Universidad de Granada

E.T.S. DE INGENIERÍA INFORMÁTICA Y TELECOMUNICACIÓN

Departamento de Ciencias de la Computación e Inteligencia Artificial

Algorítmica

Práctica 1: Análisis de Eficiencia de Algoritmos

Curso 2017-2018
Grado en Ingeniería Informática

Álvaro García Jaén
Francisco José González García
Práxedes Martínez Moreno
Ignacio Martínez Rodríguez
Pablo Robles Molina

INDICE

1. ALGORITMOS DE EFICIENCIA O(n²)

CÁLCULO DE LA EFICIENCIA EMPÍRICA

Para llevar a cabo el cálculo de la eficiencia empírica, en primer lugar mediremos el tiempo empleado para cada tamaño. De esta manera, obtendremos como salida una tabla con los diferentes tamaños y el tiempo que ha tardado cada uno. En este caso para medir los tiempos, ya que no estamos limitados a tamaños de entrada pequeños, haremos uso de *chrono*, una biblioteca que nos otorga gran precisión en lo que a medidas de tiempos se refiere.

Como ya se ha expuesto, para analizar la eficiencia de nuestro algoritmo elegido, lo ejecutaremos para distintos tamaños del vector a ordenar. Una vez obtengamos las salidas, obtendremos la gráfica correspondiente mediante el software *gnuplot*. Para automatizar la tarea, haremos uso de un script sencillo cuya función será ejecutar el algoritmo para los diferentes tamaños. Los mencionados tamaños que hemos elegido se encuentran en el rango de 100 a 10000 y los iremos recorriendo con un paso de 300.

Para conseguir unas mediciones de tiempos de mayor precisión, para cada algoritmo, hemos ejecutado cada tamaño cien veces. A continuación, hemos almacenado los tiempos en un vector de *durations<double, std::milli>* que luego hemos empleado para calcular la mediana.[1]

CÁLCULO DE LA EFICIENCIA HÍBRIDA

En este apartado calcularemos las constantes de valor desconocido que aparecen en la expresión asociada a la eficiencia de los algoritmos de ordenación obtenida teóricamente. En este caso, sabemos que es:

$$T(n) = a0*n*n + a1*n + a2$$

Hallaremos las constantes ocultas ajustando a los puntos obtenidos por los mínimos cuadrados. Así también podremos averiguar aproximadamente cuánto tiempo emplearán los algoritmos para una entrada de tamaño n.

Como podremos observar en el apartado “*Asymptotic Standard Error”* de los datos obtenidos de *GNUPLOT,* para la segunda y tercera constante, obtenemos un gran porcentaje de error, en la primera, sin embargo, es mínimo. Realmente no les daremos mucha importancia a estos errores producidos para a1 y a2 debido a que es la constante a0 la que más nos interesa, la más característica.

[1] ¿Por qué elegimos hacer la mediana y no la media? La razón es que esta última tiene la gran desventaja de verse afectada si un valor se eleva o desciende demasiado en comparación con el resto de la muestra. Es por esta razón que la mediana es considerada como una mejor referencia de un punto medio.

ALGORITMO DE INSERCIÓN

Si analizamos por encima el código de este algoritmo, podemos observar que es de orden $O(n^2)$, pues consta de dos bucles anidados:

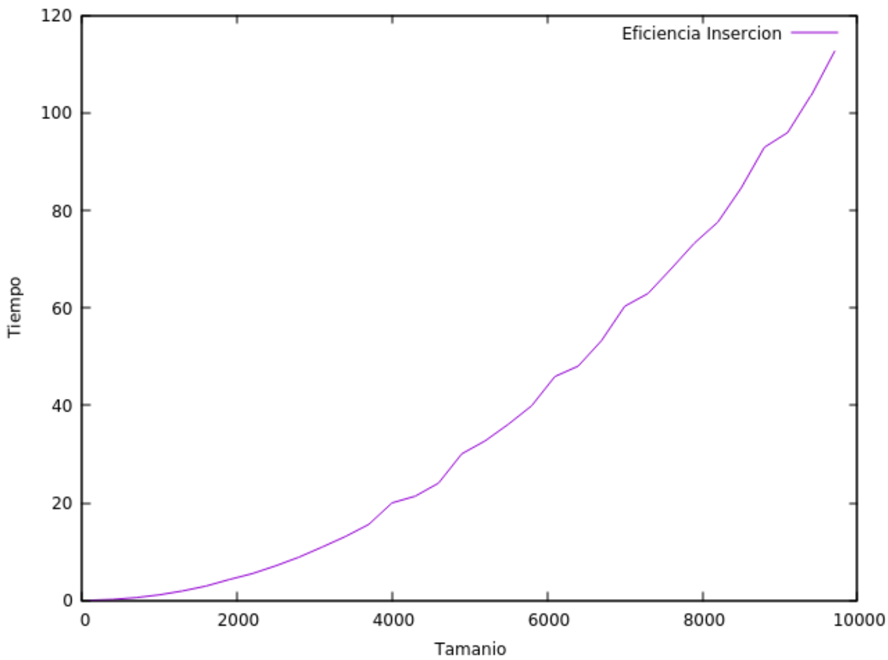
```
static void insercion_lims(int T[], int inicial, int final)
{
        int i, j;
        int aux;
        for (i = inicial + 1; i < final; i++) {                O(n²)
                j = i;
                while ((T[j] < T[j-1]) && (j > 0)) {           O(n)
                        aux = T[j];
                        T[j] = T[j-1];
                        T[j-1] = aux;
                        j--;
                }
        }
}
```

El resto de algoritmos de este bloque siguen el mismo esquema de código modificando las operaciones que se hacen en el bucle interno así que solo se comentará en este punto el orden de eficiencia asociado al código durante este bloque.

a. Eficiencia empírica

Para el algoritmo de inserción hemos obtenido las salidas representadas en la tabla que encontramos a continuación. La gráfica la generamos con gnuplot a partir de dicha tabla.

Tamaño	Tiempo(ms)
100	0,018997
400	0,201002
700	0,579019
1000	1,15315
1300	1,94089
1600	2,92321
1900	4,27198
2200	5,51283
2500	7,10065
2800	8,87176
3100	10,9348
3400	13,1025
3700	15,589
4000	19,9958
4300	21,3625
4600	24,0529
4900	30,0697
5200	32,7272
5500	36,1417
5800	39,9314
6100	45,9216
6400	48,0513
6700	53,2654
7000	60,3208
7300	62,9419
7600	68,0132
7900	73,2851
8200	77,5785
8500	84,5497
8800	92,9203
9100	95,9472
9400	103,526
9700	112,564



