Algorítmica

Práctica 1
Eficiencia

Fc.Javier Azpeitia Muñoz
Christian Andrades Molina
Guillermo Muriel Sánchez Iafuente
Miguel Keane Cañizares
Mercedes Alba Moyana

Índice:

1 Introducción:

El uso de algoritmos en la informática, es esencial, es decir, para cualquier tarea relacionada con la informática, ya sea desde a nivel teórica, o a nivel práctico, es necesario tener un conocimiento básico sobre como analizar algoritmos, saber como mejorar un algoritmo, como comprarlo con otro para que los resultados sean realmente representativos o como mínimas modificaciones en un algoritmo pueden influir fuertemente en los resultados de nuestros softwares.

En esta práctica de la asignatura Algorítmica, vamos a analizar empíricamente varios algoritmos propuestos, de distintos órdenes de eficiencia, para luego ajustarlos a sus correspondientes órdenes de eficiencia teórica, para ver como coinciden. También haremos varias comparativas en algoritmos, y algunas modificaciones en el tratamiento de los algoritmos para ver cómo pueden variar con pequeños detalles, como variaciones en compilación, o en la ejecución en distintas computadoras.

2 Análisis de Algoritmos O(n^2):

Para analizar algoritmos con orden de eficiencia O(n^2), vamos a usar 3 algoritmos clásicos de ordenación, burbuja, inserción y selección.

Para cada uno de ellos hemos realizado un programa que hace uso de cada uno de los algoritmos, y mediante el uso de macros CSH, vamos a lanzar varias veces cada programa con distintos tamaños de entrada, y obtendremos los tiempos de ejecución para cada uno de los tamaños propuestos.

Para tomar los tiempos correspondientes a cada tamaño de entrada, hemos hecho uso de la librería Chrono de STL. La principal motivación para usar chrono es la precisión de esta librería para proporcionar tiempos.

Macro CSH

```
1  #!/bin/csh -vx
2  echo "" >> salida.dat
3  @ i = 600
4  while ( $i < 18600 )
5  ./quicksort $i >> salida.dat
6  @ i += 600
7  end
```

salida.dat *//Fichero de salida con los tamaños y los tiempos*
@ i *//variable que marca el número de veces que se ejecuta el programa y el tamaño de entrada*
./quicksort *//nombre del programa a ejecutar, en este ejemplo es quicksort, pero podría ser cualquiera.*

Inclusión de Chrono

```
#include <chrono>


using namespace std::chrono;

high_resolution_clock::time_point tantes, tdespues;
duration<double> transcurrido;
```

Uso de Chrono

```
tantes = high_resolution_clock::now();
  burbuja(T, n);
tdespues = high_resolution_clock::now();
transcurrido = duration_cast<duration<double>>(tdespues - tantes);
cout << n <<" "<< transcurrido.count()<< endl;
```

2.1 Burbuja:

Tras lanzar 30 veces el programa que hace uso del algoritmo burbuja, los resultados son los siguientes:

Tamaño	**Tiempo**
600	0.001691
1200	0.00637
1800	0.012901
2400	0.019068
3000	0.026522
3600	0.034992
4200	0.044759
4800	0.0608
5400	0.08021
6000	0.094885
6600	0.11831
7200	0.141496
7800	0.167335
8400	0.19557
9000	0.226135
9600	0.2644
10200	0.301316
10800	0.334387
11400	0.369505
12000	0.413904
12600	0.4613
13200	0.512699
13800	0.558593
14400	0.615241
15000	0.668583
15600	0.730312
16200	0.783715
16800	0.852297
17400	0.92242
18000	0.998692

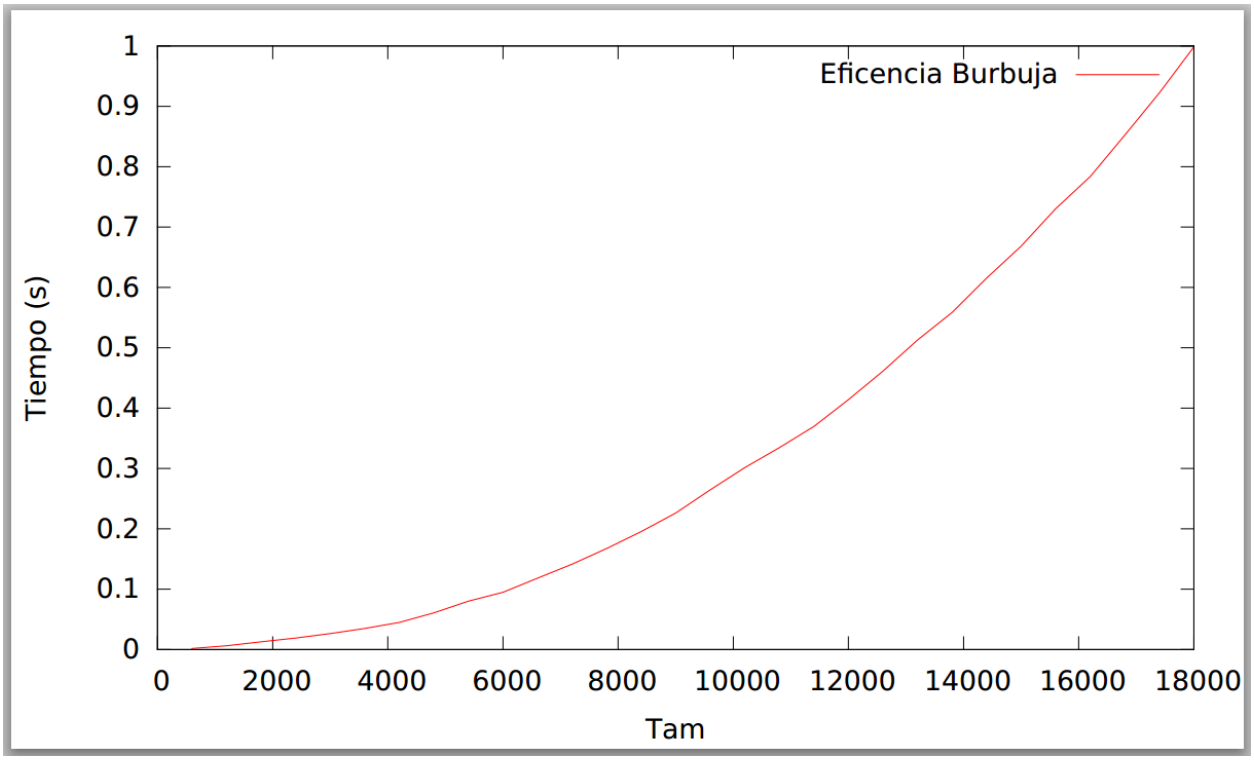
Estos tiempos han sido obtenidos lanzando el programa Burbuja en una computadora con la siguiente CPU:

CPU: Intel Cuad Core 2.4GHz

Usando GCC como compilador, sin optimizaciones en la compilación:

g++ -o burbuja burbuja.cpp -std=gnu++0x

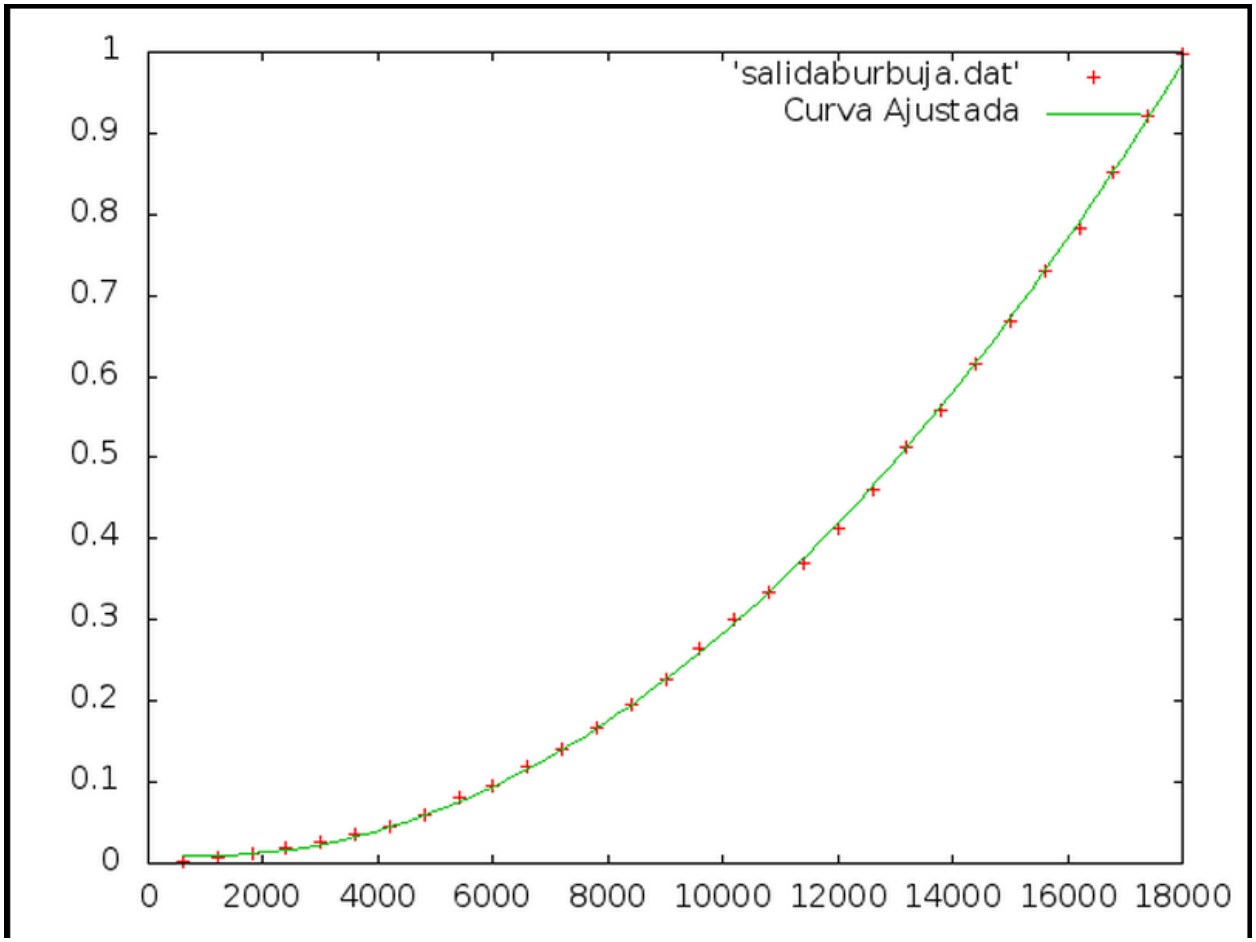
Gráfica de Burbuja



Para el ajuste híbrido de burbuja, hemos usado como ecuación teórica:

$$a2 * x^2 + a1 * x + a0$$

Haciendo uso de GNUPlot y de su opción para hacer ajustes (fit) podemos obtener la siguiente gráfica con el ajuste añadido.



