Universidad de Granada

Algorítmica

Análisis de eficiencia algorítmica

Laura Calle Caraballo
Cristina María Garrido López
Germán González Almagro
Javier León Palomares
Antonio Manuel Milán Jiménez

14 de marzo de 2016

Índice

1. Introducción.

En este documento se analiza y compara la eficiencia de ocho algoritmos diferentes. Seis de ellos abordan la ordenación de vectores; por otra parte, el algoritmo de Floyd trata la búsqueda del camino más corto entre todos los pares de nodos de un grafo dirigido cualquiera y el algoritmo restante resuelve el problema de las torres de Hanoi.

2. Algoritmos de ordenación.

2.1. Algoritmos de orden O(n^2).

En esta sección, analizaremos la eficiencia empírica e híbrida de los algoritmos Burbuja, Inserción y Selección.

Para los cálculos de eficiencia híbrida emplearemos la función: $ax^2 + bx + c$.

2.1.1. Algoritmo *Burbuja*.

A continuación se muestra la gráfica de tiempos de ejecución obtenida:

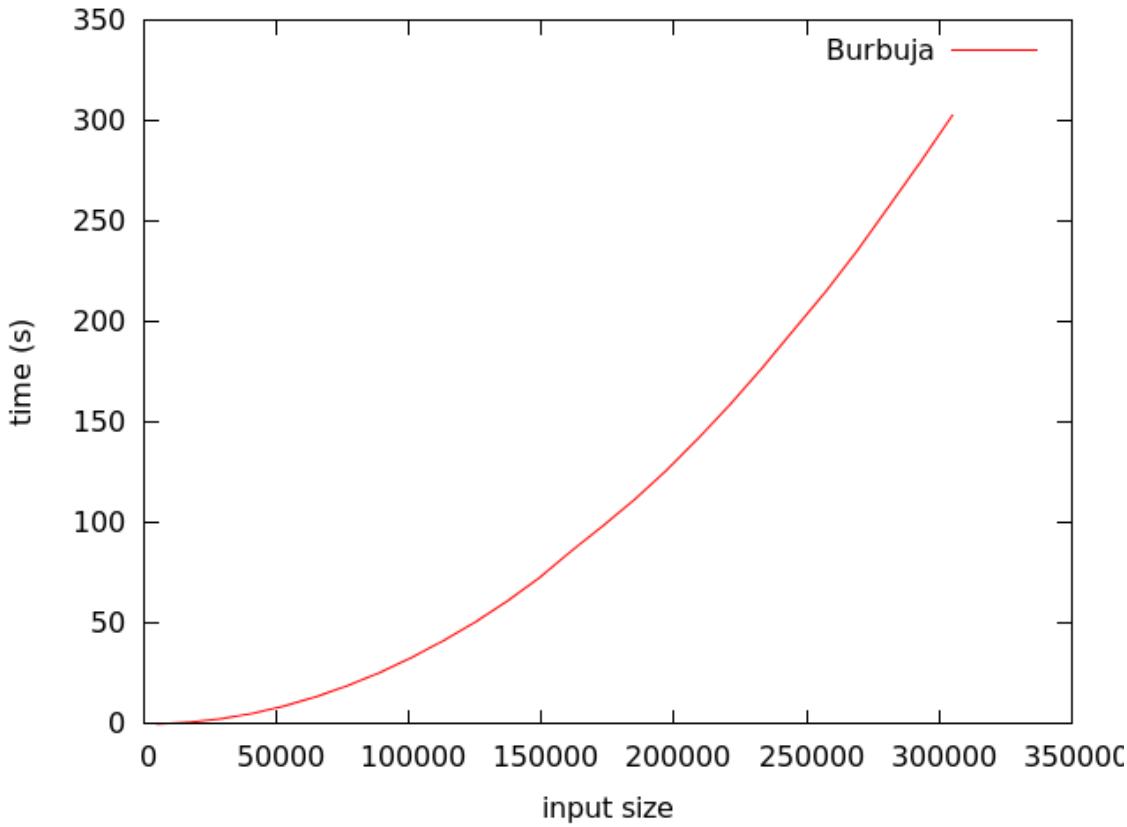

Figura 1: Gráfica del algoritmo *Burbuja*. Compilación sin optimización. Intel® Core™ i7-5500U CPU @ 2.40GHz.

El análisis híbrido realizado a partir de los datos representados en la gráfica tiene los siguientes resultados:

- $a = 3{,}25807 \cdot 10^{-9}$
- $b = -6{,}62085 \cdot 10^{-7}$
- $c = 0{,}0683299$

$f(x) = 3{,}25807 \cdot 10^{-9} \cdot x^2 - 6{,}62085 \cdot 10^{-7} \cdot x + 0{,}0683299$

La representación gráfica de los tiempos obtenidos junto con la función ajustada es la siguiente:

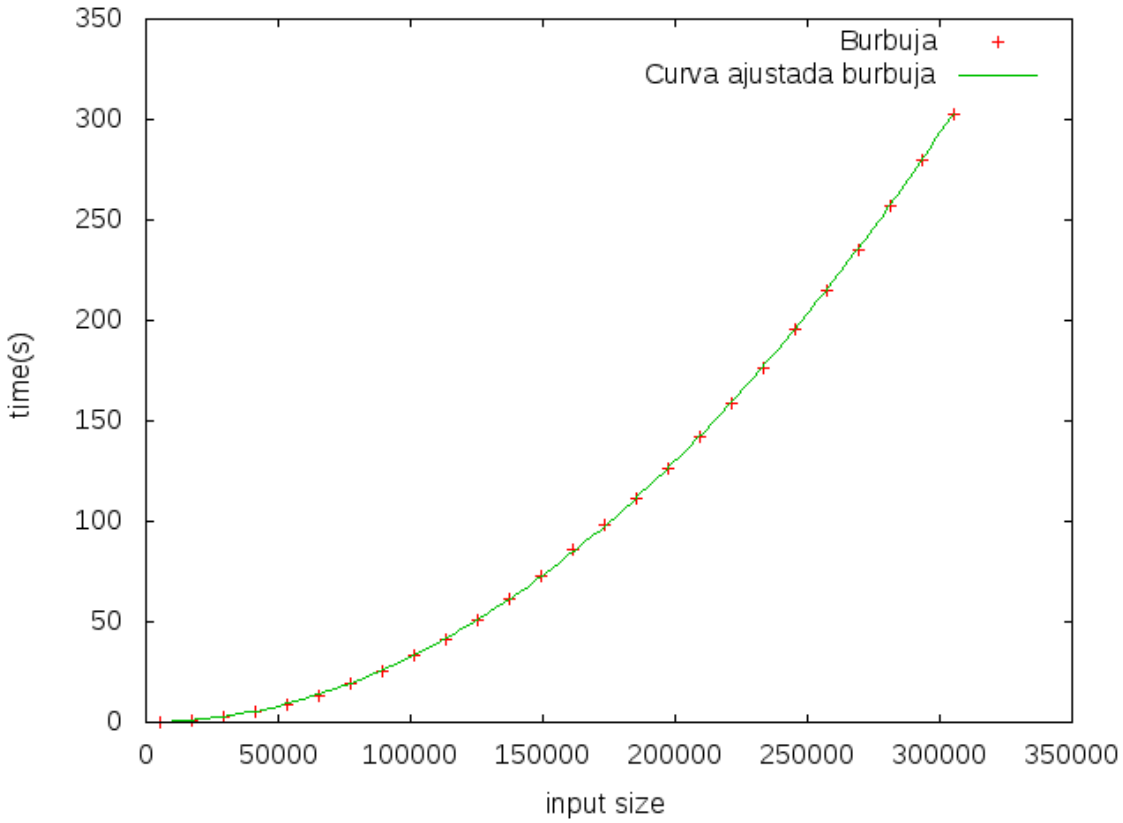

Figura 2: Gráfica del algoritmo *Burbuja* y su función ajustada. Compilación sin optimización. Intel® Core™ i7-5500U CPU @ 2.40GHz.

2.1.2. Algoritmo de Inserción.

Aquí podemos ver el crecimiento que presenta el algoritmo de Inserción conforme aumenta el volumen de la entrada:

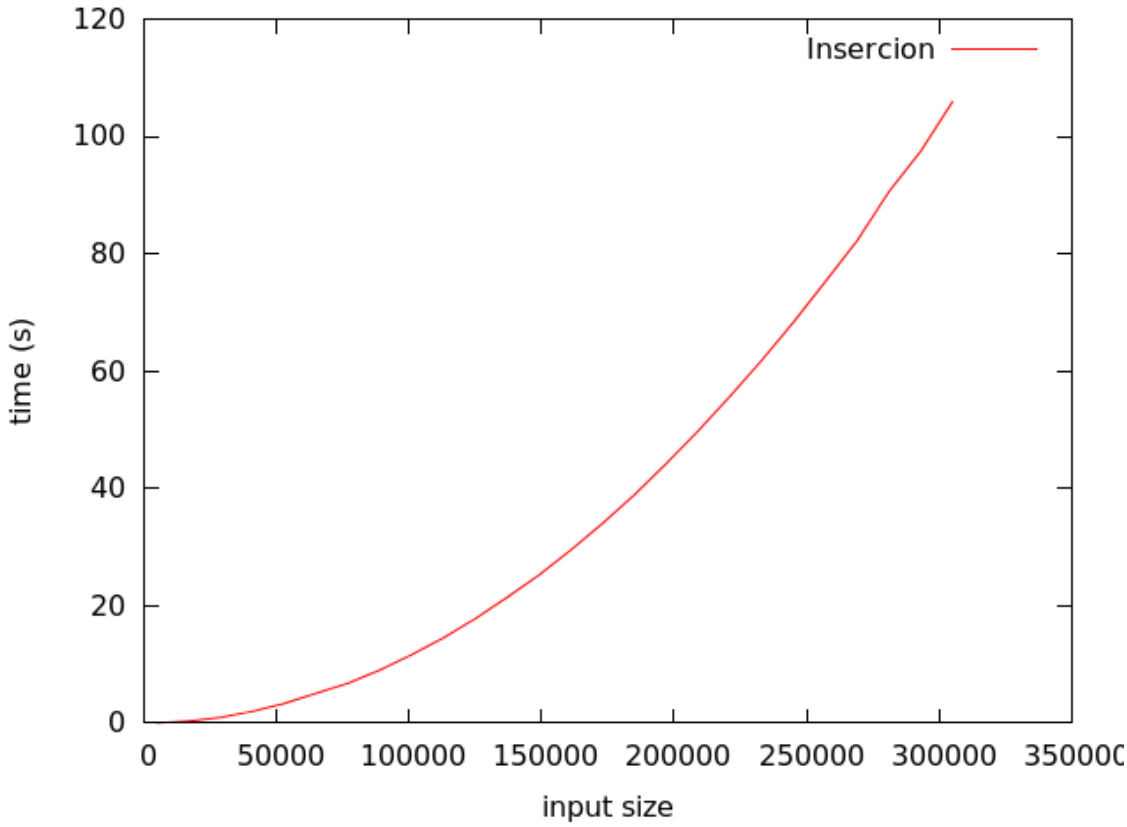

Figura 3: Gráfica del algoritmo de Inserción. Compilación sin optimización. Intel® Core™ i7-5500U CPU @ 2.40GHz.

Mediante el análisis híbrido obtenemos las siguientes constantes ocultas:

- $a = 1{,}14055 \cdot 10^{-9}$
- $b = -6{,}62085 \cdot 10^{-7}$
- $c = 0{,}0683299$

$f(x) = 1{,}14055 \cdot 10^{-9} \cdot x^2 - 6{,}62085 \cdot 10^{-7} \cdot x + 0{,}0683299$

La gráfica de los tiempos obtenidos junto con la función ajustada es la siguiente:

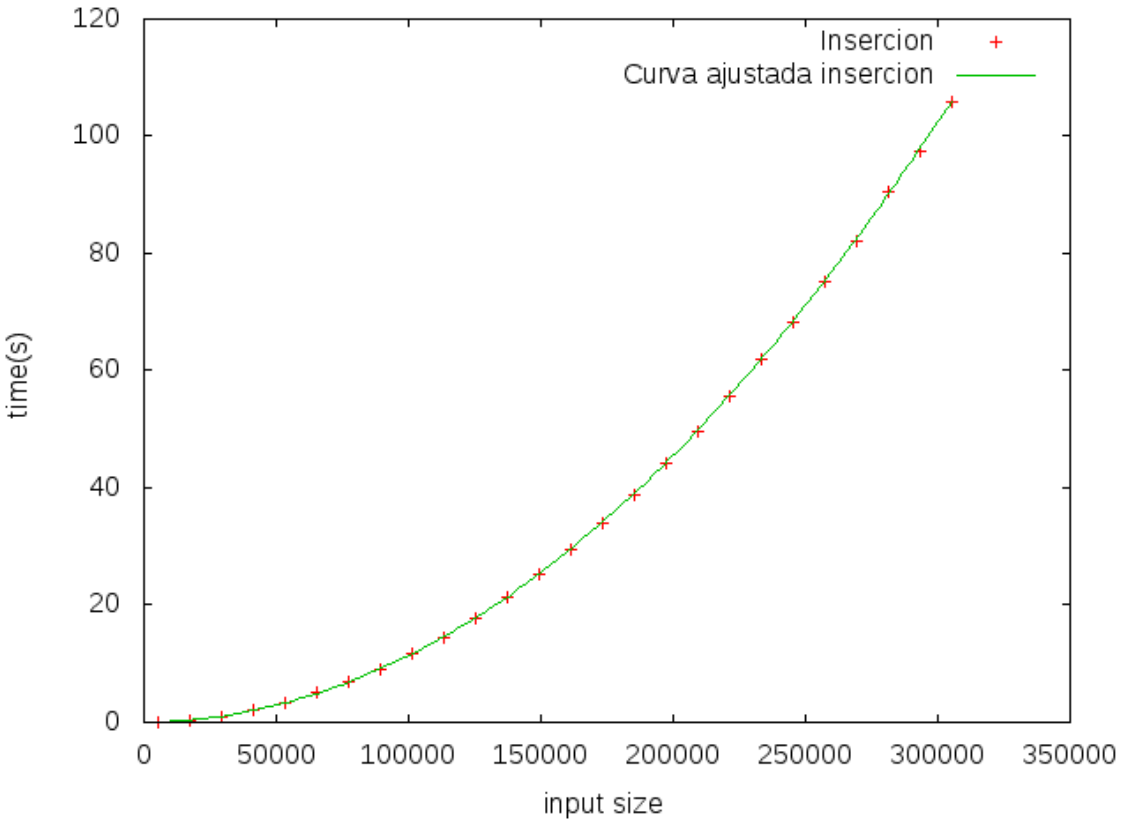

Figura 4: Gráfica del algoritmo de Inserción y su función ajustada. Compilación sin optimización. Intel® Core™ i7-5500U CPU @ 2.40GHz.