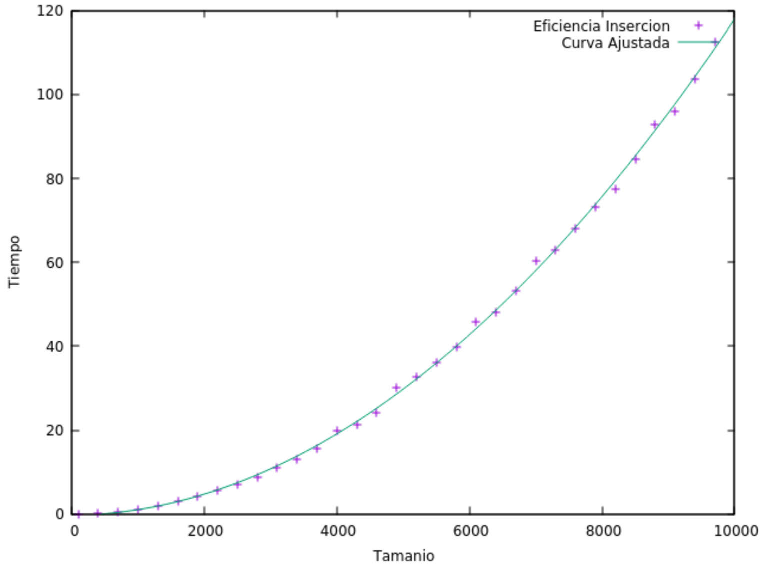
b. Eficiencia híbrida

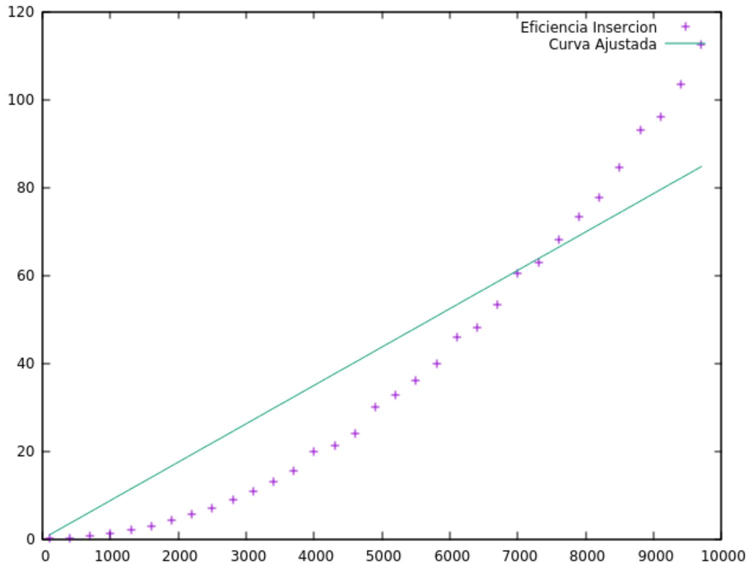
Calculando el valor de las constantes ocultas a partir del ajuste de la función por regresión por mínimos cuadrados y usando nuevamente gnuplot, obtenemos los siguientes datos:

Final set of parameters		Asymptotic Standard Error	
a0	= 1.15966e-06	+/- 2.343e-08	(2.021%)
a1	= 0.000220519	+/- 0.0002373	(107.6%)
a2	= -0.345331	+/- 0.5029	(145.6%)

c. Comparativa

Para apreciar más fácilmente este ajuste de los puntos y la función podemos observar la gráfica expuesta a continuación, la cual comparamos con la gráfica del ajuste de los datos anteriores y una función lineal.



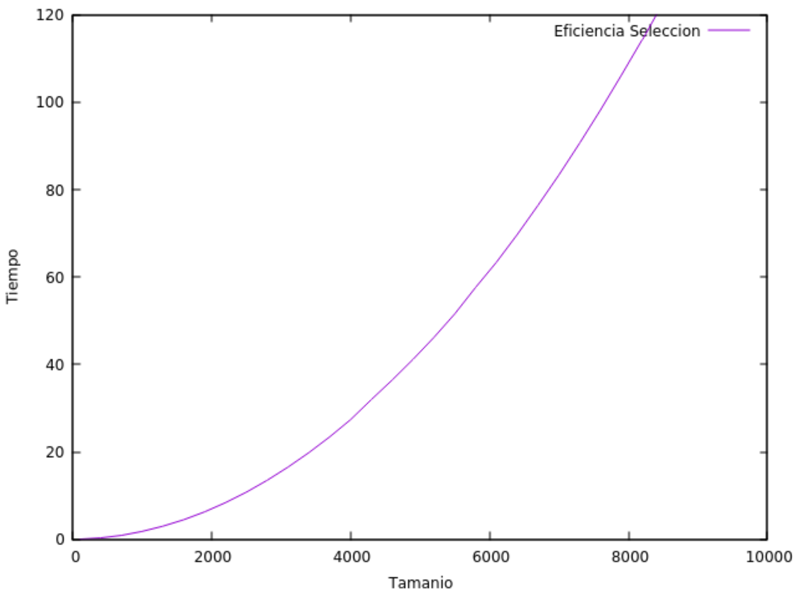


ALGORITMO DE SELECCIÓN

a. Eficiencia empírica

Para el análisis de este algoritmo, se han seguido los mismos pasos que con el algoritmo de inserción y la información obtenida ha sido la mostrada a continuación:

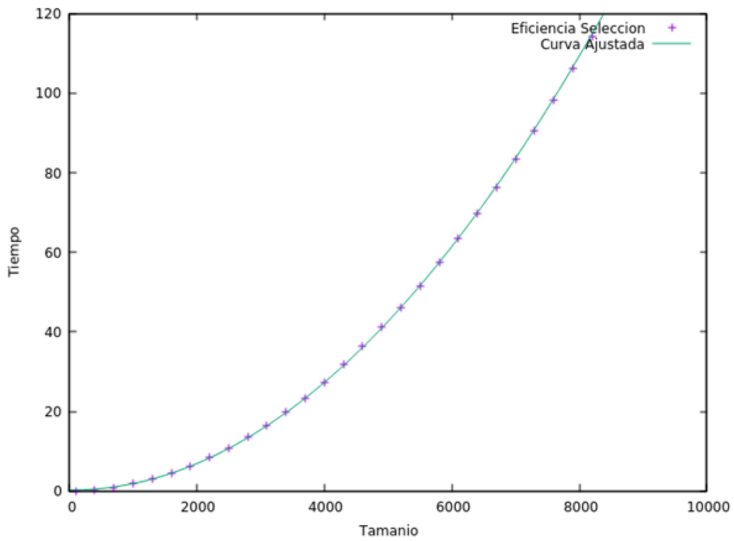
Tamaño	Tiempo(ms)
100	0,029747
400	0,303136
700	0,892665
1000	1,78741
1300	2,98411
1600	4,47788
1900	6,26304
2200	8,36624
2500	10,7788
2800	13,4803
3100	16,4879
3400	19,808
3700	23,4211
4000	27,3599
4300	31,9379
4600	36,423
4900	41,1592
5200	46,1751
5500	51,5625
5800	57,6133
6100	63,3416
6400	69,7053
6700	76,4284
7000	83,3656
7300	90,6833
7600	98,2396
7900	106,173
8200	114,341
8500	122,9
8800	132,758
9100	141,336
9400	152,435
9700	161,066



b. Eficiencia híbrida

Esta es la gráfica del ajuste y la información respectiva para el algoritmo de selección:

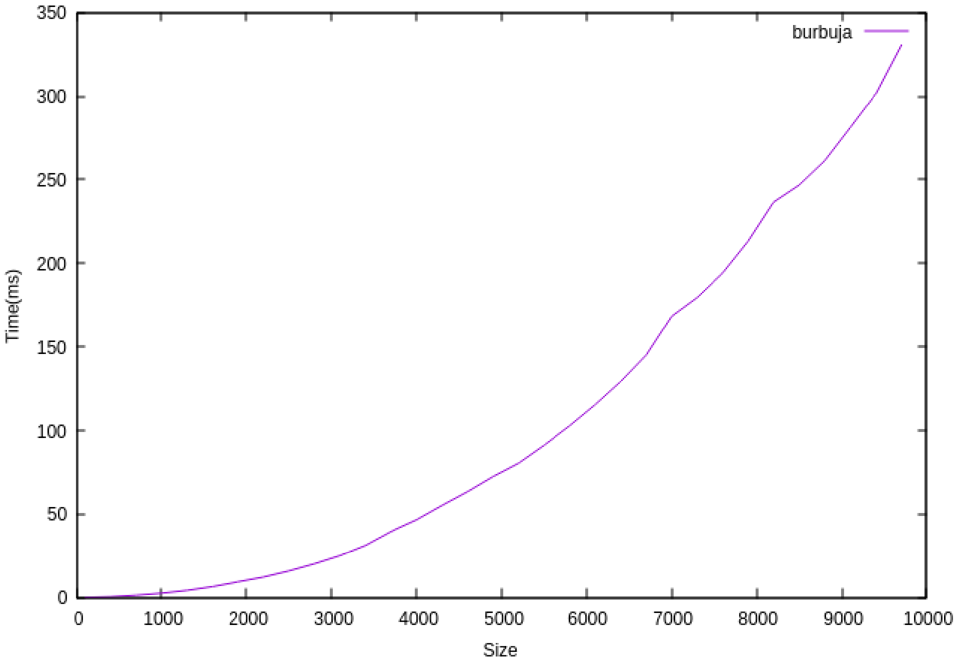
Final set of parameters		Asymptotic Standard Error	
a0	= 1.72088e-06	+/- 8.671e-09	(0.5039%)
a1	= -0.000131272	+/- 8.781e-05	(66.89%)
a2	= 0.254852	+/- 0.1861	(73.02%)



ALGORITMO DE BURBUJA

a. Eficiencia empírica

Tamaño	Tiempo(ms)
100	0,044867
400	0,451264
700	1,30026
1000	2,58446
1300	4,28663
1600	6,49397
1900	9,41792
2200	12,2197
2500	15,8972
2800	20,1811
3100	25,1522
3400	31,2958
3700	39,6335
4000	46,5931
4300	55,1726
4600	63,4595
4900	72,4835
5200	80,4542
5500	91,2036
5800	102,721
6100	115,294
6400	129,133
6700	144,983
7000	168,32
7300	179,576
7600	194,109
7900	213,047
8200	236,498
8500	246,749
8800	261,183
9100	281,075
9400	301,282
9700	330,404

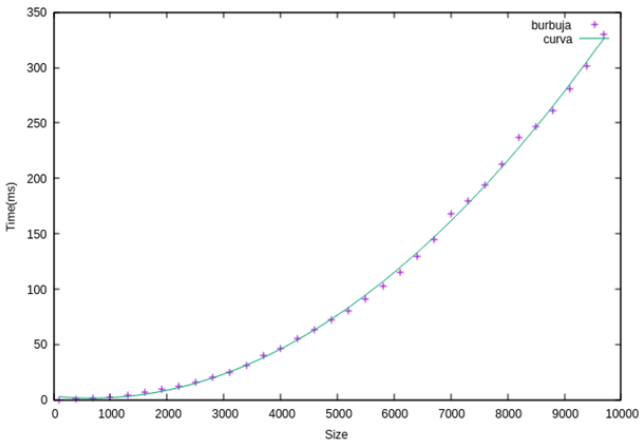


b. Eficiencia híbrida

Se obtienen los siguientes valores al ajustar los datos obtenidos anteriormente a la función f(x) = a0*x*x+a1*x+a2

Final set of parameters		Asymptotic Standard Error	
a0	= 3.98334e-06	+/- 7.47e-08	(1.875%)
a1	= -0.00527518	+/- 0.0007565	(14.34%)
a2	= 3.24064	+/- 1.603	(49.47%)

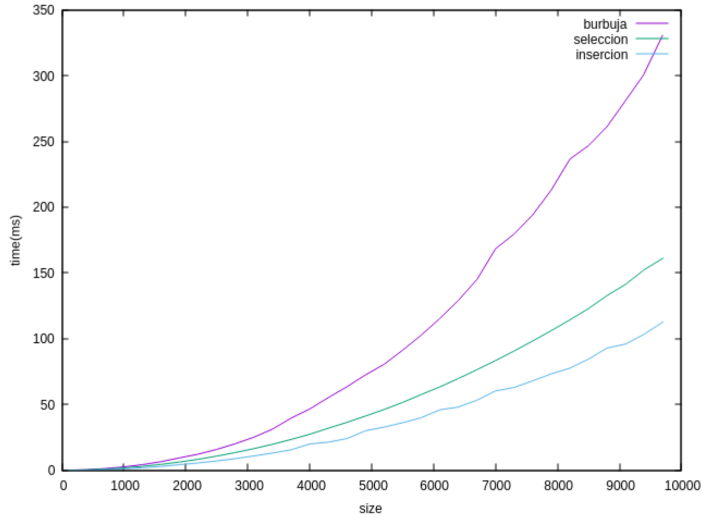
La gráfica asociada es la siguiente, que como se puede apreciar tiene un ajuste perfecto a nuestra función, con un error del 1.875% a nuestra variable más significativa.



c. Comparativas

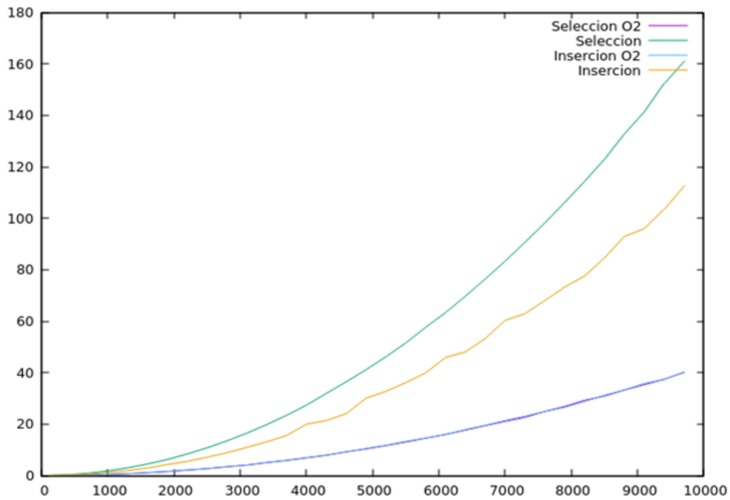
Comparativa entre los algoritmos O(n^2)

En esta comparativa podemos ver en una misma gráfica los 3 algoritmos de orden O(n^2) analizados con anterioridad. De esta comparativa podemos apreciar cómo el algoritmo de burbuja, aun siendo del mismo orden, es bastante más lento que los otros dos mientras que el algoritmo de inserción es el más rápido.