Como se puede ver, el ajuste es prácticamente perfecto, lo cual nos indica, que las pruebas empíricas, y la información teórica coinciden, y para ello solo se han tenido que ajustar las tres variables a2, a1 y a0.

2.2 Inserción:

Tras lanzar 30 veces el programa que hace uso del algoritmo inserción, los resultados son los siguientes:

Tamaño	**Tiempo**
600	0.00088
1200	0.003472
1800	0.007789
2400	0.013842
3000	0.018642
3600	0.022943
4200	0.029178
4800	0.035708
5400	0.041007
6000	0.051903
6600	0.061196
7200	0.069388
7800	0.08015
8400	0.091969
9000	0.105092
9600	0.119297
10200	0.131466
10800	0.148152
11400	0.164586
12000	0.176822
12600	0.193317
13200	0.211673
13800	0.232976
14400	0.258382
15000	0.277048
15600	0.298904
16200	0.321721
16800	0.347671
17400	0.371444
18000	0.392745

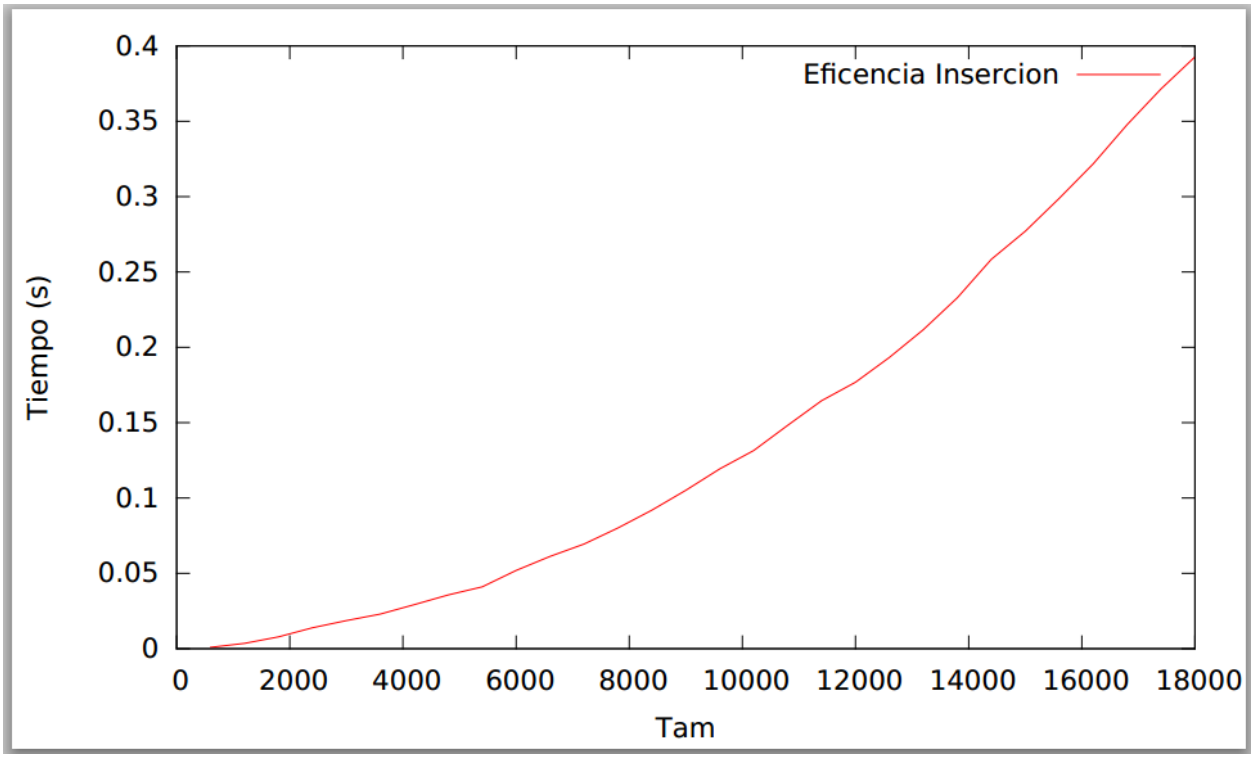
Estos tiempos han sido obtenidos lanzando el programa Inserción en una computadora con la siguiente CPU:

CPU: Intel Cuad Core 2.4GHz

Usando GCC como compilador, sin optimizaciones en la compilación:

g++ -o insercion insercion.cpp -std=gnu++0x

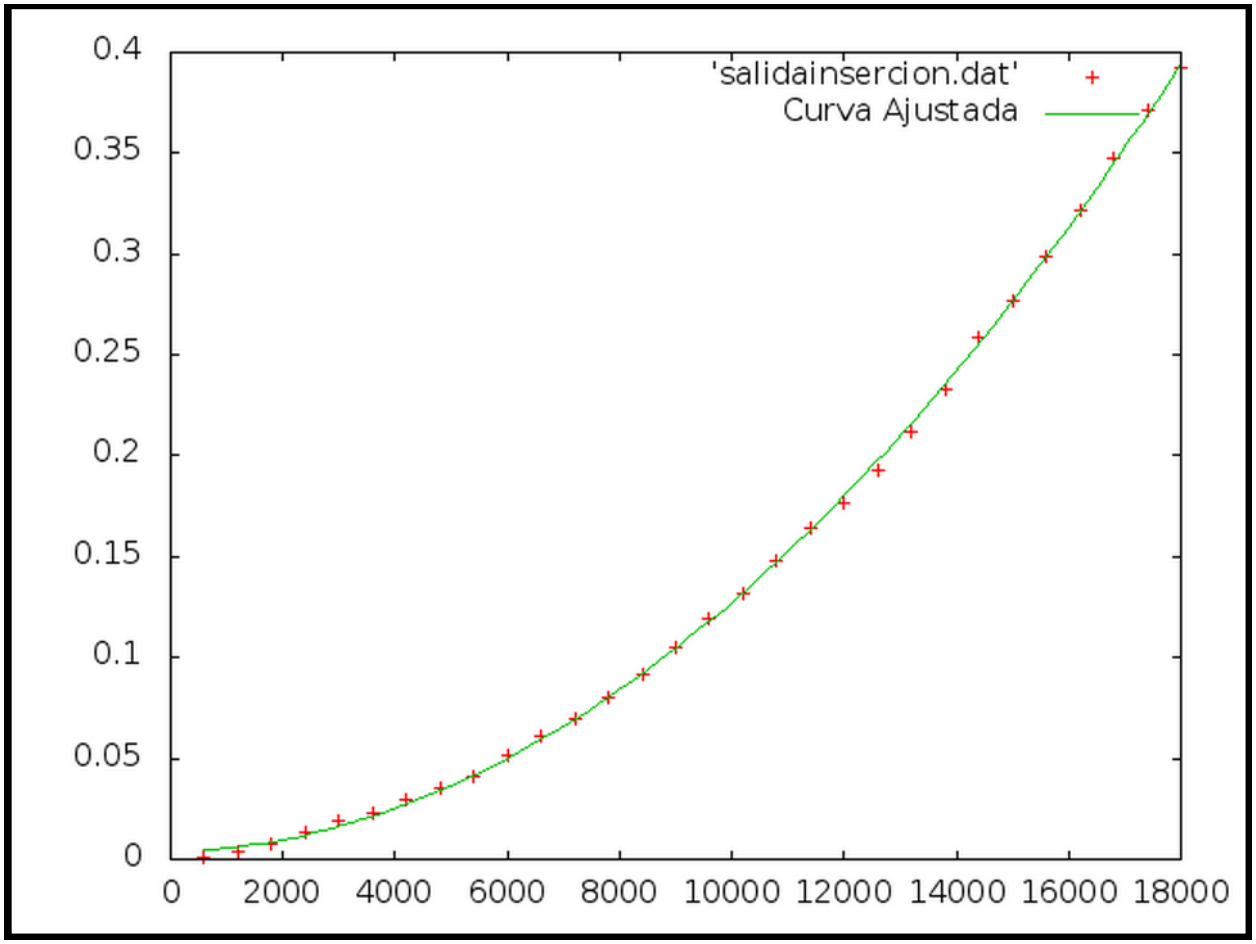
Gráfica de Inserción



Para el ajuste híbrido de inserción, hemos usado como ecuación teórica:

$$a2 * x^2 + a1 * x + a0$$

Haciendo uso de GNUPlot y de su opción para hacer ajustes (fit) podemos obtener la siguiente gráfica con el ajuste añadido.



Como se puede ver, de nuevo el ajuste es prácticamente perfecto, lo cual nos indica, que las pruebas empíricas, y la información teórica coinciden, y para ello solo se han tenido que ajustar las tres variables a2, a1 y a0.

