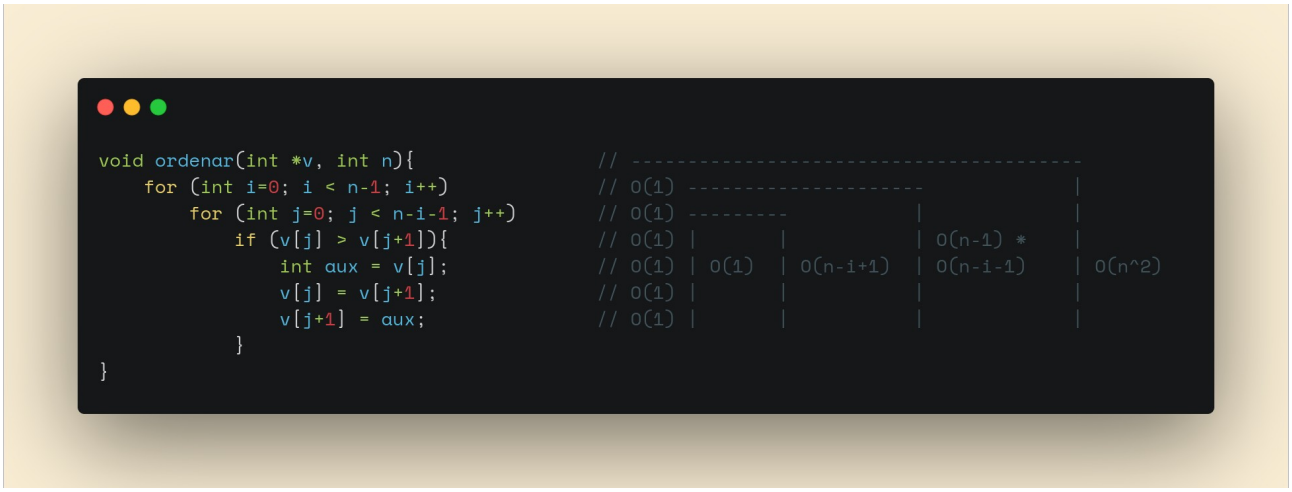
Práctica 1 - Estructura de Datos

Salvador Romero Cortés

Ejercicio 1. Ordenación de la burbuja
El siguiente código realiza la ordenación mediante el algoritmo de la burbuja:



```
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;
            }
}
```

```
// ----------------------------------------
// O(1) ---------------------              |
// O(1) ---------          |              |
// O(1) |       |          | O(n-1) *     |
// O(1) | O(1)  | O(n-i+1) | O(n-i-1)     | O(n^2)
// O(1) |       |          |              |
// O(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_ordenaciones.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?

Hemos calculado la eficiencia teórica en la captura del código, como podemos ver la eficiencia es de `O(n^2)`.
Lo calculamos también con sumatorias:

$$\Sigma_{i=0}^{n-1}(n-i-1) = n^2 - n - \frac{n(n-1)}{2}$$

$$\Sigma_{i=0}^{n-1}(n-i-1) = \frac{1}{2}(n^2-n)$$

De la última expresión nos quedamos sólo con el factor de mayor grado y por tanto obtenemos que la eficiencia es O(n^2).

Práctica 1 - Estructura de Datos

Salvador Romero Cortés

Código de `ordenacion.cpp`

```cpp
/**
 * @file Ordenacion por burbuja
 * @author Salvador Romero Cortés
 */

#include <iostream>
#include <ctime>
#include <cstdlib>

using namespace std;

void ordenar(int *v, int n){                // ----------------------------------------
    for (int i=0; i < n-1; i++)             // O(1) ---------------------              |
        for (int j=0; j < n-i-1; j++)       // O(1) ---------           |              |
            if (v[j] > v[j+1]){             // O(1) |       |           | O(n-1) *     |