2.1.3. Algoritmo de Selección.

La curva que presenta el algoritmo de Selección conforme incrementamos el tamaño de los datos es la siguiente:

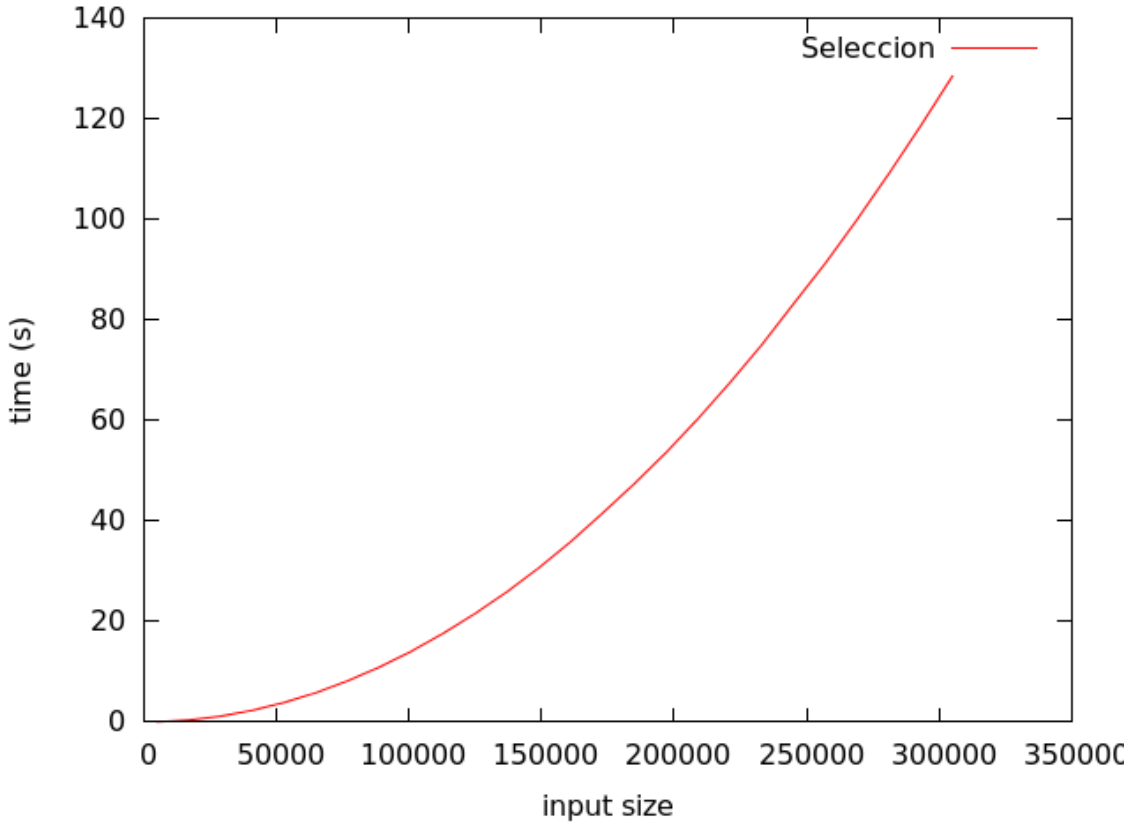

Figura 5: Gráfica del algoritmo de Selección. Compilación sin optimización. Intel® Core™ i7-5500U CPU @ 2.40GHz.

Las constantes ocultas que aproximan $f(x)$ a los datos representados en la gráfica anterior son:

- $a = 1{,}3791 \cdot 10^{-9}$
- $b = 8{,}02857 \cdot 10^{-7}$
- $c = -0{,}0274645$

$f(x) = 1{,}3791 \cdot 10^{-9} \cdot x^2 + 8{,}02857 \cdot 10^{-7} \cdot x - 0{,}0274645$

Aquí podemos ver el ajuste de los datos a la función calculada en el apartado anterior:

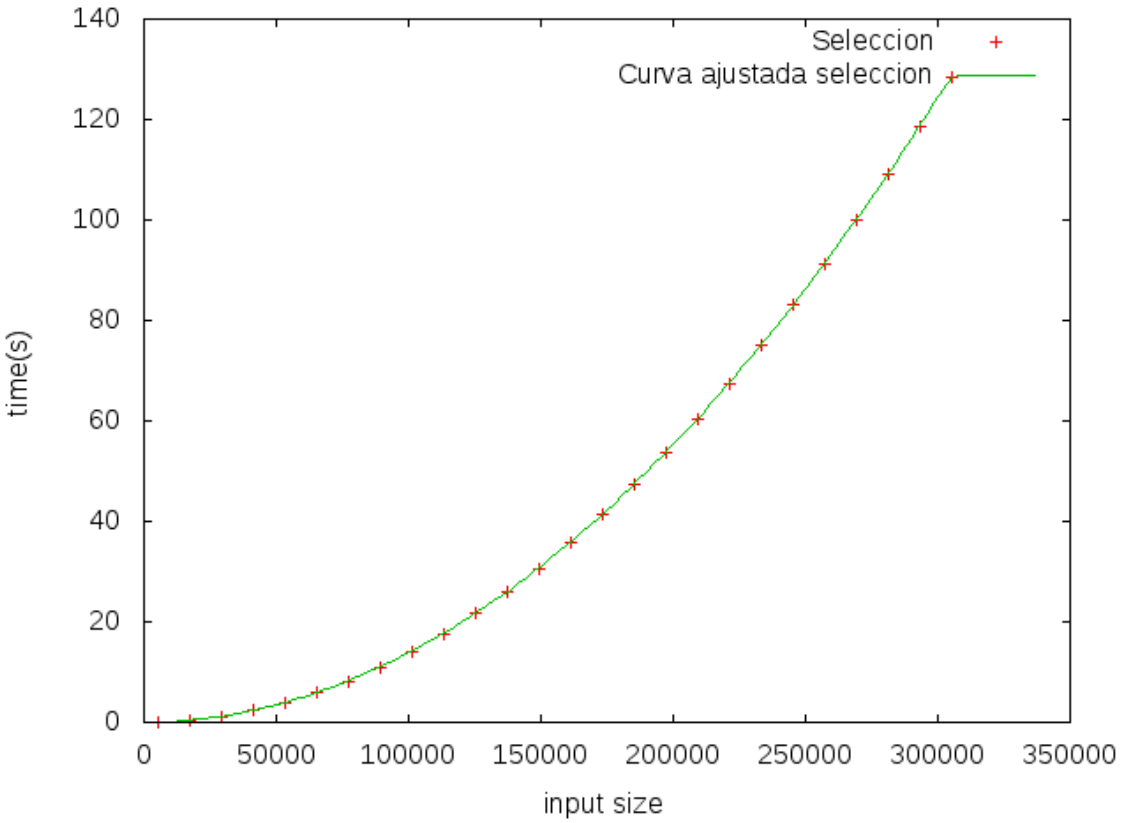

Figura 6: Gráfica del algoritmo de Selección y su función ajustada. Compilación sin optimización. Intel® Core™ i7-5500U CPU @ 2.40GHz.

2.1.4. Comparativa de los algoritmos de O(n^2).

En primer lugar, presentamos una gráfica para observar mejor las diferencias de rendimiento entre los distintos algoritmos:

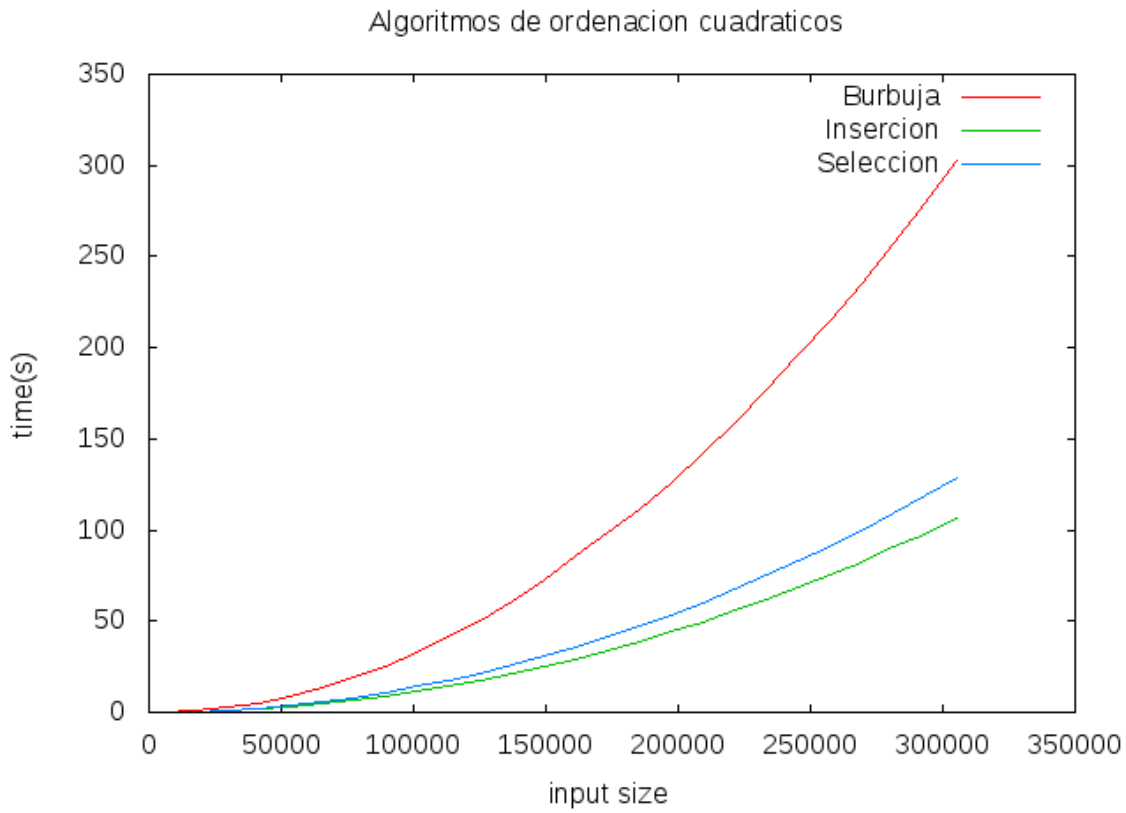

Figura 7: Gráfica comparativa de los algoritmos de ordenación cuadráticos. Compilación sin optimización. Intel® Core™ i7-5500U CPU @ 2.40GHz.

Aunque los tres algoritmos tienen una eficiencia de O(n^2) y, según la notación asintótica, los tres algoritmos tardarían potencialmente lo mismo, en esta gráfica podemos ver que no es así. El algoritmo *Burbuja* se diferencia rápidamente de los otros y presenta un crecimiento mucho más rápido, algo que veíamos ya con las constantes ocultas anteriores; por lo tanto, es bastante más costoso aun teniendo el mismo orden de eficiencia. También podemos comprobar que, si bien Inserción y Selección al principio tienen una eficiencia muy parecida, conforme se aumenta el volumen de datos el de Selección resulta algo más ineficiente.

A continuación, la tabla de tiempos de ejecución:

Tamaño de n	**Tiempo Burbuja (s)**	**Tiempo Inserción (s)**	**Tiempo Selección (s)**
5000	0.059966	0.02864	0.035099
17000	0.85589	0.331283	0.400708
29000	2.5781	0.95447	1.16477
41000	5.27445	1.98294	2.32755
53000	8.96341	3.33574	3.88535
65000	13.5704	5.05818	5.84135
77000	19.1414	6.77307	8.19565
89000	25.6015	9.02627	10.9497
101000	33.0638	11.6468	14.0946
113000	41.4656	14.5133	17.6426
125000	50.6853	17.7904	21.5895
137000	61.1032	21.4265	25.9274
149000	72.5801	25.2194	30.6664
161000	85.8144	29.4826	35.8634
173000	98.377	34.036	41.5437
185000	111.575	38.9165	47.4152
197000	126.05	44.2132	53.6117
209000	141.952	49.7696	60.3333
221000	158.771	55.6643	67.4613
233000	176.711	61.8095	74.9793
245000	195.801	68.3384	83.151
257000	214.78	75.2418	91.2287
269000	235.283	82.2099	99.932
281000	257.423	90.579	109.087
293000	279.65	97.4804	118.599
305000	302.741	105.925	128.49

2.2. Algoritmos de orden O($n \log n$).

En esta sección, analizaremos la eficiencia empírica e híbrida de los algoritmos *Heapsort*, *Mergesort* y *Quicksort*.

En los cálculos de eficiencia híbrida ajustaremos la función: $ax \log(x)$.

