ESTRUCTURA DE DATOS

EFICIENCIA

Práctica realizada por: Juan Sánchez Rodríguez

Hardware: MSI GE60 2PE ApachePro, Intel(R) Core(TM) i7-4720HQ, 2,5 Ghz (procesador de 64 bits), 16 GB RAM.

Sistema Operativo: Windows 8.1, pero la práctica está realizada en una máquina virtual de VirtualBox con las siguientes especificaciones: Sistema operativo Ubuntu 16.04.1 usando una arquitectura de 64 bits, con una memoria RAM de 7704MB.

Compilador: gcc version 5.3.1 20160413.

Ejercicio 1:

Primero analizamos la eficiencia teórica:

```
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;
                        }
}
```

Linea 2: Hay una asignación, una comparación, una resta y un incremento.

Linea 3: Hay una asignación, una comparación, dos restas y un incremento.

Linea 4: Hay dos accesos a vector y una suma.

Linea 5: Hay una asignación y un acceso a vector.

Linea 6: Hay dos accesos a vector, una asignación y una suma.

Linea 7: Hay un acceso a vector, una suma y una asignación.

Con esto podemos ver que la eficiencia de el algoritmo es de $O(n^2)$.

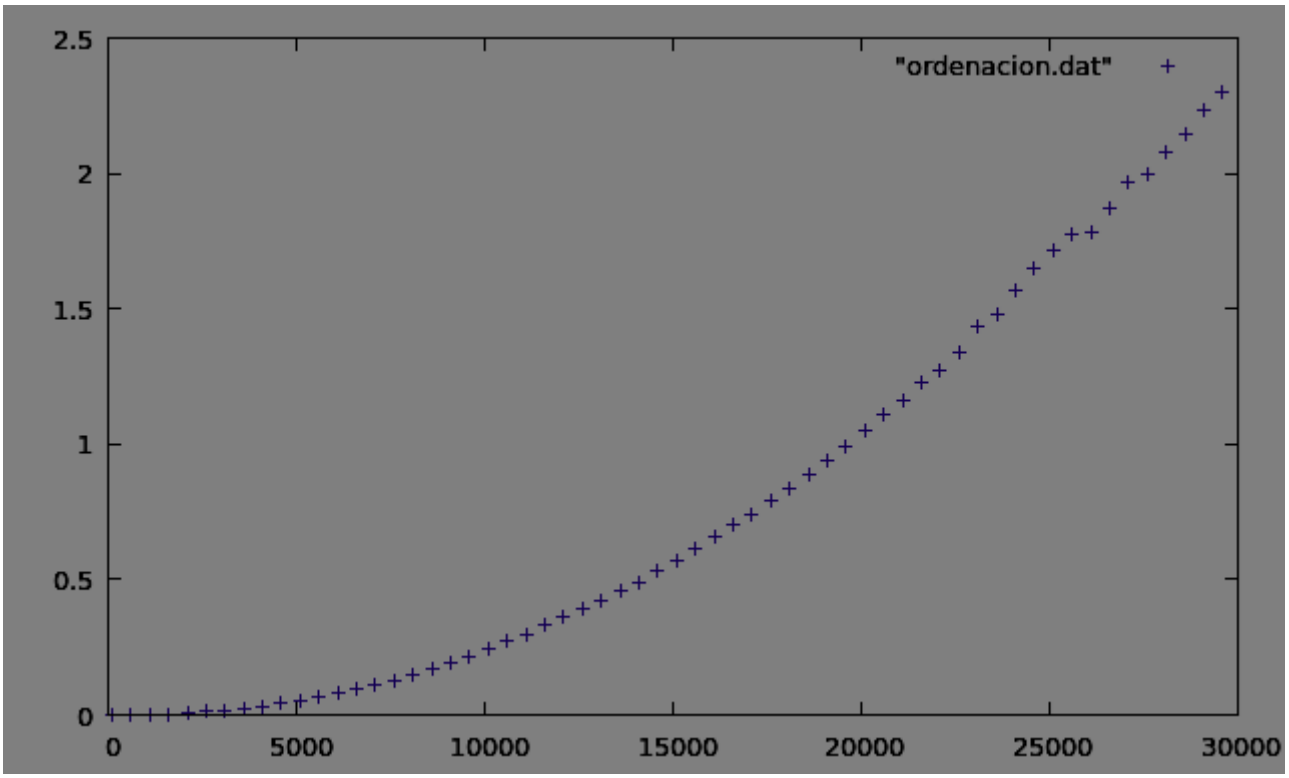
A continuación analizamos la eficiencia empírica usando el siguiente código:

```
#include <iostream>
```

