## horizontal line



Práctica 1: Análisis de Eficiencia de Algoritmos

2019

Realizado por:

Miguel Ángel Campos Cubillas

Alejandro Pinel Martínez

Guillermo Palomino Sánchez

Pablo Lombardero Ros

Nikita Stetskiy

# ÍNDICE

[**ÍNDICE**](#_o8tj1ja02pe) **1**

[**RESÚMEN**](#_9dnkizyuc296) **2**

[**PASOS PREVIOS**](#_e4z0pqttc0xw) **2**

[**EFICIENCIA EMPÍRICA**](#_6rm8p4vdua9b) **3**

[**EFICIENCIA HÍBRIDA**](#_yc9pzfermkob) **13**

[**PRINCIPIO DE INVARIANZA**](#_a5jbh4ybm1ez) **29**

[**COMPARACIÓN ENTRE TODOS LOS ALGORITMOS DE ORDENACIÓN**](#_or1j48h5xmom) **41**

# 

# 

# 

# 

# 

# RESUMEN

Durante el desarrollo de esta práctica se ha realizado la comprensión y la realización de el análisis de eficiencia de una serie de algoritmos de varios órdenes de eficiencia de diversas formas. Principalmente hemos enfocado el estudio del análisis de estos algoritmos desde un enfoque empírico e híbrido. Sin embargo, hemos realizado algunos cambios en la optimización del código y hemos ejecutado cada algoritmo en varios dispositivos para estudiar las diferencias en cada casuística.

# PASOS PREVIOS

Para medir los tiempos de ejecución para cada ejecución del algoritmo con respecto a diversos tamaños del problema, realizaremos unos cambios sustanciales en el código del \*.cpp

1. Incluimos la biblioteca time.
2. Declaramos en el main dos variables de tipo clock\_t: tantes y tdespues. Estas variables nos permitirán controlar las cotas del intervalo de ejecución en cada iteración.
3. Capturamos el tantes y el tdespues con una simple asignación del valor que produce la función clock() antes de la ejecución del algoritmo (tantes) e inmediatamente después del término de su ejecución (tdespues).
4. Una vez tenemos esos dos valores ejecutamos una conversión a segundos de ese intervalo (double) (tdespues-tantes).
5. Compilamos el programa con la orden: g++ algoritmo.cpp -o algoritmo
6. Finalmente creamos un Script para ejecutar recurrentemente el algoritmo para un conjunto de tamaños determinados necesarios para posteriormente realizar los análisis empíricos e híbridos acerca de cada uno de los algoritmos. Esto nos permitirá automatizar la tarea del análisis previo.
7. Los datos de las ejecuciones se almacenan en un archivo con formato \*.dat para posteriormente trabajar con él en gnuplot y que este reconozca los mismos.

|  |
| --- |
| #!/bin/csh -vx  echo “” >> salida.dat  @ i = 50  While ( $i < 2000 )  ./algoritmo $i >> salida.dat  @ i += 50  end |

Ejemplo del Script utilizado durante el cálculo de tiempos T(n)

# EFICIENCIA EMPÍRICA

***Esquema General***

Para mostrar la eficiencia empírica de nuestro algoritmo volcaremos los datos de la ejecución mediante el script en un fichero de salida de tipo \*.dat

1. Ejecutamos Gnuplot y generamos los siguientes comandos

1.1) Para representar la gráfica

* Set xlabel “Tamaño”
* Set ylabel “Tiempo (seg)”
* gnuplot> plot ‘salida.dat’ title ‘Eficiencia ALGORITMO’ with points/lines

1.2) Exportamos el resultado que nos brinda la herramienta Gnuplot a un fichero \*.png y a \*.pdf para salvaguardar los datos.

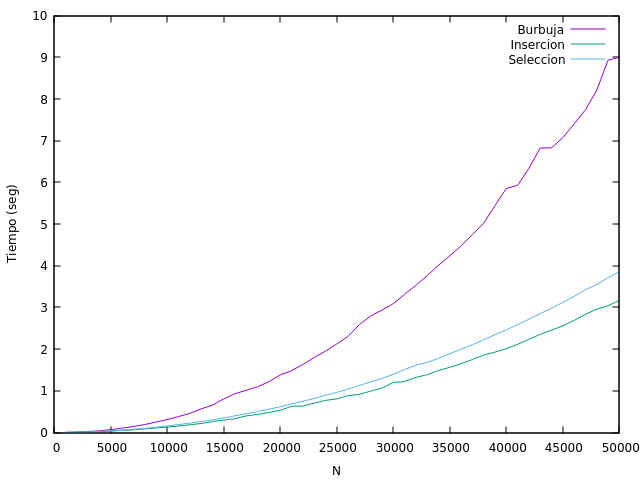
***Algoritmos de Ordenación Básicos***

Los algoritmos de ordenación Básicos que vamos a analizar son el algoritmo de la **burbuja**, la ordenación por **selección** y por **inserción**. Los tres algoritmos son de una eficiencia teórica de orden cuadrático O(n²) y su cuerpo principal es el siguiente:

|  |  |  |
| --- | --- | --- |
| **Burbuja** | **Inserción** | **Selección** |
| **int i, j;**  **int aux;**  **for (i = inicial; i < final - 1; i++)**  **for (j = final - 1; j > i; j--)**  **if (T[j] < T[j-1])**  **{**  **aux = T[j];**  **T[j] = T[j-1];**  **T[j-1] = aux;**  **}** | **int i, j;**  **int aux;**  **for (i = inicial + 1; i < final; i++) {**  **j = i;**  **while ((T[j] < T[j-1]) && (j > 0)) {**  **aux = T[j];**  **T[j] = T[j-1];**  **T[j-1] = aux;**  **j--;**  **}**  **}** | **int i, j, indice\_menor;**  **int menor, aux;**  **for (i = inicial; i < final - 1; i++) {**  **indice\_menor = i;**  **menor = T[i];**  **for (j = i; j < final; j++)**  **if (T[j] < menor) {**  **indice\_menor = j;**  **menor = T[j];**  **}**  **aux = T[i];**  **T[i] = T[indice\_menor];**  **T[indice\_menor] = aux;**  **}** |

En este caso para determinar la eficiencia empírica se medirán los tiempos de los tres algoritmos con tamaños crecientes desde un tamaño inicial de n=1000 hasta un tamaño final nfinal=50000 en intervalos de 1000. Los tiempos de ejecución son los que se muestran en la siguiente gráfica.

Con los datos anteriores, generamos la siguiente gráfica utilizando la herramienta gnuplot:



Los tres resultados dibujan gráficas parabólicas que se corresponden a sus órdenes O(n²), como era de esperar según la eficiencia teórica calculada. Sin embargo, las tres son muy dispares, debido a las constantes ocultas. A simple vista se puede observar que el algoritmo de la burbuja es mucho más ineficiente que selección e inserción. Para comprobar cuánto, procederemos al análisis híbrido más adelante.

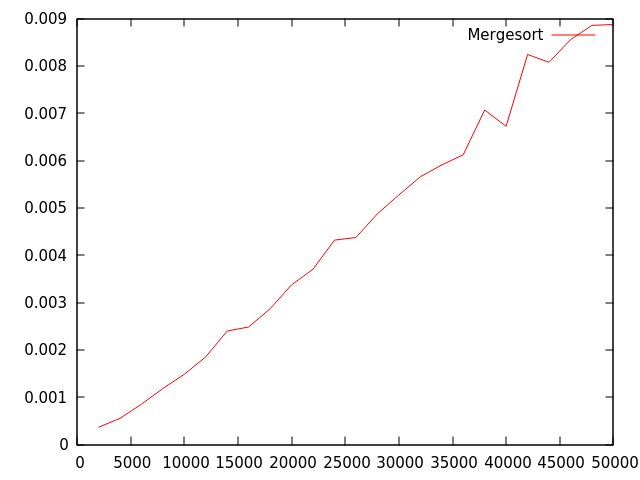
***Algortimos Mergesort, Quicksort, Heapsort***

En este caso, hemos tomado un tamaño inicial de 2000 y hemos ido aumentando el tamaño progresivamente en múltiplos del mismo llegando hasta 50000 para hacer un total de 25 mediciones del tiempo de ejecución del algortimo con respecto a 25 tamaños diferentes. Al mismo tiempo, los datos quedan registrados en un formato \*.dat compatible con gnuplot.

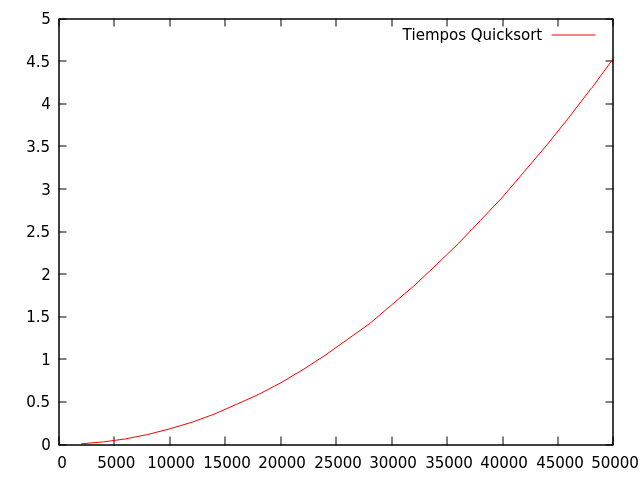
Los resultados empíricos obtenidos en relación al tamaño de la entrada son los siguientes:

|  |  |  |  |
| --- | --- | --- | --- |
| Tamaño | MERGESORT | QUICKSORT | HEAPSORT |
| 2000 | 0.0003636 | 0.0078994 | 0.0006616 |
| 4000 | 0.0005512 | 0.0303536 | 0.0006202 |
| 6000 | 0.000852 | 0.0652314 | 0.0016408 |
| 8000 | 0.001181 | 0.117550 | 0.0020936 |
| 10000 | 0.0014826 | 0.184120 | 0.0026746 |
| 12000 | 0.0018514 | 0.261592 | 0.0026896 |
| 14000 | 0.0023984 | 0.356520 | 0.0040670 |
| 16000 | 0.002482 | 0.470705 | 0.0048910 |
| 18000 | 0.0028684 | 0.588529 | 0.0049342 |
| 20000 | 0.0033752 | 0.724386 | 0.0056488 |
| 22000 | 0.0037046 | 0.878649 | 0.0065870 |
| 24000 | 0.0043186 | 1.046900 | 0.0080078 |
| 26000 | 0.0043752 | 1.232170 | 0.0083430 |
| 28000 | 0.0048756 | 1.418440 | 0.0089896 |
| 30000 | 0.005272 | 1.638450 | 0.0103810 |
| 32000 | 0.005656 | 1.867000 | 0.0117534 |
| 34000 | 0.0059076 | 2.106110 | 0.0121014 |
| 36000 | 0.0061222 | 2.357910 | 0.0133218 |
| 38000 | 0.0070676 | 2.628810 | 0.133218 |
| 40000 | 0.0067246 | 2.904000 | 0.0142410 |
| 42000 | 0.008242 | 3.209730 | 0.0145902 |
| 44000 | 0.0080756 | 3.517110 | 0.0151122 |
| 46000 | 0.0085538 | 3.841110 | 0.0163192 |
| 48000 | 0.008857 | 4.18084 | 0.0163192 |
| 50000 | 0.0088754 | 4.53860 | 0.0163192 |

