Memoria de la práctica

Eficiencia de Algoritmos

-Algorítmica-

2ºDGIIM

Por:
Álvaro López Jiménez
Antonio Martín Ruíz
Miguel Ángel Robles Urquiza
Jesús Sánchez de Lechina Tejada

Índice

1.Introducción

A continuación expondremos el conjunto de los resultados obtenidos de esta práctica, los cuales complementaremos con una pequeña explicación y una exposición mediante diapositivas.

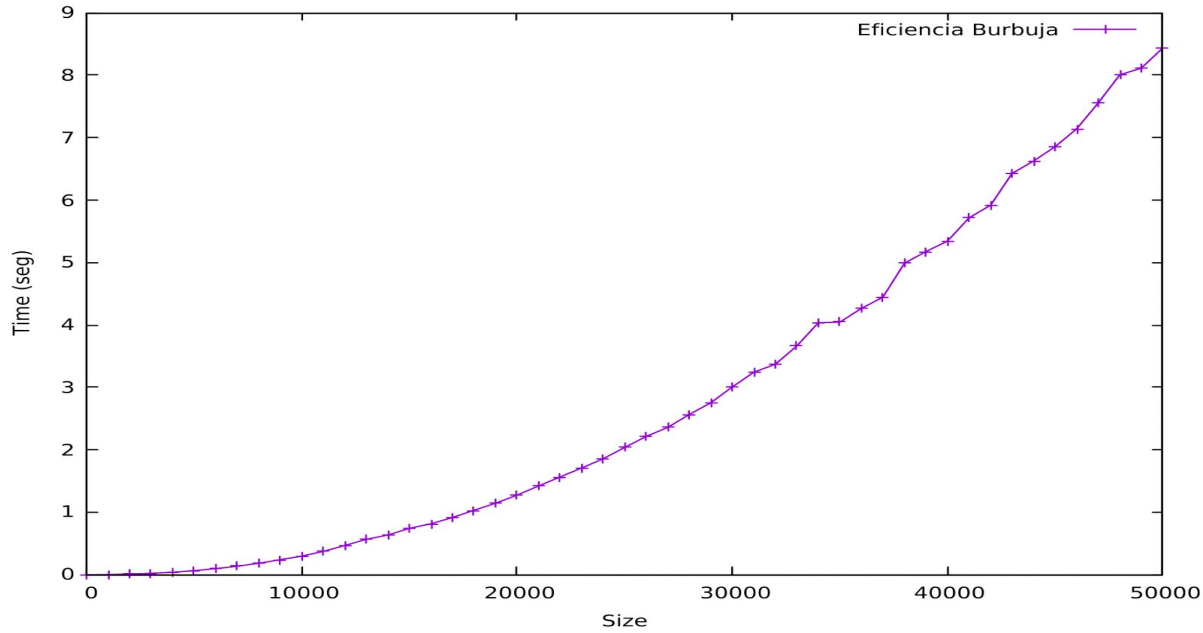
El objetivo de esta práctica es que el alumno llegue a comprender la relevancia de la eficiencia de los algoritmos mediante la ejecución de diversos programas para su posterior medida y análisis de los resultados de los mismos.

2. Algoritmos de eficiencia:

2.1 O(n²)

2.1.1. Burbuja

- Ordena un vector de tamaño n.
- Tiene una eficiencia de O(n²) en el peor de los casos.
- Consiste en ciclar repetidamente a través de la lista, comparando elementos adyacentes de dos en dos. Si un elemento es mayor que el que está en la siguiente posición se intercambian.