                int aux = v[j];             // O(1) | O(1)  | O(n-i+1)  | O(n-i-1)     | O(n^2)
                v[j] = v[j+1];              // O(1) |       |           |              |
                v[j+1] = aux;               // O(1) |       |           |              |
            }
}

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 está 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
}
```

Práctica 1 - Estructura de Datos

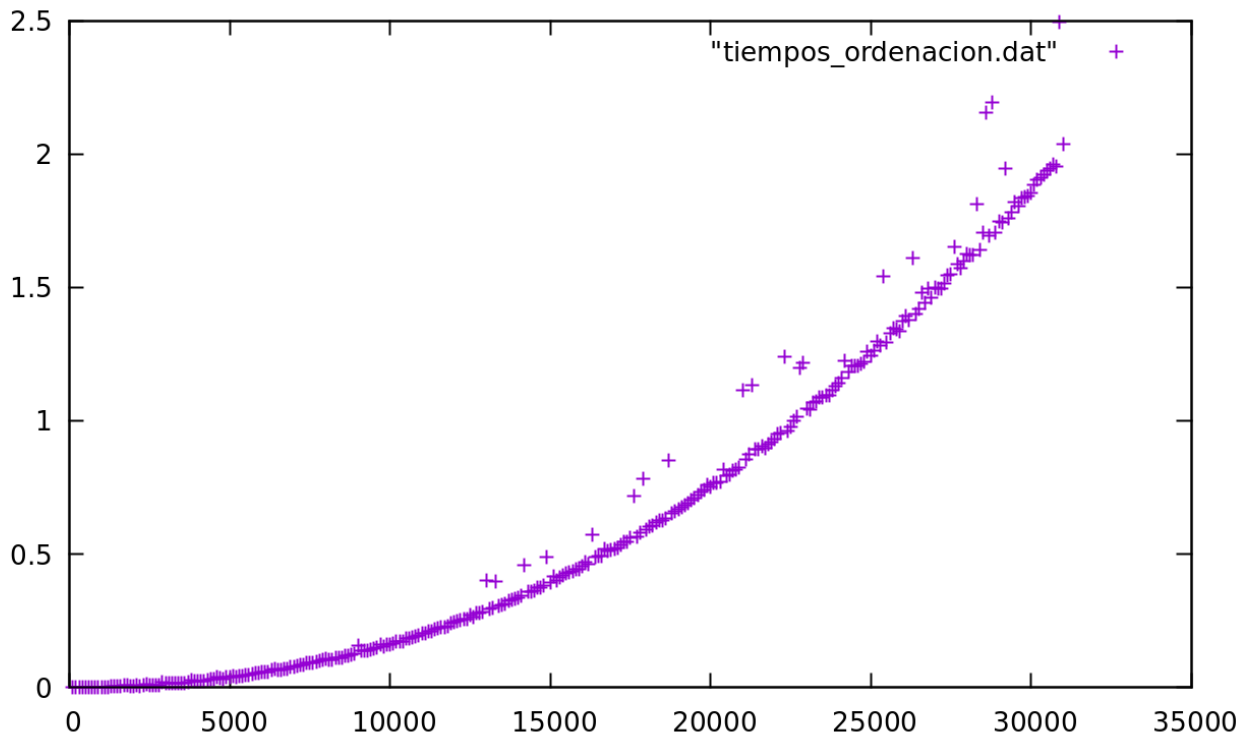
Salvador Romero Cortés

Código del `script` para ejecutarlo

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

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

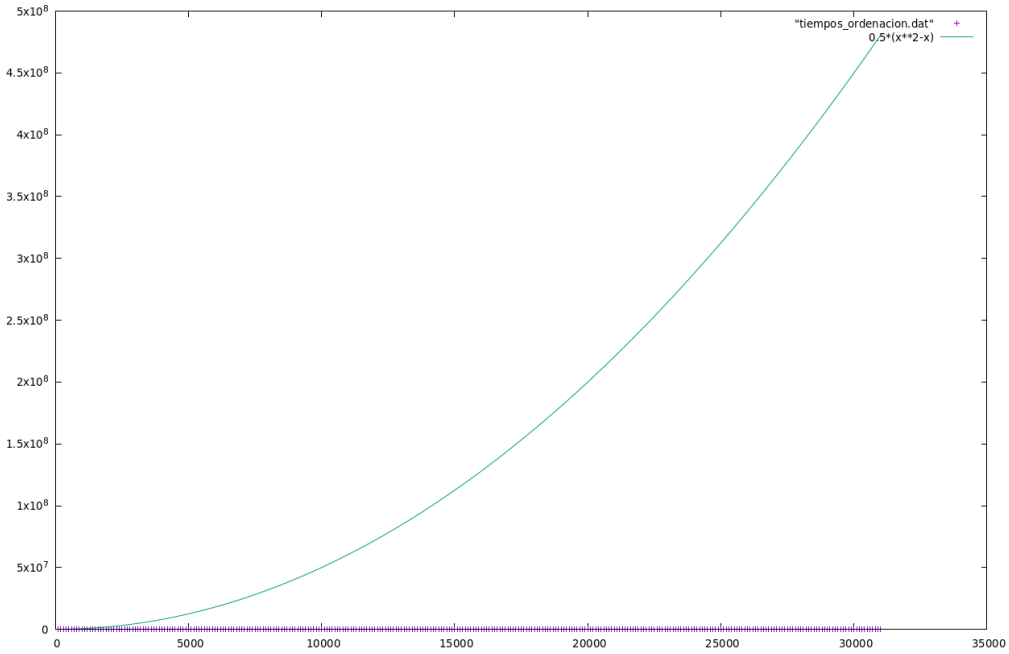
Representamos gráficamente los datos obtenidos



Práctica 1 - Estructura de Datos

Salvador Romero Cortés

Comparamos la función teórica con los resultados reales. (Sin ajustar los datos)



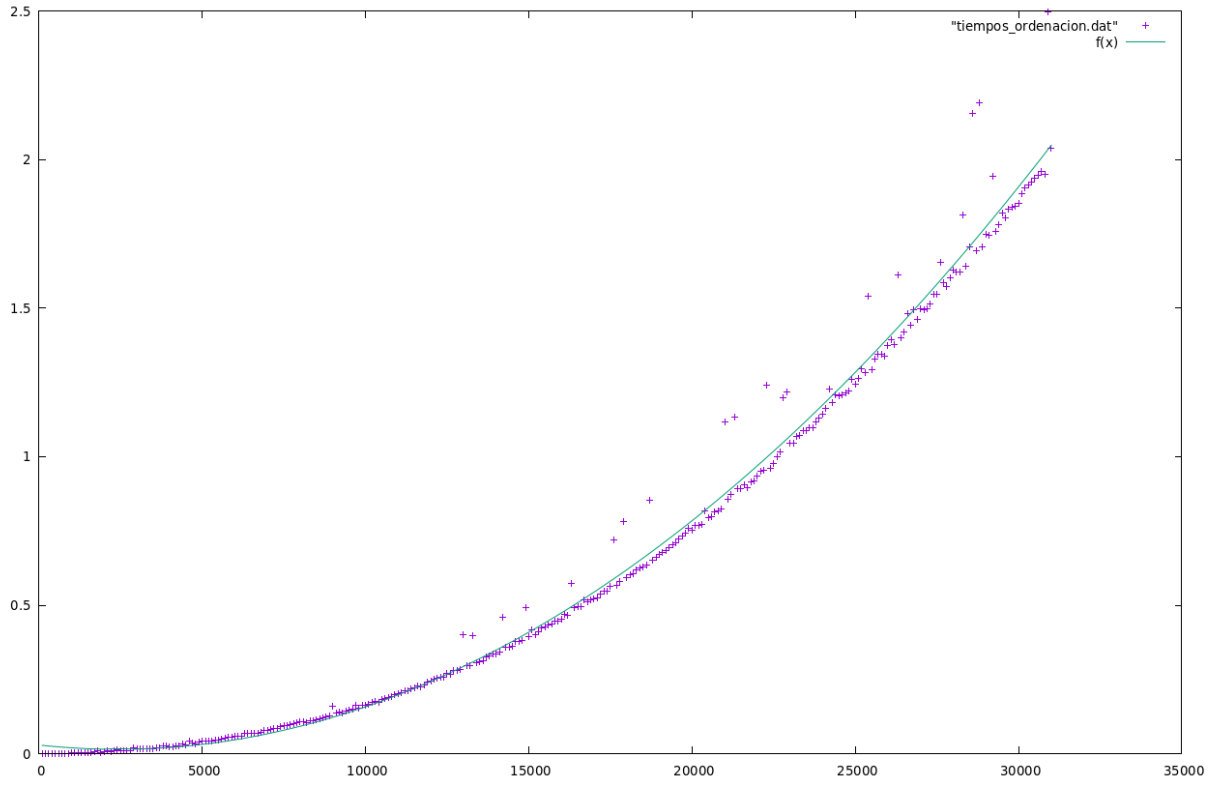
Ejercicio 2. Ajuste en la ordenación de la burbuja

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 ax2+bx+c

Ajustamos en gnuplot la función con las ordenes:

```
gnuplot> f(x) = a*x**2 +b*x +c
gnuplot> fit f(x) "tiempos_ordenacion.dat" via a,b,c
```

Y si mostramos el gráfico ahora obtenemos:



Vemos que ahora sí que se ajusta bien al resultado teórico.

Ejercicio 3. Problemas de precisión
Junto con este guion 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 evolución es correcta. Una vez resuelto el problema realice la regresión para ajustar la curva teórica a la empírica.

```
int operacion(int *v, int n, int x, int inf, int sup) {     //--------------------------|
  int med;                                                  // O(1)                     |
  bool enc=false;                                           // O(1)                     |
  while ((inf<sup) && (!enc)) {                             // O(1) ----|               |
    med = (inf+sup)/2;                                      // O(1)     |               |
    if (v[med]==x)                                          // O(1)     |               |
      enc = true;                                           // O(1)     | O(log2(n))    |
    else if (v[med] < x)                                    // O(1)     |               | O(log2(n))
      inf = med+1;                                          // O(1)     |               |
    else                                                    //          |               |
      sup = med-1;                                          // O(1)     |               |
  }                                                         // O(1) ----|               |
  if (enc)                                                  // O(1)                     |
    return med;                                             // O(1)                     |
  else                                                      //                          |
    return -1;                                              // O(1)                     |
}                                                           // -------------------------|
```

Se trata de un algoritmo de búsqueda binaria. Sirve para encontrar un elemento en un vector ordenado.
La eficiencia teórica es de O(log2(n)) puesto que con cada iteración del bucle, se reduce a la mitad el tamaño del vector. El resto de operaciones son O(1).