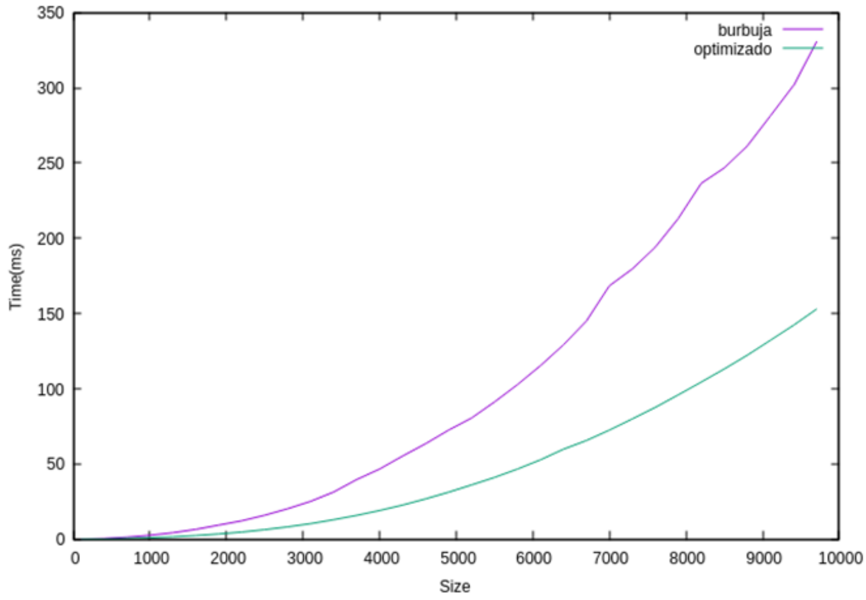


Comparativa con y sin optimización

Se ha realizado una comparativa entre los dos algoritmos de ordenación de orden n^2 y sus respectivas versiones con optimización O2 y como se puede apreciar en la gráfica que en las versiones sin optimización el algoritmo de selección es más lento que su competidor mientras que en sus versiones con optimización no existe una diferencia apreciable entre los tiempos de ejecución de ambos.



Comparativa de optimización de algoritmo burbuja



Hemos decidido optimizar el algoritmo de burbuja, que es el más lento de los tres algoritmos de ordenación, para comprobar la ganancia de velocidad que se obtiene al optimizarlo. Como podemos apreciar en la gráfica se obtiene una ganancia de más del doble con respecto a la ejecución original. Como comentario añadir que al código se ha añadido una porción para sacar por pantalla los elementos del vector una vez ordenado para que el compilador no quite la parte de código de la ordenación del vector y los tiempos sean coherentes.

2. ALGORITMOS DE EFICIENCIA O(nlg(n))

CÁLCULO DE LA EFICIENCIA EMPÍRICA

Es el turno de los algoritmos de O(n log n). Al tratarse de algoritmos de orden cuasilineal son muchos más rápidos que los estudiados anteriormente, los cuales eran cuadráticos. Por tanto, para hacer las mediciones, usaremos tamaños más grandes y con un paso más elevado: de 1000 a 9700 con un paso de 3000.

A continuación, de la misma manera que en el caso anterior, se obtiene una tabla con los resultados y los analizamos con gnuplot obteniendo así una gráfica.

EFICIENCIA HÍBRIDA

Para estudiar la eficiencia híbrida, lo primero que debemos hacer es ajustarla a una función. En el caso de estos algoritmos hemos usado:

f(x) = a0*x*log(x)

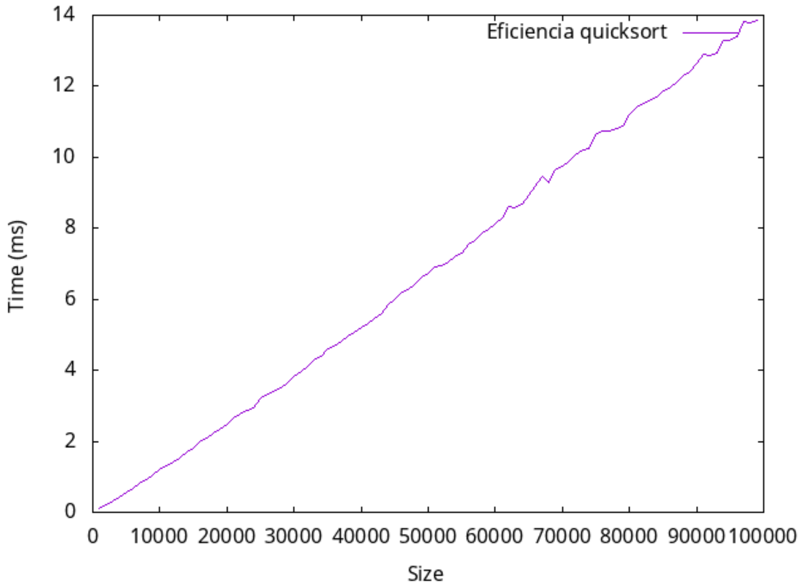
A partir de esta función y con la ayuda de fit de gnuplot, obtenemos el valor del coeficiente a0 y la curva ajusta. Lo más útil después de calcular la eficiencia híbrida es que nos queda como resultado una función a partir de la cual podemos predecir el tiempo que tarda dicho algoritmo para un tamaño concreto.

ALGORITMO QUICKSORT

a. Eficiencia empírica

A partir de los datos obtenidos, obtenemos la siguiente gráfica:

Tamaño	Tiempo(ms)
1000	0,09226578
4000	0,44128881
7000	0,80291796
10000	1,1955096
13000	1,5312903
16000	1,9718562
19000	2,3324038
22000	2,7493278
25000	3,230856
28000	3,4983627
31000	3,9762733
34000	4,4020308
37000	4,8190289
40000	5,2258262
43000	5,5833395
46000	6,2023307
49000	6,6151365
52000	6,954689
55000	7,3037725
58000	7,8652237
61000	8,2770307
64000	8,6958086
67000	9,4626056
70000	9,7636351
73000	10,1969965
76000	10,733433
79000	10,886826
82000	11,512773
85000	11,869777
88000	12,291895
91000	12,891397
94000	13,286371
97000	13,818452

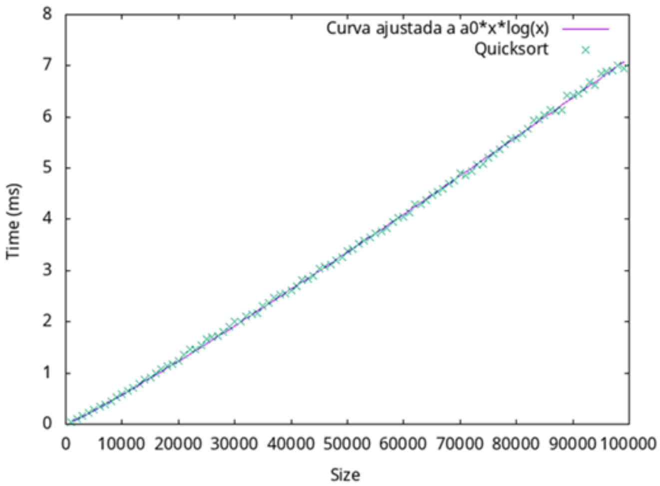


b. Eficiencia híbrida

El motivo por el que se ajusta a esta función es la siguiente:
el algoritmo en cada iteración divide el vector en dos partes, o en otras palabras, divide nuestro problema en la mitad. A cada una de estas partes, se le aplica un algoritmo de orden lineal, por tanto, nos quedaría O(nlogn).
Ajustamos el ordenamiento rápido a a0*x*log.
Una vez aclarado esto, gnuplot nos devuelve lo siguiente:

Final set of parameters	Asymptotic Standard Error
=======================	==========================
a0 = 6.21492e-06	+/- 6.51e-09 (0.1047%)