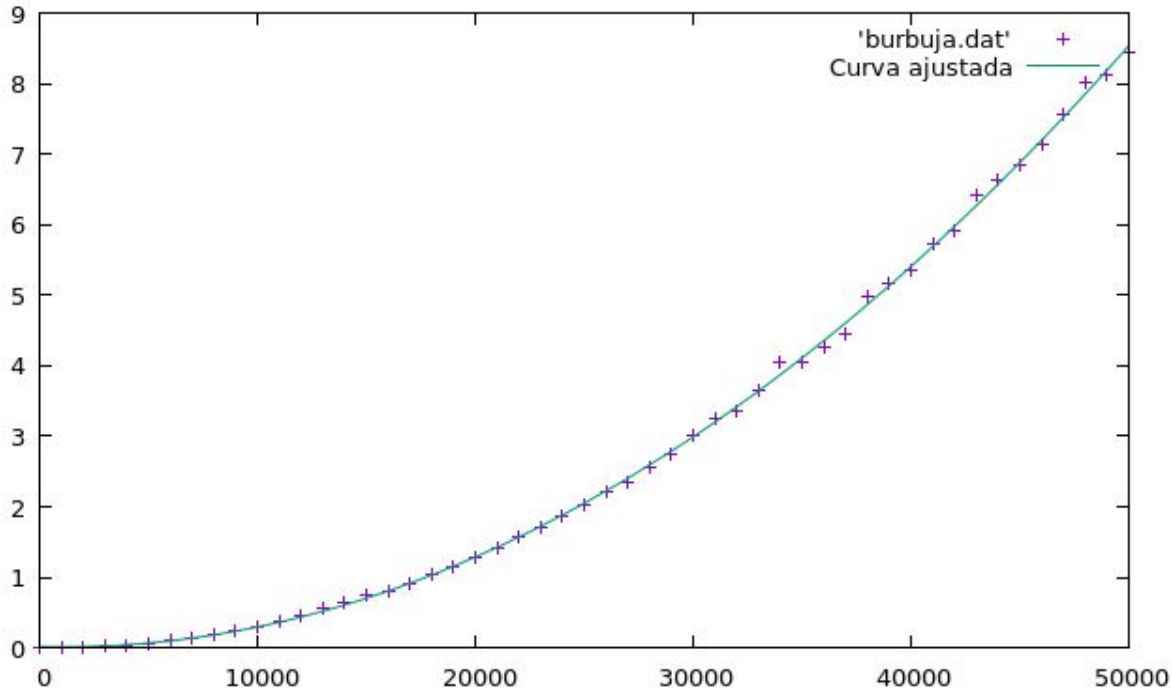


Realizando el ajuste a una función cuadrática obtenemos las variables ocultas

a0 = 1.29827e-09

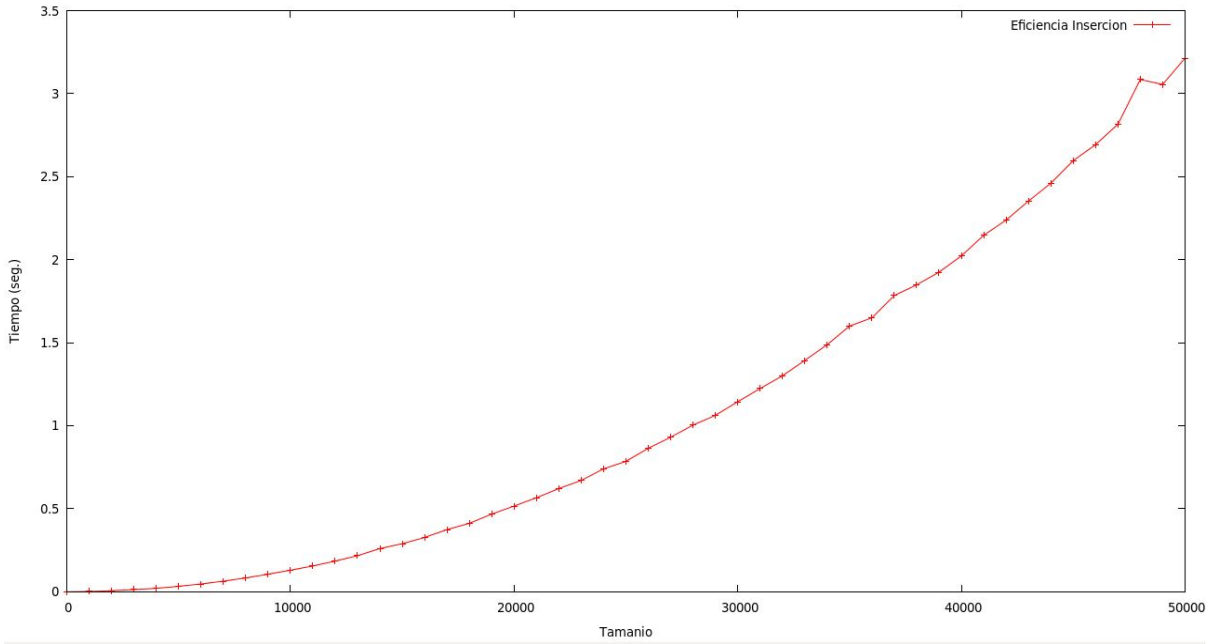
a1 = -8.31901e-07

a2 = 0.00602881


'burbuja.dat'
Curva ajustada
0
1
2
3
4
5
6
7
8
9
0
10000
20000
30000
40000
50000

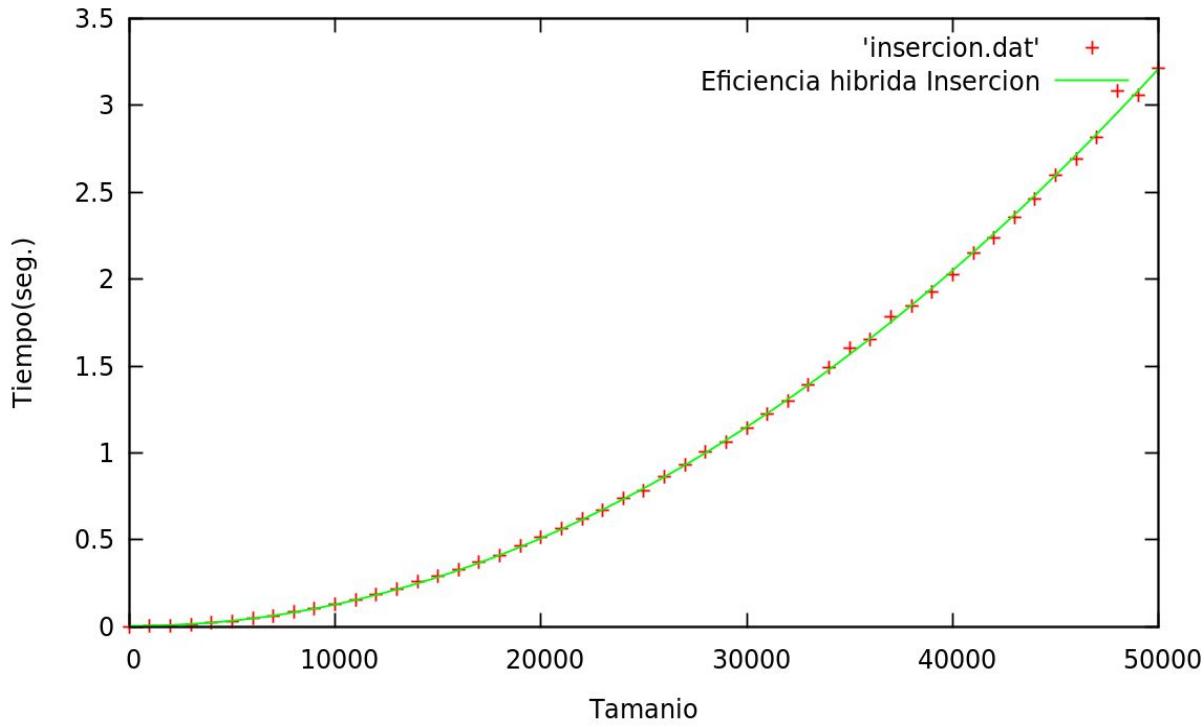
2.1.2. Inserción

- Ordena un vector de tamaño n.
- Tiene una eficiencia de O(n²) en el peor de los casos.
- Consiste en dividir el vector en dos subvectores (ordenado y no ordenado). En un primer lugar, el vector ordenado sólo tendrá un elemento (el primero). Para cada elemento restante del vector se compara con la parte ordenada y se inserta en el lugar adecuado.



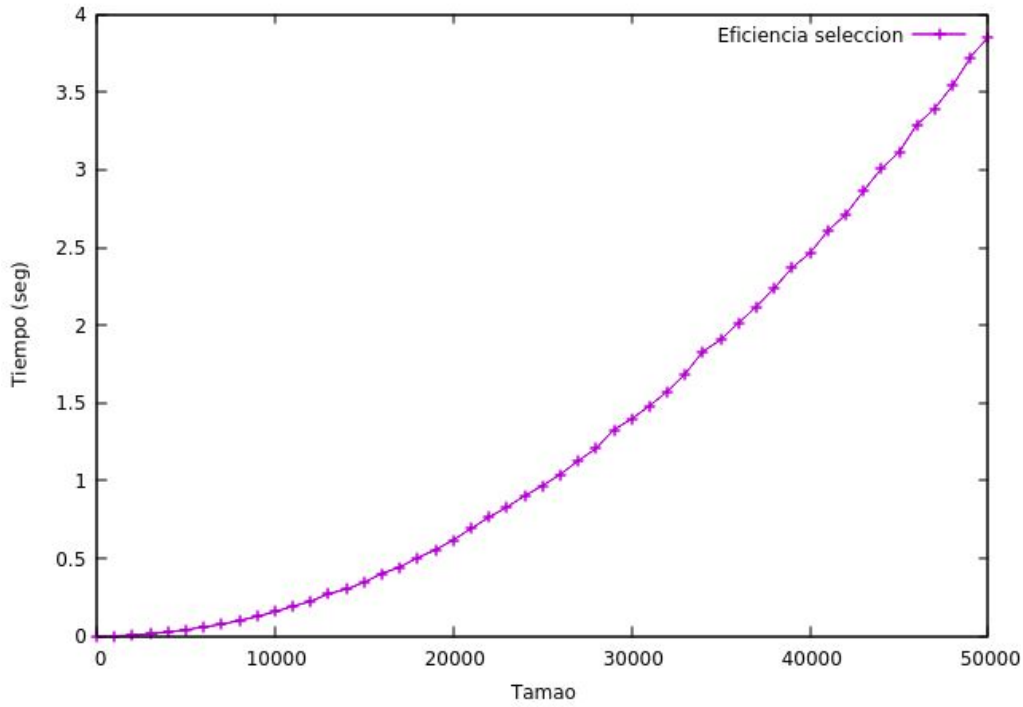
Realizando el ajuste a una función cuadrática obtenemos las variables ocultas

a0 = 3.56201e-09
a1 = -7.73511e-06
a2 = 0.0164765


'insercion.dat'
Eficiencia hibrida Insercion
Tiempo(seg.)
Tamanio
0
0.5
1
1.5
2
2.5
3
3.5
0
10000
20000
30000
40000
50000

2.1.3. Selección

- Ordena un vector de tamaño n
- Tiene una eficiencia de O(n²).
- Busca el mínimo elemento entre la posición i y el final y lo coloca en la posición i, con i = 1, 2, ... , n. Por lo tanto el proceso se realiza en n iteraciones.

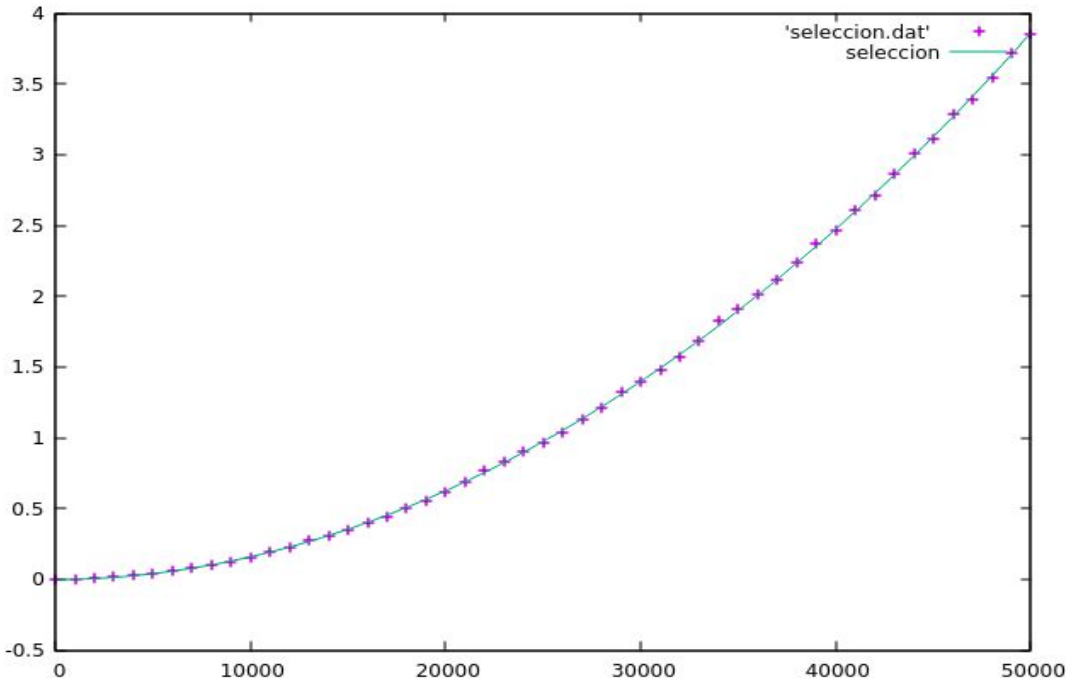


Realizando el ajuste a una función cuadrática obtenemos las variables ocultas

a0 = 1.52682e-09

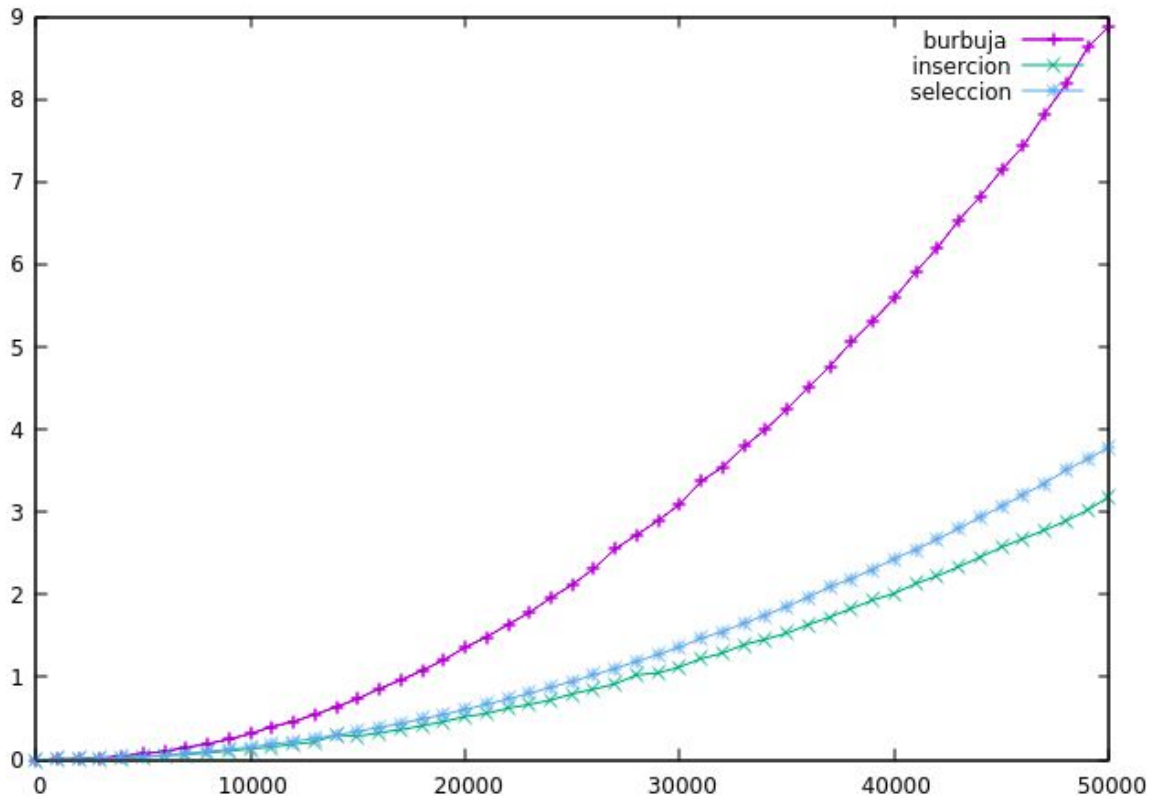
a1 = 8.89438e-07

a2 = -0.00336106



2.1.1. Comparación

A continuación se muestra una gráfica que comprar los tres algoritmos de orden cuadrático. Como podemos comprobar, el más eficiente es inserción, seguido de selección y por último burbuja.



