Practica 1 eficiencia

La práctica ha sido realizada por: Oleksandr Kudryavtsev.

Utilizando el siguiente equipo:
Procesador: i5 7200u
SO: Ubuntu 18.04, compilador: g++

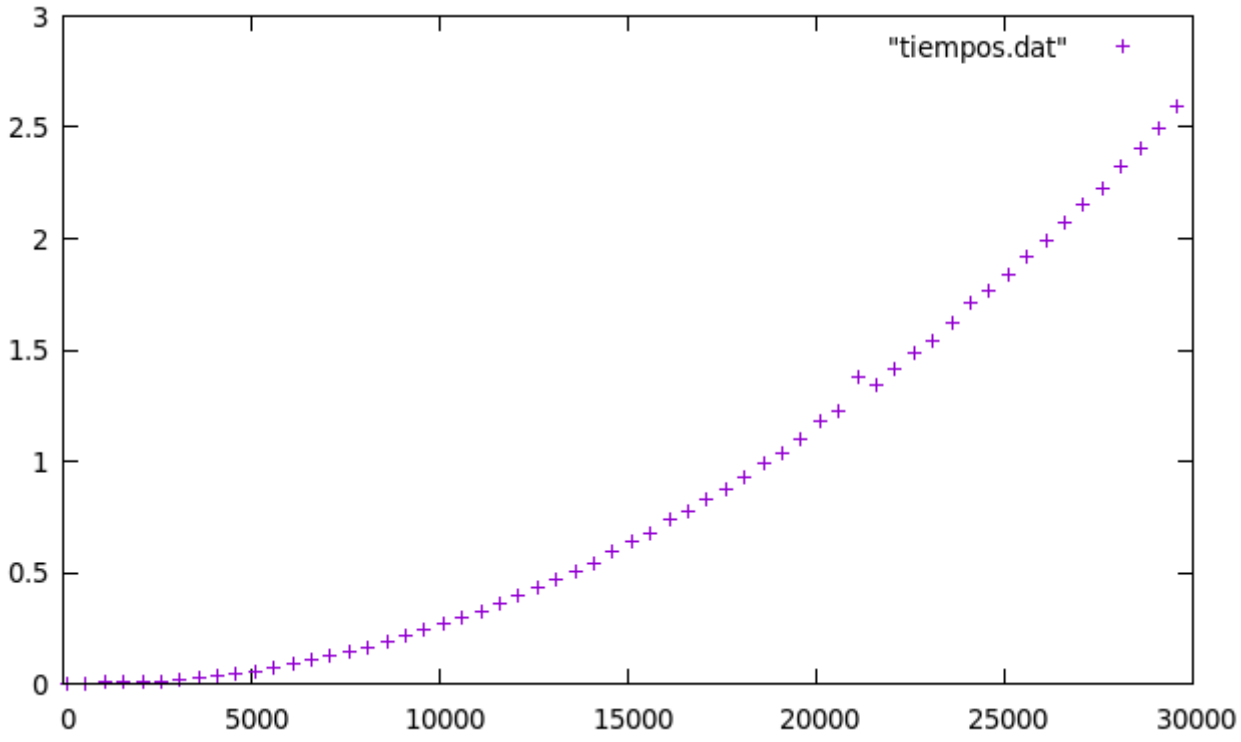
Ejercicio 1: Ordenación de la burbuja.

Eficiencia teórica:

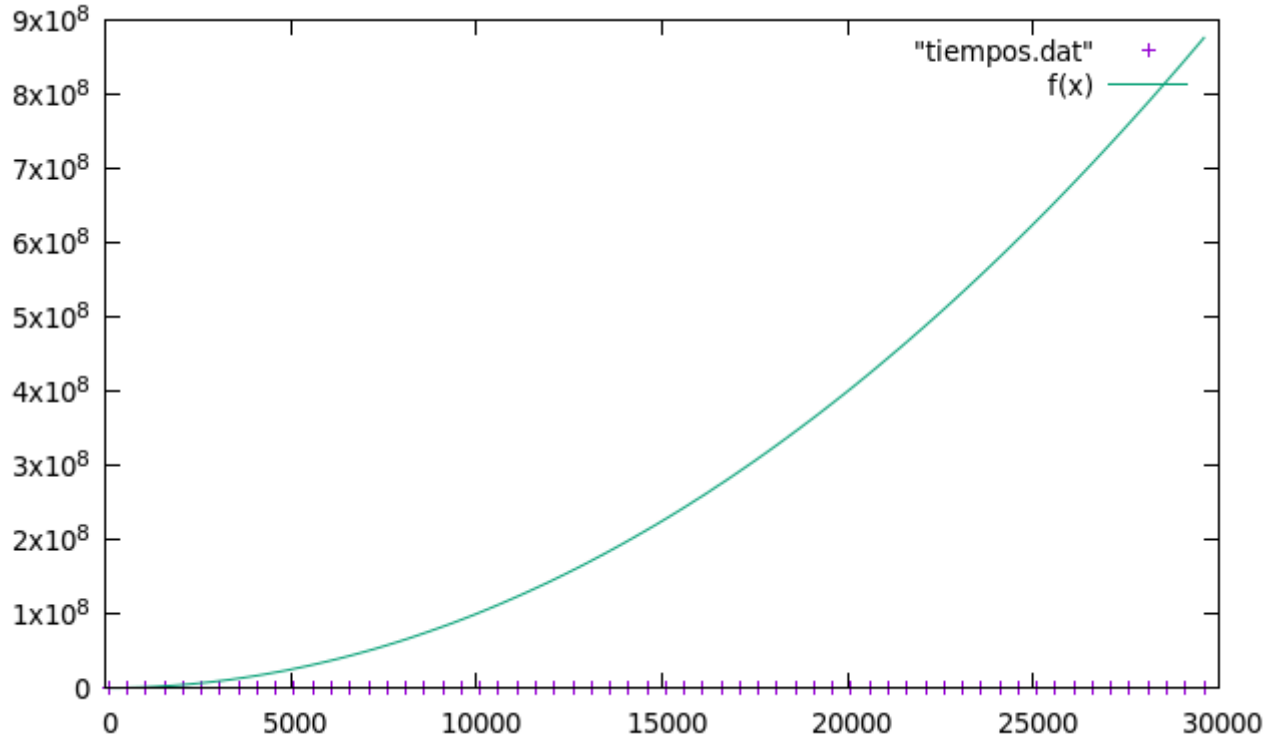
```
void ordenar(int *v, int n) {

    for (int i=0; i<n-1; i++)                          // O(n)
        for (int j=0; j<n-i-1; j++)                  // O(n)
            if (v[j]>v[j+1]) {                     // O(1)
                int aux = v[j];              // O(1)
                v[j] = v[j+1];               // O(1)
                v[j+1] = aux;                // O(1)
            }

}
```

Por la anterior captura se ve que el algoritmo destaca por un doble bucle for, con una eficiencia O(n) cada uno, las demás lineas son asignaciones y comparaciones de orden O(1). Como es un doble bucle for y su contenido es despreciable al tener eficiencia O(1) la eficiencia del algoritmo es de $O(n^2)$.

Eficiencia empírica:
Gráfica dibujada con Gnuplot.



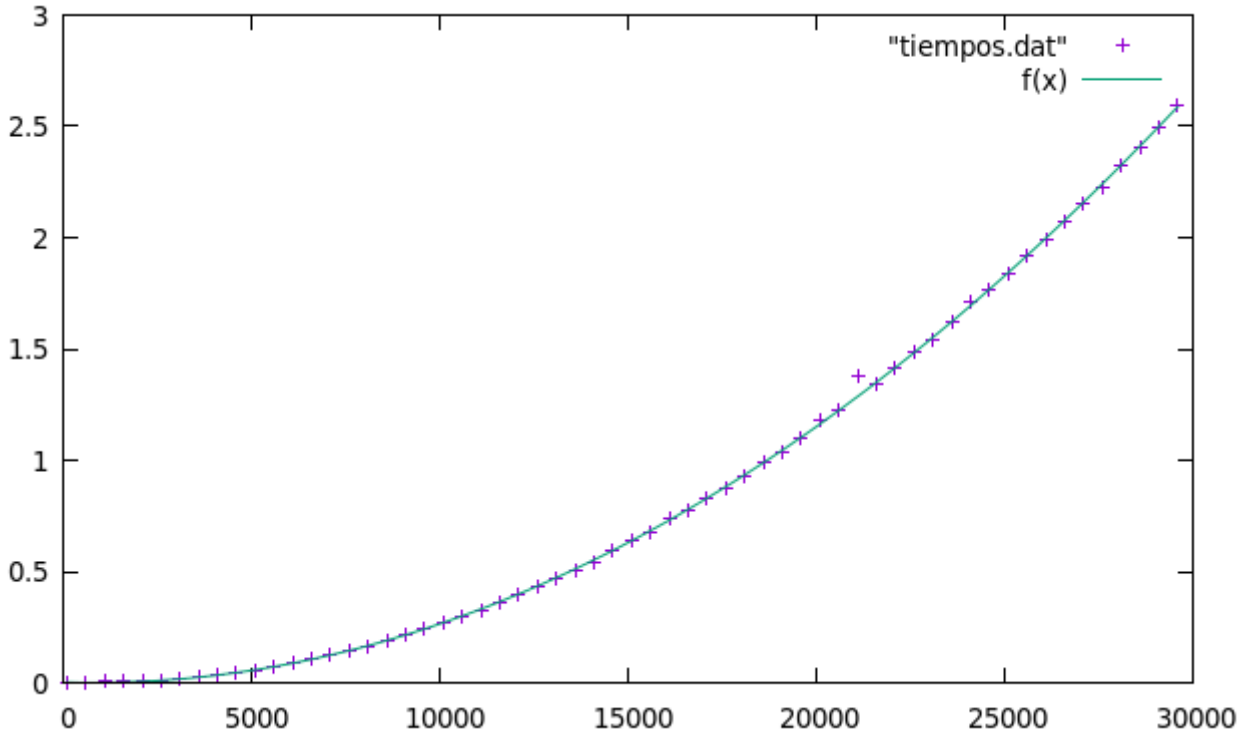
Superposición de la función de eficiencia teórica $O(n^2)$ es decir $f(x) = x^2$ y la gráfica obtenida con los datos de las ejecuciones.