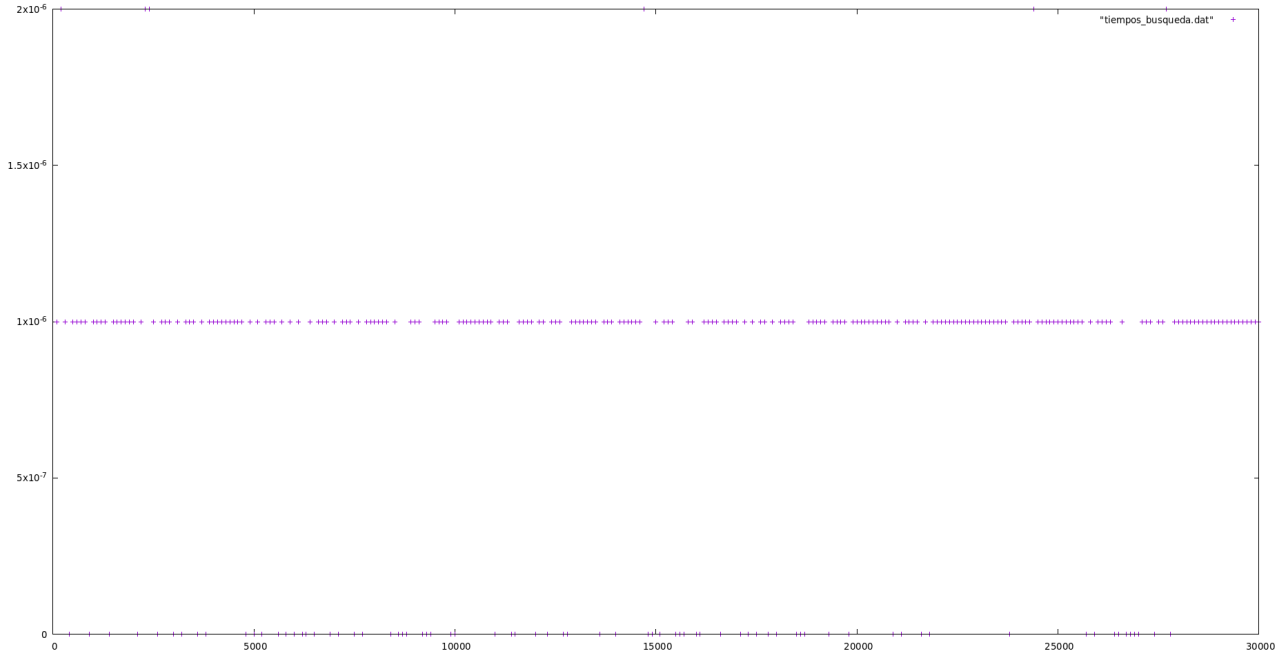
Sin embargo al compilar y ejecutar el programa nos damos cuenta de que falla un pre-requisito de la búsqueda binaria. El vector no está ordenado. Además si lo representamos gráficamente podemos observar que tarda siempre el mismo tiempo, esto se debe a un error de precisión, para solucionarlo ejecutamos muchas veces la búsqueda y dividimos por ese número el tiempo total.

Además podemos encontrar una pequeña errata en la condición del `while` ya que debería ser `(inf <= sup)`.

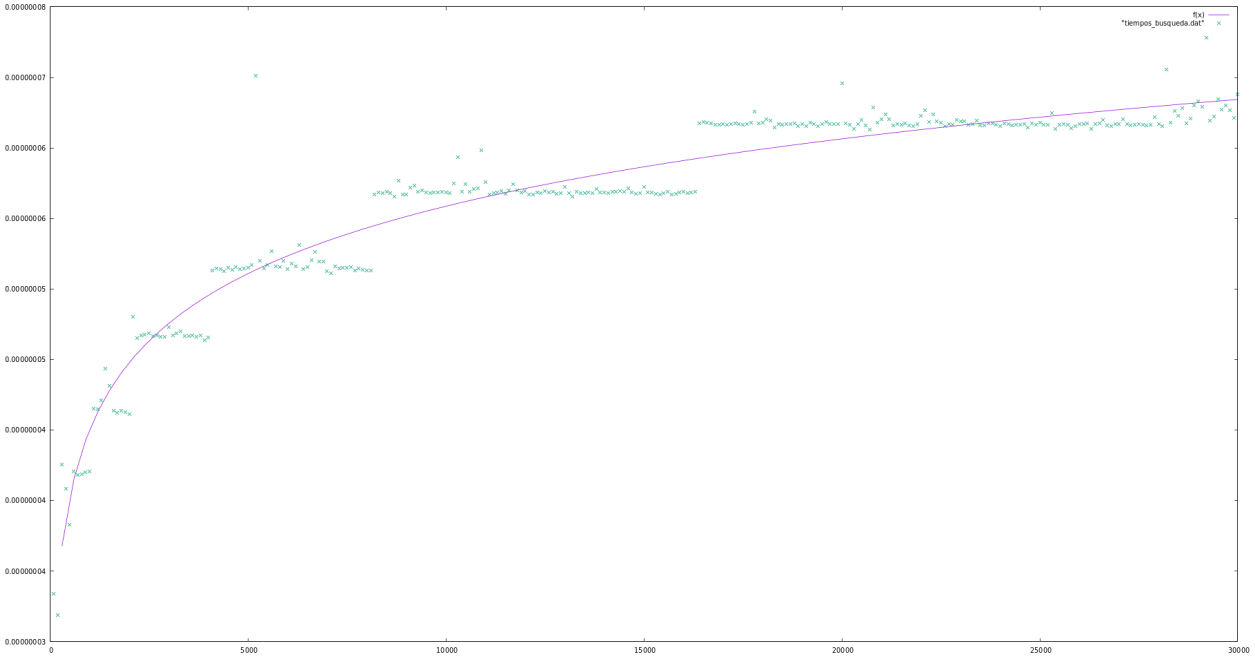
Práctica 1 - Estructura de Datos

Salvador Romero Cortés

Sin solucionar el problema de precisión:



Una vez solucionado el problema de precisión (y ajustado a la eficiencia teórica):



Vemos que se acerca a lo previsto teóricamente.