2.2 O(n·log(n))

2.2.1. Mergesort

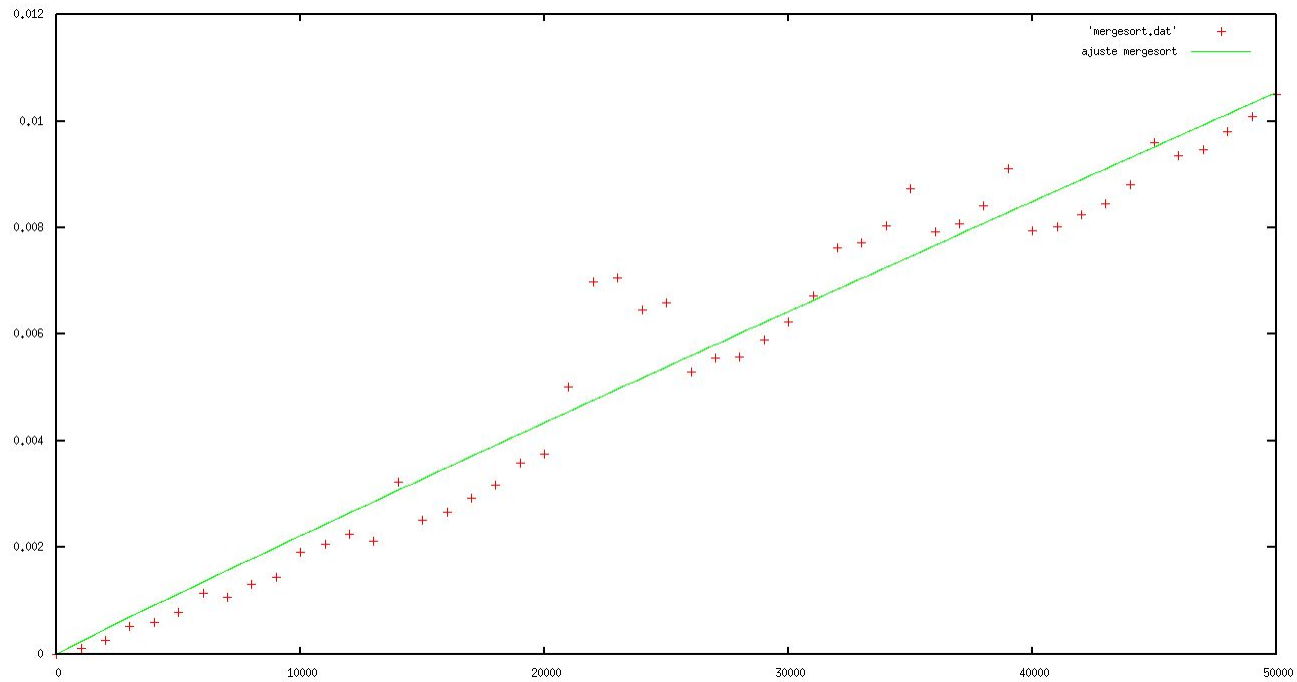
- Ordena un vector de tamaño n.
- Tiene una eficiencia de O(n·log(n))
- Si el vector es vacío o de tamaño uno está ordenado. En caso contrario divide en vectores de la mitad del tamaño que el original, se dividen las sublistas recursivamente (usando este procedimiento), y posteriormente se mezclan estas dos sublistas en una sola lista ordenada.

Realizando el ajuste a una función n·log(n) obtenemos las variables ocultas

a0 = -0.253707

a1 = 2.74939e-08

a2 = 0.253707

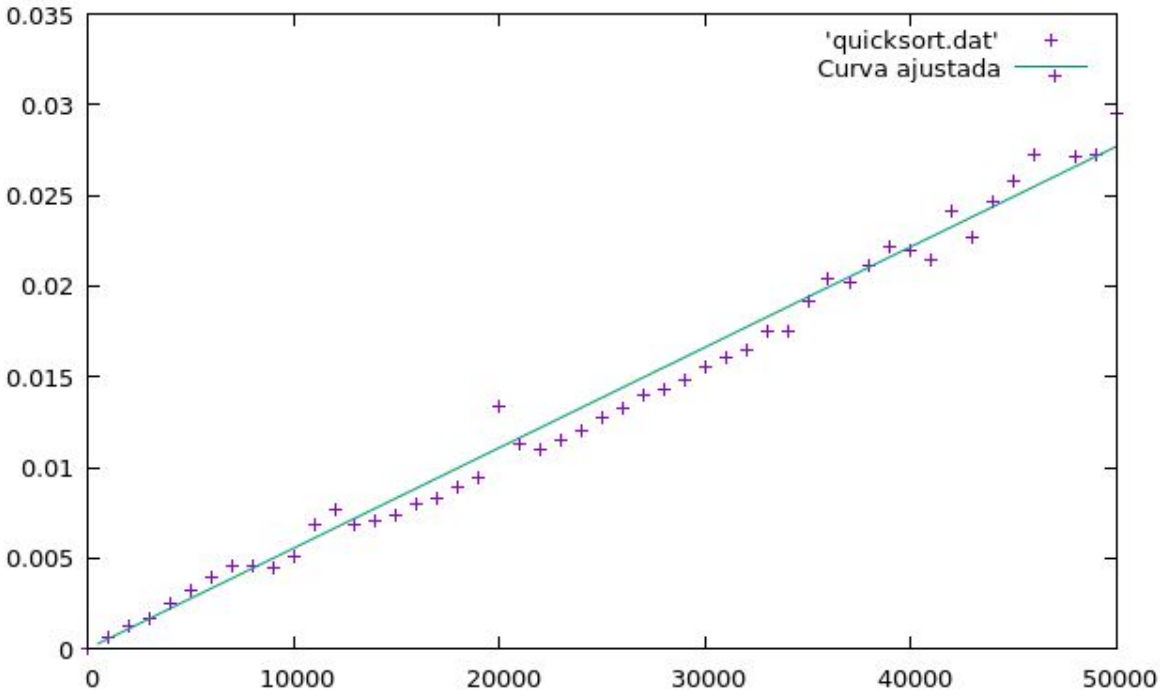


2.2.2. Quicksort

- Ordena un vector de tamaño n.
- Tiene una eficiencia de O(n·log(n)), aunque en el peor de los casos puede tener una eficiencia de O(n²), ya que el número de comparaciones no depende del orden de los términos, si no del número de términos
- Se elige un elemento v de la lista L de elementos al que se le llama pivote. Se particiona la lista L en tres listas:
    - L1 - que contiene todos los elementos de L menos v que sean menores o iguales que v
    - L2 - que contiene a v
    - L3 - que contiene todos los elementos de L menos v que sean mayores o iguales que v
- Se aplica la recursión sobre L1 y L3 y se concatenan todas al final

Realizando el ajuste a una función n·log(n) obtenemos las variables ocultas

a0 = 3.56201e-09
a1 = -7.73506e-06
a2 = 5.50681e-07



2.2.3. Heapsort

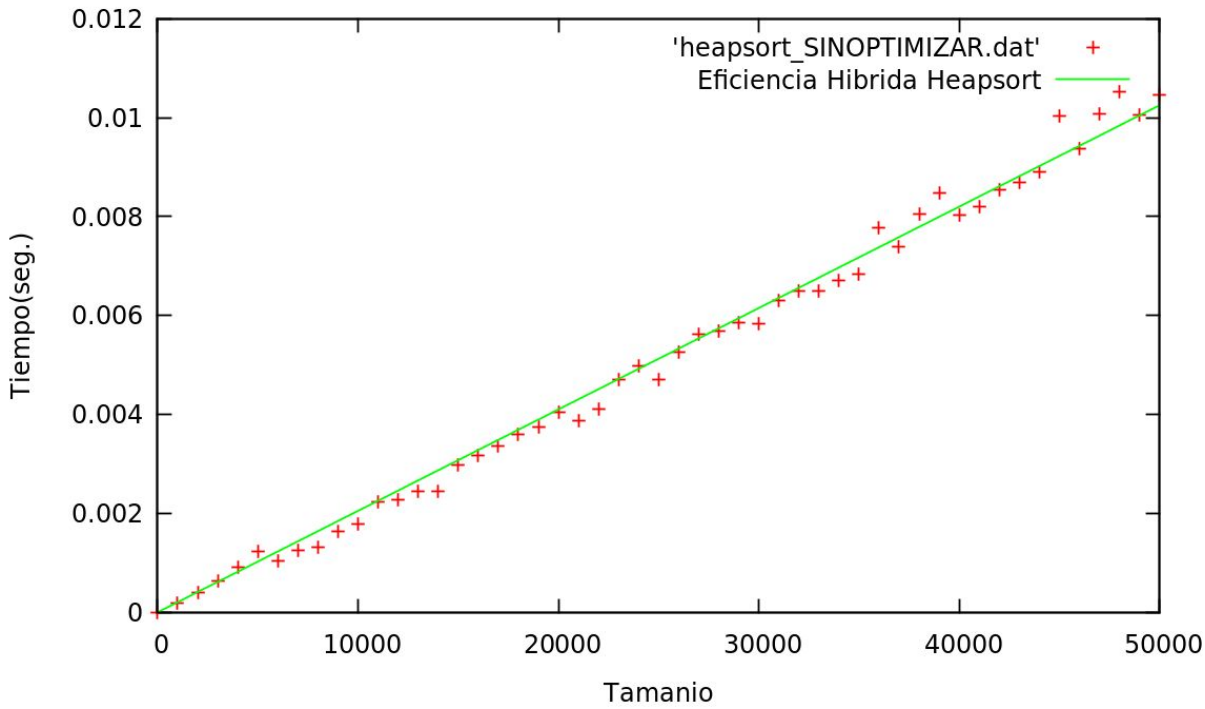
- Ordena un vector de tamaño n.
- Tiene una eficiencia de O(n·log(n)).
- Consiste en almacenar todos los elementos del vector a ordenar en un montículo (*heap*), y luego extraer el nodo que queda como nodo raíz del montículo (cima) en sucesivas iteraciones obteniendo el conjunto ordenado. Basa su funcionamiento en una propiedad de los montículos, por la cual, la cima contiene siempre el menor elemento (o el mayor, según se haya definido el montículo) de todos los almacenados.