Observando la gráfica podemos ver que la eficiencia empírica es mucho menor que la teórica.

Ejercicio 2: Ajuste en la ordenación burbuja.

Gráfica obtenida:



Ejercicio 3: Problemas de precisión.

Explicación del algoritmo:

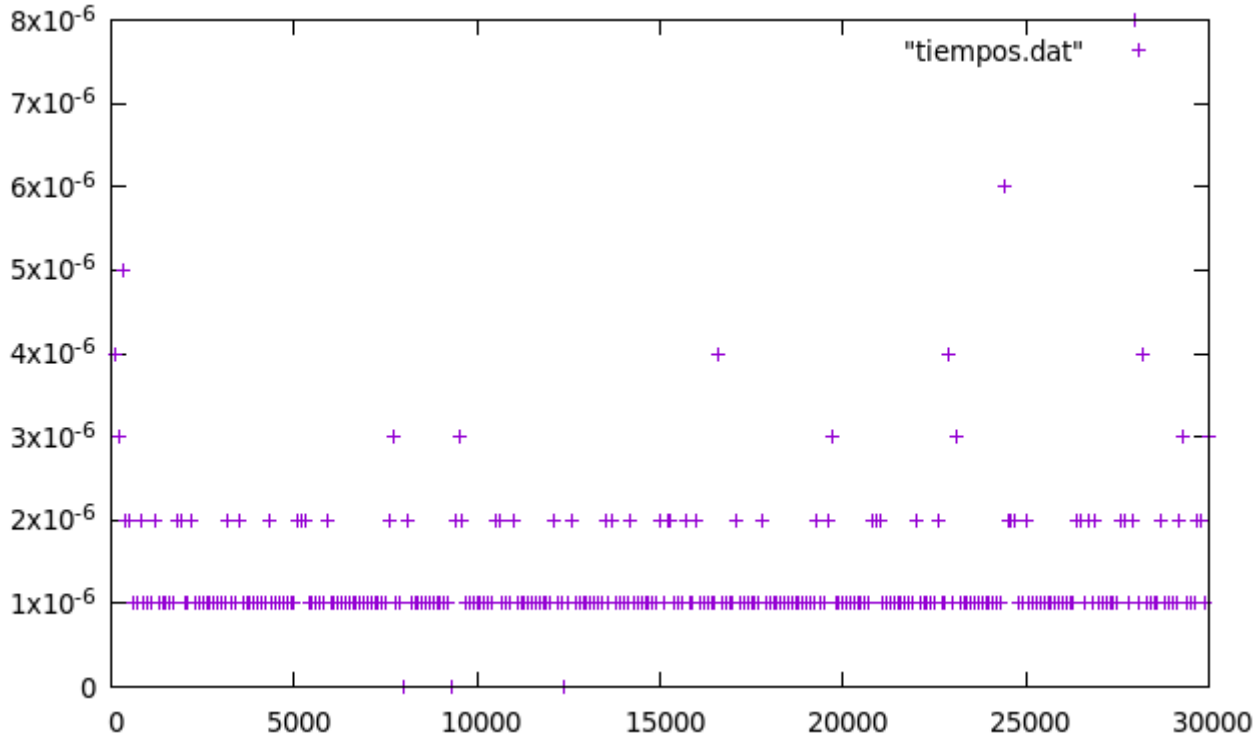
El programa recibe como argumento el tamaño que se le quiera dar al vector, a continuación este se llena con números aleatorios menores o iguales que “tam”. Teniendo este vector se llama a la función “operación”. Esta función busca el elemento “x”, que en este caso es tam+1 por lo tanto nunca lo llega a encontrar, es decir se da el peor caso. El algoritmo consiste en calcular la posición central del vector usando las variables “inf” y “sup”, y a continuación mirar si el elemento buscado esta en dicha posición. Si el elemento no se encuentra (siempre se da este caso) se mira si el elemento buscado es mayor o menor que el elemento encontrado y dependiendo del resultado se modifican los valores de “inf” y “sup”. Esto se repite hasta que no queden elementos que comprobar o se haya encontrado el elemento buscado.

