08  1.353642e-03  9.998648e-01
7  6.8356588601e+00  -3.61e+07  2.29e+01  6.647803e-09  -1.387460e-04  9.997341e-01
8  6.6596338599e+00  -2.64e+03  2.29e+00  6.602551e-09  -1.370898e-04  9.868475e-01
9  1.2564000834e+00  -4.30e+05  2.29e-01  4.520988e-09  -6.253847e-05  4.305954e-01
10 1.9177114551e-02  -6.45e+06  2.29e-02  2.951979e-09  -6.344283e-06  1.131228e-02
11 1.9106816365e-02  -3.68e+02  2.29e-03  2.940063e-09  -5.917527e-06  8.128113e-03
12 1.9106816364e-02  -2.12e-06  2.29e-04  2.940062e-09  -5.917494e-06  8.127872e-03
iter   chisq      delta/lin  lambda   a              b              c

After 12 iterations the fit converged.
final sum of squares of residuals : 0.0191068
rel. change during last iteration : -2.12191e-11

degrees of freedom      (FIT_NDF)                : 57
rms of residuals        (FIT_STDFIT) = sqrt(WSSR/ndf) : 0.0183087
variance of residuals   (reduced chisquare) = WSSR/ndf : 0.000335207

Final set of parameters          Asymptotic Standard Error
=====
a = 2.94006e-09                 +/- 3.526e-11    (1.199%)
b = -5.91749e-06                +/- 1.082e-06    (18.29%)
c = 0.00812787                  +/- 0.006954     (85.55%)