2.2.1. Algoritmo *Heapsort*.

Éste es el crecimiento que se experimenta cuando el algoritmo *Heapsort* maneja diferentes tamaños de datos:

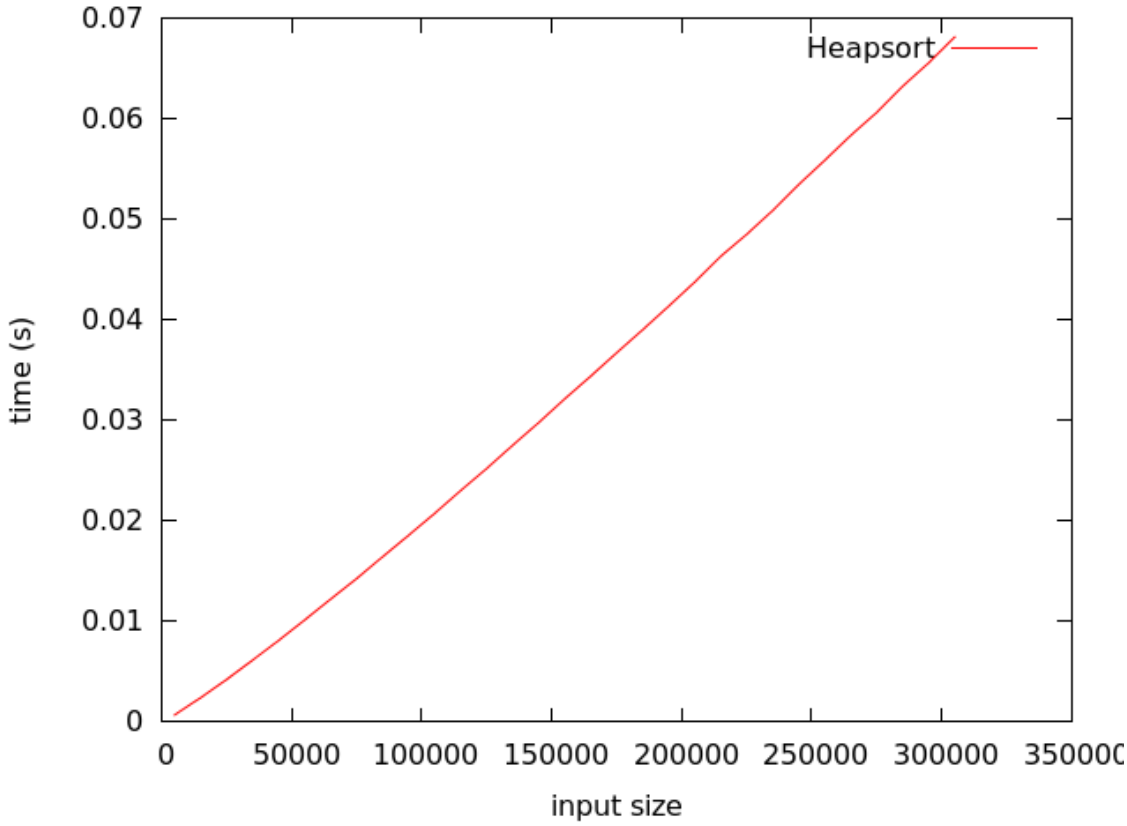

Figura 8: Gráfica del algoritmo *Heapsort*. Compilación sin optimización. Intel® Core™ i7-5500U CPU @ 2.40GHz.

Para ajustar $f(x)$ a la gráfica anterior, necesitamos los siguientes valores:

- $a = 1{,}7522 \cdot 10^{-8}$

$f(x) = 1{,}7522 \cdot 10^{-8} \cdot x \cdot \log(x)$

Si representamos los datos junto con la función ajustada, obtenemos la siguiente gráfica:

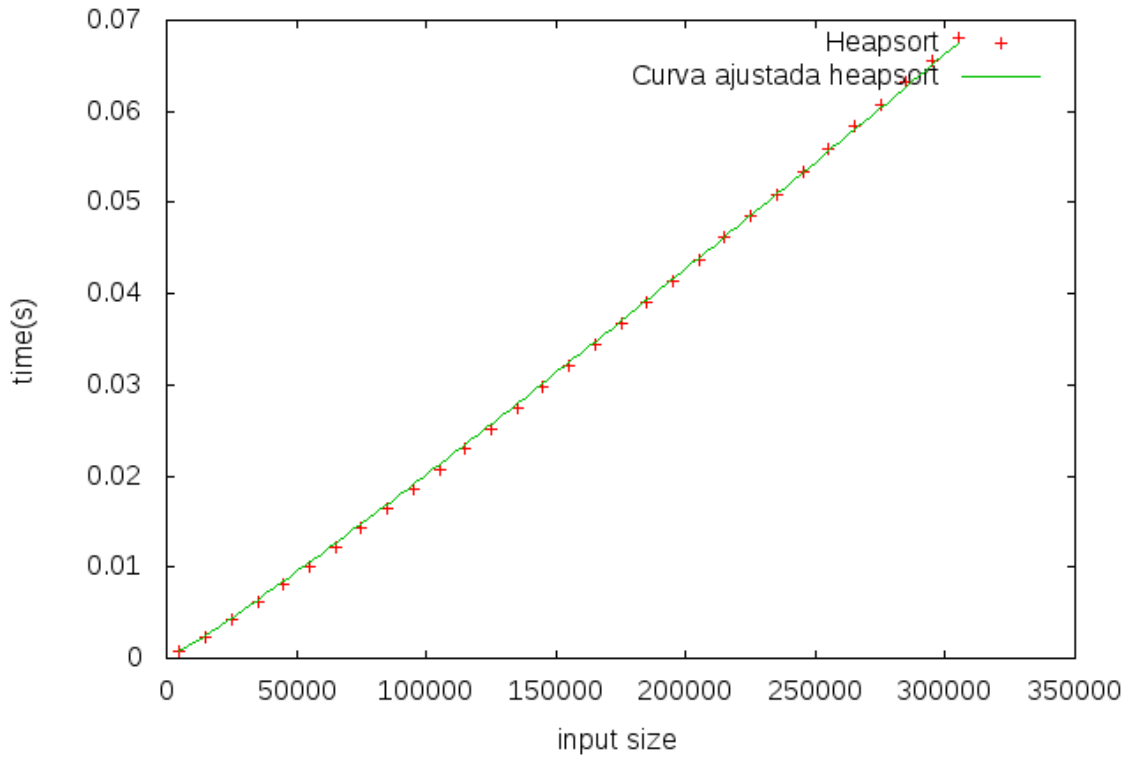

Figura 9: Gráfica del algoritmo *Heapsort* y su función ajustada. Compilación sin optimización. Intel® Core™ i7-5500U CPU @ 2.40GHz.

2.2.2. Algoritmo *Mergesort*.

A continuación mostramos gráficamente los tiempos de ejecución del algoritmo *Mergesort*:

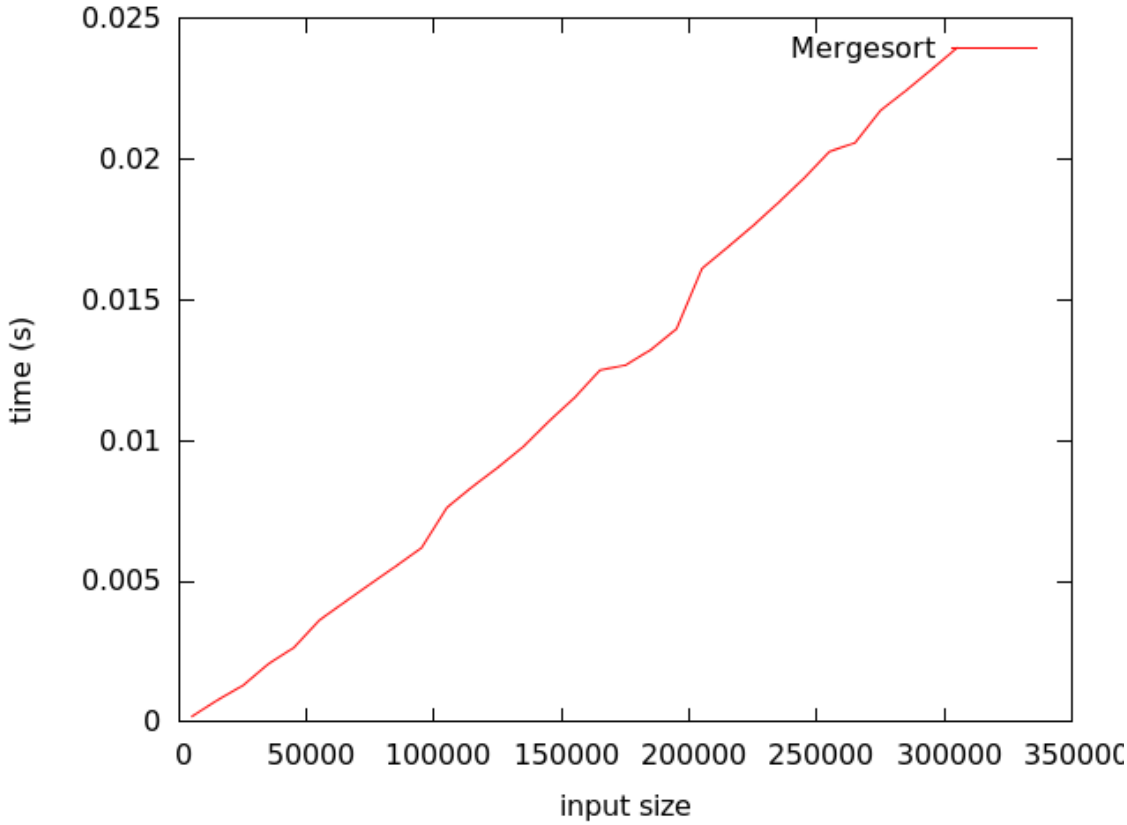

Figura 10: Gráfica del algoritmo *Mergesort*. Compilación sin optimización. Intel® Core™ i7-5500U CPU @ 2.40GHz.

El resultado que da el análisis híbrido del algoritmo *Mergesort* es:

- $a = 6{,}23896 \cdot 10^{-9}$

$f(x) = 6{,}23896 \cdot 10^{-9} \cdot x \cdot \log(x)$

Al representar la curva de $f(x)$ ajustada y los datos empíricos, obtenemos:

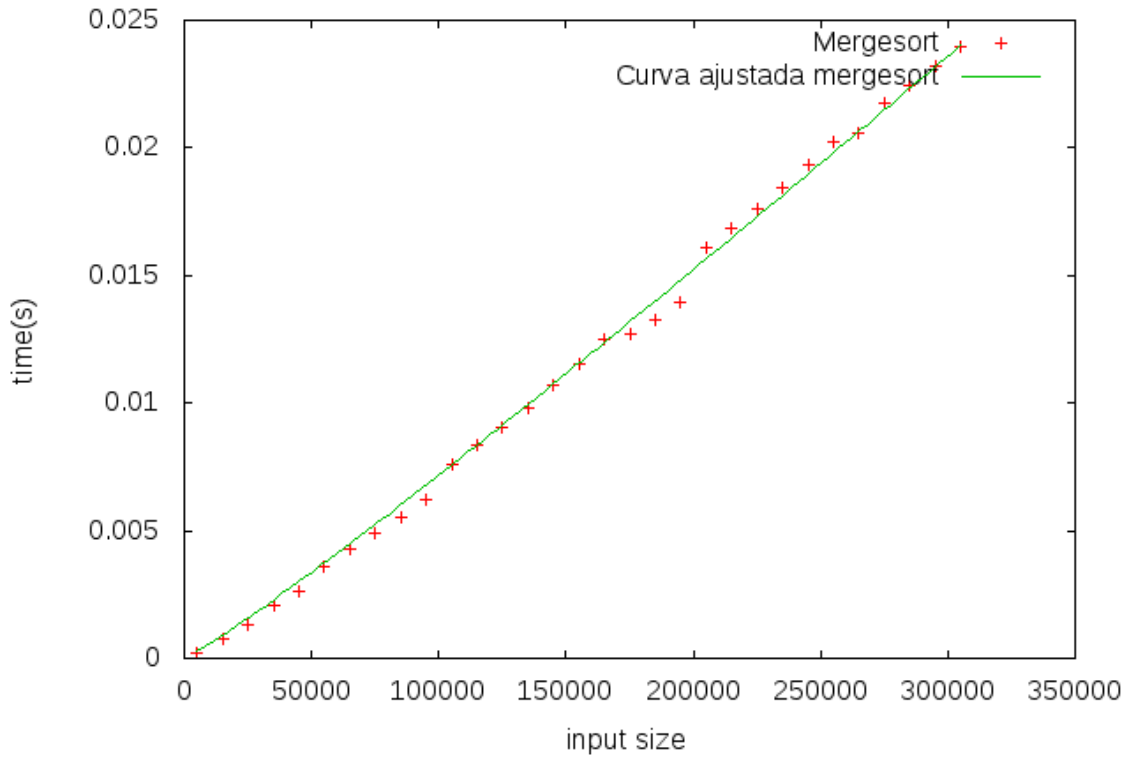

Figura 11: Gráfica del algoritmo *Mergesort* y su función ajustada. Compilación sin optimización. Intel® Core™ i7-5500U CPU @ 2.40GHz.

2.2.3. Algoritmo *Quicksort*.

En la siguiente gráfica podemos observar el crecimiento que presenta el algoritmo *Quicksort*:

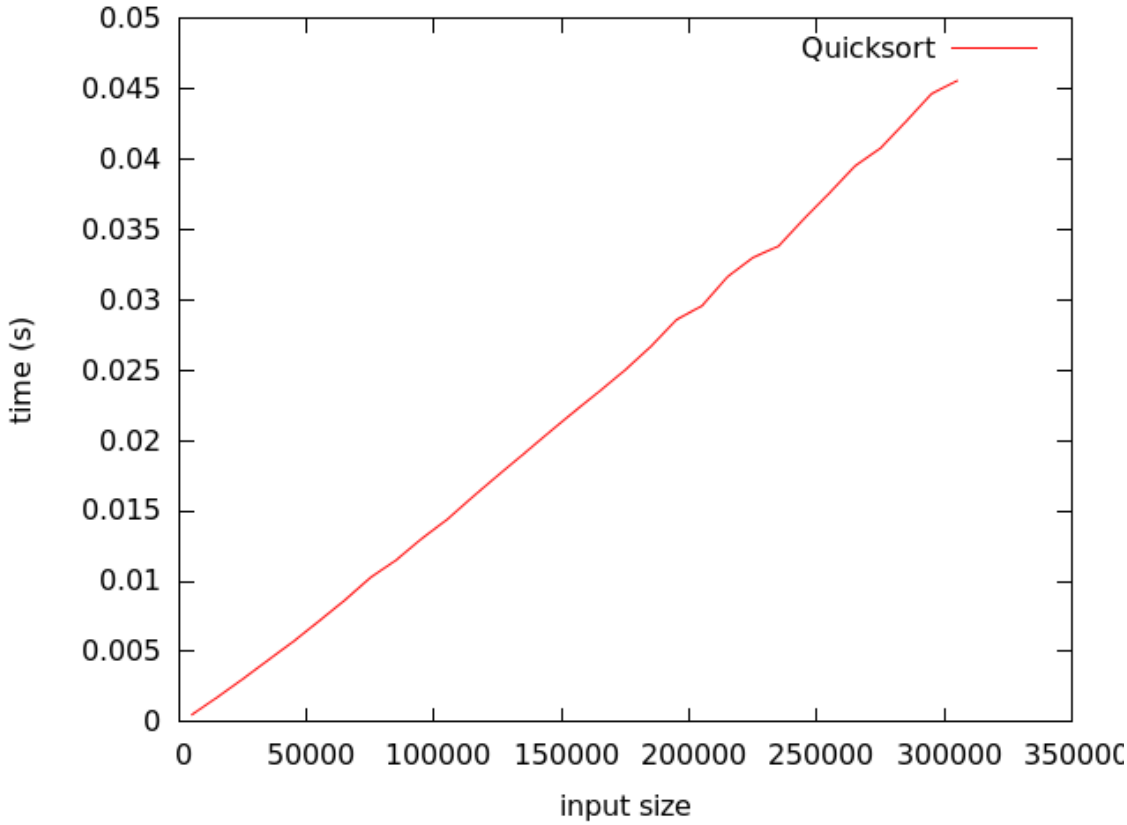

Figura 12: Gráfica del algoritmo *Quicksort*. Compilación sin optimización. Intel® Core™ i7-5500U CPU @ 2.40GHz.

