

# **INFORMESEMIGRUPO1.pdf**



danielsp10



Algorítmica



2º Grado en Ingeniería Informática



Escuela Técnica Superior de Ingenierías Informática y de  
Telecomunicación  
Universidad de Granada



**WUOLAH Print**

Lo que faltaba en Wuolah



**Imprimir**



- Todos los apuntes que necesitas están aquí
- Al mejor precio del mercado, desde **2 cent.**
- Recoge los apuntes en tu copistería más cercana o recíbelos en tu casa
- Todas las anteriores son correctas

Algorítmica – SEMIGRUPO

P1: EFICIENCIA

Daniel | Jose | Martin

 Imprimir

# PRÁCTICA 1:

## EFICIENCIA DE ALGORITMOS

Autor: SEMIGRUPO

Jose Abela Cánovas  
Martín Anaya Quesada  
Daniel Pérez Ruiz

## ÍNDICE DE CONTENIDOS DEL INFORME

<b>0. INTRODUCCIÓN.....</b>	4
<b>1. ESPECIFICACIONES TÉCNICAS DE ORDENADORES UTILIZADOS .....</b>	5
<b>2. PROCEDIMIENTO PARA LA OBTENCIÓN DE LA EFICIENCIA EMPÍRICA.....</b>	6
<b>2.1 ESTRUCTURA DE FICHEROS UTILIZADA.....</b>	6
<b>2.2 ESTRUCTURA DE CÓDIGOS FUENTE .....</b>	7
<b>2.3 SCRIPTS C-SHELL UTILIZADOS PARA LA OBTENCIÓN DE LOS TIEMPOS.....</b>	8
<b>2.4 SCRIPTS GNUPLOT PARA LA OBTENCIÓN DE GRÁFICAS .....</b>	10
<b>2.5 OBTENCIÓN DE AJUSTE Y TABLAS DE TIEMPOS.....</b>	11
<b>3. BLOQUE 1: ALGORITMOS DE ORDENACIÓN .....</b>	12
<b>3.1 ALGORITMOS DE EFICIENCIA <math>O(n^2)</math> :: ALGORITMO DE LA BURBUJA.....</b>	12
➤ <i>Descripción .....</i>	12
➤ <i>Expresión del algoritmo.....</i>	12
➤ <i>Eficiencia Teórica.....</i>	12
➤ <i>Eficiencia Empírica .....</i>	12
➤ <i>Análisis gráfico.....</i>	14
➤ <i>Comparación Final.....</i>	16
<b>3.2 ALGORITMOS DE EFICIENCIA <math>O(n^2)</math> :: ALGORITMO DE INSERCIÓN .....</b>	17
➤ <i>Descripción .....</i>	17
➤ <i>Expresión del algoritmo.....</i>	17
➤ <i>Eficiencia Teórica.....</i>	17
➤ <i>Eficiencia Empírica .....</i>	17
➤ <i>Análisis gráfico.....</i>	19
➤ <i>Comparación Final.....</i>	21
<b>3.3 ALGORITMOS DE EFICIENCIA <math>O(n^2)</math> :: ALGORITMO DE SELECCIÓN .....</b>	22
➤ <i>Descripción .....</i>	22
➤ <i>Expresión del algoritmo.....</i>	22
➤ <i>Eficiencia Teórica.....</i>	22
➤ <i>Eficiencia Empírica .....</i>	22
➤ <i>Análisis gráfico.....</i>	24
➤ <i>Comparación Final.....</i>	26
<b>3.5 ALGORITMOS DE EFICIENCIA <math>O(n \cdot \log(n))</math> :: ALGORITMO MERGESORT .....</b>	27
➤ <i>Descripción .....</i>	27
➤ <i>Expresión del algoritmo.....</i>	27
➤ <i>Eficiencia Empírica .....</i>	28
➤ <i>Análisis gráfico.....</i>	29

NEW

# WUOLAH Print

Lo que faltaba en Wuolah



Imprimir



- Todos los apuntes que necesitas están aquí
- Al mejor precio del mercado, desde **2 cent.**
- Recoge los apuntes en tu copistería más cercana o recíbelos en tu casa
- Todas las anteriores son correctas



➤ <i>Comparación Final</i> .....	31
➤ <i>Comprobación teórica del Umbral de Mergesort</i> .....	32
<b>3.6 ALGORITMOS DE EFICIENCIA O(n·log(n)) :: ALGORITMO QUICKSORT</b> .....	33
➤ <i>Descripción</i> .....	33
➤ <i>Expresión del algoritmo</i> .....	33
➤ <i>Eficiencia Empírica</i> .....	34
➤ <i>Análisis gráfico</i> .....	35
➤ <i>Comparación Final</i> .....	37
<b>3.7 ALGORITMOS DE EFICIENCIA O(n·log(n)) :: ALGORITMO HEAPSORT</b> .....	38
➤ <i>Descripción</i> .....	38
➤ <i>Expresión del Algoritmo</i> .....	38
➤ <i>Eficiencia Empírica</i> .....	39
➤ <i>Análisis gráfico</i> .....	40
➤ <i>Comparación Final</i> .....	42
<b>3.9 COMPARACIÓN FINAL DE ALGORITMOS DE ORDENACIÓN</b> .....	43
<b>4. BLOQUE 2: ALGORITMO DE FLOYD</b> .....	44
➤ <i>Descripción</i> .....	44
➤ <i>Expresión del algoritmo</i> .....	44
➤ <i>Eficiencia teórica</i> .....	44
➤ <i>Eficiencia Empírica</i> .....	44
➤ <i>Análisis gráfico</i> .....	45
➤ <i>Comparación Final</i> .....	49
<b>5. BLOQUE 3: ALGORITMO DE HANOI</b> .....	50
➤ <i>Descripción</i> .....	50
➤ <i>Expresión del Algoritmo</i> .....	50
➤ <i>Eficiencia Teórica</i> .....	50
➤ <i>Eficiencia Empírica</i> .....	50
➤ <i>Análisis gráfico</i> .....	51
➤ <i>Comparación Final</i> .....	55
<b>6. COMPARACIÓN DE NIVELES DE OPTIMIZACIÓN</b> .....	56
<b>7. CONCLUSIONES FINALES</b> .....	57

## 0. INTRODUCCIÓN

- En este informe se presenta el estudio completo de la eficiencia de una serie de algoritmos que resuelven un determinado problema.
- Se proponen tres bloques fundamentales de algoritmos: el primero comprende a todos los algoritmos de ordenación de un vector aleatorio de dimensión N; el segundo, conocido como algoritmo de Floyd, que calcula el costo del camino mínimo entre cada par de nodos de un grafo dirigido; y por último el algoritmo que resuelve el tan famoso problema de las torres de Hanoi.
- Uno de los problemas a los que se enfrentan los informáticos, y más concretamente los programadores, es la obtención de algoritmos que sean capaces de resolver en un tiempo razonable un determinado problema, para tamaños y dimensiones totalmente arbitrarios y de razón suficientemente grande. Como comprobaremos cuando estudiemos el primer bloque de los anteriores descritos, existen algoritmos que resuelven el mismo problema mejor que otros (siendo ejecutados sobre la misma máquina), así que es importante saber escoger cuándo y cuál es el que elegiremos para nuestras condiciones concretas.
- En base a esto, todos estos códigos que vamos a analizar se han ejecutado sobre el mismo ordenador, con todos los recursos disponibles para el proceso creado por el Sistema Operativo para el programa, sin interferencias de otros programas abiertos u otras ejecuciones que pudiesen afectar en el rendimiento y estudio de la eficiencia empírica. Es por ello por lo que a continuación muestro las especificaciones técnicas de nuestros PC.

- Todos los apuntes que necesitas están aquí
- Al mejor precio del mercado, desde **2 cent.**
- Recoge los apuntes en tu copistería más cercana o recíbelos en tu casa
- Todas las anteriores son correctas

Algorítmica – SEMIGRUPO

P1: EFICIENCIA

Daniel | Jose | Martin

## 1. ESPECIFICACIONES TÉCNICAS DE ORDENADORES UTILIZADOS

Todo lo necesario para realizar esta práctica se ha realizado sobre los siguientes ordenadores:

Imprimir 

### ORDENADOR DANIEL

- **MODELO:** *Lenovo ideapad 330*
- **CPU:** Intel Core™ i7-7500U 7th Generation ~ @2.70 GHz up to @2.90 GHz
- **RAM:** 16 GB de tipo DDR4
- **MEMORIA AUXILIAR:** Disco duro HDD 1TB
- **TARJETA GRÁFICA:** AMD Radeon™ 530 ~ 2 GB VRAM
- **SISTEMA OPERATIVO:** Linux Mint 19.3 Tara de 64 bits

### ORDENADOR MARTÍN

- **CPU:** Intel Core™ i5 8th Generation ~ @1.66 GHz up to @3.40 GHz
- **RAM:** 8 GB de tipo DDR4
- **MEMORIA AUXILIAR:** 128 GB SSD + 1000HDD
- **TARJETA GRÁFICA:** Intel Graphics HD
- **SISTEMA OPERATIVO:** Ubuntu 18.04 de 64 bits

### ORDENADOR JOSE

- **CPU:** Intel Core™ i7-7500U 8th Generation
- **RAM:** 8 GB de tipo DDR4
- **MEMORIA AUXILIAR:** Disco duro sólido SSD 256 GB
- **TARJETA GRÁFICA:** Intel Graphics HD
- **SISTEMA OPERATIVO:** Ubuntu 18.04 de 64 bits

A continuación, procedemos a especificar el software utilizado:

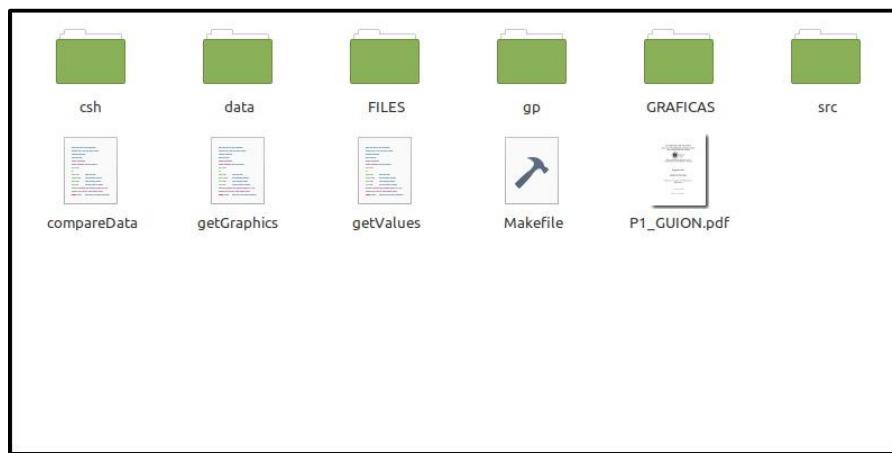
- **EDITOR DE CODIGO:** Atom x64 bits
- **LENGUAJE DE PROGRAMACIÓN DE ALGORITMOS:** C++ (std 11) [Compilación con g++]
- **GESTIÓN DE CÓDIGOS:** Hemos generado una estructura con Makefile para la compilación de todos los programas atendiendo a las necesidades del estudio. Así como la disponibilidad de macros en C-Shell para la obtención de los tiempos.
- **GENERADOR DE GRÁFICAS:** Gnuplot
- **REDACCIÓN DE INFORME Y PRESENTACIÓN:** Microsoft Office 2019 x64 bits (Windows)

## 2. PROCEDIMIENTO PARA LA OBTENCIÓN DE LA EFICIENCIA EMPÍRICA

A continuación, se muestra detalladamente cómo se han obtenido los datos, así como las gráficas. Todo lo que se observe en los siguientes apartados será en base a lo especificado en esta sección del documento.

### 2.1 ESTRUCTURA DE FICHEROS UTILIZADA

- Para una mayor sincronización de los ficheros que se han generado, hemos creado un sistema de directorios como el indicado en la siguiente captura.



- *Csh*: Contiene los scripts de c-shell que se encargan de obtener los tiempos de ejecución de cada algoritmo.
- *Data*: Almacena los datos de cada integrante del grupo, generados por los scripts.
- *Gp*: Scripts de gnuplot que automatizan la obtención de las gráficas
- *GRAFICAS*: Almacena las gráficas de cada integrante del grupo, generados por los scripts de gnuplot
- *Src*: Almacena los códigos fuente de los algoritmos

En esta estructura se cuenta con scripts que ejecutan todos los códigos y genera todas las gráficas de una vez, así como la disposición de un makefile que compila los programas atendiendo a la necesidad de cada momento.

## 2.2 ESTRUCTURA DE CÓDIGOS FUENTE

- Hemos modificado los ficheros que se nos proporcionaba para encapsular aún más el código y diferenciar entre qué es lo que se iba a medir y cuál va a ser la estructura principal del programa.
- Todos los códigos poseen el mismo *main*, tal y como se indica en la siguiente imagen:

```
/**
 * @brief MAIN FUNCTION
 */
int main(int narg, char * argv[]) {
    //COMPRACION DE ARGUMENTOS
    if(narg != 2) {
        cerr << "USO: ./" << argv[0] << " <num_elem>" << endl;
        exit(-1);
    }

    //CREACION DE VECTOR
    int n = atoi(argv[1]);
    int * T = new int[n];
    assert(T);

    // !!!TEST PRINCIPAL!!!
    double test = runTest(T,n,NUM_EJEC);

    //OPTIONAL: MOSTRAR VECTOR
    //mostrarVector(T,n);

    //SALIDA DE RESULTADOS
    cout << n << " " << test << endl;

    //LIBERACION DE VECTOR
    delete [] T;

    return 0;
}
```

- En la función *runTest(int T[], int num\_elem, int NUM\_EJEC)* se calcula el tiempo de ejecución del algoritmo. Para ello se ejecuta una serie de veces el algoritmo (parámetro *NUM\_EJEC*), y se hace la media de todas las mediciones. Por supuesto, en cada ejecución se tiene un vector de la misma dimensión pero de diferentes datos. A continuación se muestra un ejemplo de *runTest*.

```

static double runTest(int T[], int n, int NUM_EJEC) {
    double resultado=0, transcurrido=0;
    //CONTROLADORES DEL TIEMPO
    clock_t tantes, tdespues;

    //INICIALIZACION SEMILLA PARA NUMEROS ALEATORIOS
    srand(time(0));

    for(int z=0; z<NUM_EJEC; z++) {
        //VECTOR RANDOM
        for(int i=0; i<n; i++) {
            T[i] = random();
        }

        //MEDICION DEL TIEMPO
        tantes = clock();
        burbuja(T,n);
        tdespues = clock();

        transcurrido += (double) (tdespues - tantes);
    }

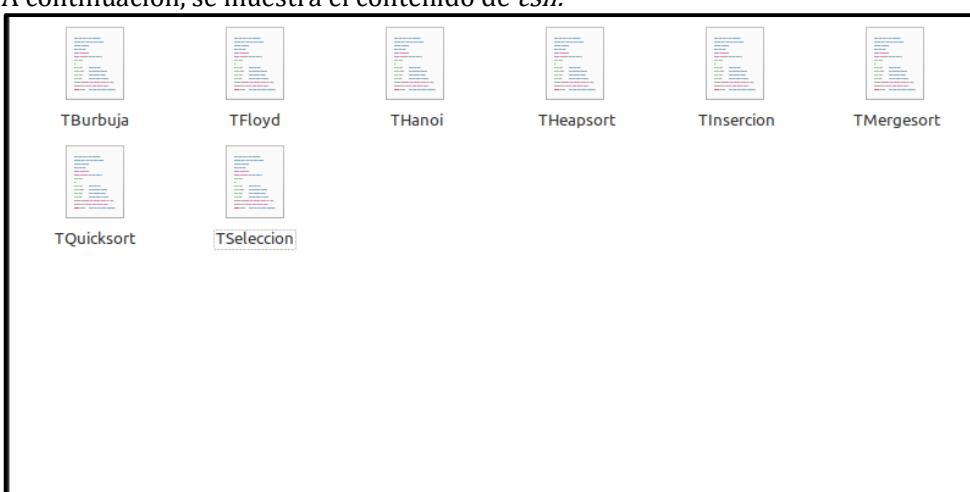
    //CALCULO FINAL DE RESULTADO
    resultado = transcurrido / (CLOCKS_PER_SEC * (double) NUM_EJEC);

    return resultado;
}

```

### 2.3 SCRIPTS C-SHELL UTILIZADOS PARA LA OBTENCIÓN DE LOS TIEMPOS

- Hemos utilizado una serie de scripts para obtener los tiempos de ejecución en un fichero de datos de extensión .dat. Cada tipo de algoritmo tiene una serie de ejecuciones predeterminadas que se han elegido cuidadosamente para que no se produzcan demoras con dimensiones demasiado grandes y sobre todo, para que a la hora de dibujarlas se vea lo mejor posible, es por eso por lo que los siguientes intervalos son:
  - Algoritmos de Ordenación: 2000-5200 [incrementos de 2000]
  - Algoritmo Floyd: 50-1350 [incrementos de 50]
  - Algoritmo Hanoi: 2-29 [incrementos de 1]
- A continuación, se muestra el contenido de *csh*:



- Todos los apuntes que necesitas están aquí
- Al mejor precio del mercado, desde **2 cent**.
- Recoge los apuntes en tu copistería más cercana o recíbelos en tu casa
- Todas las anteriores son correctas

Algorítmica – SEMIGRUPO

P1: EFICIENCIA

Daniel | Jose | Martin

- Y aquí se muestra un ejemplo de una parte del script csh:

```
#!/bin/csh
#Author: SEMIGRUPO
#Brief: Obtiene los tiempos de ejecucion para algoritmo burbuja
#Uso: ./TBurbuja <NIVELES_FLAG>

=====
# PARAMETROS DE SCRIPT
=====
set FILE_FOLDER = "bin"
set USER_FOLDER = `whoami`
set DATA = "data"
set TARGET = "burbuja"
set OPTIMIZADO = "_OPTIMIZADO"

set EXTENSION = ".dat"
set FILEDATA = "TBURBUJA_`whoami`"
set FILEDATA_OPTIMIZADO = "TBURBUJA_OPTIMIZADO_`whoami`"

set NIVEL0 = "_0NIVEL"
set NIVEL1 = "_1NIVEL"
set NIVEL2 = "_2NIVEL"
set NIVEL3 = "_3NIVEL"
set NIVELES = "_sNIVEL"

set inicio = 2000
set fin = 52000
set inc = 2000

=====
# CREACION DE FICHEROS
=====

make BURBUJA

if ($1 == "NIVELES") then
    make BURBUJA_01 BURBUJA_02 BURBUJA_03 BURBUJA_05
else
    make BURBUJA_OPTIMIZADO
endif

=====
# OBTENCION DE DATOS CON CODIGO SIN OPTIMIZAR
=====

mkdir -p $DATA/$USER_FOLDER

echo "" >> $DATA/$USER_FOLDER/$FILEDATA$EXTENSION
@ i=$inicio
while ($i < $fin)
echo "EJECUCION DE [$TARGET] PARA DIMENSION [$i] :: RESTANTE [$i < $fin]"
./$FILE_FOLDER/$TARGET $i >> $DATA/$USER_FOLDER/$FILEDATA$EXTENSION
@ i+=$inc
end
echo ">>> FICHERO [$FILEDATA$EXTENSION] CREADO EN [$DATA/$USER_FOLDER]"
```

Imprimir 

- Estos scripts se ejecutan de manera secuencial con el script en Bash llamado *getValues.sh*. Es importante recalcar que **EN LOS 3 ORDENADORES EN LOS QUE SE HAN CALCULADO LOS TIEMPOS, EL SCRIPT SE HA EJECUTADO CON TODO EL ESPACIO DE MEMORIA DISPONIBLE EN ESE MOMENTO, SIN TAREAS EN SEGUNDO PLANO, NI NINGÚN OTRO PROGRAMA EXTERNO EJECUTÁNDOSE A LA VEZ.**
- Una vez terminado el trabajo del script, se almacena en la carpeta *data/`whoami`* los tiempos en ficheros de nombre “**T[ALGORITMO]\_[OPTIMIZADO]\_`whoami`.dat**”.

