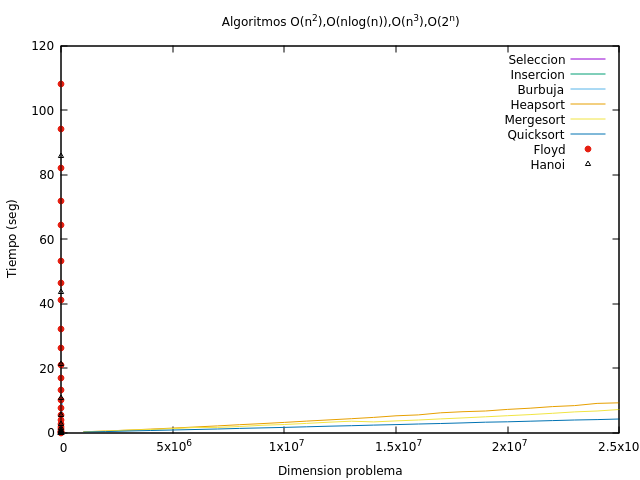
**ALGORÍTMICA UGR (2019-2020)**

***Práctica 1:***

***Análisis de Eficiencia de Algoritmos***



***Trabajo realizado por: Jose Luis Pedraza Román***

***Contenidos:***

1. ***Introducción***
2. ***Algoritmos de Ordenación O(n2)***
3. ***Algoritmos de Ordenación O(nlog(n))***
4. ***Comparativa Algoritmos de Ordenación***
5. ***Comparativa Algoritmos de Ordenación con y sin Optimización***
6. ***Algoritmo de Floyd***
7. ***Algoritmo de las Torres de Hanoi***
8. ***Introducción:***

En esta práctica se analizan diferentes algoritmos, estos son algunos algoritmos de ordenación, el algoritmo de Floyd y el algoritmo que resuelve el juego de las torres de Hanoi.

Para llevar acabo dicho estudio se han realizado los análisis empíricos e híbridos de cada uno de ellos.

En el análisis empírico se han hecho varias ejecuciones y calculado la media de los tiempos de ejecución (posteriormente explicados) para graficar los resultados y poder sacar conclusiones comparando algoritmos del mismo tipo y probando con diferentes opciones de compilación, como son –O2 y –O3 las cuales aplican un factor de optimización al código. También decir que se ha usado la librería “chrono” (incluir la opción de compilación –std=gnu++0x) para medir los tiempos de ejecución de cada algoritmo, incluyéndola dentro de los códigos y modificándolos sustancialmente para calcular la media de los tiempos de 3 ejecuciones para cada entrada del programa y para poder ejecutarlos a través de un script.sh y así automatizar las entradas del programa como se muestra a continuación:

#!/bin/bash

#./script.sh ejecutable liminf limsup intervalo

#./script.sh burbuja 1000 50000 2000

#el script genera un fichero .dat con los resultados y con nombre salida(loquesea).dat para que este identificado

# $2 $3 $4

# burbuja: 1000 a 63500 de 2500 en 2500

# insercion: 1000 a 63500 de 2500 en 2500

# seleccion: 1000 a 63500 de 2500 en 2500

# mergesort: 1000000 a 26000000 de 1000000 en 1000000

# quicksort: 1000000 a 26000000 de 1000000 en 1000000

# heapsort: 1000000 a 26000000 de 1000000 en 1000000

# floyd: 120 a 3000 de 120 en 120

# hanoi: 15 a 40 de 1 en 1

echo "" >> salida$1.dat

let i=$2

while (($i < $3))

do

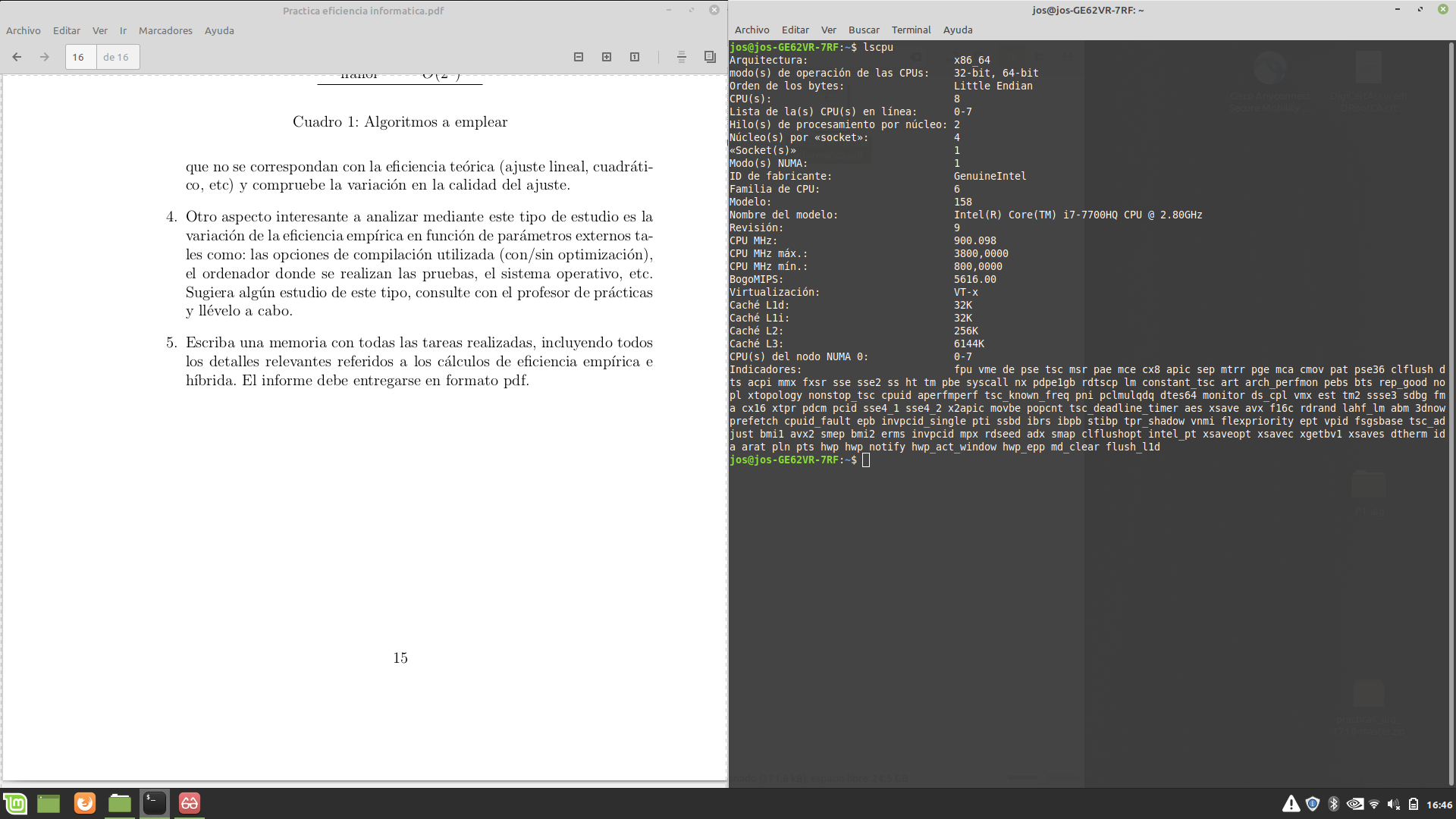
./$1 $i >> salida$1.dat

let i+=$4

done

En el análisis híbrido se ha ajustado cada una de las gráficas que representan los tiempos de ejecución de los algoritmos para diferentes tamaños de problema con la función correspondiente a su orden de eficiencia.

También cabe destacar el equipo en donde se han realizado las pruebas:



Y que para las mismas no se ha estado realizando ninguna tarea a la vez, por lo que se ha aprovechado prácticamente en su totalidad la potencia del equipo para obtener unos resultados correctos.

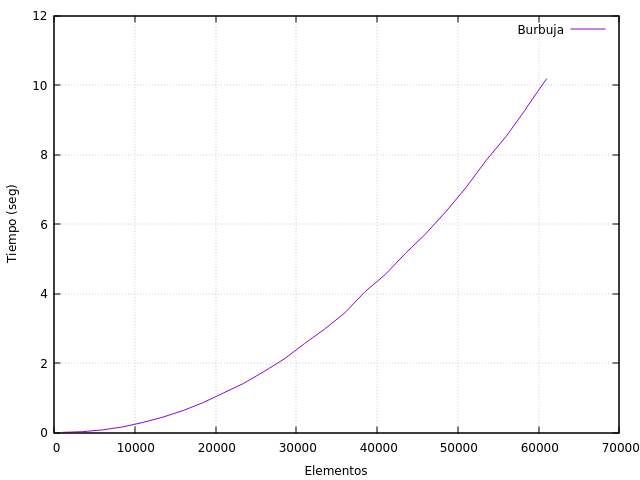
1. ***Algoritmos de Ordenación de Orden O(n2)***

Para este tipo de algoritmos con una eficiencia teórica de orden O(n2) se han obtenido los siguientes tiempos de ejecución en función del tamaño de la entrada del problema, en este caso, el número de componentes del vector a ordenar comenzando en 1000, de 2500 en 2500, hasta 61000.