El análisis híbrido de los datos representados en la gráfica produce el siguiente resultado:

- $a = 1{,}18815 \cdot 10^{-8}$

$f(x) = 1{,}18815 \cdot 10^{-8} \cdot x \cdot \log(x)$

En la siguiente gráfica podemos ver el ajuste de $f(x)$ a las sucesivas ejecuciones del algoritmo:

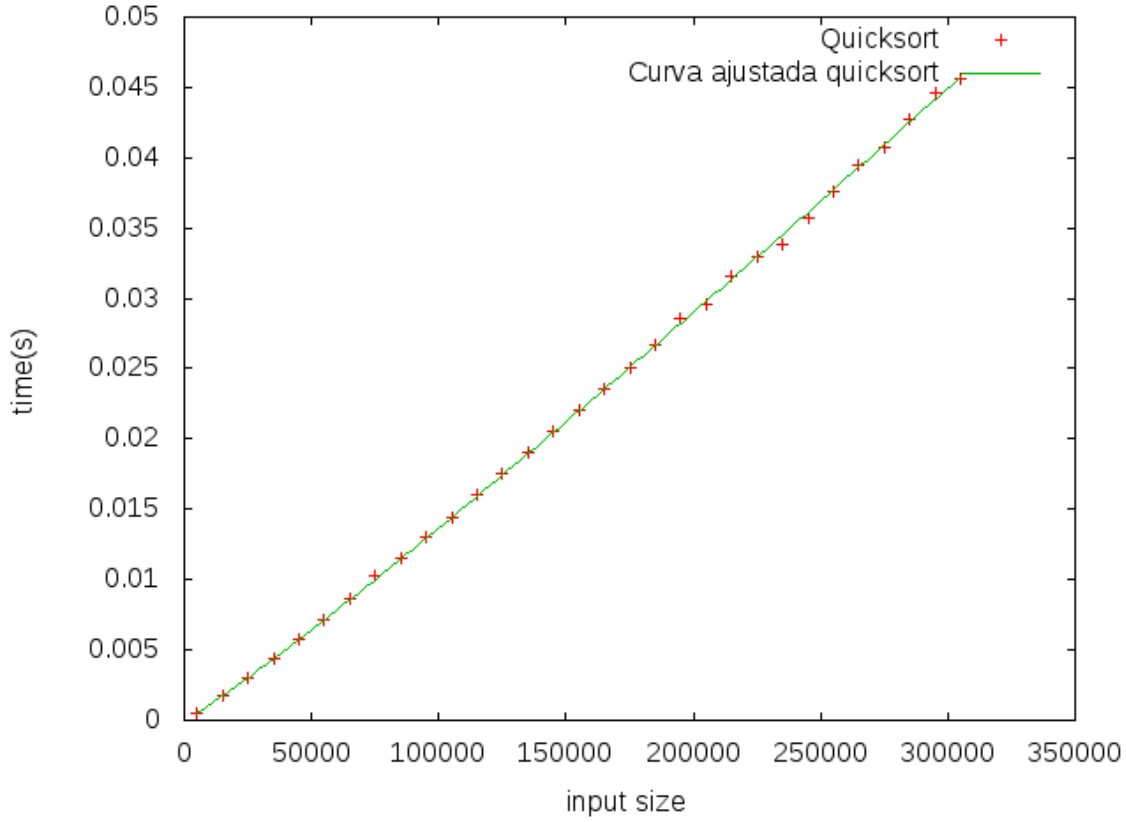

Figura 13: Gráfica del algoritmo *Quicksort* y su función ajustada. Compilación sin optimización. Intel® Core™ i7-5500U CPU @ 2.40GHz.

2.2.4. Comparativa de los algoritmos de O($n \log n$)

En primer lugar, presentamos una gráfica para observar mejor las diferencias de rendimiento entre los distintos algoritmos:

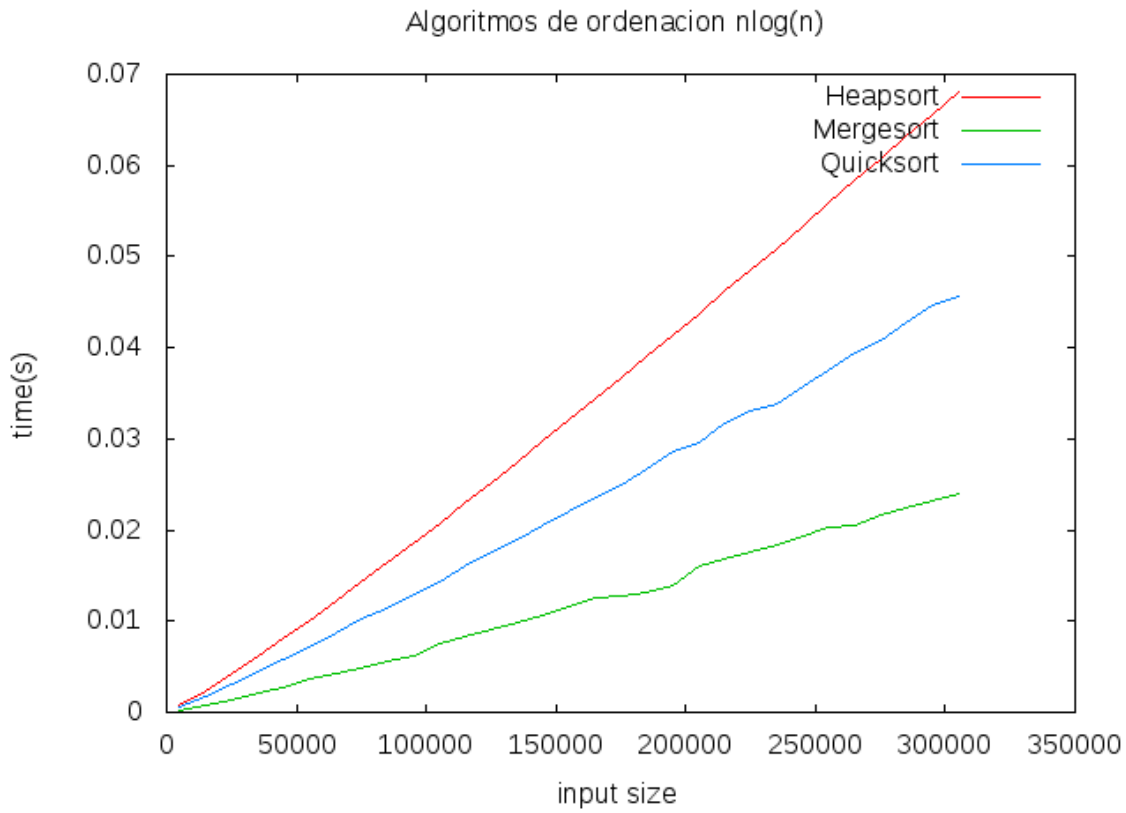

Figura 14: Gráfica comparativa de los algoritmos de ordenación cuadráticos. Compilación sin optimización. Intel® Core™ i7-5500U CPU @ 2.40GHz.

Los datos muestran que, en principio, el algoritmo *Mergesort* es el mejor, seguido de *Quicksort* y *Heapsort*. Sin embargo, los tamaños están ajustados con el objetivo de poder compararlos con los de los algoritmos de ordenación cuadráticos, por lo que son muy pequeños para un orden O($n \log n$) y podrían no ser representativos de la eficiencia en condiciones más realistas.

A continuación, la tabla de tiempos de ejecución:

Tamaño de n	**Tiempo** ***Heapsort*** (s)	**Tiempo** ***Mergesort*** (s)	**Tiempo** ***Quicksort*** (s)
15000	0.002412	0.000780945	0.001771
25000	0.004227	0.00129756	0.003062
35000	0.006153	0.00206483	0.004411
45000	0.008111	0.00264398	0.005754
55000	0.01014	0.00361779	0.007196
65000	0.012223	0.00426566	0.008663
75000	0.014257	0.00490762	0.010267
85000	0.016429	0.00553744	0.011495
95000	0.018575	0.00618519	0.013014
105000	0.020746	0.00762148	0.014378
115000	0.023012	0.00836004	0.015983
125000	0.025217	0.00904335	0.017521
135000	0.027526	0.0097835	0.01904
145000	0.029776	0.0106871	0.020568
155000	0.032133	0.0115184	0.022066
165000	0.034391	0.0125027	0.023528
175000	0.036719	0.0126728	0.02503
185000	0.039012	0.0132283	0.026689
195000	0.04136	0.0139646	0.028577
205000	0.043756	0.0161064	0.029569
215000	0.046311	0.0168623	0.031633
225000	0.048505	0.0176279	0.032997
235000	0.050869	0.018451	0.033803
245000	0.053452	0.0193148	0.035749
255000	0.055863	0.0202648	0.037587
265000	0.058341	0.0205711	0.039503
275000	0.060633	0.0217225	0.040769
285000	0.063236	0.0224305	0.042672
295000	0.065567	0.0231778	0.044629
305000	0.068137	0.0239459	0.04555

2.3. Comparativa general de algoritmos de ordenación.

A continuación, una gráfica que contrasta los tiempos de los seis algoritmos analizados previamente:

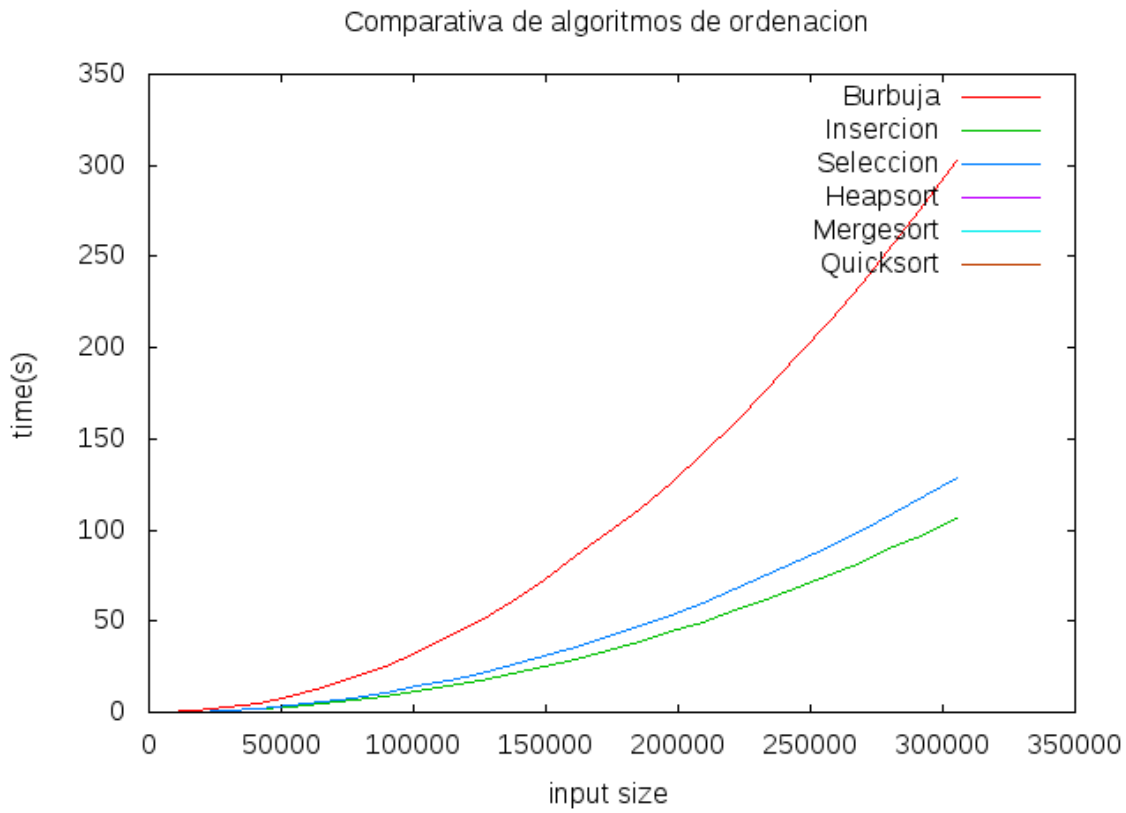

Figura 15: Gráfica comparativa de los algoritmos de ordenación. Compilación sin optimización. Intel® Core™ i7-5500U CPU @ 2.40GHz.

En esta gráfica se puede ver la gran diferencia que hay entre una eficiencia de n^2 y una de $n \log n$, tanta que ni siquiera se aprecian en la gráfica las curvas de los algoritmos *Heapsort*, *Mergesort* y *Quicksort*. Mientras que para tamaños de varios cientos de miles los algoritmos de $O(n^2)$ tardan varios minutos, los de $O(n \log n)$ terminan en cuestión de centésimas de segundo. Esta diferencia es una clara demostración de la importancia de elegir un algoritmo con el menor orden posible (aunque teniendo cuidado con las constantes ocultas), ya que no es sólo cuestión de unos pocos segundos.