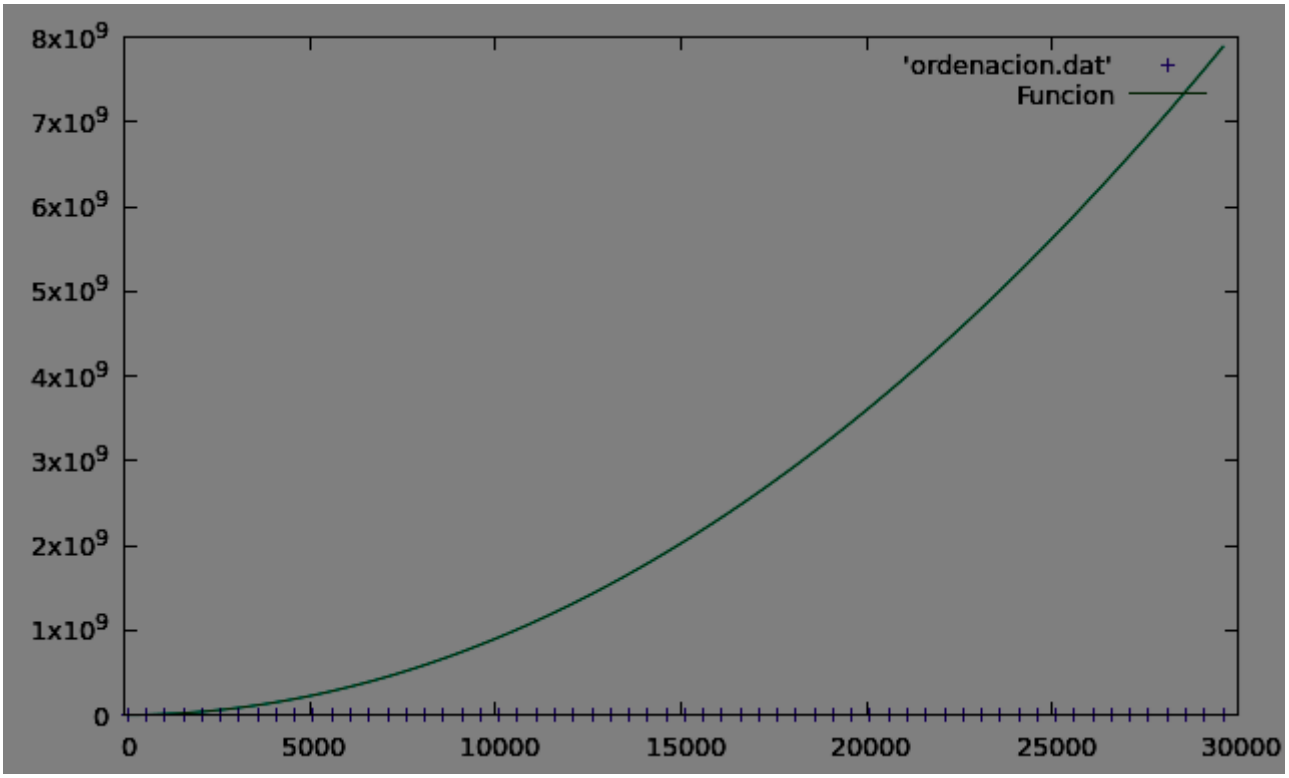
```
#include <ctime> // Recursos para medir tiempos
#include <cstdlib> // Para generación de números pseudoaleatorios

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 << "Genera un vector de TAM números aleatorios en [0,VMAX[" << endl;
	exit(EXIT_FAILURE);
}
int main(int argc, char * argv[]) {
	if (argc!=3)     // Lectura de parámetros
		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 generador números pseudoaleatorios
```

```
        for (int i=0; i<tam; i++) // Recorrer vector
                v[i] = rand() % vmax;// Generar aleatorio [0,vmax[
        clock_t tini;
        tini=clock();// Anotamos el tiempo de inicio
        int x = vmax+1;
        ordenar(v,tam);
        clock_t tfin;
        tfin=clock();// Anotamos el tiempo de finalización
        // Mostramos resultados (Tamaño del vector y tiempo de ejecución en seg.)
        cout << tam << "\t" << (tfin-tini)/(double)CLOCKS_PER_SEC << endl;


        delete [] v; // Liberamos memoria dinámica
}
```