2.3 Selección:

Tras lanzar 30 veces el programa que hace uso del algoritmo selección, los resultados son los siguientes:

Tamaño	Tiempo
600	0.00106
1200	0.004108
1800	0.009173
2400	0.01279
3000	0.021651
3600	0.022833
4200	0.029475
4800	0.035496
5400	0.042734
6000	0.052437
6600	0.062053
7200	0.075544
7800	0.088085
8400	0.104864
9000	0.11895
9600	0.135256
10200	0.150376
10800	0.166775
11400	0.180581
12000	0.199393
12600	0.219266
13200	0.240365
13800	0.261795
14400	0.286097
15000	0.308235
15600	0.333488
16200	0.360024
16800	0.388219
17400	0.41434
18000	0.444877

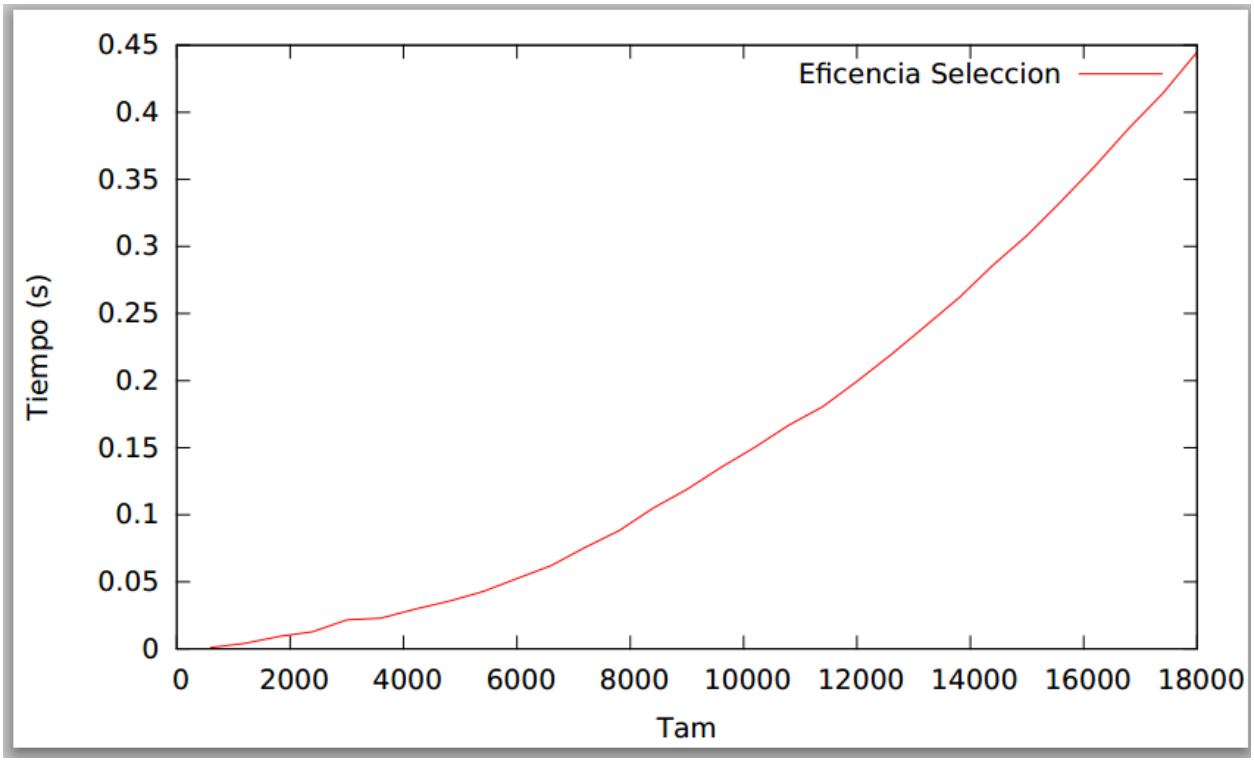
Estos tiempos han sido obtenidos lanzando el programa Selección en una computadora con la siguiente CPU:

CPU: Intel Cuad Core 2.4GHz

Usando GCC como compilador, sin optimizaciones en la compilación:

g++ -o seleccion seleccion.cpp -std=gnu++0x

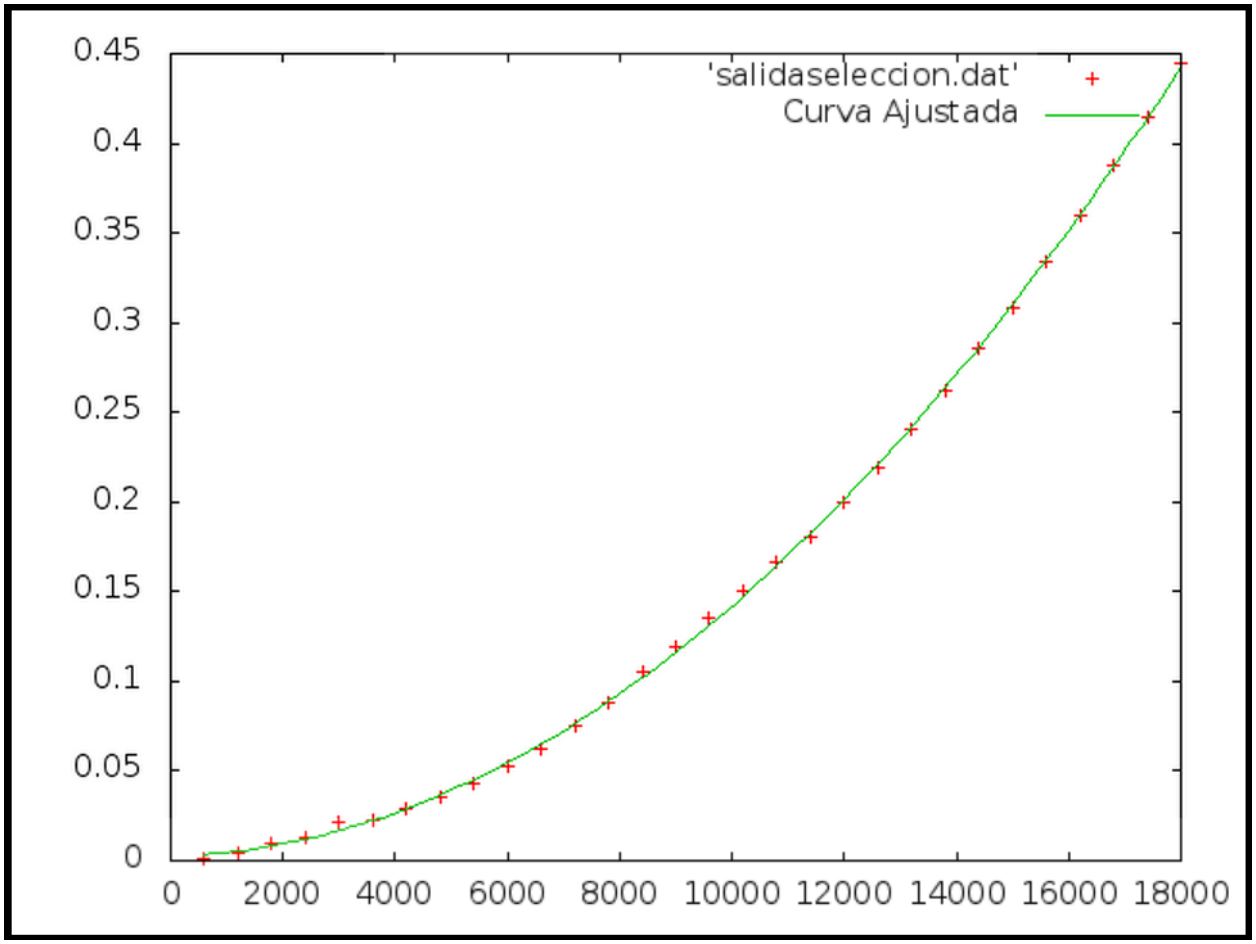
Gráfica de Selección



Para el ajuste híbrido de selección, hemos usado como ecuación teórica:

$$a2 * x^2 + a1 * x + a0$$

Haciendo uso de GNUPlot y de su opción para hacer ajustes (fit) podemos obtener la siguiente gráfica con el ajuste añadido.



Como se puede ver, de nuevo el ajuste es prácticamente perfecto, lo cual nos indica, que las pruebas empíricas, y la información teórica coinciden, y para ello solo se han tenido que ajustar las tres variables a2, a1 y a0.

3 Análisis de Algoritmos O($n * log(n)$):

Para analizar algoritmos con orden de eficiencia O($n * log(n)$), vamos a usar 3 algoritmos clásicos de ordenación, mergesort, heapsort y quicksort.

Para cada uno de ellos hemos realizado un programa que hace uso de cada uno de los algoritmos y vamos a seguir la misma metodología que con los 3 algoritmos anteriores, con la única diferencia de que los tamaños tomados son mucho mayores, ya que la eficiencia es mayor.

3.1 Mergesort:

Tras lanzar 30 veces el programa que hace uso del algoritmo mergesort, los resultados son los siguientes:

Tamaño	**Tiempo**
440000	0.117216
880000	0.247595
1320000	0.423726
1760000	0.527631
2200000	0.691052
2640000	0.878596
3080000	1.10718
3520000	1.07859
3960000	1.26248
4400000	1.43233
4840000	1.63946
5280000	1.82988
5720000	2.04043
6160000	2.31724
6600000	2.18719
7040000	2.26642
7480000	2.44213
7920000	2.69676
8360000	2.93796
8800000	3.06547