#### **2.4 SCRIPTS GNUPLOT PARA LA OBTENCIÓN DE GRÁFICAS**

- Al igual que hemos automatizado el proceso de obtención de los tiempos de ejecución para cada algoritmo, se ha realizado lo mismo para las gráficas. Gnuplot permite realizar scripts que se ejecutan como un trabajo cualquiera. Cuando se finaliza, devuelve el control al sistema operativo.
- A continuación, un ejemplo de script:

```
#SCRIPT PARA OBTENCION DE TIEMPOS DE BURBUJA
#USO >>> gnuplot graphic_burbuja.gp

#GENERAL TERMS

set term png
set xlabel "Dimension Vector"
set ylabel "Tiempo (seg)"
set xrange [0:52000]
set yrange [0:*]

#GRAFICA 1: COMPARACION CON/SIN OPTIMIZACION

set output "GRAFICAS/`whoami`/BURBUJA_COMPARE_`whoami`.png"
set title "TIEMPOS BURBUJA (CON/SIN OPTIMIZACION) - `whoami`"
plot "data/`whoami`/TBURBUJA_`whoami`.dat" title "TBURBUJA - `whoami`"
with linespoints linecolor 3,
"data/`whoami`/TBURBUJA_OPTIMIZADO_`whoami`.dat" title "TBURBUJA
OPTIMIZADO - `whoami`" with linespoints linecolor 4

#GRAFICA 2: AJUSTE A POTENCIAL

set output "GRAFICAS/`whoami`/BURBUJA_AJUSTE_`whoami`.png"
set title "TIEMPOS BURBUJA + AJUSTE - `whoami`"
f(x)=a*x*x+b*x+c
fit f(x) "data/`whoami`/TBURBUJA_`whoami`.dat" via a,b,c
plot "data/`whoami`/TBURBUJA_`whoami`.dat" title "TBURBUJA - `whoami`"
with linespoints linecolor 1, f(x) title "FUNCION AJUSTADA" with lines
linecolor 2
```

## 2.5 OBTENCIÓN DE AJUSTE Y TABLAS DE TIEMPOS

- Cuando el ajuste se ha realizado, el programa gnuplot arroja un fichero de salida llamado **fit.log**. Este fichero contiene los coeficientes del ajuste, así como otra información adicional.
- Para saber cómo de bueno es el ajuste que hemos hecho con respecto a una nube de puntos hemos utilizado el coeficiente de regresión, que tiene como fórmula la siguiente:

$$\eta = 1 - \frac{\sigma_{residuos\_tiempos}}{\sigma_{tiempos}}$$

- La varianza de los residuos se obtiene en el fichero mencionado anteriormente.
- El cálculo de este coeficiente, así como la administración y gestión de los datos a comparar por todos los integrantes del grupo se han realizado en un fichero de Excel, generando una tabla para cada algoritmo. A continuación, una captura del documento:

COMPARACIÓN DE TIEMPOS :: TIEMPOS HANOI			
DIMENSIÓN	DANIELSP	MARTIN	JOSE
2	0.000007	0.000015	0.000007
3	0.000006	0.000012	0.000008
4	0.000006	0.000030	0.000007
5	0.000008	0.000017	0.000007
6	0.000013	0.000020	0.000010
7	0.000014	0.000029	0.000014
8	0.000020	0.000044	0.000020
9	0.000033	0.000076	0.000035
10	0.000061	0.000134	0.000067
11	0.0000127	0.0000253	0.0000114
12	0.0000248	0.0000516	0.0000226
13	0.0000483	0.0000973	0.0000444
14	0.0000962	0.0001931	0.0000888
15	0.0001984	0.0003856	0.0001774
16	0.0004224	0.0007710	0.0003553
17	0.0007719	0.0015483	0.0007054
18	0.0014349	0.0030725	0.0013485
19	0.0028604	0.0061442	0.0026741
20	0.0056495	0.0122925	0.0051210
21	0.0113505	0.0245375	0.0101234
22	0.0225182	0.0490613	0.0198098
23	0.0449603	0.0981136	0.0395590
24	0.0898702	0.1962000	0.0794563
25	0.1799030	0.3925080	0.1581470
26	0.3597550	0.7848810	0.3160780
27	0.7187450	1.5699900	0.6323490
28	1.4376100	3.1410300	1.2667800

VARIANZA DE TIEMPOS		
DANIELSP	MARTIN	JOSE
0.09421086	0.4496635	0.07308897

VARIANZA RESIDUAL (FIT.LOG)		
DANIELSP	MARTIN	JOSE
6.55E-09	1.31E-08	5.51E-08

COEFICIENTES DE REGRESIÓN		
DANIELSP	MARTIN	JOSE
0.9999993	0.9999997	0.99999925

- La tabla posee, además, un código de colores programado por Excel con el siguiente motivo:
  - VERDE: Es el menor tiempo de los 3 comparados.
  - AMARILLO: Es el tiempo que se sitúa entre el más rápido y más lento.
  - ROJO: El mayor tiempo de los 3 comparados.

Una vez expuesto todo lo que hemos realizado, vamos a ponernos manos a la obra 😊.

### 3. BLOQUE 1: ALGORITMOS DE ORDENACIÓN

A continuación, mostraremos información detallada para cada algoritmo de ordenación.

#### 3.1 ALGORITMOS DE EFICIENCIA $O(n^2)$ :: ALGORITMO DE LA BURBUJA

- *Descripción:* El algoritmo de la burbuja consiste en intercambiar las componentes de un par consecutivo del vector en el caso de que la componente  $i$ -ésima tenga un mayor valor que el de su siguiente.
- *Expresión del algoritmo:*

```

1 inline void burbuja(int T[], int num_elem)
2 {
3     burbuja_lims(T, 0, num_elem);
4 }
5
6
7 static void burbuja_lims(int T[], int inicial, int final)
8 {
9     int i, j;
10    int aux;
11    for (i = inicial; i < final - 1; i++)
12        for (j = final - 1; j > i; j--)
13            if (T[j] < T[j-1])
14            {
15                aux = T[j];
16                T[j] = T[j-1];
17                T[j-1] = aux;
18            }
19 }
```

- *Eficiencia Teórica:* Sin meternos en muchos detalles técnicos, todos los accesos a vector, comparaciones, asignaciones e incrementos son de orden  $O(1)$ . Tenemos dos bucles anidados. El más exterior, se ejecutará  $n$  veces (siendo  $n$  el tamaño del vector), y el bucle interior se ejecutará otras  $n$  veces, por tanto, la eficiencia de este código es de orden  $O(n^2)$ .
- *Eficiencia Empírica:* Siguiendo el procedimiento del Punto 2 de este informe, se han obtenido los siguientes datos:

- Todos los apuntes que necesitas están aquí
- Al mejor precio del mercado, desde **2 cent.**
- Recoge los apuntes en tu copistería más cercana o recíbelos en tu casa
- Todas las anteriores son correctas

Algorítmica – SEMIGRUPO

P1: EFICIENCIA

Daniel | Jose | Martin

COMPARACIÓN DE TIEMPOS :: TIEMPOS BURBUJA

DIMENSIÓN	DANIELSP	MARTIN	JOSE
2000	0.0065493	0.0073921	0.0060152
4000	0.0286492	0.0296046	0.0252192
6000	0.0728419	0.0750621	0.0645100
8000	0.1412240	0.1451740	0.1241300
10000	0.1412240	0.2381450	0.2031410
12000	0.3410270	0.3513680	0.3006700
14000	0.5279260	0.4897120	0.4184590
16000	0.7105000	0.6512650	0.5570900
18000	0.9071240	0.8365480	0.7161650
20000	1.1517700	1.0445300	0.8929480
22000	1.3972100	1.2778300	1.0873200
24000	1.6864500	1.5271500	1.3088600
26000	1.6864500	1.8015000	1.5413400
28000	2.3493600	2.1016600	1.7985300
30000	2.6727500	2.4176900	2.0843200
32000	2.9711000	2.7710300	2.3715100
34000	3.3720000	3.1395300	2.6822200
36000	3.7860500	3.5458200	3.0303000
38000	4.3114300	3.9457700	3.4409200
40000	4.7133200	4.3912400	3.7839000
42000	5.1992200	4.8533700	4.2241300
44000	5.8902700	5.3381900	4.5577500
46000	6.2606100	5.8415000	5.0370700
48000	6.8208700	6.3924200	5.4413700
50000	7.4361900	6.9767100	6.0672700

COMPARACIÓN DE TIEMPOS :: TIEMPOS BURBUJA OPTIMIZADO

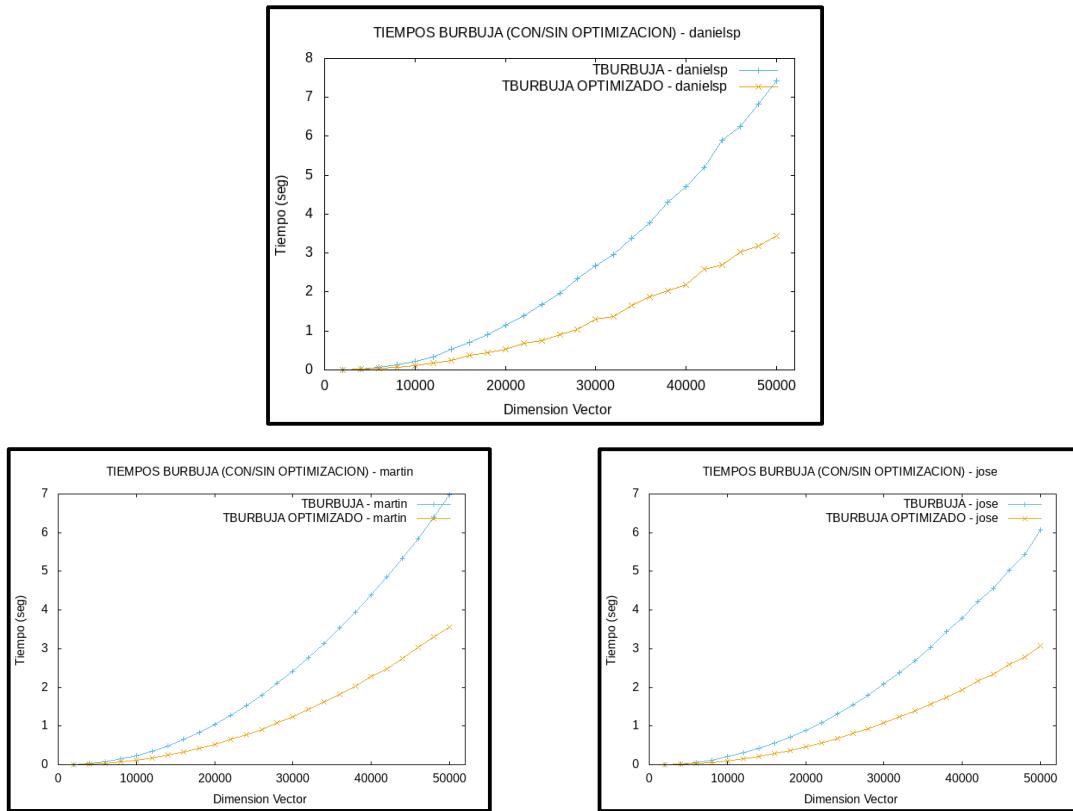
DIMENSIÓN	DANIELSP	MARTIN	JOSE
2000	0.0029976	0.0030376	0.0028488
4000	0.0139209	0.0143353	0.0138351
6000	0.0355399	0.0365427	0.0320417
8000	0.0691563	0.0704782	0.0615169
10000	0.1138420	0.1174850	0.1016990
12000	0.1696930	0.1749620	0.1512530
14000	0.2392310	0.2444060	0.2116800
16000	0.3690150	0.3283550	0.2838300
18000	0.4480260	0.4236810	0.3606540
20000	0.5253100	0.5266250	0.4563010
22000	0.6804760	0.6484780	0.5563110
24000	0.7503170	0.7794040	0.6815740
26000	0.8989290	0.9160200	0.8070870
28000	1.0427700	1.0812900	0.9214240
30000	1.3116700	1.2338800	1.0911400
32000	1.3711100	1.4302000	1.2321100
34000	1.6665400	1.6206000	1.3879700
36000	1.8841600	1.8235600	1.5744200
38000	2.0290300	2.0394900	1.7354700
40000	2.1967000	2.2788800	1.9290500
42000	2.5892900	2.4786200	2.1562500
44000	2.6893300	2.7413300	2.3408400
46000	3.0192700	3.0321200	2.5830500
48000	3.1836900	3.3045200	2.7831500
50000	3.4562000	3.5669900	3.0775400



Imprimir



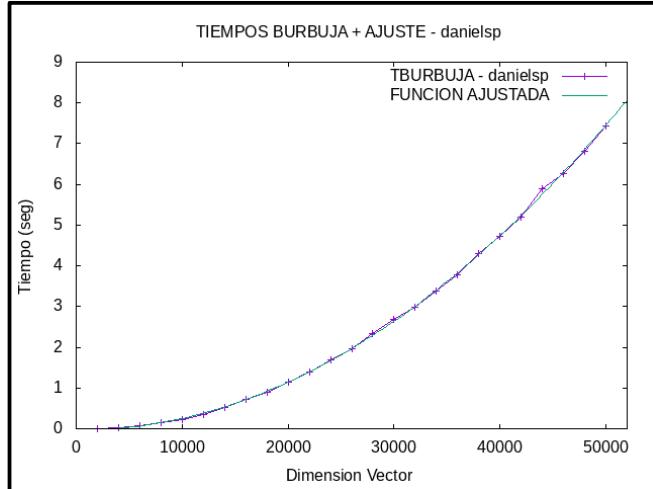
- *Análisis gráfico:* En primer lugar, vamos a comparar los resultados que se obtienen optimizando el código y sin optimizar de todos los integrantes de este grupo.



Como se aprecia, las tres gráficas muestran la mayor eficiencia del código optimizado frente al que no está optimizado. De hecho, en algunos casos el código optimizado llega a duplicar en velocidad al no optimizado.

Ahora vamos a realizar el ajuste de la nube de puntos de cada integrante obtenida con el código sin optimizar. Basándonos en el estudio de la eficiencia teórica vamos realizar una aproximación a una función cuadrática, es decir, a una del tipo:

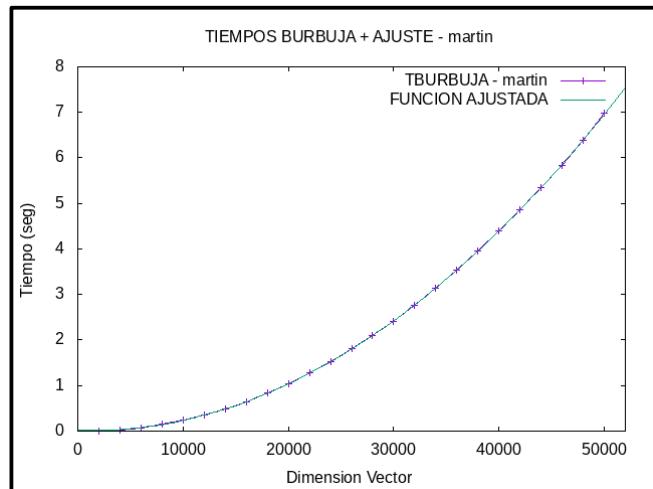
$$f(x) = ax^2 + bx + c$$



La función de ajuste para estos datos es la siguiente:

$$f(x) = 3.031920 \cdot 10^{-9} \cdot x^2 - 2.099215 \cdot 10^{-6} \cdot x - 2.764090 \cdot 10^{-2}$$

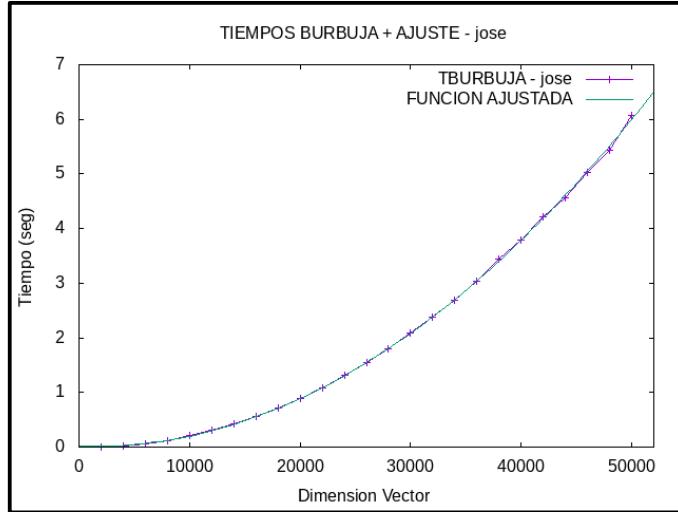
El coeficiente de regresión de este ajuste es:  $\eta = 0.99968762$ . Lo que nos indica que es un ajuste prácticamente excelente.



La función de ajuste para estos datos es la siguiente:

$$f(x) = 2.916717 \cdot 10^{-9} \cdot x^2 - 7.176975 \cdot 10^{-6} \cdot x + 1.583203 \cdot 10^{-2}$$

El coeficiente de regresión de este ajuste es:  $\eta = 0.99998475$ . Lo que nos indica que es un ajuste prácticamente excelente.

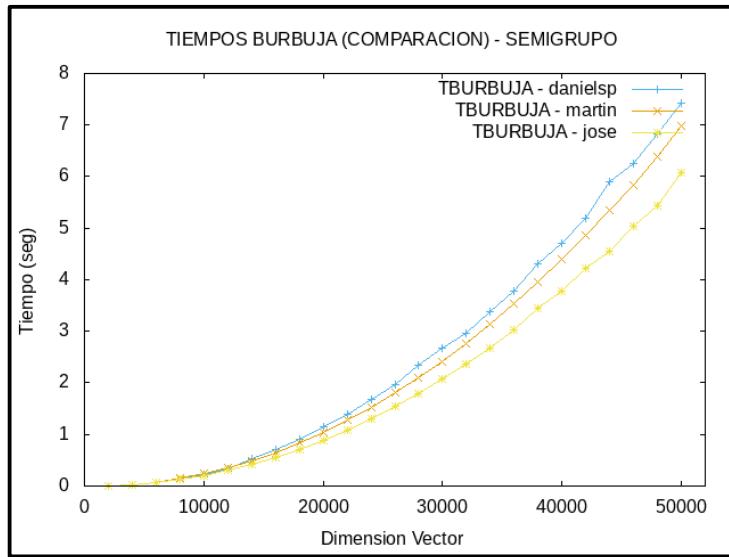


La función de ajuste para estos datos es la siguiente:

$$f(x) = 2.532852 \cdot 10^{-9} \cdot x^2 - 7.179141 \cdot 10^{-6} \cdot x - 1.822288 \cdot 10^{-2}$$

El coeficiente de regresión de este ajuste es:  $\eta = 0.99978801$ . Lo que nos indica que es un ajuste prácticamente excelente.

- *Comparación Final:* En base a la tabla resultante obtenida con los tiempos de todos los integrantes del grupo, se ha realizado la siguiente gráfica de comparación final.



En este caso, la CPU más potente ha sido la de Jose, seguida por la de Martín y la de Daniel. Sin embargo, en este caso no hay mucha diferencia entre los tiempos en cada PC, diferencia que era más notable en otros algoritmos.

- Todos los apuntes que necesitas están aquí
- Al mejor precio del mercado, desde **2 cent.**
- Recoge los apuntes en tu copistería más cercana o recíbelos en tu casa
- Todas las anteriores son correctas

### 3.2 ALGORITMOS DE EFICIENCIA $O(n^2)$ :: ALGORITMO DE INSERCIÓN

➤ *Descripción:* El algoritmo de inserción realiza una ordenación del vector de dimensión **n** de menor a mayor. Para ello, compara la posición i-ésima con las i-1 posiciones restantes, deteniéndose cuando se encuentre con un elemento menor o cuando ya no se encuentran elementos. Y es entonces, donde se inserta dicho elemento en la posición correspondiente, desplazando los demás elementos.

➤ *Expresión del algoritmo:*

```

1 inline static void insercion(int T[], int num_elem)
2 {
3     insercion_lims(T, 0, num_elem);
4 }
5
6
7 static void insercion_lims(int T[], int inicial, int final)
8 {
9     int i, j;
10    int aux;
11    for (i = inicial + 1; i < final; i++) {
12        j = i;
13        while ((T[j] < T[j-1]) && (j > 0)) {
14            aux = T[j];
15            T[j] = T[j-1];
16            T[j-1] = aux;
17            j--;
18        };
19    };
20 }
```

➤ *Eficiencia Teórica:* Tenemos dos bucles anidados, un for y un while. Ambos bucles se ejecutarán **n** veces, siendo **n** la dimensión del vector. Esto sucede siempre que el vector esté ordenado de forma decreciente (peor de los casos) o no esté ordenado. Por lo que la eficiencia teórica de este algoritmo es  $O(n^2)$ .