Usando la herramienta gnuplot obtenemos la gráfica de la eficiencia empírica:



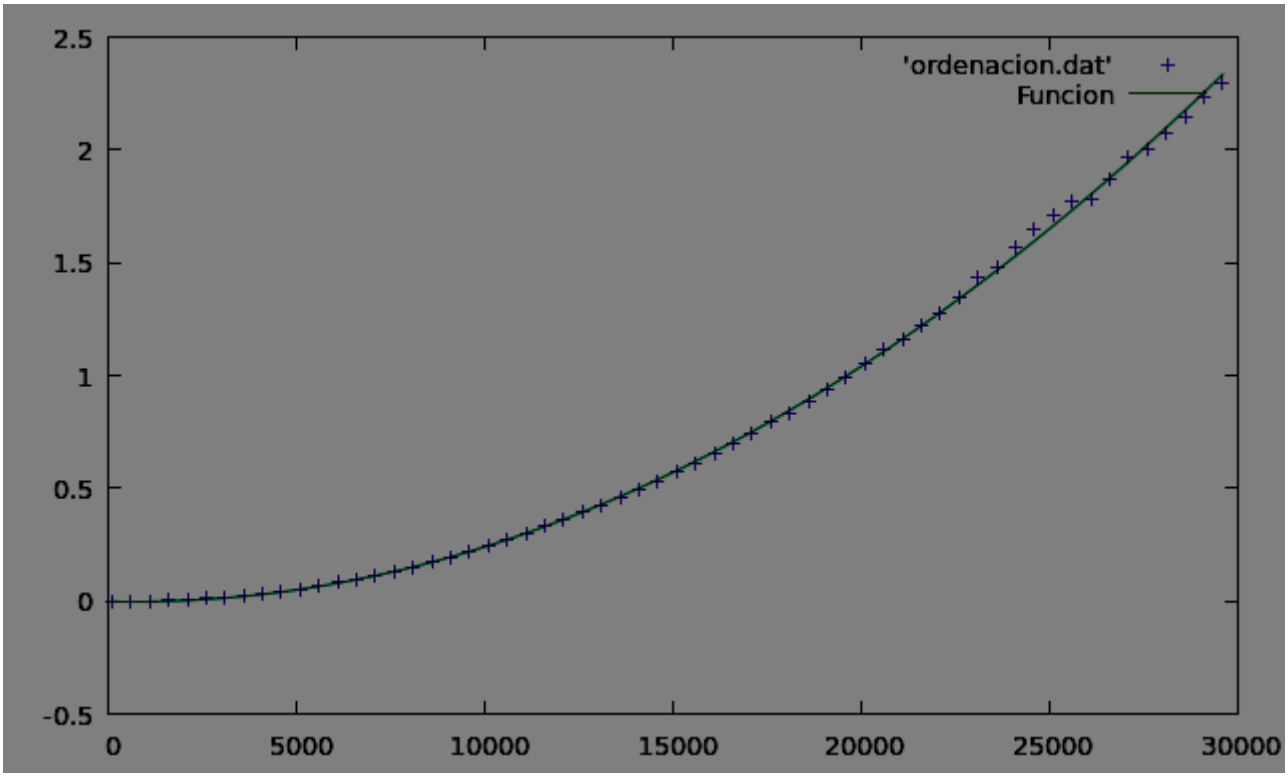
Usando la herramienta gnuplot obtenemos la eficiencia teórica y la empírica superpuestas:



Podemos observar que la curva de la eficiencia teórica es muy similar a la empírica, en esta superposición no podemos ver esto superpuesto ya que las constantes no coinciden, es decir, no están ajustadas.

Ejercicio 2:

Usando la herramienta gnuplot obtenemos la eficiencia teórica y la empírica superpuestas, pero esta vez ajustadas:



Al estar ajustada podemos ver como ambas curvas coinciden.