9240000	3.27539
9680000	3.38329
10120000	3.59226
10560000	3.76451

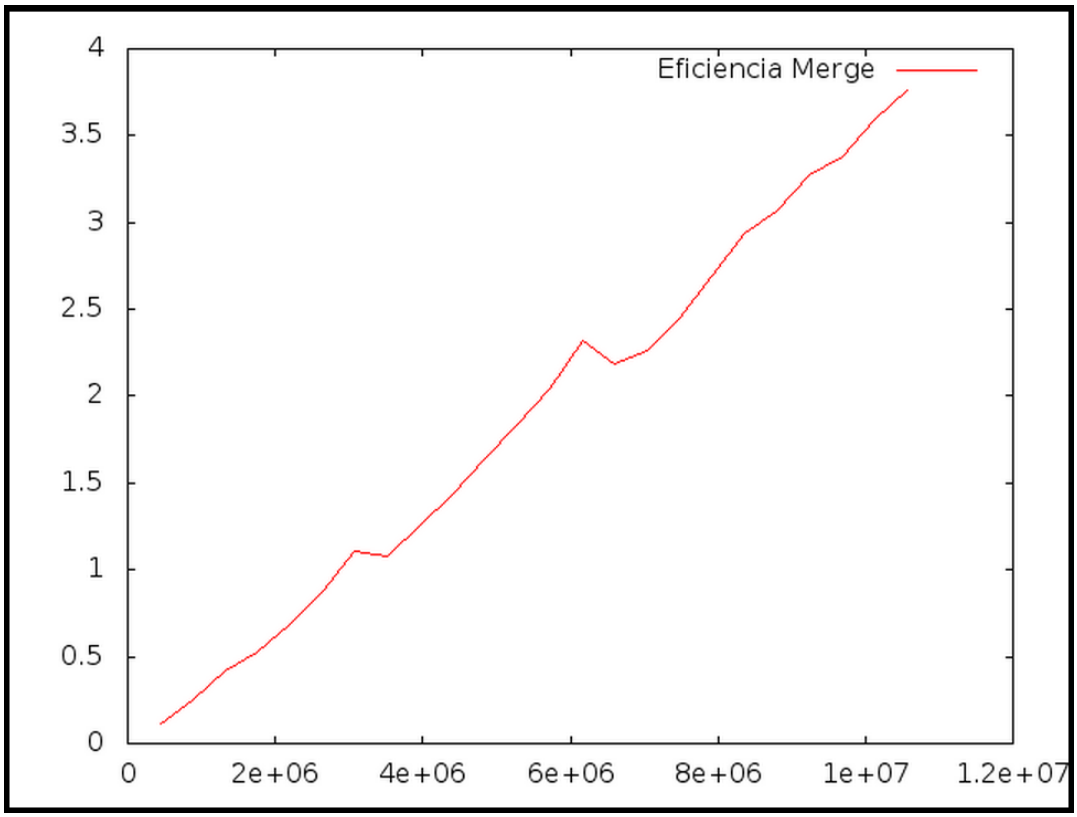
Estos tiempos han sido obtenidos lanzando el programa Mergesort en una computadora con la siguiente CPU:

CPU: Intel i5 1.8GHz

Usando GCC como compilador, sin optimizaciones en la compilación:

g++ -o mergesort mergesort.cpp -std=gnu++0x

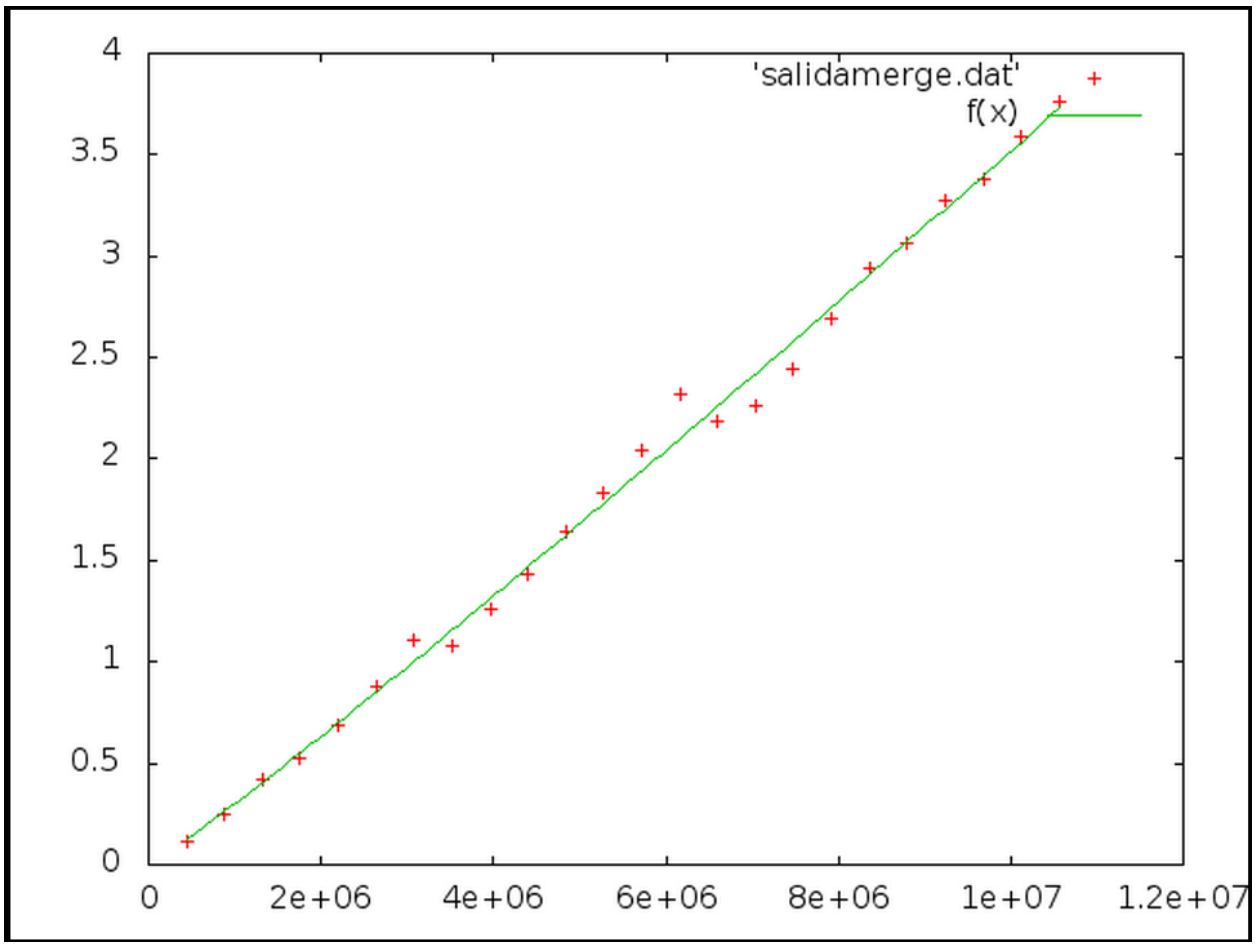
Gráfica de Mergesort



Para el ajuste híbrido de mergesort, hemos usado como ecuación teórica:

$$a1 * log(n) + a2 * n + a3$$

Haciendo uso de GNUPlot y de su opción para hacer ajustes (fit) podemos obtener la siguiente gráfica con el ajuste añadido.



Como se puede ver, de nuevo el ajuste es prácticamente perfecto, lo cual nos indica, que las pruebas empíricas, y la información teórica coinciden, y para ello solo se han tenido que ajustar las tres variables a2, a1 y a0.

3.2 Heapsort:

Tras lanzar 30 veces el programa que hace uso del algoritmo heapsort, los resultados son los siguientes:

Tamaño	Tiempo
440000	0.117886
880000	0.266093
1320000	0.423627
1760000	0.622004
2200000	0.832199
2640000	0.997926
3080000	1.19385
3520000	1.39661
3960000	1.59869
4400000	1.85857
4840000	2.01449
5280000	2.26401
5720000	2.47571
6160000	2.68167
6600000	2.91293
7040000	3.13056
7480000	3.38183
7920000	3.59564
8360000	3.81913
8800000	4.10482
9240000	4.73184
9680000	4.82404
10120000	5.50716
10560000	5.49905

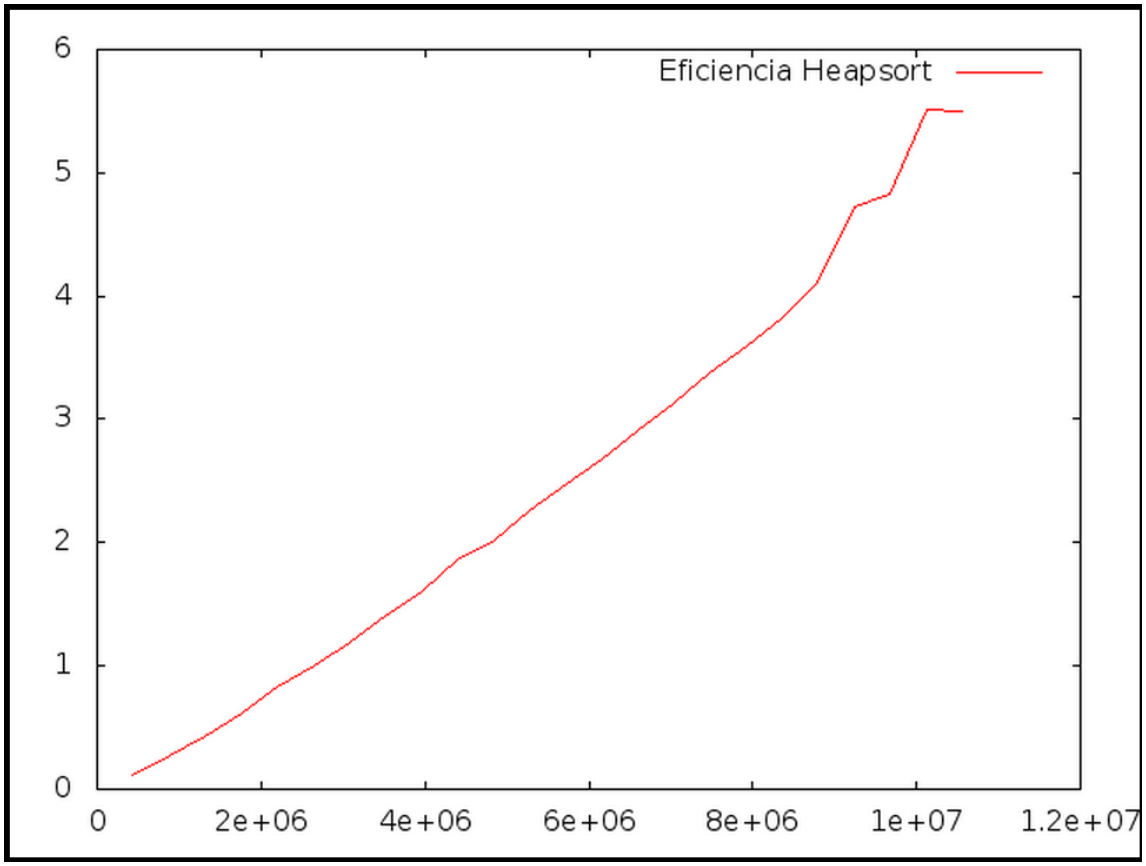
Estos tiempos han sido obtenidos lanzando el programa Heapsort en una computadora con la siguiente CPU:

CPU: Intel i5 1.8GHz

Usando GCC como compilador, sin optimizaciones en la compilación:

g++ -o heapsort heapsort.cpp -std=gnu++0x

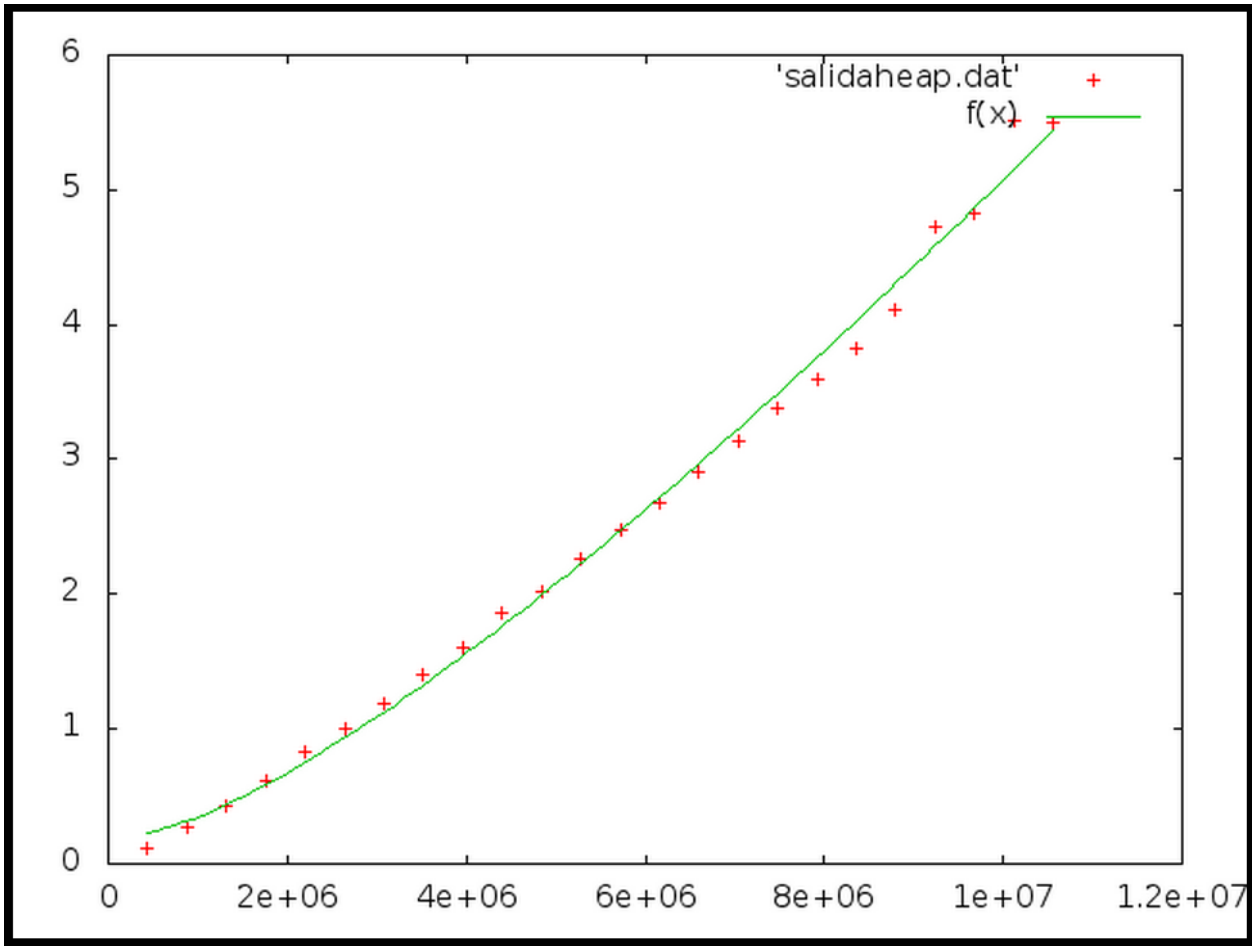
Gráfica de Heapsort



Para el ajuste híbrido de heapsort, hemos usado como ecuación teórica:

$$a1 * log(n) + a2 * n + a3$$

Haciendo uso de GNUPlot y de su opción para hacer ajustes (fit) podemos obtener la siguiente gráfica con el ajuste añadido.



Como se puede ver, de nuevo el ajuste es prácticamente perfecto, lo cual nos indica, que las pruebas empíricas, y la información teórica coinciden, y para ello solo se han tenido que ajustar las tres variables a2, a1 y a0.

3.3 Quicksort:

Tras lanzar 30 veces el programa que hace uso del algoritmo quicksort, los resultados son los siguientes:

Tamaño	**Tiempo**
440000	0.083309
880000	0.173852
1320000	0.267762
1760000	0.364681
2200000	0.471275
2640000	0.56256
3080000	0.666503
3520000	0.750256
3960000	0.856058
4400000	0.96754
4840000	1.07496
5280000	1.1632
5720000	1.31597
6160000	1.49036
6600000	1.49915
7040000	1.61589
7480000	1.74239
7920000	1.87777
8360000	1.90773
8800000	1.99199
9240000	2.12387
9680000	2.21699
10120000	2.3158
10560000	2.47077

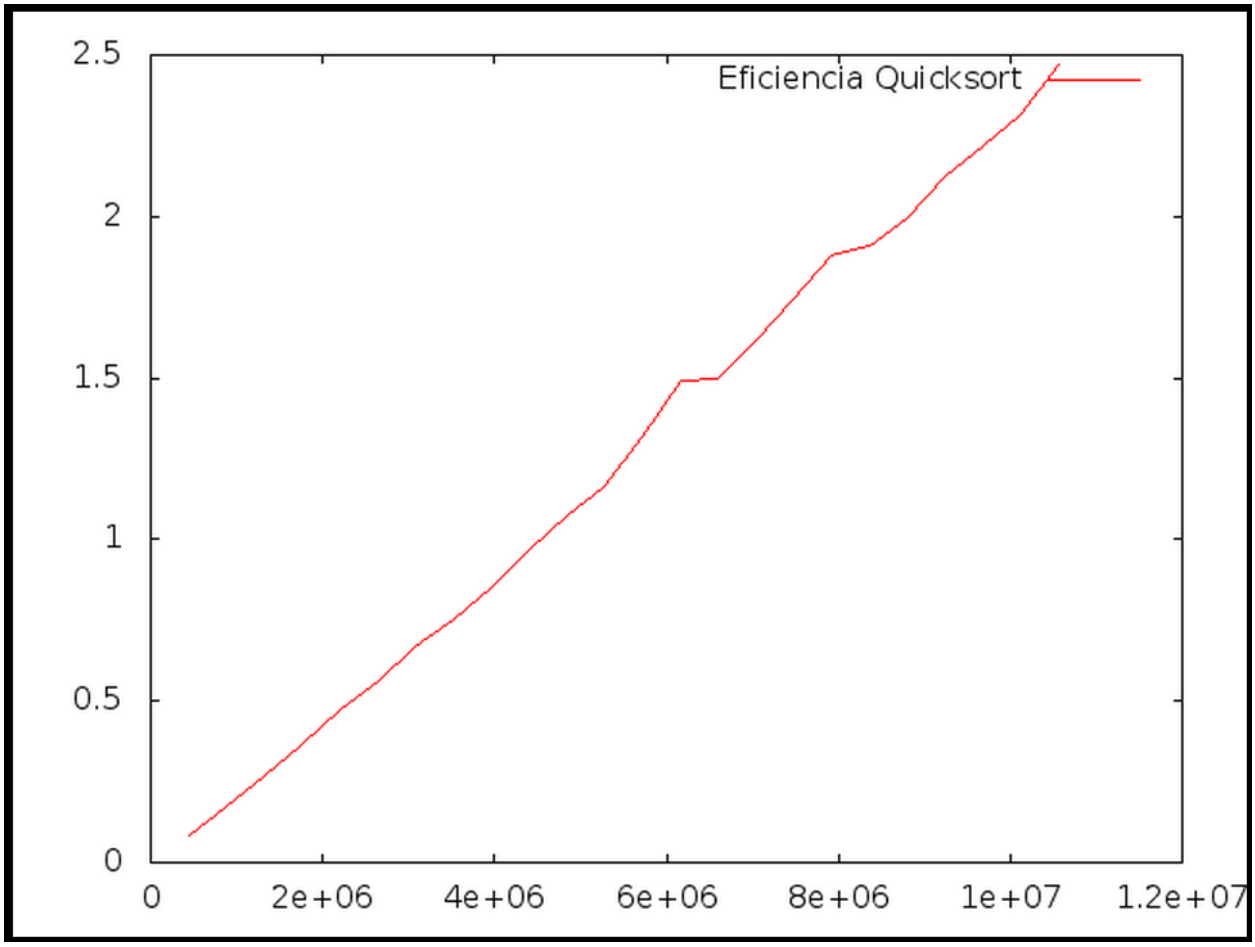
Estos tiempos han sido obtenidos lanzando el programa Quicksort en una computadora con la siguiente CPU:

CPU: Intel i5 1.8GHz

Usando GCC como compilador, sin optimizaciones en la compilación:

g++ -o quicksort quicksort.cpp -std=gnu++0x

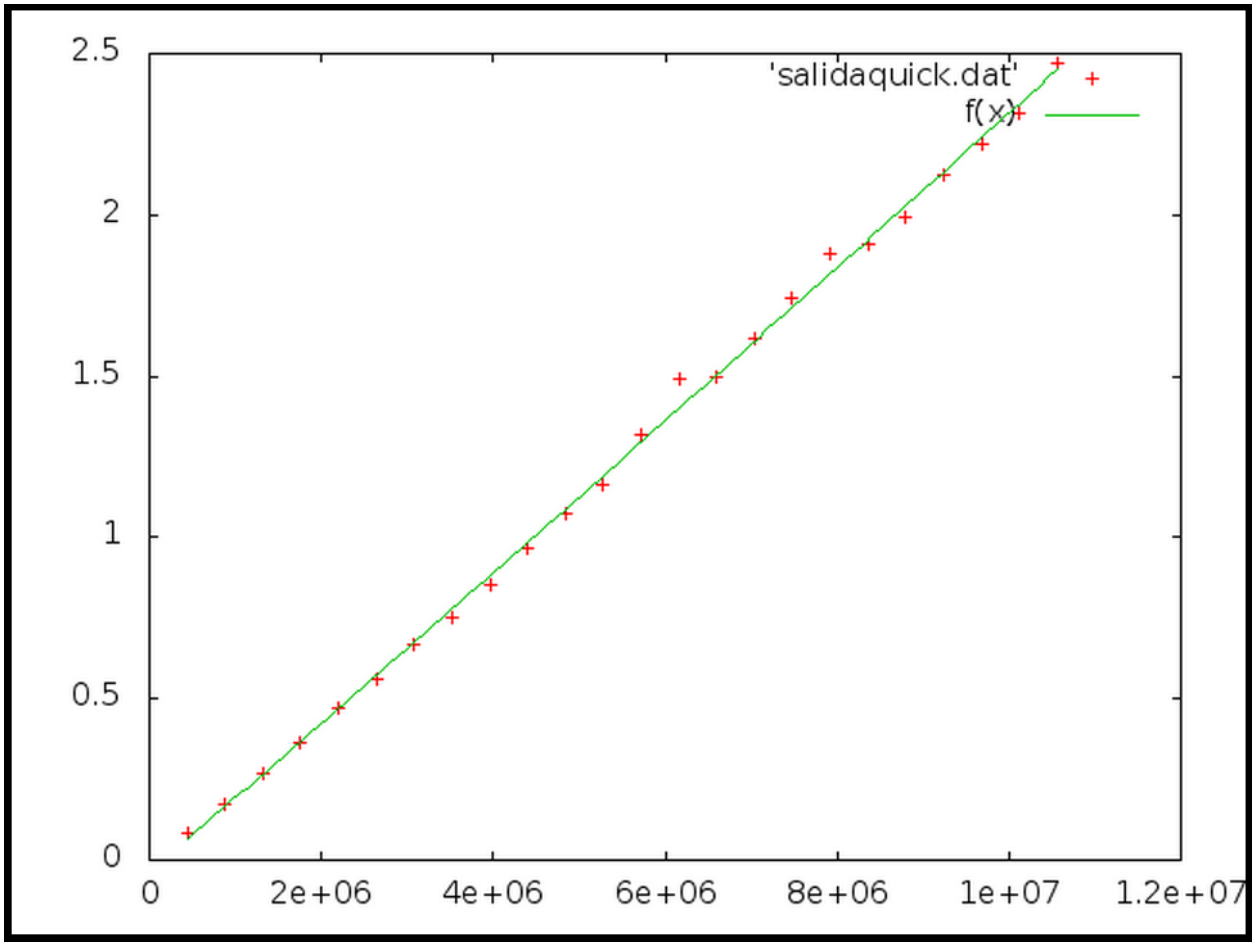
Gráfica de Quicksort



Para el ajuste híbrido de quicksort, hemos usado como ecuación teórica:

$$a1 * log(n) + a2 * n + a3$$

Haciendo uso de GNUPlot y de su opción para hacer ajustes (fit) podemos obtener la siguiente gráfica con el ajuste añadido.



Como se puede ver, de nuevo el ajuste es prácticamente perfecto, lo cual nos indica, que las pruebas empíricas, y la información teórica coinciden, y para ello solo se han tenido que ajustar las tres variables a2, a1 y a0.

4 Análisis de Algoritmos O(n^3):

Para analizar algoritmos con orden de eficiencia O(n^3), vamos a usar 1 algoritmo llamado floyd..

Hemos realizado un programa que hace uso de floyd, y vamos a seguir la metodología anterior para este mismo algoritmo, con la diferencia de que vamos a usar tamaños algo menos, ya que su eficiencia es peor que los anteriores analizados.

4.1 Floyd:

Tras lanzar 30 veces el programa que hace uso del algoritmo floyd, los resultados son los siguientes:

Tamaño	**Tiempo**
50	0.002859
100	0.011746
150	0.037103
200	0.088961
250	0.152044
300	0.260451
350	0.412891
400	0.664453
450	0.873957
500	1.25139
550	1.64292
600	2.12009
650	2.69488
700	3.4053
750	4.16914
800	5.06214
850	6.01345
900	7.12761
950	8.58812
1000	10.0742
1050	12.8864
1100	14.1108
1150	15.4335
1200	18.1446

1250	20.9482
1300	22.3479
1350	27.1117
1400	27.8493
1450	31.8034
1500	33.912

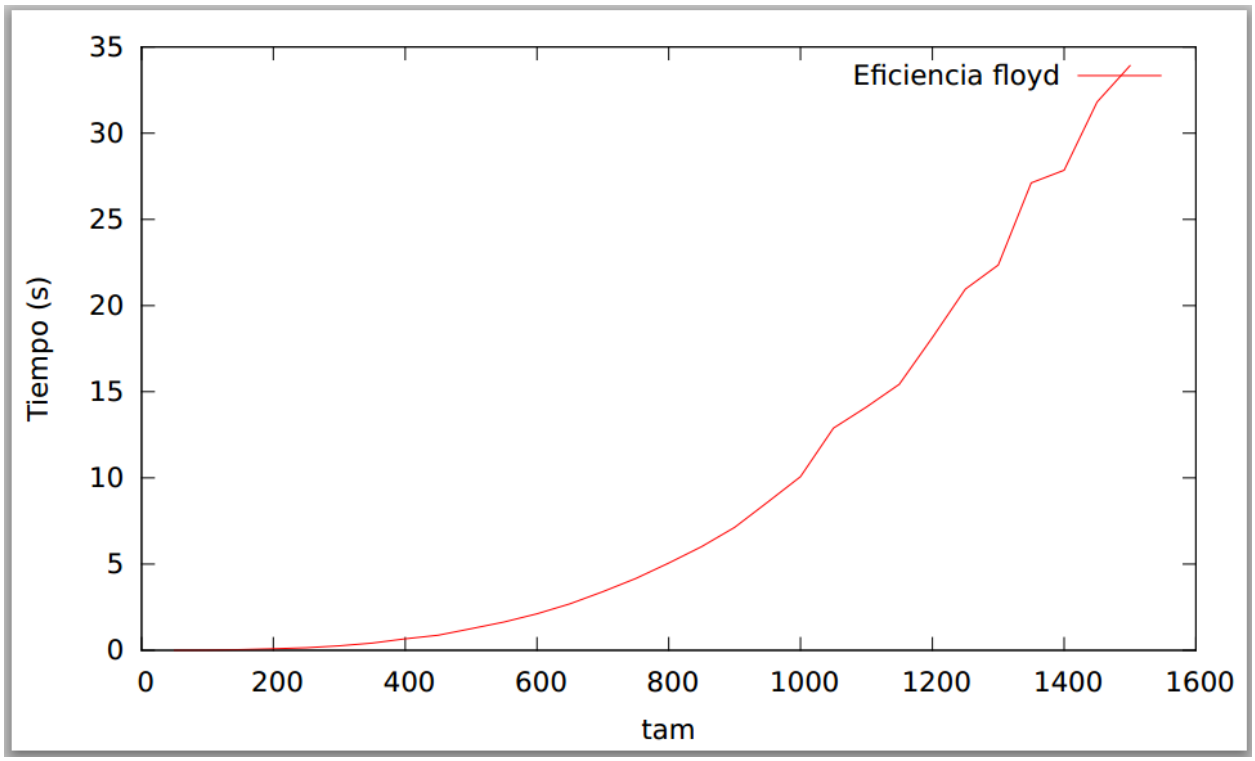
Estos tiempos han sido obtenidos lanzando el programa Floyd en una computadora con la siguiente CPU:

CPU: Intel i3 1.8GHz

Usando GCC como compilador, sin optimizaciones en la compilación:

g++ -o floyd floyd.cpp -std=gnu++0x

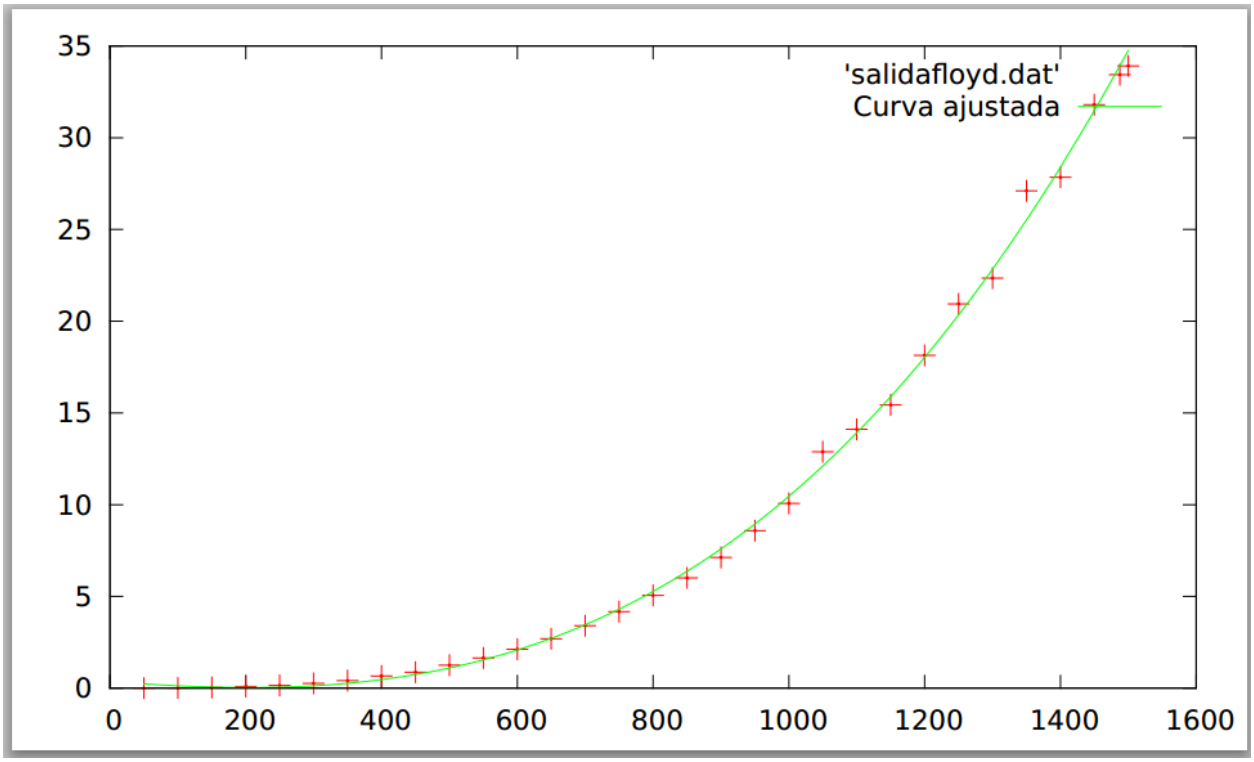
Gráfica de Floyd



Para el ajuste híbrido de floyd, hemos usado como ecuación teórica:

$$a0 * n^3 + a1 * n^2 + a2 * n + a3$$

Haciendo uso de GNUPlot y de su opción para hacer ajustes (fit) podemos obtener la siguiente gráfica con el ajuste añadido.