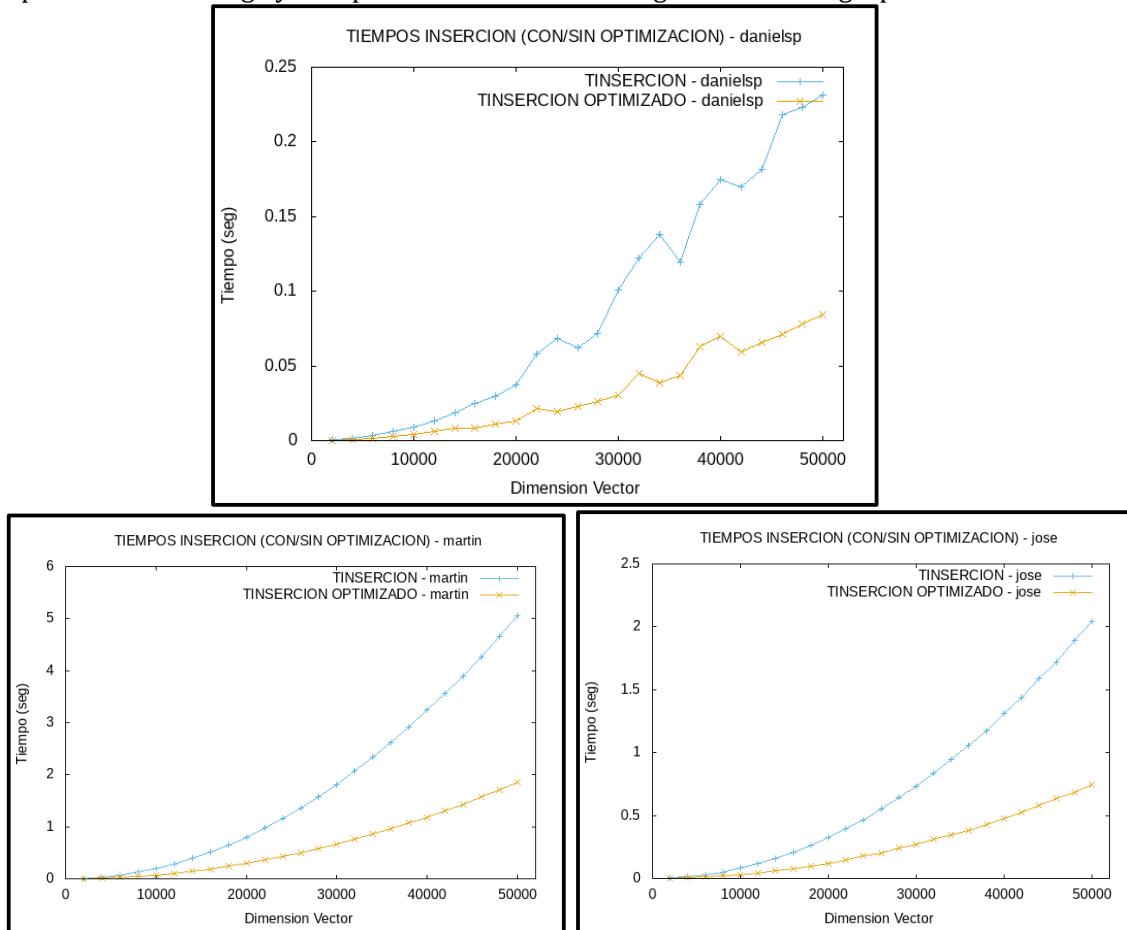
➤ *Eficiencia Empírica:* Siguiendo el procedimiento del **Punto 2** de este informe, se han obtenido los siguientes datos:



COMPARACIÓN DE TIEMPOS :: TIEMPOS INSERCIÓN			
DIMENSIÓN	DANIELSP	MARTIN	JOSE
2000	0.0003771	<b>0.0081443</b>	0.0035612
4000	0.0014731	<b>0.0326798</b>	0.0131260
6000	0.0032790	<b>0.0732114</b>	0.0296146
8000	0.0059403	<b>0.1281170</b>	0.0514741
10000	0.0091529	<b>0.1999450</b>	0.0810369
12000	0.0131185	<b>0.2884810</b>	0.1175750
14000	0.0183694	<b>0.3969510</b>	0.1581410
16000	0.0247833	<b>0.5135810</b>	0.2077860
18000	0.0299349	<b>0.6512160</b>	0.2618860
20000	0.0371335	<b>0.8038280</b>	0.3249990
22000	0.0579397	<b>0.9767060</b>	0.3929880
24000	0.0686375	<b>1.1590600</b>	0.4660740
26000	0.0621740	<b>1.3665700</b>	0.5509130
28000	0.0720226	<b>1.5808200</b>	0.6420480
30000	0.1005880	<b>1.8106700</b>	0.7336920
32000	0.1223010	<b>2.0662000</b>	0.8349970
34000	0.1382120	<b>2.3350400</b>	0.9455470
36000	0.1195360	<b>2.6159200</b>	1.0549600
38000	0.1583580	<b>2.9155000</b>	1.1743800
40000	0.1749130	<b>3.2428300</b>	1.3111400
42000	0.1698810	<b>3.5622200</b>	1.4337100
44000	0.1813440	<b>3.9027900</b>	1.5856900
46000	0.2185200	<b>4.2627800</b>	1.7186000
48000	0.2232980	<b>4.6592500</b>	1.8919400
50000	0.2310320	<b>5.0559100</b>	2.0466400

COMPARACIÓN DE TIEMPOS :: TIEMPOS INSERCIÓN OPTIMIZADO			
DIMENSIÓN	DANIELSP	MARTIN	JOSE
2000	0.0001394	<b>0.0030190</b>	0.0013031
4000	0.0005339	<b>0.0119410</b>	0.0051104
6000	0.0012266	<b>0.0267284</b>	0.0109062
8000	0.0027682	<b>0.0476785</b>	0.0192208
10000	0.0043470	<b>0.0741469</b>	0.0298456
12000	0.0062868	<b>0.1067420</b>	0.0432054
14000	0.0085124	<b>0.1451590</b>	0.0600842
16000	0.0086244	<b>0.1894330</b>	0.0761684
18000	0.0108439	<b>0.2404600</b>	0.0981244
20000	0.0133375	<b>0.2973290</b>	0.1186400
22000	0.0212209	<b>0.3583900</b>	0.1476690
24000	0.0193995	<b>0.4268410</b>	0.1763100
26000	0.0228156	<b>0.4996770</b>	0.2001460
28000	0.0264435	<b>0.5816050</b>	0.2383610
30000	0.0303928	<b>0.6655420</b>	0.2668190
32000	0.0447120	<b>0.7608180</b>	0.3103150
34000	0.0386656	<b>0.8573400</b>	0.3448340
36000	0.0434552	<b>0.9603090</b>	0.3829480
38000	0.0628661	<b>1.0714500</b>	0.4297230
40000	0.0699052	<b>1.1849500</b>	0.4797800
42000	0.0591387	<b>1.3078500</b>	0.5255790
44000	0.0653157	<b>1.4319800</b>	0.5815060
46000	0.0714635	<b>1.5692300</b>	0.6358500
48000	0.0777866	<b>1.7053200</b>	0.6864450
50000	0.0842529	<b>1.8520800</b>	0.7452310

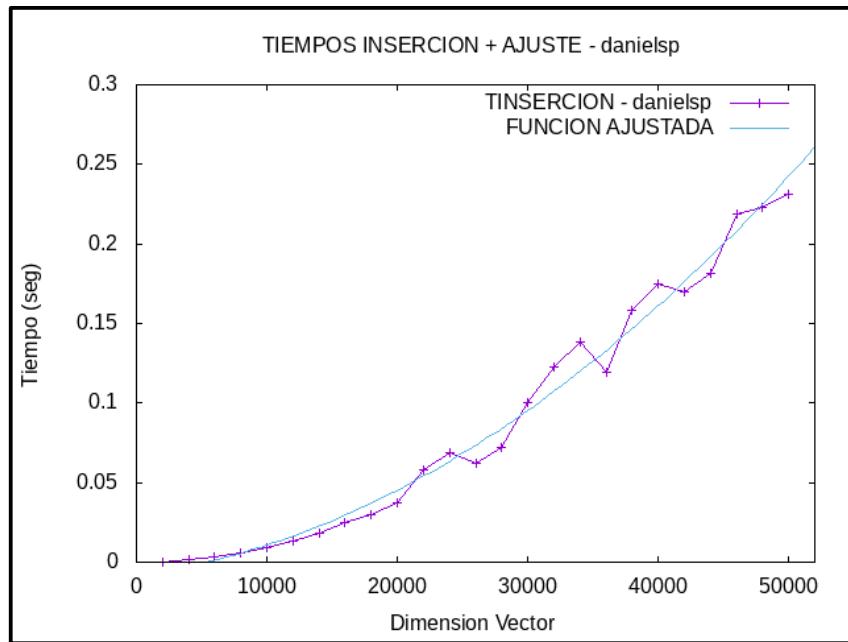
- *Análisis gráfico:* En primer lugar, vamos a comparar los resultados que se obtienen optimizando el código y sin optimizar de todos los integrantes de este grupo.



Como podemos ver, la diferencia de tiempos de ejecución, como en anteriores casos, solo es más apreciable en cuanto se aumenta el tamaño del vector, saliendo victoriosas las ejecuciones con optimización.

Ahora vamos a realizar el ajuste de la nube de puntos de cada integrante obtenida con el código sin optimizar. Basándonos en el estudio de la eficiencia teórica vamos realizar una aproximación a una función cuadrática, es decir, a una del tipo:

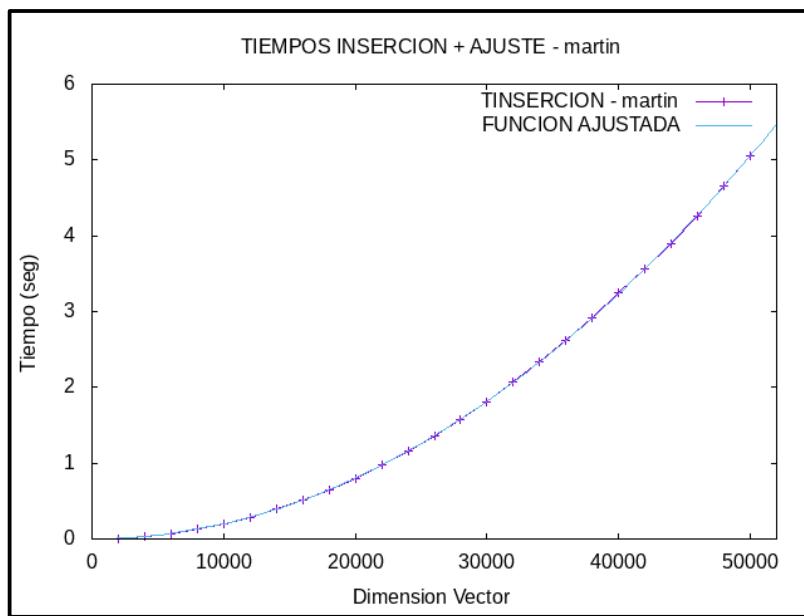
$$f(x) = ax^2 + bx + c$$



La función de ajuste para estos datos es la siguiente:

$$f(x) = 7.809122 \cdot 10^{-11} \cdot x^2 + 1.111699 \cdot 10^{-6} \cdot x - 8.244057 \cdot 10^{-3}$$

El coeficiente de regresión de este ajuste es:  $\eta = 0.98486018$ . Lo que nos indica que es un ajuste prácticamente excelente



La función de ajuste para estos datos es la siguiente:

$$f(x) = 2.026247 \cdot 10^{-9} \cdot x^2 - 3.083071 \cdot 10^{-7} \cdot x + 1.150851 \cdot 10^{-3}$$

El coeficiente de regresión de este ajuste es:  $\eta = 0.99999204$ . Lo que nos indica que es un ajuste prácticamente excelente

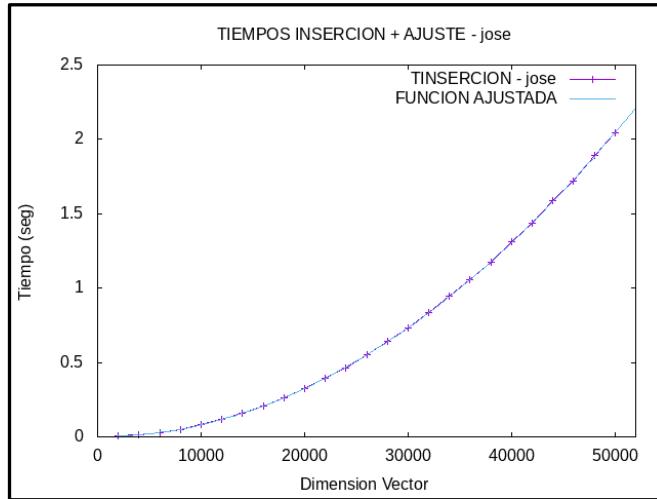
- Todos los apuntes que necesitas están aquí
- Al mejor precio del mercado, desde **2 cent**.
- Recoge los apuntes en tu copistería más cercana o recíbelos en tu casa
- Todas las anteriores son correctas

Algorítmica – SEMIGRUPO

P1: EFICIENCIA

Daniel | Jose | Martin

Imprimir

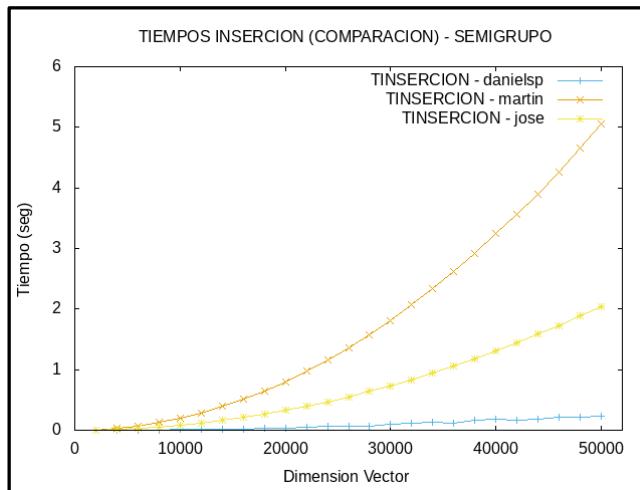


La función de ajuste para estos datos es la siguiente:

$$f(x) = 8.237956 \cdot 10^{-10} \cdot x^2 + 8.237956 \cdot 10^{-7} \cdot x + 1.703473 \cdot 10^{-3}$$

El coeficiente de regresión de este ajuste es:  $\eta = 0.99996338$ . Lo que nos indica que es un ajuste prácticamente excelente.

- *Comparación Final:* En base a la tabla resultante obtenida con los tiempos de todos los integrantes del grupo, se ha realizado la siguiente gráfica de comparación final.



En este caso, se observa una gran diferencia entre los tres ordenadores. La CPU de Daniel supera con creces a las otras dos CPU de Jose y Martín, aunque entre estas dos últimas también existe una diferencia, saliendo perjudicada la CPU de Martín.

### 3.3 ALGORITMOS DE EFICIENCIA $O(n^2)$ :: ALGORITMO DE SELECCIÓN

- *Descripción:* Al igual que el resto, se encarga de ordenar los elementos de un vector de tamaño  $n$  en sentido creciente. Para ello, busca la posición  $i$ -ésima del menor elemento de todo el vector y lo intercambia por la primera posición, segunda, etc.
- *Expresión del algoritmo:*

```

1 void seleccion(int T[], int num_elem)
2 {
3     seleccion_lims(T, 0, num_elem);
4 }
5
6 static void seleccion_lims(int T[], int inicial, int final)
7 {
8     int i, j, indice_menor;
9     int menor, aux;
10    for (i = inicial; i < final - 1; i++) {
11        indice_menor = i;
12        menor = T[i];
13        for (j = i; j < final; j++) {
14            if (T[j] < menor) {
15                indice_menor = j;
16                menor = T[j];
17            }
18            aux = T[i];
19            T[i] = T[indice_menor];
20            T[indice_menor] = aux;
21        };
22    }

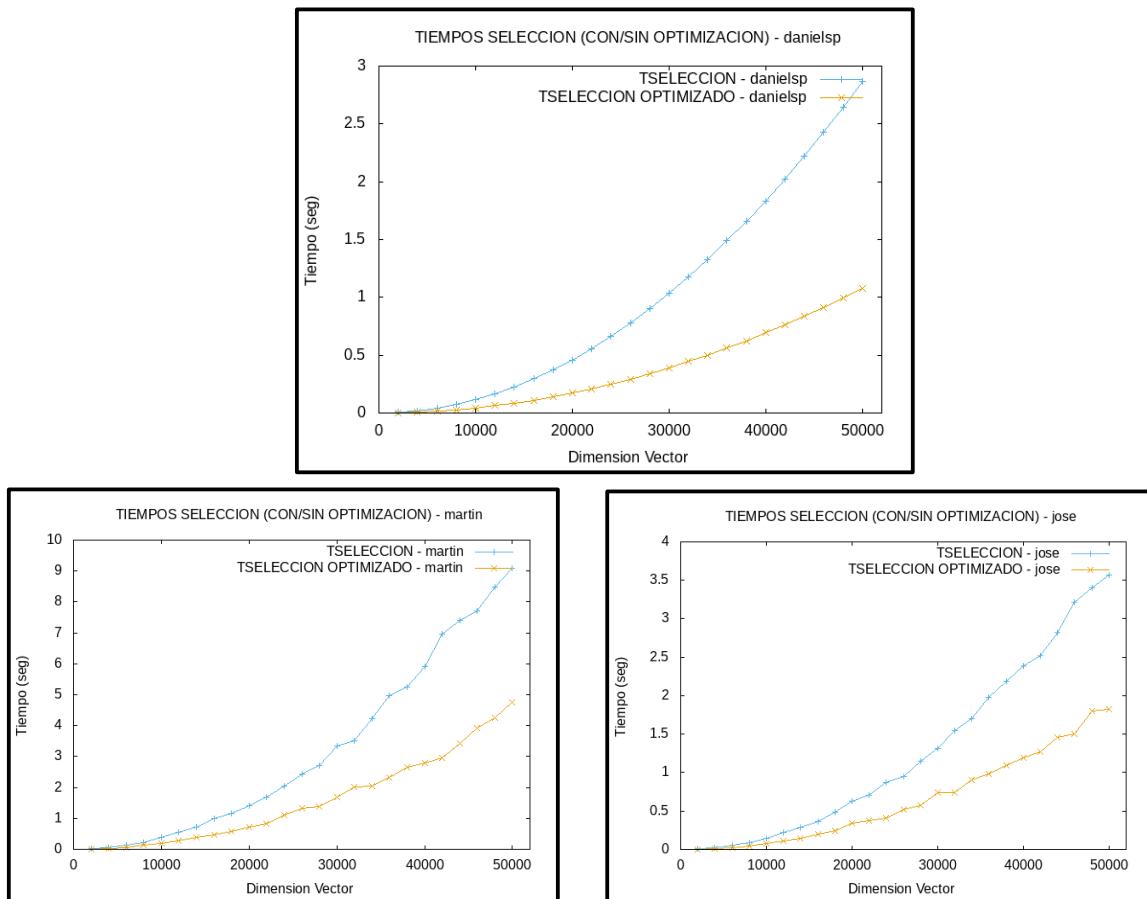
```

- *Eficiencia Teórica:* Como siempre, todos los accesos, incrementos, asignaciones, comparaciones y operaciones son  $O(1)$ . Tenemos dos bucles anidados que se ejecutan  $n$  veces cada uno, el más interno desde  $j$  hasta  $final$ , y el más externo desde la posición  $inicial$  hasta  $final - 1$  por lo que podemos decir que es de orden  $O(n^2)$ .
- *Eficiencia Empírica:* Siguiendo el procedimiento del **Punto 2** de este informe, se han obtenido los siguientes datos:

COMPARACIÓN DE TIEMPOS :: TIEMPOS SELECCIÓN			
DIMENSIÓN	DANIELSP	MARTIN	JOSE
2000	0.0044922	0.0147511	0.0061956
4000	0.0186691	0.0591092	0.0252377
6000	0.0417971	0.1345510	0.0546205
8000	0.0740273	0.2329120	0.0887981
10000	0.1154210	0.3890280	0.1423320
12000	0.1659890	0.5414770	0.2193850
14000	0.2255280	0.7234430	0.2874370
16000	0.2944440	0.9954040	0.3641540
18000	0.3725740	1.1582300	0.4838410
20000	0.4598400	1.4005300	0.6314240
22000	0.5563170	1.6986000	0.7121050
24000	0.6612570	2.0309700	0.8734470
26000	0.7765540	2.4429200	0.9517000
28000	0.9003490	2.7049500	1.1491900
30000	1.0334400	3.3301000	1.3117800
32000	1.1744800	3.5048400	1.5470900
34000	1.3266700	4.2385000	1.6968700
36000	1.4881500	4.9699100	1.9819900
38000	1.6566700	5.2486500	2.1915100
40000	1.8353900	5.9035600	2.3849000
42000	2.0232200	6.9535900	2.5187200
44000	2.2202700	7.3972200	2.8123200
46000	2.4264100	7.6984100	3.2106900
48000	2.6417200	8.4691900	3.4070100
50000	2.8689300	9.0868100	3.5727800

COMPARACIÓN DE TIEMPOS :: TIEMPOS SELECCIÓN OPTIMIZADO			
DIMENSIÓN	DANIELSP	MARTIN	JOSE
2000	0.0018272	0.0077359	0.0031712
4000	0.0072832	0.0299220	0.0117653
6000	0.0160074	0.0642065	0.0275807
8000	0.0282446	0.1264010	0.0486027
10000	0.0439414	0.1956450	0.0721168
12000	0.0629061	0.2853010	0.1052880
14000	0.0855276	0.3746030	0.1388630
16000	0.1113370	0.4634690	0.1934790
18000	0.1410470	0.5822080	0.2393080
20000	0.1740830	0.7257850	0.3372470
22000	0.2101200	0.8338840	0.3798730
24000	0.2498070	1.1006700	0.4098080
26000	0.2930170	1.3226300	0.5167670
28000	0.3400560	1.3832100	0.5721840
30000	0.3899260	1.6859400	0.7411530
32000	0.4440980	2.0255500	0.7371800
34000	0.5006290	2.0471800	0.9070670
36000	0.5612930	2.3268400	0.9861780
38000	0.6239300	2.6559700	1.0895300
40000	0.6920300	2.7775500	1.1942300
42000	0.7626550	2.9620300	1.2690600
44000	0.8374520	3.4124600	1.4638900
46000	0.9146420	3.9186500	1.4975100
48000	0.9966810	4.2468000	1.8018600
50000	1.0791900	4.7583800	1.8215400

- *Análisis gráfico:* En primer lugar, vamos a comparar los resultados que se obtienen optimizando el código y sin optimizar de todos los integrantes de este grupo.