correlation matrix of the fit parameters:
a      b      c
a      1.000
b      -0.968  1.000
c      0.738 -0.861  1.000
```



Ejercicio 3: Junto con este guión se le ha suministrado un fichero `ejercicio_desc.cpp`. En él se ha implementado un algoritmo. Se pide que:

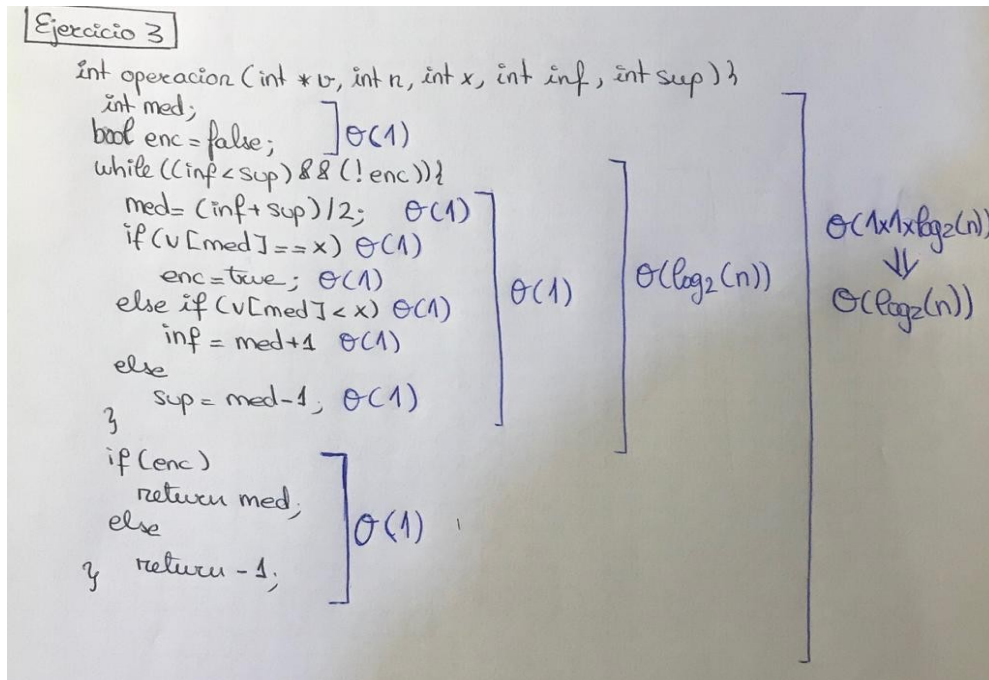
- Explique qué hace este algoritmo.
- Calcule su eficiencia teórica.
- Calcule su eficiencia empírica.

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 regresión para ajustar la curva teórica a la empírica.

El algoritmo implementado en `"ejercicio_desc.cpp"` busca un valor que se desea encontrar en un vector ordenado. Mientras que el número inferior sea menor que el superior, el vector será partido por la mitad y quedándose con la parte en la que se pueda encontrar el valor a buscar ($\text{inf} = \text{mitad} + 1$ o $\text{sup} = \text{mitad} - 1$); hasta encontrar el valor (en el caso en el que no se encuentre, se ejecutará `"return -1"`).

Para calcular la eficiencia teórica, nos situamos en el peor de los casos, en el que no se encuentra el valor y el bucle debe hacer n iteraciones.

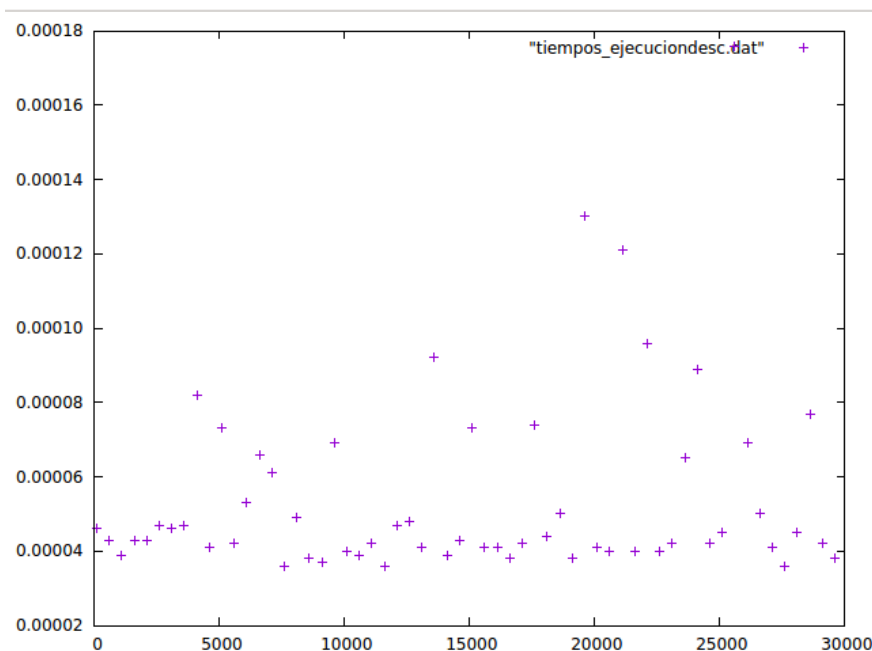
Como dentro del bucle `while` se reduce a la mitad el número de iteraciones hasta que los valores de `"inf"` y `"sup"` se crucen, ya que no se encontrará el valor buscado ($n, n/2, n/4, n/6 \dots 1$); el orden del algoritmo sería: $O(\log_2(n))$.



Creamos un script llamado "tiempos_ejecucionesdesc.csh" con el objetivo de calcular la eficiencia empírica del algoritmo.

Encontramos un hecho peculiar al dibujar la gráfica con gnuplot
plot "tiempos_ejecucionesdesc.dat"

Ya que obtenemos una gráfica prácticamente constante.



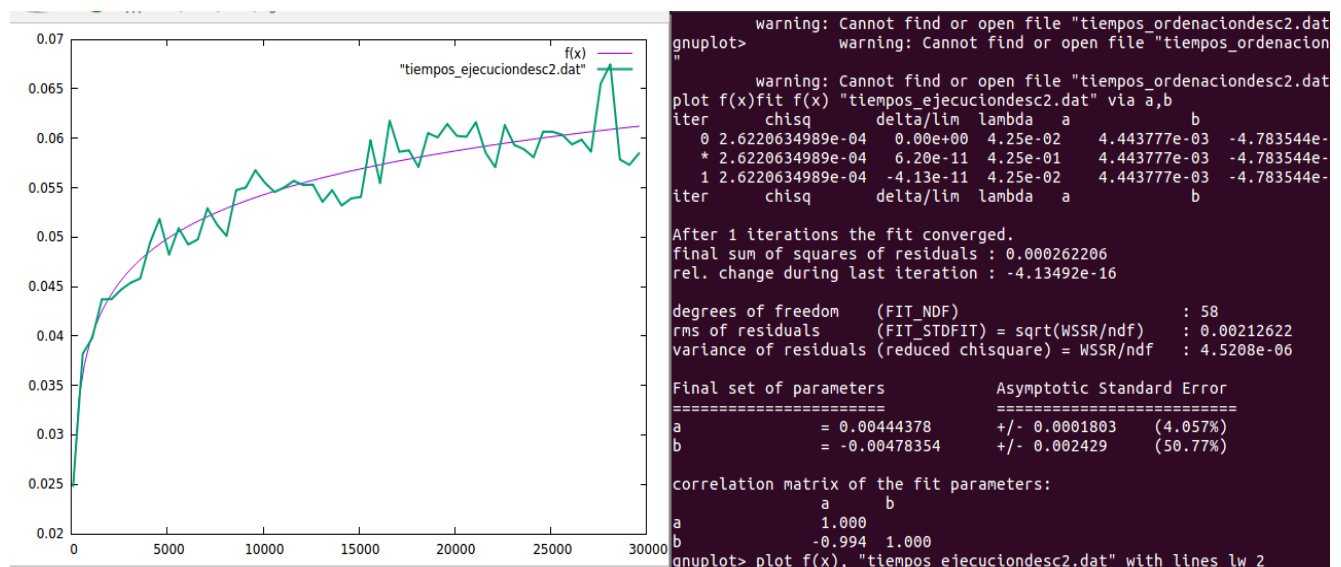
Es una situación anormal ya que nuestro algoritmo (al calcular la eficiencia teórica) obtenemos que es de orden logarítmico.

La razón es porque el algoritmo se ejecuta de forma tan rápida que no se llega a apreciar que sea logarítmica.

Como solución, colocamos un bucle for en el archivo cpp que ralentiza el tiempo de ejecución cuando sucede la llamada de la función.