Como se puede ver,el ajuste es prácticamente perfecto, lo cual nos indica, que las pruebas empíricas, y la información teórica coinciden, y para ello solo se han tenido que ajustar las cuatro variables a3, a2, a1 y a0.

5 Análisis de Algoritmos O(2^n):

Para analizar algoritmos con orden de eficiencia O(2^n), vamos a usar 1 algoritmo llamado Hanoi.

Hemos realizado un programa que hace uso de hanoi, y vamos a seguir la

metodología anterior para este mismo algoritmo, con la diferencia de que vamos a usar tamaños mucho menores, ya que su eficiencia es mucho peor que los anteriores analizados.

5.1 Hanoi:

Tras lanzar 30 veces el programa que hace uso del algoritmo hanoi, los resultados son los siguientes:

Tamaño	**Tiempo**
14	0.000302
15	0.000357
16	0.000449
17	0.000996
18	0.002246
19	0.004538
20	0.007229
21	0.014434
22	0.034936
23	0.057539
24	0.118805
25	0.22651
26	0.46398
27	0.912926
28	1.81517
29	3.67474
30	7.28479
31	14.4094
32	29.1799
33	57.7322
34	116.553

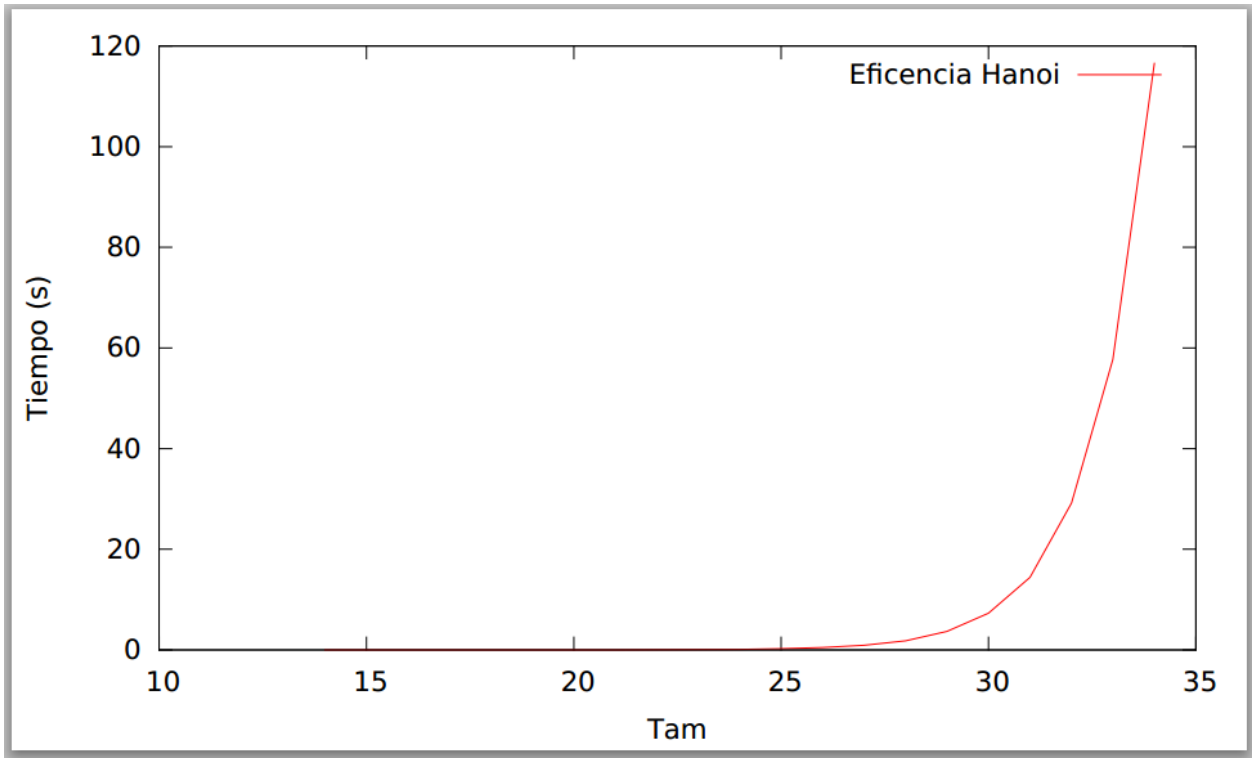
Estos tiempos han sido obtenidos lanzando el programa Hanoi en una computadora con la siguiente CPU:

CPU: Intel i7 2.43GHz

Usando GCC como compilador, sin optimizaciones en la compilación:

g++ -o hanoi hanoi.cpp -std=gnu++0x

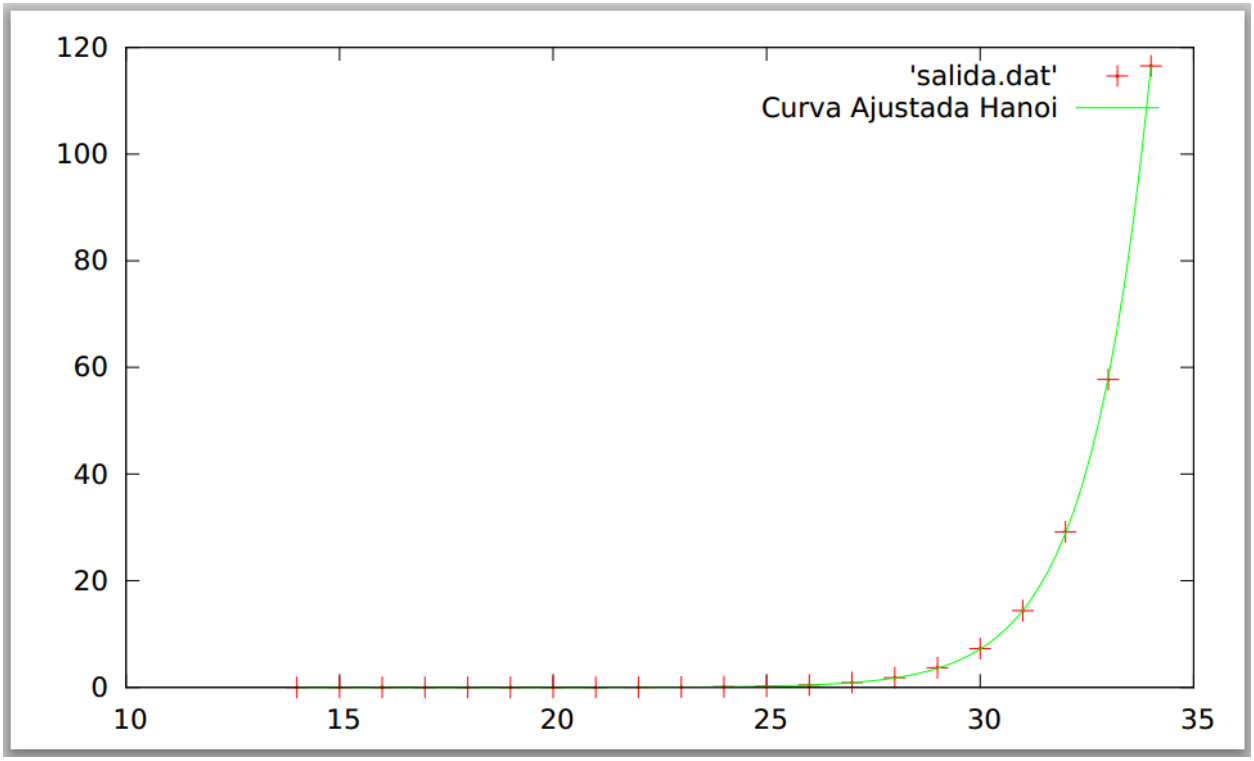
Gráfica de Hanoi



Para el ajuste híbrido de hanoi, hemos usado como ecuación teórica:

$$2^{a1*n+a0}$$

Haciendo uso de GNUPlot y de su opción para hacer ajustes (fit) podemos obtener la siguiente gráfica con el ajuste añadido.



Como se puede ver,el ajuste es prácticamente perfecto, lo cual nos indica, que las pruebas empíricas, y la información teórica coinciden, y para ello solo se han tenido que ajustar las dos variables a1 y a0.