Como podemos ver, al principio de las gráficas no existe una gran diferencia entre los tiempos de ejecución, pero conforme aumenta el tamaño del vector, la diferencia se hace más notoria y apreciable.

Ahora vamos a realizar el ajuste de la nube de puntos de cada integrante obtenida con el código sin optimizar. Basándonos en el estudio de la eficiencia teórica vamos realizar una aproximación a una función cuadrática, es decir, a una del tipo:

$$f(x) = ax^2 + bx + c$$

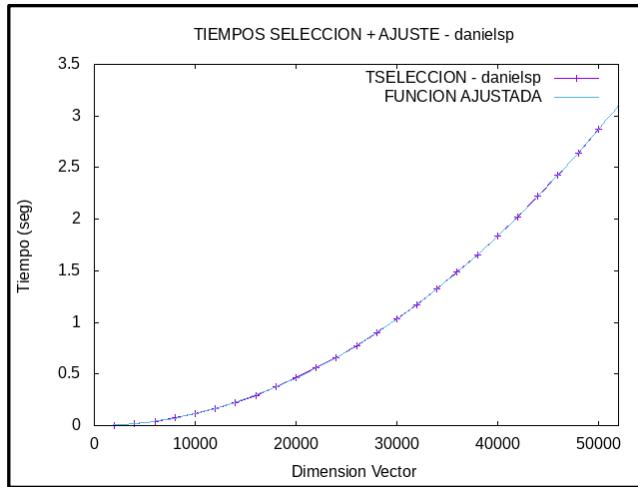
- Todos los apuntes que necesitas están aquí
- Al mejor precio del mercado, desde **2 cent**.
- Recoge los apuntes en tu copistería más cercana o recíbelos en tu casa
- Todas las anteriores son correctas

Algorítmica – SEMIGRUPO

P1: EFICIENCIA

Daniel | Jose | Martin

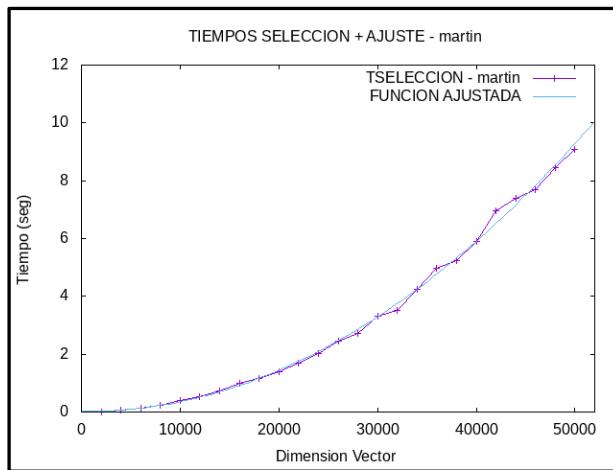
Imprimir 



La función de ajuste para estos datos es la siguiente:

$$f(x) = 1.145322 \cdot 10^{-9} \cdot x^2 + 7.644629 \cdot 10^{-8} \cdot x + 5.173652 \cdot 10^{-5}$$

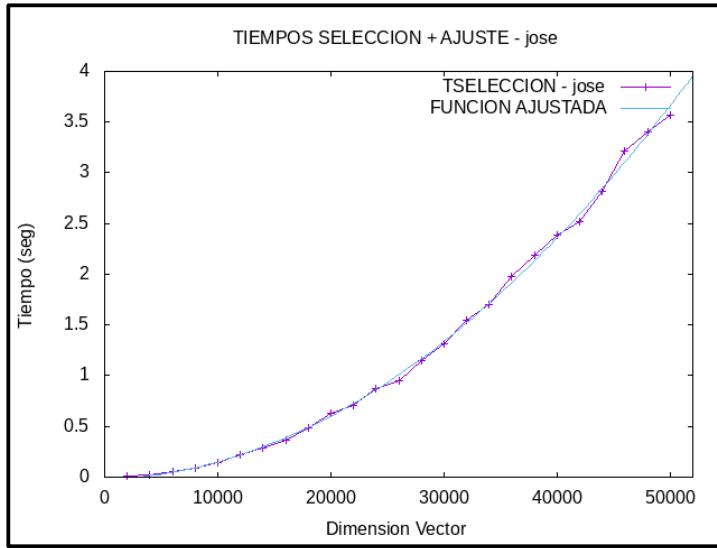
El coeficiente de regresión de este ajuste es:  $\eta = 0.99999962$ . Lo que nos indica que es un ajuste prácticamente excelente.



La función de ajuste para estos datos es la siguiente:

$$f(x) = 3.809258 \cdot 10^{-9} \cdot x^2 - 5.335713 \cdot 10^{-6} \cdot x + 2.445088 \cdot 10^{-2}$$

El coeficiente de regresión de este ajuste es:  $\eta = 0.997577$ . Lo que nos indica que es un ajuste prácticamente excelente.

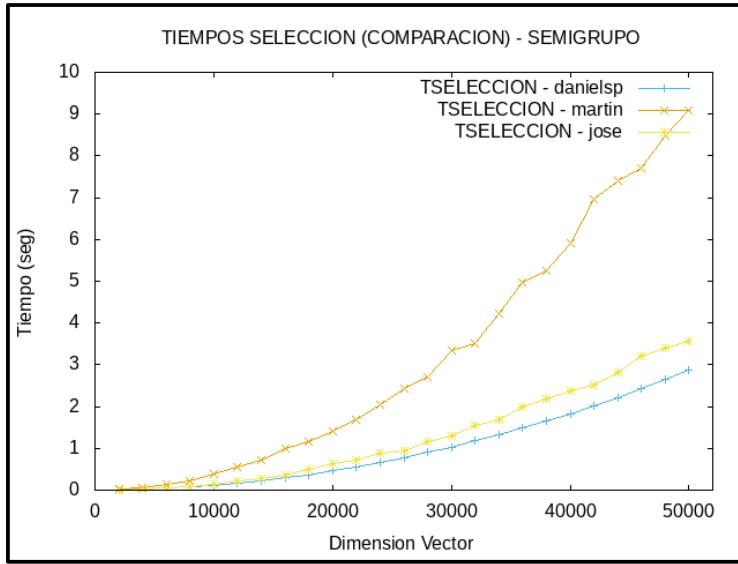


La función de ajuste para estos datos es la siguiente:

$$f(x) = 1.421859 \cdot 10^{-9} \cdot x^2 + 2.535798 \cdot 10^{-6} \cdot x - 1.783317 \cdot 10^{-2}$$

El coeficiente de regresión de este ajuste es:  $\eta = 0.99854006$ . Lo que nos indica que es un ajuste prácticamente excelente.

- *Comparación Final:* En base a la tabla resultante obtenida con los tiempos de todos los integrantes del grupo, se ha realizado la siguiente gráfica de comparación final.



En este algoritmo, la CPU que ha salido más victoriosa ha sido la de Daniel, seguida por poco de la de Jose, dejando más atrás la de Martín.

### 3.5 ALGORITMOS DE EFICIENCIA $O(n \cdot \log(n))$ :: ALGORITMO MERGESORT

- *Descripción:* El algoritmo de mezcla (o mergesort) se encarga al igual que el resto de ordenar los elementos de un vector de tamaño  $n$  en orden creciente. Se basa en la técnica “Divide y Vencerás”, es decir, divide el vector en 2 partes y aplica el algoritmo a cada parte y así con cada una de las divisiones hasta llegar a un caso que sea fácilmente resoluble.
- *Expresión del algoritmo:*

```

1 void mergesort(int T[], int num_elem)
2 {
3     mergesort_lims(T, 0, num_elem);
4 }
5
6 static void mergesort_lims(int T[], int inicial, int final)
7 {
8     if (final - inicial < UMBRAL_MS)
9     {
10         insercion_lims(T, inicial, final);
11     } else {
12         int k = (final - inicial)/2;
13
14         int * U = new int [k - inicial + 1];
15         assert(U);
16         int l, l2;
17         for (l = 0, l2 = inicial; l < k; l++, l2++)
18             U[l] = T[l2];
19         U[l] = INT_MAX;
20
21         int * V = new int [final - k + 1];
22         assert(V);
23         for (l = 0, l2 = k; l < final - k; l++, l2++)
24             V[l] = T[l2];
25         V[l] = INT_MAX;
26
27         mergesort_lims(U, 0, k);
28         mergesort_lims(V, 0, final - k);
29         fusion(T, inicial, final, U, V);
30         delete [] U;
31         delete [] V;
32     };
33 }

```

Este algoritmo en concreto se ejecuta siempre y cuando la dimensión del vector sea superior a un umbral predefinido (*UMBRALMS*). En caso contrario, la ordenación se realizará siguiendo el algoritmo de inserción ya estudiado previamente.

- ***Eficiencia Empírica:*** Siguiendo el procedimiento del **Punto 2** de este informe, se han obtenido los siguientes datos:

COMPARACIÓN DE TIEMPOS :: TIEMPOS MERGESORT			
DIMENSIÓN	DANIELSP	MARTIN	JOSE
2000	0.0002386	0.0005185	0.0002140
4000	0.0005271	0.0011331	0.0004655
6000	0.0009517	0.0020757	0.0008532
8000	0.0011188	0.0024687	0.0010332
10000	0.0015419	0.0033923	0.0013725
12000	0.0021060	0.0045320	0.0018920
14000	0.0020670	0.0048290	0.0018220
16000	0.0024480	0.0055960	0.0021890
18000	0.0032610	0.0063910	0.0028400
20000	0.0033900	0.0073680	0.0033970
22000	0.0038200	0.0083960	0.0034300
24000	0.0047810	0.0101690	0.0039710
26000	0.0040090	0.0086960	0.0040560
28000	0.0043640	0.0096890	0.0039940
30000	0.0047850	0.0106450	0.0042680
32000	0.0051980	0.0114520	0.0049330
34000	0.0060950	0.0124810	0.0051500
36000	0.0061390	0.0136520	0.0059410
38000	0.0066420	0.0145320	0.0063400
40000	0.0070660	0.0158310	0.0063240
42000	0.0078030	0.0168850	0.0069040
44000	0.0081000	0.0181070	0.0076330
46000	0.0086530	0.0189400	0.0083270
48000	0.0094880	0.0204490	0.0084650
50000	0.0097930	0.0216690	0.0092350

COMPARACIÓN DE TIEMPOS :: TIEMPOS MERGESORT OPTIMIZADO			
DIMENSIÓN	DANIELSP	MARTIN	JOSE
2000	0.0001087	0.0002394	0.0000964
4000	0.0002485	0.0005407	0.0002220
6000	0.0004382	0.0009516	0.0003881
8000	0.0005464	0.0011931	0.0004839
10000	0.0007367	0.0016125	0.0006464
12000	0.0009890	0.0021820	0.0008660
14000	0.0010270	0.0022660	0.0009050
16000	0.0012250	0.0026880	0.0010610
18000	0.0014380	0.0031150	0.0012650
20000	0.0016840	0.0036060	0.0014240
22000	0.0019640	0.0040500	0.0020450
24000	0.0020870	0.0046040	0.0019260
26000	0.0020130	0.0044660	0.0018600
28000	0.0022440	0.0048690	0.0019350
30000	0.0024860	0.0053280	0.0021720
32000	0.0026100	0.0057060	0.0025030
34000	0.0028270	0.0062080	0.0025600
36000	0.0030670	0.0066490	0.0029230
38000	0.0033370	0.0071950	0.0029190
40000	0.0035290	0.0076580	0.0030980
42000	0.0037300	0.0081290	0.0034270
44000	0.0043950	0.0086580	0.0034890
46000	0.0042170	0.0091950	0.0042140
48000	0.0044730	0.0097010	0.0039810
50000	0.0047110	0.0102630	0.0045490

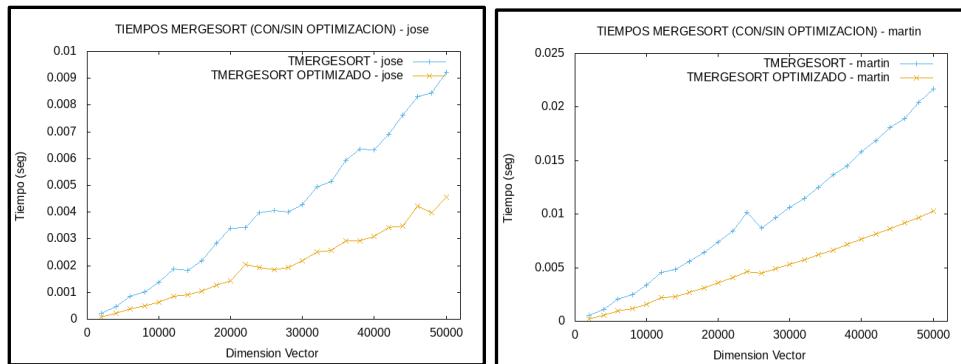
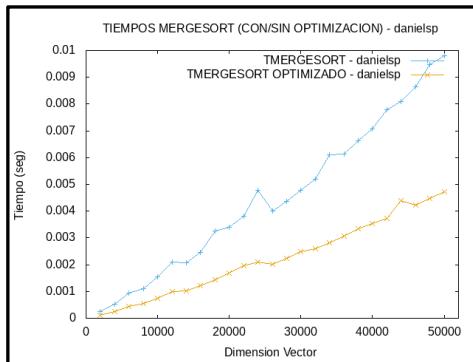
- Todos los apuntes que necesitas están aquí
- Al mejor precio del mercado, desde **2 cent**.
- Recoge los apuntes en tu copistería más cercana o recíbelos en tu casa
- Todas las anteriores son correctas

Algorítmica – SEMIGRUPO

P1: EFICIENCIA

Daniel | Jose | Martin

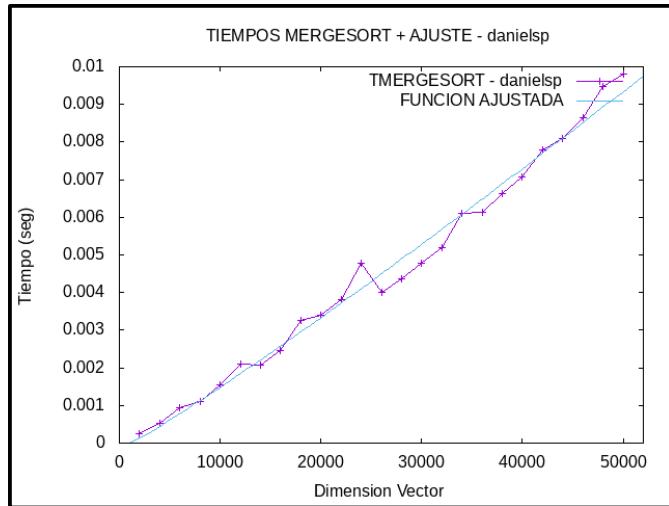
➤ *Análisis gráfico:* En primer lugar, vamos a comparar los resultados que se obtienen optimizando el código y sin optimizar de todos los integrantes de este grupo.



Como podemos ver, el código optimizado rinde mejor que el no optimizado en términos de tiempo. Incluso en algunos casos el tiempo de ejecución del algoritmo se ve reducido hasta la mitad con respecto al no optimizado.

Ahora vamos a realizar el ajuste de la nube de puntos de cada integrante obtenida con el código sin optimizar. Basándonos en el estudio de la eficiencia teórica vamos realizar una aproximación a una función logarítmica, es decir, a una del tipo:

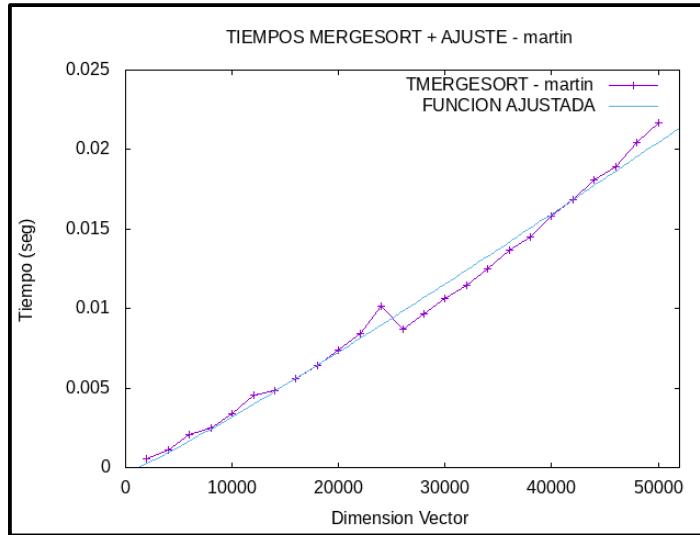
$$f(x) = ax \cdot \log(x) + b$$



La función de ajuste para estos datos es la siguiente:

$$f(x) = 1.75064 \cdot 10^{-8} \cdot x \cdot \log(x) - 0.000131236$$

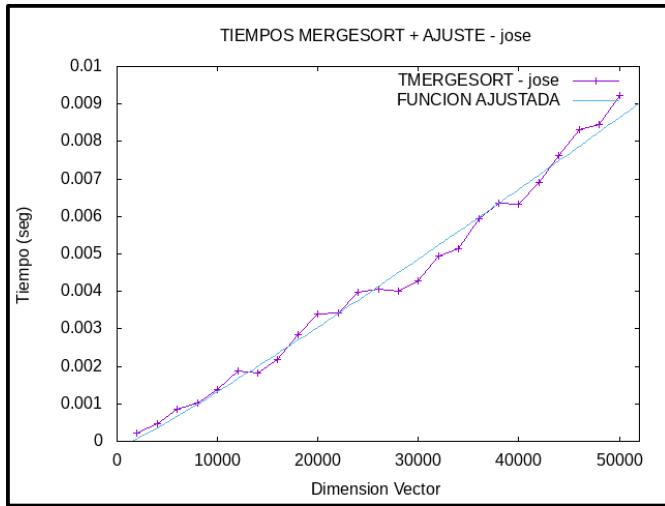
El coeficiente de regresión de este ajuste es:  $\eta = 0.98655443$ . Lo que nos indica que es un ajuste prácticamente excelente.



La función de ajuste para estos datos es la siguiente:

$$f(x) = 3.84269 \cdot 10^{-8} \cdot x \cdot \log(x) - 0.000335396$$

El coeficiente de regresión de este ajuste es:  $\eta = 0.98886565$ . Lo que nos indica que es un ajuste prácticamente excelente.

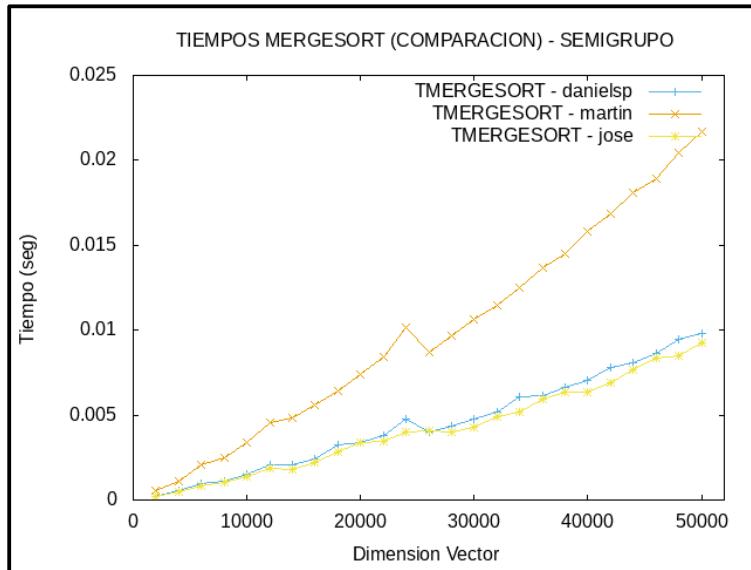


La función de ajuste para estos datos es la siguiente:

$$f(x) = 1.62904 \cdot 10^{-8} \cdot x \cdot \log(x) - 0.000177802$$

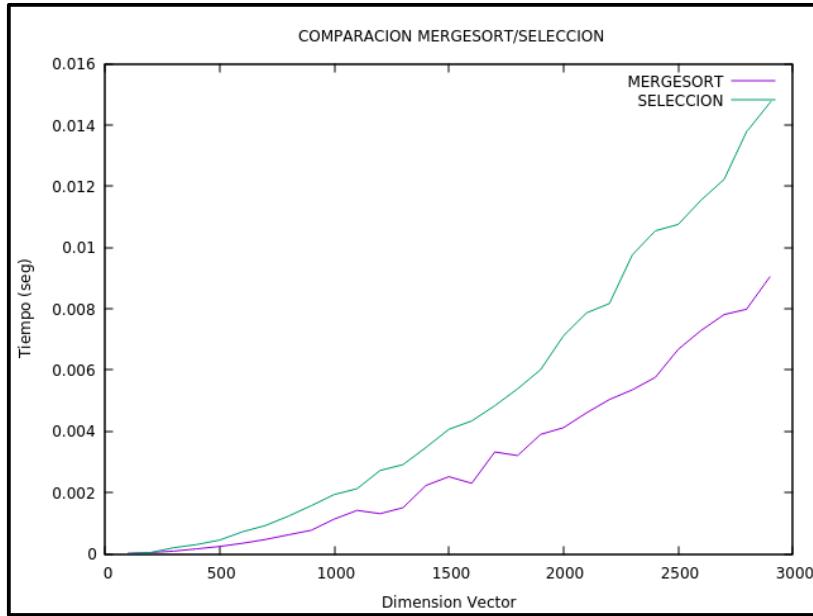
El coeficiente de regresión de este ajuste es:  $\eta = 0.98686081$ . Lo que nos indica que es un ajuste prácticamente excelente.