Realizando el ajuste a una función n·log(n) obtenemos las variables ocultas

a0 = 1.95648e-05

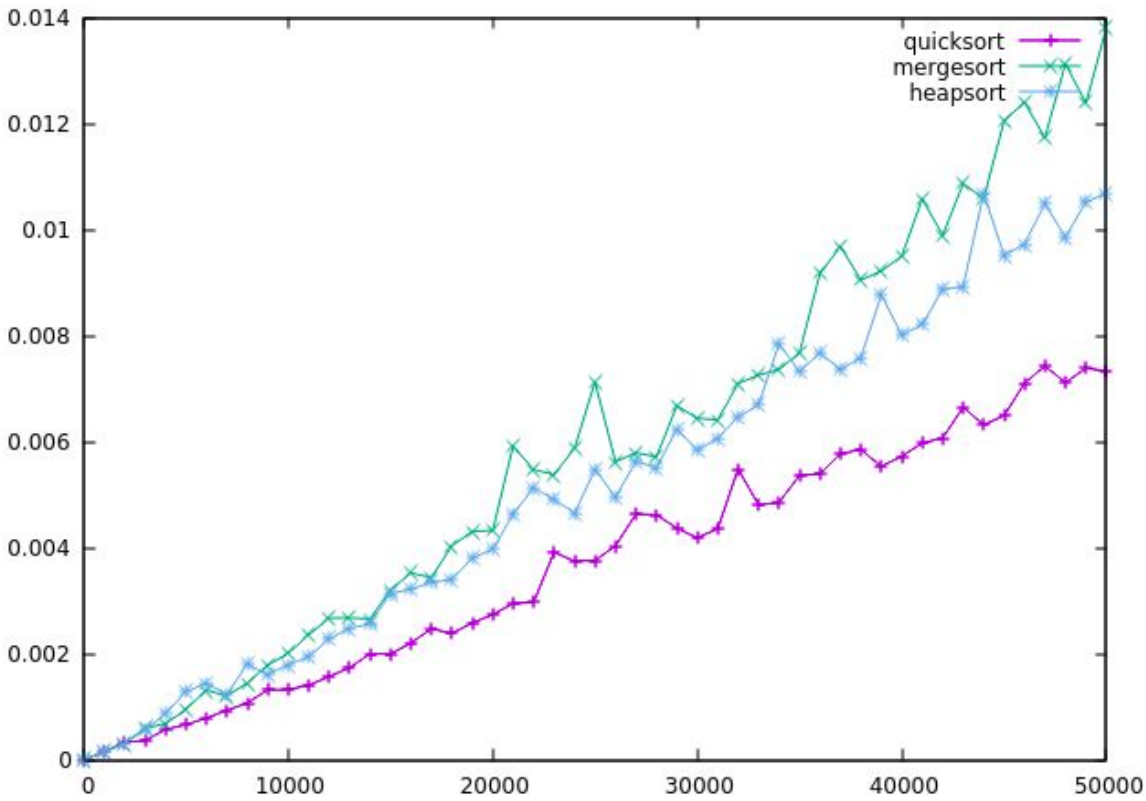
a1 = 0.0661408

a2 = 1.95648e-05



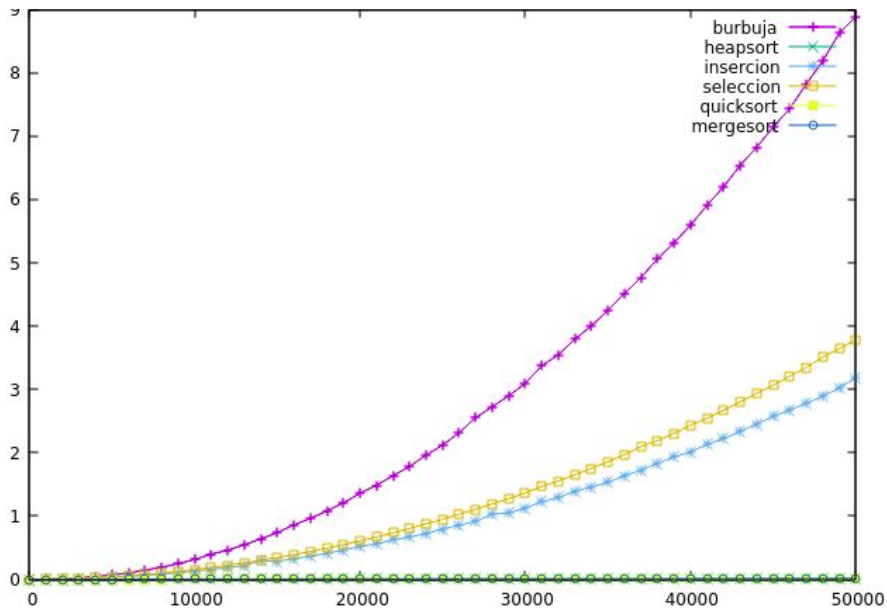
2.2.4. Comparación

La gráfica siguiente compara los tres algoritmos de ordenación con eficiencia O(nlogn). Podemos comprobar que el más eficiente es el quicksort, seguido de heapsort y por último heapsort.



2.3 Comparación

En la gráfica siguiente se comparan los seis algoritmos de ordenación. Queda claro que los algoritmos de eficiencia O(nlogn) son mucho más eficientes que los de eficiencia cuadrática.



2.3 O(n³)

2.3.1. Floyd

- Encuentra caminos mínimos entre grafos dirigidos ponderados.
- Tiene una eficiencia de O(n³).
- El algoritmo compara todos los posibles caminos entre cada par de nodos del grafo y selecciona el mínimo. En cada iteración almacena los resultados en una matriz. Por tanto, en una sola ejecución se obtienen todos los caminos mínimos entre cada par de nodos del grafo.

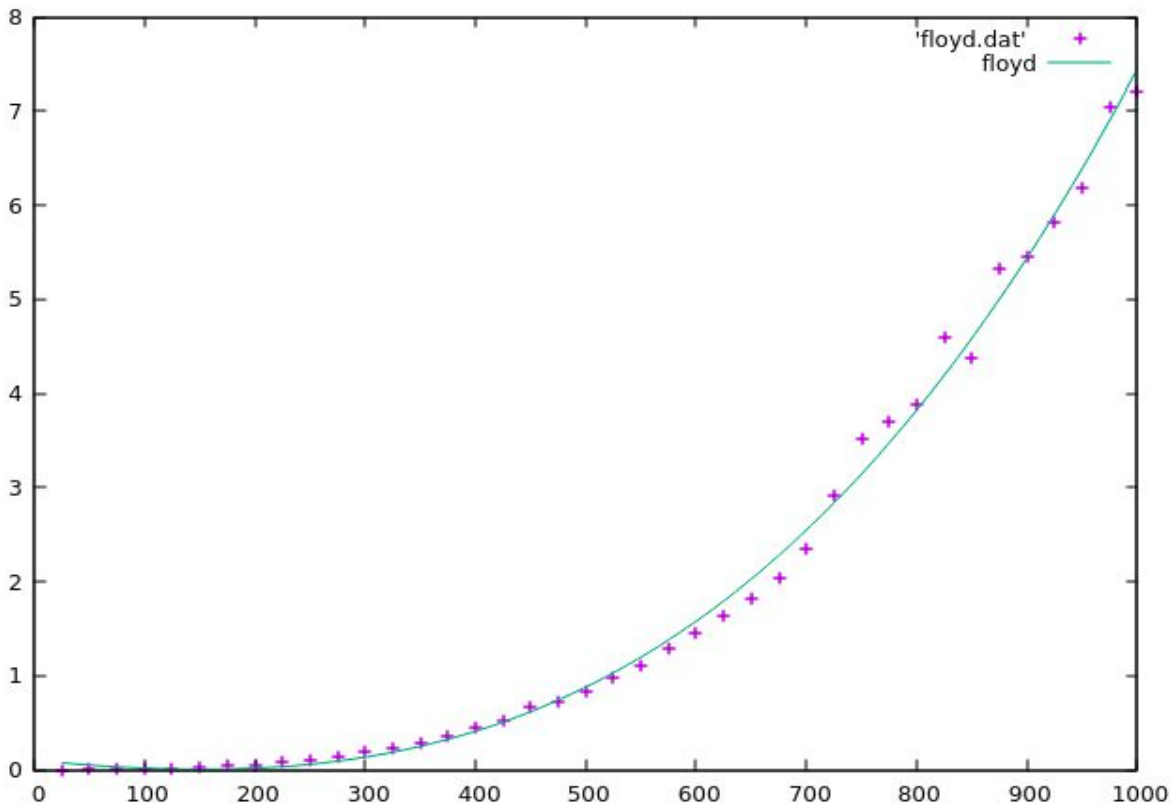
Realizando el ajuste a una función cúbica obtenemos las variables ocultas