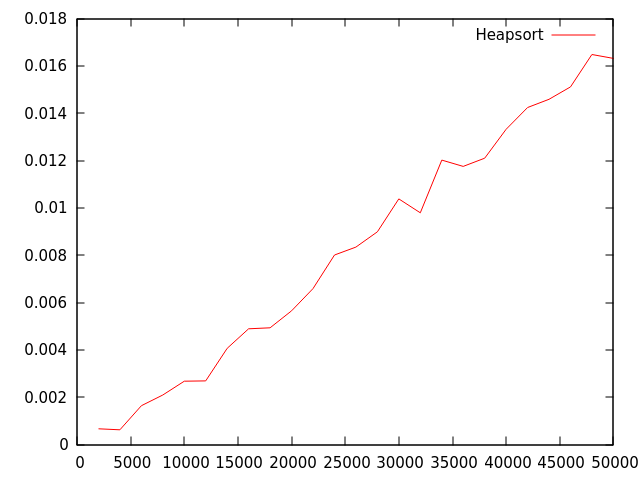
**Utilizando Gnuplot:**



Mergesort

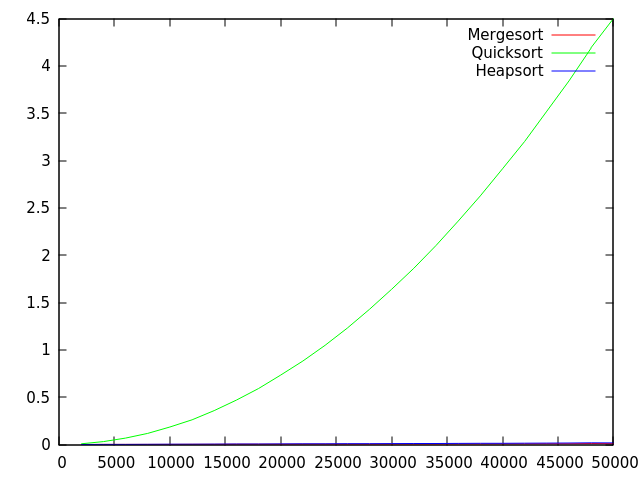


Quicksort



Heapsort

**Representamos Mergesort, Quicksort, Heapsort**

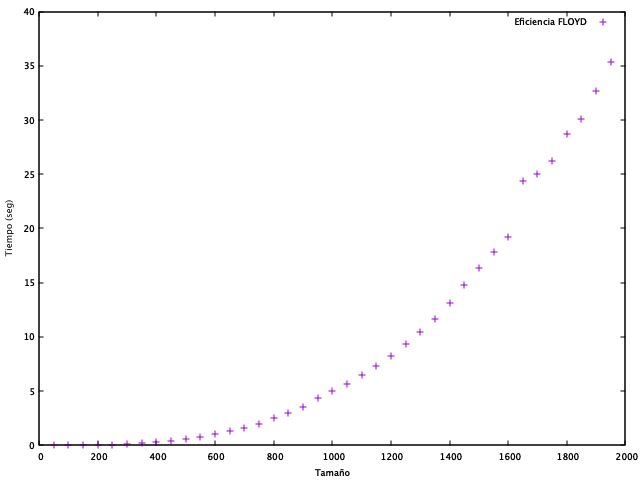


***Algoritmo de Floyd***

En este caso para determinar la eficiencia empírica se medirán los tiempos del algortimo de floyd con tamaños crecientes desde un tamaño inicial de n=50 hasta un tamaño final nfinal=1950 en intervalos de 50. Los tiempos de ejecución son los que se muestran en la siguiente tabla.

|  |  |
| --- | --- |
| ALGORITMO DE FLOYD | |
| 50 | 0.00074 |
| 100 | 0.006217 |
| 150 | 0.016754 |
| 200 | 0.038553 |
| 250 | 0.073673 |
| 300 | 0.127003 |
| 350 | 0.205754 |
| 400 | 0.303654 |
| 450 | 0.436177 |
| 500 | 0.589987 |
| 550 | 0.801326 |
| 600 | 1.01493 |
| 650 | 1.28623 |
| 700 | 1.63048 |
| 750 | 2.0098 |
| 800 | 2.52284 |
| 850 | 2.96109 |
| 900 | 3.55805 |
| 950 | 4.33398 |
| 1000 | 5.00068 |
| 1050 | 5.69223 |
| 1100 | 6.45785 |
| 1150 | 7.35483 |
| 1200 | 8.20163 |
| 1250 | 9.3145 |
| 1300 | 10.4202 |
| 1350 | 11.6516 |
| 1400 | 13.1334 |
| 1450 | 14.7526 |
| 1500 | 16.3643 |
| 1550 | 17.7944 |
| 1600 | 19.2537 |
| 1650 | 24.3572 |
| 1700 | 25.0607 |
| 1750 | 26.2477 |
| 1800 | 28.7303 |
| 1850 | 30.0915 |
| 1900 | 32.6493 |
| 1950 | 35.3584 |

Para graficar los datos obtenidos en el análisis empírico utilizamos gnuplot:

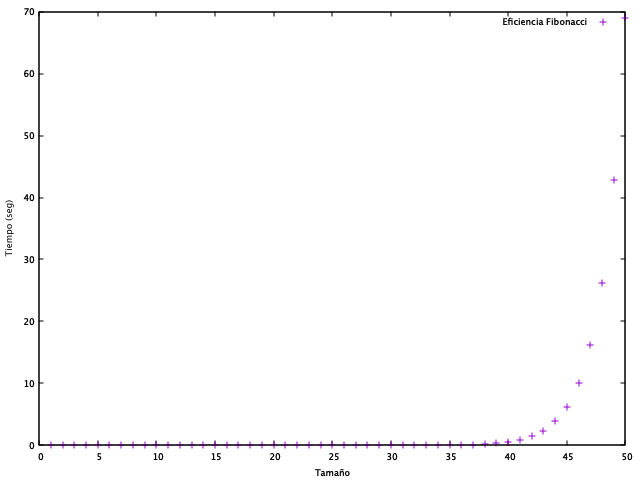


***Algoritmo de Fibonacci***

En este caso, hemos comenzado con un tamaño inicial de 1 y hemos ido incrementando en una unidad el tamaño hasta un tamaño final de 50. Obteniendo de esta manera 50 puntos para volcarlos en una gráfica. Los datos obtenidos se muestran en la siguiente gráfica.

|  |  |
| --- | --- |
| ALGORITMO DE FIBONACCI | |
| 1 | 1.66E-07 |
| 2 | 1.98E-07 |
| 3 | 2.72E-07 |
| 4 | 2.86E-07 |
| 5 | 3.67E-07 |
| 6 | 3.89E-07 |
| 7 | 5.46E-07 |
| 8 | 5.67E-07 |
| 9 | 8.20E-07 |
| 10 | 1.01E-06 |
| 11 | 1.61E-06 |
| 12 | 1.85E-06 |
| 13 | 2.52E-06 |
| 14 | 3.46E-06 |
| 15 | 5.58E-06 |
| 16 | 8.07E-06 |
| 17 | 1.33E-05 |
| 18 | 2.04E-05 |
| 19 | 3.30E-05 |
| 20 | 9.31E-05 |
| 21 | 6.40E-05 |
| 22 | 0.000139155 |
| 23 | 0.000196859 |
| 24 | 0.00033394 |
| 25 | 0.000638919 |
| 26 | 0.000995018 |
| 27 | 0.00151573 |
| 28 | 0.00247577 |
| 29 | 0.00381071 |
| 30 | 0.00662628 |
| 31 | 0.0100557 |
| 32 | 0.0132889 |
| 33 | 0.0219859 |
| 34 | 0.0361832 |
| 35 | 0.0565988 |
| 36 | 0.0839683 |
| 37 | 0.135266 |
| 38 | 0.213905 |
| 39 | 0.346029 |
| 40 | 0.568877 |
| 41 | 0.915853 |
| 42 | 1.50638 |
| 43 | 2.39434 |
| 44 | 3.86634 |
| 45 | 6.22593 |
| 46 | 10.1041 |
| 47 | 16.264 |
| 48 | 26.2641 |
| 49 | 42.8112 |
| 50 | 68.9658 |

Tras obtener estos datos pasamos a volcar los datos obtenidos en una gráfica para determinar el crecimiento asintótico de la eficiencia del algoritmo.



# EFICIENCIA HÍBRIDA

***Esquema General***

Para completar el estudio sobre la eficiencia de los algoritmos, vamos a recurrir a la eficiencia híbrida, que permite hallar la función de eficiencia precisa y concreta para condiciones de implementación en las que se van a ejecutar estos algoritmos: características del ordenador, lenguaje de programación elegido, compilador, etc.

Para calcular la eficiencia híbrida usaremos de nuevo la herramienta Gnuplot. Para ello definiremos una función para ajustarla con respecto a nuestros datos obtenidos en los pasos previos. Usaremos una función que se ajuste al orden de eficiencia equivalente al obtenido con respecto a la eficiencia teórica de cada algoritmo. Tomamos de ejemplo una función de orden O(n^3).

Hay un paso general para todos los algoritmos previo al resto, definir la función teórica. Lo que hemos hecho ha sido añadir una constante multiplicativa (también llamada constante oculta), c en nuestro caso, que es la que va a permitir el ajuste y de la que depende el rendimiento real del algoritmo.