- *Comparación Final:* En base a la tabla resultante obtenida con los tiempos de todos los integrantes del grupo, se ha realizado la siguiente gráfica de comparación final.

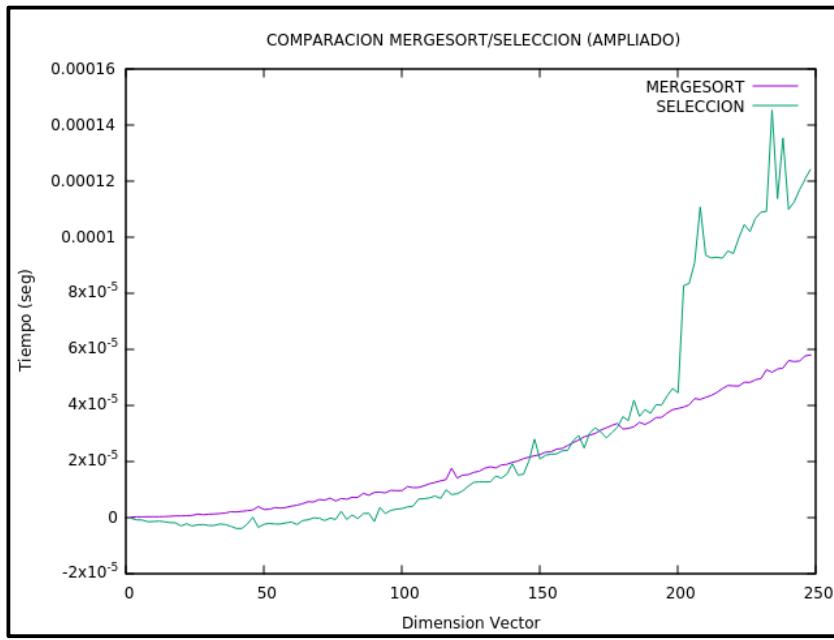


Se puede apreciar que, para este caso, la CPU que peor ha rendido ha sido la de Martín con diferencia, mientras que tanto la de José como la de Daniel consiguen mejores tiempos, siendo la de José algo mejor.

- *Comprobación teórica del Umbral de Mergesort:* En este apartado extra de “Mergesort” vamos a ver cuándo es mejor utilizar un algoritmo cuadrático y cuándo uno logarítmico como éste.
- Se han ampliado las gráficas en primer lugar de 0 a 3000 en el Eje X.



*COMENTARIO: No se puede apreciar bien cuál es el punto de corte entre las dos gráficas. Es por eso por lo que se adjunta una ampliación mucho mayor del eje X de 2 hasta 200.*



*COMENTARIO: Podemos ver que a partir de 150 elementos se entremezclan las gráficas, siendo en 178 el último corte. A partir de ahí, las gráficas se separan, siendo mejor mergesort. Conlcuimos que el umbral óptimo estaría en 178.*

- Todos los apuntes que necesitas están aquí
- Al mejor precio del mercado, desde **2 cent**.
- Recoge los apuntes en tu copistería más cercana o recíbelos en tu casa
- Todas las anteriores son correctas

### 3.6 ALGORITMOS DE EFICIENCIA $O(n \cdot \log(n))$ :: ALGORITMO QUICKSORT

- *Descripción:* Otro algoritmo que se encarga de la ordenación de un vector. La forma de proceder es la siguiente: Selecciona un elemento de las componentes del vector a ordenar, al que se llamará pivote, y a continuación se sitúan a la izquierda del pivote los elementos menores que éste, asimismo, a la derecha quedarán los mayores. Se producirán dos subvectores en torno al pivote, por lo que basta repetir este proceso recursivamente para cada una de las nuevas divisiones que se vayan generando hasta finalizar.
- *Expresión del algoritmo:*

```

1 inline void quicksort(int T[], int num_elem)
2 {
3     quicksort_lims(T, 0, num_elem);
4 }
5
6 static void quicksort_lims(int T[], int inicial, int final)
7 {
8     int k;
9     if (final - inicial < UMBRAL_QS) {
10         insercion_lims(T, inicial, final);
11     } else {
12         dividir_qs(T, inicial, final, k);
13         quicksort_lims(T, inicial, k);
14         quicksort_lims(T, k + 1, final);
15     };
16 }
17
18
19 static void dividir_qs(int T[], int inicial, int final, int & pp)
20 {
21     int pivote, aux;
22     int k, l;
23
24     pivote = T[inicial];
25     k = inicial;
26     l = final;
27     do {
28         k++;
29     } while ((T[k] <= pivote) && (k < final-1));
30     do {
31         l--;
32     } while (T[l] > pivote);
33     while (k < l) {
34         aux = T[k];
35         T[k] = T[l];
36         T[l] = aux;
37         do k++; while (T[k] <= pivote);
38         do l--; while (T[l] > pivote);
39     };
40     aux = T[inicial];
41     T[inicial] = T[l];
42     T[l] = aux;
43     pp = l;
44 }

```

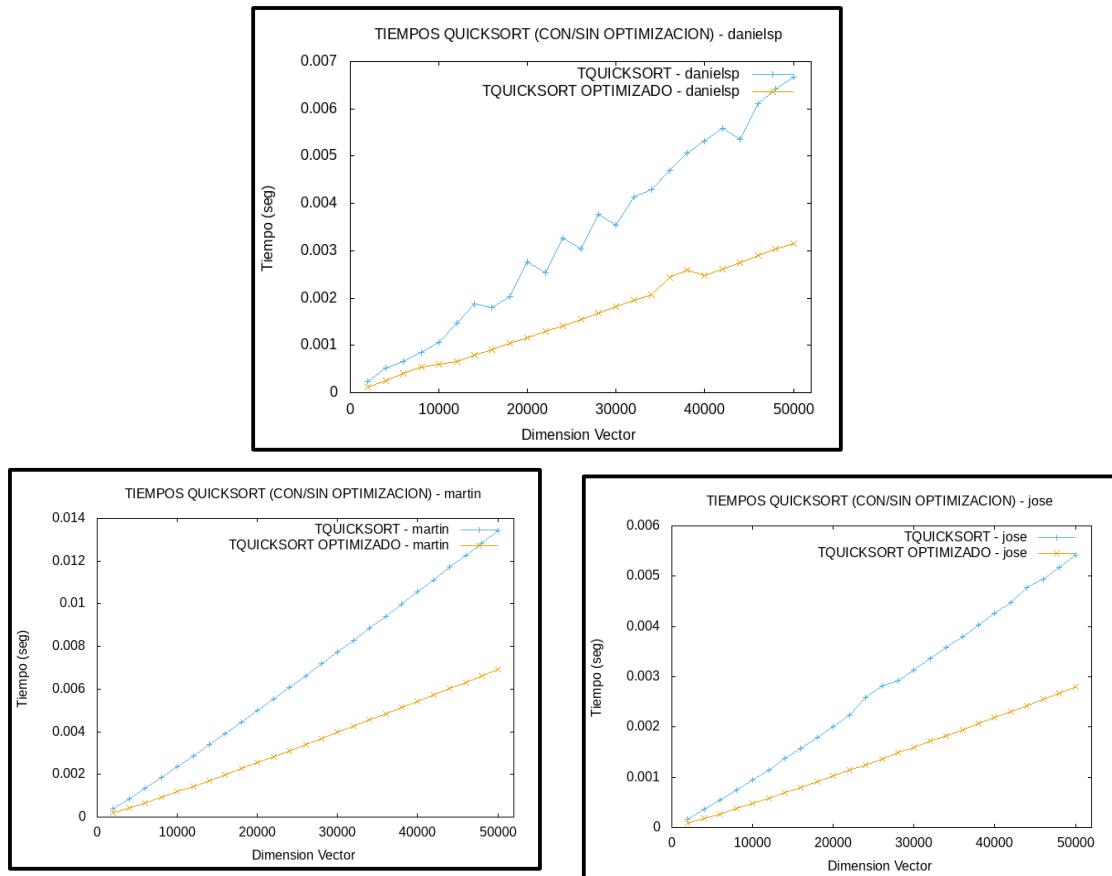


- Al igual que sucedía con Mergesort, existe un umbral para la realización de este proceso. En caso de que no se supere dicho parámetro, se procedería a ordenar los elementos mediante el algoritmo de Inserción.
- *Eficiencia Empírica:* Siguiendo el procedimiento del **Punto 2** de este informe, se han obtenido los siguientes datos:

COMPARACIÓN DE TIEMPOS :: TIEMPOS QUICKSORT			
DIMENSIÓN	DANIELSP	MARTÍN	JOSE
2000	0.0002395	0.0004019	0.0001728
4000	0.0005132	0.0008605	0.0003573
6000	0.0006508	0.0013407	0.0005456
8000	0.0008420	0.0018401	0.0007434
10000	0.0010727	0.0023427	0.0009520
12000	0.0014623	0.0028682	0.0011513
14000	0.0018822	0.0033855	0.0013683
16000	0.0017898	0.0039066	0.0015781
18000	0.0020334	0.0044445	0.0017907
20000	0.0027678	0.0049801	0.0020121
22000	0.0025308	0.0055217	0.0022320
24000	0.0032686	0.0060828	0.0025878
26000	0.0030274	0.0066164	0.0028191
28000	0.0037734	0.0071792	0.0029157
30000	0.0035341	0.0077238	0.0031260
32000	0.0041379	0.0082852	0.0033581
34000	0.0042915	0.0088474	0.0035738
36000	0.0046908	0.0094139	0.0038001
38000	0.0050572	0.0099724	0.0040279
40000	0.0053144	0.0105458	0.0042583
42000	0.0055841	0.0111143	0.0044767
44000	0.0053500	0.0117057	0.0047796
46000	0.0061057	0.0122675	0.0049412
48000	0.0064165	0.0128585	0.0051672
50000	0.0066761	0.0134109	0.0054146

COMPARACIÓN DE TIEMPOS :: TIEMPOS QUICKSORT OPTIMIZADO			
DIMENSIÓN	DANIELSP	MARTÍN	JOSE
2000	0.0001177	0.0001968	0.0000864
4000	0.0002556	0.0004280	0.0001751
6000	0.0004003	0.0006730	0.0002727
8000	0.0005501	0.0009250	0.0003814
10000	0.0005985	0.0011854	0.0004809
12000	0.0006631	0.0014500	0.0005854
14000	0.0007859	0.0017186	0.0006915
16000	0.0009093	0.0019892	0.0008014
18000	0.0010349	0.0022699	0.0009142
20000	0.0011631	0.0025453	0.0010227
22000	0.0012907	0.0028282	0.0011382
24000	0.0014193	0.0031084	0.0012503
26000	0.0015504	0.0033919	0.0013655
28000	0.0016799	0.0036806	0.0014862
30000	0.0018122	0.0039692	0.0015959
32000	0.0019496	0.0042562	0.0017174
34000	0.0020783	0.0045452	0.0018301
36000	0.0024380	0.0048394	0.0019471
38000	0.0025862	0.0051354	0.0020672
40000	0.0024830	0.0054329	0.0021901
42000	0.0026191	0.0057382	0.0023078
44000	0.0027545	0.0060296	0.0024226
46000	0.0028910	0.0063231	0.0025506
48000	0.0030323	0.0066212	0.0026667
50000	0.0031614	0.0069274	0.0027944

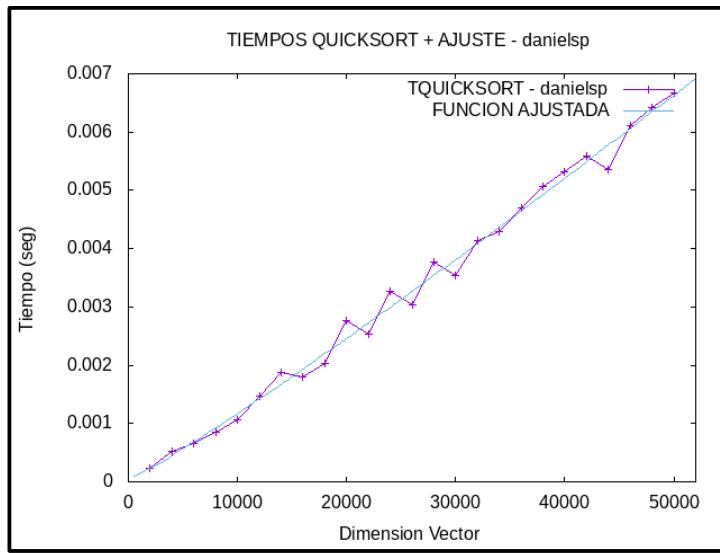
- *Análisis gráfico:* En primer lugar, vamos a comparar los resultados que se obtienen optimizando el código y sin optimizar de todos los integrantes de este grupo.



Una vez más, en las gráficas anteriores podemos ver con facilidad que los tiempos conseguidos en las ejecuciones con optimización son mucho mejores que los que no la tienen, diferencia que se acentúa conforme aumenta el tamaño del vector de datos.

Ahora vamos a realizar el ajuste de la nube de puntos de cada integrante obtenida con el código sin optimizar. Basándonos en el estudio de la eficiencia teórica vamos realizar una aproximación a una función logarítmica, es decir, a una del tipo:

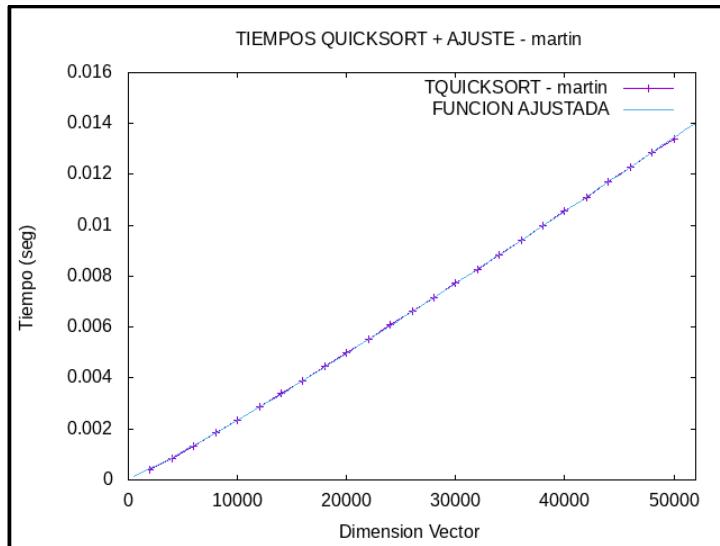
$$f(x) = ax \cdot \log(x) + b$$



La función de ajuste para estos datos es la siguiente:

$$f(x) = 1.21744 \cdot 10^{-8} \cdot x \cdot \log(x) - 4.73132 \cdot 10^{-5}$$

El coeficiente de regresión de este ajuste es:  $\eta = 0.99160924$ . Lo que nos indica que es un ajuste prácticamente excelente.

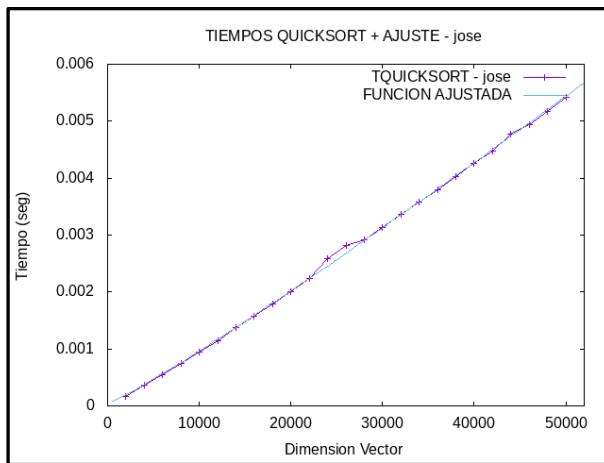
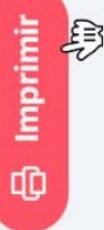


La función de ajuste para estos datos es la siguiente:

$$f(x) = 2.47303 \cdot 10^{-8} \cdot x \cdot \log(x) - 6.77704 \cdot 10^{-5}$$

El coeficiente de regresión de este ajuste es:  $\eta = 0.99998033$ . Lo que nos indica que es un ajuste prácticamente excelente.

- Todos los apuntes que necesitas están aquí
- Al mejor precio del mercado, desde **2 cent**.
- Recoge los apuntes en tu copistería más cercana o recíbelos en tu casa
- Todas las anteriores son correctas

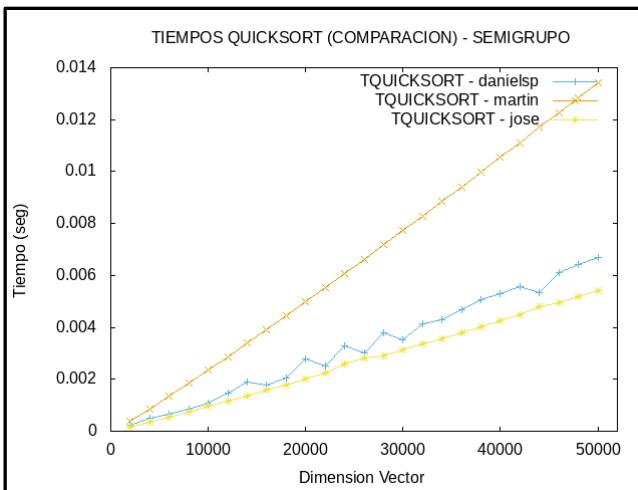


La función de ajuste para estos datos es la siguiente:

$$f(x) = 9.97138 \cdot 10^{-9} \cdot x \cdot \log(x) - 4.51082 \cdot 10^{-5}$$

El coeficiente de regresión de este ajuste es:  $\eta = 0.999249$ . Lo que nos indica que es un ajuste prácticamente excelente.

- *Comparación Final:* En base a la tabla resultante obtenida con los tiempos de todos los integrantes del grupo, se ha realizado la siguiente gráfica de comparación final.



Una vez más, las CPU de Daniel y de José están enfrentadas en cuanto a tiempos, siendo la de José algo más veloz mientras que la de Martín ha obtenido peores tiempos con diferencia.