```
for(int i= 0; i < 1000000; i++){
// Algoritmo a evaluar
operacion(v,tam,tam+1,0,tam-1);
}
```

Dentro de gnuplot, comparamos ahora la eficiencia teórica con la empírica, la cual ya adopta una curva logarítmica.



Ejercicio 4: 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.

En el mejor caso posible, suponemos que el vector está ya ordenado, por lo cual solamente recorreremos el vector y realizamos la asignación:

```
int *v=new int[tam]; // Reserva de memoria
for (int i = 0; i < tam; i++){ //Solamente recorremos el vector ya que
    v[i] = i; //ya está ordenado
}
```

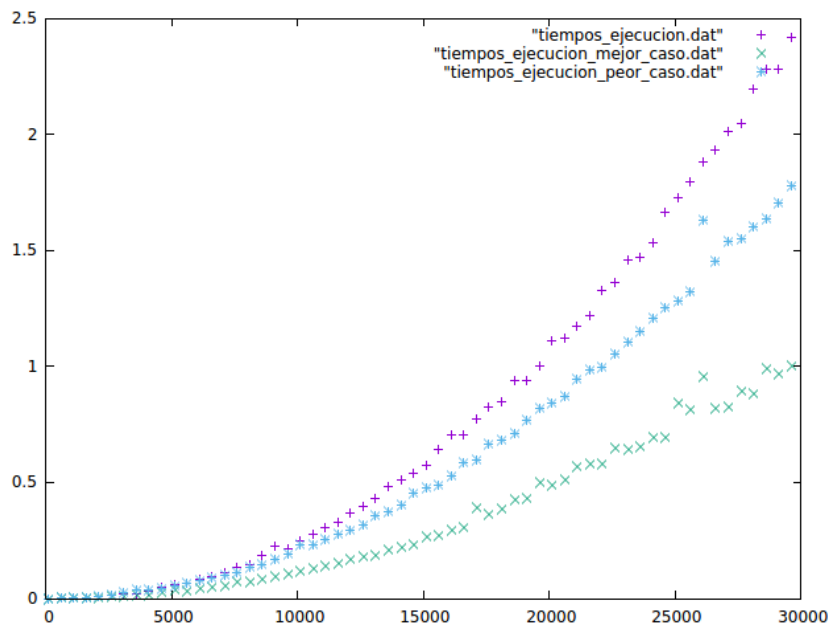

- El peor caso posible. Si la entrada es un vector ordenado en orden inverso estaremos en la peor situación posible ya que en cada iteración del bucle interno hay que hacer un intercambio.
Calcule la eficiencia empírica en ambos escenarios y compárela con el resultado del ejercicio 1.

Al obtener un vector en orden inverso, recorremos el vector a la inversa para así añadir los elementos al vector desde 0 a “tam – 1”