1. Comandos en Gnuplot:

1.1) Definimos la función:

- gnuplot> f(x) = a0\*x\*x\*x+a1\*x\*x+a2\*x+a3

1.2) Indicamos la regresión a gnuplot:

- gnuplot> fit f(x) ’salida.dat’ via a0,a1,a2,a3

1.3) Ajustamos esa función a nuestros datos en un gráfico:

- gnuplot> plot ’salida.dat’, f(x) title ’Curva ajustada’

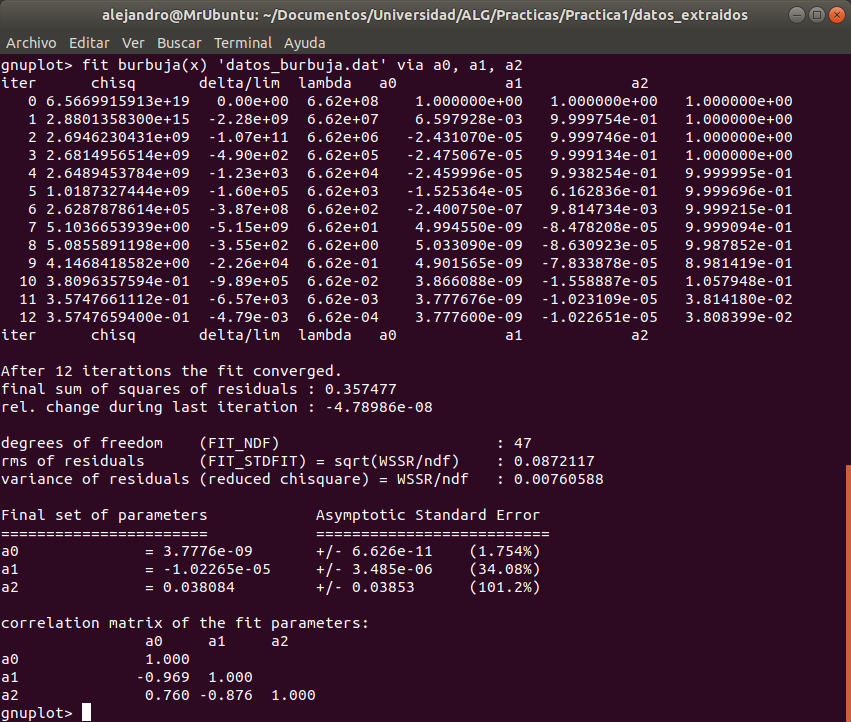
***Algoritmos de Ordenación Básicos***

Para calcular las constantes ocultas seguiremos utilizando gnuplot. Calcularemos las constantes ocultas teniendo en cuenta que la eficiencia teórica es de O(n²), luego la función asociada es *f(n) = a\*n² + b\*n +c.*

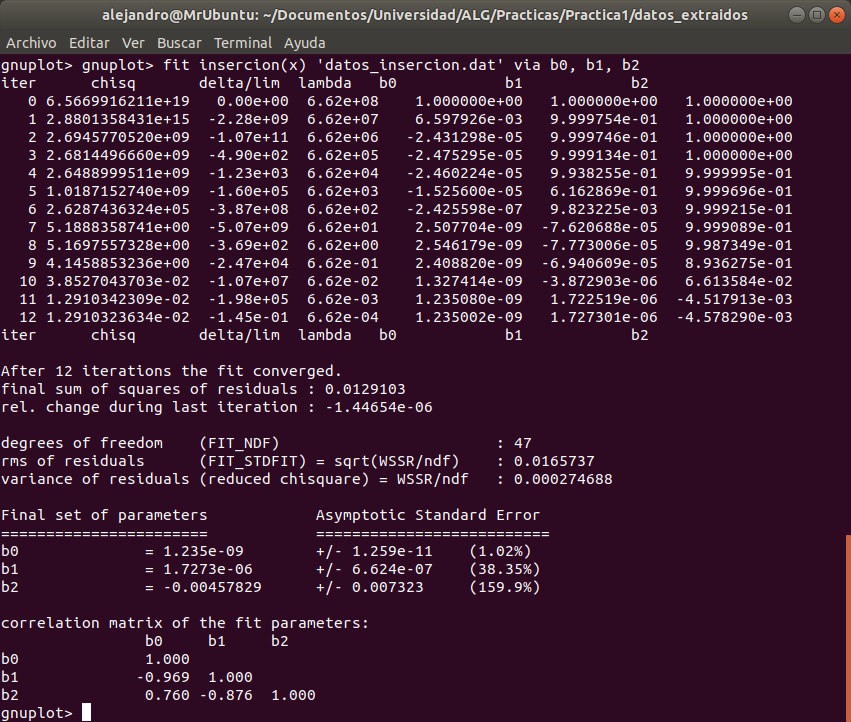
Sin embargo, el término dominante es *a*, luego nos centraremos únicamente es él para comparar la eficiencia de los distintos algoritmos.

A continuación, mostraremos los resultados del ajuste de esta función con los datos anteriormente calculados.

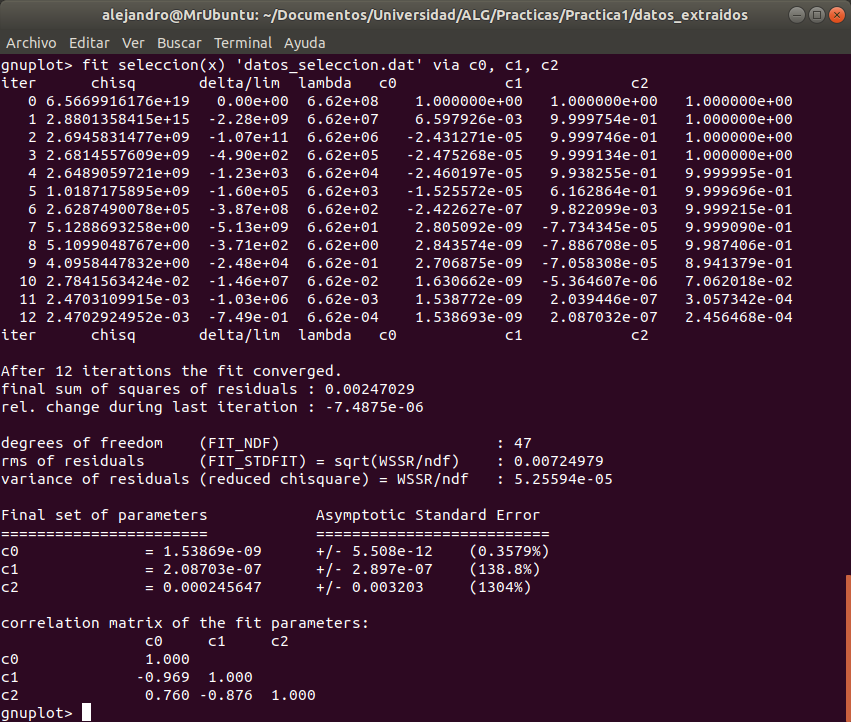
**Burbuja**

****

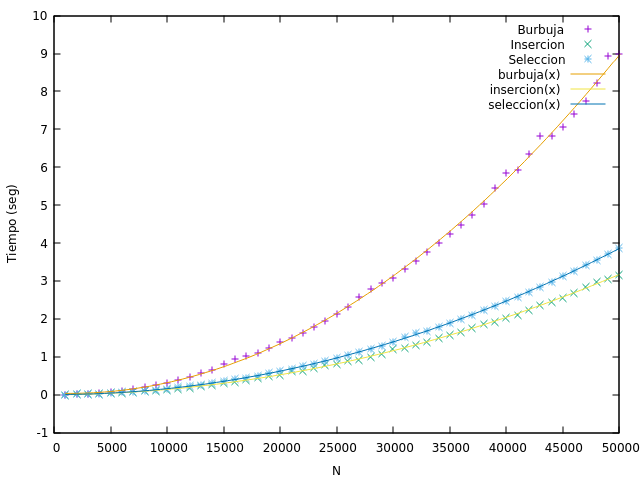
**Inserción**

****

**Selección**

****

**Ajuste**

****

**Comparación**

**Comparación**

Como podemos observar los términos dominantes de las funciones son respectivamente:

- Burbuja: 3.7776e-9

- Inserción: 1.235e-9

- Selección: 1.53869e-9

El resto de constantes menores podemos despreciarlos, ya que no influyen en el comportamiento asintótico, además de que no están calculadas de manera exacta.

Con esto podemos ver que el mejor algoritmo es el de inserción, que es ligeramente más rápido que selección y ambos superan ampliamente a la burbuja.

Concretamente, si calculamos:

3,7776÷1,235 = 3,058

1,53869÷1,235 = 1,245902834

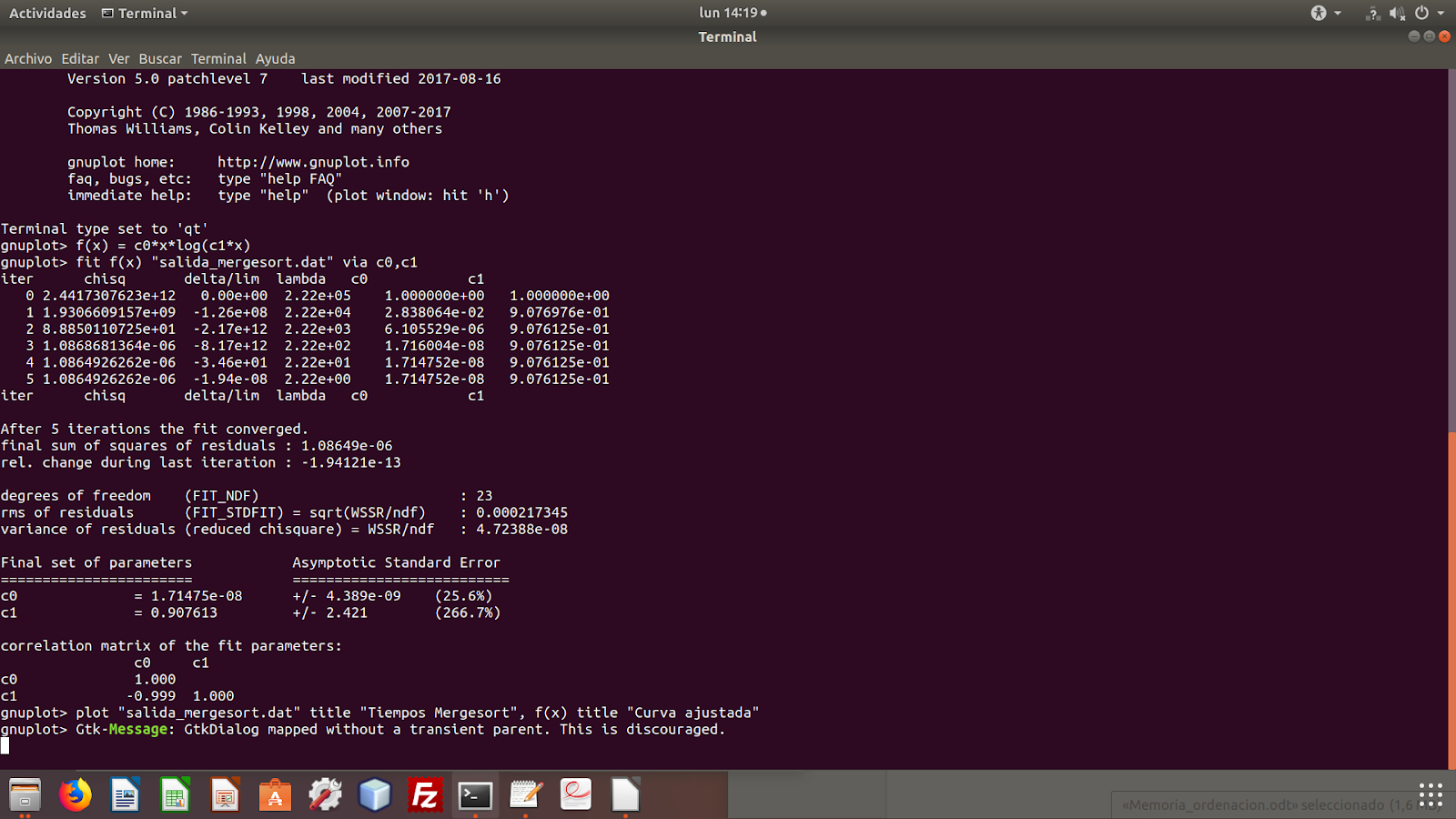
Por tanto, inserción es el triple de eficiente que burbuja y un 24% más eficiente que selección.

