Práctica 1: Análisis de Eficiencia de Algoritmos

Adrián Portillo Sánchez
Alejandro Durán Castro
Javier Labrat Rodríguez
Jose Antonio Martínez López
Jose Juán Pérez González

15 de marzo de 2015

1. Introducción

En esta práctica aplicaremos las técnicas de cálculo de eficiencia empírica e híbrida para el análisis de un conjunto de algorítmos determinados por el profesor.
Para cada uno de ellos realizaremos el cálculo de la eficiencia empírica y de la eficiencia híbrida, estos cálculos quedarán reflejados en diversas gráficas y tablas que dejaremos plasmadas en esta memoria.
Comenzaremos por los algoritmos de ordenación típicos burbuja, inserción y selección.

Índice

2. Análisis de Algoritmos de Ordenación con Eficiencia $O(n^2)$

2.1. Características de la máquina utilizada

Hardware

Procesador: Intel Core i5 CPU-M 430 @ 2.27GHz x 4

Memoria: 3,7 GiB

Sistema Operativo: Ubuntu 12.04 LTS 64-bit

2.2. Compilador usado y opciones de compilación

g++ -O2 -o burbuja burbuja.cpp

g++ -O2 -o insercion insercion.cpp

g++ -O2 -o seleccion seleccion.cpp

2.3. Script de ejecución

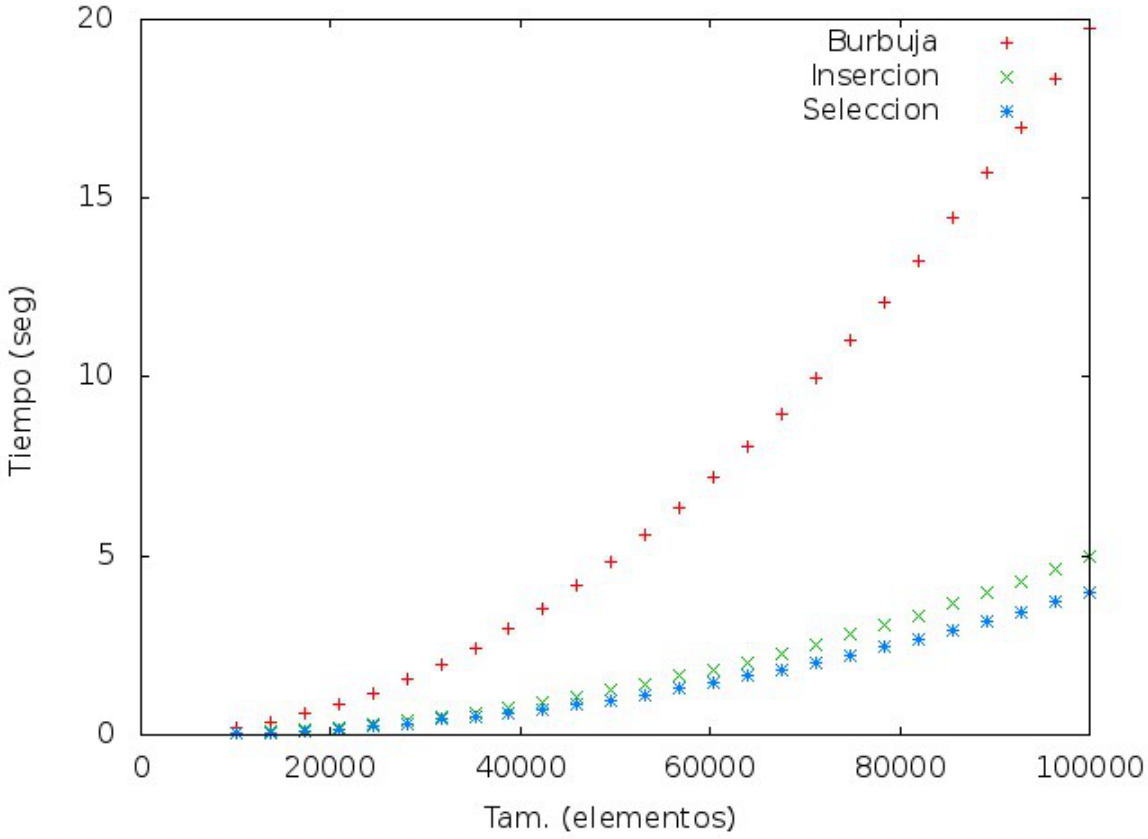
```
#!/bin/csh

set ejecutable = burbuja
@ i = 10000
while ( $i <= 100000 )
./{$ejecutable} $i >> 1_{$ejecutable}.dat
@ i += 3600
end

set ejecutable = insercion
@ i = 10000
while ( $i <= 100000 )
./{$ejecutable} $i >> 2_{$ejecutable}.dat
@ i += 3600
end

set ejecutable = seleccion
@ i = 10000
while ( $i <= 100000 )
./{$ejecutable} $i >> 3_{$ejecutable}.dat
@ i += 3600
end
```