6 Pruebas Adicionales:

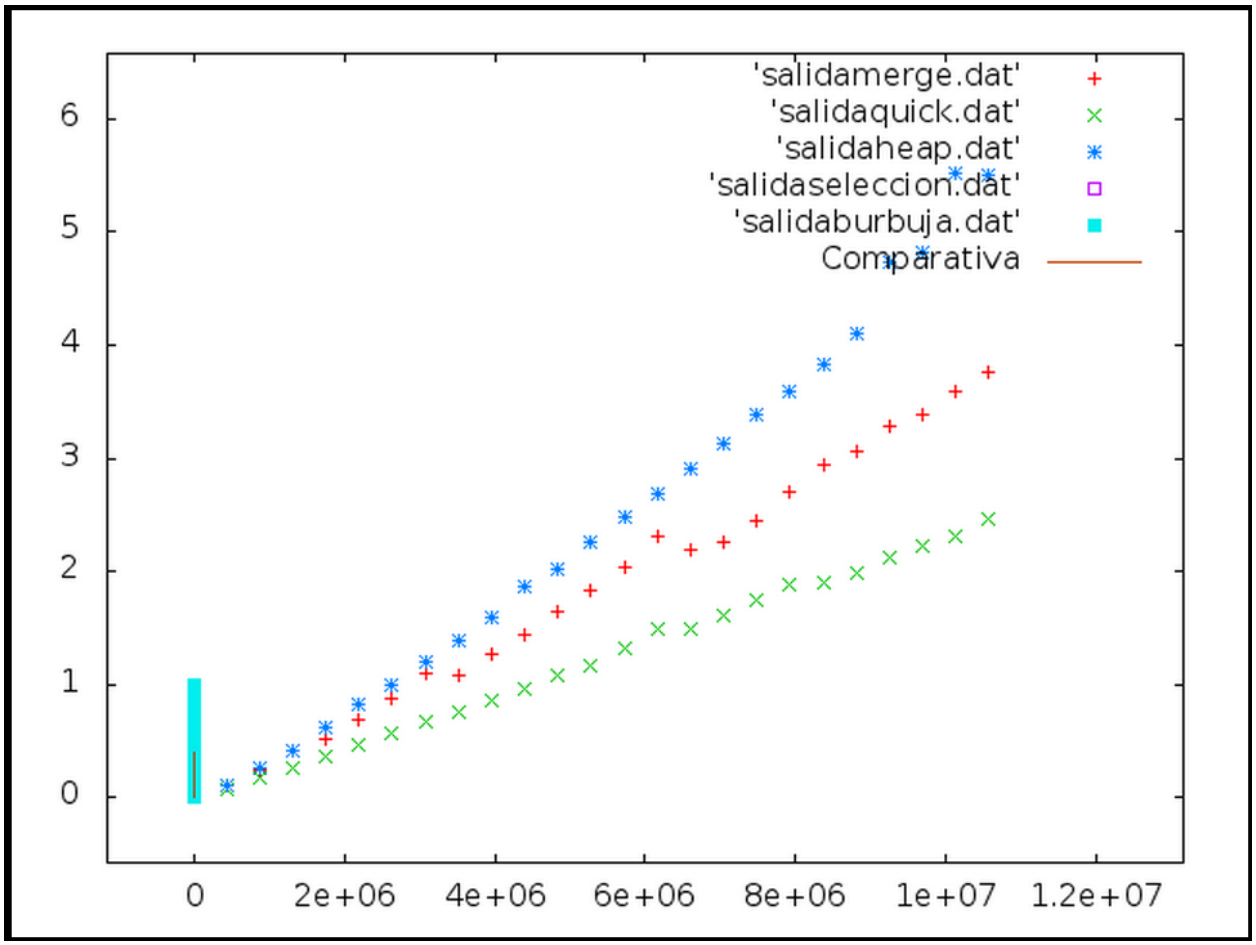
En este apartado vamos a realizar algunas pruebas adicionales, como es la comparativa entre los algoritmos de ordenación, o algunas variaciones en el analisis de algunos algoritmos anteriormente analizados.

6.1 Comparación entre los Algoritmos de ordenación:

Comprar los algoritmos de ordenación entre sí conlleva algunos problemas que debemos tener en cuenta. En primer lugar debemos tener en cuenta que no podemos lanzar baterías extremadamente exhaustivas a nuestros algoritmos debido a que tenemos tiempo limitado para la realización de esta práctica. También es importante tener

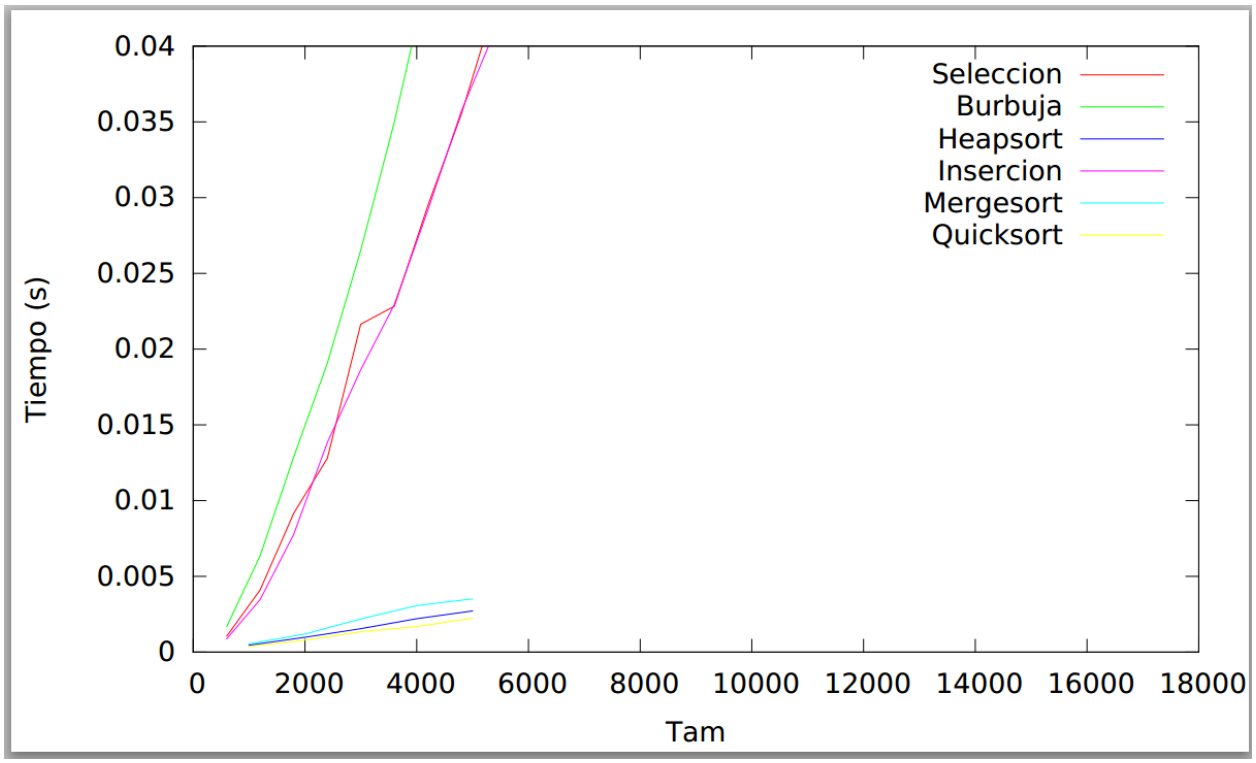
presente que comparar algoritmos del mismo orden de eficiencia puede variar mucho según los tamaños tomados, y por último tener presente que comparar algoritmos con diferente orden de eficiencia resulta solo útil para ver la diferencia existente en los órdenes de eficiencia y no para ver la diferencia entre los algoritmos en sí mismos.

En la siguiente gráfica vamos a comprar los 6 algoritmos de ordenación, Burbuja, Inserción, Selección, Mergesort, Heapsort y Quicksort. Vamos a poder apreciar que si queremos ver la progresión de los algoritmos de orden $n * log(n)$ no vamos a poder apreciar los de orden n^2 debido a los tamaños mucho mayores de los anteriores.



Para que se pueda apreciar un poco mejor el crecimiento mucho mayor de

los algoritmos de orden n^2 vamos a centrar de otra forma los ejes, obteniendo la siguiente gráfica:



En esta última gráfica podemos ver claramente que los algoritmos O(n^2) crecen mucho mayor en tiempo que los de orden O($n * log(n)$), lo cual se ajusta perfectamente con nuestra teoría

6.2 Variaciones:

A continuación vamos a analizar de nuevo algunos algoritmos, pero con ciertas variaciones, como pueden ser, cambiar la eficiencia de compilación, o cambiar la ejecución a otro PC con una CPU mejor o peor.

En primer lugar veamos qué le ocurre al Algoritmo de Hanoi si le variamos la compilación:

Compilación original:
g++ -o hanoi hanoi.cpp -std=gnu++0x

Variación de compilación:
g++ -O2 -o hanoi hanoi.cpp -std=gnu++0x

Tamaños	**Tiempo Original**	**Tiempo Variación**
14	0.000302	0.00013
15	0.000357	0.000262
16	0.000449	0.000846
17	0.000996	0.001047
18	0.002246	0.002332
19	0.004538	0.004498
20	0.007229	0.007866
21	0.014434	0.014877
22	0.034936	0.011904
23	0.057539	0.023105
24	0.118805	0.050784
25	0.22651	0.10073
26	0.46398	0.10073
27	0.912926	0.364788
28	1.81517	0.734411
29	3.67474	1.45692
30	7.28479	2.91583
31	14.4094	5.86446
32	29.1799	11.6285
33	57.7322	23.2579
34	116.553	46.499

Como podemos ver los resultados están considerablemente mejorados, llegando a dividir por la mitad o más el tiempo necesario para la ejecución con los mismos tamaños.

Ahora vamos a ver qué ocurre si lanzamos el programa Burbuja en un PC algo más rápido que el original.

El PC original tenía la siguiente CPU:
CPU: Intel Quad Core 2.4GHz

El PC diferente tiene la siguiente CPU:
CPU: Intel i7 3.4GHz

Estos son los resultados obtenidos:

Tamaño	Tiempo Original	Tiempo Nuevo
600	0.001691	0.000681
1200	0.00637	0.000775
1800	0.012901	0.000968
2400	0.019068	0.001082
3000	0.026522	0.001165
3600	0.034992	0.001232
4200	0.044759	0.001286
4800	0.0608	0.001298
5400	0.08021	0.001379
6000	0.094885	0.001399
6600	0.11831	0.001421
7200	0.141496	0.001572
7800	0.167335	0.001695
8400	0.19557	0.001709
9000	0.226135	0.001731
9600	0.2644	0.001795
10200	0.301316	0.001871
10800	0.334387	0.001901
11400	0.369505	0.001926
12000	0.413904	0.001955
12600	0.4613	0.001971
13200	0.512699	0.002001
13800	0.558593	0.002025
14400	0.615241	0.002081
15000	0.668583	0.002097
15600	0.730312	0.002117
16200	0.783715	0.002165
16800	0.852297	0.002189
17400	0.92242	0.002207
18000	0.998692	0.002235

Como podemos ver, los resultados son ampliamente diferentes, es decir, en la primera computadora, con el máximo tamaño probado, casi tardaba 1 segundo, mientras que con la segunda computadora, no llega ni a 0.0023 segundos.