***Algoritmos Mergesort, Quicksort, Heapsort***

**Mergesort**

Procedemos a ajustar la función teórica de este algoritmo, que en este caso será

f(x)=c0\*x\*log(c1\*x) al ser O(nlog(n)), a la empírica por medio de gnuplot y una vez definida dicha función hacemos lo siguiente:



Se ajusta la función con el siguiente comando: fit f(x) "salida\_mergesort.dat" via c, siendo f(x) la función teórica, “salida\_mergesort.dat” el archivo con los resultados de las mediciones (de los que se extrae la función empírica) y c0 y c1 las constantes oculta cuyo valor queremos encontrar. Hay que fijarse en el par de tablas final que se obtiene como resultado:

Final set of parameters Asymptotic Standard Error

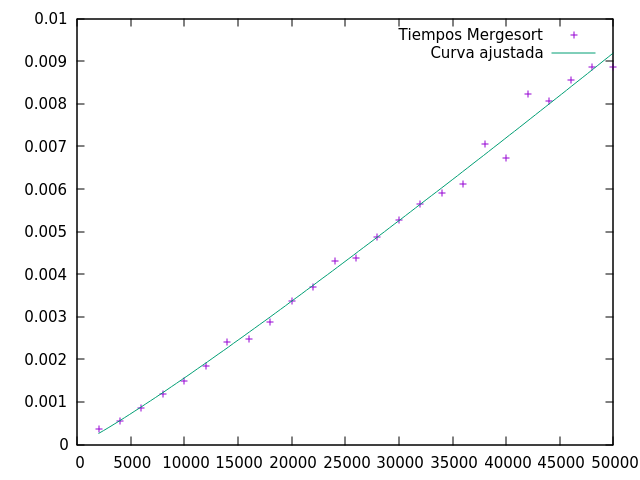
======================= ==========================

c0 = 1.71475e-08 +/- 4.389e-09 (25.6%)

c1 = 0.907613 +/- 2.421 (266.7%)

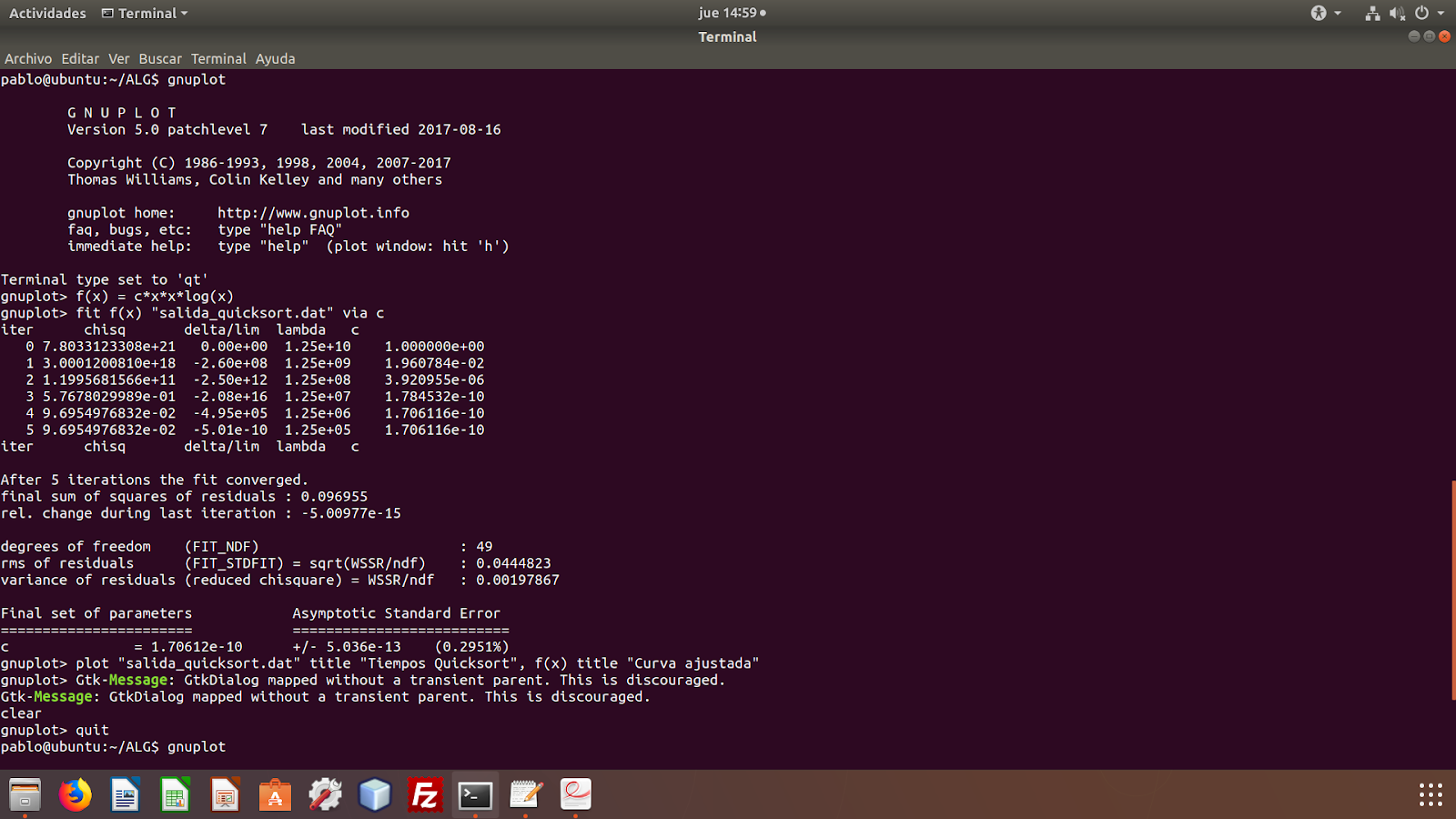
Nos centramos en el valor de c0 en la tabla "final set of parameters", que es igual a 1.71475e-08, obviamente el menor por ahora.

Finalmente, representamos la función teórica ajustada y la empírica en una misma gráfica, con el fin de ver hasta qué punto coinciden:



**Quicksort**

Procedemos a ajustar la función teórica de este algoritmo. A diferencia de lo que ocurre con mergesort y con heapsort, quicksort es de orden cuadrático, lo que significa que en este caso ajustaremos tres constantes multiplicativas en lugar de una, por medio de la función teórica f(x)=a0\*x\*x+a1\*x+a2, a la empírica por medio de gnuplot y una vez definida dicha función:



Se ajusta la función con el siguiente comando: fit f(x) "salida\_quicksort.dat" via a0, a1, a2, siendo f(x) la función teórica, “salida\_quicksort.dat” el archivo con los resultados de las mediciones (de los que se extrae la función empírica) y a0, a1 y a2 la constantes ocultas cuyo valor queremos encontrar. Hay que fijarse en el par de tablas final que se obtiene como resultado:

Final set of parameters Asymptotic Standard Error

======================= ==========================

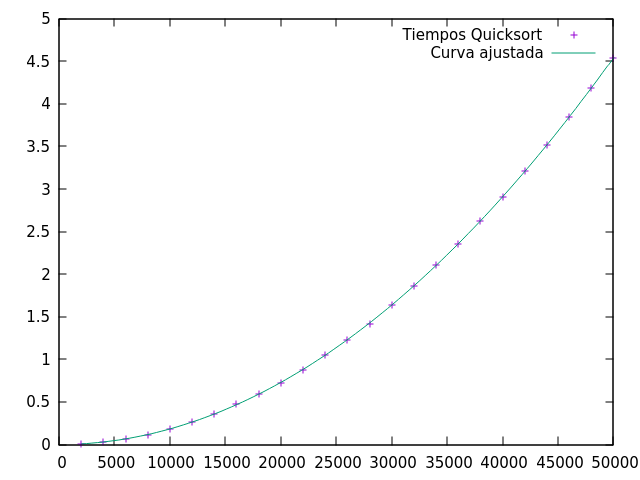
a0 = 1.81187e-09 +/- 3.444e-12 (0.1901%)

a1 = 2.13631e-07 +/- 1.845e-07 (86.37%)

a2 = -0.000592363 +/- 0.002082 (351.5%)

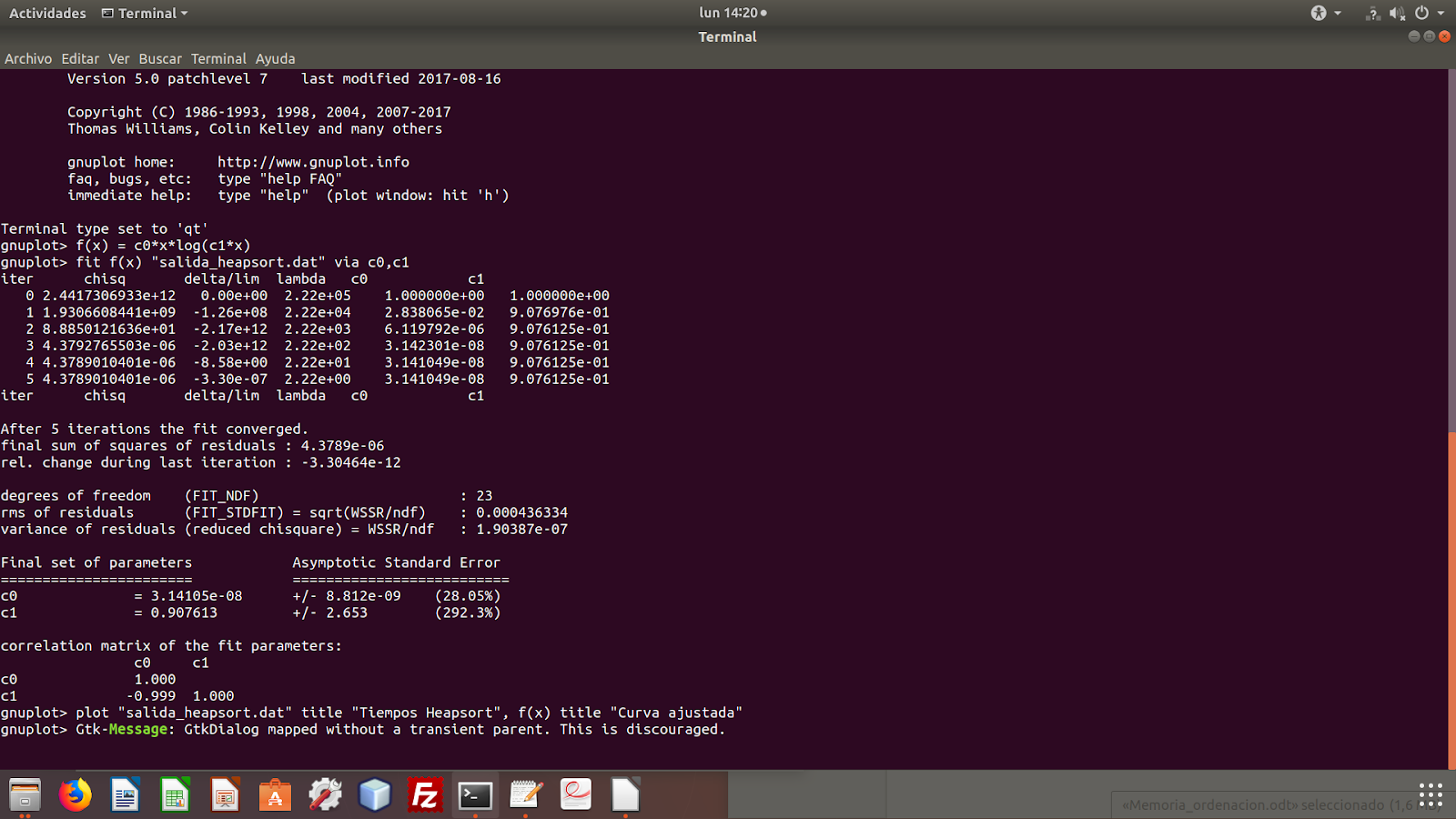
Nos centramos en el valor de a0 en la tabla "final set of parameters", que es igual a 1.81187e-09.