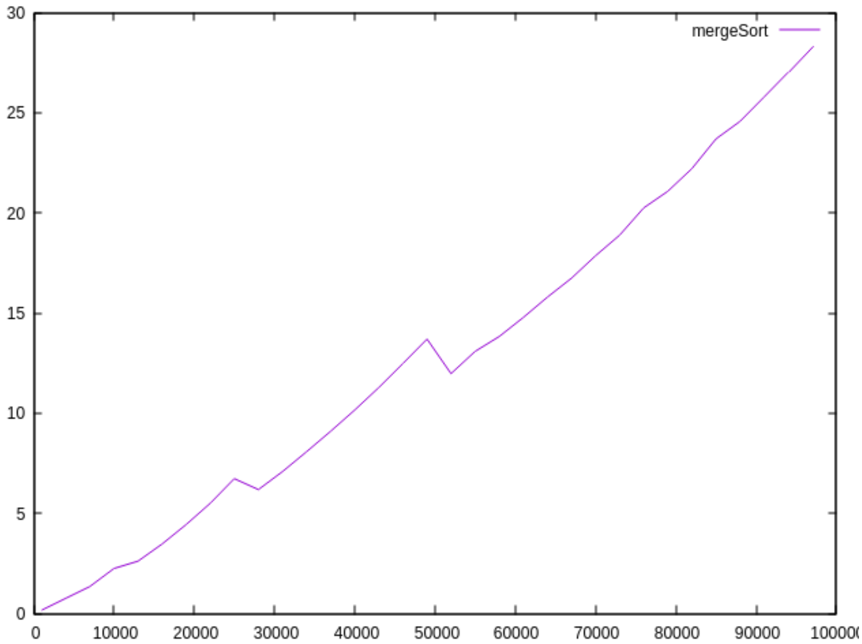
Nuestra función queda tal que así: f(x) = 6.21492e-06*x*log(x)



ALGORITMO MERGESORT

a. Eficiencia empírica

Tamaño	Tiempo(ms)
1000	0,054742
4000	0,223404
7000	0,333123
10000	0,468899
13000	0,69822
16000	0,847162
19000	1,01421
22000	1,1943
25000	1,85262
28000	1,70086
31000	1,83628
34000	2,01396
37000	2,182
40000	2,36243
43000	2,55231
46000	2,7582
49000	2,96541
52000	3,49397
55000	3,68855
58000	3,83114
61000	4,07245
64000	4,23567
67000	4,39933
70000	4,70587
73000	5,24299
76000	5,26641
79000	5,52066
82000	5,79934
85000	6,0796
88000	6,09011
91000	5,9936
94000	6,19331
97000	6,36997

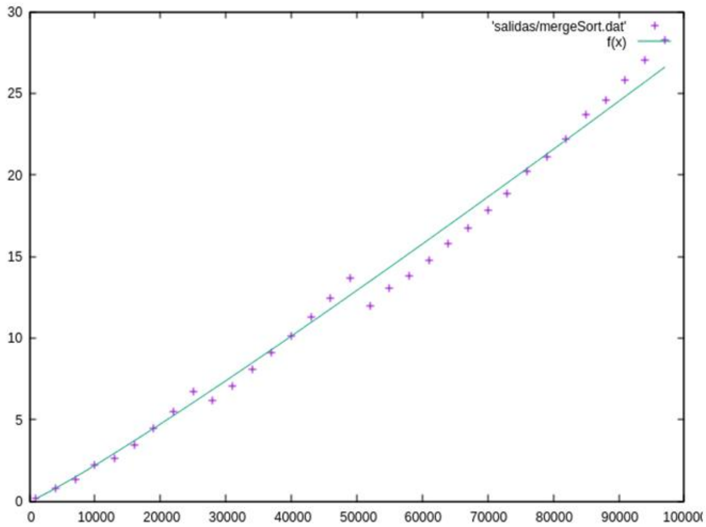


b. Eficiencia híbrida

Se obtienen los siguientes valores al ajustar los datos obtenidos anteriormente a la función g(x) = c0*x*log(x):

Final set of parameters		Asymptotic Standard Error	
======================		=========================	
a0	= 2.38757e-05	+/- 2.2e-07	(0.9213%)

En este caso no hemos tenido en cuenta el término de menor orden ya que para nuestro objetivo de calcular la cota superior nos basta con analizar el término de c0*n*log(n) y hemos obtenido un ajuste de 0.7781% con los datos obtenidos anteriormente.

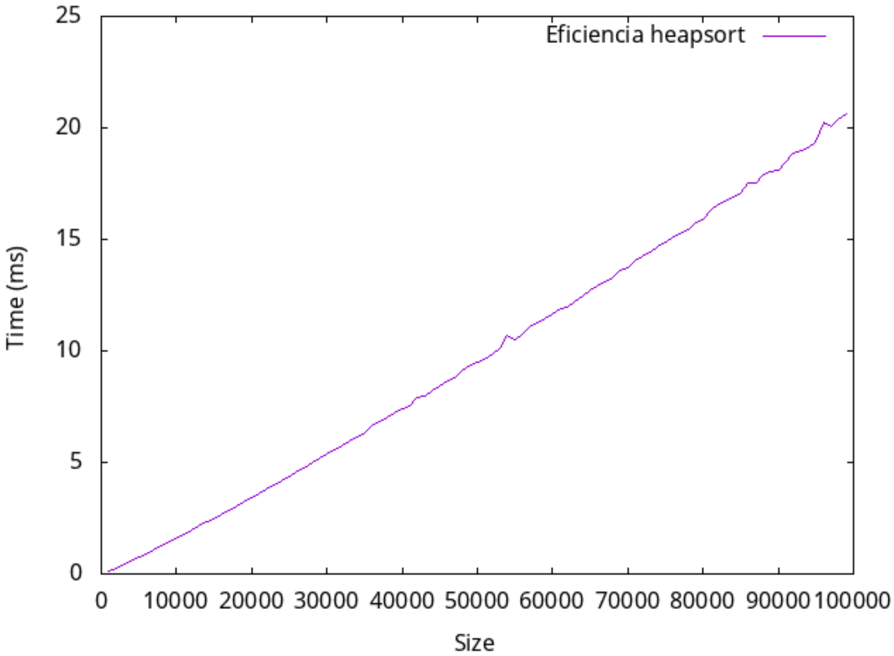


ALGORITMO HEAPSORT

a. Eficiencia empírica

La gráfica que obtenemos a partir de la salida es la siguiente:

Tamaño	Tiempo(ms)
1000	0,12037841
4000	0,58227857
7000	1,0864724
10000	1,5936302
13000	2,1490249
16000	2,6596617
19000	3,2007926
22000	3,7901708
25000	4,3649581
28000	5,0197501
31000	5,5643928
34000	6,1455152
37000	6,848974
40000	7,4264587
43000	7,9935542
46000	8,6506421
49000	9,3451443
52000	9,8565943
55000	10,491525
58000	11,2557
61000	11,882194
64000	12,473078
67000	13,07127
70000	13,734465
73000	14,461762
76000	15,109737
79000	15,771536
82000	16,610012
85000	17,103487
88000	17,909018
91000	18,471557
94000	19,094739
97000	20,068536