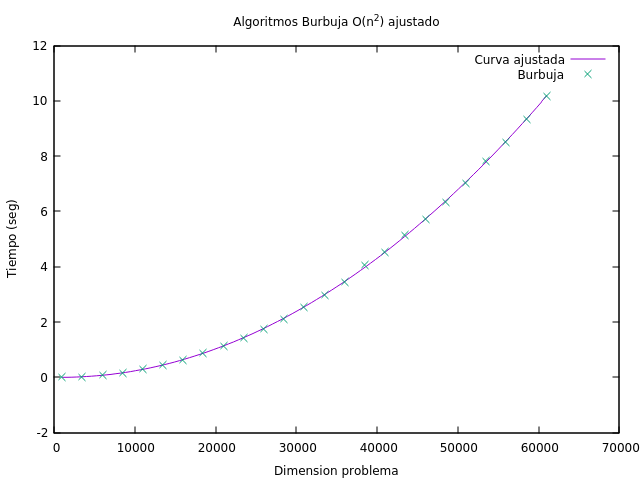
|  |  |  |  |
| --- | --- | --- | --- |
| Orden de eficiencia O(n2) | | | |
| Dimensión (elementos) | Burbuja | Inserción | Selección |
| 1000 | 0.00233345 | 0.00161735 | 1.11161e-09 |
| 3500 | 0.0220215 | 0.0107103 | 0.0133117 |
| 6000 | 0.0741775 | 0.0304569 | 0.0387415 |
| 8500 | 0.162025 | 0.0609614 | 0.0774742 |
| 11000 | 0.288338 | 0.10288 | 0.129626 |
| 13500 | 0.441998 | 0.154063 | 0.195214 |
| 16000 | 0.632698 | 0.22659 | 0.273711 |
| 18500 | 0.864927 | 0.295406 | 0.36757 |
| 21000 | 1.14052 | 0.375318 | 0.471659 |
| 23500 | 1.4169 | 0.468104 | 0.590739 |
| 26000 | 1.75495 | 0.57592 | 0.723723 |
| 28500 | 2.11671 | 0.693095 | 0.867589 |
| 31000 | 2.55638 | 0.834557 | 1.02573 |
| 33500 | 2.97524 | 0.956152 | 1.19957 |
| 36000 | 3.44262 | 1.10781 | 1.38369 |
| 38500 | 4.05224 | 1.26848 | 1.55658 |
| 41000 | 4.54637 | 1.49898 | 1.79299 |
| 43500 | 5.15496 | 1.60774 | 2.0181 |
| 46000 | 5.71493 | 1.79192 | 2.25633 |
| 48500 | 6.35276 | 1.9982 | 2.46992 |
| 51000 | 7.0463 | 2.34461 | 2.83972 |
| 53500 | 7.83095 | 2.48081 | 3.08651 |
| 56000 | 8.52961 | 2.67549 | 3.34637 |
| 58500 | 9.33606 | 2.9197 | 3.61743 |
| 61000 | 10.1715 | 3.3904 | 3.96781 |

A continuación, se muestran las gráficas obtenidas con gnuplot y su ajuste, donde se puede apreciar claramente la función característica de los algoritmos cuadráticos.

* + 1. ***Algoritmo Burbuja***



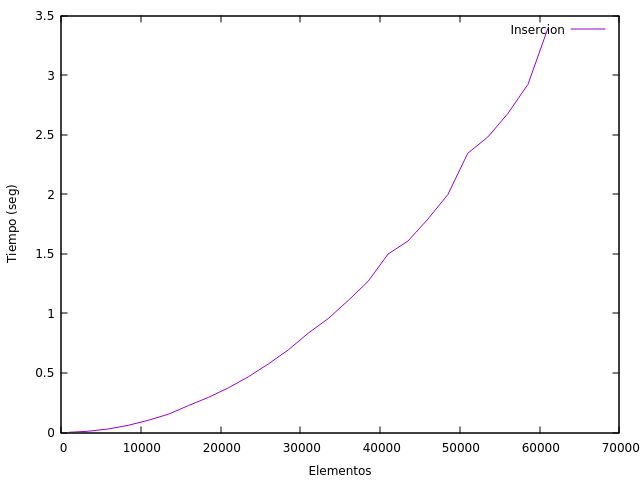
Enfoque híbrido: Burbuja ajustado a T(n) = a0\*n2 + a1\*n + a2



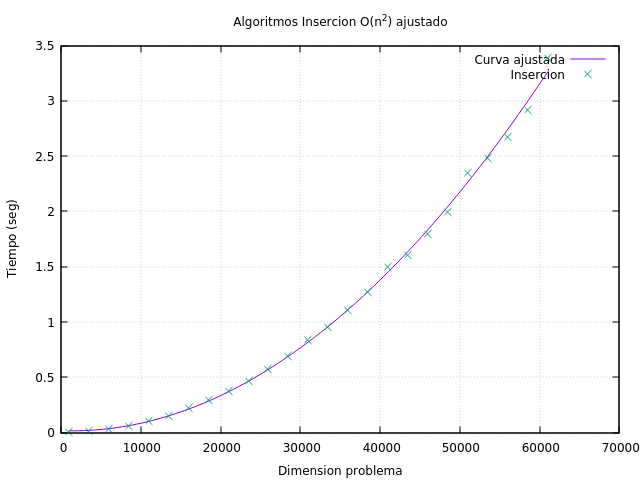
Valor de las constantes tras el ajuste:



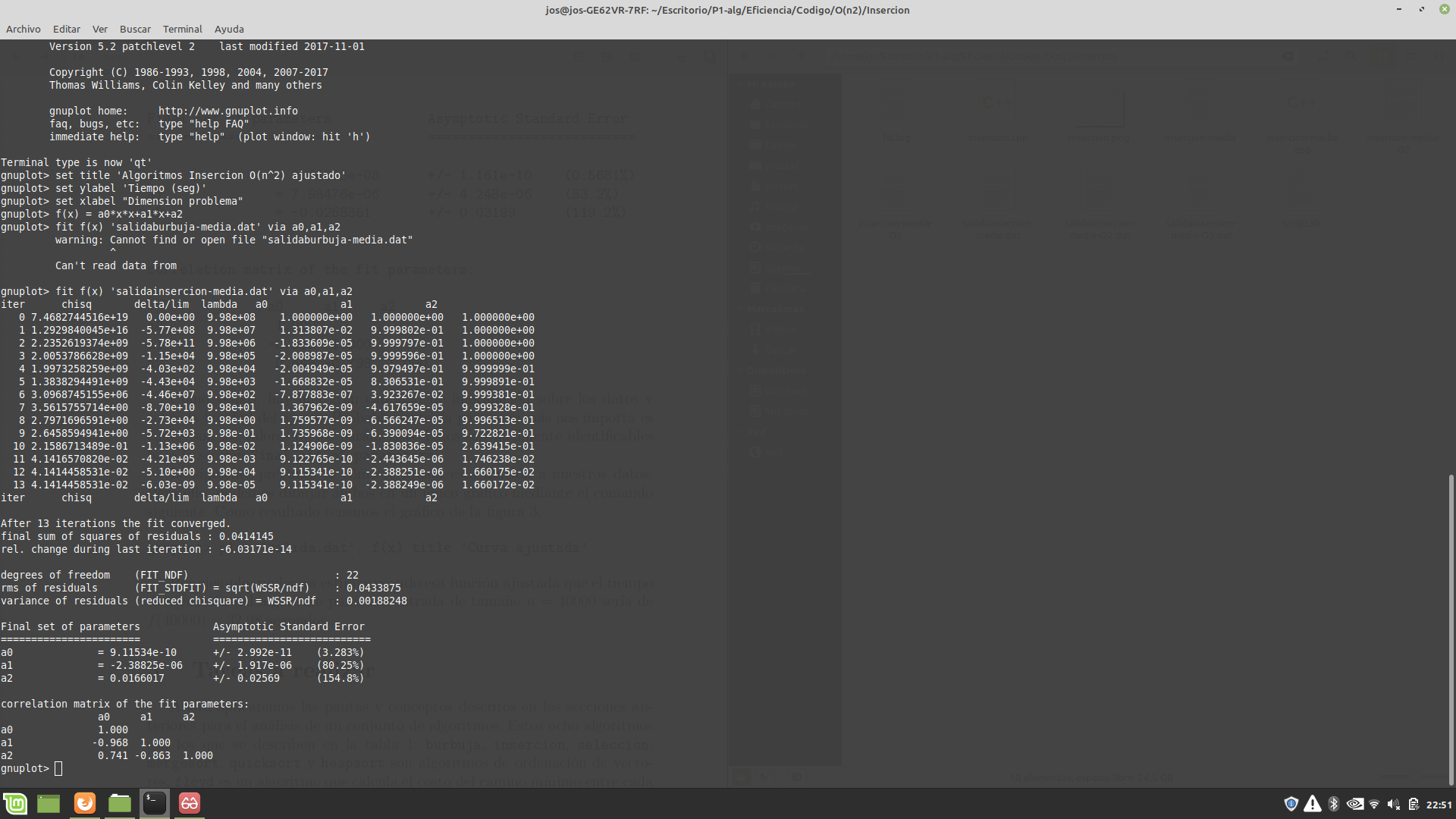
* + 1. ***Algoritmo de Inserción***



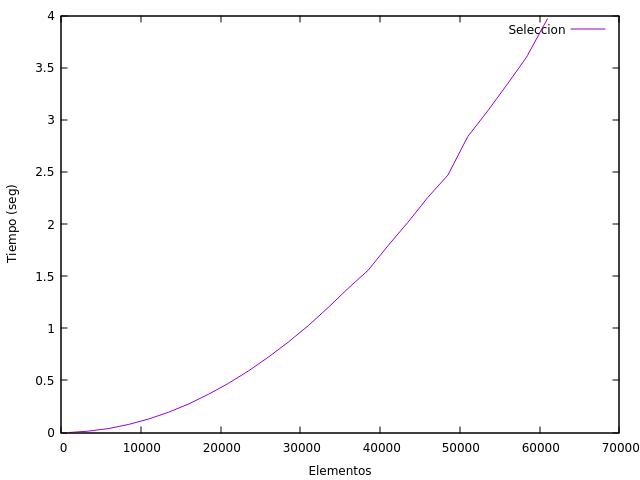
Enfoque híbrido: Inserción ajustado a T(n) = a0\*n2 + a1\*n + a2