Finalmente, representamos la función teórica ajustada y la empírica en una misma gráfica, con el fin de ver hasta qué punto coinciden:



**Heapsort**

Pasamos a ajustar la función teórica de este algoritmo, que en este caso será f(x)=c0\*x\*log(c1\*x) al ser O(nlog(n)), a la empírica por medio de gnuplot y una vez definida dicha función hacemos lo siguiente:



El proceso de ajuste es en todo equivalente al realizado con Mergesort. Fijándonos de nuevo en el par de tablas final que se obtiene como resultado:

Final set of parameters Asymptotic Standard Error

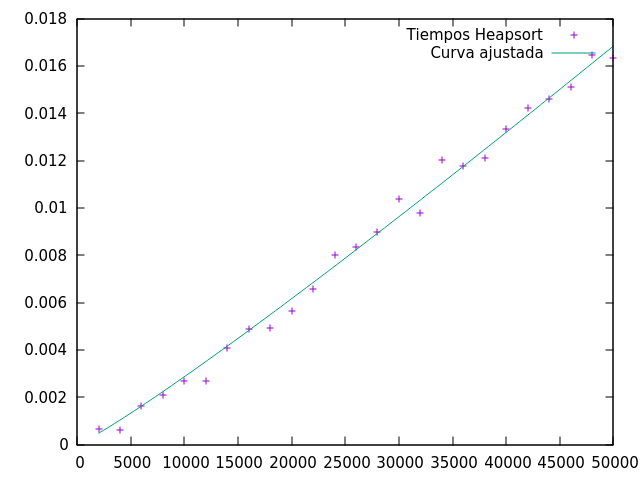
======================= ==========================

c0 = 3.14105e-08 +/- 8.812e-09 (28.05%)

c1 = 0.907613 +/- 2.653 (292.3%)

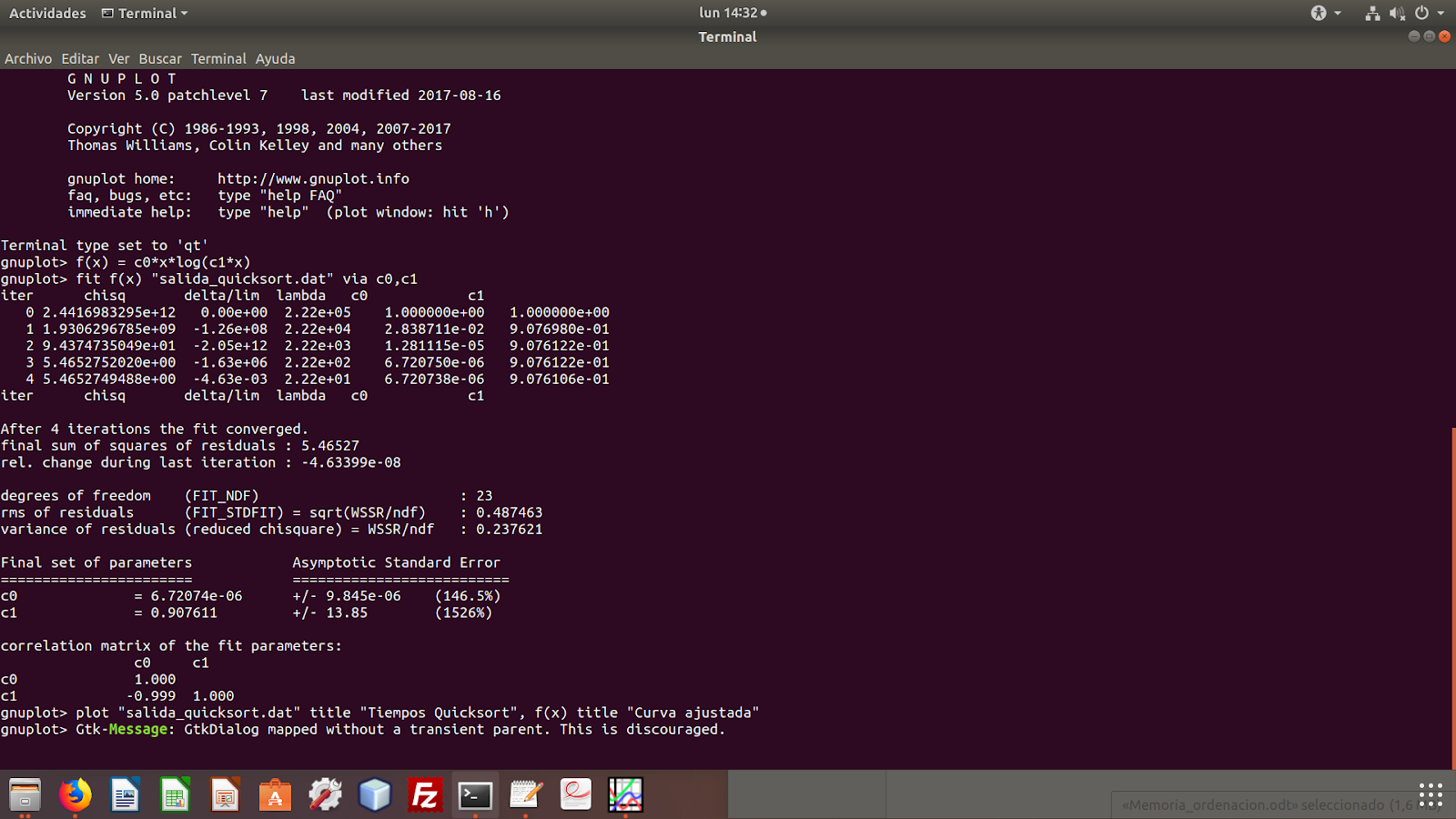
Nos centramos en el valor de c0 en la tabla "final set of parameters", que es igual a 3.14105e-08 que, comparado con el valor de la constante oculta de Mergesort (que es el otro algoritmo de O(nlog(n))), es mayor. Esto significa que Heapsort tiene peor rendimiento que Mergesort pese a tener el mismo orden de eficiencia.

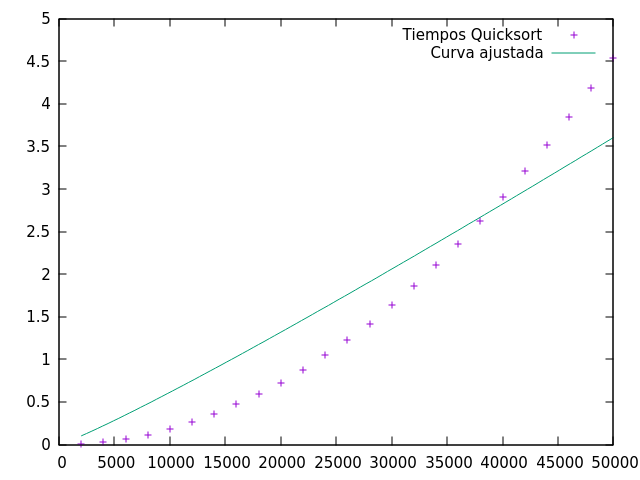
Finalmente, representamos la función teórica ajustada y la empírica en una misma gráfica, con el fin de ver hasta qué punto coinciden:



Por último, vamos a probar a intercambiar funciones empíricas y teóricas. Se va a realizar un ajuste del mismo modo que se ha venido describiendo, solo que en esta ocasión vamos a ignorar las conclusiones del análisis teórico deliberadamente para comprobar qué ocurre si ajustamos una función teórica a una empírica cuyo algoritmo no se corresponde con dicha función teórica.

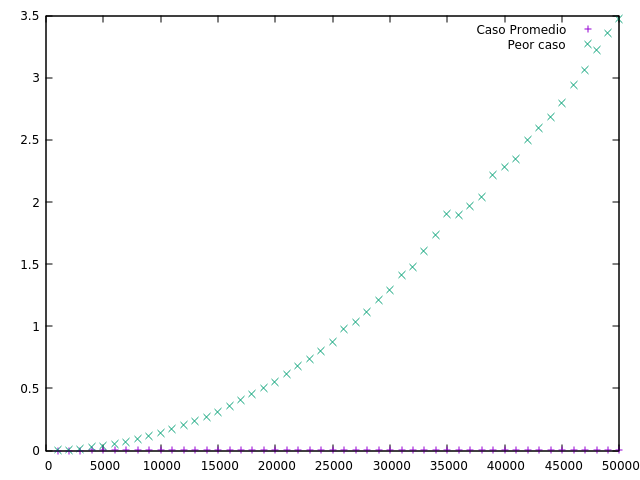
En este caso, adaptamos Quicksort, de O(n²), a f(x)=c0\*x\*log(c1\*x) como si fuera de O(nlog(n)):





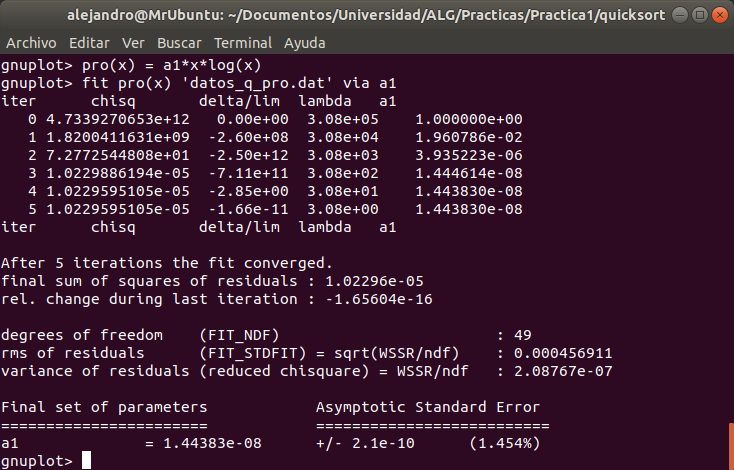
Se aprecia bastante diferencia entre una y otra curva, como era de esperar al ajustar con la función equivocada.