### 3.7 ALGORITMOS DE EFICIENCIA $O(n \cdot \log(n))$ :: ALGORITMO HEAPSORT

- *Descripción:* Este es el último de los algoritmos del bloque de ordenación de vectores. Consiste en almacenar todas las componentes en un heap (o montículo) y luego extraer la cima del montón en sucesivas iteraciones, consiguiendo así el conjunto ordenado.
- *Expresión del Algoritmo:*

```

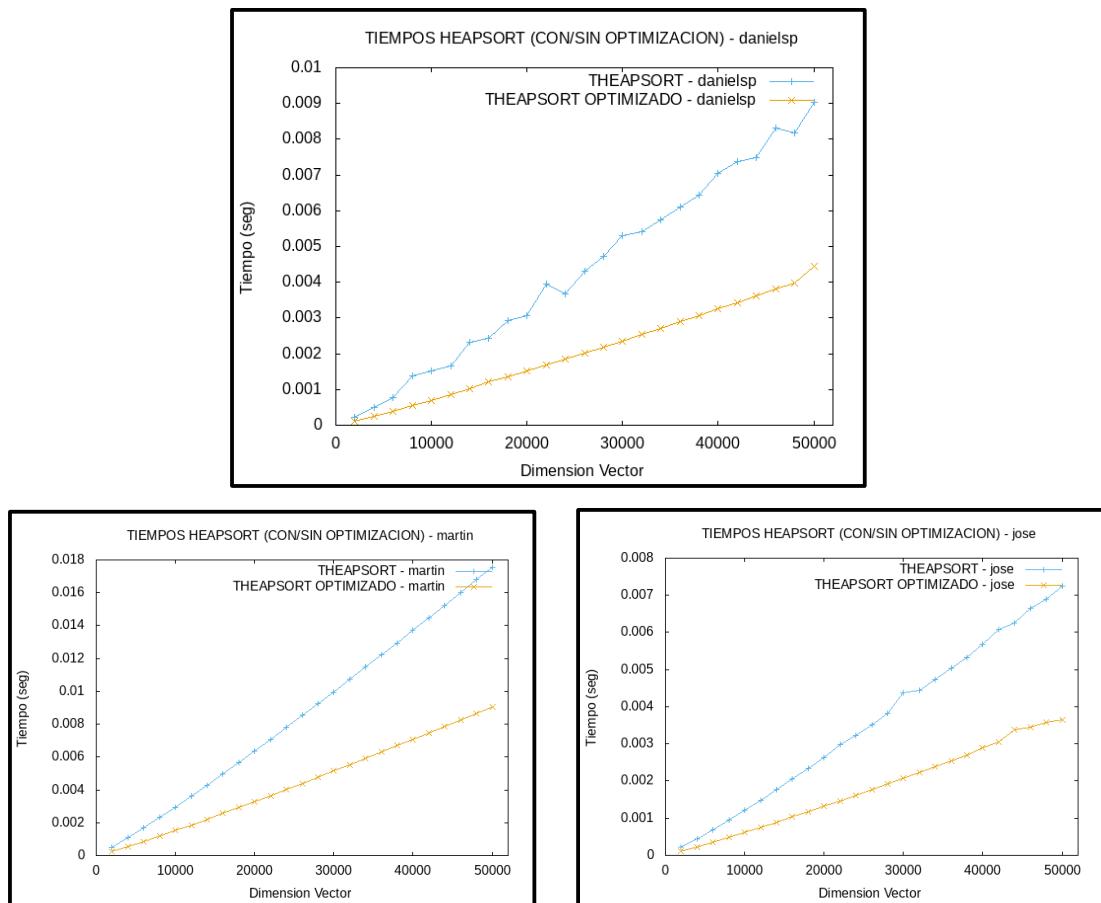
1 static void heapsort(int T[], int num_elem)
2 {
3     int i;
4     for (i = num_elem/2; i >= 0; i--)
5         reajustar(T, num_elem, i);
6     for (i = num_elem - 1; i >= 1; i--)
7     {
8         int aux = T[0];
9         T[0] = T[i];
10        T[i] = aux;
11        reajustar(T, i, 0);
12    }
13 }
14
15
16 static void reajustar(int T[], int num_elem, int k)
17 {
18     int j;
19     int v;
20     v = T[k];
21     bool esAPO = false;
22     while ((k < num_elem/2) && !esAPO)
23     {
24         j = k + k + 1;
25         if ((j < (num_elem - 1)) && (T[j] < T[j+1]))
26             j++;
27         if (v >= T[j])
28             esAPO = true;
29         T[k] = T[j];
30         k = j;
31     }
32     T[k] = v;
33 }
```

- ***Eficiencia Empírica:*** Siguiendo el procedimiento del **Punto 2** de este informe, se han obtenido los siguientes datos:

COMPARACIÓN DE TIEMPOS :: TIEMPOS HEAPSORT			
DIMENSIÓN	DANIELSP	MARTIN	JOSE
2000	0.0002273	0.0004956	0.0002184
4000	0.0004906	0.0010712	0.0004507
6000	0.0007676	0.0016762	0.0006911
8000	0.0013710	0.0023179	0.0009467
10000	0.0015282	0.0029488	0.0012157
12000	0.0016564	0.0036434	0.0014851
14000	0.0023096	0.0042910	0.0017672
16000	0.0024280	0.0049780	0.0020539
18000	0.0029352	0.0056739	0.0023333
20000	0.0030674	0.0063835	0.0026248
22000	0.0039598	0.0070751	0.0029734
24000	0.0036713	0.0078061	0.0032241
26000	0.0043153	0.0085335	0.0035088
28000	0.0047273	0.0092429	0.0038143
30000	0.0053057	0.0099684	0.0043819
32000	0.0054147	0.0107317	0.0044373
34000	0.0057499	0.0114944	0.0047372
36000	0.0060947	0.0122092	0.0050395
38000	0.0064328	0.0129497	0.0053213
40000	0.0070325	0.0137199	0.0056746
42000	0.0073665	0.0144619	0.0060847
44000	0.0074763	0.0152223	0.0062629
46000	0.0083200	0.0160098	0.0066429
48000	0.0081857	0.0168062	0.0068926
50000	0.0090245	0.0175453	0.0072407

COMPARACIÓN DE TIEMPOS :: TIEMPOS HEAPSORT OPTIMIZADO			
DIMENSIÓN	DANIELSP	MARTIN	JOSE
2000	0.0001180	0.0002569	0.0001035
4000	0.0002532	0.0005494	0.0002221
6000	0.0003959	0.0008605	0.0003516
8000	0.0005450	0.0011832	0.0004775
10000	0.0006969	0.0015176	0.0006114
12000	0.0008545	0.0018557	0.0007511
14000	0.0010129	0.0022076	0.0008889
16000	0.0012039	0.0025622	0.0010307
18000	0.0013403	0.0029200	0.0011728
20000	0.0015056	0.0032850	0.0013219
22000	0.0016725	0.0036462	0.0014662
24000	0.0018419	0.0040109	0.0016150
26000	0.0020134	0.0043886	0.0017655
28000	0.0021880	0.0047615	0.0019192
30000	0.0023613	0.0051481	0.0020679
32000	0.0025402	0.0055267	0.0022227
34000	0.0027123	0.0059275	0.0023789
36000	0.0028901	0.0062964	0.0025389
38000	0.0030715	0.0066908	0.0026876
40000	0.0032516	0.0070724	0.0028936
42000	0.0034261	0.0074728	0.0030532
44000	0.0036229	0.0078591	0.0033727
46000	0.0037984	0.0082576	0.0034404
48000	0.0039900	0.0086566	0.0035894
50000	0.0044454	0.0090666	0.0036385

- *Análisis gráfico:* En primer lugar, vamos a comparar los resultados que se obtienen optimizando el código y sin optimizar de todos los integrantes de este grupo.



Como era de esperar, los tiempos obtenidos tras optimizar el código son mejores que los que no lo están, y la diferencia se acentúa a medida que crecen los datos.

Ahora vamos a realizar el ajuste de la nube de puntos de cada integrante obtenida con el código sin optimizar. Basándonos en el estudio de la eficiencia teórica vamos realizar una aproximación a una función logarítmica, es decir, a una del tipo:

$$f(x) = ax \cdot \log(x) + b$$

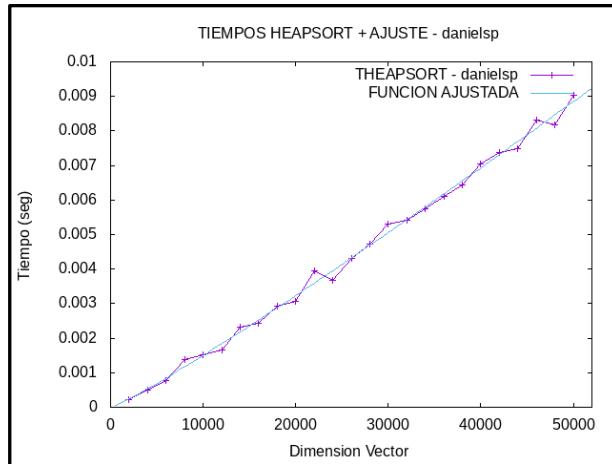
- Todos los apuntes que necesitas están aquí
- Al mejor precio del mercado, desde **2 cent**.
- Recoge los apuntes en tu copistería más cercana o recíbelos en tu casa
- Todas las anteriores son correctas

Algorítmica – SEMIGRUPO

P1: EFICIENCIA

Daniel | Jose | Martin

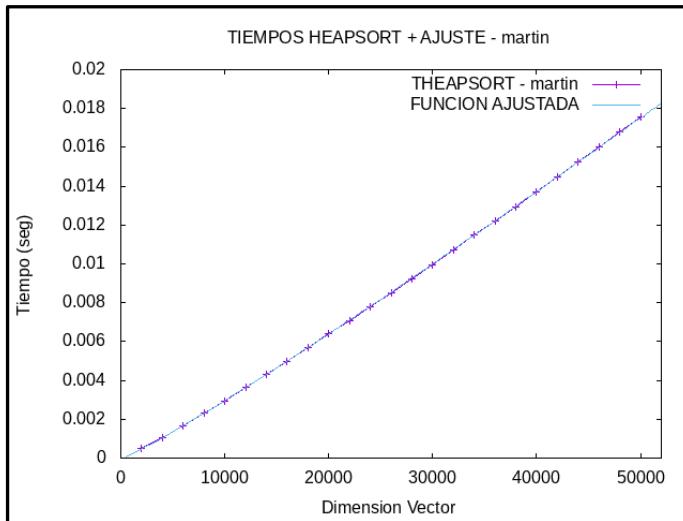
Imprimir 



La función de ajuste para estos datos es la siguiente:

$$f(x) = 1.63733 \cdot 10^{-8} \cdot x \cdot \log(x) - 7.72432 \cdot 10^{-6}$$

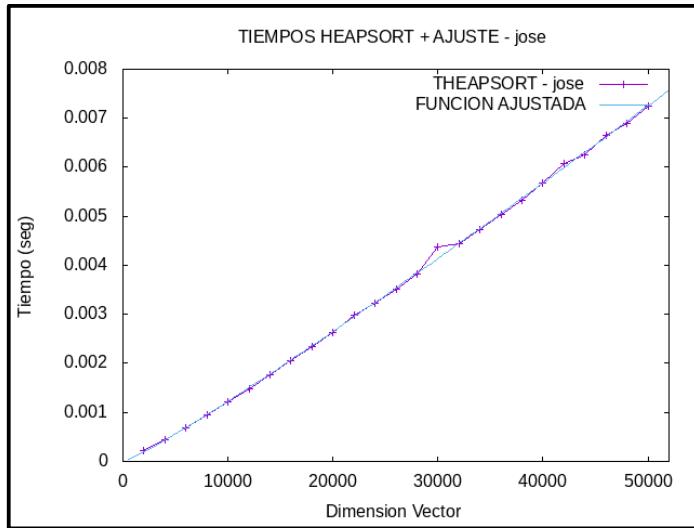
El coeficiente de regresión de este ajuste es:  $\eta = 0.99582204$ . Lo que nos indica que es un ajuste prácticamente excelente.



La función de ajuste para estos datos es la siguiente:

$$f(x) = 3.24438 \cdot 10^{-8} \cdot x \cdot \log(x) - 3.24668 \cdot 10^{-5}$$

El coeficiente de regresión de este ajuste es:  $\eta = 0.99998177$ . Lo que nos indica que es un ajuste prácticamente excelente.

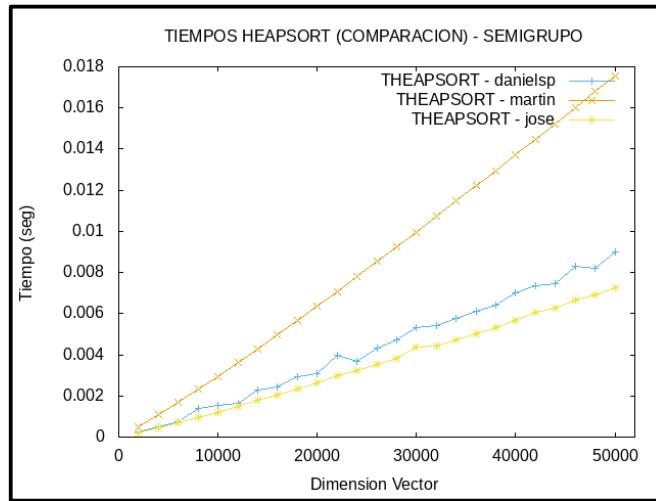


La función de ajuste para estos datos es la siguiente:

$$f(x) = 1.34255 \cdot 10^{-8} \cdot x \cdot \log(x) - 8.59276 \cdot 10^{-6}$$

El coeficiente de regresión de este ajuste es:  $\eta = 0.9992625$ . Lo que nos indica que es un ajuste prácticamente excelente.

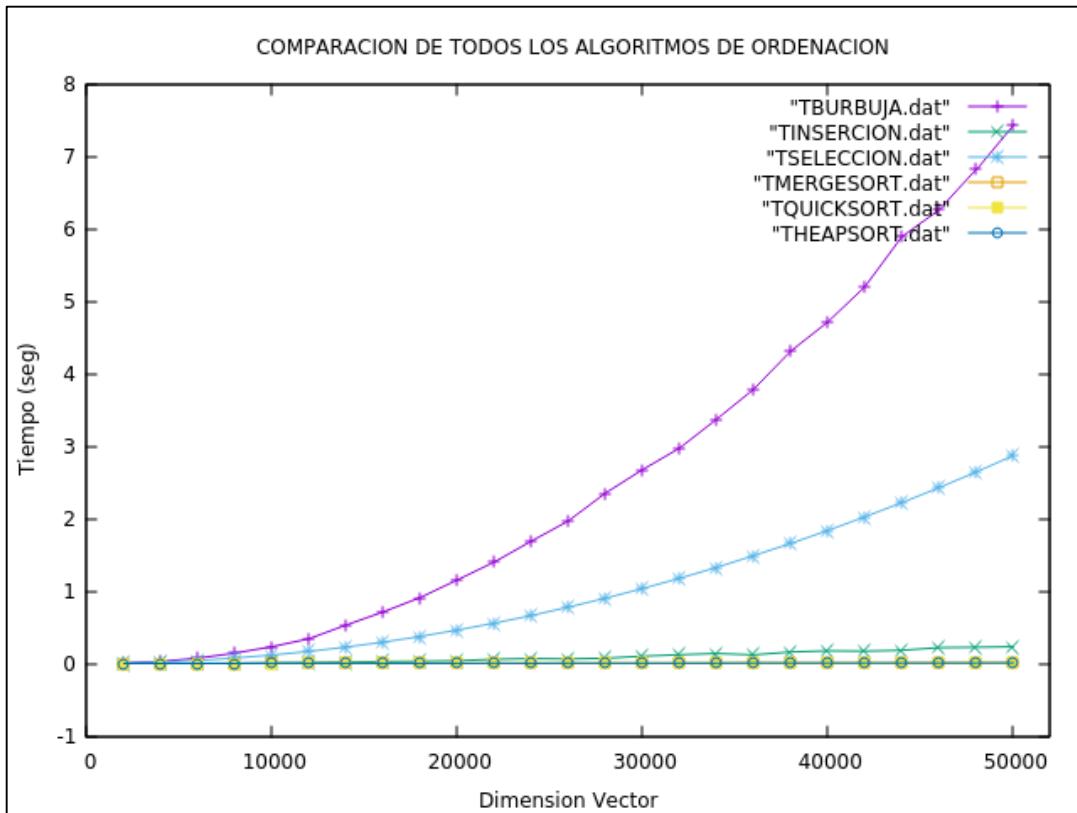
- **Comparación Final:** En base a la tabla resultante obtenida con los tiempos de todos los integrantes del grupo, se ha realizado la siguiente gráfica de comparación final.



De nuevo la CPU más lenta es la de Martín, que obtiene tiempos mucho mayores que los de José y Daniel, los cuales son cercanos, pero la CPU de José es algo más rápida.

### 3.9 COMPARACIÓN FINAL DE ALGORITMOS DE ORDENACIÓN

- Hemos visto que hay muchas maneras de ordenar un vector, todas y cada una resuelven este problema en un tiempo que puede ser razonable o no dependiendo de las necesidades del usuario. Como resumen final de este bloque se exponen gráficamente una comparación de todos los algoritmos de ordenación.



Podemos ver que peor algoritmo es el de burbuja en comparación con el resto. Es un tiempo bastante excesivo para el tamaño de vector indicado si lo comparamos con los de orden logarítmico.

Concluimos que la mejor opción para ordenar un conjunto, es utilizar cualquiera de los de orden  $O(n \cdot \log(n))$ . Como vencedor absoluto de esta sección podemos decir que Quicksort es el que consigue realizarlo en el menor tiempo posible. 😊

**NOTA: SE HA ESCOGIDO LOS DATOS DE DANIEL PARA REALIZAR ESTA COMPARACIÓN Y EVITAR REDUNDANCIAS. SIN EMBARGO, HEMOS COMPROBADO LOS DATOS DE LOS 3, Y SE OBTIENEN GRÁFICAS SIMILARES.**

## 4. BLOQUE 2: ALGORITMO DE FLOYD

- *Descripción:* Este algoritmo se encarga de calcular el mínimo coste que se produce al recorrer el camino entre un par de nodos o vértices de un grafo dirigido. Para ello utiliza una representación matricial del grafo y realiza los cálculos en función de la matriz obtenida.
- *Expresión del algoritmo:*

```

1 void Floyd(int **M, int dim)
2 {
3     for (int k = 0; k < dim; k++)
4         for (int i = 0; i < dim; i++)
5             for (int j = 0; j < dim; j++)
6             {
7                 int sum = M[i][k] + M[k][j];
8                 M[i][j] = (M[i][j] > sum) ? sum : M[i][j];
9             }
10 }
```

- *Eficiencia teórica:* Tenemos tres bucles anidados for. Cada uno se ejecuta n veces por lo que la eficiencia final será de O(n<sup>3</sup>).
- *Eficiencia Empírica:* Siguiendo el procedimiento del Punto 2 de este informe, se han obtenido los siguientes datos:

COMPARACIÓN DE TIEMPOS :: TIEMPOS FLOYD			
DIMENSIÓN	DANIELSP	MARTIN	JOSE
50	0.0006767	0.0014738	0.0006674
100	0.0050833	0.0109501	0.0054343
150	0.0167380	0.0363110	0.0158182
200	0.0391282	0.0852164	0.0356695
250	0.0761400	0.1661040	0.0672267
300	0.1306510	0.2862460	0.1158190
350	0.2247890	0.4534500	0.1832970
400	0.3423380	0.6745120	0.2724020
450	0.4382530	0.9593480	0.3875180
500	0.6531320	1.3126900	0.5299900
550	0.8492160	1.7545300	0.7050440
600	1.1394500	2.2672700	0.9118110
650	1.3139700	2.8789100	1.1610500
700	1.6940100	3.5930700	1.4464700
750	2.2250300	4.4180800	1.7989700
800	2.7025800	5.3576600	2.1538000
850	2.9409700	6.4177700	2.5928100
900	3.6957300	7.6466600	3.0611300
950	4.5253600	8.9646000	3.6037900
1000	5.1491200	10.4701000	4.2749600
1050	6.0332400	12.0839000	4.9189900
1100	6.9670600	13.9141000	5.5884400
1150	8.0597500	15.9021000	6.4061600
1200	9.2008600	18.0882000	7.2945100
1250	10.5509000	20.4183000	8.2616700
1300	11.9373000	22.9844000	9.2880000

- Todos los apuntes que necesitas están aquí
- Al mejor precio del mercado, desde **2 cent**.
- Recoge los apuntes en tu copistería más cercana o recíbelos en tu casa
- Todas las anteriores son correctas