```
int *v=new int[tam];           // Reserva de memoria
for(int i=tam-1; i>=0; i--){
    v[tam-i-1] = i; //Añade elementos al vector desde 0 a tam-1
```

En la terminal, dentro de gnuplot, escribimos la sentencia:
plot “tiempos_ejecucion.dat”, tiempos_ejecucion_mejor_caso.dat”, tiempos_ejecucion_peor_caso.dat”

De esta manera podemos comparar gráficamente la eficiencia empírica del mejor y peor caso junto a la eficiencia empírica calculada en el Ejercicio 1.



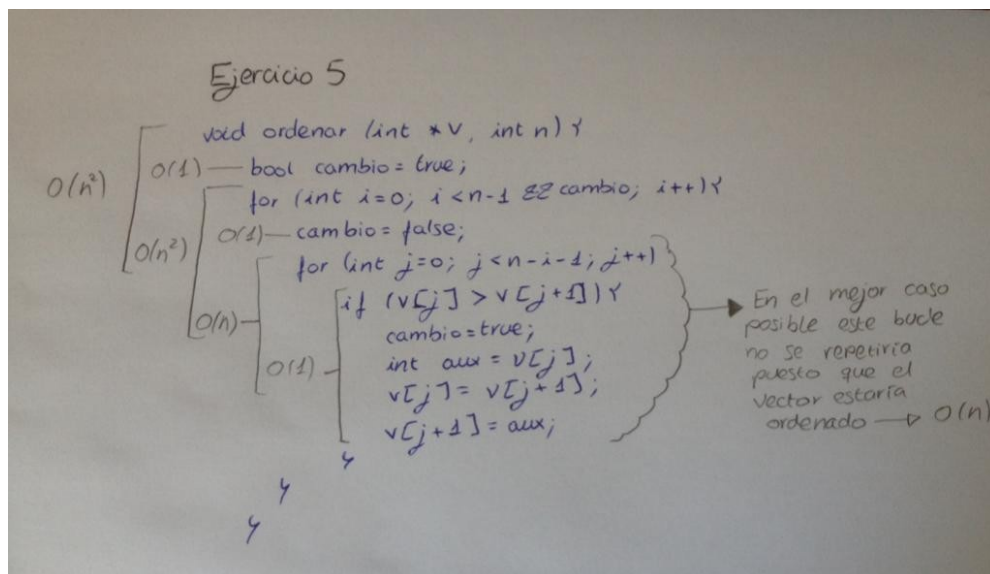
Ejercicio 5: Dependencia de la implementación

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.

Aunque este algoritmo de ordenación tiene un orden de eficiencia de n^2 , si consideramos que el vector de entrada ya se encuentra ordenado, solo se produciría una iteración: el orden de eficiencia teórica sería de n .

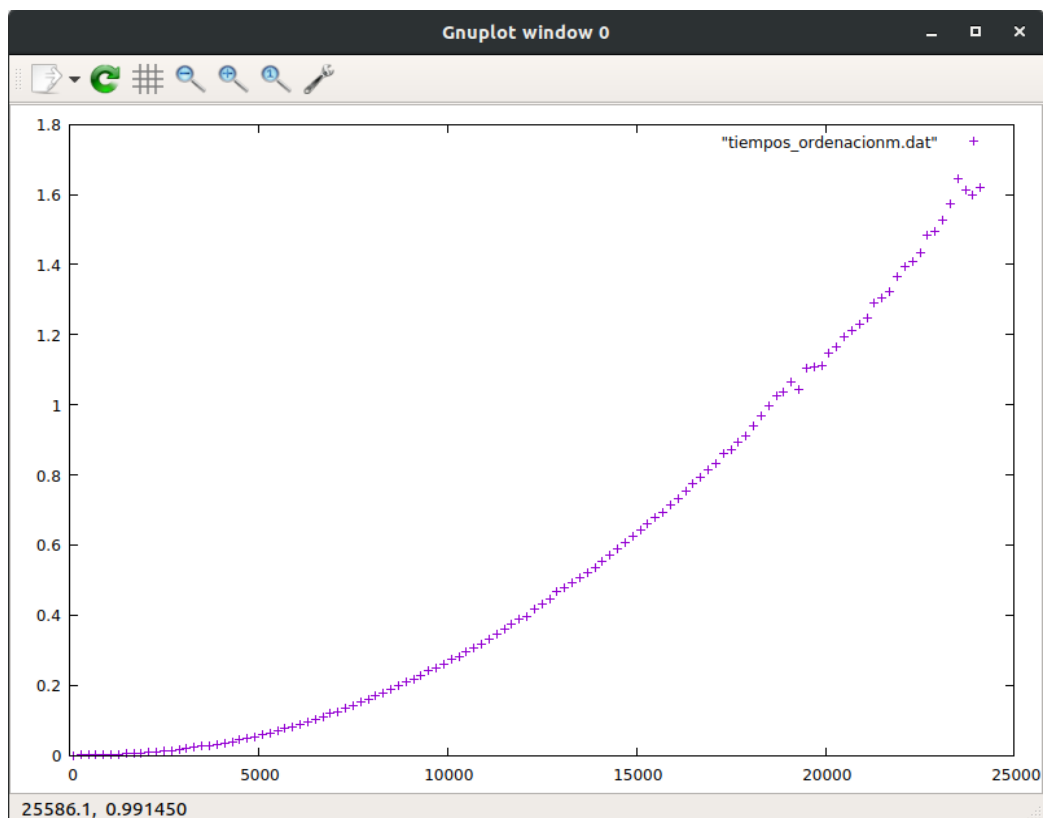


Para comprobar la eficiencia empírica, después de haber editado el código del programa “ordenacion.cpp” y de haberlo compilado, debemos crear un archivo tipo csh que contenga lo siguiente:

```
#!/bin/csh
@ inicio = 100
@ fin = 100000
@ incremento = 500
set ejecutable = ordenacion_mejora
set salida = tiempos_ordenacion_mejora.dat

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

Una vez hecho, le damos derechos de ejecución al archivo csh y lo ejecutamos. Los tiempos se guardaran en un archivo llamado tiempos_ordenacion.dat, archivo que reutilizaremos para generar una gráfica ayudándonos de gnuplot.