a0 = 6.24086e-09

a1 = 2.20775e-06

a2 = -0.00110777

a3 = 0.0990287



2.4 $O(2^n)$

2.4.1. Hanoi

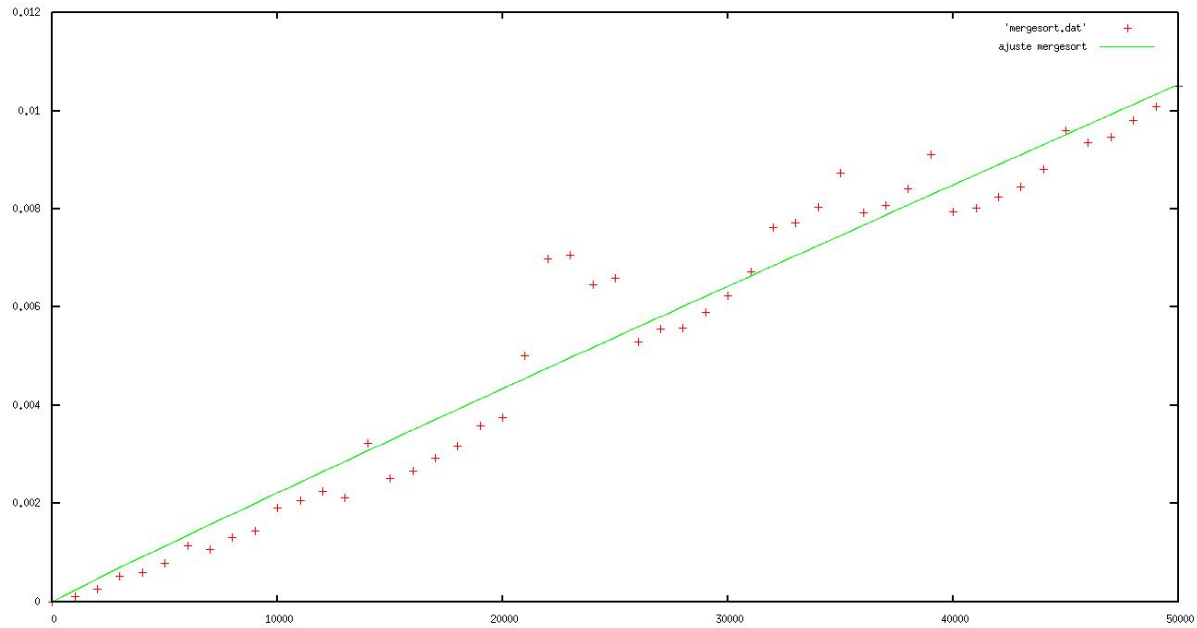
- Resuelve el problema matemático de las torres de Hanoi
- Tiene una eficiencia de $O(2^n)$
- Consiste en la resolución recursiva de este algoritmo. Si el número de discos es 1, simplemente mueve disco a otra varilla y termina el problema. En caso contrario se llama recursivamente para desplazar los discos a su varilla correspondiente.

Realizando el ajuste a una función exponencial obtenemos las variables ocultas

a0 = -0.253707

a1 = 2.74939e-08

a2 = 0.253707



3.Optimización

Uno de los factores que más influye en la eficiencia de un algoritmo y en sus tiempos resultantes es el de la optimización que se haya usado.

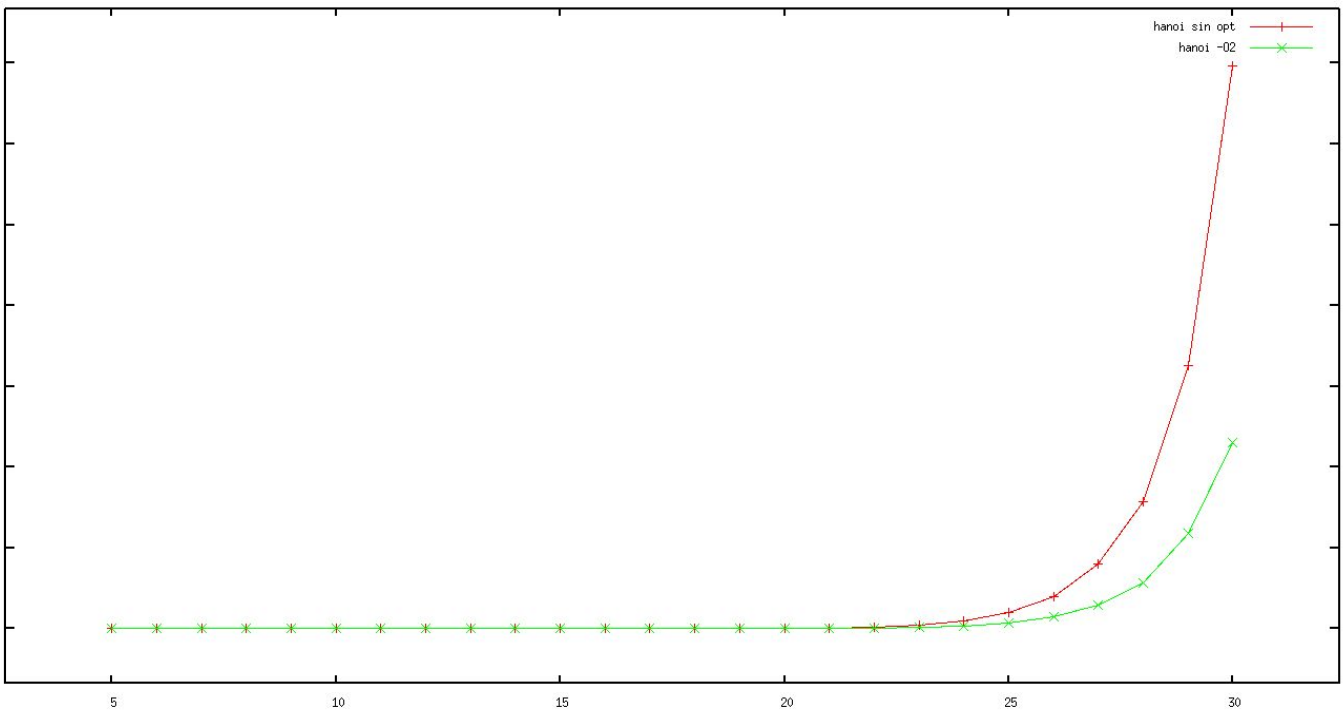
A continuación detallamos los resultados y contrastamos los algoritmos ejecutados con:

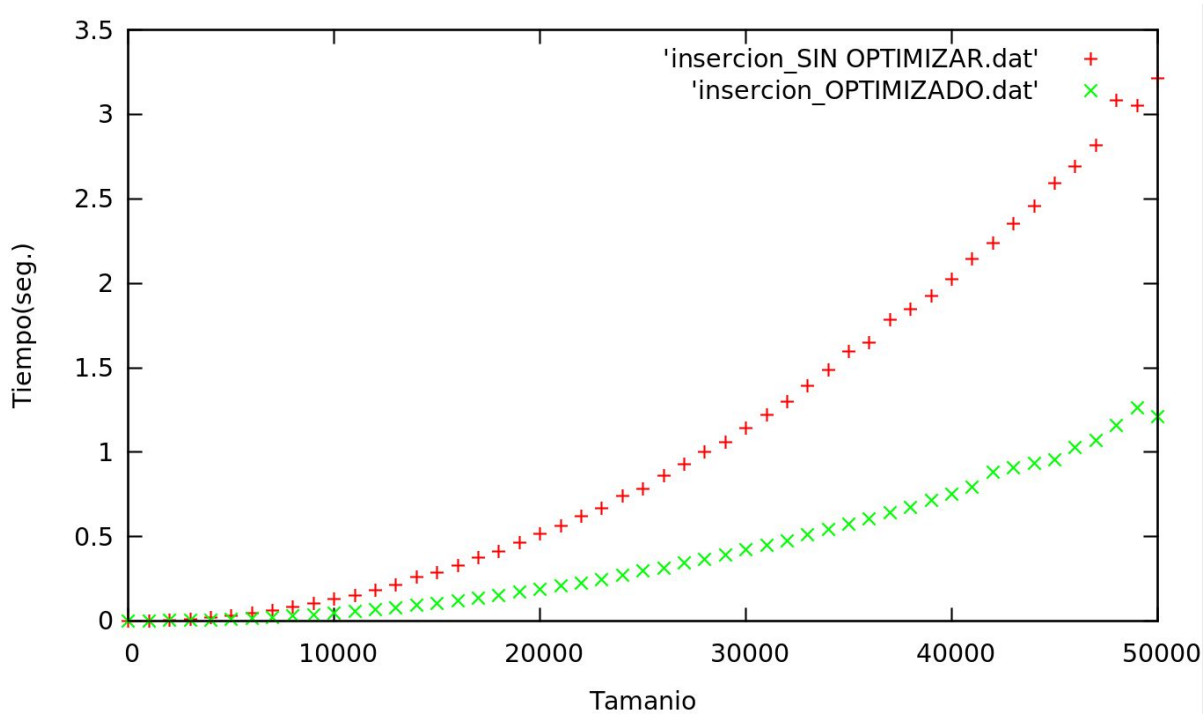
g++ source.cpp -o executable (sin optimizar)