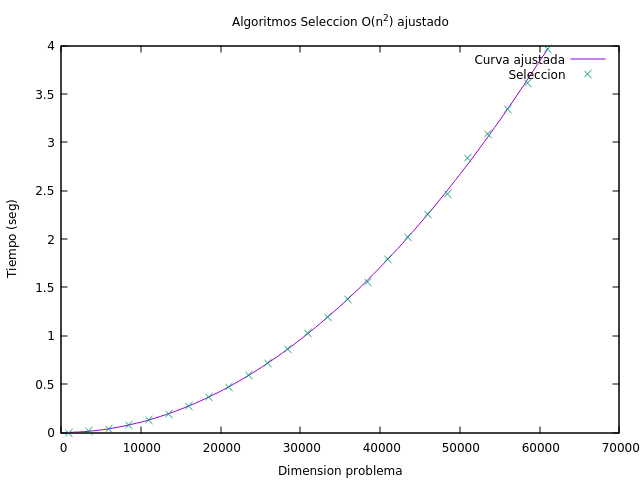
Valor de las constantes tras el ajuste:



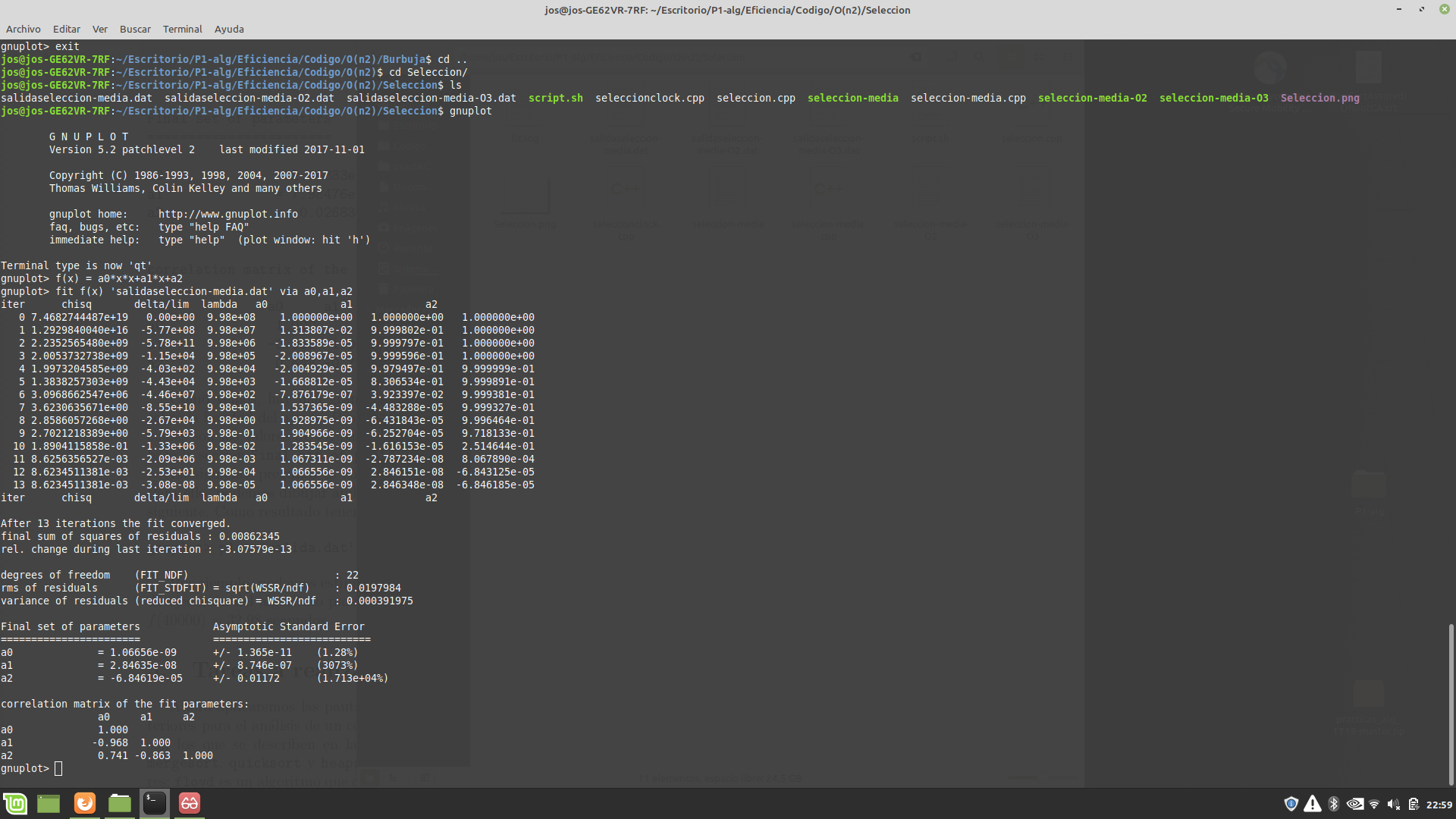
* + 1. ***Algoritmo de Selección***



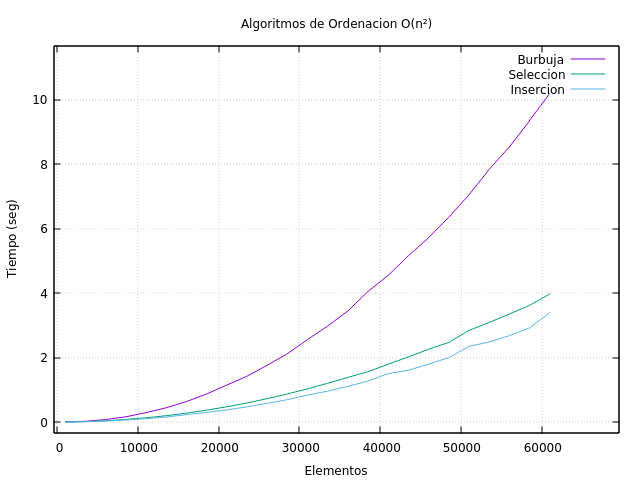
Enfoque híbrido: Selección ajustado a T(n) = a0\*n2 + a1\*n + a2



Valor de las constantes tras el ajuste:



Una vez hecho el análisis empírico e híbrido de cada algoritmo de ordenación de orden O(n2) por separado es el momento de compararlos en una misma gráfica:



Como podemos ver, cuanto mayor es la dimensión del problema (número de elementos del vector a ordenar), más se aprecia la diferencia de tiempo entre los algoritmos. El mejor tiempo se obtiene para el algoritmo de ordenación por Selección. Por el contrario, el algoritmo de ordenación Burbuja obtiene los peores tiempos, con gran diferencia respecto a Inserción y Selección, que para tamaños de problema grandes tienen una diferencia de menos de medio segundo aproximadamente.

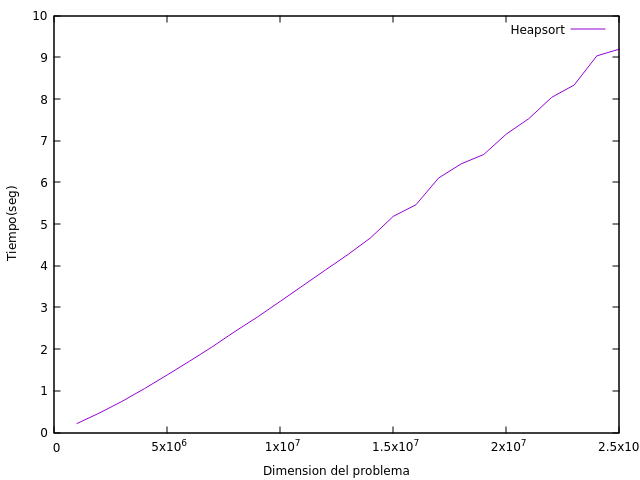
1. ***Algoritmos de Ordenación de Orden O(nlogn)***

Para este tipo de algoritmos con una eficiencia teórica de orden O(n2) se han obtenido los siguientes tiempos de ejecución en función del tamaño de la entrada del problema, en este caso, el número de componentes del vector a ordenar comenzando en 1000000, de 1000000 en 1000000, hasta 25000000.

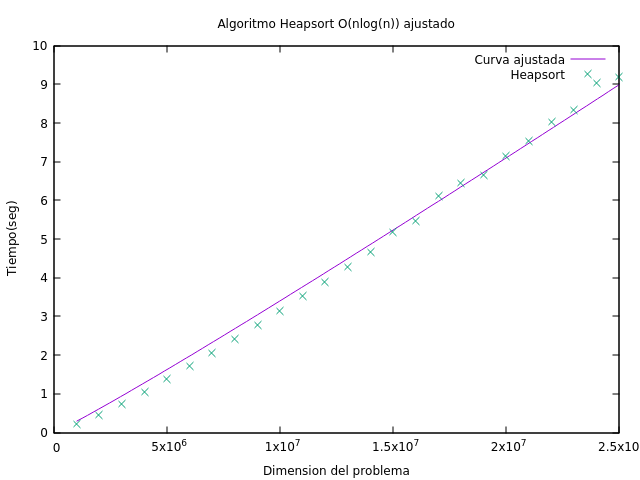
|  |  |  |  |
| --- | --- | --- | --- |
| Orden de eficiencia O(nlog(n)) | | | |
| Dimensión (elementos) | Heapsort | Mergesort | Quicksort |
| 1\*106 | 0.21187 | 0.208997 | 0.138882 |
| 2\*106 | 0.464664 | 0.43057 | 0.291777 |
| 3\*106 | 0.743614 | 0.743943 | 0.449823 |
| 4\*106 | 1.05295 | 0.897515 | 0.609289 |
| 5\*106 | 1.37959 | 1.19975 | 0.762986 |
| 6\*106 | 1.71161 | 1.5662 | 0.933892 |
| 7\*106 | 2.05634 | 1.57882 | 1.09681 |
| 8\*106 | 2.42214 | 1.8639 | 1.27618 |
| 9\*106 | 2.77046 | 2.24335 | 1.43591 |
| 10\*106 | 3.14536 | 2.49551 | 1.59502 |
| 11\*106 | 3.52204 | 2.83289 | 1.77258 |
| 12\*106 | 3.89851 | 3.18597 | 1.94187 |
| 13\*106 | 4.26904 | 3.49516 | 2.11638 |
| 14\*106 | 4.67064 | 3.28462 | 2.28479 |
| 15\*106 | 5.18346 | 3.58936 | 2.46421 |
| 16\*106 | 5.46264 | 3.88189 | 2.64738 |
| 17\*106 | 6.10263 | 4.22679 | 2.80091 |
| 18\*106 | 6.44357 | 4.52828 | 2.96646 |
| 19\*106 | 6.66715 | 4.86093 | 3.17993 |
| 20\*106 | 7.15683 | 5.20619 | 3.33331 |
| 21\*106 | 7.53007 | 5.56328 | 3.49112 |
| 22\*106 | 8.03744 | 5.95751 | 3.67838 |
| 23\*106 | 8.33518 | 6.3802 | 3.88384 |
| 24\*106 | 9.03367 | 6.65836 | 4.0132 |
| 25\*106 | 9.1974 | 7.12851 | 4.20191 |