Este algoritmo esta pensado para buscar un elemento en un vector ordenado pero en este caso el elemento buscado no está en el vector y este esta desordenado. Es el peor caso posible para la eficiencia.

Eficiencia teórica:

El algoritmo recorre todos los elementos del vector por lo que la eficiencia del algoritmo es O(n). Los demás elementos son despreciables ya que son de tiempo constante.

Eficiencia empírica:
Sin solucionar el problema del algoritmo

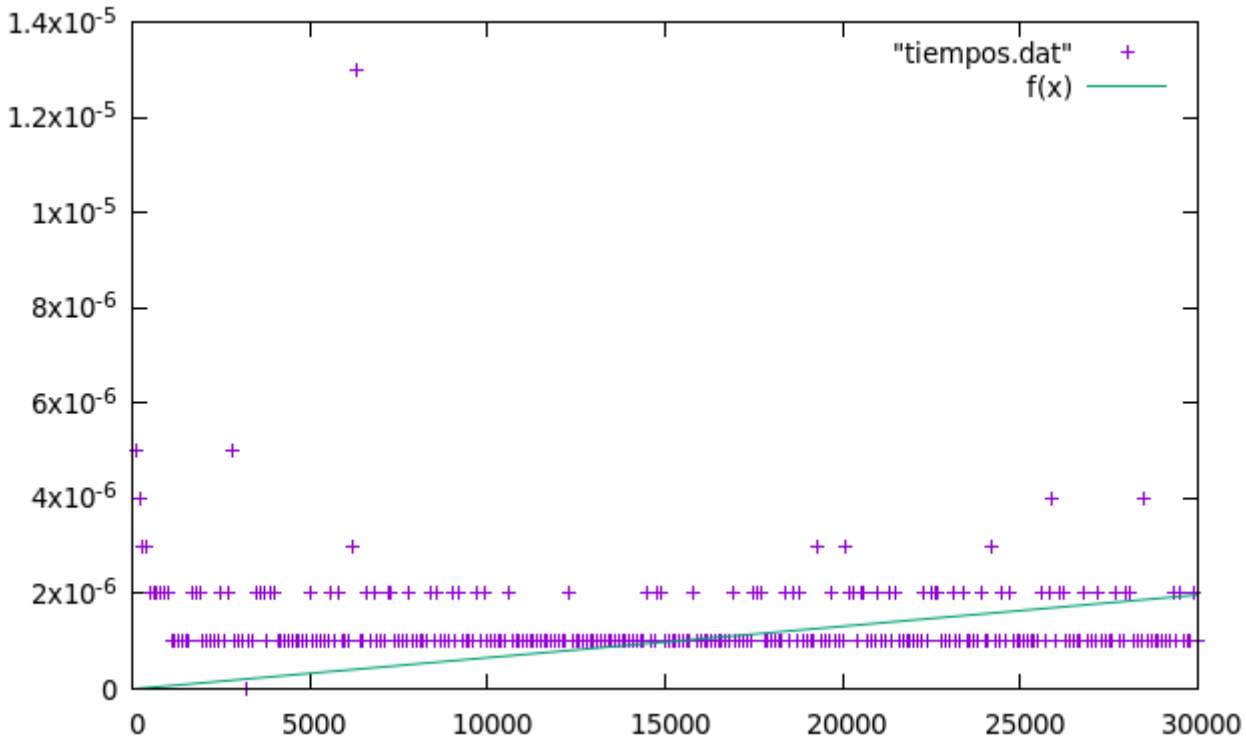


Observando el algoritmo se ve que nunca puede encontrar el elemento buscado y que el vector dado es desordenado cuando el algoritmo esta pensado para vectores ordenados. Una solución seria modificar el código para que se busque un elemento posible de encontrar y que se busque en un vector ordenado.