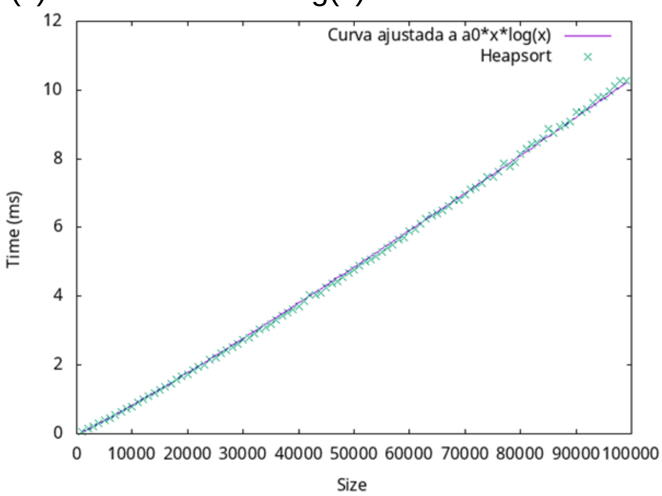


b. Eficiencia híbrida

Al ajustarlo a a0*x*log(x) obtenemos la siguiente salida mediante la orden fit de gnuplot:

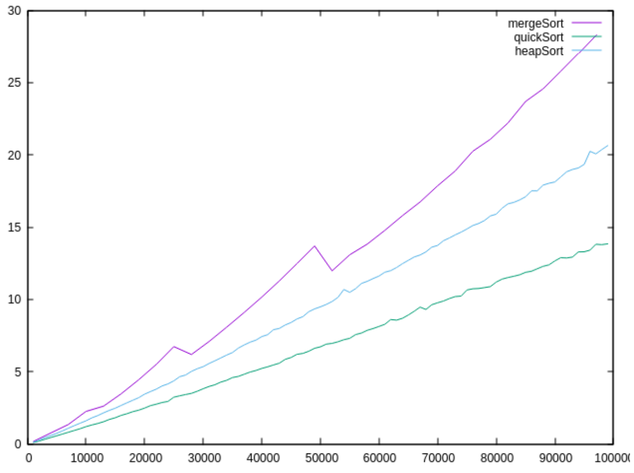
Final set of parameters   Asymptotic Standard Error
=======================   ==========================
a0   = 8.9647e-06   +/- 1.051e-08(0.1172%)

Por tanto, nuestra función resultante una vez hemos obtenido el valor del coeficiente a0 es la siguiente:
f(x) = 8.9647e-06*x*log(x)



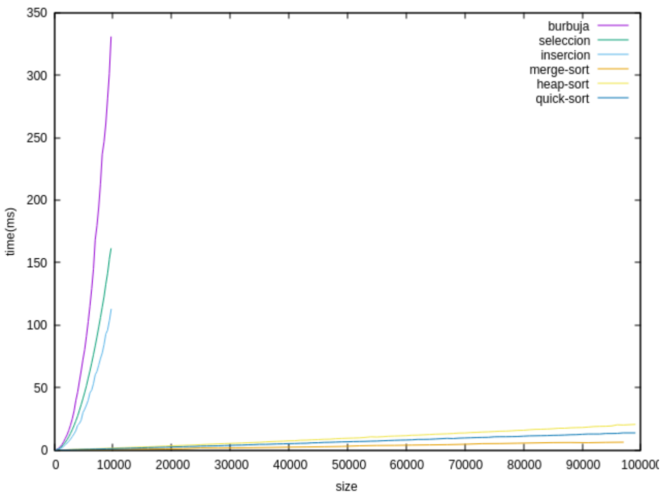
c. Comparativas

Comparativa entre los algoritmos de orden cuasilineal



En esta gráfica en la que comparamos los tres algoritmos de orden nlg(n) se puede observar que todos siguen la misma tendencia, pero como hemos comprobado en los puntos anteriores, debido a las constantes ocultas asociadas a cada algoritmo tenemos que el quicksort es el más veloz de los tres con una constante de 6.21492e-06 y por otro lado, el mergesort es el más lento con una constante asociada de 2.38757e-05.

Comparativa entre los algoritmos de ordenación



En esta comparativa final de los seis algoritmos de ordenación obtenemos una vista general de la eficiencia entre los dos grupos de algoritmos analizados, los de $O(n^2)$ y los de O(nlgn). Se puede observar la enorme diferencia de eficiencia que hay entre los dos grupos: mientras que los algoritmos cuadráticos tardan mucho tiempo en ordenar vectores pequeños, los algoritmos de orden cuasilineal son capaces de ordenar vectores mucho mayores en un tiempo casi imperceptible.

3. ALGORITMO DE EFICIENCIA O(n³)

ALGORITMO DE FLOYD

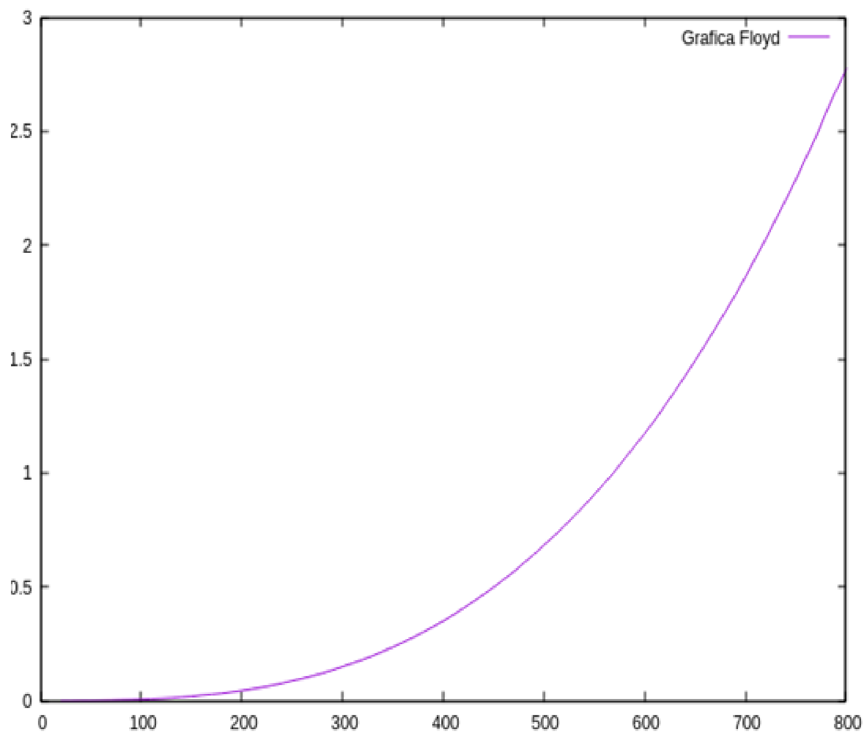
El algoritmo de Floyd como podemos ver fácilmente sobre el código tiene una eficiencia de O(n³):

```
void Floyd(int **M, int dim)
{
        for (int k = 0; k < dim; k++)                                   O(n³)
                for (int i = 0; i < dim;i++)                            O(n²)
                        for (int j = 0; j < dim;j++) {                  O(n)
                                int sum = M[i][k] + M[k][j];
                                M[i][j] = (M[i][j] > sum) ? sum : M[i][j];
                        }
}
```

Al ser O(n³), se han utilizado tamaños mayores que para el algoritmo de Hanoi pero más pequeños que para los algoritmos de ordenación. En concreto los tamaños oscilan entre 10 y 800.