A continuación, se muestran las gráficas obtenidas de los algoritmos de orden logarítmico con gnuplot y su ajuste.

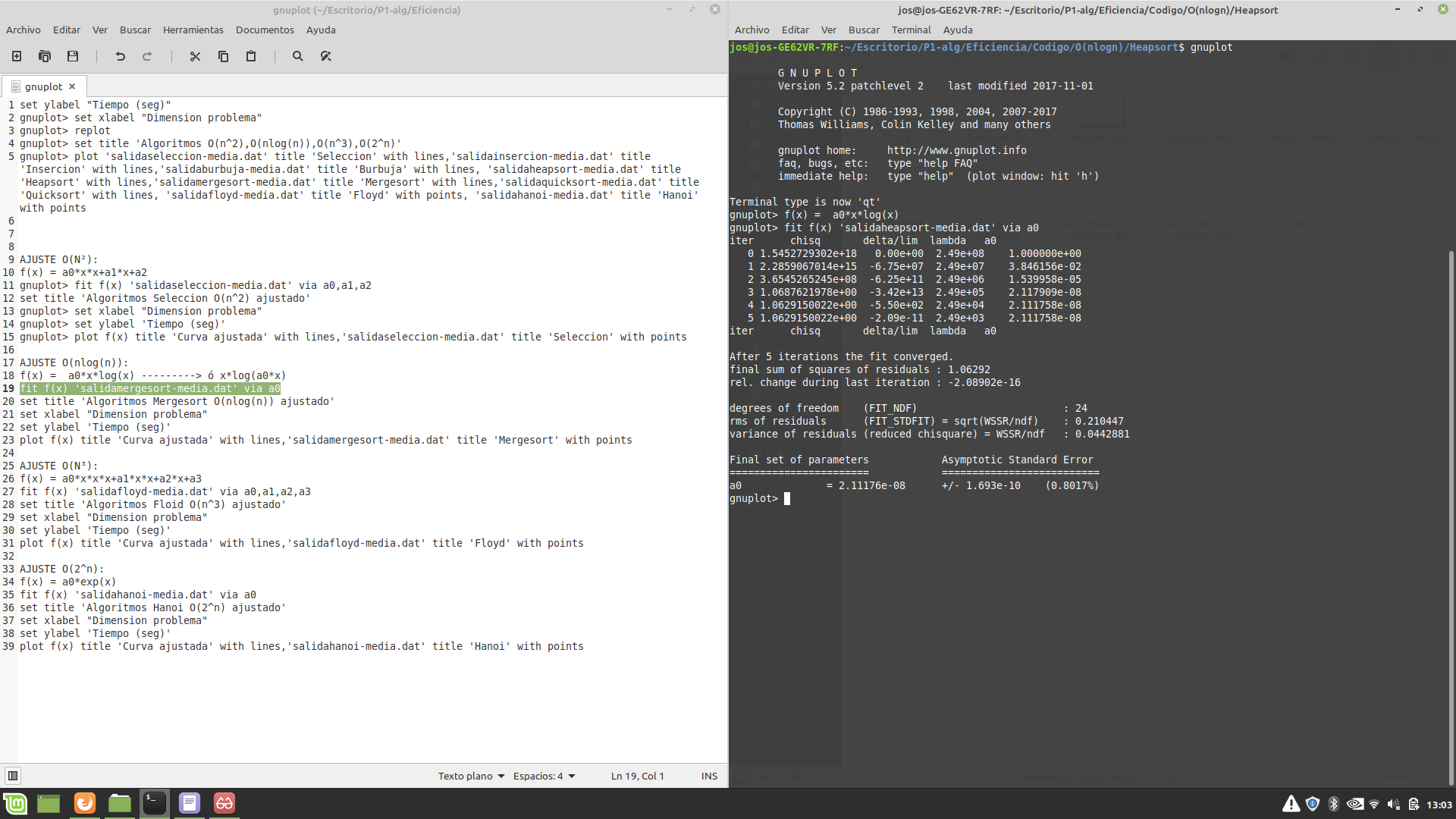
* + 1. ***Algoritmo Heapsort***



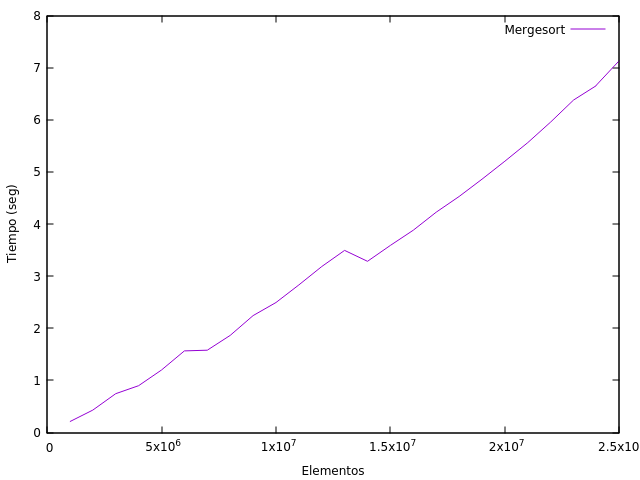
Enfoque híbrido: Heapsort ajustado a T(n) = a0\*n\*log(n)



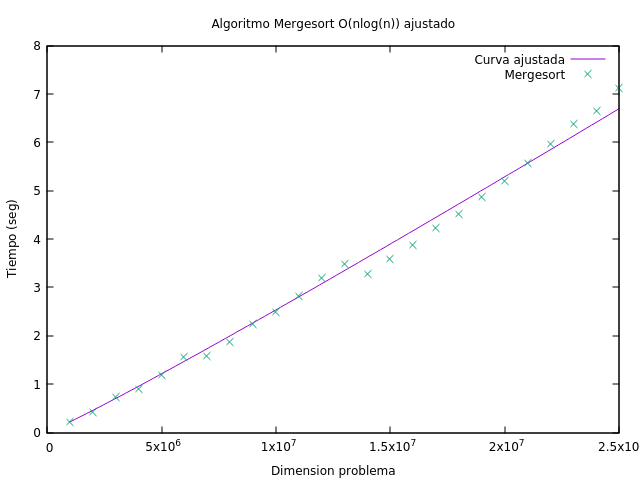
Valor de la constante tras el ajuste:



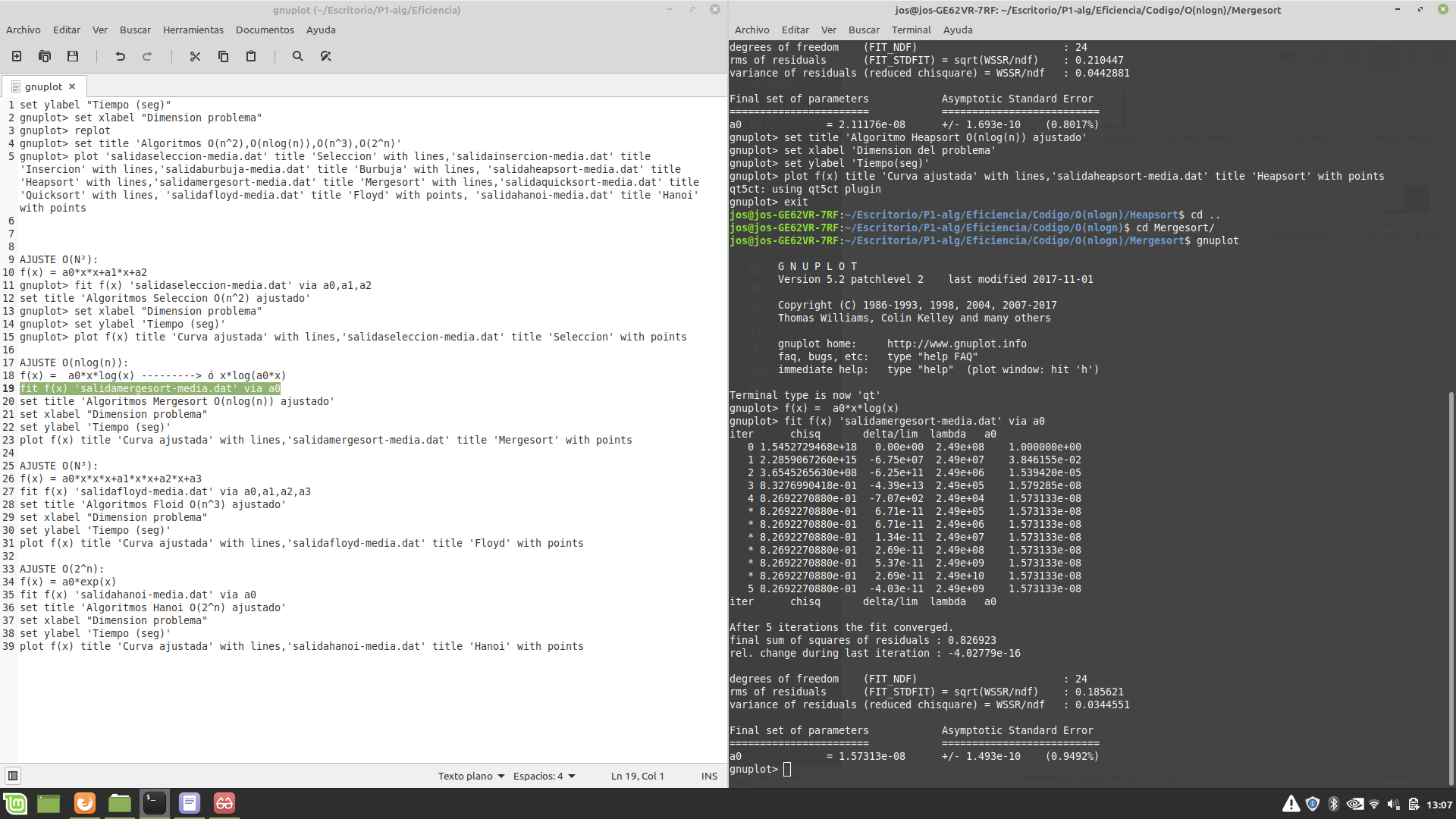
* + 1. ***Algoritmo Mergesort***



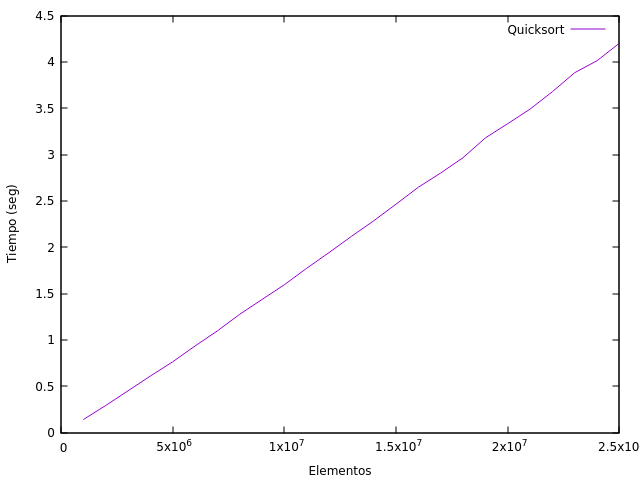
Enfoque híbrido: Mergesort ajustado a T(n) = a0\*n\*log(n)



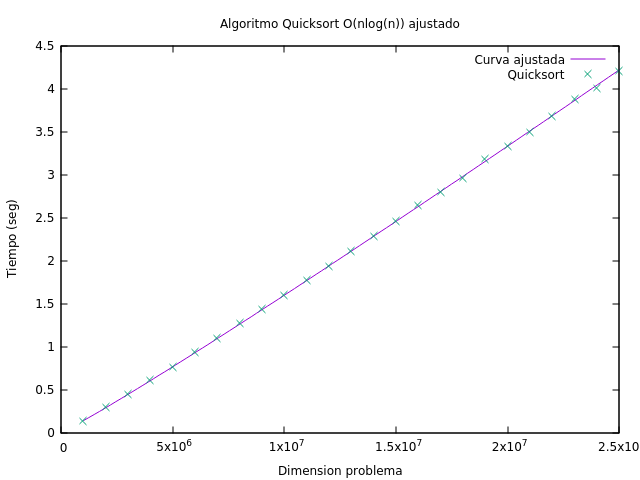
Valor de la constante tras el ajuste:



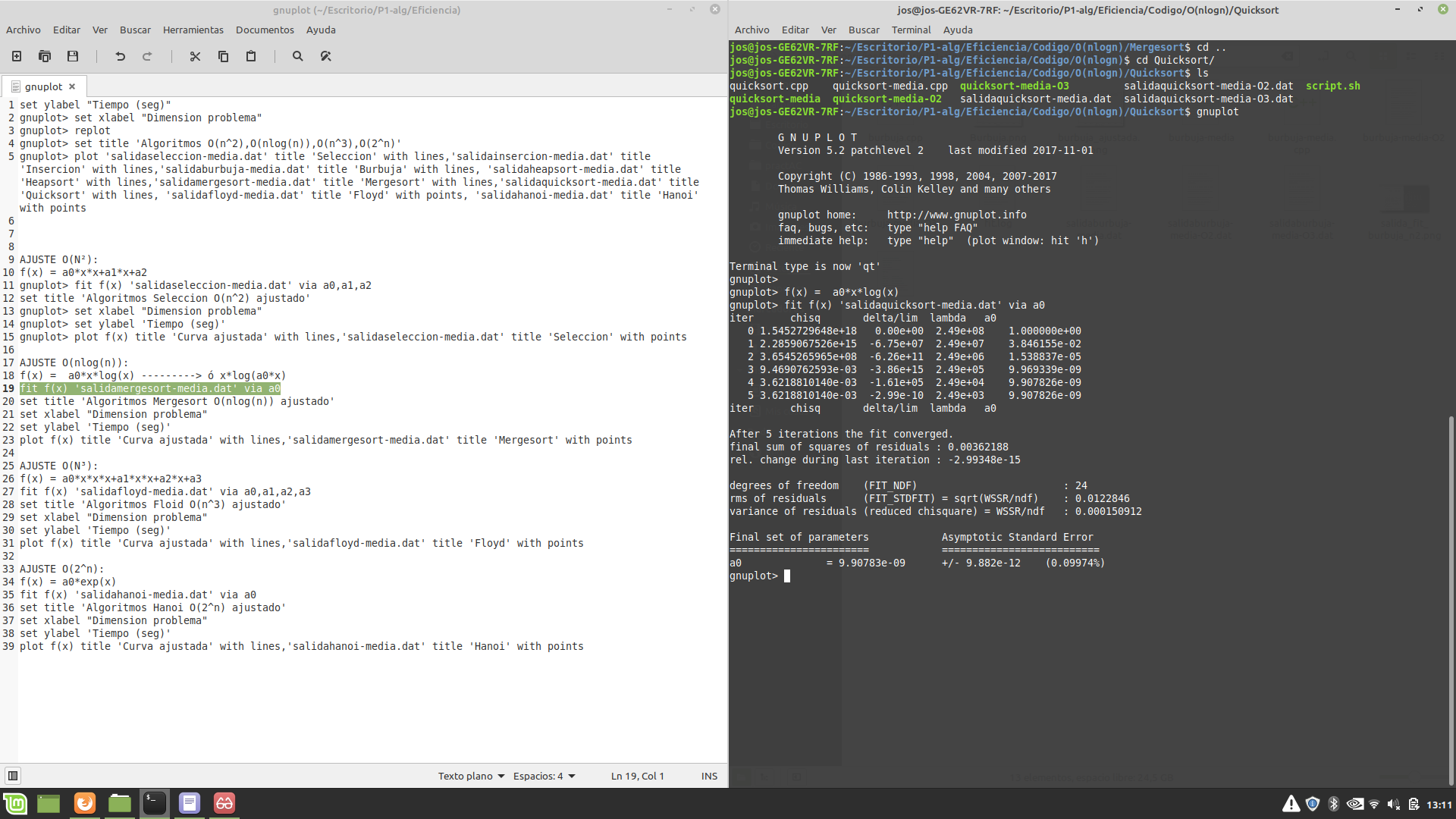
* + 1. ***Algoritmo Quicksort***



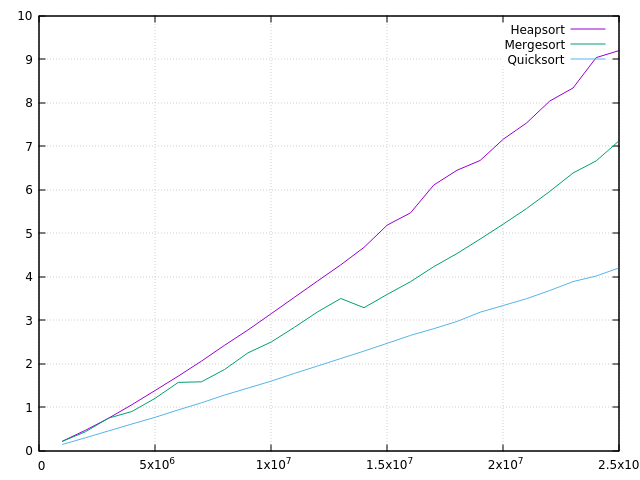
Enfoque híbrido: Mergesort ajustado a T(n) = a0\*n\*log(n)



Valor de la constante tras el ajuste:



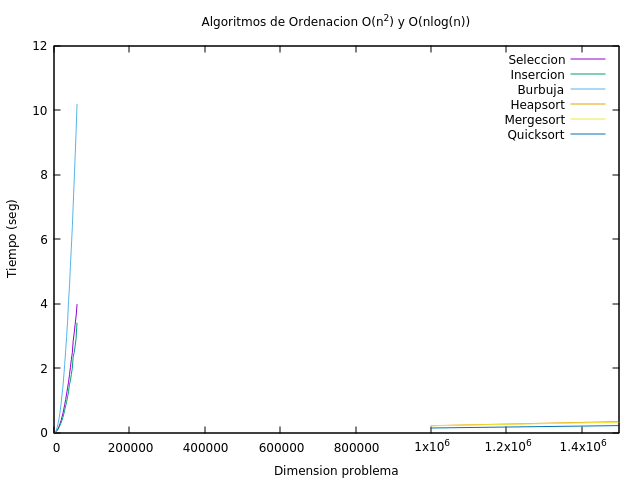
Una vez hecho el análisis empírico e híbrido de cada algoritmo de ordenación de orden O(nlog(n)) por separado es el momento de compararlos en una misma gráfica:



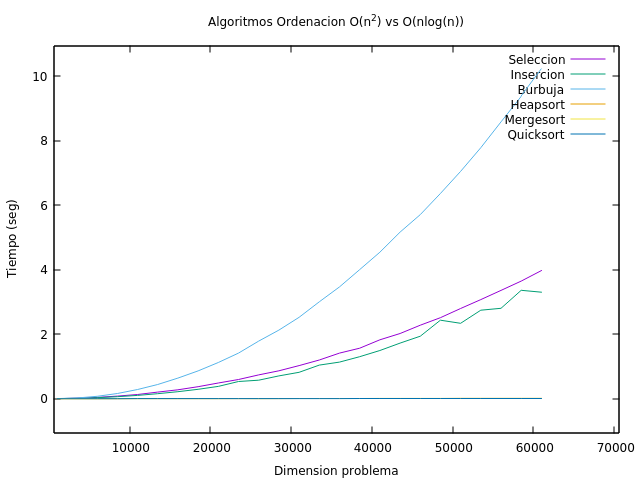
Como podemos ver, cuanto mayor es la dimensión del problema (número de elementos del vector a ordenar), más se aprecia la diferencia de tiempo entre los algoritmos. El mejor tiempo se obtiene para el algoritmo de ordenación por Quicksort. Por el contrario, el algoritmo de ordenación Heapsort obtiene los peores tiempos, con una diferencia de 2 segundos respecto a Mergesort y 5 segundos respecto al Quicksort, para 25000000 de elementos.

1. ***Comparativa Algoritmos de Ordenación de orden O(n2) y O(nlog(n))***

Para tamaños de problema grandes es claro que es necesario usar los algoritmos de ordenación del orden de O(nlog(n)) dado que los cuadráticos les sería prácticamente imposible alcanzar el objetivo de ordenar un vector de 1000000 de elementos.



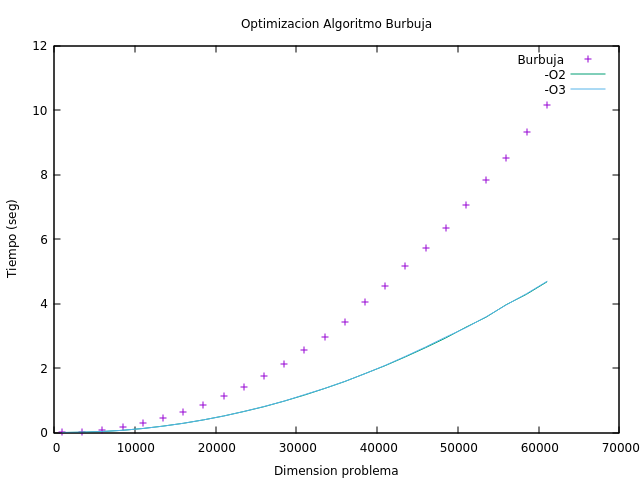
Para tamaños de problema pequeños los algoritmos de orden O(nlog(n)) son los más recomendables dado que tardan todos menos de medio segundo para el mayor tamaño probado para los algoritmos cuadráticos (60000 elementos). La diferencia en los algoritmos de orden logarítmico para tamaños pequeños es prácticamente despreciable aunque sigue siendo Quicksort el mejor y Heapsort el peor.



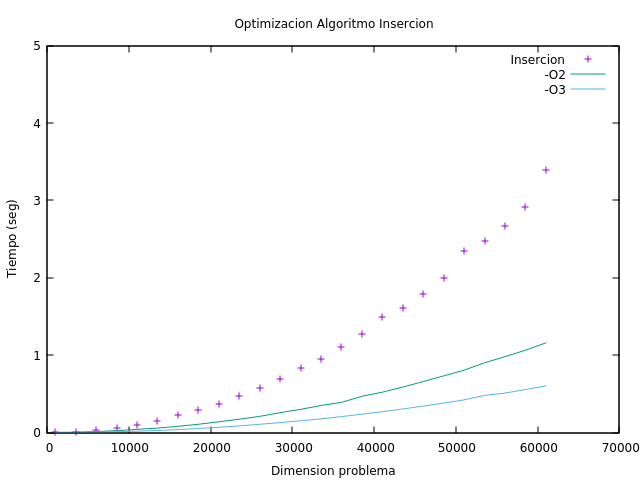
1. ***Comparativa Algoritmos de Ordenación con Optimización***

Se ha realizado el mismo análisis anterior para los algoritmos con diferentes opciones de optimización (-O2 y –O3) en la compilación del programa de prueba y se han comparado los resultados con y sin optimización. Estas opciones “simplifican” en número y forma a nivel de ensamblador el código generado en el compilado.

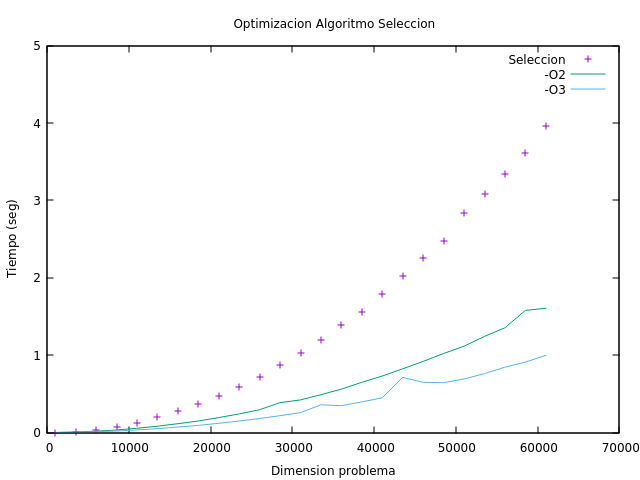
1. ***Algoritmos O(n2)***
   1. ***Burbuja:***