Código modificado:

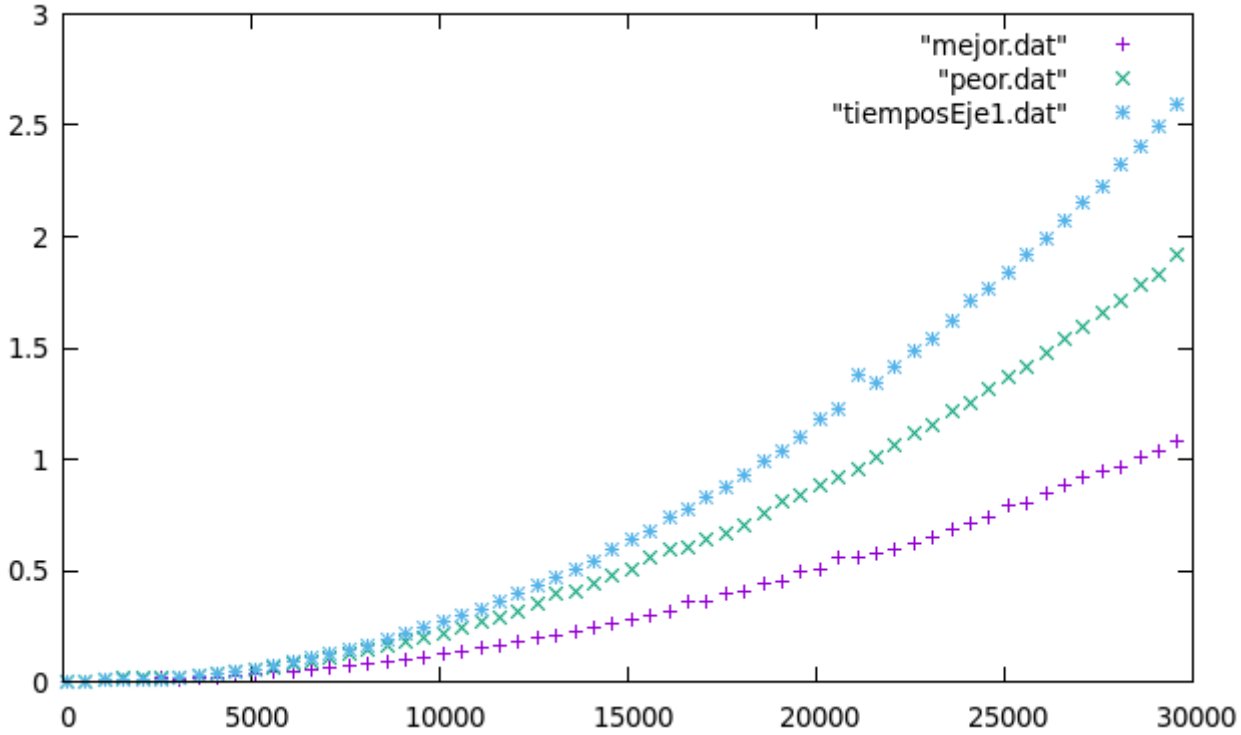
```
// Generación del vector ordenado
int *v=new int[tam];
for (int i=0; i<tam; i++)
  v[i] = i;

//Buscamos un numero aleatorio dentro del vector
srand(time(0));
int numeroBuscar = rand()%tam-1;

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

// Algoritmo a evaluar
operacion(v,tam,numeroBuscar,0,tam-1);
```

Resultados obtenidos con el código modificado:



Ejercicio 4: Mejor y peor caso.

Gráfica resultante:

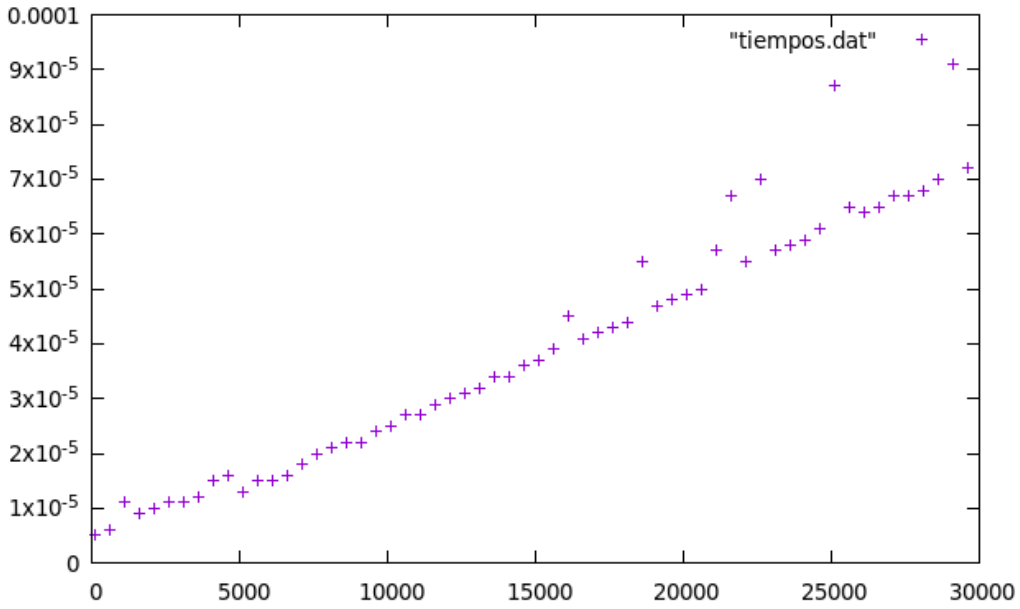


Es muy notable la diferencia entre el mejor y el peor caso. También se aprecia que en el ejercicio 1 donde eran elementos aleatorios se necesita aun mas tiempo.