Puede sorprender que quicksort es un algoritmo catalogado como de eficiencia nlog(n) pero que se está comportando como uno de n². Esta paradoja se puede explicar con que nosotros estamos midiendo los peores casos. Quicksort, sin embargo tiene una gran variabilidad de eficiencia y el peor caso se aleja mucho de un caso promedio. Para ejemplificar, comparemos dos gráficas de la ejecución de quicksort:

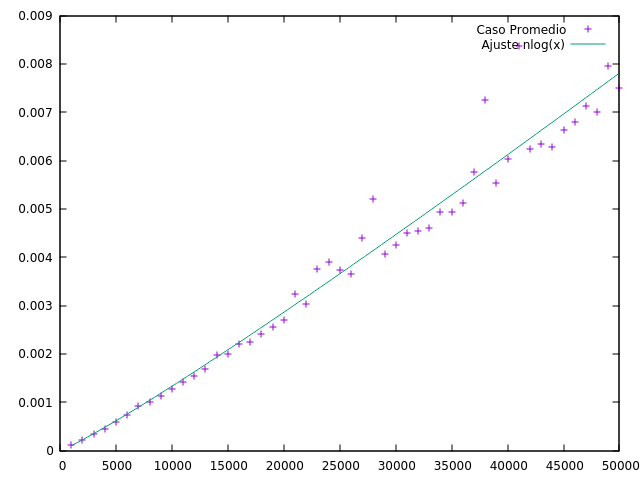


El peor caso es mucho más ineficiente que el caso promedio (tal es la diferencia que apenas se pueden distinguir los resultados de 0.

Si cogemos los resultados promedio y los ajustamos con gnuplot obtenemos:



Con esto podemos comprobar que, al contrario de lo que pasaba con los peores casos, el caso promedio se ajusta perfectamente a la gráfica de nlog(n).



**Algoritmo de Floyd**

**Regresión de f(x) con respecto a la ejecución de Floyd**

1. Definimos la función O(n^3) para hacer regresión con respecto a f(x)

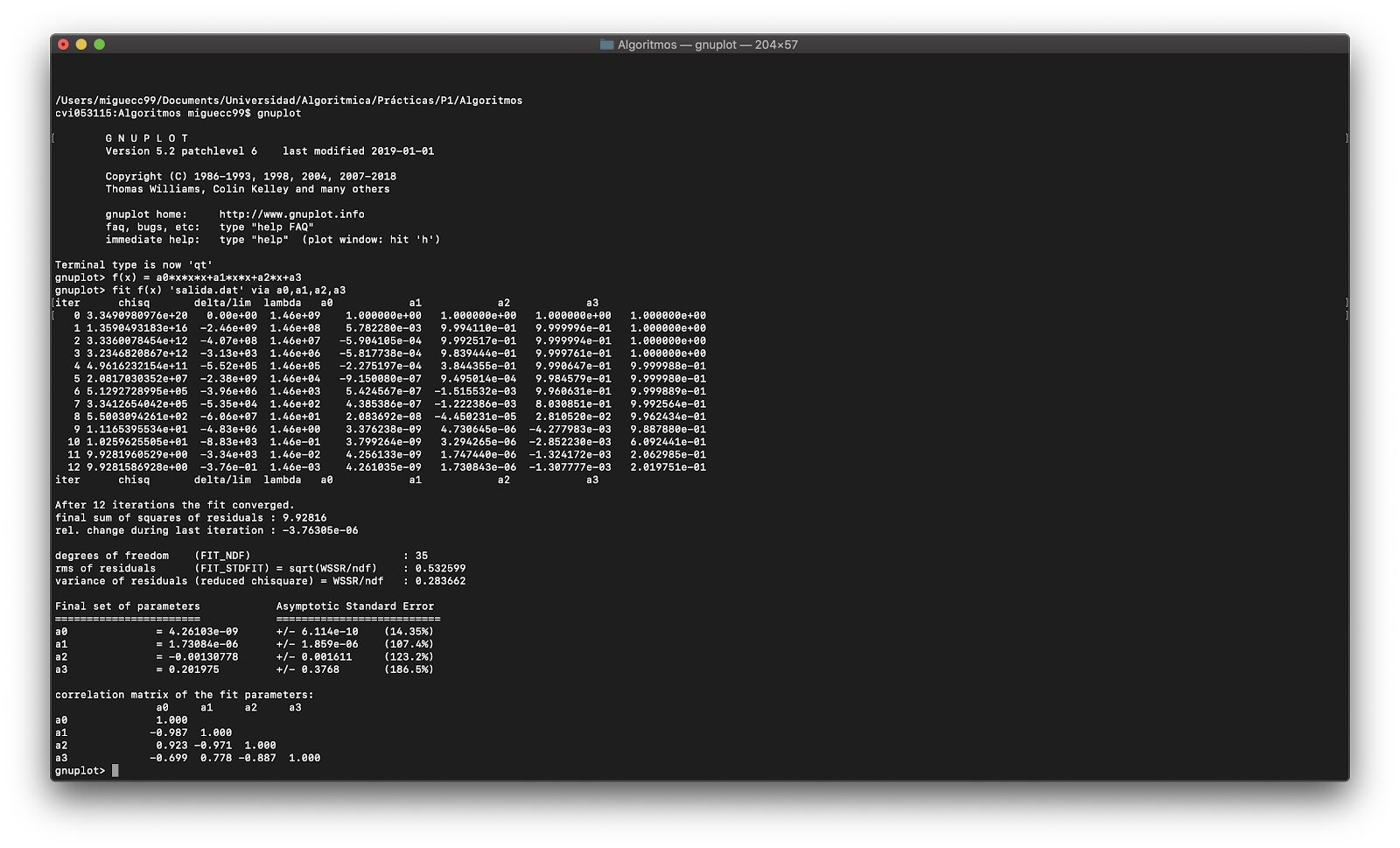
- gnuplot> f(x) = a0\*x\*x\*x+a1\*x\*x+a2\*x+a3

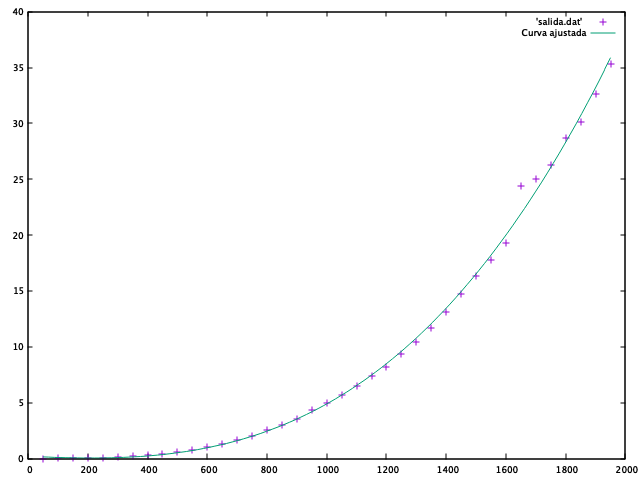
2) Indicamos la regresión a Gnuplot (figura 2):

- gnuplot> fit f(x) ’salida.dat’ via a0,a1,a2,a3

3) Ajustamos esa función a nuestros datos en un gráfico:

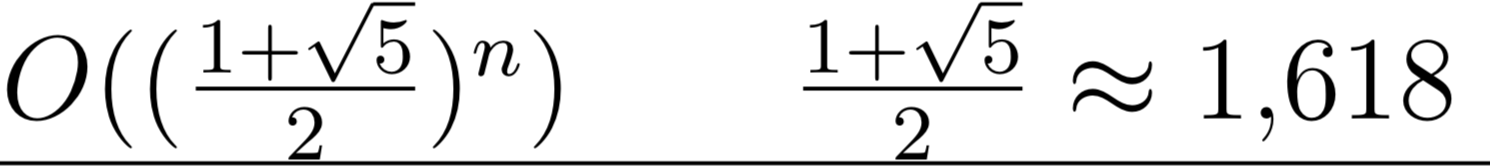
- gnuplot> plot ’salida.dat’, f(x) title ’Curva ajustada’

****

****

**Algoritmo de Fibonacci**

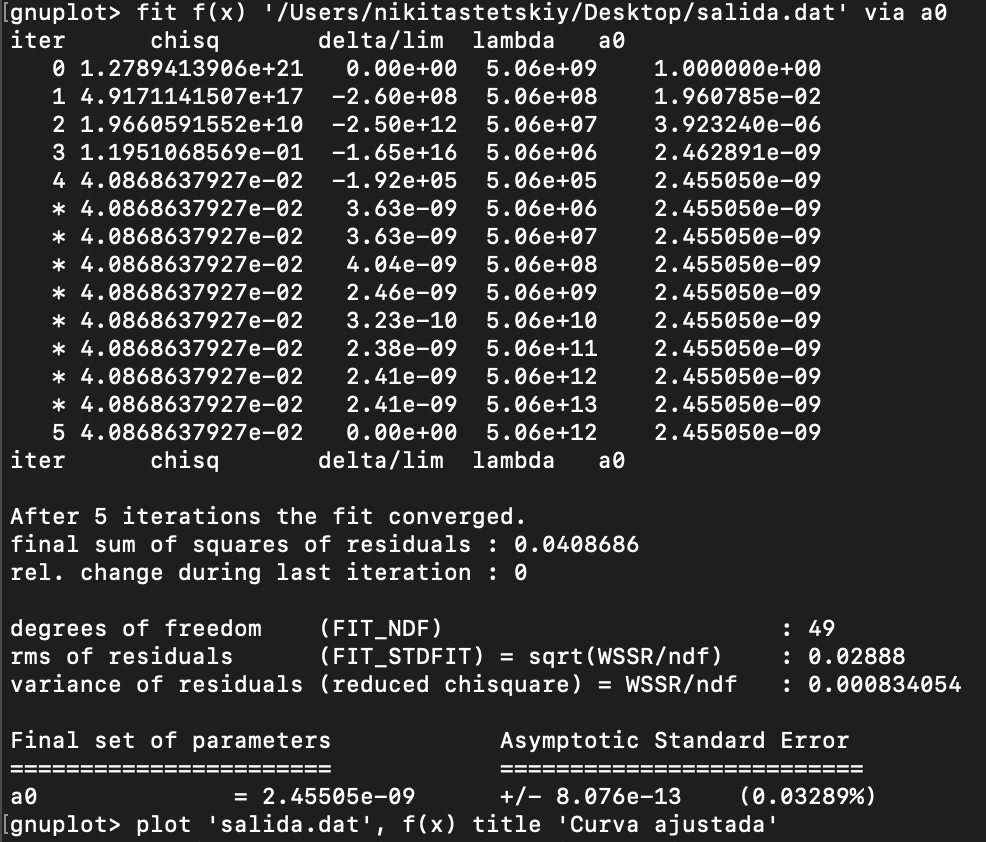
Usaremos de nuevo GNUPLOT pero de una manera distinta, para ello definimos directamente la funcion para poder comprarla con los datos obtenidos. Nuestra función en nuestro caso seria :