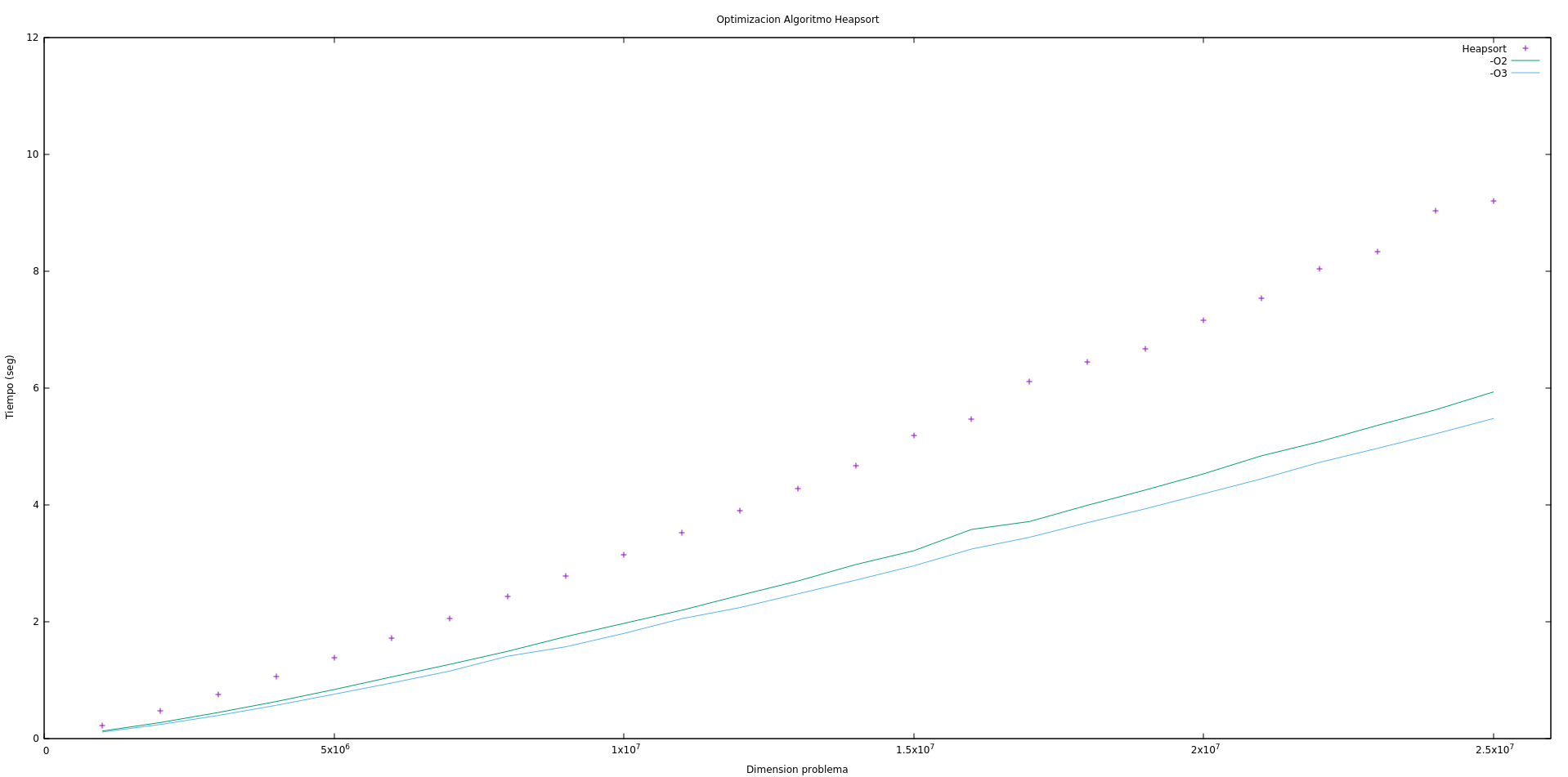
* 1. ***Inserción:***



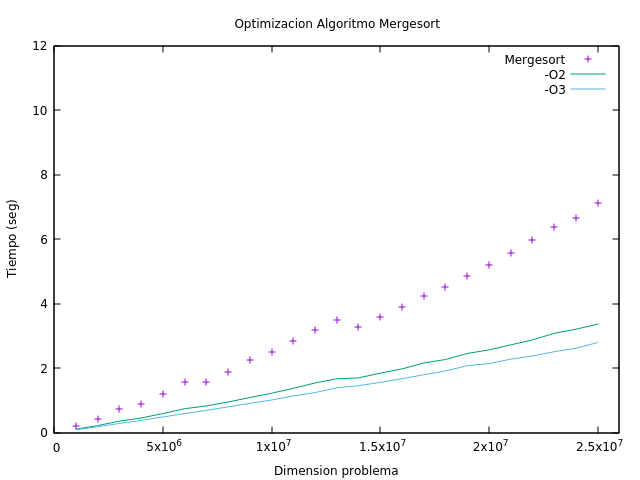
* 1. ***Selección:***



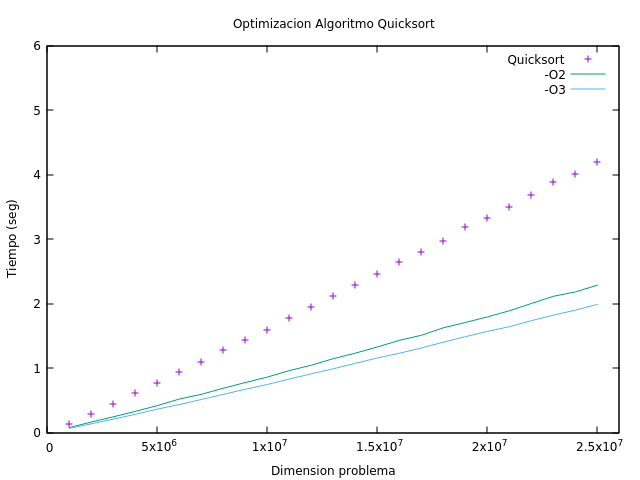
1. ***Algoritmos O(nlog(n))***
   1. ***Heapsort:***



* 1. ***Mergesort:***



* 1. ***Quicksort:***

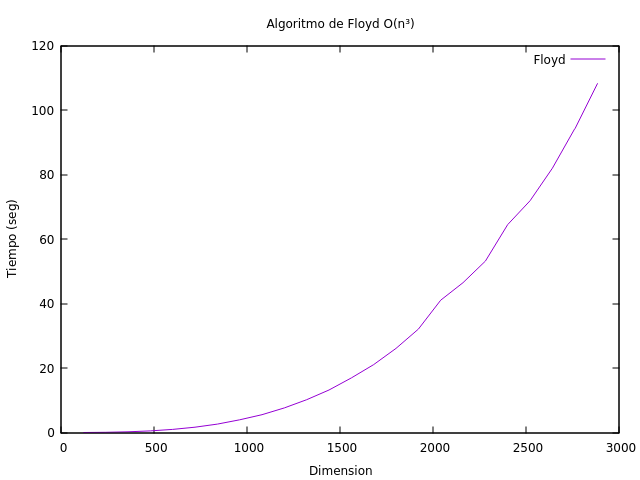


1. ***Algoritmo de Floyd O(n3)***

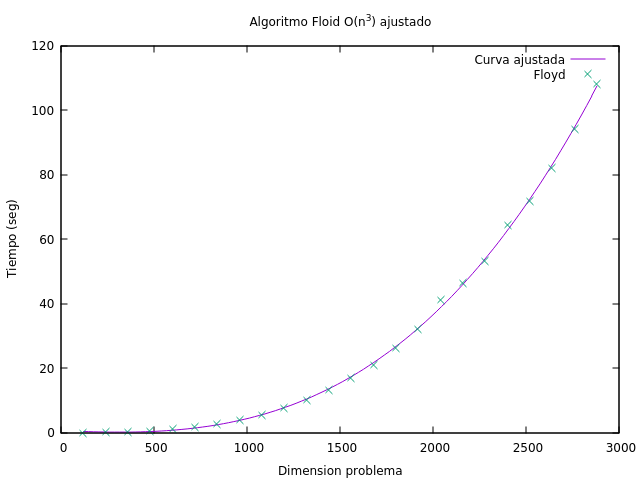
Para este tipo de algoritmos con una eficiencia teórica de orden O(n3) se han obtenido los siguientes tiempos de ejecución en función del tamaño de la entrada del problema, en este caso, la dimensión de la matriz que representa los pares de nodos de un grafo dirigido y del cual se quiere calcular el costo del camino mínimo entre cada par de nodos. comenzando en 120, de 120 en 120, hasta 2880.

|  |  |
| --- | --- |
| Orden de eficiencia O(n3) | |
| Dimensión | Foyd |
| 120 | 0.00878197 |
| 240 | 0.0625277 |
| 360 | 0.207162 |
| 480 | 0.504981 |
| 600 | 0.947648 |
| 720 | 1.64487 |
| 840 | 2.60446 |
| 960 | 3.9263 |
| 1080 | 5.5313 |
| 1200 | 7.62077 |
| 1320 | 10.1595 |
| 1440 | 13.1828 |
| 1560 | 16.9363 |
| 1680 | 21.0648 |
| 1800 | 26.0691 |
| 1920 | 32.0736 |
| 2040 | 41.019 |
| 2160 | 46.4523 |
| 2280 | 53.1511 |
| 2400 | 64.4914 |
| 2520 | 71.893 |
| 2640 | 81.9666 |
| 2760 | 94.2081 |
| 2880 | 108.139 |

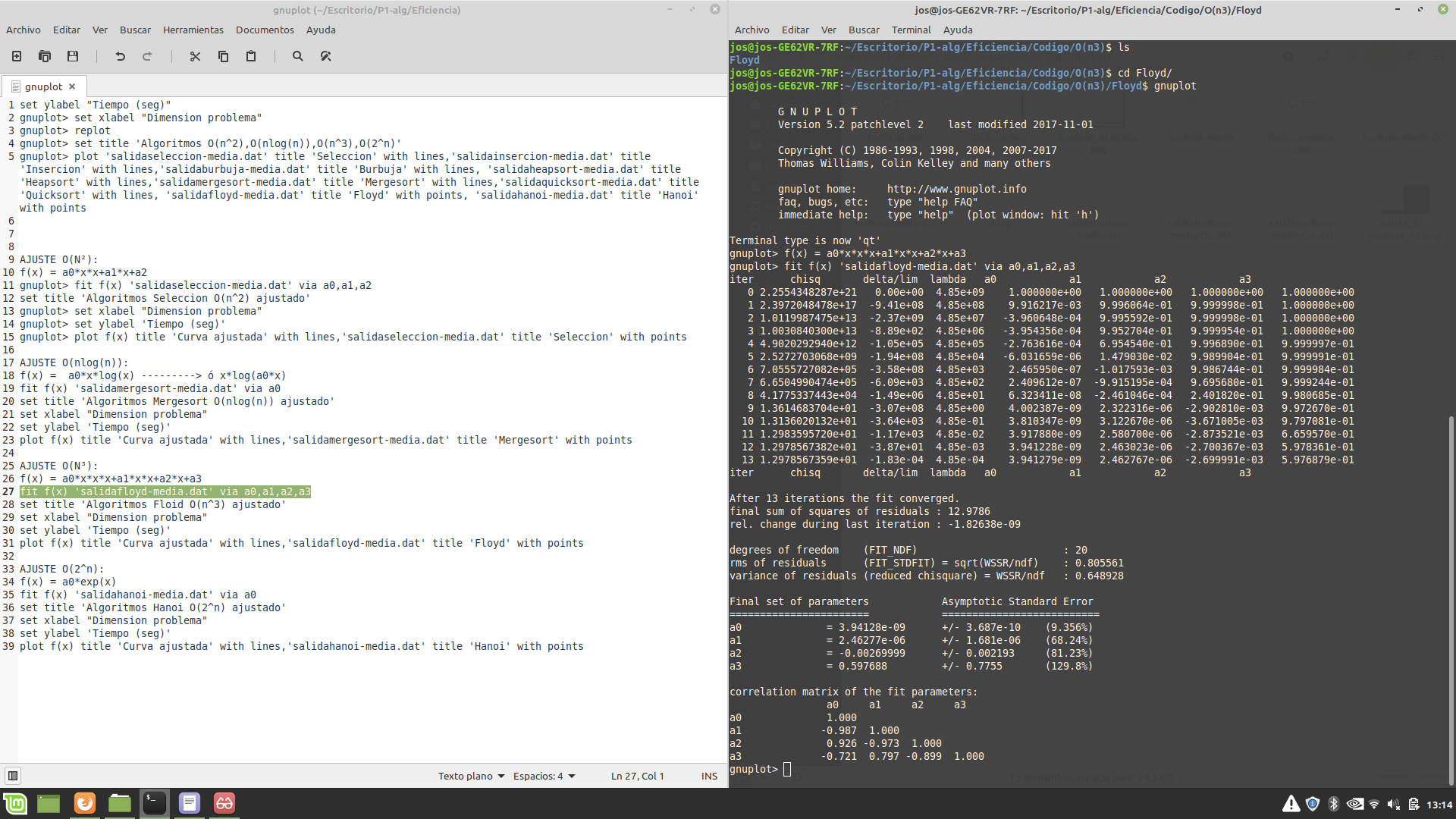
A continuación, se muestran las gráficas obtenidas con gnuplot y su ajuste, donde se puede apreciar claramente la función característica de los algoritmos cúbicos.