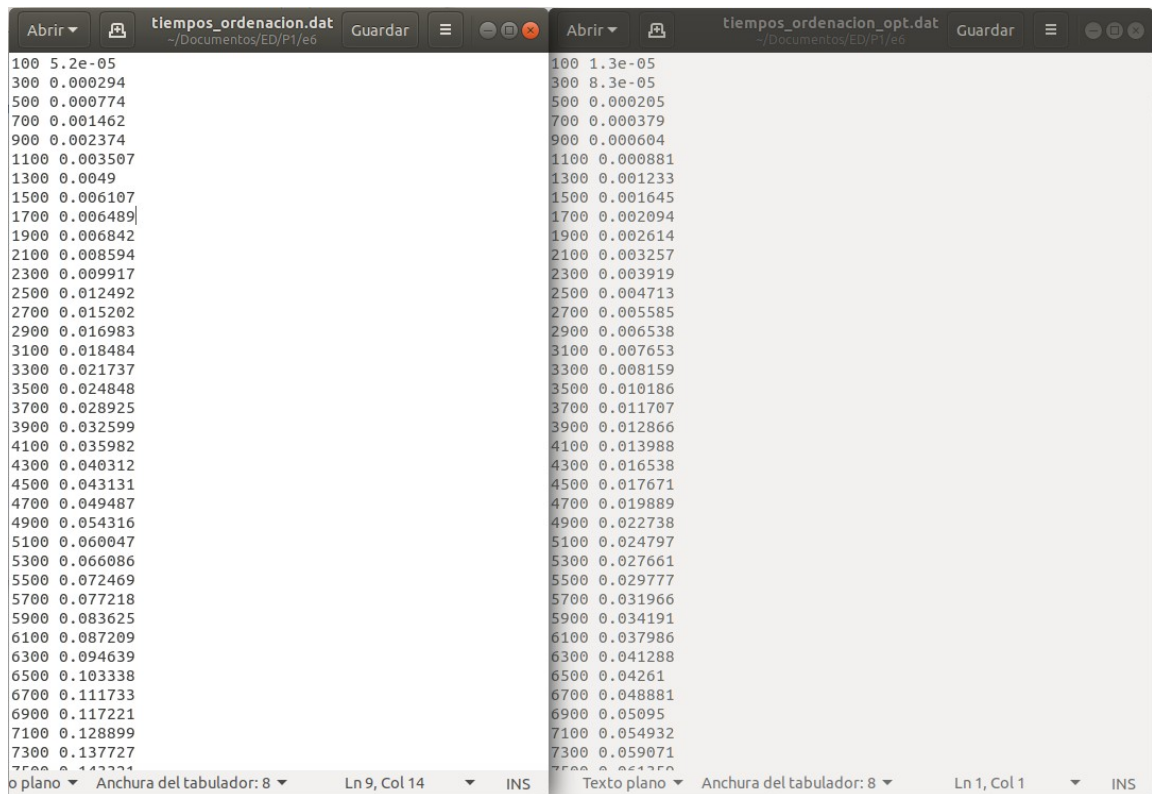
Ejercicio 6: Influencia del proceso de compilación