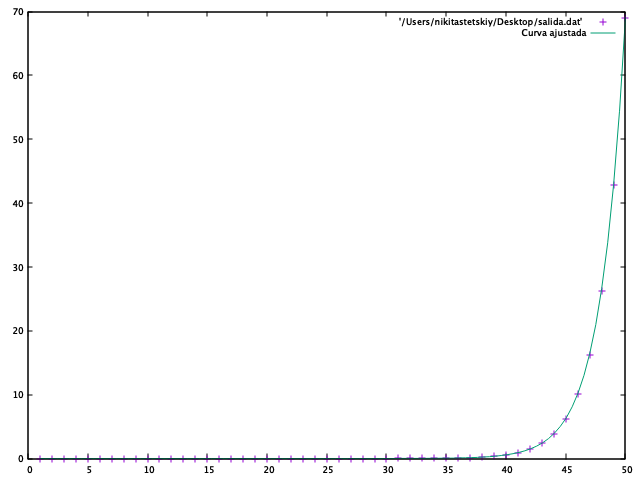


Por lo tanto escribimos en gnuplot:

- gnuplot> f(x) = 1.618\*\*x\*a0

- gnuplot> fit f(x) ’salida.dat’ via a0



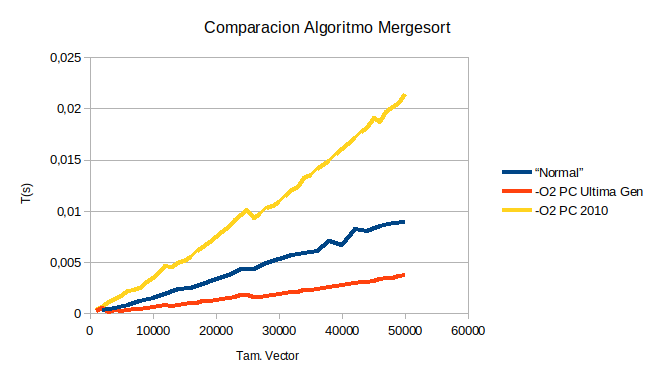
****

# PRINCIPIO DE INVARIANZA

Por último se va a estudiar la eficiencia empírica de los distintos algoritmos (1 algoritmo por orden de eficiencia) que se han visto durante la realización de la práctica, en función de parámetros externos, para así demostrar el principio de invarianza una vez más.

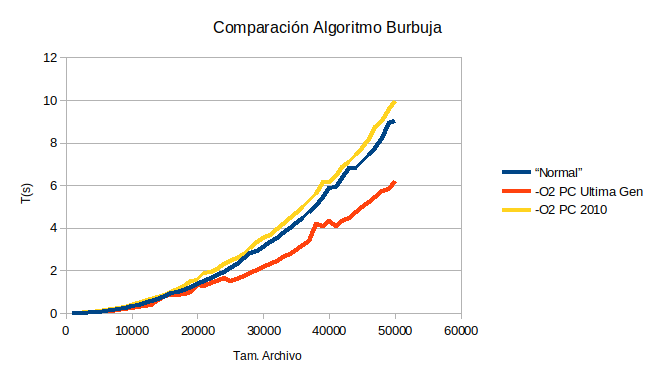
Para ello, se ha compilado el código de dichos algoritmos con un orden de optimización de 2 para después ser ejecutados en dos máquinas completamente distintas, un netbook de 2010 (características) y un PC de “Última Generación”, con un Intel i7-8750H de 6 cores y 16GB de RAM ddr4. Los resultados obtenidos en dichas ejecuciones se compararán a los resultados obtenidos en los apartados anteriores, estos serán los valores de referencia. Primero se estudiará el algoritmo Mergesort.

Como se puede observar, la función ligada a cada conjunto de datos es prácticamente la misma, y lo único que los diferencia son las constantes ocultas que se aplican en cada caso. En este caso estamos tratando con un algoritmo de orden n\*log n.

****

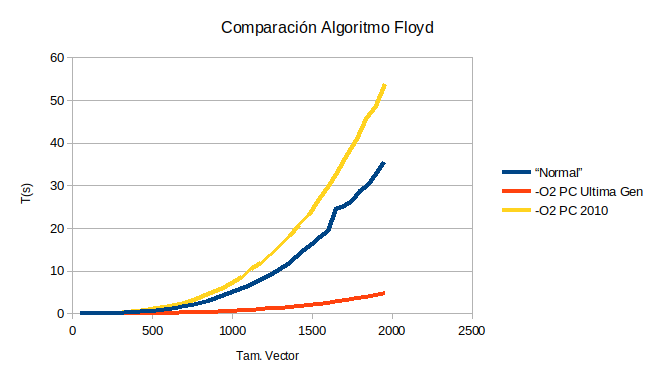
|  |  |  |  |
| --- | --- | --- | --- |
| **Tamaño del Vector** | **Resultados apartado**  **Eficiencia O(n\*log n)**  **Para alg. Mergesort** | **Resultados con Opción**  **de compilación -O2 en**  **Un PC de Última Gen.** | **Resultados con opción**  **de compilación -O2 en**  **Un netbook de 2010** |
| **2000** | **0,0003636** | **0,0005** | **0,000576** |
| **4000** | **0,0005512** | **0,000345** | **0,001374** |
| **6000** | **0,000852** | **0,000372** | **0,002162** |
| **8000** | **0,001181** | **0,000454** | **0,00251** |
| **10000** | **0,0014826** | **0,000607** | **0,003452** |
| **12000** | **0,0018514** | **0,000788** | **0,004579** |
| **14000** | **0,0023984** | **0,00081** | **0,004875** |
| **16000** | **0,002482** | **0,000968** | **0,005551** |
| **18000** | **0,0028684** | **0,001192** | **0,006525** |
| **20000** | **0,0033752** | **0,001301** | **0,007464** |
| **22000** | **0,0037046** | **0,001493** | **0,008447** |
| **24000** | **0,0043186** | **0,001781** | **0,009565** |
| **26000** | **0,0043752** | **0,001605** | **0,009286** |
| **28000** | **0,0048756** | **0,001725** | **0,010321** |
| **30000** | **0,005272** | **0,001899** | **0,010923** |
| **32000** | **0,005656** | **0,002073** | **0,012023** |
| **34000** | **0,0059076** | **0,002252** | **0,013238** |
| **36000** | **0,0061222** | **0,002403** | **0,014057** |
| **38000** | **0,0070676** | **0,002569** | **0,014875** |
| **40000** | **0,0067246** | **0,002775** | **0,016043** |
| **42000** | **0,008242** | **0,002938** | **0,01714** |
| **44000** | **0,0080756** | **0,003092** | **0,01805** |
| **46000** | **0,0085538** | **0,003352** | **0,018675** |
| **48000** | **0,008857** | **0,003503** | **0,020146** |
| **50000** | **0,0088754** | **0,00373** | **0,021451** |

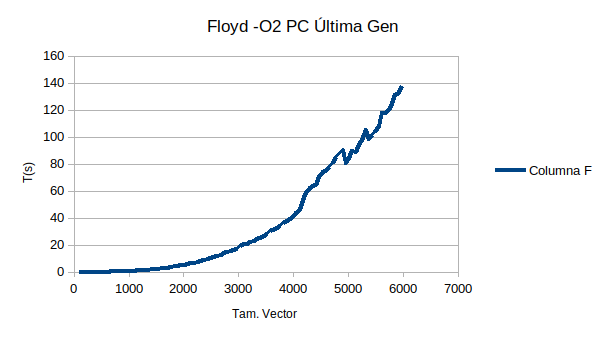
**En el caso del algoritmo de la burbuja este principio de invarianza se hace más evidente, ya que los resultados obtenidos al ejecutar dicho algoritmo en el netbook de 2010 son muy parecidos a nuestros resultados de referencia. Esto es debido a que los factores externos en ambos casos son muy similares, dando como resultado constantes ocultas bastante similares.**

****

|  |  |  |  |
| --- | --- | --- | --- |
| **Tamaño del Vector** | **Resultados apartado**  **Eficiencia O(n²)**  **Para alg. Burbuja** | **Resultados con Opción**  **de compilación -O2 en**  **Un PC de Última Gen.** | **Resultados con opción**  **de compilación -O2 en**  **Un netbook de 2010** |
| **1000** | **0,003477** | **0,002045** | **0,004288** |
| **2000** | **0,010923** | **0,008227** | **0,017096** |
| **3000** | **0,02389** | **0,018447** | **0,03822** |
| **4000** | **0,039762** | **0,037636** | **0,060265** |
| **5000** | **0,066352** | **0,052677** | **0,092496** |
| **6000** | **0,102998** | **0,085641** | **0,131187** |
| **7000** | **0,143562** | **0,101099** | **0,178621** |
| **8000** | **0,190101** | **0,14296** | **0,237353** |
| **9000** | **0,25063** | **0,187083** | **0,302893** |
| **10000** | **0,309138** | **0,216673** | **0,375711** |
| **11000** | **0,380806** | **0,265113** | **0,456547** |
| **12000** | **0,454806** | **0,324246** | **0,545315** |
| **13000** | **0,56574** | **0,39487** | **0,6521** |
| **14000** | **0,655213** | **0,629795** | **0,775618** |
| **15000** | **0,805358** | **0,780186** | **0,860922** |
| **16000** | **0,92982** | **0,83849** | **0,987199** |
| **17000** | **1,01116** | **0,869689** | **1,1251** |
| **18000** | **1,09543** | **0,89062** | **1,25118** |
| **19000** | **1,21768** | **0,989027** | **1,48765** |
| **20000** | **1,38015** | **1,33371** | **1,55189** |
| **21000** | **1,48211** | **1,26133** | **1,90029** |
| **22000** | **1,63134** | **1,41167** | **1,94107** |
| **23000** | **1,79689** | **1,52947** | **2,08148** |
| **24000** | **1,95051** | **1,65697** | **2,28982** |
| **25000** | **2,12165** | **1,51364** | **2,43216** |
| **26000** | **2,30391** | **1,61492** | **2,58994** |
| **27000** | **2,58852** | **1,74251** | **2,78916** |
| **28000** | **2,79554** | **1,894** | **3,06734** |
| **29000** | **2,93555** | **2,02718** | **3,31478** |
| **30000** | **3,08992** | **2,17175** | **3,54627** |
| **31000** | **3,31357** | **2,29164** | **3,68478** |
| **32000** | **3,53173** | **2,45056** | **3,93125** |
| **33000** | **3,76348** | **2,63495** | **4,17667** |
| **34000** | **4,01057** | **2,79011** | **4,43853** |
| **35000** | **4,23865** | **2,97355** | **4,6997** |
| **36000** | **4,47823** | **3,20899** | **4,97002** |
| **37000** | **4,74606** | **3,37127** | **5,24622** |
| **38000** | **5,02114** | **4,16778** | **5,56787** |
| **39000** | **5,44318** | **4,07667** | **6,1477** |
| **40000** | **5,85292** | **4,34394** | **6,12822** |
| **41000** | **5,92983** | **4,08932** | **6,44425** |
| **42000** | **6,34013** | **4,30547** | **6,83151** |
| **43000** | **6,82234** | **4,47682** | **7,09049** |
| **44000** | **6,8298** | **4,7404** | **7,43626** |
| **45000** | **7,07407** | **4,95868** | **7,76322** |
| **46000** | **7,40308** | **5,19974** | **8,16304** |
| **47000** | **7,74252** | **5,42435** | **8,74267** |
| **48000** | **8,21199** | **5,70972** | **9,00664** |
| **49000** | **8,93191** | **5,8189** | **9,59427** |
| **50000** | **9,00074** | **6,17558** | **9,93562** |

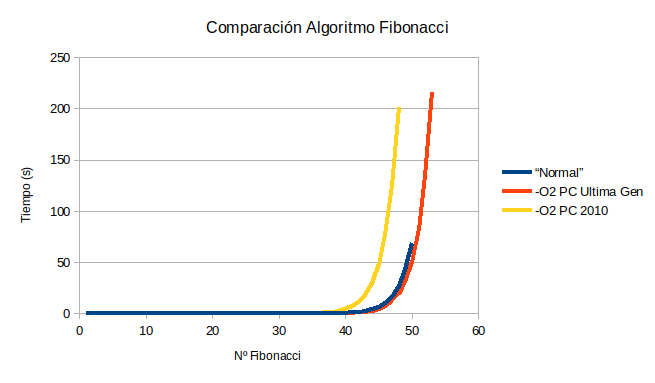
El caso del algoritmo de Floyd es algo más complicado, ya que la ejecución de dicho algoritmo en el PC de última generación daba resultados muy inferiores a los obtenidos con las otras 2 ejecuciones, es por esto que en la gráfica conjunta no puede apreciarse su crecimiento. Sin embargo, si ponemos dichos datos en una gráfica separada se puede ver perfectamente que el crecimiento es el mismo que en el caso de las ejecuciones anteriormente mencionadas.