2.4. Eficiencia Empirica

Tabla de resultados de eficiencia empírica

Tamaño(nº elementos)	Burbuja	Inserción	Selección
10000	0.2	0.05	0.04
13600	0.37	0.09	0.07
17200	0.58	0.14	0.12
20800	0.86	021	0.17
24400	1.17	0.29	0.24
28000	1.54	0.4	0.31
31600	1.97	0.5	0.43
35200	2.44	0.62	0.49
38800	2.96	0.75	0.6
42400	3.54	0.9	0.72
46000	4.16	1.06	0.85
49600	2.85	1.24	0.98
53200	5.6	1.42	1.13
56800	6.36	1.64	1.29
60400	7.18	1.86	1.46
64000	8.08	2.03	1.64
67600	8.99	2.27	1.83
71200	9.98	2.52	2.03
74800	11.02	2.8	2.24
78400	12.09	3.06	2.45
82000	13.25	3.35	2.69
85600	14.44	3.66	2.93
89200	15.73	3.97	3.18
92800	16.98	4.3	3.45
96400	18.32	4.63	3.72
100000	19.73	4.99	4

Gráfica de resultados de eficiencia empírica



2.5. Eficiencia Híbrida

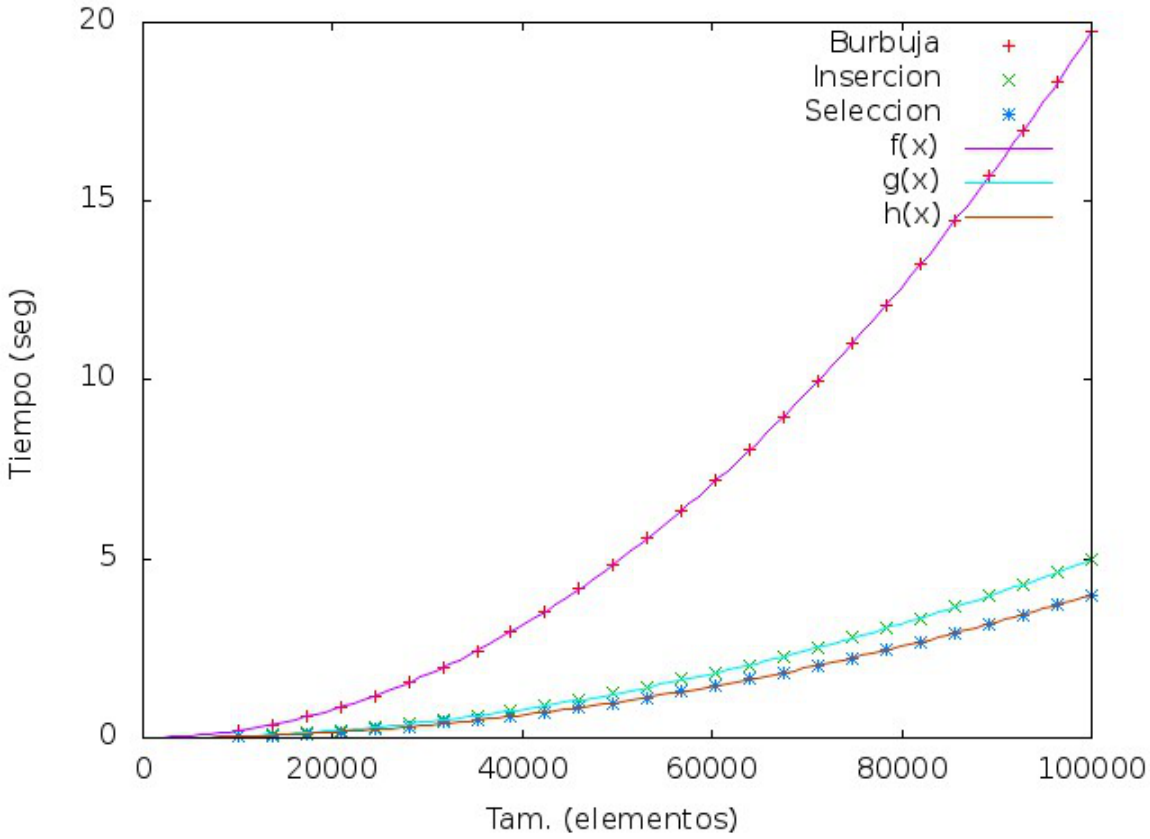
Ajuste de la función

burbuja, $f(x) = a_0 * x * x + a_1 * x * a_2$

insercion, $g(x) = a_3 * x * x + a_4 * x * a_5$

seleccion, $h(x) = a_6 * x * x + a_7 * x * a_8$

Final set of parameters	Asymptotic Standard Error
$a0 = 1{,}97866e^{-09}$	$+/- 3{,}971e^{-12}(0{,}2007\,\%)$
$a1 = -7{,}59247e^{-07}$	$+/- 4{,}472e{-}07(58{,}9\,\%)$
$a2 = 0{,}0134747$	$+/- 0{,}01084(80{,}44\,\%)$
$a3 = 4{,}94854e^{10}$	$+/- 2{,}723e^{-12}(0{,}5503\,\%)$
$a4 = 4{,}45443e^{-07}$	$+/- 3{,}067e^{-07}(68{,}85\,\%)$
$a5 = -0{,}00833544$	$+/- 0{,}007434(89{,}19\,\%)$
$a6 = 3{,}99793e^{-10}$	$+/- 2{,}149e^{-12}(0{,}5375\,\%)$
$a7 = 1{,}13621e^{-08}$	$+/- 2{,}42e^{-07}(2130\,\%)$
$a8 = 0{,}000859449$	$+/- 0{,}005866(682{,}6\,\%)$