Ejercicio 5: Dependencia de la implementación.

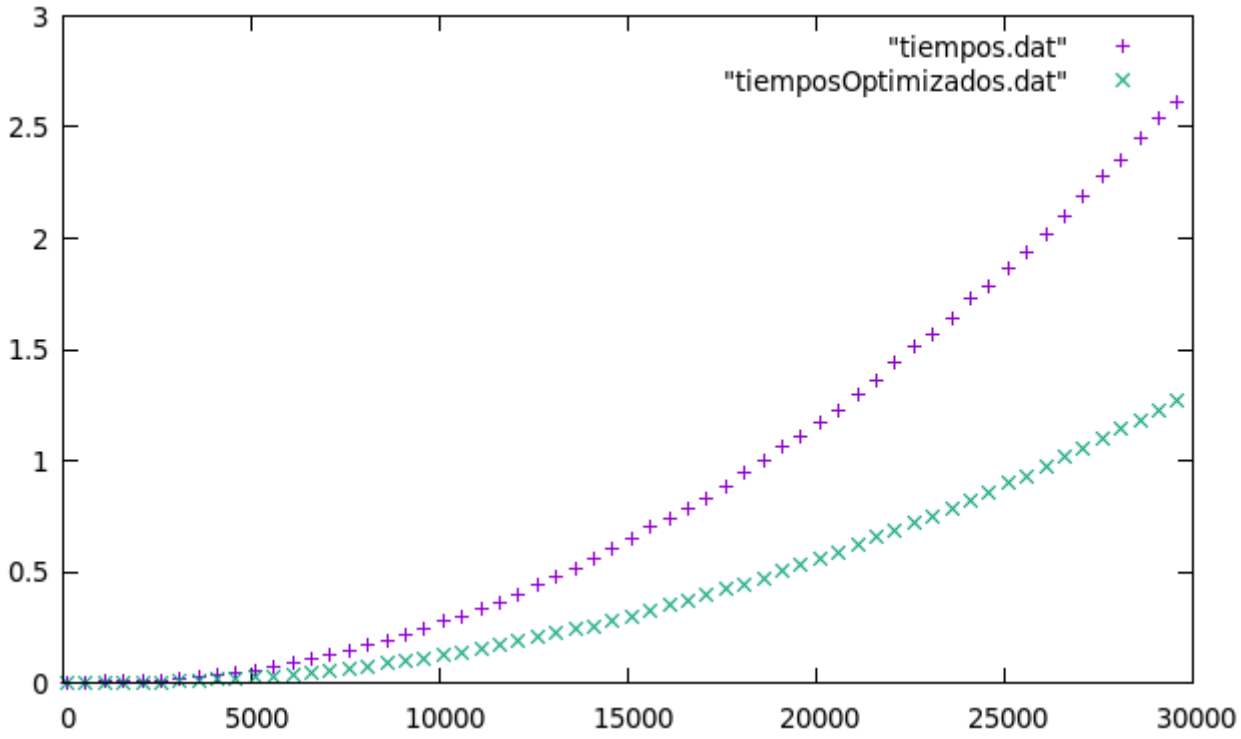
A pesar de añadir esta variable que registra tras cada pasada si el vector ha sido modificado o no (ordenado o no) la eficiencia teórica sigue siendo $O(n^2)$ debido al bucle for, sin embargo sin necesidad de ejecutar el algoritmo se aprecia que en practica el tiempo de ejecución será O(n). Esto se debe a que el vector esta ordenado y nunca hará falta ordenarlo, tan solo recorrerlo y comprobar que los elementos están ya ordenados.

La gráfica obtenida:



Se ajusta a la previsión ya que el tiempo necesario va aumentando poco a poco debido al aumento del tamaño del vector sin embargo la función no crece tanto como la función x^2 que se previó que seguiría.

Ejercicio 6: Influencia del proceso de compilación.