3. Algoritmo de Floyd.

3.1. Descripción breve.

El algoritmo de Floyd se utiliza para encontrar el camino más corto entre cada par de nodos en un grafo dirigido. Su orden es O(n^3).

3.2. Tabla de tiempos de ejecución.

Los resultados para n en el rango [100,1550] son:

Tamaño de n	**Tiempo (s)**
100	0.006108
150	0.020072
200	0.047014
250	0.091181
300	0.156999
350	0.248901
400	0.370784
450	0.528139
500	0.722818
550	0.961853
600	1.24689
650	1.58944
700	1.99242
750	2.431558
800	2.96925
850	3.68909
900	4.24717
950	5.02479
1000	5.88652
1050	6.81454
1100	7.88513
1150	9.0024
1200	10.247
1250	11.5879
1300	12.99
1350	14.5377
1400	16.2152
1450	18.0179
1500	19.9648
1550	21.9085

3.3. Gráfica del algoritmo.

Los tiempos de ejecución representados en forma de curva son:

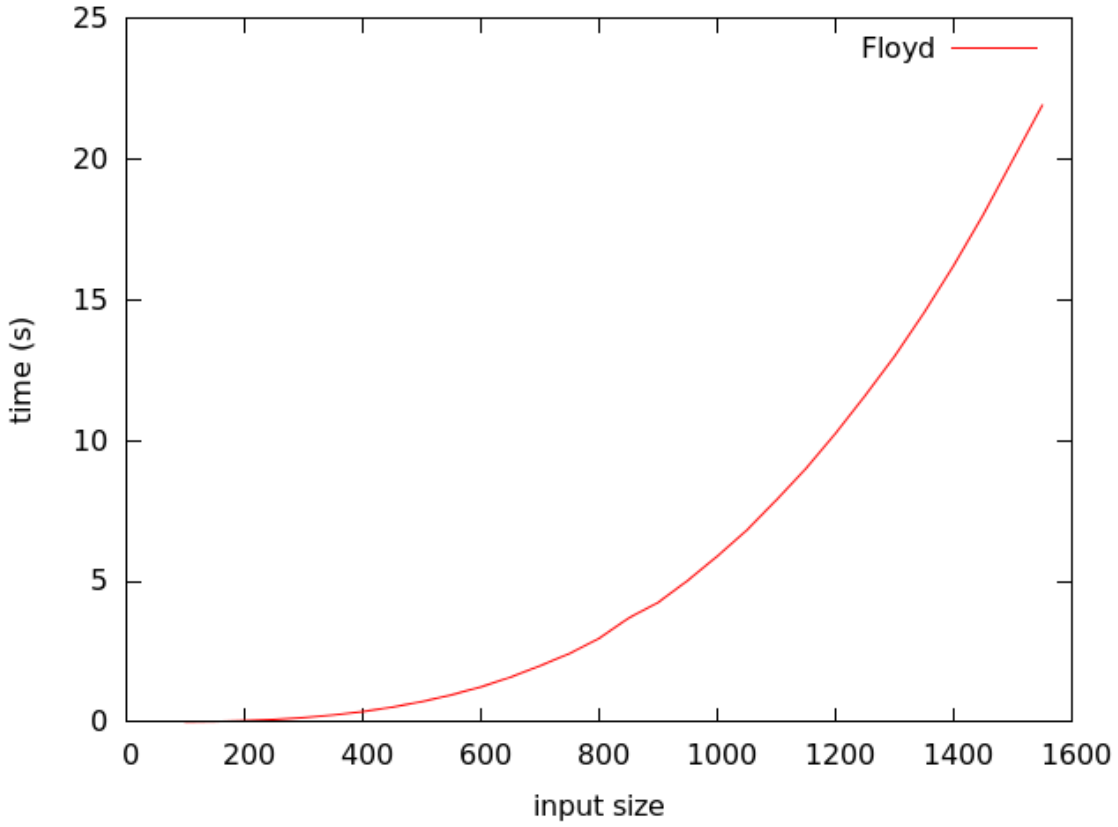

Figura 16: Gráfica del algoritmo de Floyd. Compilación sin optimización. Intel® Core™ i7-5500U CPU @ 2.40GHz.

3.4. Análisis híbrido.

Procedemos a ajustar los datos obtenidos a la siguiente función: $ax^3 + bx^2 + cx + d$.

- $a = 5{,}69357 \cdot 10^{-9}$
- $b = 5{,}61065 \cdot 10^{-7}$
- $c = -0{,}000409355$
- $d = 0{,}0612349$

$f(x) = 5{,}69357 \cdot 10^{-9} \cdot x^3 + 5{,}61065 \cdot 10^{-7} \cdot x^2 - 0{,}000409355 \cdot x + 0{,}0612349$

La representación gráfica del análisis híbrido es la siguiente:

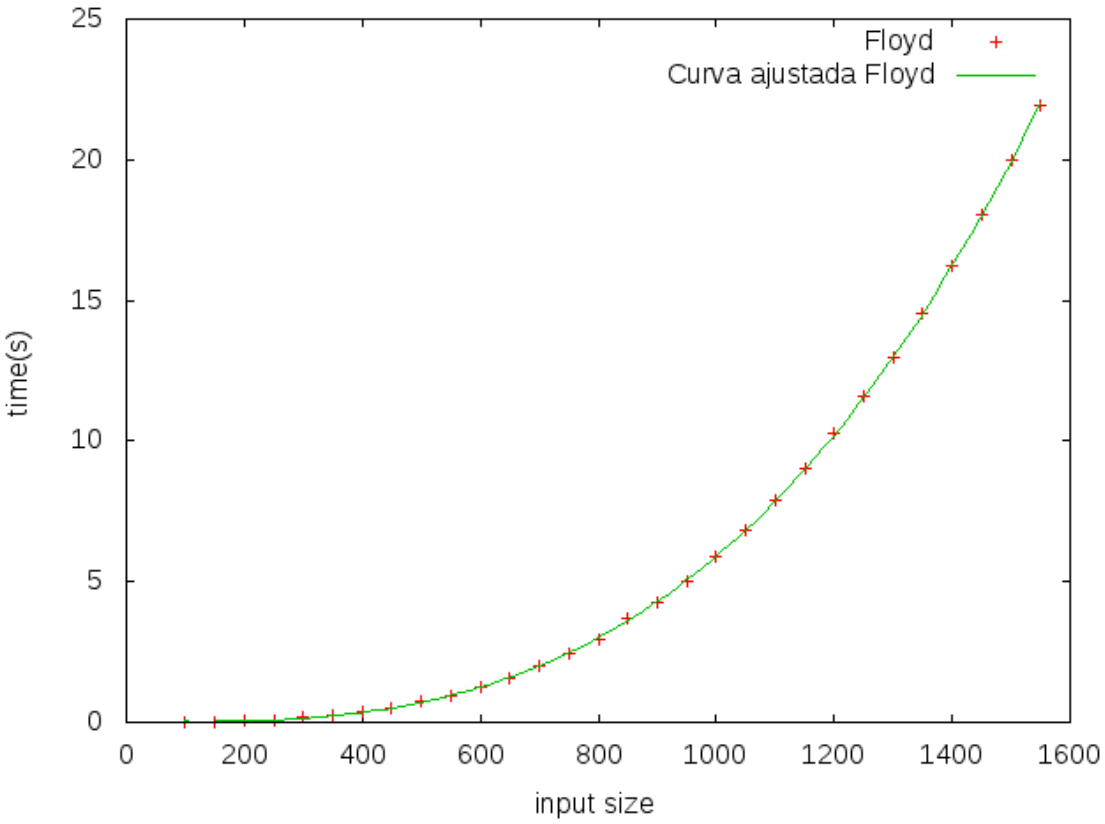

Figura 17: Gráfica del algoritmo de Floyd y su función ajustada. Compilación sin optimización. Intel® Core™ i7-5500U CPU @ 2.40GHz.

4. Algoritmo de las torres de Hanoi.

4.1. Descripción breve.

El problema de las torres de Hanoi consiste en mover una serie de discos de radio creciente desde una estaca a otra de un tablero con ayuda de una tercera. Los movimientos se deben hacer teniendo en cuenta que ningún disco puede estar encima de otro de menor tamaño, que sólo se puede mover uno a la vez y que sólo se puede mover el que esté encima de los demás en su correspondiente estaca. El orden de este algoritmo es $O(2^n)$.

4.2. Tabla de tiempos de ejecución.

A continuación se muestran los resultados para n en el rango [8,33]:

Tamaño de n	**Tiempo (s)**
8	0.000002
9	0.000004
10	0.000007
11	0.000014
12	0.000027
13	0.000053
14	0.000105
15	0.000223
16	0.000435
17	0.001049
18	0.001676
19	0.00336
20	0.006766
21	0.013481
22	0.026936
23	0.053744
24	0.107664
25	0.215053
26	0.429739
27	0.859611
28	1.71886
29	3.43784
30	6.88029
31	13.7576
32	27.6634
33	55.0232

4.3. Gráfica del algoritmo.

La siguiente curva corresponde a la ejecución sucesiva del algoritmo de las torres de Hanoi:

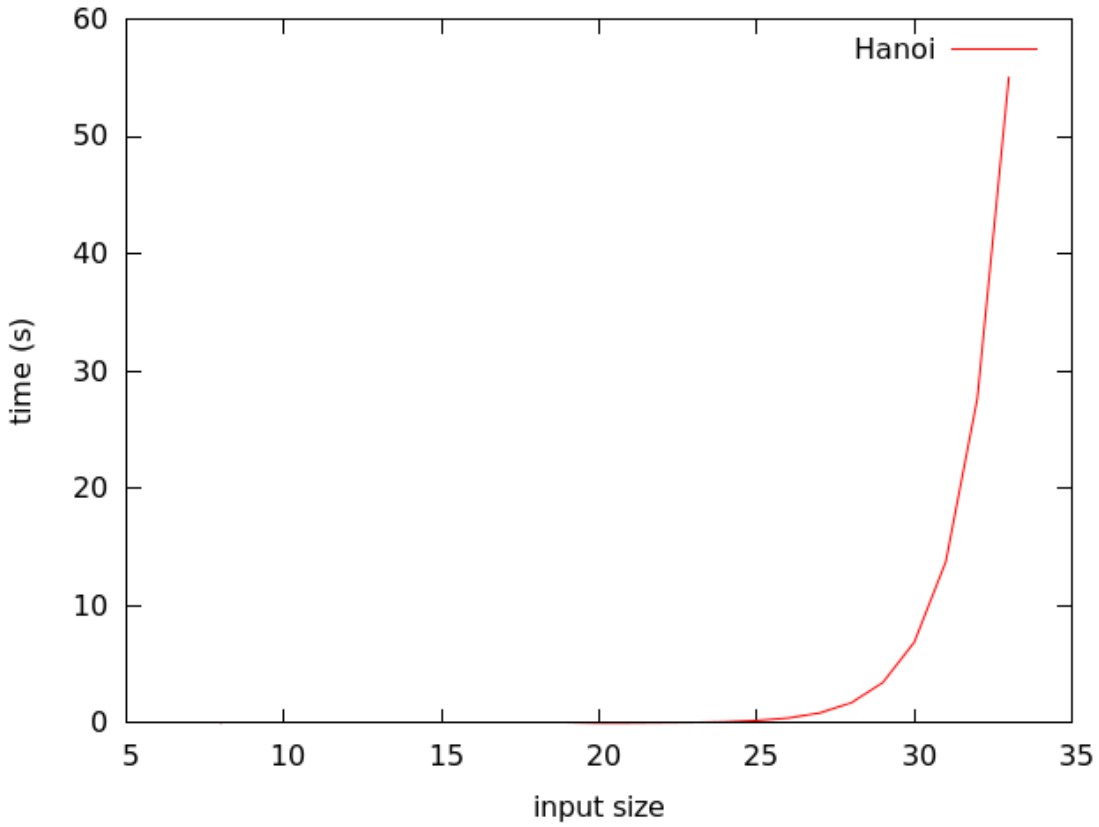

Figura 18: Gráfica del algoritmo de las torres de Hanoi. Compilación sin optimización. Intel® Core™ i7-5500U CPU @ 2.40GHz.

Este algoritmo, al ser de orden $O(2^n)$, resulta tremendamente ineficiente. En la gráfica vemos que para valores muy pequeños ni se aprecia el tiempo que tarda, pero a partir de tan sólo un tamaño de 25, la curva empieza a experimentar un crecimiento enorme. Con un tamaño de únicamente 40 (que es un valor muy pequeño en los otros algoritmos), el algoritmo de Hanoi tardaría unas 2 horas; para un tamaño de 50 tendríamos que esperar unos 3 meses para ver a nuestro algoritmo terminar.

4.4. Análisis híbrido.

En esta sección vamos a ajustar los datos obtenidos a la siguiente función: $a \cdot b^x$.