Ejercicio 3:

-Explicación del algoritmo de ejercicio_desc.cpp:

Es un algoritmo de búsqueda, hay un vector v de tamaño n y busca un entero x, además se pasa donde empieza el vector (inf) y donde acaba (sup). El proceso de búsqueda es el siguiente:

Se crea un bucle while donde se sale en el caso de que inf >sup o que se encuentre el número,

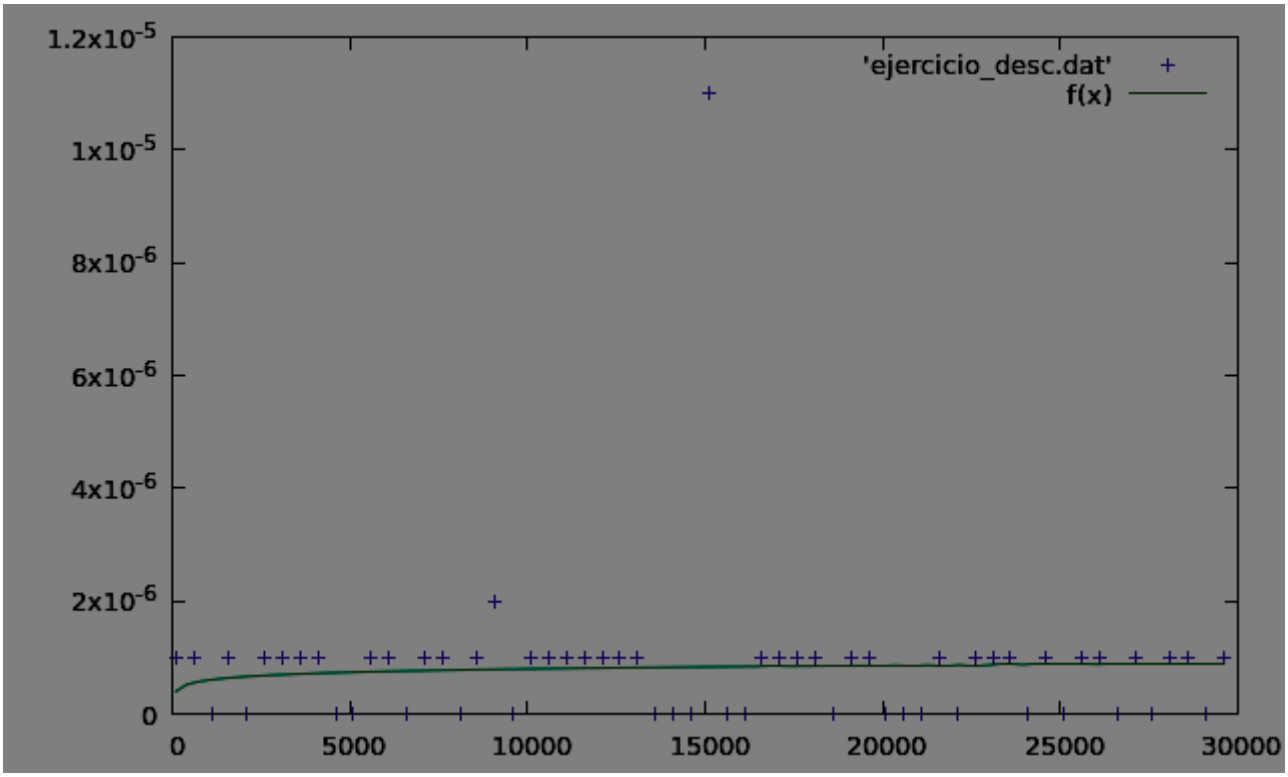
a continuación se establece la posición med =(inf+sup)/2 y comprueba si x está en la posición med del vector, en el caso de que sea así sale del vector, en caso contrario comprueba si el número en la posición med es menor que x, en el caso de que lo sea hace inf=med+1, si no cumple nada de lo anterior hace sup=med-1. Una vez que sale del bucle comprueba si se ha encontrado x en el vector, en caso afirmativo devuelve la posición, en caso contrario devuelve -1.

-Eficiencia teórica:

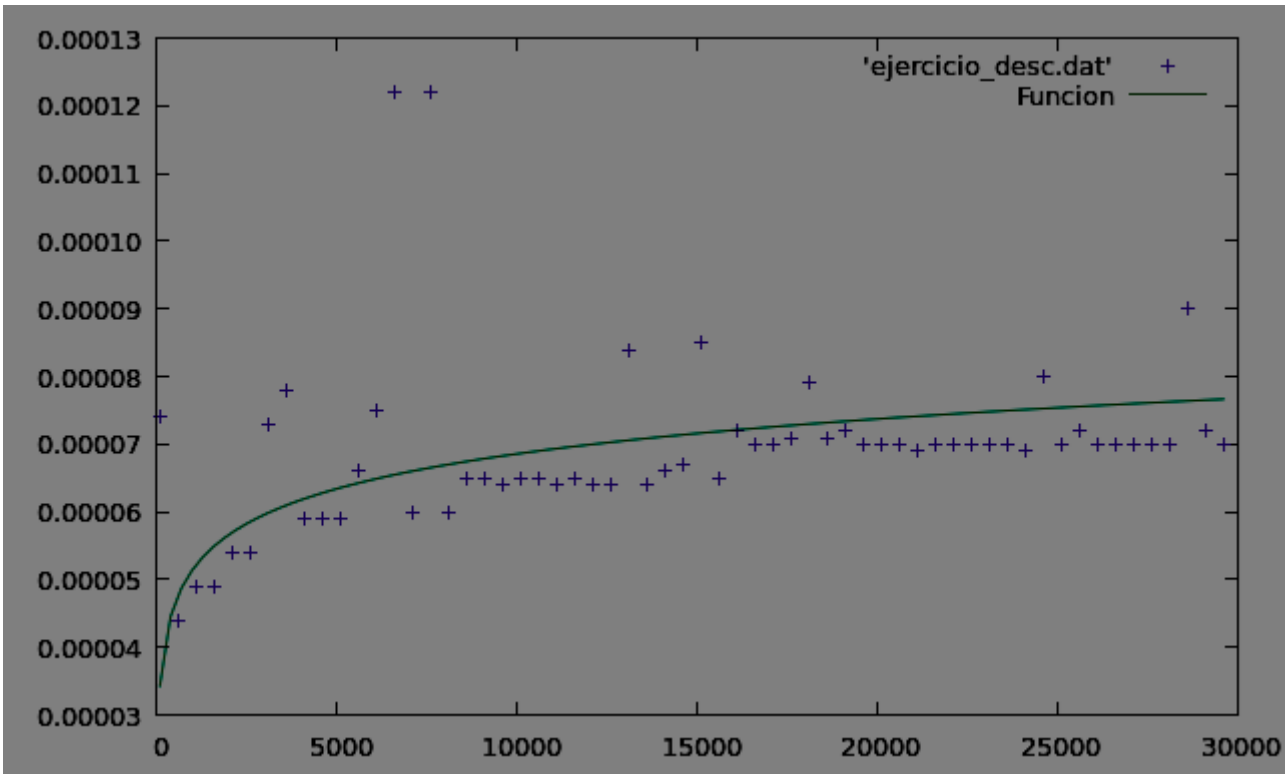
El algoritmo divide en 2 un vector de tamaño n repitiendo el proceso hasta que encuentre x o hasta que no se puedan hacer más divisiones, el número máximo de veces en el que se puede dividir por la mitad el vector es log2(n). Con esto sacamos que la eficiencia teórica es O(log(n)).

-Eficiencia empírica:

Usando la herramienta gnuplot obtenemos la eficiencia teórica y la empírica superpuestas:



La gráfica es así de horizontal ya que el algoritmo es tan rápido que es imposible apreciar cambios en el tiempo de ejecución por ello vamos a ejecutar 1000 veces cada tamaño:



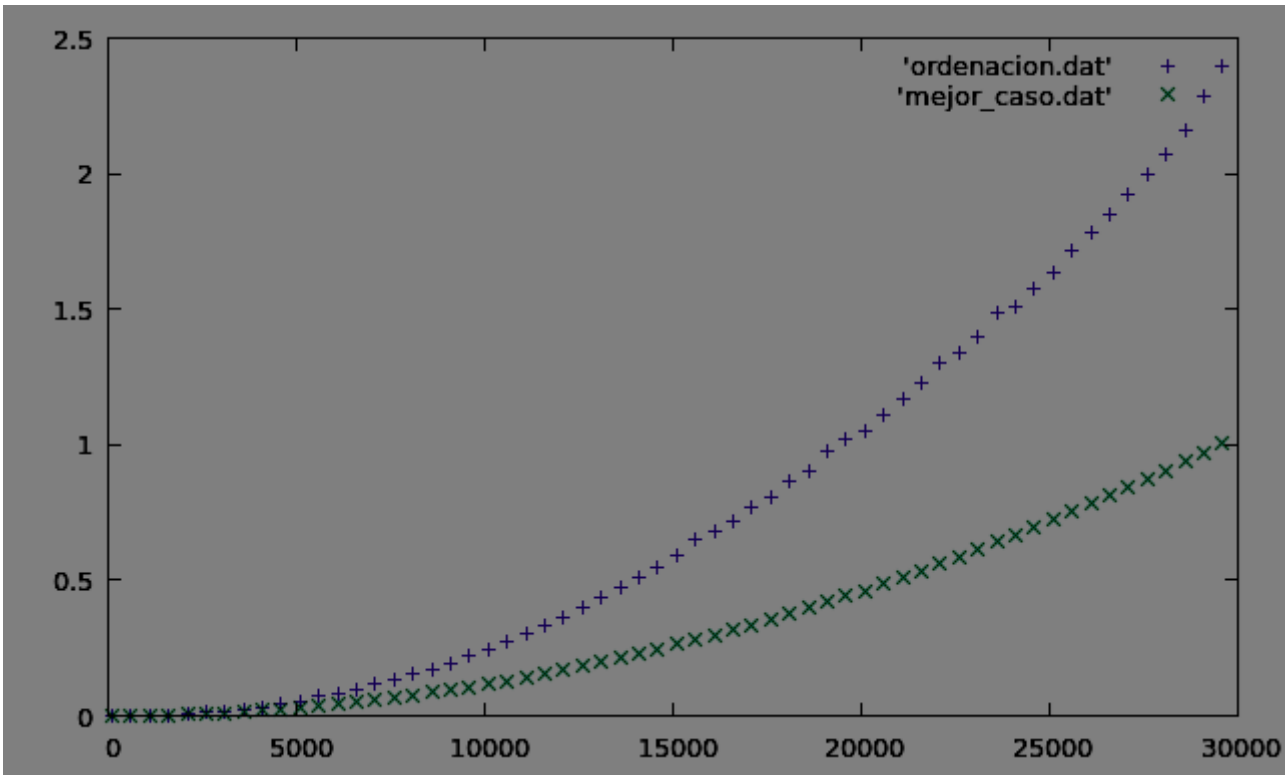
Ejercicio 4:

-Mejor Caso: para este caso vamos a hacer un vector ordenado usando el siguiente código:

v[0]=0;

```
for (int i=1; i<tam; i++) // Recorrer vector
    v[i]=v[i-1]+1;
```

La gráfica superpuesta con la del algoritmo burbuja es la siguiente:

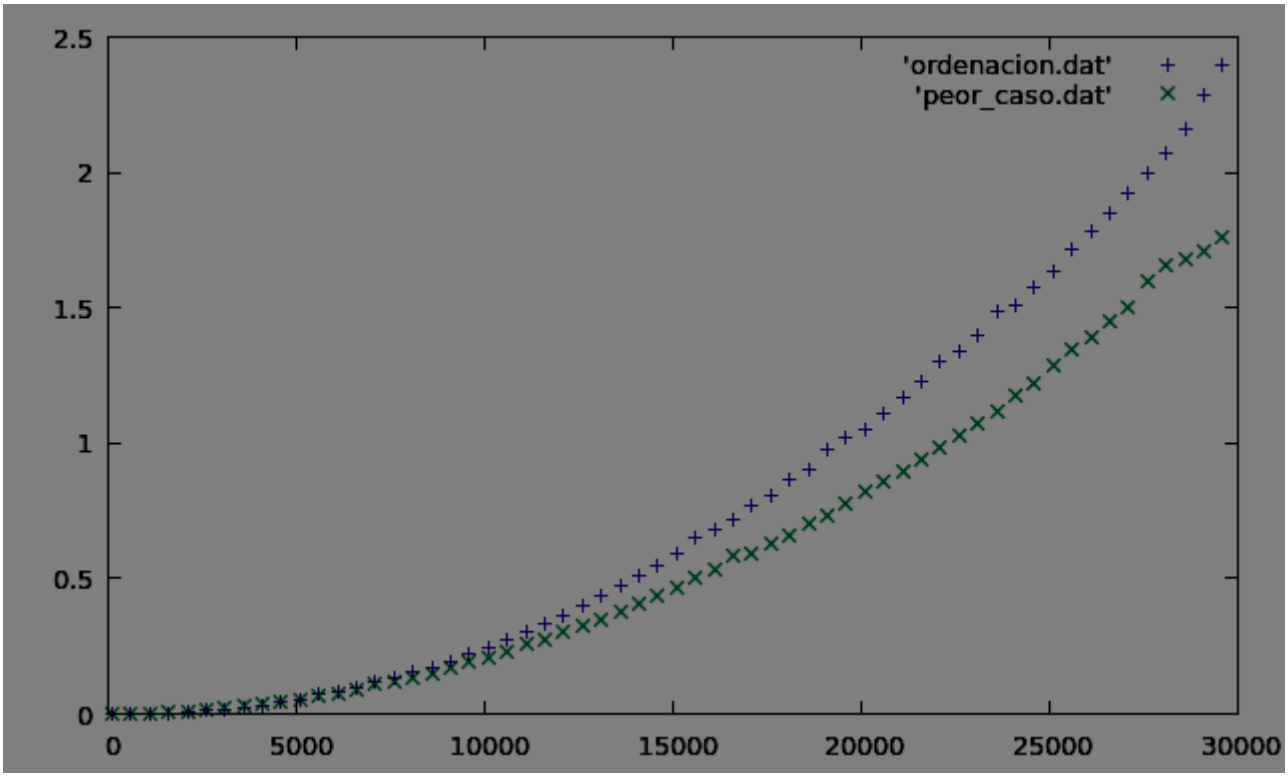


Podemos ver que el tiempo de ejecución del mejor caso es mucho menor al generado por números aleatorios.

-Peor Caso: para este caso vamos a hacer un vector ordenado inversamente usando el siguiente código:

```
v[0]=vmax;
    for (int i=1; i<tam; i++) // Recorrer vector
        v[i]=v[i-1]-1;
```

La gráfica superpuesta con la del algoritmo burbuja es la siguiente:

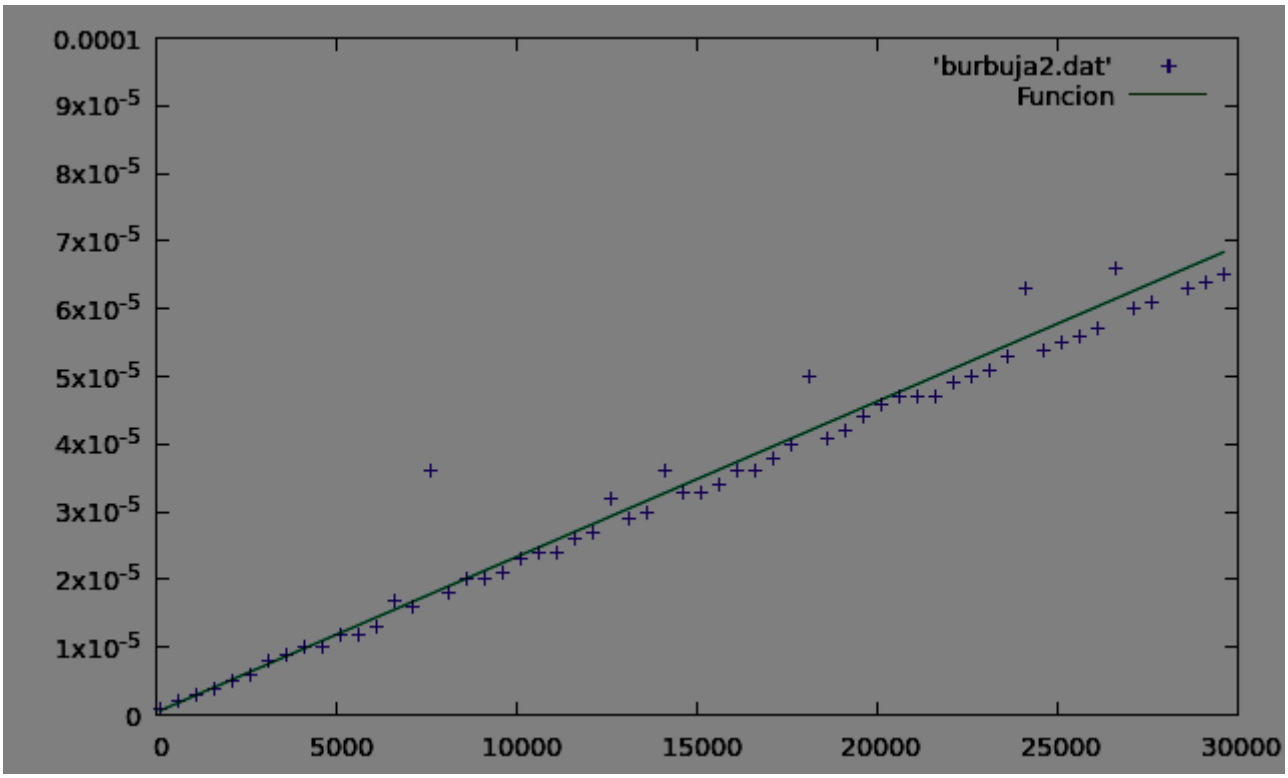


Curiosamente observamos que el tiempo de ejecución del peor caso es menor al generado por números aleatorios, esto se debe a que cuando hay saltos condicionales el procesador sufre retardos, por esto los procesadores poseen un predictor de saltos con la idea de analizar los saltos anteriores para predecir los siguientes como en este caso el vector está completamente ordenado se puede predecir cual será el siguiente salto con mucha facilidad, por ello el peor caso es más rápido que el de números aleatorios.

Ejercicio 5:

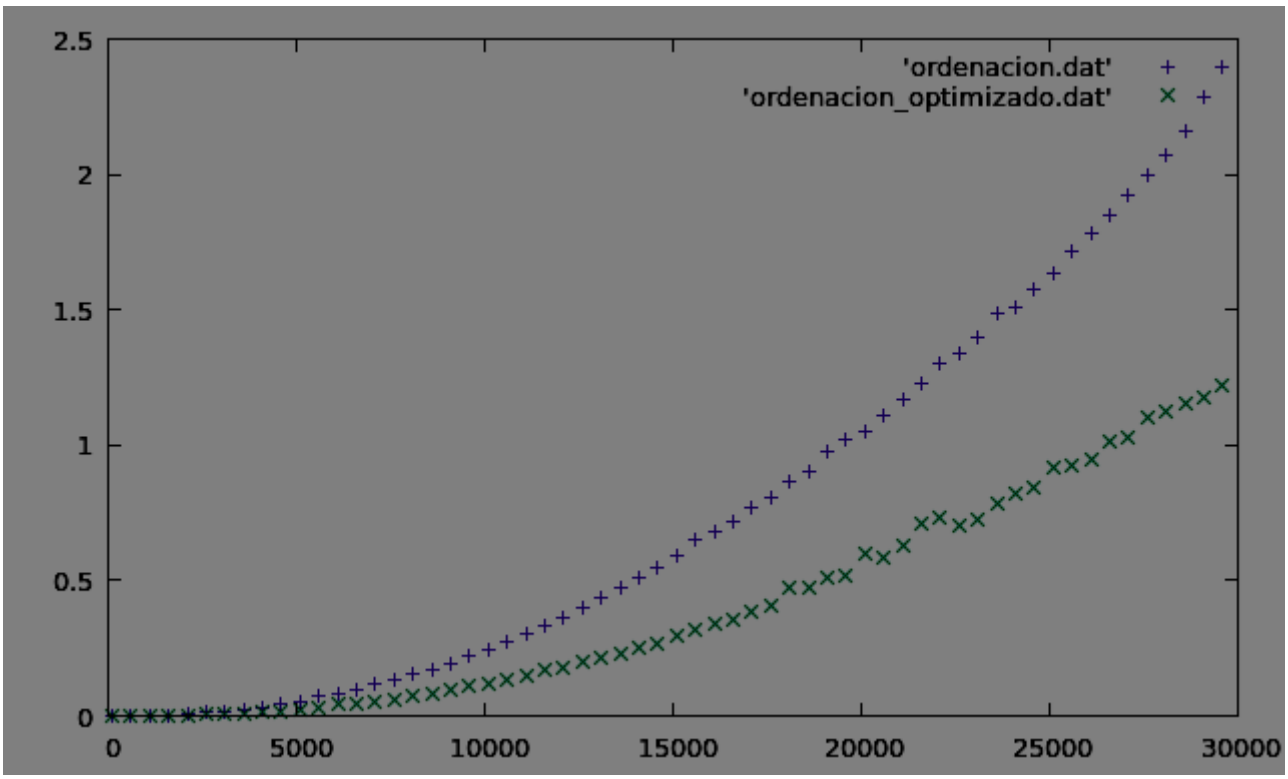
En esta versión del algoritmo burbuja hay una variable que permite saber si en una de las iteraciones del bucle externo no se ha modificado el vector, si esto ocurre significa que el vector

está ordenado y que no hay que continuar, considerando el mejor caso vemos que la eficiencia es lineal, O(n).



Ejercicio 6:

Usando la herramienta gnuplot obtenemos las eficiencias superpuestas:



Como podemos observar el programa optimizado es mucho más rápido.