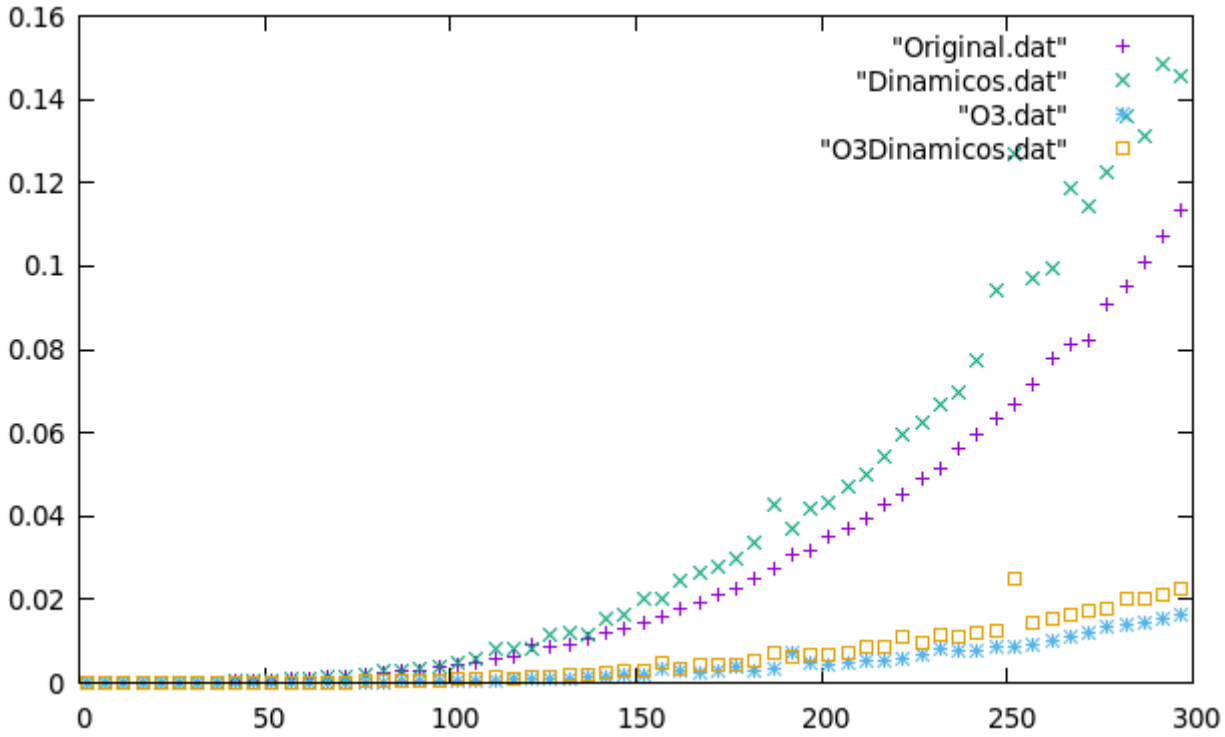


Es notable que la ejecución necesita menos tiempo si se utiliza una optimización al compilar el programa.

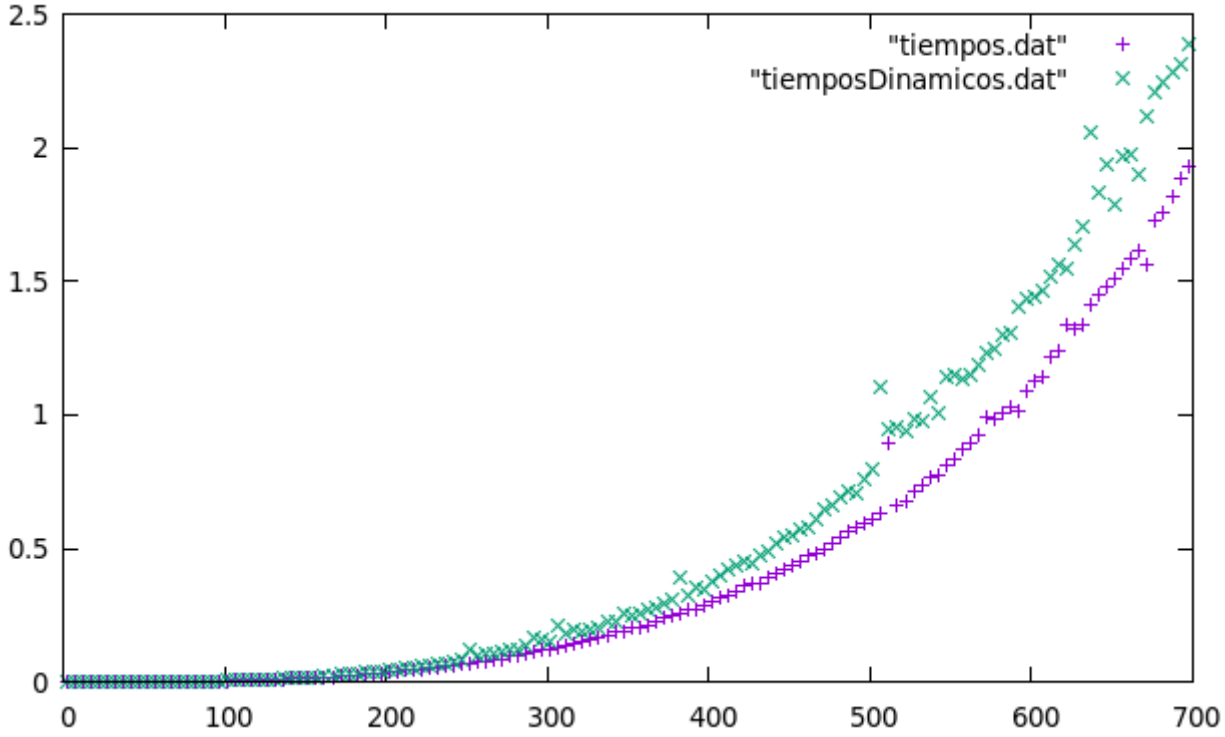
Ejercicio 7: Multiplicación matricial.