- $a = 6{,}76475 \cdot 10^{-9}$
- $b = 1{,}99673$

$f(x) = 6{,}76475 \cdot 10^{-9} \cdot 1{,}99673^x$

Al representar los tiempos de ejecución y la función ajustada a ellos, obtenemos:

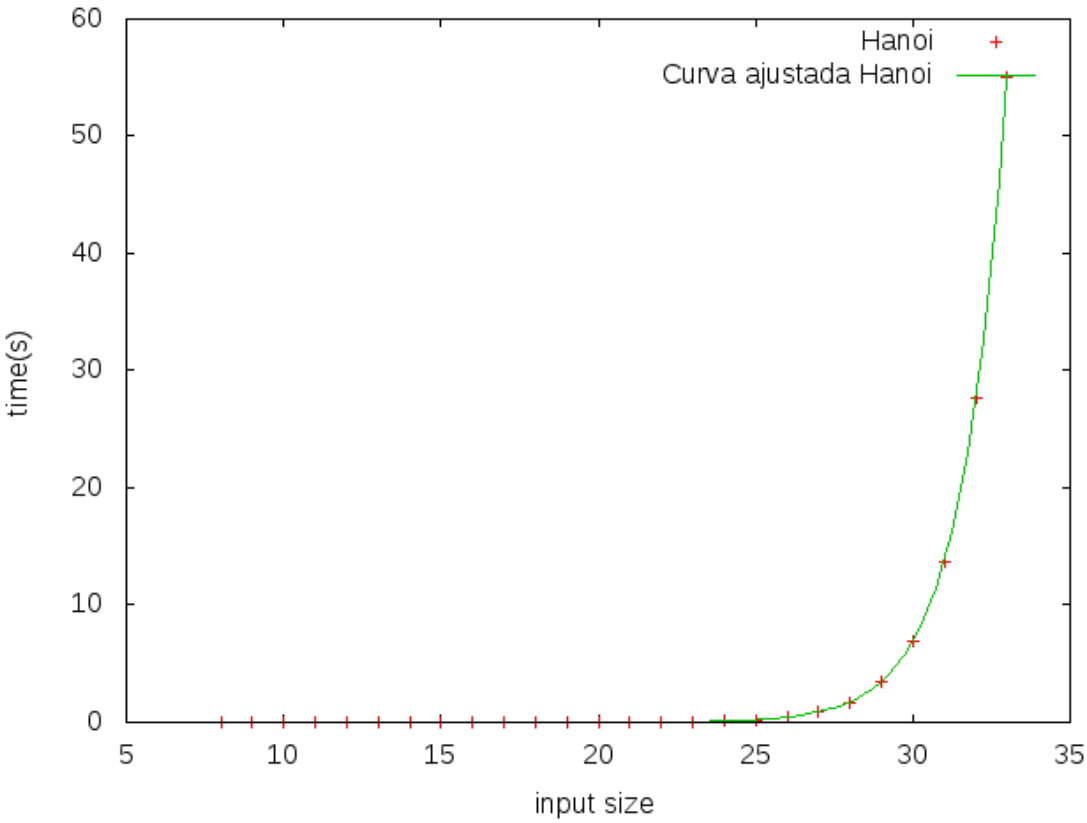

Figura 19: Gráfica del algoritmo de las torres de Hanoi y su función ajustada. Compilación sin optimización. Intel® Core™ i7-5500U CPU @ 2.40GHz.

5. Análisis de parámetros externos.

5.1. Nivel de optimización al compilar.

En esta sección estudiaremos el efecto que tienen distintas optimizaciones al compilar en los diferentes tipos de algoritmos.

5.1.1. Algoritmos de orden O(n^2).

Para realizar este análisis, tomaremos el algoritmo de Selección como representante de los algoritmos cuadráticos.

En la siguiente gráfica podemos observar las diferencias en el tiempo de ejecución, que son bastante notables en los casos de optimización de nivel 1 y 2; por su parte, el nivel 3 no sólo no aporta ninguna mejora sino que pierde casi la mitad de efectividad.

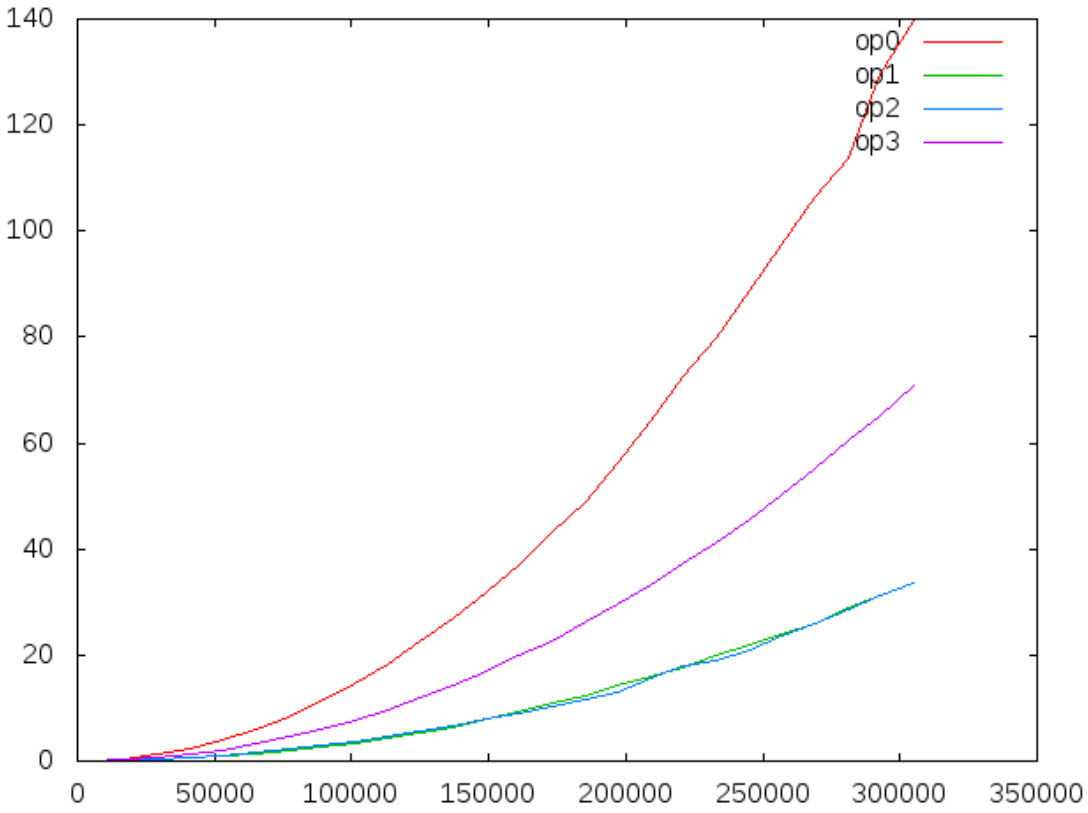

Figura 20: Gráfica del algoritmo de Selección con distintas opciones de compilación. Intel® Core™ i5-2320 CPU @ 3.00GHz.

Para mayor claridad, adjuntamos la tabla de tiempos:

Tamaño de n	**Nivel 0**	**Nivel 1**	**Nivel 2**	**Nivel 3**
5000	0.036124	0.009041	0.00938	0.019137
17000	0.414943	0.09991	0.11193	0.21985
29000	1.20676	0.287832	0.312588	0.64235
41000	2.40054	0.569582	0.620342	1.28279
53000	4.01078	0.948723	1.04029	2.14851
65000	6.03459	1.43653	1.5454	3.20979
77000	8.46338	2.02502	2.19426	4.50478
89000	11.3048	2.68759	2.84887	6.02035
101000	14.5591	3.47514	3.65536	7.7515
113000	18.2305	4.32426	4.66081	9.8025
125000	22.5115	5.25863	5.53469	11.9881
137000	27.0404	6.35793	6.71818	14.3984
149000	31.9853	7.85416	7.97568	17.0428
161000	37.0007	9.25359	9.12481	19.8977
173000	43.0687	10.927	10.3218	22.7449
185000	48.9193	12.2514	11.5484	26.1665
197000	56.0373	14.3025	13.0519	29.4895
209000	64.295	16.0266	15.6726	33.365
221000	72.4991	17.7103	17.9539	37.1059
233000	79.7657	19.8308	18.9754	41.2455
245000	88.8895	21.9423	20.8413	45.6001
257000	97.5091	24.0771	23.7747	50.1777
269000	106.297	25.9644	25.8485	55.3627
281000	113.799	28.8626	28.6332	60.4656
293000	129.573	31.3742	31.4038	65.2064
305000	139.702	33.5263	33.49	70.7969

5.1.2. Algoritmos de orden O($n \log n$).

En este caso, el algoritmo *Quicksort* servirá como caso de estudio para los algoritmos de orden O($n \log n$).

Mediante la gráfica podemos comprobar que la optimización a distintos niveles puede ser beneficiosa pero no hay una gran diferencia de rendimiento. En particular, la optimización de nivel 1 parece ser peor que la de nivel 0 según los datos obtenidos.

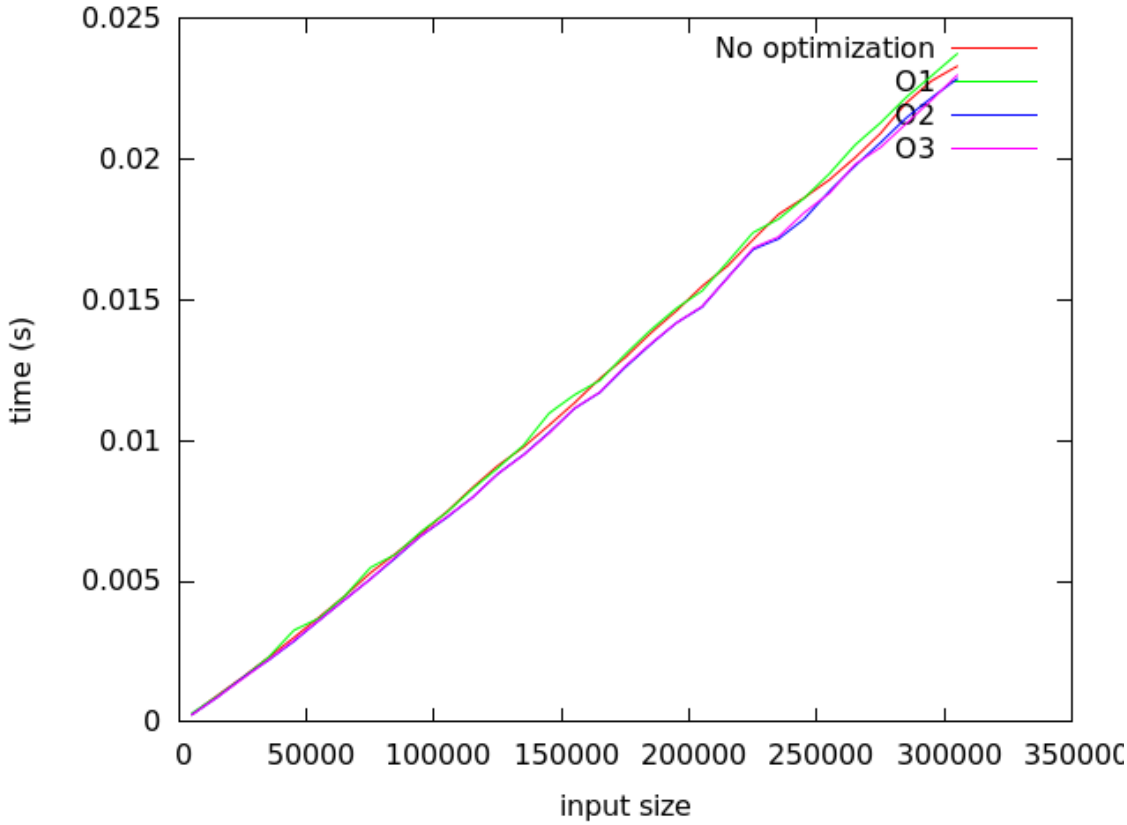

Figura 21: Gráfica del algoritmo *Quicksort* con distintas opciones de compilación. Intel® Core™ i7-5500U CPU @ 2.40GHz.

Para mayor claridad, adjuntamos la tabla de tiempos:

Tamaño de n	**Nivel 0**	**Nivel 1**	**Nivel 2**	**Nivel 3**
15000	0.001771	0.000928	0.000882	0.000882
25000	0.003062	0.001575	0.001576	0.001547
35000	0.004411	0.002297	0.002194	0.002204
45000	0.005754	0.003259	0.002868	0.0029
55000	0.007196	0.003694	0.003621	0.003622
65000	0.008663	0.004483	0.004355	0.004354
75000	0.010267	0.00549	0.005083	0.005079
85000	0.011495	0.005963	0.005843	0.005879
95000	0.013014	0.006773	0.006628	0.00663
105000	0.014378	0.00745	0.007269	0.007295
115000	0.015983	0.008277	0.007978	0.007971
125000	0.017521	0.009028	0.00882	0.008821
135000	0.01904	0.009832	0.009495	0.00948
145000	0.020568	0.010966	0.010287	0.010253
155000	0.022066	0.011623	0.011137	0.011144
165000	0.023528	0.012134	0.011709	0.011721
175000	0.02503	0.01307	0.012608	0.01265
185000	0.026689	0.013931	0.013422	0.013448
195000	0.028577	0.014698	0.014178	0.01418
205000	0.029569	0.01531	0.014735	0.014761
215000	0.031633	0.01634	0.015788	0.01577
225000	0.032997	0.017385	0.016784	0.016836
235000	0.033803	0.017863	0.017162	0.017235
245000	0.035749	0.018605	0.01786	0.01809
255000	0.037587	0.019483	0.01886	0.018783
265000	0.039503	0.020494	0.01975	0.019807
275000	0.040769	0.021284	0.020579	0.020408
285000	0.042672	0.022184	0.021453	0.021249
295000	0.044629	0.022953	0.022175	0.022093
305000	0.045557	0.023731	0.022839	0.022972

5.1.3. Algoritmo de Floyd.

Hemos hecho mediciones de tiempo con las cuatro optimizaciones para el algoritmo de Floyd, obteniendo los siguientes resultados:

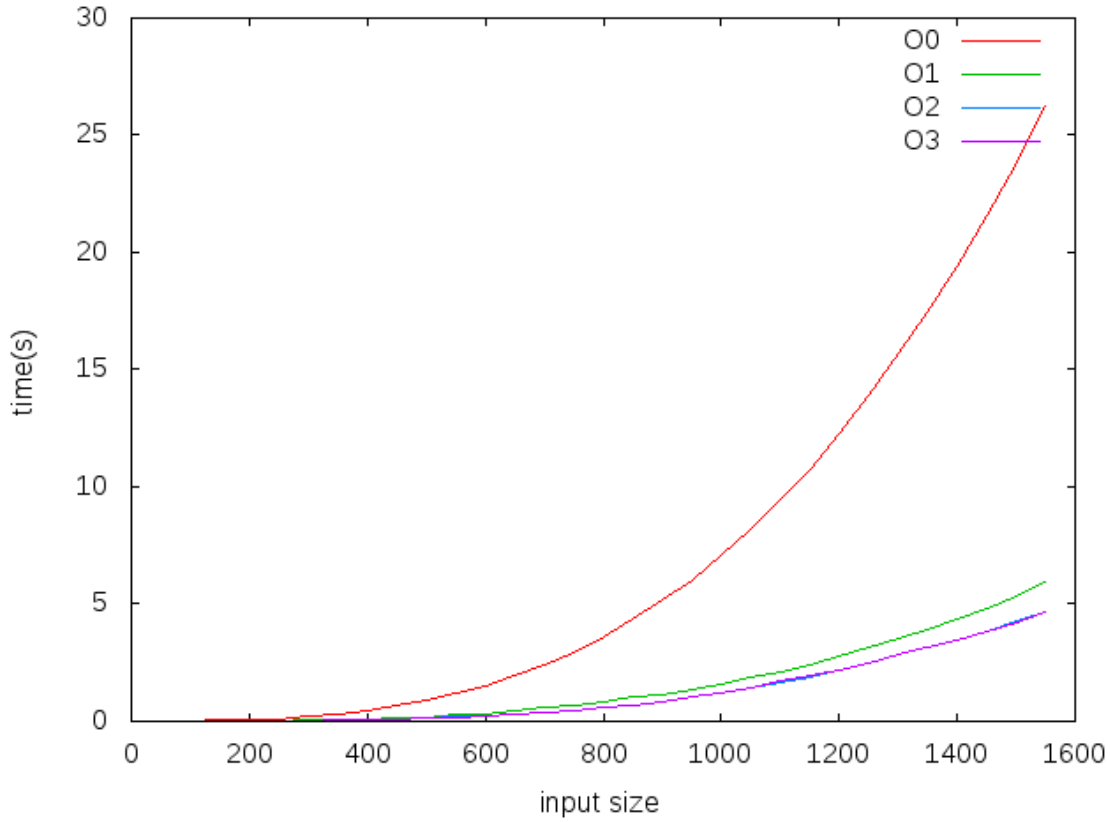

Figura 22: Gráfica del algoritmo de Floyd con distintas opciones de compilación. Intel® Core™ i5-4200U CPU @ 1.60GHz.