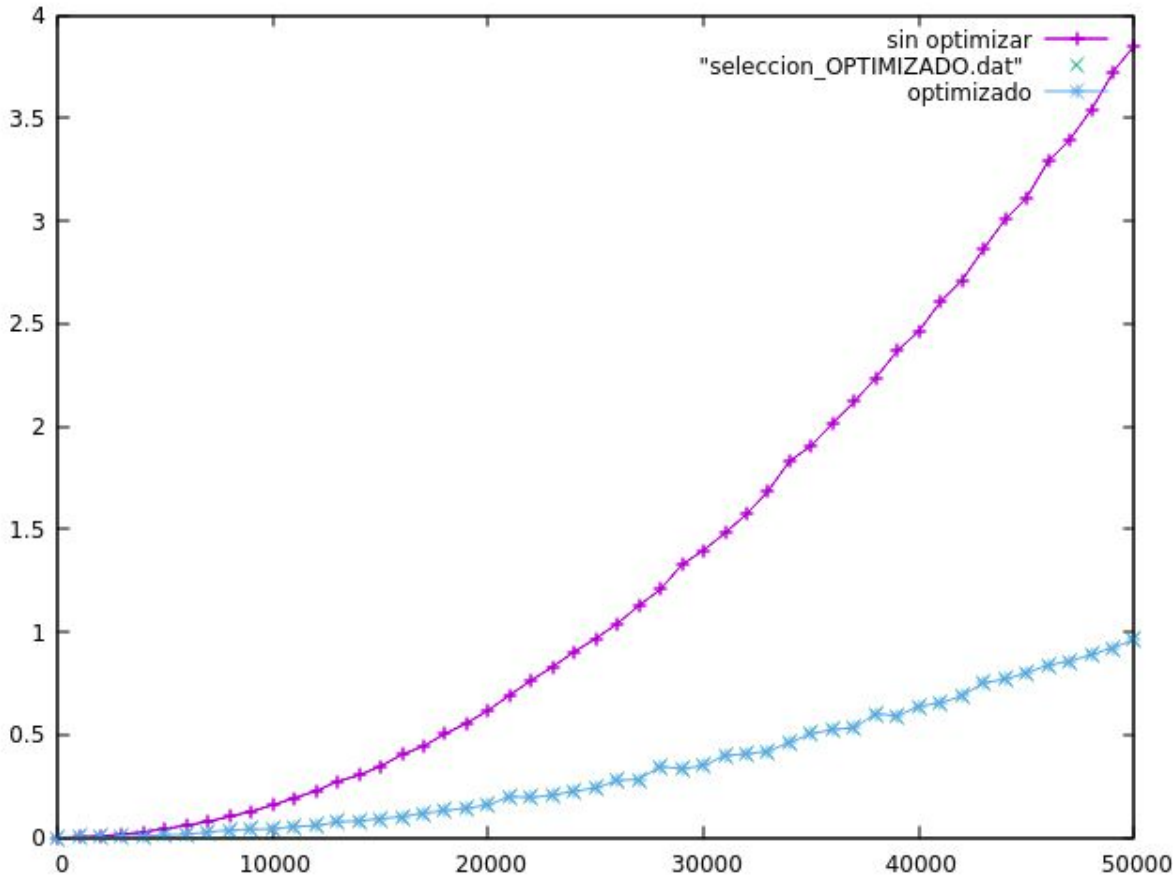
Frente a los mismos pero añadida la opción de compilación *-O2*:

g++ -O2 source.cpp -o executable

Al optimizar en compilación, hemos comprobado experimentalmente que el tiempo de ejecución necesario se reduce considerablemente en todos los algoritmos. Estos son algunas gráficas comparando la ejecución optimizada y sin optimizar de diferentes algoritmos. El resto de comparaciones pueden consultarse en las gráficas adjuntas en la entrega o en la presentación.




'insercion_SIN OPTIMIZAR.dat'
'insercion_OPTIMIZADO.dat'
Tiempo(seg.)
Tamanio
0
0.5
1
1.5
2
2.5
3
3.5
0
10000
20000
30000
40000
50000


sin optimizar
"seleccion_OPTIMIZADO.dat"
optimizado
0
0.5
1
1.5
2
2.5
3
3.5
4
0
10000
20000
30000
40000
50000

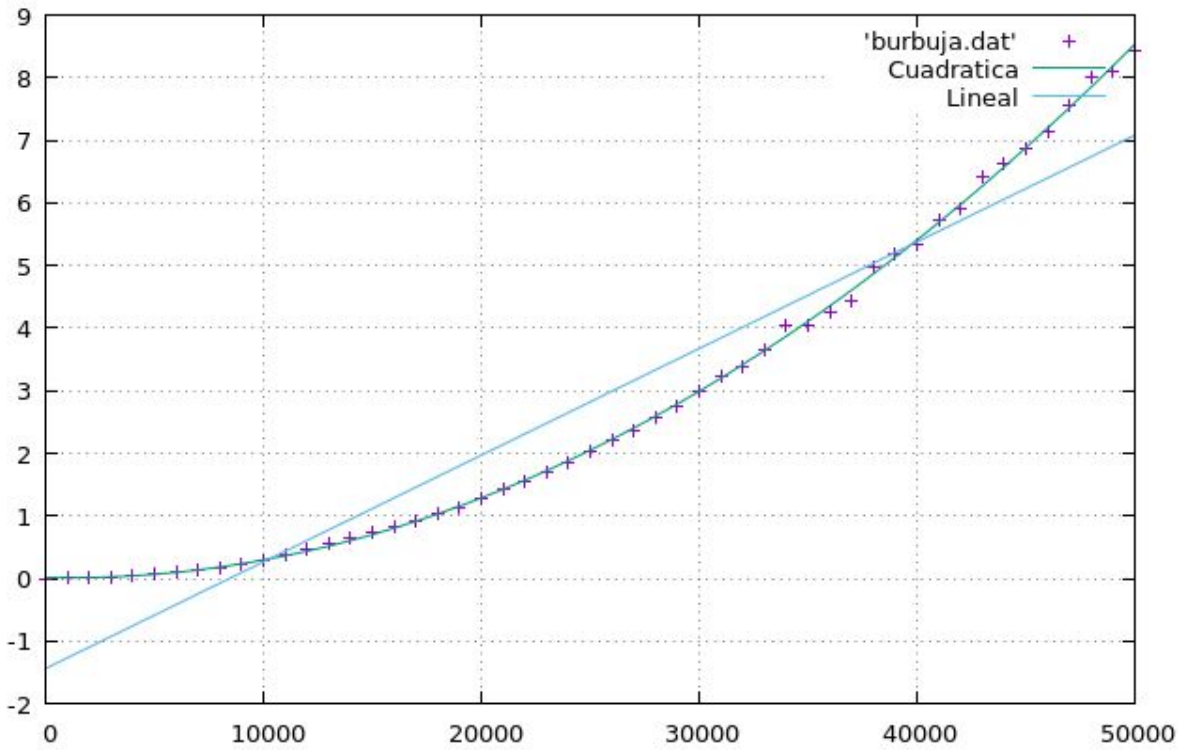
4.Correlación de los diversos ajustes

Para tener una idea del futuro comportamiento de los algoritmos para un tamaño determinado del problema es necesario hacer uso de algún tipo de estimación que nos permita realizar dicha estimación. Un método para realizar esto es estudiar la eficiencia híbrida. Buscar una función que se corresponda con el comportamiento de los datos tras haber realizado previamente un ajuste por mínimos cuadrados.

Podemos comprobar la bondad de este ajuste frente a la elección de otras funciones a ajustar comparando cómo se ajustan estas otras. Aquí se encuentran recogidas estas comparativas y sus resultados.

Burbuja:

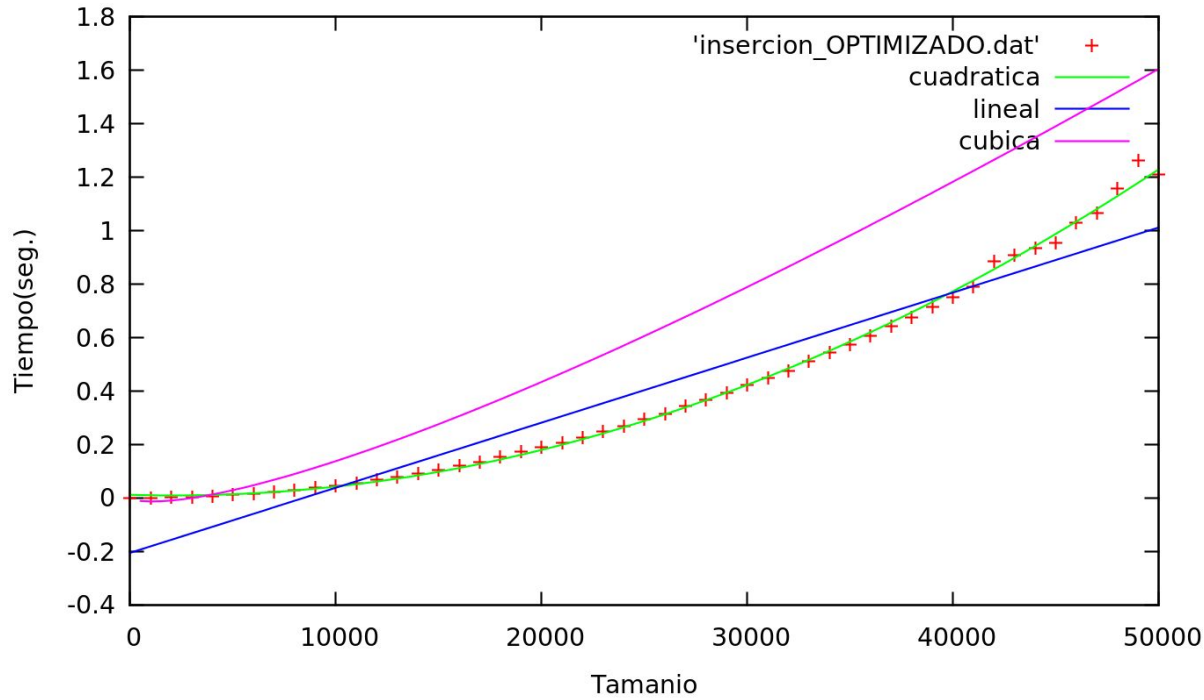
Su mejor ajuste para la eficiencia híbrida, como comprobamos, era la cuadrática.
Pero… ¿Cómo se ajustan las otras funciones?



Como podemos observar, otros ajustes como el lineal la bondad del ajuste es pésima para estimar el tiempo de cómputo.