Algorítmica – SEMIGRUPO

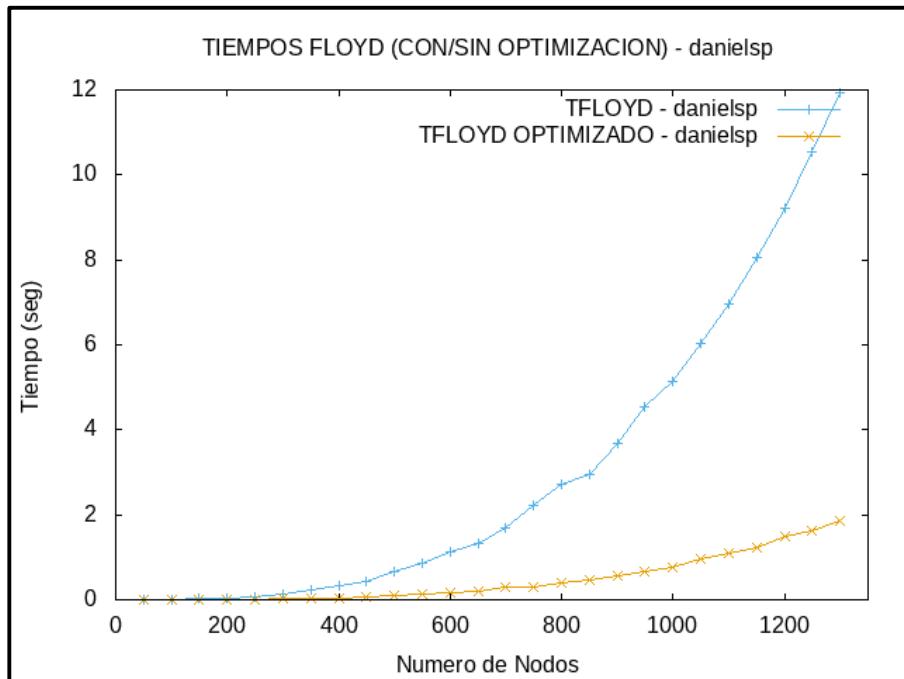
P1: EFICIENCIA

Daniel | Jose | Martin

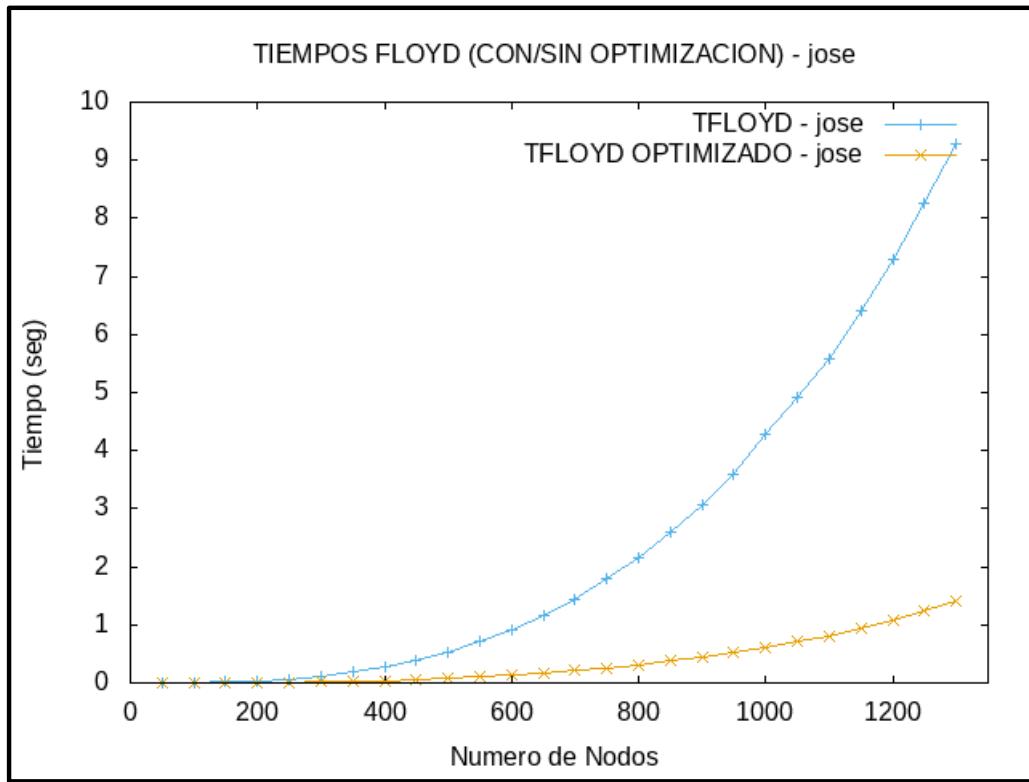
Imprimir 

COMPARACIÓN DE TIEMPOS :: TIEMPOS FLOYD OPTIMIZADO			
DIMENSIÓN	DANIELSP	MARTIN	JOSE
50	0.0001885	0.0001897	0.0000780
100	0.0014483	0.0017228	0.0006701
150	0.0026174	0.0053179	0.0026480
200	0.0055927	0.0123810	0.0053934
250	0.0108841	0.0239503	0.0099988
300	0.0185994	0.0411540	0.0177216
350	0.0294229	0.0651999	0.0268827
400	0.0437652	0.0971924	0.0403324
450	0.0622568	0.1378430	0.0570792
500	0.0981390	0.1885930	0.0780046
550	0.1464070	0.2495450	0.1028040
600	0.1509530	0.3239980	0.1328290
650	0.1861820	0.4095410	0.1692980
700	0.2834710	0.5118420	0.2194310
750	0.2836850	0.6279020	0.2509390
800	0.3984190	0.7619370	0.3048220
850	0.4726120	0.9132640	0.3747000
900	0.5547110	1.0810300	0.4400790
950	0.6529390	1.2752200	0.5143830
1000	0.7762690	1.4849600	0.6156330
1050	0.9664430	1.7247700	0.7145130
1100	1.0830400	1.9931800	0.8128570
1150	1.2307600	2.2804200	0.9363750
1200	1.5035500	2.6466500	1.0709900
1250	1.6085100	3.0021300	1.2296900
1300	1.8621700	3.3714300	1.4030300

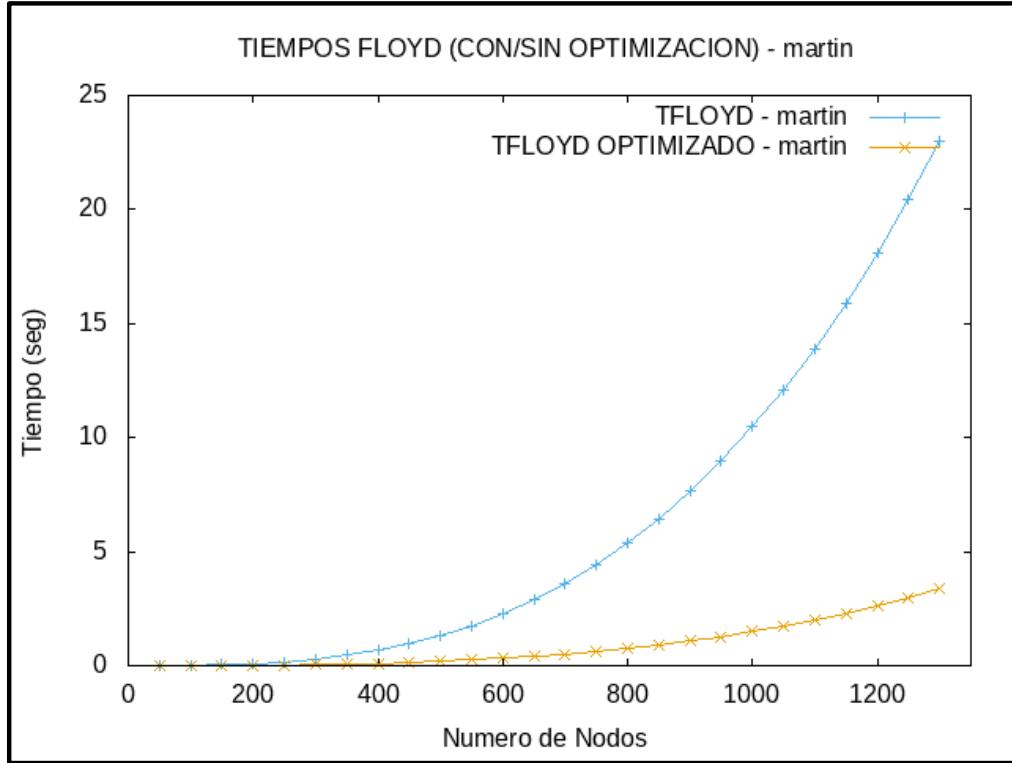
- *Análisis gráfico:* En primer lugar, vamos a comparar los resultados que se obtienen optimizando el código y sin optimizar de todos los integrantes de este grupo.



*COMENTARIO: La reducción de los tiempos con el código optimizado es más que considerable, hasta un total de 10 segundos se consigue decrementar el tiempo de ejecución para el valor máximo.*



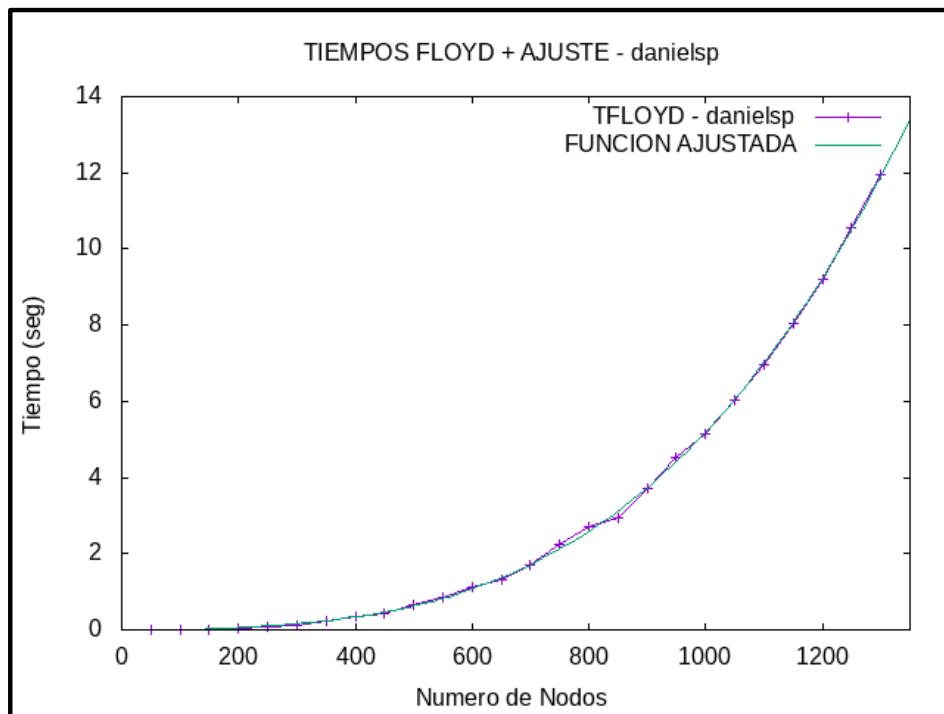
*COMENTARIO: Al igual que sucedía en la gráfica anterior. La reducción del tiempo de ejecución es importante para tamaños grandes.*



*COMENTARIO: Sigue exactamente lo mismo que en los otros dos casos. Sin embargo, aquí el cambio es muchísimo más significativo. ¡¡La diferencia para el tamaño más grande es de más de 20 segundos!!*

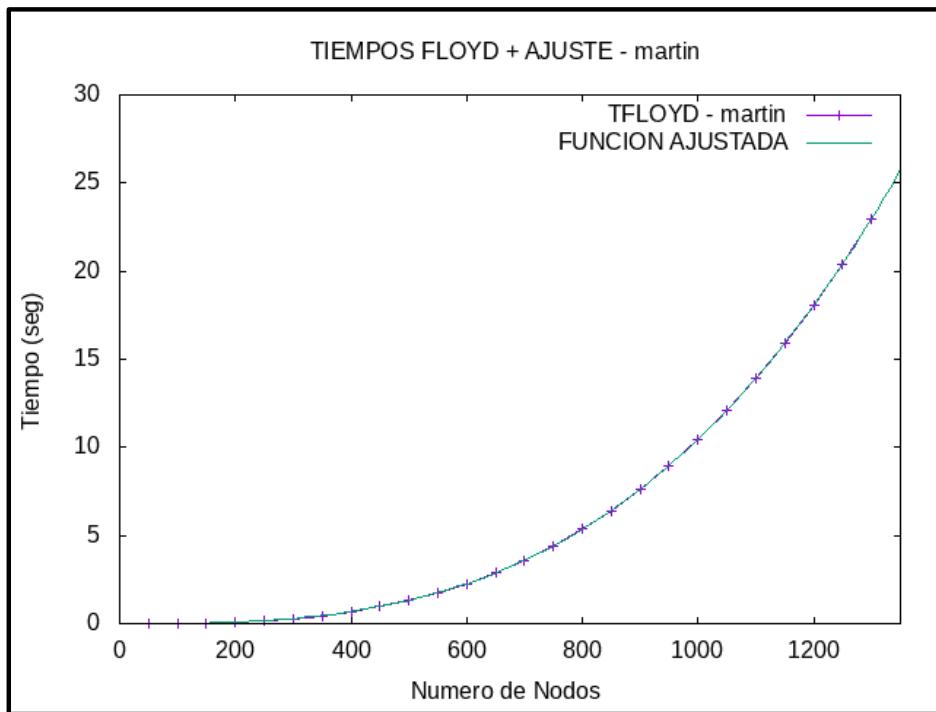
Ahora vamos a realizar el ajuste de la nube de puntos de cada integrante obtenida con el código sin optimizar. Basándonos en el estudio de la eficiencia teórica vamos realizar una aproximación a una función cúbica, es decir, a una del tipo:

$$f(x) = ax^3 + bx^2 + cx + d$$



La función de ajuste para estos datos es la siguiente:  
 $f(x) = 6.964 \cdot 10^{-9}x^3 - 2.888 \cdot 10^{-6}x^2 + 0.001x - 0.138$

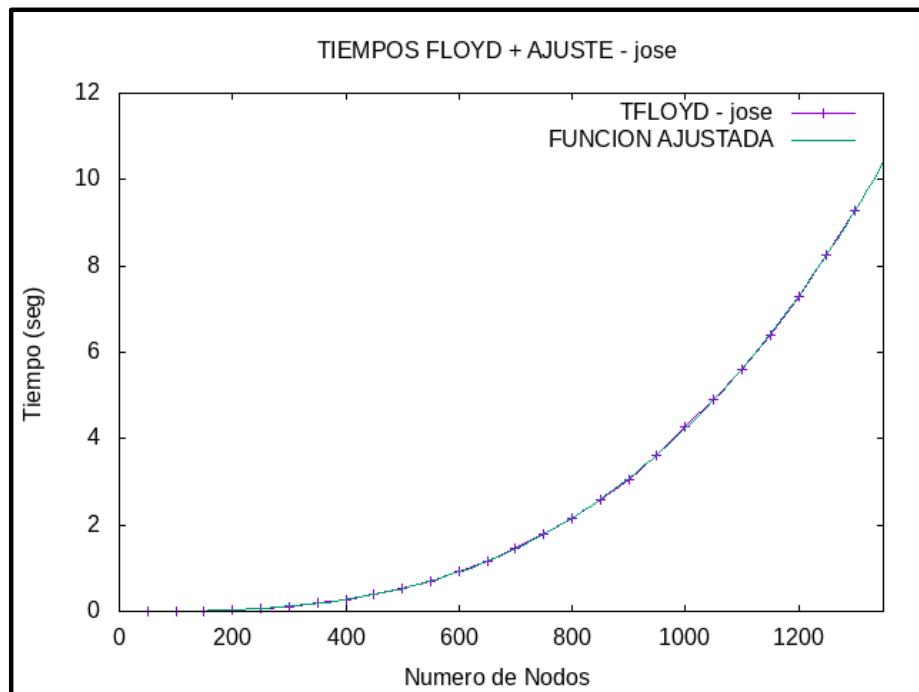
El coeficiente de regresión de este ajuste es:  $\eta = 0.99966658$ . Lo que nos indica que es un ajuste prácticamente excelente.



La función de ajuste para estos datos es la siguiente:

$$f(x) = 1.05032 \cdot 10^{-8}x^3 - 1.07683 \cdot 10^{-7}x^2 + 7.03799 \cdot 10^{-5} \cdot x - 0.00657949$$

El coeficiente de regresión de este ajuste es:  $\eta = 0.99999846$ . Lo que nos indica que es un ajuste prácticamente excelente.



- Todos los apuntes que necesitas están aquí
- Al mejor precio del mercado, desde **2 cent**.
- Recoge los apuntes en tu copistería más cercana o recíbelos en tu casa
- Todas las anteriores son correctas

Algorítmica – SEMIGRUPO

P1: EFICIENCIA

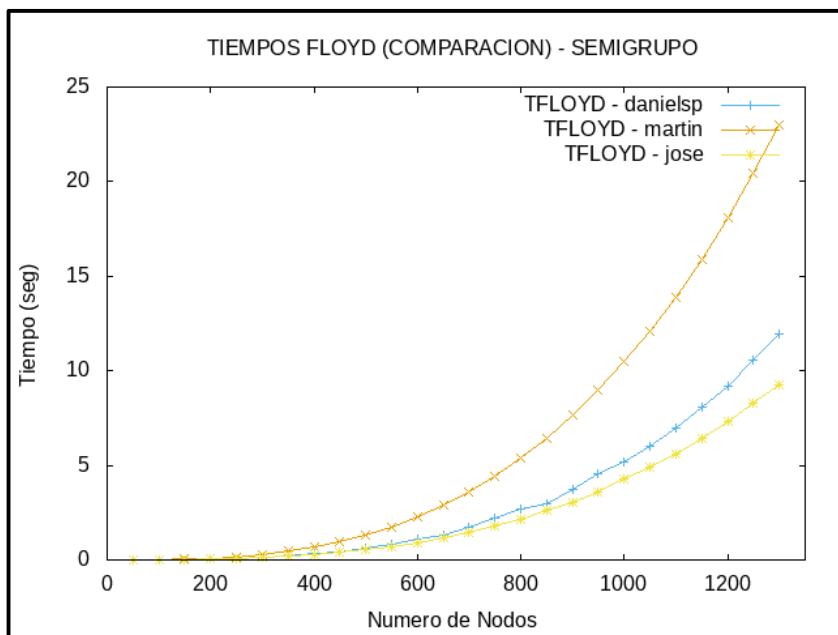
Daniel | Jose | Martin

La función de ajuste para estos datos es la siguiente:

$$f(x) = 4.24604 \cdot 10^{-9}x^3 - 4.30338 \cdot 10^{-8}x^2 + 2.20239 \cdot 10^{-5} \cdot x - 0.00105608$$

El coeficiente de regresión de este ajuste es:  $\eta = 0.99996551$ . Lo que nos indica que es un ajuste prácticamente excelente.

- *Comparación Final:* En base a la tabla resultante obtenida con los tiempos de todos los integrantes del grupo, se ha realizado la siguiente gráfica de comparación final.



*COMENTARIO:* Podemos ver que el que menos tarda es Jose, seguido de Dani y por último de Martín, habiendo una diferencia bastante importante entre la gráfica de Martín con respecto a las otras 2, que se distancian a partir de los 950 nodos.

## 5. BLOQUE 3: ALGORITMO DE HANOI

- *Descripción:* Este algoritmo resuelve el tan famoso juego de las Torres de Hanoi. En el que consiste en desplazar una pirámide de discos apilados de diferentes diámetros de una columna a otra, con la restricción de que no se puede situar un disco de diámetro más grande sobre otro que tiene un diámetro más pequeño. La resolución que se propone es realizar la resolución del problema para **n-1** discos, y así recursivamente hasta obtener el caso más trivial posible.
- *Expresión del Algoritmo:*

```

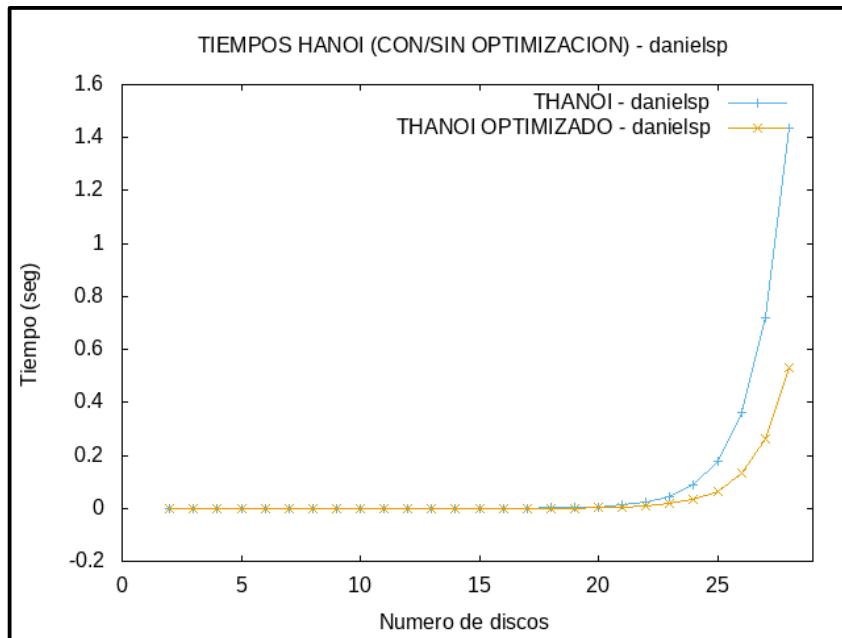
1 void hanoi (int M, int i, int j)
2 {
3     if (M > 0)
4     {
5         hanoi (M-1, i, 6-i-j);
6         cout << i << " -> " << j << endl;
7         hanoi (M-1, 6-i-j, j);
8     }
9 }
```

- *Eficiencia Teórica:* Para 2 discos serán necesarios un total de 3 movimientos, para 3 discos necesitaremos 7 movimientos, para 4 discos 15 movimientos... Obtenemos así una fórmula para el algoritmo que atiende a la forma:  $M(n) = 2^n - 1$ , que nos indica que la eficiencia es de  $O(2^n)$ .
- *Eficiencia Empírica:* Siguiendo el procedimiento del Punto 2 de este informe, se han obtenido los siguientes datos:

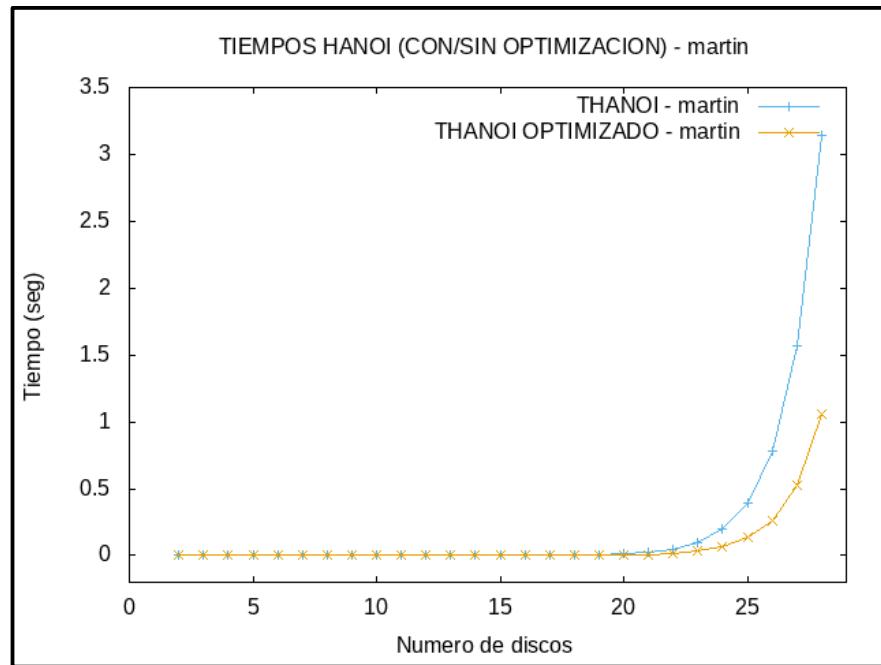
COMPARACIÓN DE TIEMPOS :: TIEMPOS HANOI			
DIMENSIÓN	DANIELSP	MARTIN	JOSE
2	0.0000007	0.0000015	0.0000007
3	0.0000006	0.0000012	0.0000008
4	0.0000006	0.0000030	0.0000007
5	0.0000008	0.0000017	0.0000007
6	0.0000013	0.0000020	0.0000010
7	0.0000014	0.0000029	0.0000014
8	0.0000020	0.0000044	0.0000020
9	0.0000033	0.0000076	0.0000035
10	0.0000061	0.0000134	0.0000067
11	0.0000127	0.0000253	0.0000114
12	0.0000248	0.0000516	0.0000226
13	0.0000483	0.0000973	0.0000444
14	0.0000962	0.0001931	0.0000888
15	0.0001984	0.0003856	0.0001774
16	0.0004224	0.0007710	0.0003553
17	0.0007719	0.0015483	0.0007054
18	0.0014349	0.0030725	0.0013485
19	0.0028604	0.0061442	0.0026741
20	0.0056495	0.0122925	0.0051210
21	0.0113505	0.0245375	0.0101234
22	0.0225182	0.0490613	0.0198098
23	0.0449603	0.0981136	0.0395590
24	0.0898702	0.1962000	0.0794563
25	0.1799030	0.3925080	0.1581470
26	0.3597550	0.7848810	0.3160780
27	0.7187450	1.5699900	0.6323490
28	1.4376100	3.1410300	1.2667800