Podemos observar que la mejora de los niveles 1, 2 y 3 respecto al nivel 0 de optimización es más que notable, ya que reduce en más de dos tercios el tiempo de ejecución del algoritmo. Los niveles 2 y 3 de optimización apenas difieren y, aunque sí que suponen una mejora respecto al nivel 1, no es tan pronunciada.

Para mayor claridad, adjuntamos la tabla de tiempos:

Tamaño de n	**Nivel 0**	**Nivel 1**	**Nivel 2**	**Nivel 3**
100	0.019403	0.004196	0.003485	0.004847
150	0.041889	0.007138	0.009218	0.011776
200	0.057889	0.012596	0.016861	0.020785
250	0.111783	0.024436	0.018309	0.03239
300	0.18856	0.039956	0.031289	0.034652
350	0.300613	0.063244	0.052344	0.04781
400	0.452591	0.093873	0.071509	0.07209
450	0.676688	0.133829	0.101763	0.103639
500	0.881619	0.183502	0.147238	0.138195
550	1.23415	0.27344	0.189409	0.184614
600	1.4987	0.314622	0.242244	0.23895
650	1.93165	0.425116	0.309741	0.310549
700	2.38591	0.566917	0.380176	0.38189
750	2.92394	0.713505	0.473913	0.465974
800	3.60982	0.792758	0.583663	0.578305
850	4.3496	1.03309	0.705663	0.712704
900	5.19879	1.12136	0.853371	0.859039
950	6.01972	1.33587	1.0296	1.03618
1000	7.07046	1.58757	1.22454	1.22418
1050	8.19999	1.85009	1.43525	1.4529
1100	9.43852	2.13969	1.67268	1.7051
1150	10.7766	2.44081	1.91303	1.93099
1200	12.2693	2.79433	2.18412	2.20683
1250	13.8743	3.12842	2.50924	2.50058
1300	15.6962	3.51319	2.82064	2.82453
1350	17.4077	3.94437	3.16143	3.19427
1400	19.3832	4.38754	3.4905	3.45987
1450	21.5901	4.84861	3.85593	3.83144
1500	23.8717	5.36099	4.28894	4.23618
1550	26.2314	5.91859	4.6492	4.66486

5.1.4. Algoritmo de las torres de Hanoi.

Los resultados obtenidos con los diferentes niveles de optimización para el algoritmo de las torres de Hanoi pueden verse en la siguiente gráfica:

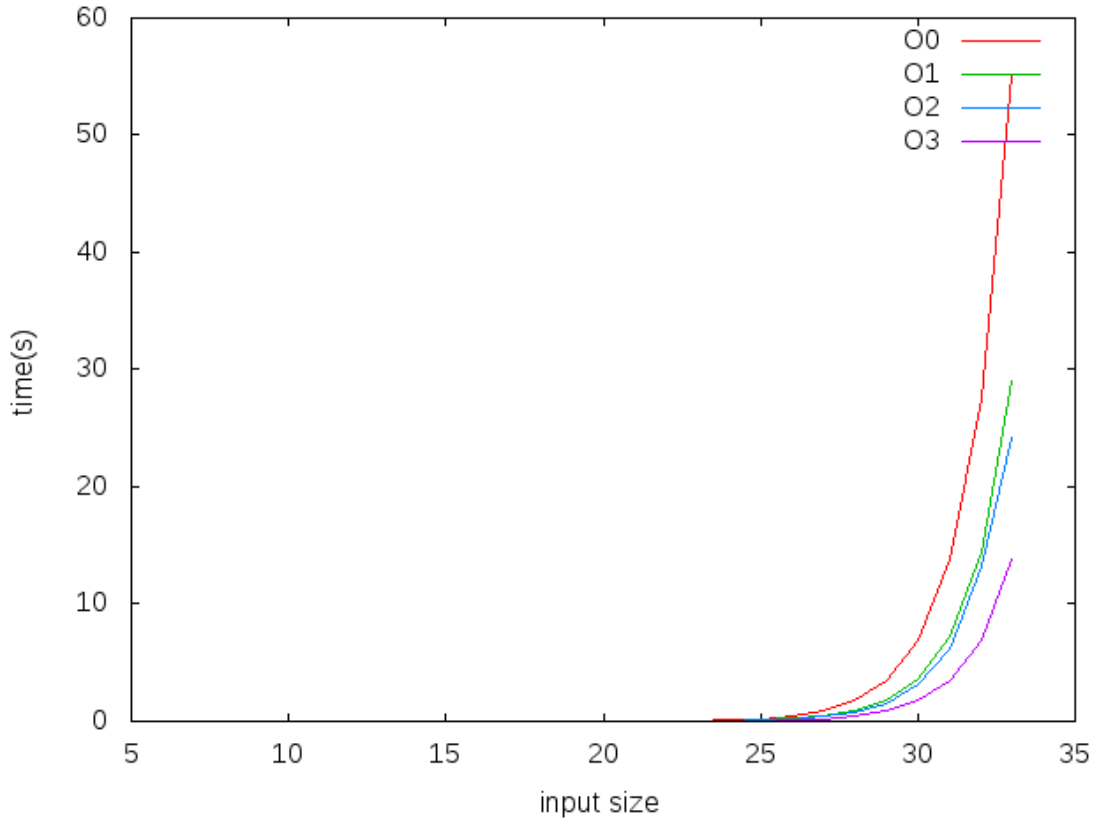

Figura 23: Gráfica del algoritmo de las torres de Hanoi con distintas opciones de compilación. Intel® Core™ i7-5500U CPU @ 2.40GHz.

Como podemos comprobar, el nivel 1 de optimización reduce casi a la mitad el tiempo de ejecución respecto al nivel 0, lo cual es una mejora considerable si tenemos en cuenta que el algoritmo tiene un orden exponencial. El nivel 2 es ligeramente mejor que el 1, pero sin duda el que mejores tiempos obtiene es el nivel 3 de optimización, que vuelve a reducir a la mitad el tiempo respecto al nivel 1, obteniendo así casi cuatro veces mejores resultados que el nivel 0.

Para mayor claridad, adjuntamos la tabla de tiempos:

Tamaño de n	**Nivel 0**	**Nivel 1**	**Nivel 2**	**Nivel 3**
8	0.00001	0.000002	0.000002	0.000002
9	0.00001	0.000004	0.000002	0.000003
10	0.000013	0.000005	0.000004	0.000004
11	0.000018	0.000008	0.000007	0.000007
12	0.000032	0.000015	0.000013	0.000011
13	0.000056	0.00003	0.000024	0.000018
14	0.000141	0.000085	0.000048	0.000031
15	0.000218	0.000124	0.000093	0.000085
16	0.000574	0.000228	0.000187	0.000114
17	0.00092	0.000453	0.000374	0.000222
18	0.001742	0.000907	0.000822	0.000447
19	0.003557	0.001873	0.001848	0.000903
20	0.007103	0.003635	0.003209	0.002319
21	0.014194	0.00723	0.006186	0.00472
22	0.027314	0.014383	0.012254	0.009848
23	0.054407	0.028384	0.024428	0.015368
24	0.108365	0.058189	0.048759	0.029236
25	0.216321	0.113425	0.095812	0.057161
26	0.434826	0.233242	0.191445	0.114506
27	0.862026	0.454714	0.385982	0.224239
28	1.74946	0.92143	0.802508	0.449479
29	3.47242	1.8215	1.53576	0.877691
30	6.91911	3.66455	3.08881	1.77166
31	13.812	7.22065	6.14723	3.5071
32	27.5176	14.4925	13.1772	6.93418
33	55.1899	29.0435	24.2387	13.81

5.2. Hardware utilizado.

En esta sección vamos a ver cómo puede afectar el hardware a los tiempos de ejecución de los algoritmos.

5.2.1. Mismo algoritmo en distintos procesadores.

La ejecución de un mismo algoritmo con distintos procesadores puede variar significativamente en términos de tiempo. Por ello, hemos preparado una gráfica que muestra una comparativa entre modelos de tres gamas diferentes de procesadores:

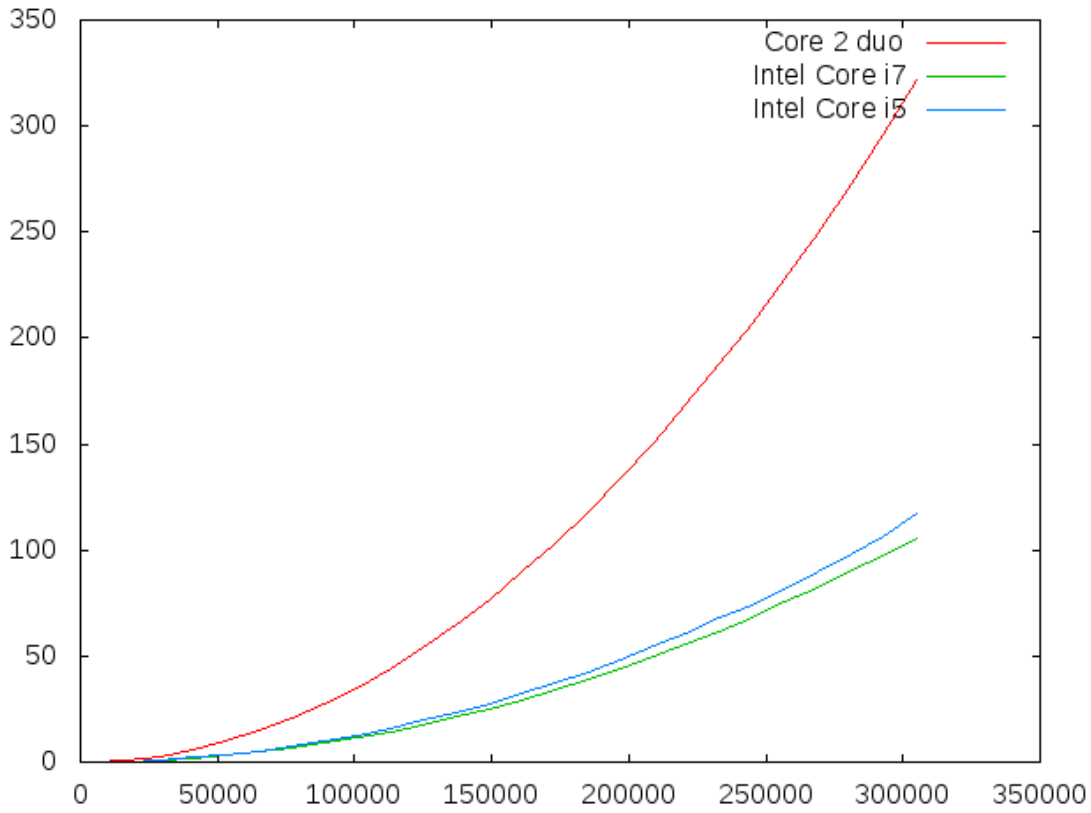

Figura 24: Gráfica con tiempos de ejecución del algoritmo de Inserción en tres procesadores. Intel® Core™ i7-5500U CPU @ 2.40GHz, Intel® Core™ i5-2320 CPU @ 3.00GHz, Intel® Core™ 2 Duo T2350 @ 1.86GHz.

Como se puede observar, los tiempos varían de forma nada despreciable; es claro, sobre todo, el salto de velocidad entre el Intel Core 2 Duo y los otros dos debido a la distancia temporal y tecnológica entre ellos. Sin embargo, lo único que conseguimos es cambiar las constantes ocultas, por lo que generalmente es preferible buscar un algoritmo con un orden un poco mejor a costa de usar un hardware menos potente.