****

****

|  |  |  |  |
| --- | --- | --- | --- |
| **Tamaño del Vector** | **Resultados apartado**  **Eficiencia Floyd** | **Resultados con Opción**  **de compilación -O2 en**  **Un PC de Última Gen.** | **Resultados con opción**  **de compilación -O2 en**  **Un netbook de 2010** |
| **50** | **0,00074** | **0,001871** | **0,008344** |
| **100** | **0,006217** | **0,002759** | **0,034077** |
| **150** | **0,016754** | **0,007161** | **0,077687** |
| **200** | **0,038553** | **0,01356** | **0,149504** |
| **250** | **0,073673** | **0,024328** | **0,269124** |
| **300** | **0,127003** | **0,038836** | **0,453232** |
| **350** | **0,205754** | **0,059465** | **0,725358** |
| **400** | **0,303654** | **0,085409** | **1,18096** |
| **450** | **0,436177** | **0,116811** | **1,4331** |
| **500** | **0,589987** | **0,155969** | **1,89578** |
| **550** | **0,801326** | **0,204255** | **2,50179** |
| **600** | **1,01493** | **0,283144** | **3,23336** |
| **650** | **1,28623** | **0,336229** | **3,96428** |
| **700** | **1,63048** | **0,412699** | **4,89457** |
| **750** | **2,0098** | **0,513834** | **5,95339** |
| **800** | **2,52284** | **0,612038** | **7,21373** |
| **850** | **2,96109** | **0,7446** | **8,51697** |
| **900** | **3,55805** | **0,896102** | **10,3905** |
| **950** | **4,33398** | **1,04803** | **11,7226** |
| **1000** | **5,00068** | **1,14649** | **13,5914** |
| **1050** | **5,69223** | **1,31387** | **15,6961** |
| **1100** | **6,45785** | **1,519** | **18,2163** |
| **1150** | **7,35483** | **1,75642** | **20,6532** |
| **1200** | **8,20163** | **2,03026** | **23,1118** |
| **1250** | **9,3145** | **2,23818** | **26,3281** |
| **1300** | **10,4202** | **2,52714** | **29,4975** |
| **1350** | **11,6516** | **2,8335** | **32,7819** |
| **1400** | **13,1334** | **3,13612** | **37,0828** |
| **1450** | **14,7526** | **3,51035** | **40,8316** |
| **1500** | **16,3643** | **3,94091** | **45,6201** |
| **1550** | **17,7944** | **4,28816** | **48,6349** |
| **1600** | **19,2537** | **4,75266** | **53,6751** |
| **1650** | **24,3572** | **5,19061** | **59,1792** |
| **1700** | **25,0607** | **5,68657** | **64,2079** |
| **1750** | **26,2477** | **6,19485** | **70,0971** |
| **1800** | **28,7303** | **6,72578** | **76,0655** |
| **1850** | **30,0915** | **7,31353** | **82,575** |
| **1900** | **32,6493** | **7,89234** | **88,9231** |
| **1950** | **35,3584** | **8,52718** | **96,9076** |

Por último se encuentra el algoritmo de Fibonacci. En este caso se puede ver un caso contrario al visto con el algoritmo de la burbuja, nuestros valores de referencia se asemejan más a los resultados obtenidos tras la ejecución en el PC de última generación. Pese a esto, al igual que con el algoritmo de la burbuja, se produce una situación idónea ya que el principio de invarianza es más fácil ver.

****

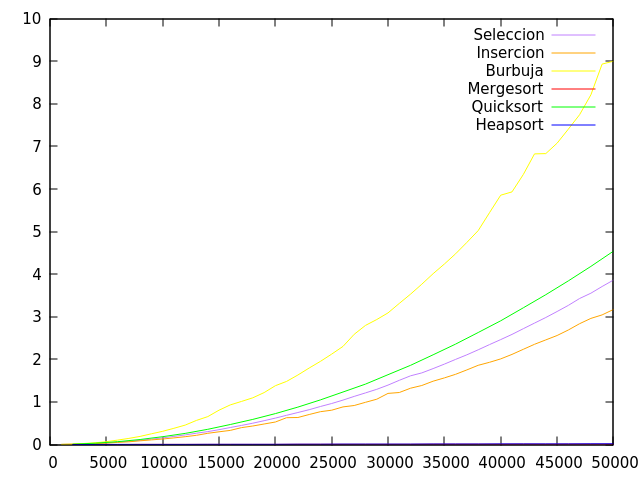
|  |  |  |  |
| --- | --- | --- | --- |
| **N.º de Fibonacci** | **Resultados apartado**  **Eficiencia Fibonacci** | **Resultados con Opción**  **de compilación -O2 en**  **un PC de Última Gen.** | **Resultados con opción de compilación -O2 en un netbook de 2010** |
| **1** | **1,66E-07** | **3E-06** | **1,1E-05** |
| **2** | **1,98E-07** | **4E-06** | **1E-05** |
| **3** | **2,72E-07** | **1E-06** | **1,3E-05** |
| **4** | **2,86E-07** | **1E-06** | **1,1E-05** |
| **5** | **3,67E-07** | **1E-06** | **1,1E-05** |
| **6** | **3,89E-07** | **1E-06** | **1,2E-05** |
| **7** | **5,46E-07** | **1E-06** | **1,1E-05** |
| **8** | **5,67E-07** | **1E-06** | **1E-05** |
| **9** | **8,2E-07** | **1E-06** | **1,4E-05** |
| **10** | **1,01E-06** | **1E-06** | **1,5E-05** |
| **11** | **1,61E-06** | **1E-06** | **1,1E-05** |
| **12** | **1,85E-06** | **2E-06** | **2E-05** |
| **13** | **2,52E-06** | **2E-06** | **2,2E-05** |
| **14** | **3,46E-06** | **3E-06** | **3,1E-05** |
| **15** | **5,58E-06** | **4E-06** | **4E-05** |
| **16** | **8,07E-06** | **1,4E-05** | **5,7E-05** |
| **17** | **1,33E-05** | **8E-06** | **9,4E-05** |
| **18** | **2,04E-05** | **1,2E-05** | **0,00014** |
| **19** | **3,3E-05** | **1,8E-05** | **0,000223** |
| **20** | **9,31E-05** | **2,9E-05** | **0,000361** |
| **21** | **6,4E-05** | **4,7E-05** | **0,000563** |
| **22** | **0,000139155** | **7,5E-05** | **0,000986** |
| **23** | **0,000196859** | **0,00012** | **0,00149** |
| **24** | **0,00033394** | **0,000189** | **0,002394** |
| **25** | **0,000638919** | **0,000314** | **0,004** |
| **26** | **0,000995018** | **0,000507** | **0,00649** |
| **27** | **0,00151573** | **0,00082** | **0,010592** |
| **28** | **0,00247577** | **0,001325** | **0,017056** |
| **29** | **0,00381071** | **0,002155** | **0,027286** |
| **30** | **0,00662628** | **0,003567** | **0,04284** |
| **31** | **0,0100557** | **0,005638** | **0,060272** |
| **32** | **0,0132889** | **0,008804** | **0,094737** |
| **33** | **0,0219859** | **0,01448** | **0,150602** |
| **34** | **0,0361832** | **0,023862** | **0,24775** |
| **35** | **0,0565988** | **0,039551** | **0,397229** |
| **36** | **0,0839683** | **0,062363** | **0,634355** |
| **37** | **0,135266** | **0,100374** | **1,03119** |
| **38** | **0,213905** | **0,162289** | **1,81574** |
| **39** | **0,346029** | **0,263629** | **2,68335** |
| **40** | **0,568877** | **0,418399** | **4,29729** |
| **41** | **0,915853** | **0,701978** | **6,88617** |
| **42** | **1,50638** | **1,12752** | **11,1514** |
| **43** | **2,39434** | **1,76563** | **18,0204** |
| **44** | **3,86634** | **2,83008** | **29,4717** |
| **45** | **6,22593** | **4,56437** | **47,8479** |
| **46** | **10,1041** | **7,39719** | **76,8855** |
| **47** | **16,264** | **11,9466** | **124,534** |
| **48** | **26,2641** | **19,3384** | **201,887** |
| **49** | **42,8112** | **31,2706** | **326,236** |
| **50** | **68,9658** | **50,5574** | **525,789** |

Como conclusión a este punto destacar que estos resultados, son muy reveladores porque una vez más demuestran el principio de invarianza del que se ha hecho tanto hincapié durante el primer tema de la asignatura. Así mismo, podemos asegurar que el principio de invarianza se va a cumplir siempre, independientemente de los factores externos.

Aún así este hecho no implica que no sea mejor utilizar una máquina más potente a la hora de realizar algún tipo de tarea que exija recursos a la máquina, ya que aunque el principio de invarianza se cumple, y el orden de eficiencia es el mismo, obviamente el tiempo que va a tardar en ejecutarse va a variar, beneficiando en este caso a la máquina mejor preparada para dicha tarea.

# COMPARACIÓN ENTRE TODOS LOS ALGORITMOS DE ORDENACIÓN

Representamos los seis algoritmos de ordenación propuestos en una sola gráfica para comparar su rendimiento:



Se puede observar de forma directa la gran diferencia que existe en el tiempo consumido. Mientras que los algoritmos de O(n²), es decir, burbuja, inserción, quicksort y selección, tienen una pendiente pronunciada y alcanzan tiempos de ejecución bastante altos en los tamaños mayores, los de O(nlog(n)), Mergesort y Heapsort, tienen un ritmo de crecimiento mucho más lento y se quedan muy lejos de los tiempos de ejecución de los otros cuatro algoritmos.