Eficiencia teórica: El algoritmo de multiplicación de dos matrices cuadradas implica el uso de tres bucles for anidados lo que hace que la eficiencia teórica del algoritmo sea $O(n^3)$.

Eficiencia empírica: Voy a comparar las gráficas resultantes de la ejecución del algoritmo implementado con vectores estáticos, dinámicos y utilizando o no la optimización O3 a la hora de compilar.

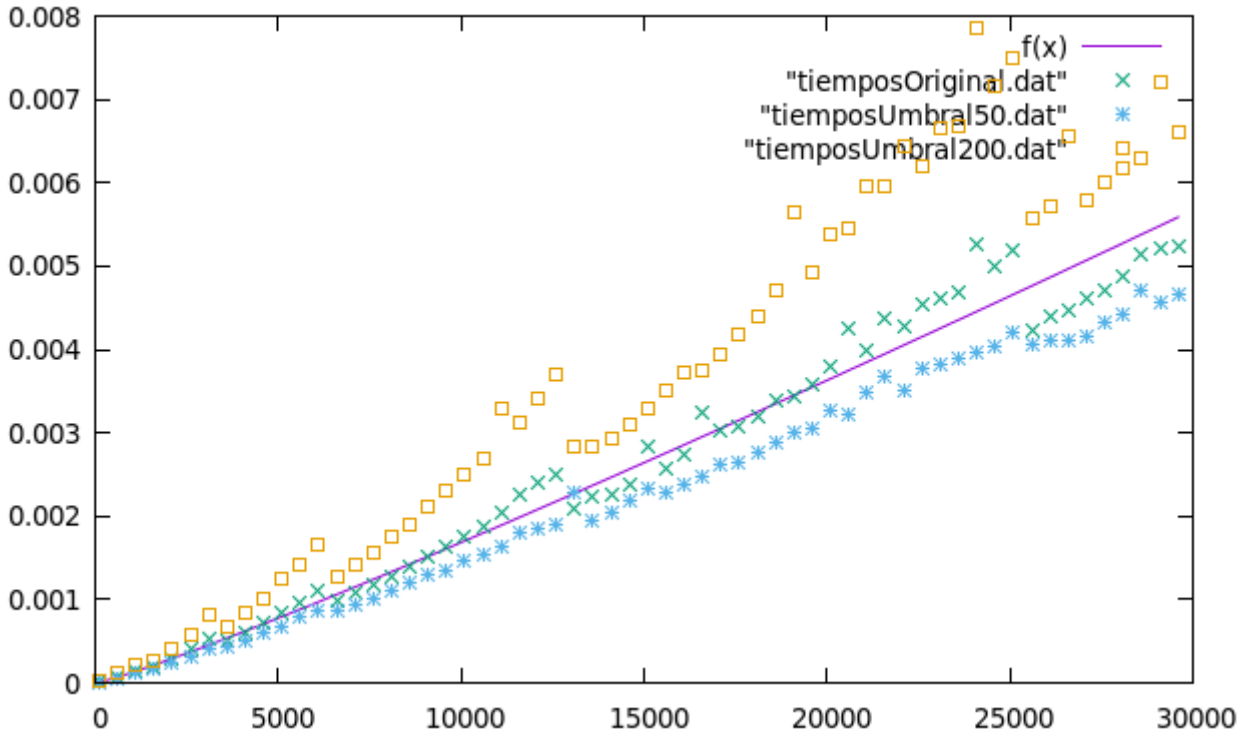


Al observar que usando vectores estáticos el algoritmo es mas rápido he decidido repetir la prueba pero observar el resultado si las matrices son mas grandes. Obteniendo el mismo resultado.



Ejercicio 8: Ordenación por mezcla.

Gráfica obtenida:



Se puede observar que con un umbral menor el tiempo de ejecución es mejor.