5.2.2. Algoritmos del mismo orden en distintos procesadores.

Por otra parte, queremos observar qué ocurre si ejecutamos un algoritmo lento de un determinado orden de eficiencia en un procesador potente y un algoritmo rápido de ese mismo orden en un procesador poco potente. Como muestras tomaremos los algoritmos *Burbuja* (lento) y de Inserción (rápido), correspondientes a un $O(n^2)$:

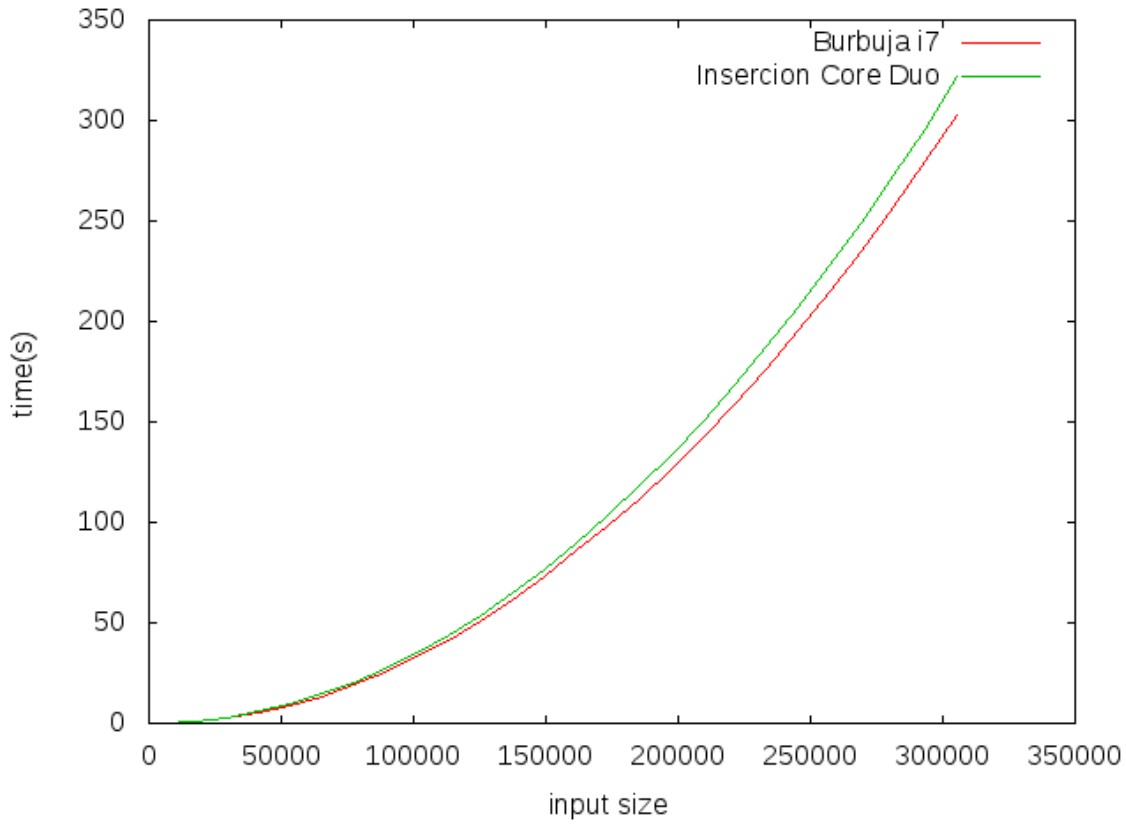

Figura 25: Gráfica con tiempos de ejecución de los algoritmos *Burbuja* e Inserción. Intel® Core™ i7-5500U CPU @ 2.40GHz, Intel® Core™ 2 Duo T2350 @ 1.86GHz.

En la comparativa de algoritmos de orden $O(n^2)$ veíamos que, bajo las mismas condiciones, el algoritmo *Burbuja* es claramente más ineficiente que el de Inserción. Sin embargo, empleando hardware de muy distintas capacidades, podemos conseguir que el algoritmo *Burbuja* gane al de Inserción; no obstante, entre la salida al mercado de cada uno de los dos procesadores comparados hay aproximadamente 10 años, lo cual dice mucho nuevamente acerca de la importancia de la eficiencia, incluso dentro del mismo orden, a la hora de tomar una decisión.

Para mayor claridad, adjuntamos la tabla de tiempos:

Tamaño de n	**Core 2 Duo** (s)	**Core i5** (s)	**Core i7** (s)
5000	0.092425	0.031133	0.02864
17000	0.994135	0.364622	0.331283
29000	2.90596	1.04985	0.95447
41000	5.819	2.10168	1.98294
53000	9.72427	3.49405	3.33574
65000	14.6285	5.24682	5.05818
77000	20.5247	7.36656	6.77307
89000	27.3593	9.88673	9.02627
101000	35.2544	12.5851	11.6468
113000	44.1562	15.7986	14.5133
125000	53.8809	19.978	17.7904
137000	64.7697	23.2261	21.4265
149000	76.4518	27.5321	25.2194
161000	89.8139	32.2	29.4826
173000	103.434	37.4744	34.036
185000	118.276	42.5004	38.9165
197000	133.757	48.0315	44.2132
209000	150.313	54.6543	49.7696
221000	168.566	60.7121	55.6643
233000	187.793	68.1661	61.8095
245000	206.859	74.2389	68.3384
257000	227.995	81.7298	75.2418
269000	249.363	89.945	82.2099
281000	272.411	98.2273	90.579
293000	295.85	106.383	97.4804
305000	322.121	117.112	105.925

5.3. Sistema operativo.

Al igual que con distintas optimizaciones o distinto hardware, podríamos pensar que diferentes sistemas operativos harán nuestros algoritmos más lentos o más rápidos. En este apartado tratamos de dar respuesta a esa cuestión probando el algoritmo de Inserción en dos sistemas operativos.

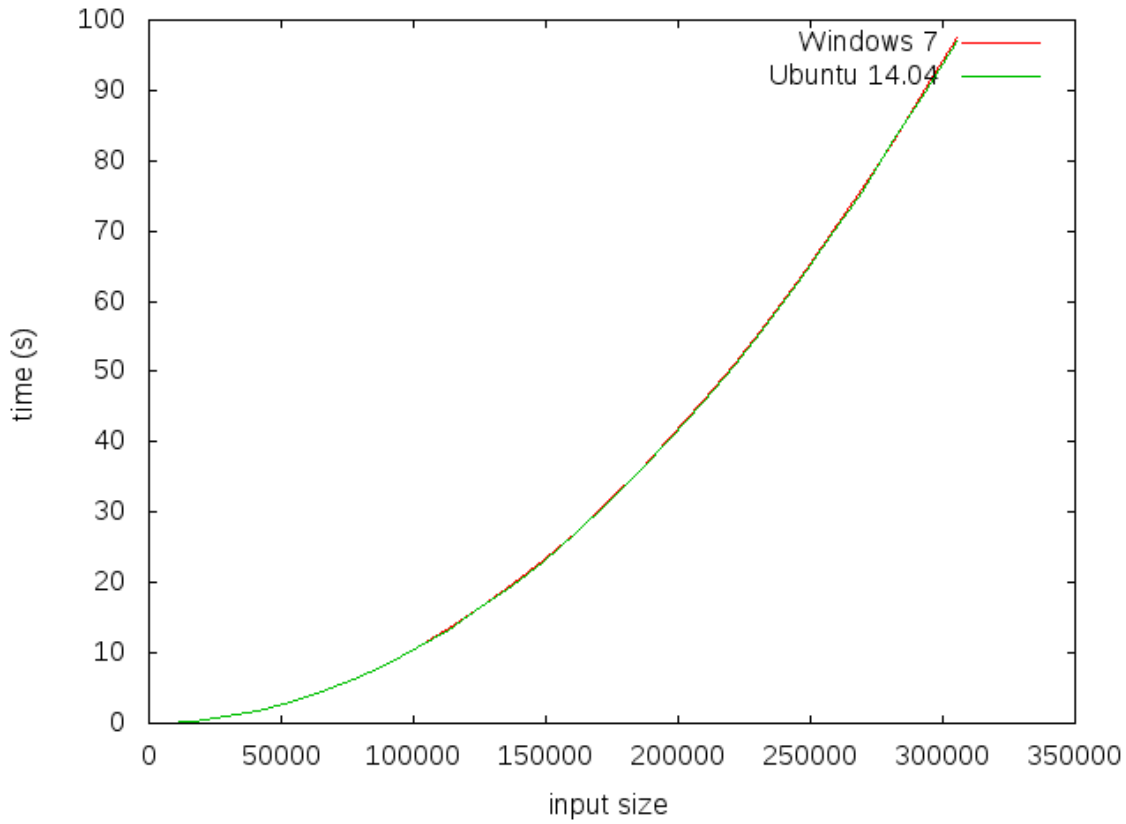

Figura 26: Gráfica con tiempos de ejecución del algoritmo de Inserción en dos sistemas operativos distintos: Ubuntu 14.04 y Windows 7. Intel® Core™ i5-3470 CPU @ 3.20GHz.

De aquí se extrae que, al menos en esta prueba, las diferencias observadas no son suficientes para poder afirmar que el rendimiento cambia de un sistema operativo a otro; Windows 7 parece ser ligeramente más lento conforme aumentamos el tamaño de la entrada del algoritmo, pero apenas se nota.

Para mayor claridad, adjuntamos la tabla de tiempos:

Tamaño de n	**Tiempo Ubuntu 14.04** (s)	**Tiempo Windows 7** (s)
5000	0.029967	0.031
17000	0.303031	0.312
29000	0.883463	0.906
41000	1.76988	1.779
53000	2.952	2.948
65000	4.45062	4.448
77000	6.20768	6.24
89000	8.29843	8.348
101000	10.6655	10.697
113000	13.3013	13.437
125000	16.4174	16.417
137000	19.6679	19.721
149000	23.176	23.312
161000	27.1208	27.167
173000	31.271	31.477
185000	35.7505	35.922
197000	40.5161	40.801
209000	45.621	45.794
221000	51.001	51.184
233000	56.6762	56.847
245000	62.722	63.025
257000	69.054	69.375
269000	75.3449	75.949
281000	82.8552	82.769
293000	89.6191	90.176
305000	97.1005	97.467

6. Análisis de eficiencia erróneos.

Antes de concluir, veamos qué ocurre si ajustamos una función errónea a los resultados de la ejecución de un algoritmo. En particular, la siguiente gráfica muestra el intento de ajustar una función $f(x) = n \log n$ a un algoritmo cuadrático como es el *Burbuja*:

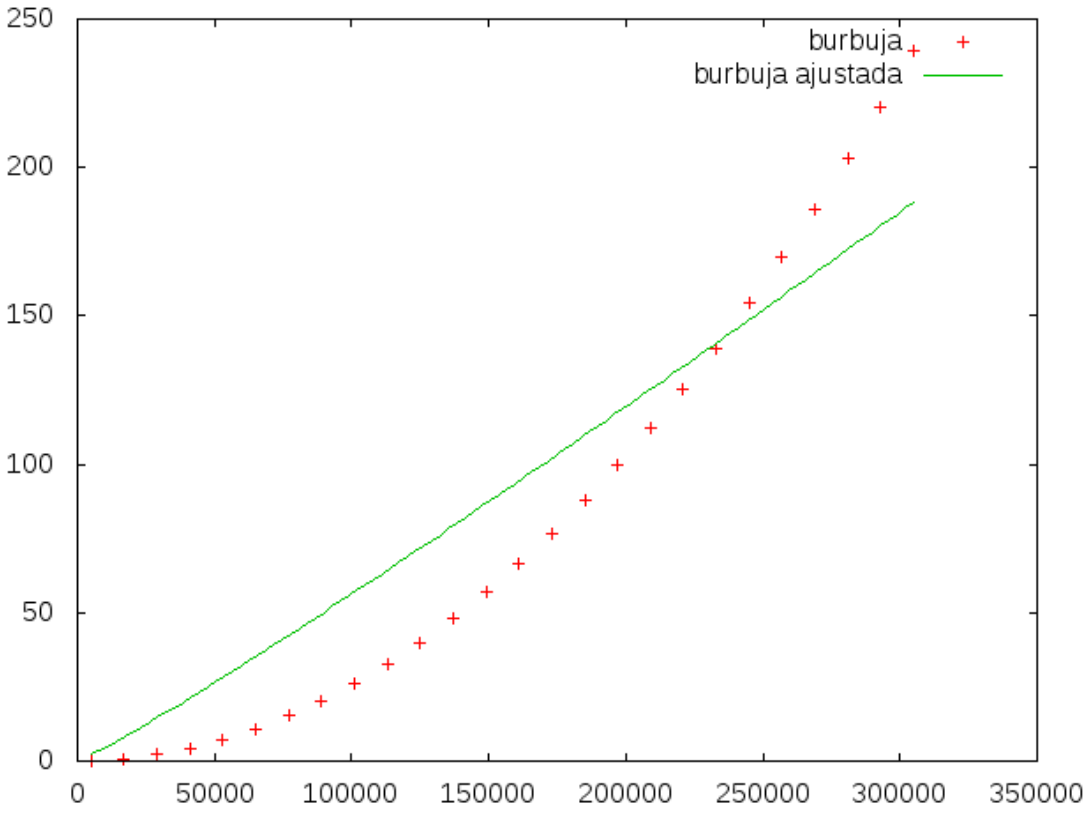

Figura 27: Gráfica del algoritmo *Burbuja* ajustada a $f(x)$. Compilación sin optimización. Intel® Core™ i7-5500U CPU @ 2.40GHz.

Se puede constatar que $f(x)$ difiere de la curva del algoritmo hasta tal punto que no podemos encontrar parecido. Con este ejemplo se aprecia bien la distinción entre órdenes de eficiencia.

7. Conclusión.

Tras ver las diferentes comparaciones de tiempo entre todos estos algoritmos, podemos asegurar que el orden de eficiencia de un algoritmo no lo hará mejor a otro bajo cualquier circunstancia. Incluso puede resultar más eficiente un algoritmo de mayor orden para pequeñas cantidades de datos debido a las constantes ocultas.

Otro de los hechos que hemos podido constatar es la diferencia entre los algoritmos que tienen el mismo orden de eficiencia. Centrándonos solo en la notación asintótica, se podría pensar que los algoritmos de ordenación por *Burbuja*, Inserción o Selección son iguales a efectos de eficiencia; no obstante, si los estudiamos un poco más a fondo teniendo en cuenta sus constantes ocultas, veremos que existen grandes diferencias entre ellos.

Para cerrar, diremos que la elección de los algoritmos basada en la eficiencia de los mismos y en las circunstancias de su uso es fundamental para obtener soluciones adecuadas y competitivas en términos de tiempo y de recursos hardware necesarios; en definitiva, mejores.