<u>a. Eficiencia empírica</u>

Tamaño	Tiempo(seg)
20	0.000166
30	0.00035
40	0.000871
100	0.00638
110	0.007595
120	0.01062
130	0.01294
200	0.044344
210	0.051193
220	0.058786
300	0.148222
310	0.163352
320	0.179883
400	0.35182
410	0.378203
420	0.406157
430	0.436645
500	0.683943
510	0.724357
520	0.765658
600	1.17276
610	1.23703
620	1.29764
700	1.8663
710	1.94651
720	2.02504
780	2.58089
790	2.69022
800	2.78273



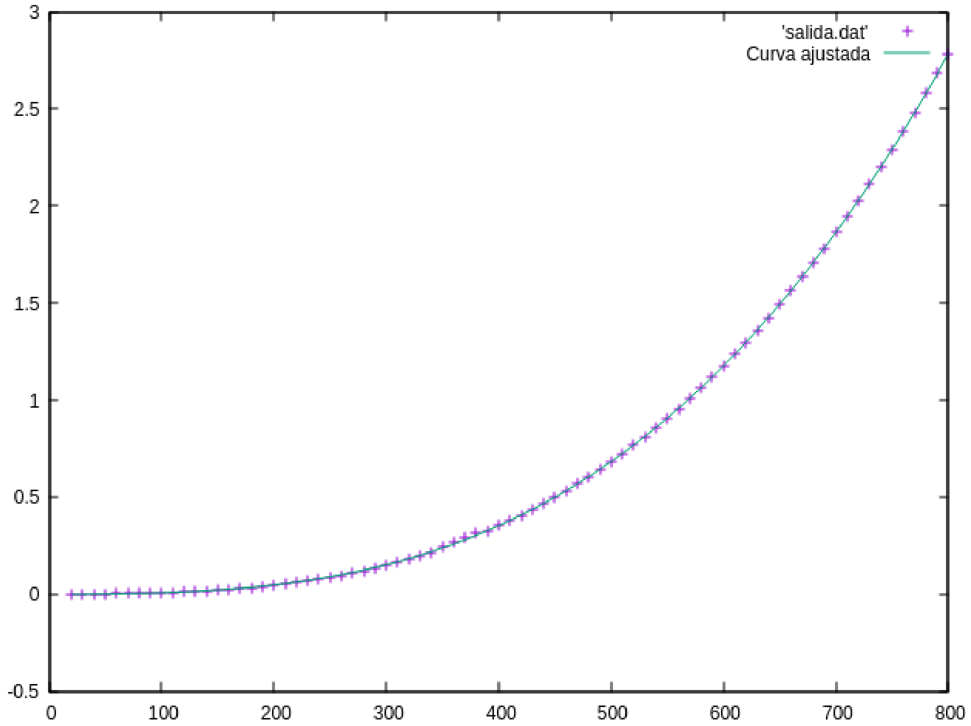
Como podemos ver con la gráfica del algoritmo O(n³) requiere de mucho más tiempo de ejecución para tamaños mayores.

b. Eficiencia híbrida

Al calcular la eficiencia híbrida nos da las siguientes constantes ocultas: (Hemos reducido el contenido del .log)

Final set of parameters	*Asymptotic Standard Error*
a0 = 5.48785e-09	*+/- 4.583e-11 (0.8351%)*
a1 = -1.03719e-07	*+/- 5.711e-08 (55.06%)*
a2 = 5.12045e-05	*+/- 2.036e-05 (39.75%)*
a3 = -0.00312988	*+/- 0.001966 (62.81%)*

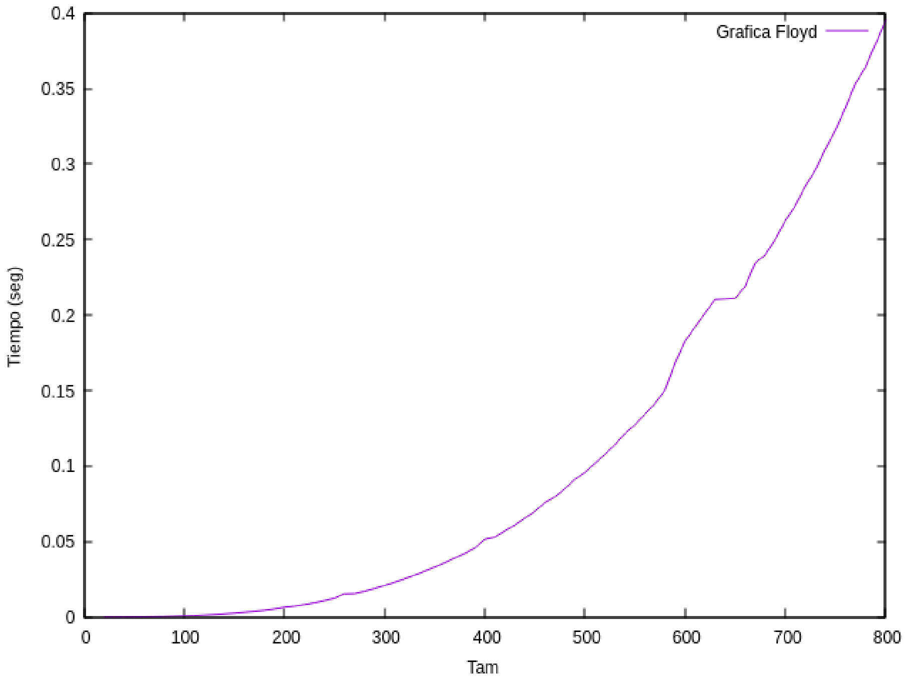
Y ajustando a f(x) = a0*x*x*x+a1*x*x+a2*x+a3 nos da la siguiente gráfica:



c. Variaciones

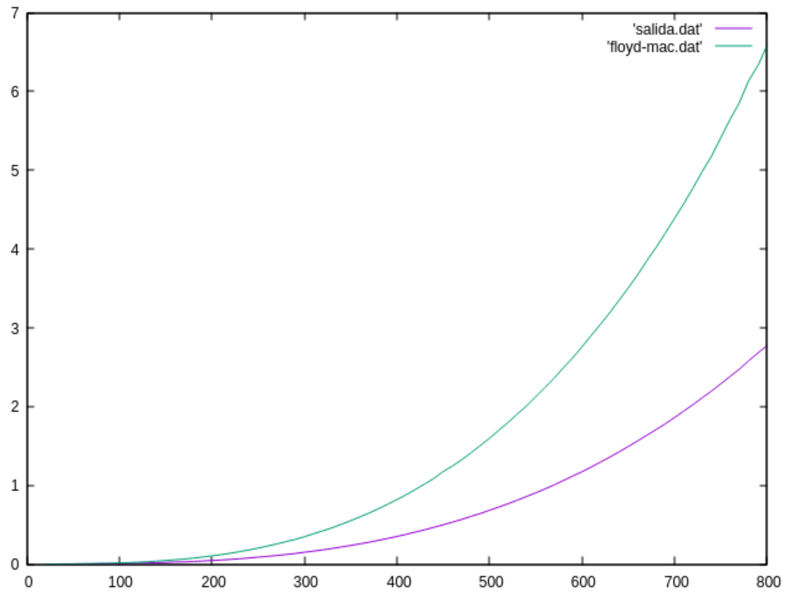
En este caso hemos compilado con optimización -O2 para comparar entre versiones, la gráfica resultante es:

Tamaño	Tiempo(seg)
20	9.00E-06
30	2.60E-05
40	6.40E-05
100	0.000962
110	0.001233
120	0.001605
130	0.00203
200	0.00631
210	0.007368
220	0.008349
300	0.021007
310	0.023088
320	0.025349
410	0.053049
420	0.057046
430	0.061029
500	0.09585
510	0.101888
520	0.108901
530	0.114083
610	0.197445
620	0.196238
630	0.191886
700	0.263618
710	0.274978
720	0.288814
780	0.367284
790	0.388104
800	0.402286



Como podemos ver el algoritmo con -O2 es casi 10 veces más rápido.