2.6. Comparación de ajustes para el algoritmo Burbuja

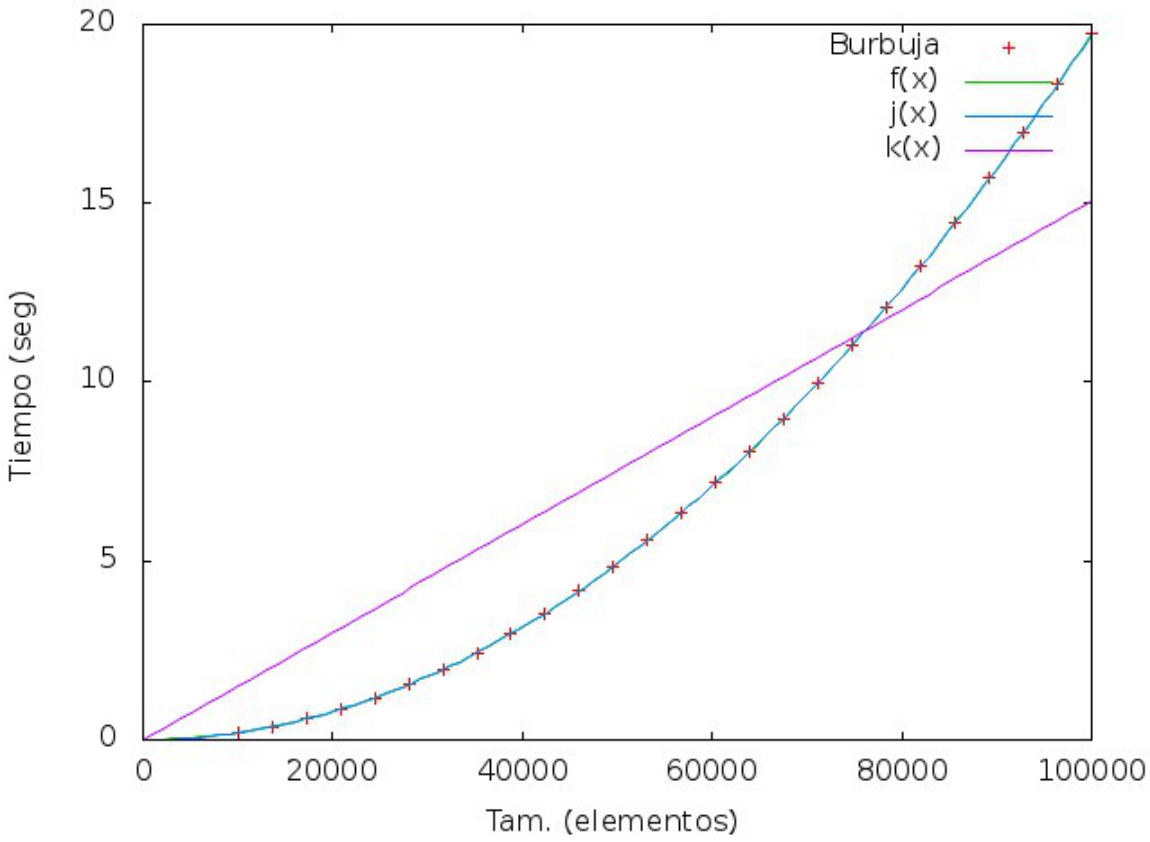
$f(x) = a_0 * x^2 + a_1 * x * a_2$

$j(x) = b * x^2$

$k(x) = c * x$

Final set of parameters	Asymptotic Standard Error
$a_0 = 1{,}97866e^{-09}$	$+/- 3{,}971e{-}12(0{,}2007\,\%)$
$a_1 = -7{,}59247e^{-07}$	$+/- 4{,}472e^{-07}(58{,}9\,\%)$
$a_2 = 0{,}0134747$	$+/- 0{,}01084(80{,}44\,\%)$
$b = 1{,}9715e^{-09}$	$+/- 5{,}52e^{-13}(0{,}028\,\%)$
$c = 0{,}000150531$	$+/- 7{,}753e^{-06}(5{,}151\,\%)$

Podemos observar como el segundo ajuste cuadrático "j(x)" es incluso mejor que el ajuste "f(x)" mientras que el ajuste lineal "k(x)" es bastante peor que los dos anteriores.



3. Análisis de Algoritmos de Ordenación con Eficiencia O(n log n)

3.1. Características de la máquina utilizada

Hardware