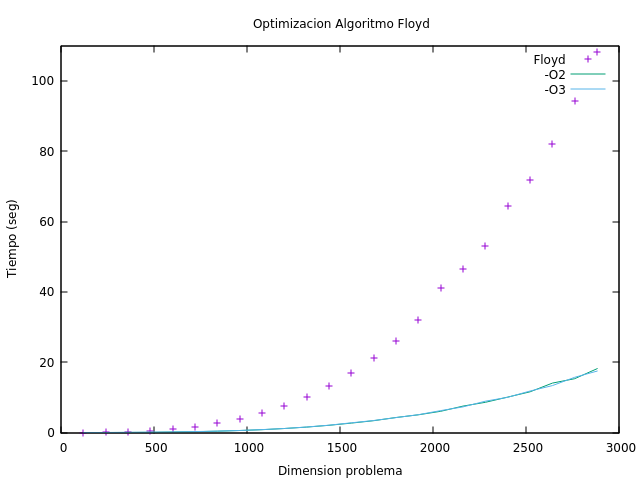
Enfoque híbrido: Floyd ajustado a T(n) = a0\*x\*x\*x+a1\*x\*x+a2\*x+a3



Valor de las constantes tras el ajuste:



1. ***Comparativa Algoritmo de Floyd con Optimización –O2 y –O3***



1. ***Algoritmo para Torres de Hanoi O(2n)***

Las Torres de Hanoi es un rompecabezas inventado en 1883 por el matemático francés Édouard Lucas.

Consiste en tres columnas y un número indeterminado de discos que definen la complejidad de la solución, en tanto su cantidad aumente.

Los discos tienen diferente tamaño y están ordenados de forma ascendente.

El objetivo del juego es traspasar todos los discos de la primera columna hacia la tercera manteniendo el orden ascendente (el disco más grande abajo), tal como estaban al inicio.

Las reglas del juego son:

* + - Solo un disco se puede mover a la vez.
    - Un disco de mayor tamaño no puede ponerse sobre uno más pequeño que él.
    - Solo se puede desplazar el disco en la parte superior de una columna.

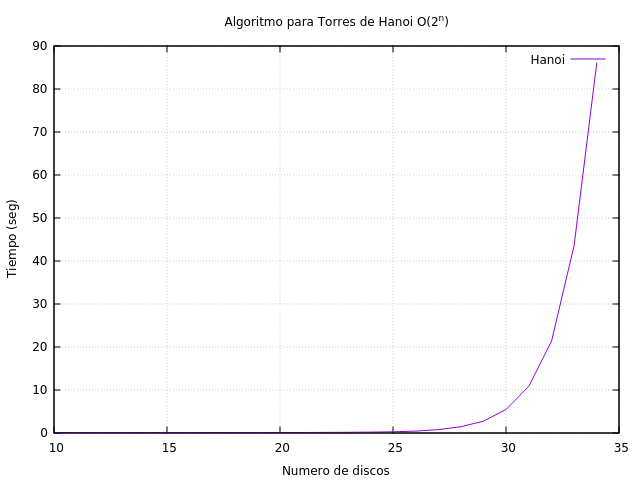
Para mover n discos son necesarios 2n-1 pasos.

En nuestro caso se tiene una implementación recursiva del algoritmo, dado que es una función que se llama a sí misma hasta que se cumple la condición, aunque también se podría resolver de manera iterativa.

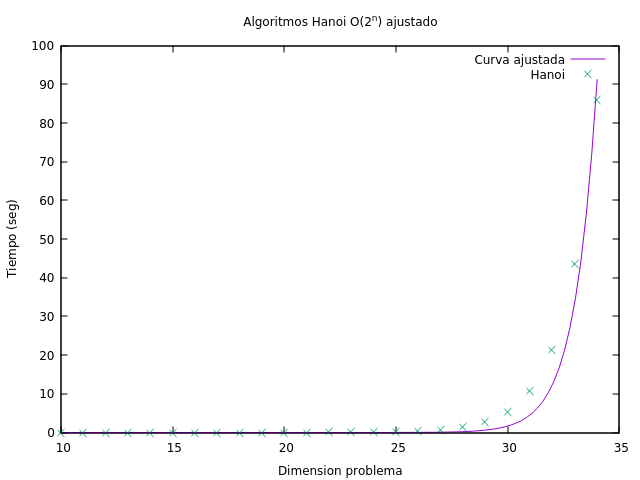
Para este tipo de algoritmos con una eficiencia teórica de orden O(2n) se han obtenido los siguientes tiempos de ejecución en función del tamaño de la entrada del problema, en este caso, el número de discos con los que se quiere resolver el rompecabezas. comenzando en 120, de 120 en 120, hasta 2880.

|  |  |
| --- | --- |
| Orden de eficiencia O(nn) | |
| Numero de Discos | Hanoi |
| 10 | 2.0342e-05 |
| 11 | 3.98473e-05 |
| 12 | 7.76167e-05 |
| 13 | 0.000182285 |
| 14 | 0.000306611 |
| 15 | 0.000207873 |
| 16 | 0.000456533 |
| 17 | 0.000779292 |
| 18 | 0.0014682 |
| 19 | 0.00288642 |
| 20 | 0.00569848 |
| 21 | 0.0106663 |
| 22 | 0.0211244 |
| 23 | 0.0431034 |
| 24 | 0.0852143 |
| 25 | 0.168702 |
| 26 | 0.333921 |
| 27 | 0.663048 |
| 28 | 1.33978 |
| 29 | 2.65751 |
| 30 | 5.40084 |
| 31 | 10.7781 |
| 32 | 21.2854 |
| 33 | 43.5722 |
| 34 | 85.9018 |

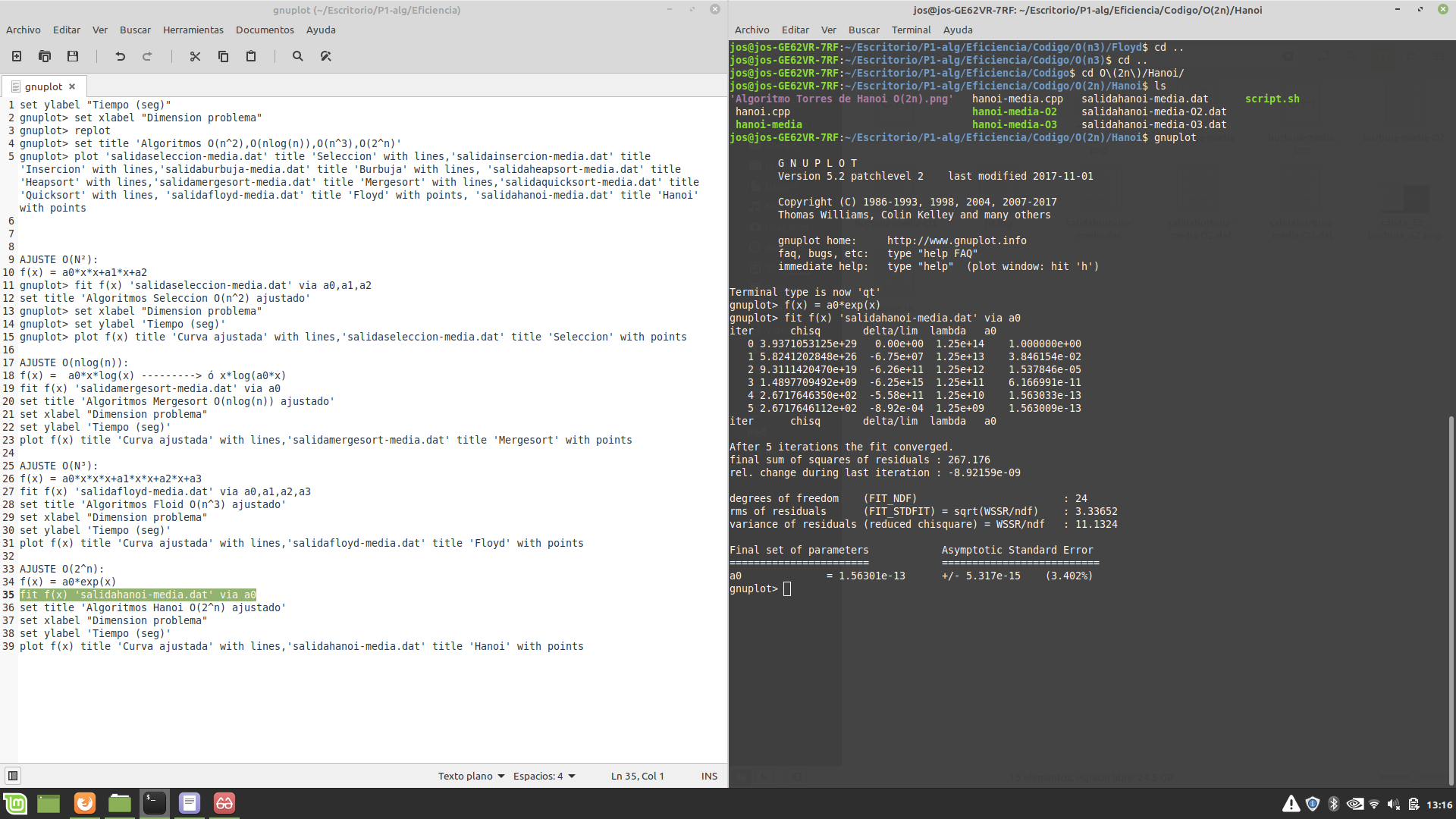
A continuación, se muestran las gráficas obtenidas con gnuplot y su ajuste, donde se puede apreciar claramente la función característica de los algoritmos exponenciales.



Enfoque híbrido: Hanoi ajustado a T(n) = a0\*exp(x)



Valor de la constante tras el ajuste:



1. ***Comparativa Algoritmo Hanoi con Optimización –O2 y –O3***