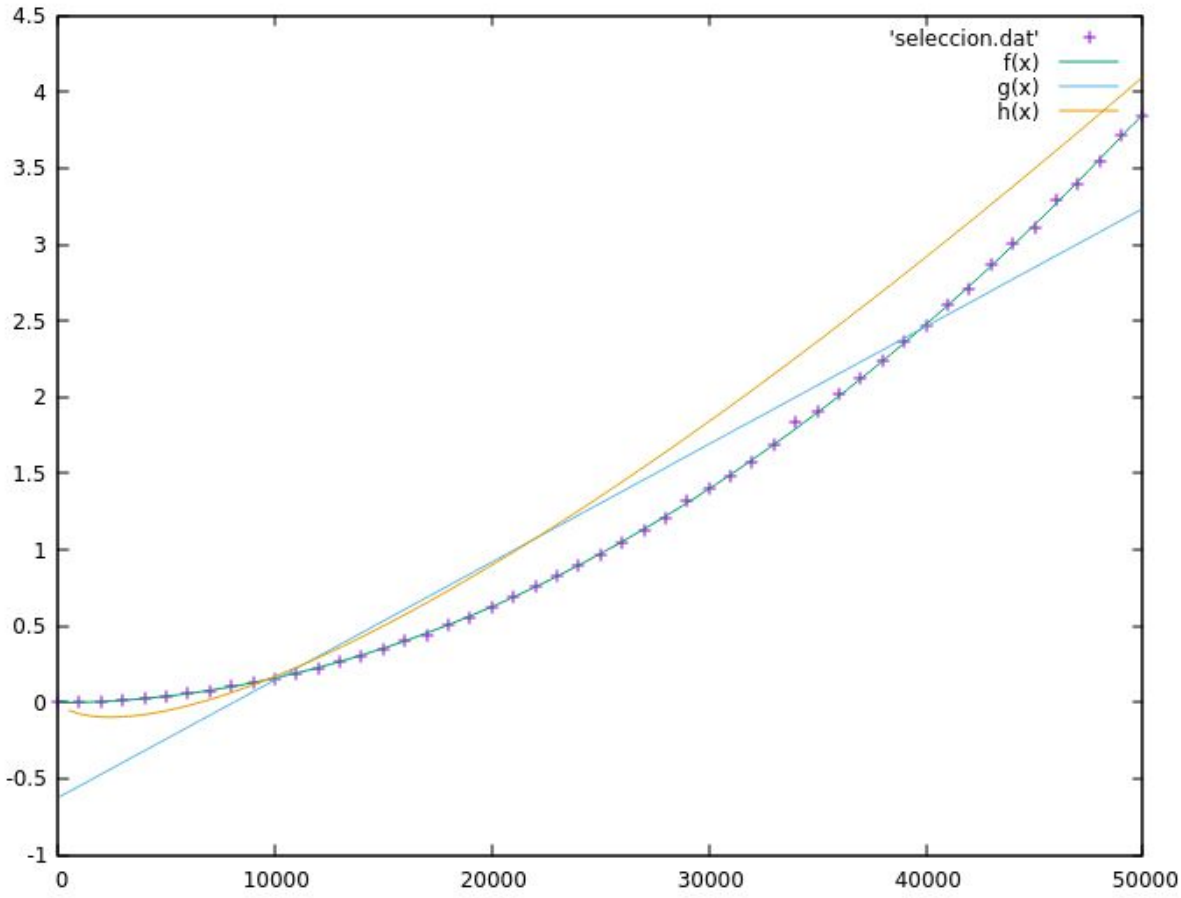
Inserción:

En lo sucesivo probaremos más ajustes y veremos cómo difieren los otros ajustes del utilizado en la eficiencia híbrida.



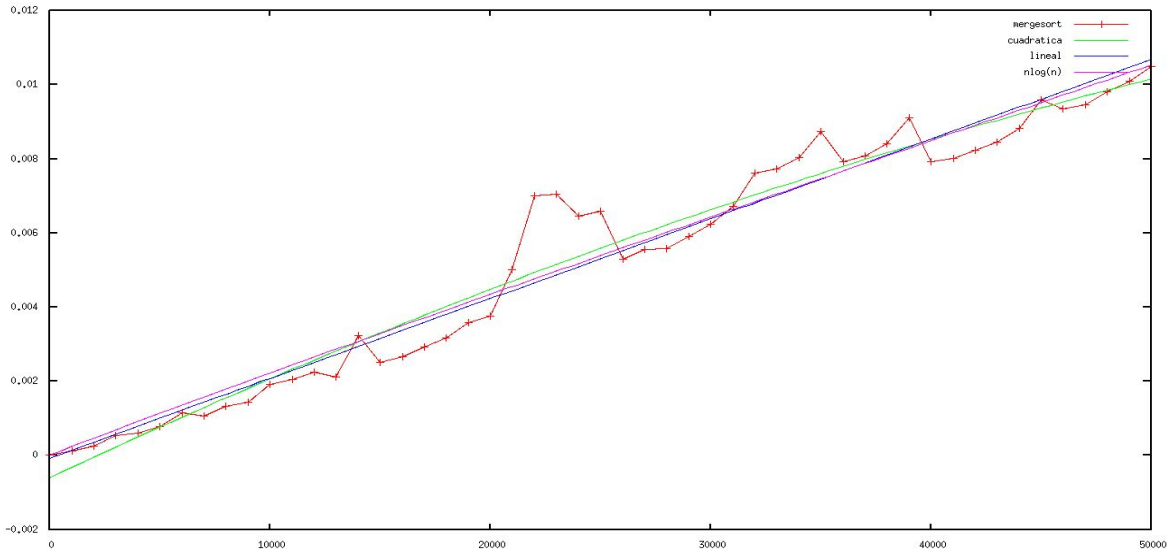
Selección:

Para el algoritmo de selección obtenemos lo siguiente:

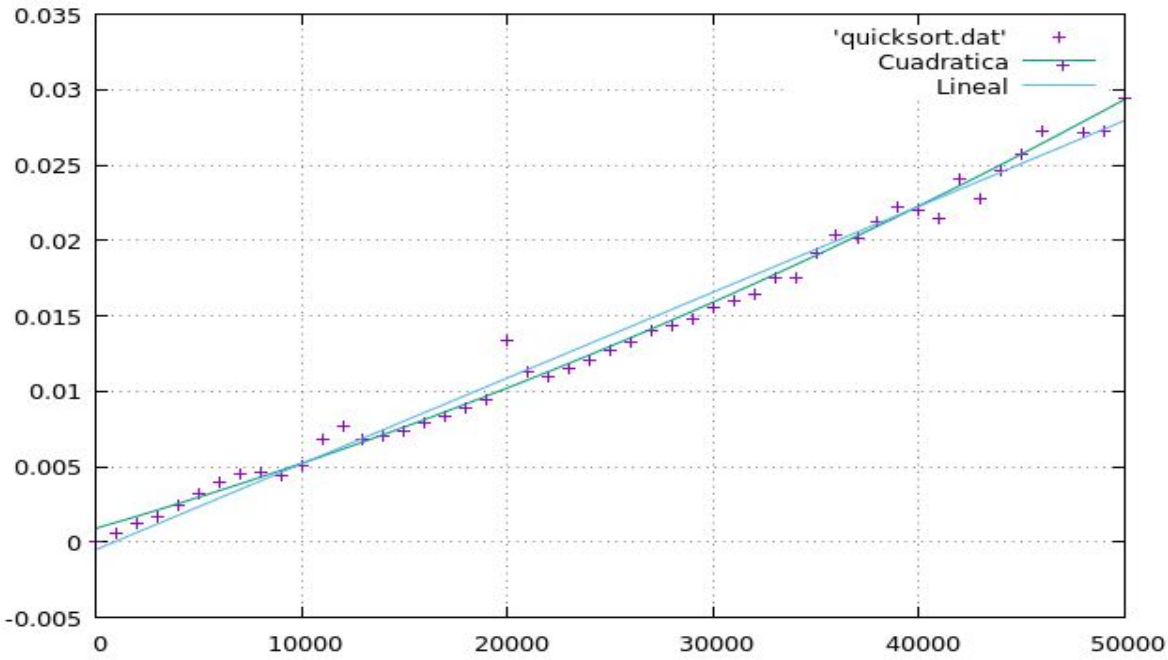


Como podemos apreciar, todos los algoritmos del orden O(n²) tienen un comportamiento similar al ser analizados con otros ajustes no cuadráticos.

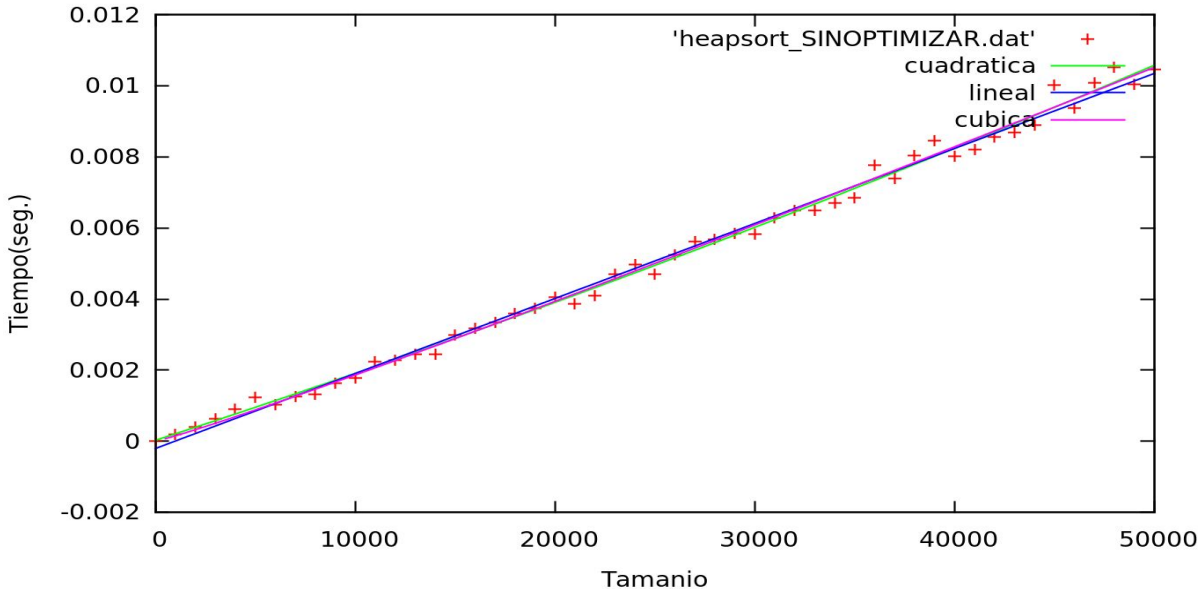
Mergesort:



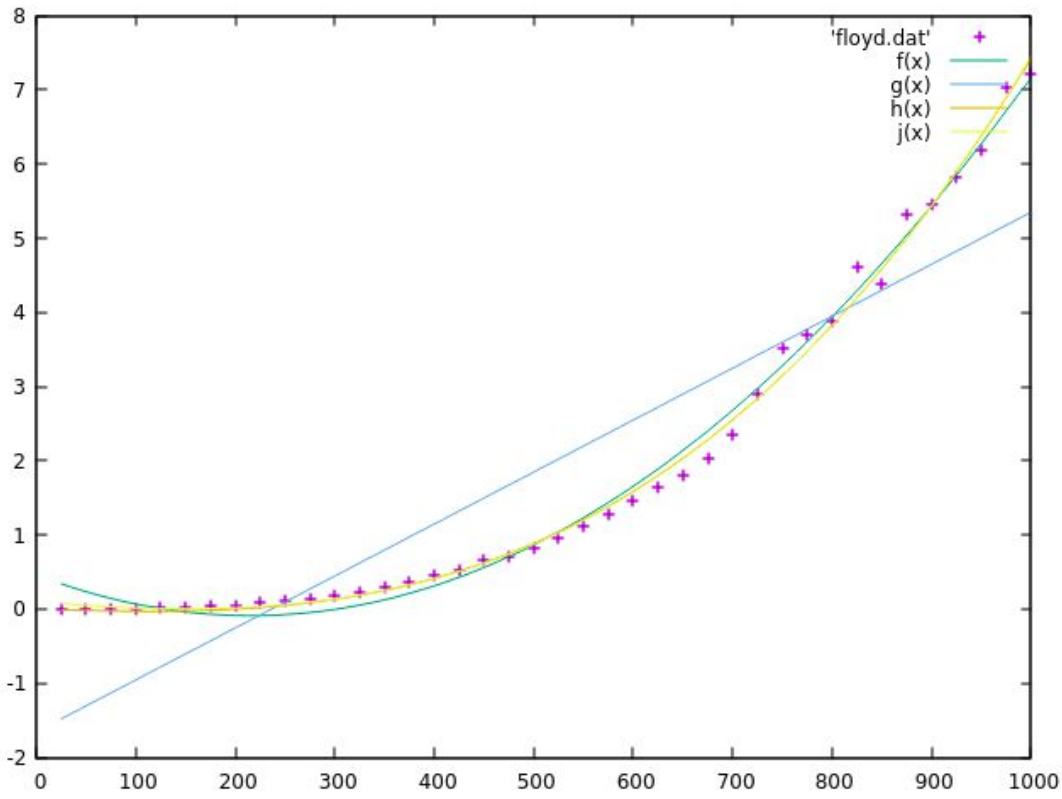
Quicksort:



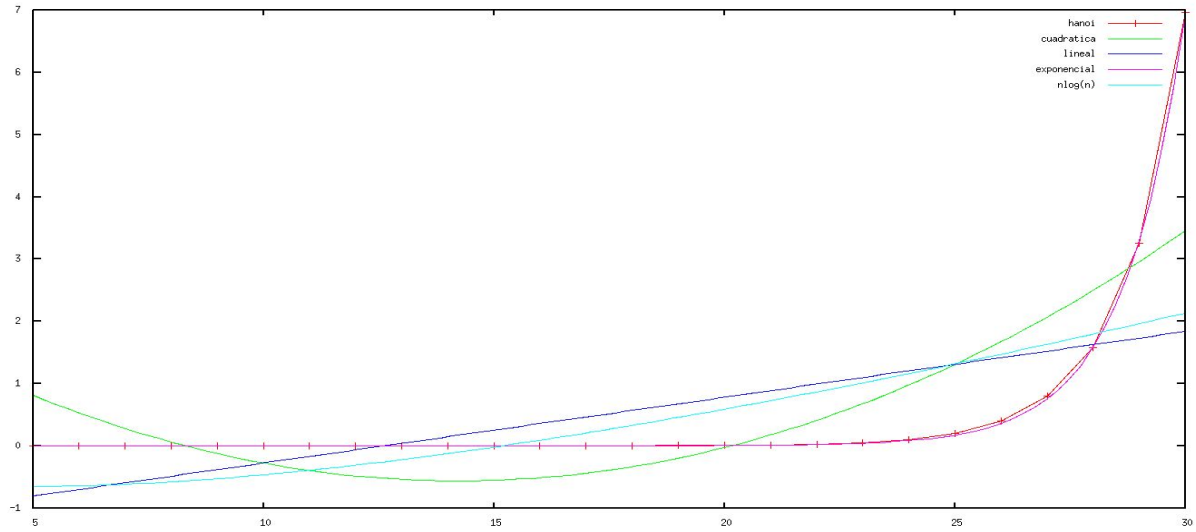
Heapsort:



Floyd:



Hanoi:



5.Anexo

5.1 Tablas de contenidos

Adjuntamos aquí las tablas con las medidas de tiempos obtenidas:

Para algoritmos de ordenación:

TAMAÑO	BURBUJA	INSERCIÓN	SELECCIÓN	MERGESORT	QUICKSORT	HEAPSORT
1000	2.214	1.778	164.537	164.175	1.428	1.778
2000	8.955	6.253	626.577	3.515	3.542	6.253

3000	20.496	13.151	1.469	5.048	4.371	13.151
4000	38.176	20.717	26.347	7.929	5.763	20.717
5000	63.669	32.399	41.315	11.119	7.292	32.399
6000	95.338	46.728	58.564	11.291	8.937	46.728
7000	134.255	63.608	76.383	13.667	10.569	63.608
8000	181.965	82.974	101.169	16.779	12.138	82.974
9000	239.492	105.393	126.078	19.816	13.792	105.393
10000	296.445	129.617	157.276	23.346	16.478	129.617
11000	37.284	154.756	192.277	21.314	16.817	154.756
12000	466.712	184.139	22.436	23.561	18.674	184.139
13000	56.722	217.044	272.343	25.558	2.131	217.044
14000	634.589	259.526	302.247	28.153	24.204	259.526
15000	740.756	289.151	346.635	30.876	25.408	289.151
16000	81.417	326.618	402.541	3.357	26.318	326.618
17000	914.631	373.849	443.763	36.454	29.023	373.849
18000	102.941	411.767	504.259	4.037	31.048	411.7 67
19000	114.328	467.207	555.423	44.068	32.921	467 .207
20000	127.296	51.575	618.515	46.656	34.278	5 1.575
21000	141.853	565.619	691.752	40.295	34.846	565.619
22000	156.425	620.984	764.913	43.096	36.667	620.984
23000	170.751	670.039	830.085	45.229	3.925	670.039

24000	185.605	739.461	903.014	47.684	39.677	739.461
25000	203.655	784.994	967.005	50.926	42.548	784.9 94
26000	221.413	863.755	1.0406	53.947	43.583	863 .755
27000	2.3564	930.259	112.763	55.618	4.599	930.259
28000	256.513	100.311	120.698	5.928	47.861	100.311
29000	275.286	106.213	132.689	61.019	49.079	106.213
30000	300.578	114.407	139.705	64.688	50.484	114.407
31000	324.119	122.357	148.068	67.217	52.612	122.357
32000	337.007	1.3005	157.418	70.906	53.411	1.3005
33000	3.6603	139.292	168.567	7.335	54.981	139.292
34000	403.552	148.754	183.146	77.651	56.942	148.754
35000	404.552	159.976	190.573	104.912	58.158	159.976
36000	426.455	164.932	201.562	87.897	61.641	164.932
37000	444.631	178.378	212.027	86.633	61.663	178.378
38000	498.867	184.791	223.652	89.012	6.467	184.791
39000	517.591	192.475	236.924	92.804	66.154	192.475
40000	534.389	202.232	246.294	96.255	6.897	202.232
41000	571.467	214.684	260.588	82.431	69.857	214.684
42000	591.824	223.814	271.113	8.533	72.537	223.8 14
43000	642.654	235.266	286.286	88.287	73.106	235.2 66
44000	6.6248	2.4601	300.644	90.708	7.419	2.4 601

45000	685.601	259.602	311.329	92.375	76.507	259.602
46000	714.013	269.239	328.998	95.856	78.765	269.239
47000	755.452	281.596	339.398	98.175	79.531	281.596
48000	800.657	308.483	354.084	101.088	80.645	308.483
49000	811.365	3.0554	371.885	103.271	83.564	3.0554
50000	843.534	321.349	384.767	105.535	84.764	321.349

Algoritmo de grafos:

FLOYD TAMAÑO	T.EJEC
25	186
50	1.691
75	5.549
100	9.614
125	16.725
150	25.805
175	39.787
200	55.406
225	81.352
250	107.115
275	144.312
300	184.749
325	231.902
350	287.274
375	358.944
400	444.626
425	517.884
450	661.725
475	717.625
500	827.658
525	969.267
550	111.342
575	128.322
600	145.198
625	163.393
650	181.041
675	203.766
700	235.723
725	290.577
750	351.404
775	369.138
800	387.428
825	460.406
850	437.671

875	532.042
900	545.157
925	581.666
950	618.858
975	704.082
1000	720.832

Algoritmo de Hanoi:

HANOI	
5	2,00E-06
6	3,00E-06
7	5,00E-06
8	5,00E-06
9	8,00E-06
10	1,00E-05
11	1.9e-05
12	3.4e-05
13	6.6e-05
14	132
15	273
16	494
17	1.038
18	1.654
19	3.422
20	6.869
21	13.102
22	25.947
23	50.695
24	101.825
25	202.328
26	404.072
27	821.772
28	159.081
29	320.046
30	641.383

NOTA: Todos estos tiempos se pueden consultar detenidamente en los archivos de datos que se adjuntan.

6.Conclusión

A la vista de los resultados obtenidos nos damos cuenta de que hay múltiples factores que influyen en el tiempo de ejecución de un algoritmo.

Fundamentalmente su orden de eficiencia será el más determinante, pero existen otros factores que pueden alterar notablemente su tiempo de ejecución y hacernos sacar conclusiones precipitadas. Estos son tales como las propiedades del hardware en el que se ejecute el programa (pues un procesador de última generación será más eficiente que uno más antiguo) pero también el factor de optimización -O, el cual puede hacer que un procesador más antiguo pueda ejecutar un programa en un tiempo mucho mejor que el mejor de los procesadores sin usar ningún tipo de optimización.