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
```

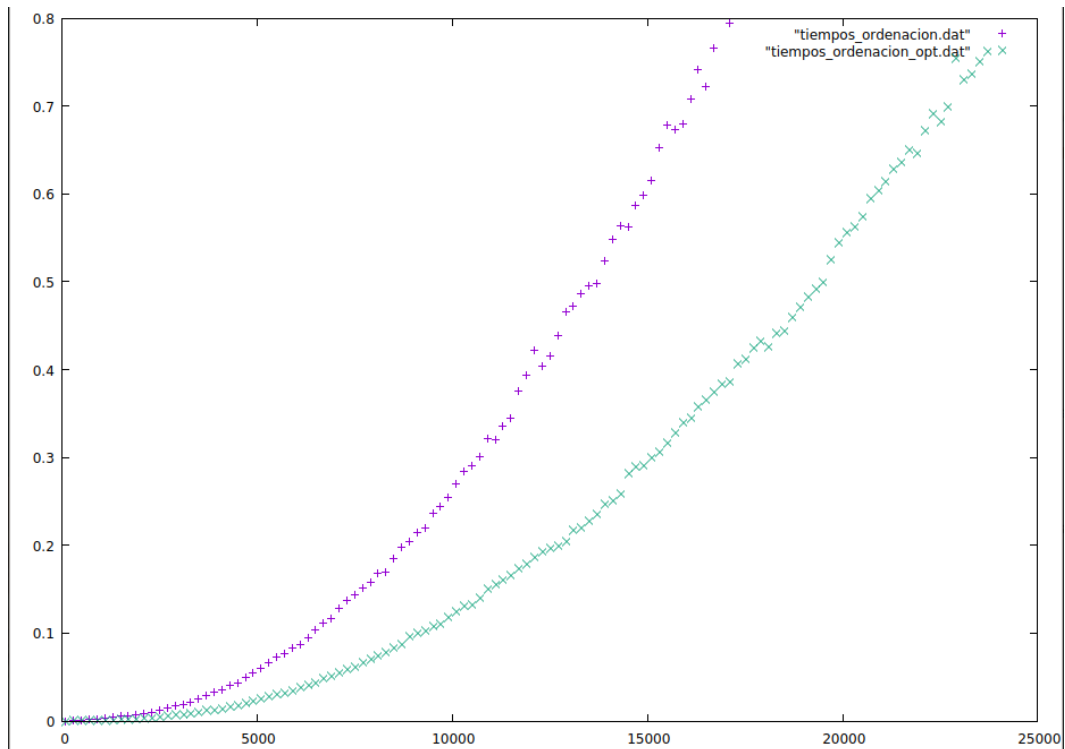
Compare las curvas de eficiencia empírica para ver cómo mejora esto la eficiencia del programa.

Comenzamos compilando el programa de forma optimizada, y creamos otro archivo csh para que lo ejecute y recoja los tiempos en otro fichero distinto al creado para el programa sin optimizar.



tiempos_ordenacion.dat	tiempos_ordenacion_opt.dat
100 5.2e-05	100 1.3e-05
300 0.000294	300 8.3e-05
500 0.000774	500 0.000205
700 0.001462	700 0.000379
900 0.002374	900 0.000604
1100 0.003507	1100 0.000881
1300 0.0049	1300 0.001233
1500 0.006107	1500 0.001645
1700 0.006489	1700 0.002094
1900 0.006842	1900 0.002614
2100 0.008594	2100 0.003257
2300 0.009917	2300 0.003919
2500 0.012492	2500 0.004713
2700 0.015202	2700 0.005585
2900 0.016983	2900 0.006538
3100 0.018484	3100 0.007653
3300 0.021737	3300 0.008159
3500 0.024848	3500 0.010186
3700 0.028925	3700 0.011707
3900 0.032599	3900 0.012866
4100 0.035982	4100 0.013988
4300 0.040312	4300 0.016538
4500 0.043131	4500 0.017671
4700 0.049487	4700 0.019889
4900 0.054316	4900 0.022738
5100 0.060047	5100 0.024797
5300 0.066086	5300 0.027661
5500 0.072469	5500 0.029777
5700 0.077218	5700 0.031966
5900 0.083625	5900 0.034191
6100 0.087209	6100 0.037986
6300 0.094639	6300 0.041288
6500 0.103338	6500 0.04261
6700 0.111733	6700 0.048881
6900 0.117221	6900 0.05095
7100 0.128899	7100 0.054932
7300 0.137727	7300 0.059071
7500 0.147224	7500 0.061250

Si observamos los tiempos de ejecución, ya es notoria la diferencia. A continuación, ayudados por gnuplot, estudiaremos las curvas de eficiencia.



En esta gráfica podemos observar la mejora en la eficiencia del programa optimizado (azul) comparado con el programa inicial (morado).

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.

Implementamos una función llamada `multiplicar` que pide como argumento dos matrices bidimensionales y nos devuelve el producto de las mismas (también implementamos la clase `Matriz2D` y un programa donde se generan dos matrices aleatorias, a las que aplicaremos esta función).

La función `multiplicar` consiste en un algoritmo de orden de eficiencia n^3 . A continuación se muestra el código del programa implementado

```

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

using namespace std;

class Matriz2D { //Definimos la clase Matriz2D
private:
    int **matriz;
    int numeroFilas;
    int numeroColumnas;

    //Metodos privados
    void reservarEspacio(){
        matriz = new int *[numeroFilas];
        for (int i=0; i<numeroFilas; i++){
            matriz[i] = new int [numeroColumnas];
        }
    }

    void liberarEspacio(){
        for(int i=0; i<numeroFilas; i++){
            delete [] matriz[i];
        }
        delete [] matriz;
    }

public:
    //Implementamos un constructor y un destructor
    Matriz2D(int numeroFilas, int numeroColumnas){
        this->numeroFilas=numeroFilas;
        this->numeroColumnas=numeroColumnas;
        reservarEspacio();
    }

    ~Matriz2D(){
        liberarEspacio();
    }

    //Implementamos un metodo para asignar un valor
    bool asignarValor(int fila, int columna, int valor){
        bool asignado=false;
        if(fila<numeroFilas && columna < numeroColumnas){
            asignado=true;
            matriz[fila][columna]=valor;
        }
        return asignado;
    }

    //Definimos metodos constructores
    int getValor(int fila, int columna){
        if(fila<numeroFilas && columna < numeroColumnas){
            return matriz[fila][columna];
        }
    }

    int getNumCol(){
        return numeroColumnas;
    }

    int getNumFil(){
        return numeroFilas;
    }

    //Declaramos nuestra funcion multiplicar amiga
    friend Matriz2D multiplicar(Matriz2D matriz_a,
                                Matriz2D matriz_b);