COMPARACIÓN DE TIEMPOS :: TIEMPOS FLOYD OPTIMIZADO			
DIMENSIÓN	DANIELSP	MARTIN	JOSE
50	0.0001885	0.0001897	0.0000780
100	0.0014483	0.0017228	0.0006701
150	0.0026174	0.0053179	0.0026480
200	0.0055927	0.0123810	0.0053934
250	0.0108841	0.0239503	0.0099988
300	0.0185994	0.0411540	0.0177216
350	0.0294229	0.0651999	0.0268827
400	0.0437652	0.0971924	0.0403324
450	0.0622568	0.1378430	0.0570792
500	0.0981390	0.1885930	0.0780046
550	0.1464070	0.2495450	0.1028040
600	0.1509530	0.3239980	0.1328290
650	0.1861820	0.4095410	0.1692980
700	0.2834710	0.5118420	0.2194310
750	0.2836850	0.6279020	0.2509390
800	0.3984190	0.7619370	0.3048220
850	0.4726120	0.9132640	0.3747000
900	0.5547110	1.0810300	0.4400790
950	0.6529390	1.2752200	0.5143830
1000	0.7762690	1.4849600	0.6156330
1050	0.9664430	1.7247700	0.7145130
1100	1.0830400	1.9931800	0.8128570
1150	1.2307600	2.2804200	0.9363750
1200	1.5035500	2.6466500	1.0709900
1250	1.6085100	3.0021300	1.2296900
1300	1.8621700	3.3714300	1.4030300

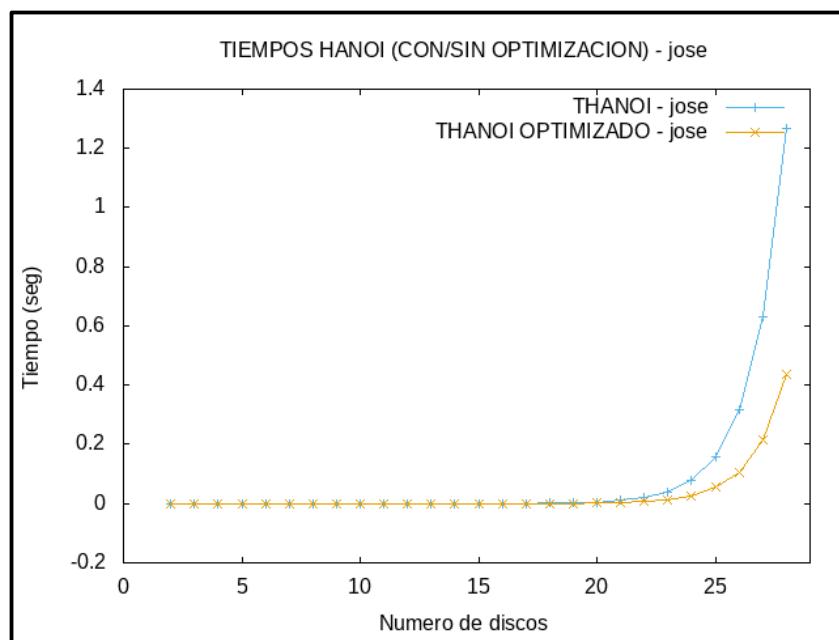
- *Análisis gráfico:* En primer lugar, vamos a comparar los resultados que se obtienen optimizando el código y sin optimizar de todos los integrantes de este grupo.



*COMENTARIO: No se aprecia gran diferencia entre ambos códigos salvo en los 3 últimos datos introducidos donde se empieza a disparar el tiempo para el código no optimizado.*



*COMENTARIO: Sigue lo mismo que en la gráfica anterior. Salvo en el tramo final, apenas se nota diferencia entre códigos.*



*COMENTARIO: Ocurre igual que en los anteriores casos.*

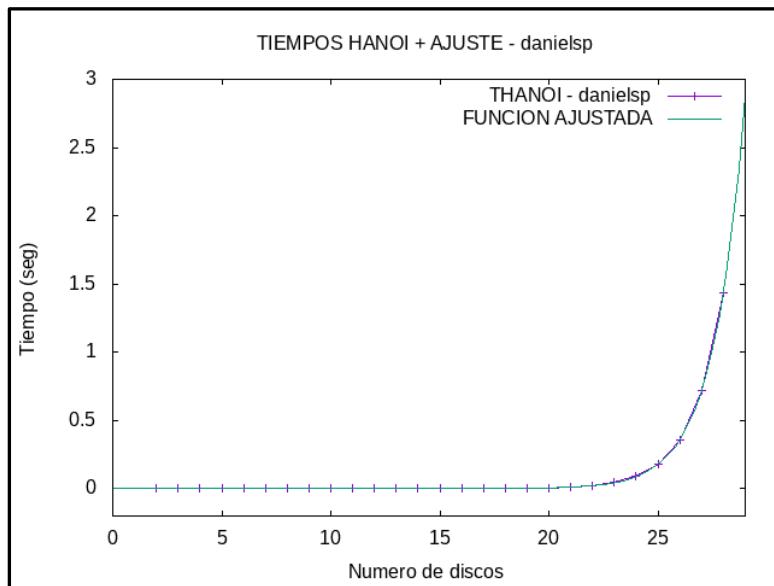
- Todos los apuntes que necesitas están aquí
- Al mejor precio del mercado, desde **2 cent**.
- Recoge los apuntes en tu copistería más cercana o recíbelos en tu casa
- Todas las anteriores son correctas

Algorítmica – SEMIGRUPO

P1: EFICIENCIA

Daniel | Jose | Martin

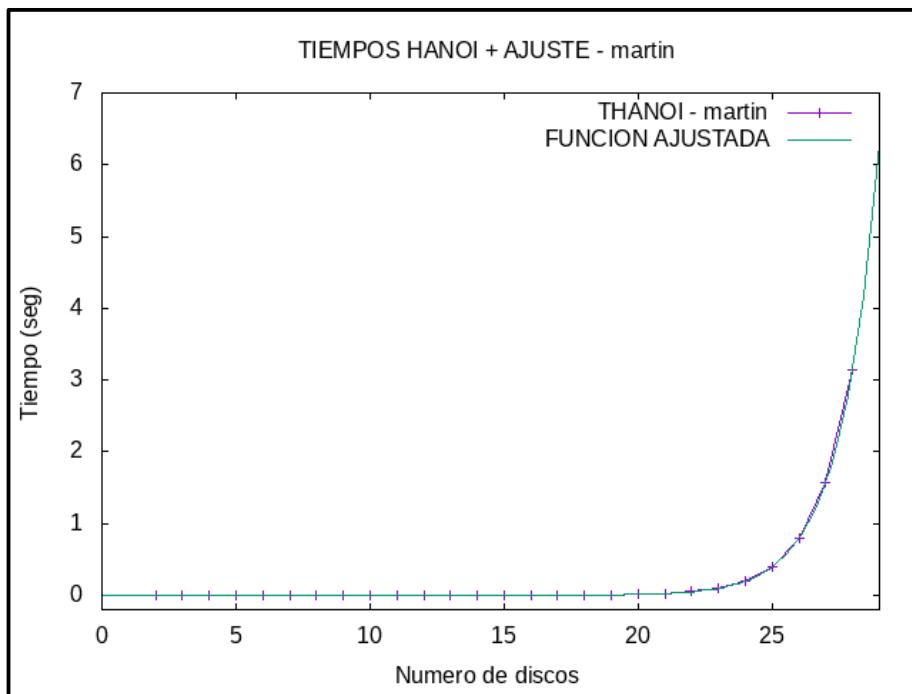
Ahora vamos a realizar el ajuste de la nube de puntos de cada integrante obtenida con el código sin optimizar. Basándonos en el estudio de la eficiencia teórica vamos realizar una aproximación a una función exponencial de base 2, es decir, a una del tipo:



La función de ajuste para estos datos es la siguiente:

$$f(x) = 5.356 \cdot 10^{-9} \cdot 2^x + 3.737 \cdot 10^{-5}$$

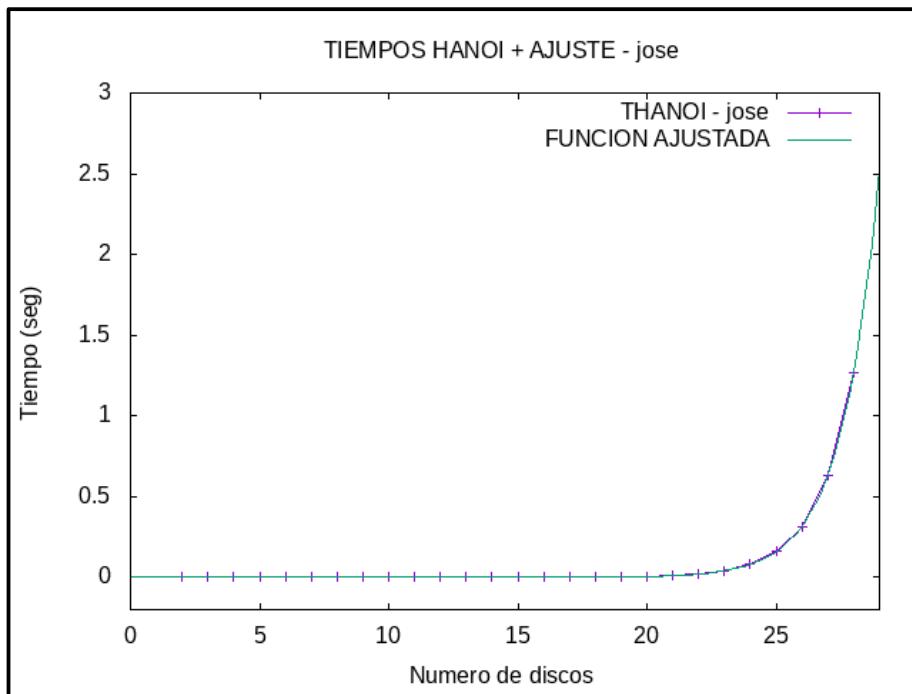
El coeficiente de regresión de este ajuste es:  $\eta = 0.99999993$ . Lo que nos indica que es un ajuste prácticamente excelente.



La función de ajuste para estos datos es la siguiente:

$$f(x) = 1.17003 \cdot 10^{-8} \cdot 2^x - 2.28694 \cdot 10^{-5}$$

El coeficiente de regresión de este ajuste es:  $\eta = 0.99999997$ . Lo que nos indica que es un ajuste prácticamente excelente.

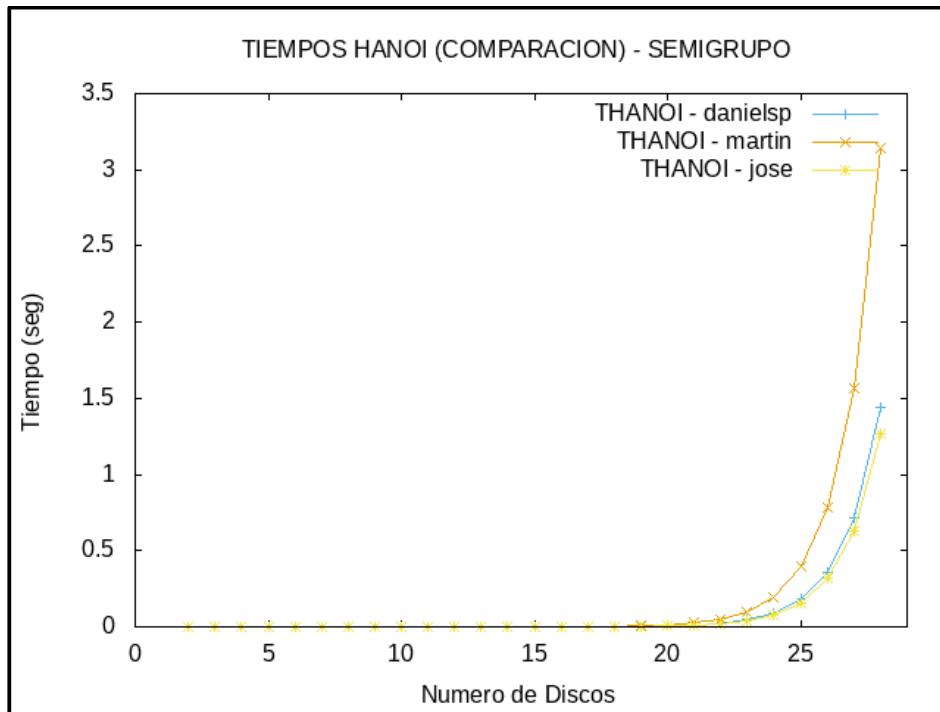


La función de ajuste para estos datos es la siguiente:

$$f(x) = 1.37833 \cdot 10^{-6} \cdot 2^x - 2.60871 \cdot 10^{-13}$$

El coeficiente de regresión de este ajuste es:  $\eta = 0.99999925$ . Lo que nos indica que es un ajuste prácticamente excelente.

- *Comparación Final:* En base a la tabla resultante obtenida con los tiempos de todos los integrantes del grupo, se ha realizado la siguiente gráfica de comparación final.

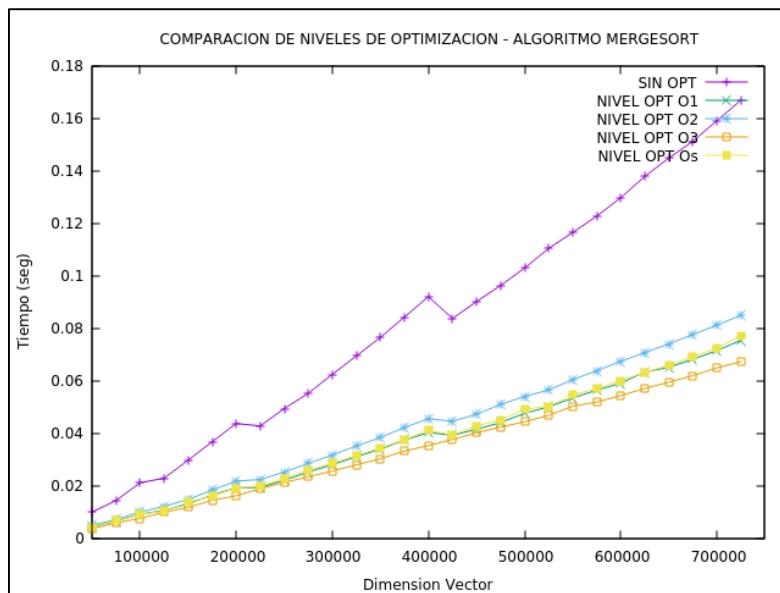
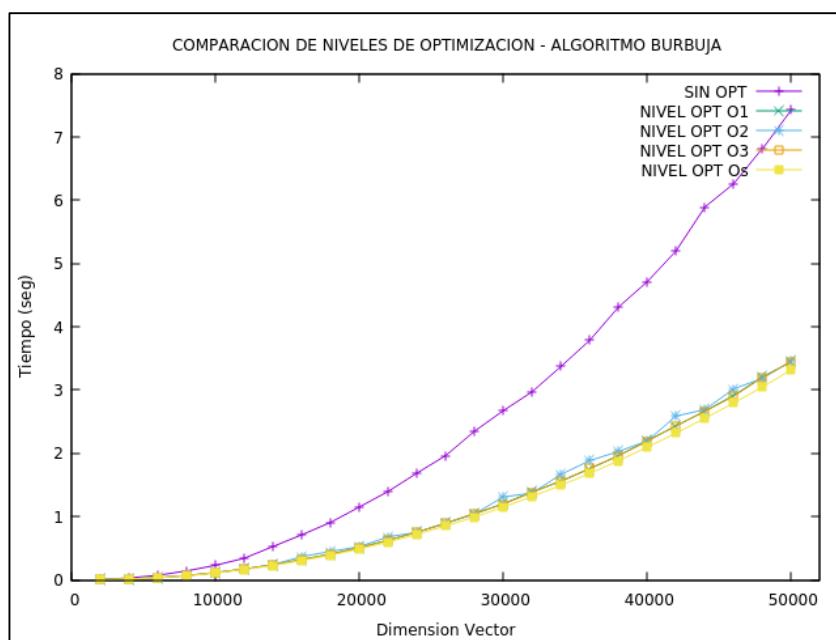


*COMENTARIO: Vemos que las gráficas de Jose y Dani están muy cercas, siendo ligeramente más rápido el de Jose. Sin embargo, se nota la diferencia con respecto la de Martín, duplicando el tiempo de ejecución para 28 discos.*

## 6. COMPARACIÓN DE NIVELES DE OPTIMIZACIÓN

**NOTA: SE HAN COMPARADO LOS TIEMPOS DE DANIEL.**

- Está claro que cuando se compilan los programas con optimización se reduce notoriamente el tiempo de ejecución, especialmente se nota para tamaños suficientemente grandes. La pregunta que nos cabe hacer es: ¿Cuál es mejor nivel de optimización para que el algoritmo sea lo más rápido posible? ¿Compilar en Os es mejor que con O3 realmente?
- Para esta sección se han compilado con diferentes niveles de optimización el algoritmo de la burbuja y el algoritmo de mergesort, y se han representado conjuntamente para observar la diferencia. Aquí se muestran los resultados:



- Todos los apuntes que necesitas están aquí
- Al mejor precio del mercado, desde **2 cent.**
- Recoge los apuntes en tu copistería más cercana o recíbelos en tu casa
- Todas las anteriores son correctas

Algorítmica – SEMIGRUPO

P1: EFICIENCIA

Daniel | Jose | Martin

## 7. CONCLUSIONES FINALES

- Con respecto a los algoritmos de ordenación, hemos comprobado que realmente se nota la diferencia entre uno que posee eficiencia cuadrática con otro que posee eficiencia logarítmica. A la hora de elegir, si lo que queremos es una respuesta inmediata lo mejor será optar por Mergesort, Quicksort o Heapsort, siempre y cuando estemos hablando de dimensiones muy grandes. Si no, será suficiente considerar Inserción, Selección o Burbuja. Podríamos establecer el límite en 2000 elementos.
- También hemos visto que la optimización de los códigos consigue muchísimo reducir los tiempos de ejecución como se ha indicado en otros apartados. Y en referencia al apartado 6 se puede concluir que el mejor nivel de optimización no necesariamente hace que la ejecución sea la más rápida. Por ejemplo, en la gráfica de Mergesort el nivel que menos tarda en terminar es el 3, seguido del 1. En el método de la Burbuja sí que el nivel de optimización **s** es el que tarda menos, pero es una diferencia apenas observable con respecto a cualquier otro.
- Con respecto a Floyd, es un algoritmo de orden cúbico como se mencionó. A partir de 1350 nodos la tardanza sin optimizar es bastante grande.
- Con Hanoi, hasta 10 discos los tiempos eran prácticamente 0, de 10 a 28 son tiempos rápidos también. Pero a partir de esa cifra la tardanza aumenta de manera abismal. En el informe no se deja constancia de lo que sucede con 30 discos, pero si se realizase la prueba ¡¡estaría entre 3 y 5 minutos!!
- En general, poniendo en común todo lo que hemos hecho entre los 3, podemos decir que aunque haya variaciones en los tiempos de ejecución, la eficiencia sigue siendo **EXACTAMENTE LA MISMA**, no varía si se realizan las pruebas en el ordenador de Martín, en el de Dani o en el de Jose.

En definitiva, es todo un reto conseguir crear e implementar algoritmos que sean rigurosos, estables y sobre todo que sean capaces de dar solución en tiempo real. Para ciertos problemas todavía no es posible esto, pero quién sabe, quizás dentro de unos años alguien sea capaz de escribir un programa que resuelva Hanoi o Floyd con una eficiencia de  $n \cdot \log(n)$ , o una ordenación de vectores de eficiencia  $n$ .