La solución de repetir la búsqueda muchas veces aparece en el `main` como:

```
for (int i=0; i < 1000000; i++)
  bool enc = operacion(v,tam, tam+1, 0, tam - 1) == -1;
//acaba el reloj...
cout << tam << "\t" << ((tfin - tini)/1000000.0) / (double)CLOCKS_PER_SEC << endl;
```

Práctica 1 - Estructura de Datos

Salvador Romero Cortés

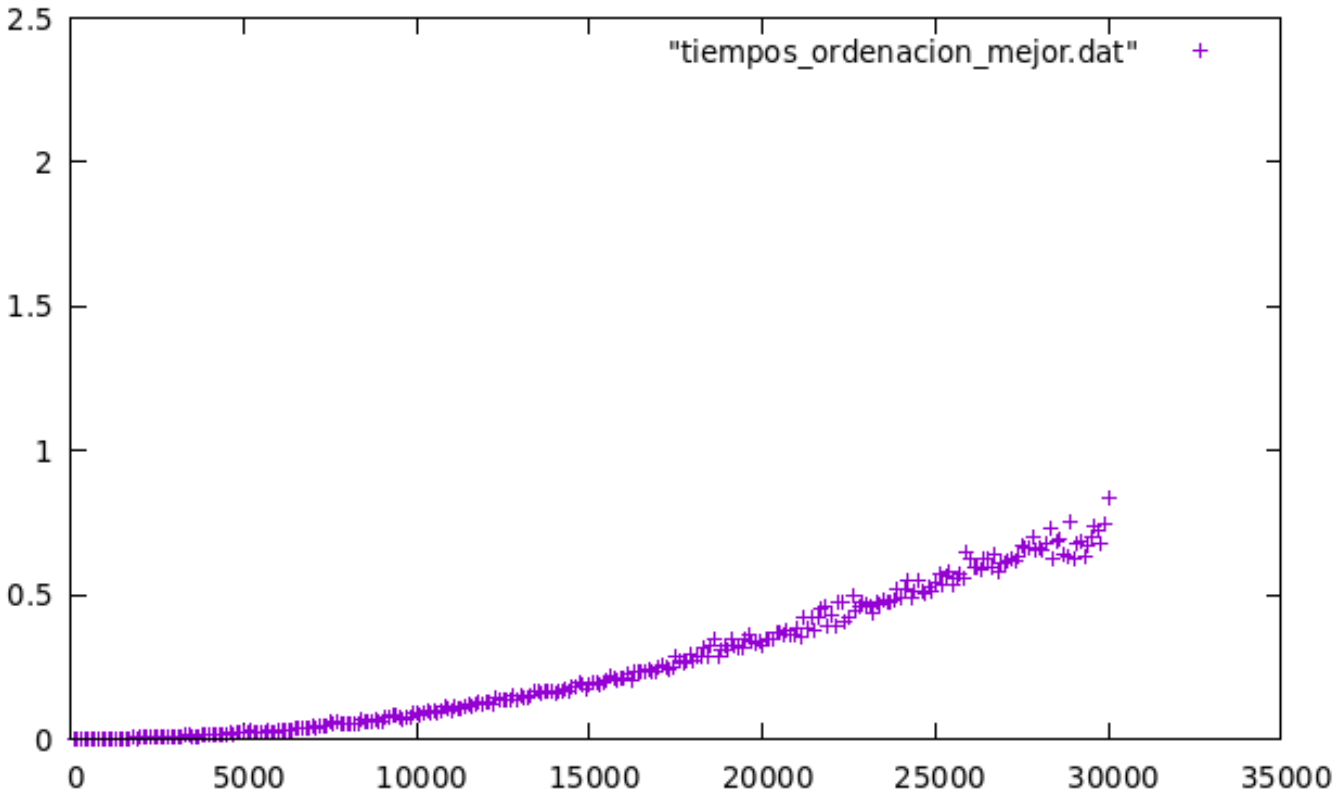
Ejercicio 4. Mejor y peor caso.

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:

- 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.
- 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. Aunque lo más frecuente será preguntar por números que estén en posiciones arbitrarias del vector (casos promedio).