    //Operador de asignacion
    const Matriz2D & operator=(const Matriz2D & otra){
        liberarEspacio();

        numeroFilas=otra.numeroFilas;
        numeroColumnas=otra.numeroColumnas;

        reservarEspacio();

        for(int i=0; i < numeroFilas; i++){
            for(int j=0; j < numeroColumnas; j++){
                matriz[i][j]=otra.matriz[i][j];
            }
        }

        return *this;
    }

    //Implementamos la funcion multiplicar
    Matriz2D multiplicar (Matriz2D *matriz_a, Matriz2D *matriz_b) {
        Matriz2D *matriz_c = new Matriz2D(matriz_a->getNumFil(),
            matriz_b->getNumCol());
        for(int i=0; i<matriz_c->getNumFil(); i++) {
            for(int j=0; j<matriz_c->getNumCol(); j++){
                int k=0;
                int valor=0;
                for(k=0; k<matriz_a->getNumFil(); k++){
                    valor += (matriz_a->getValor(i,k)*
                        matriz_b->getValor(k,j));
                }
                matriz_c->asignarValor(j, k, valor);
            }
        }
        return *matriz_c;
    }

    void sintaxis()

```

```

    cerr << " TAM: Tamaño de las matrices (>0)" << endl;
    cerr << "Se genera una matriz de tamaño TAM*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]);
    int vmax=atoi(argv[2]);
    if (tam<0 || vmax<=0)
        sintaxis();

    Matriz2D *matriz_a = new Matriz2D(tam, tam);
    Matriz2D *matriz_b = new Matriz2D(tam, tam);
    Matriz2D *matriz_c = new Matriz2D(tam, tam);

    // Generación de dos matrices aleatorias
    srand(time(0));
    for (int i=0; i<tam; i++) {
        for (int j=0; j<tam;j++) {
            matriz_a->asignarValor(i, j, rand() % vmax);
        }
    }

    srand(time(0));
    for (int i=0; i<tam; i++) {
        for (int j=0; j<tam;j++) {
            matriz_b->asignarValor(i, j, rand() % vmax);
        }
    }

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

    *matriz_c = multiplicar(matriz_a, matriz_b);

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

    // Mostramos tiempo
    cout << tam << " (tfin-tini)/(double)CLOCKS_PER_SEC << endl;

    delete matriz_a;
    delete matriz_b;
    delete matriz_c;
}

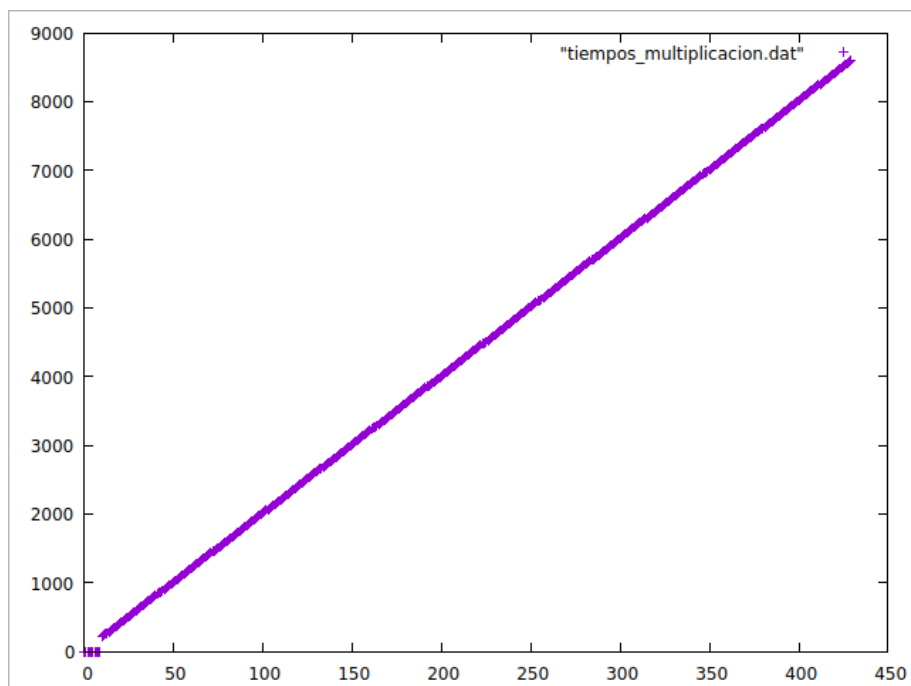
```

Como hemos realizado en ejercicios anteriores, vamos a estudiar la eficiencia de este algoritmo con la ayuda de gnuplot. Comenzamos creando un archivo csh.

```
#!/bin/csh
@ inicio = 1
@ fin = 1000
@ incremento = 2
set ejecutable = multiplicacion
set salida = tiempos_multiplicacion.dat

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

Al ejecutar este archivo se genera “tiempos_multiplicacion.dat”, el cual nos proporcionará la siguiente gráfica, de la cual deducimos que nuestra función multiplicar no es muy eficiente.