Una segunda variación que hemos hecho en este ejercicio ha sido ejecutarlo en distintos procesadores para ver las diferencias de tiempos:



salida.dat ha sido obtenido en un i5-7200K con 2,50 GHz y *floyd-mac.dat* ha sido obtenido en un Intel Core Duo 2 con 2,40 GHz y como se puede ver la diferencia es importante.

4. ALGORITMO DE EFICIENCIA O(2^n)

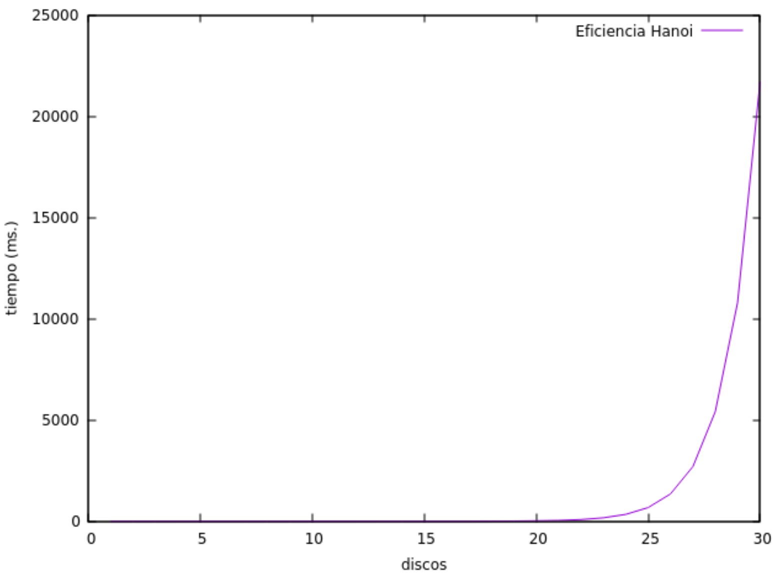
ALGORITMO DE HANOI

La función del algoritmo de las torres Hanoi es de orden 2^n porque una llamada a la función genera a su vez dos sub-llamadas a la misma función (hablamos por tanto de recursividad).

a. Eficiencia empírica

Dado que el algoritmo de Hanoi es de O(2^n) se elegirá una batería de datos desde 1 hasta 30 para probar el número de discos y sus correspondientes tiempos de ejecución.

Nº discos	Tiempo (ms)
1	0.00088
2	0.00092
3	0.00104
4	0.00128
5	0.00168
6	0.00248
7	0.00416
8	0.00652
9	0.012401
10	0.023761
11	0.050843
12	0.088805
13	0.17681
14	0.352779
15	0.672797
16	1.34783
17	2.69639
18	5.3883
19	10.7773
20	21.5522
21	42.8662
22	85.214
23	169.931
24	338.678
25	676.949
26	1353.34
27	2706.91
28	5411.98
29	10829
30	21686.6



En la gráfica se puede apreciar fácilmente el orden O(2^n) del algoritmo.

b. Eficiencia empírica

Para calcular la eficiencia híbrida realizamos un ajuste con la expresión general de la función 2^n.

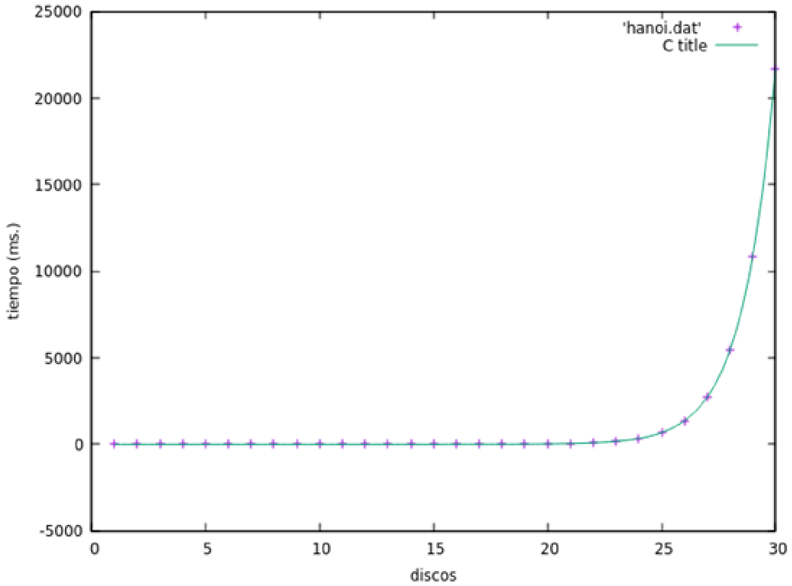
Con esto podemos calcular el valor de las constantes ocultas y así describir completamente la ecuación de tiempo. Para esto ajustaremos la función a un conjunto de puntos. Nuestra función será la expresión general y nuestros puntos serán los obtenidos en el análisis empírico y para el ajuste emplearemos regresión por mínimos cuadrados. Al realizar dicho ajuste obtenemos los valores de la constante oculta a0.

En el archivo *fit.log* generado tras el ajuste obtenemos:

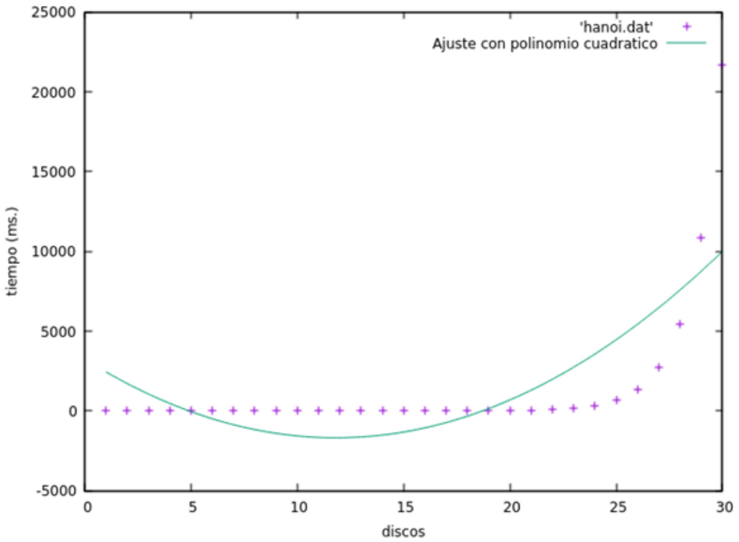
```
function used for fitting: f(x)=a0*2**x

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a0 = 2.01915e-05                   +/- 2.497e-09 (0.01237%)
```

En la siguientes gráficas podemos ver la calidad del ajuste respecto del cálculo teórico correspondiente (la expresión general) y de otro ajuste que no corresponde al orden del algoritmo (en este caso un polinomio cuadrático):



Ajuste con su expresión general correspondiente.



Ajuste con otro orden