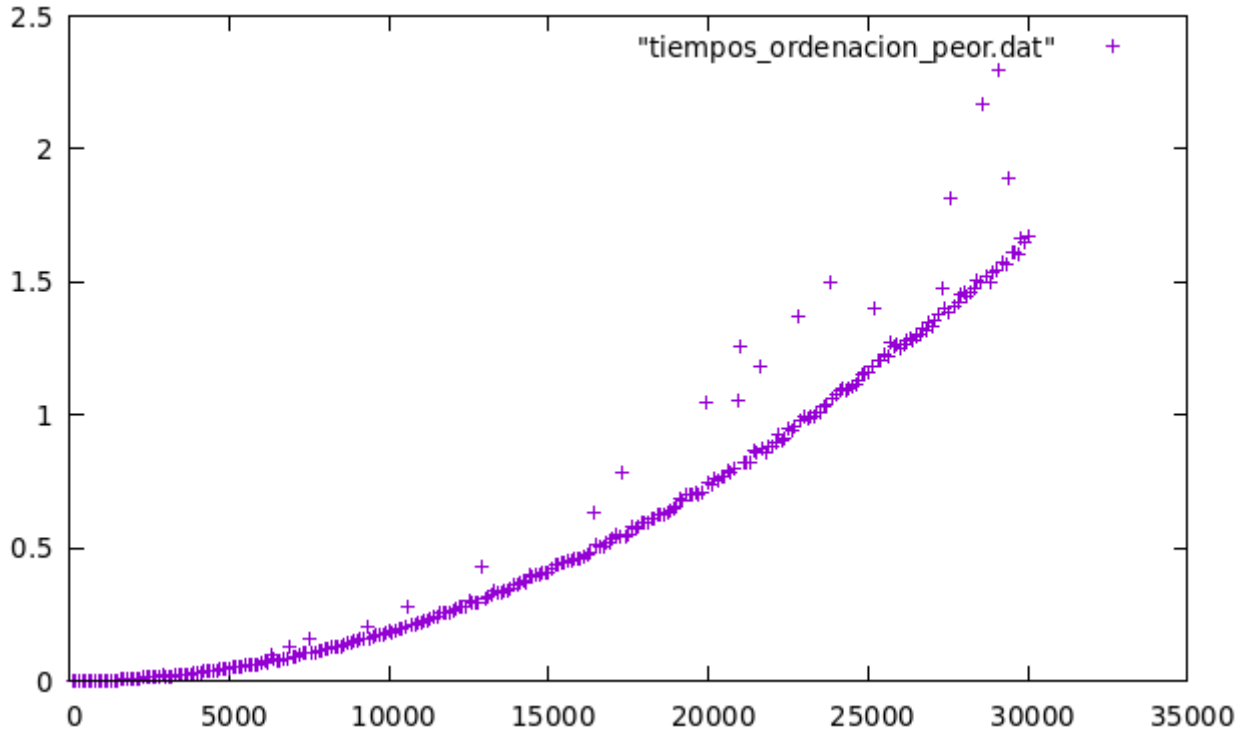
Mejor caso posible:



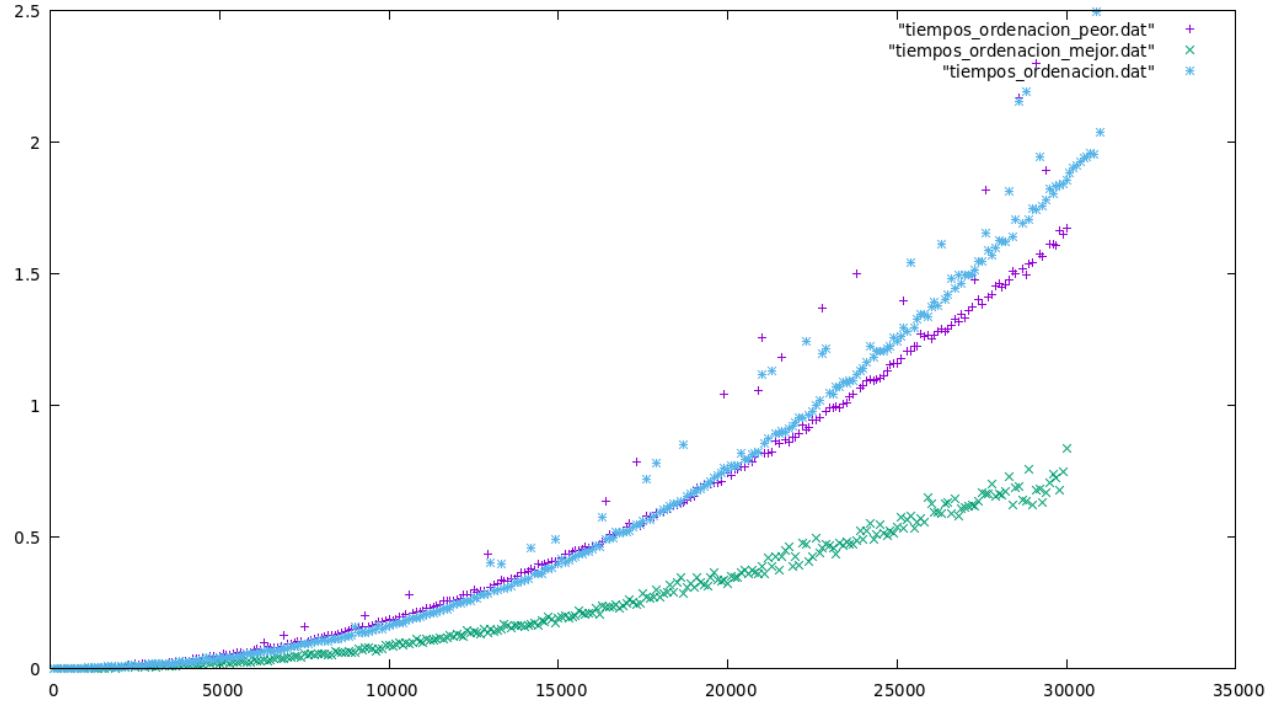
Práctica 1 - Estructura de Datos

Salvador Romero Cortés

Peor caso



Comparación del mejor, peor y promedio.