Ejercicio 8: Ordenación por Mezcla

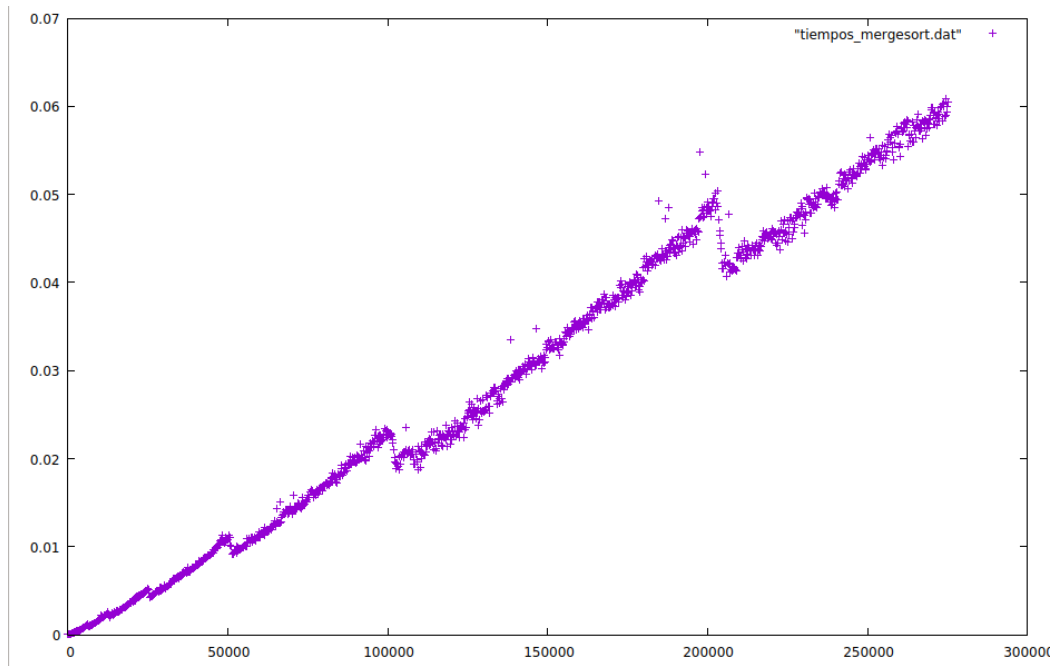
Estudie el código del algoritmo recursivo disponible en el fichero Mergesort.cpp. En él, se integran dos algoritmos de ordenación: inserción y mezcla (o mergesort). El parámetro UMBRAL_MS condiciona el tamaño mínimo del vector para utilizar el algoritmo de inserción en vez de seguir aplicando de forma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

Primero compilamos mergesort.cpp, y creamos el archivo csh que ejecutará el programa y nos dará los tiempos de ejecución del mismo, lo cual nos facilitará el estudio de su eficiencia empírica.

```
#!/bin/csh
@ inicio = 100
@ fin = 1000000
@ incremento = 200
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
```

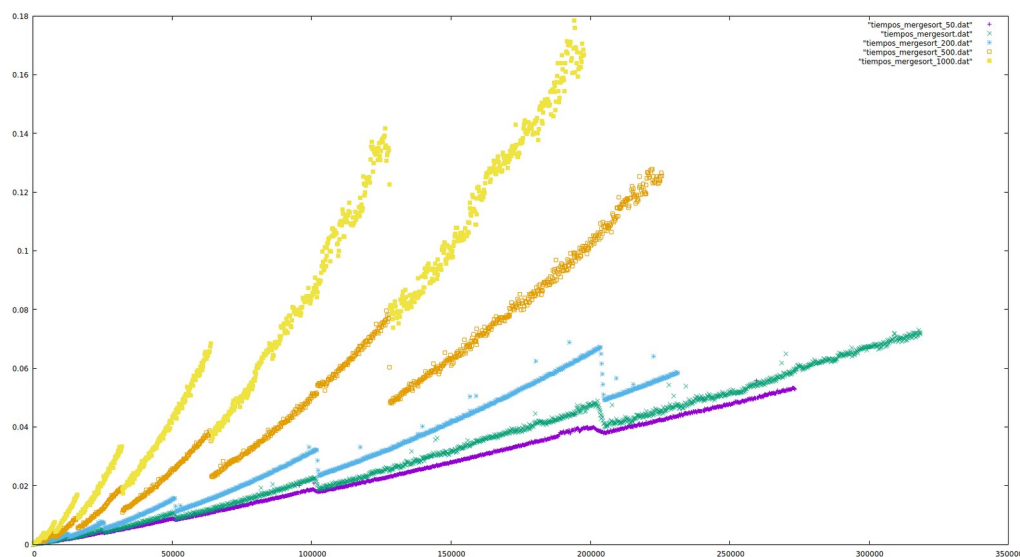
Una vez tenemos los datos guardados en tiempos_mergesort.dat, llamamos a gnuplot desde el terminal.



A continuación estudiaremos el efecto del parámetro UMBRAL_MS sobre la eficiencia del programa. Para ello, cambiaremos el valor del mismo y compararemos las gráficas.

Los valores con los que probaremos el programa serán 100 (valor por defecto), 50, 200, 500 y 1000.

A continuación mostramos una gráfica con el tiempo de ejecución de dichos valores, realizada mediante gnuplot:



En la gráfica observamos que los tiempos de ejecución del programa con el valor de 50 presenta una disposición que podría considerarse constante. Sin embargo, a medida que aumenta el valor del parámetro, aparecen discontinuidades (saltos finitos).