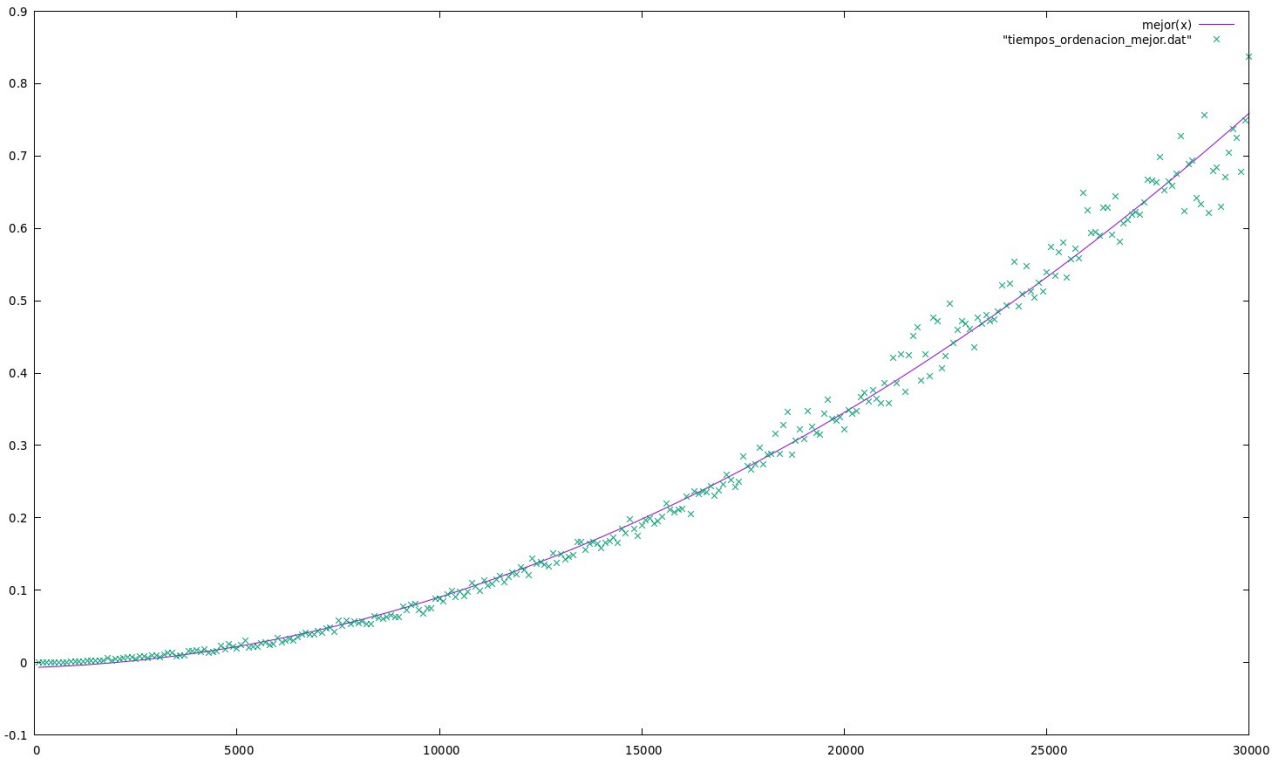


Verde: mejor caso
Morado: peor caso
Azul: promedio

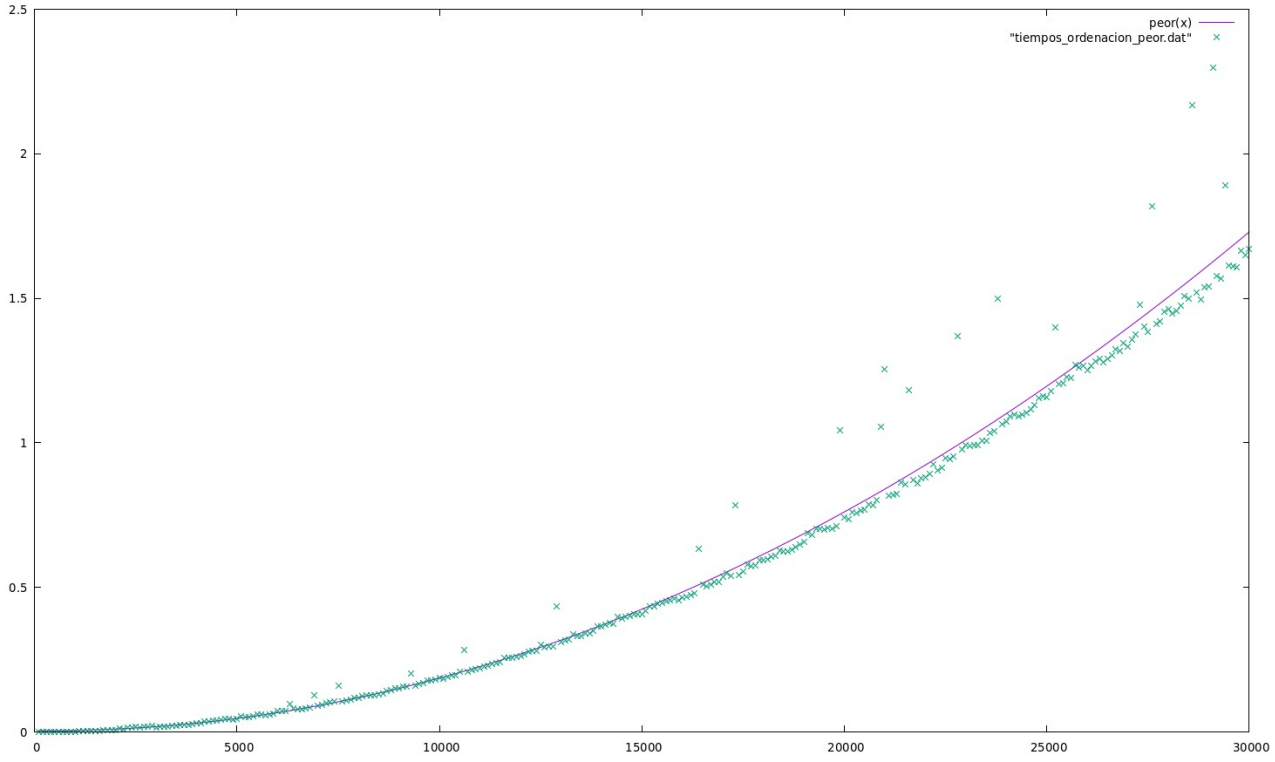
Práctica 1 - Estructura de Datos

Salvador Romero Cortés

Hacemos regresión sobre el mejor y peor caso
Mejor:



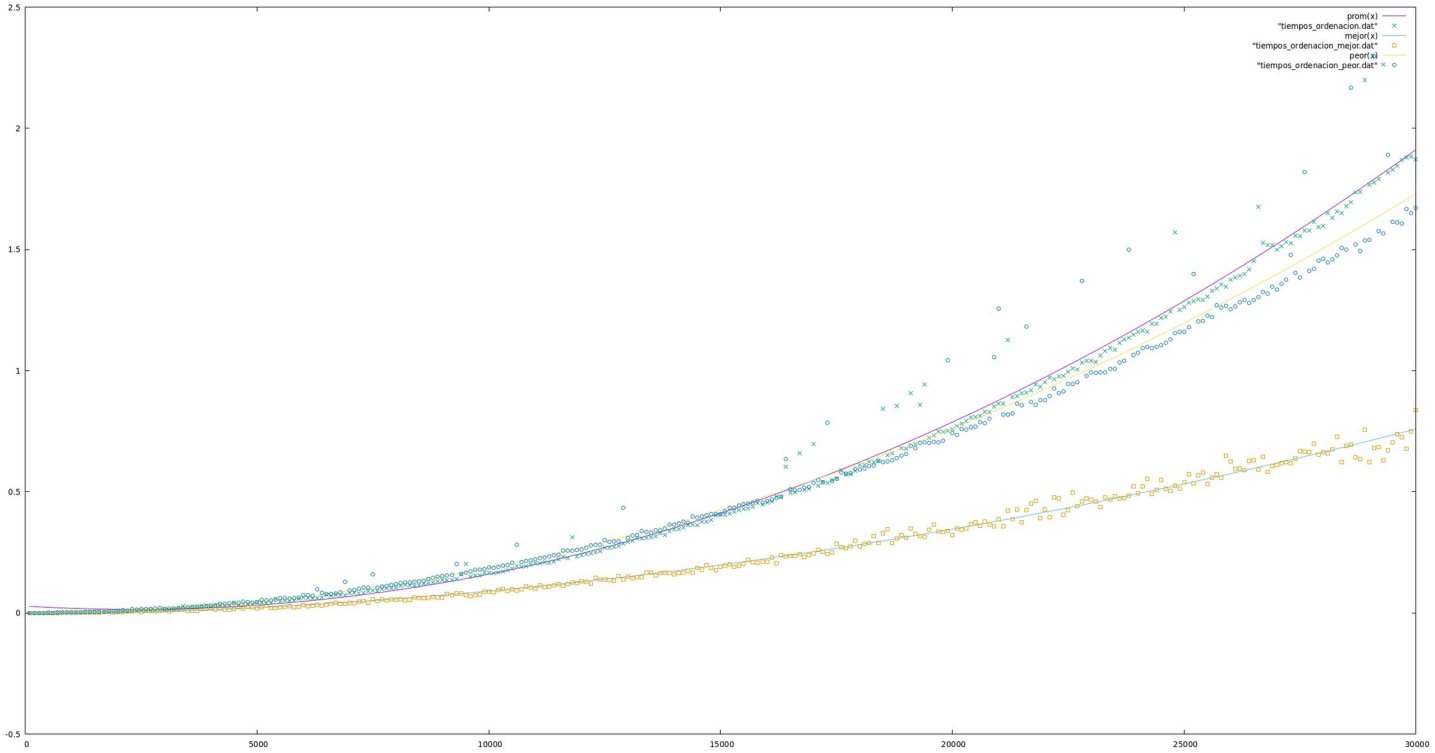
Peor:



Práctica 1 - Estructura de Datos

Salvador Romero Cortés

También podemos realizar la regresión sobre los 3 resultados.



Práctica 1 - Estructura de Datos

Salvador Romero Cortés

Detalles técnicos sobre la práctica.

- Código fuente: ficheros en el zip
- Hardware usado:
  - CPU: AMD Ryzen 5 3600, 3,6GHz-4,2GHz
  - Memoria RAM: 4x4GB DDR4 2400MHz.
- Sistema Operativo: Ubuntu 20.04
- Compilador: g++ sin ninguna opción extra.

Todas las imágenes se incluyen como ficheros aparte para poder apreciarlas mejor.