Procesador: Intel Pentium Dual CPU T2390 @ 1.86GHz x 2
Memoria: 1,8 GiB

Sistema Operativo: Ubuntu 14.04 LTS 64-bit

3.2. Compilador usado y opciones de compilación

g++ -O2 -o heapsort heapsort.cpp

g++ -O2 -o mergesort mergesort.cpp

g++ -O2 -o quicksort quicksort.cpp

3.3. Script de ejecución

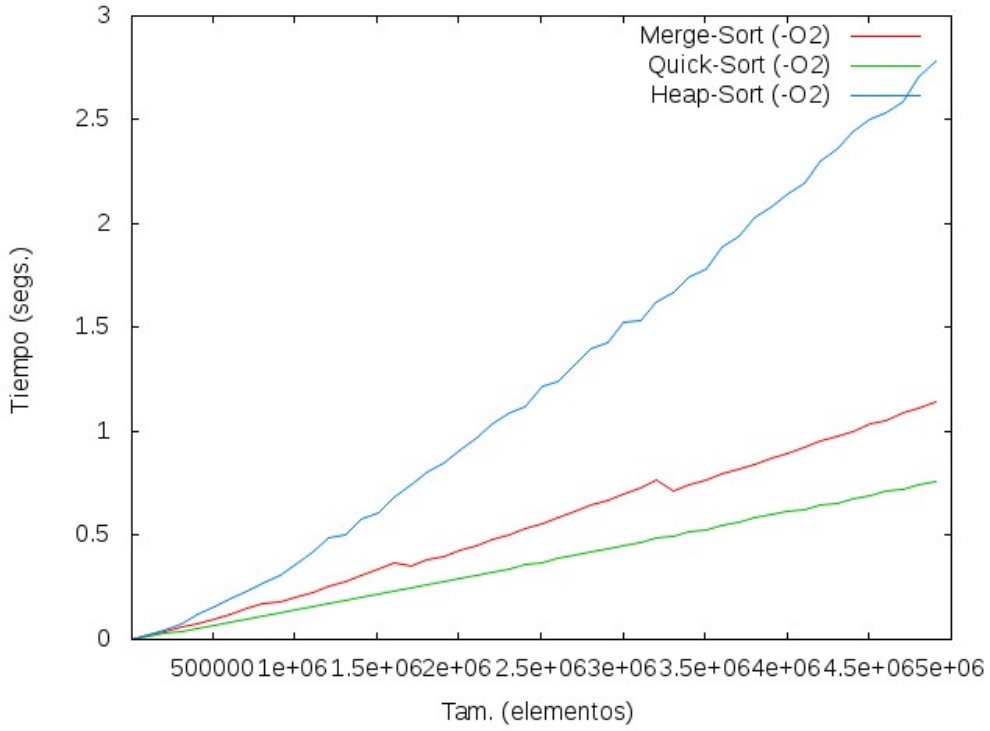
```
#!/bin/csh -vx

@ini = 10000
@fin = 3700000
@inc = 99800


set ejecutable = heapsort
@i = $ini
while ( $i < $fin )
        ./{$ejecutable} $i >> salida_{$ejecutable}.dat
        @i += $inc
end

set ejecutable = mergesort
@i = $ini
while ( $i < $fin )
        ./{$ejecutable} $i >> salida_{$ejecutable}.dat
        @i += $inc
end

set ejecutable = quicksort
@i = $ini
while ( $i < $fin )
        ./{$ejecutable} $i >> salida_{$ejecutable}.dat
        @i += $inc
end
```

3.4. Eficiencia empírica

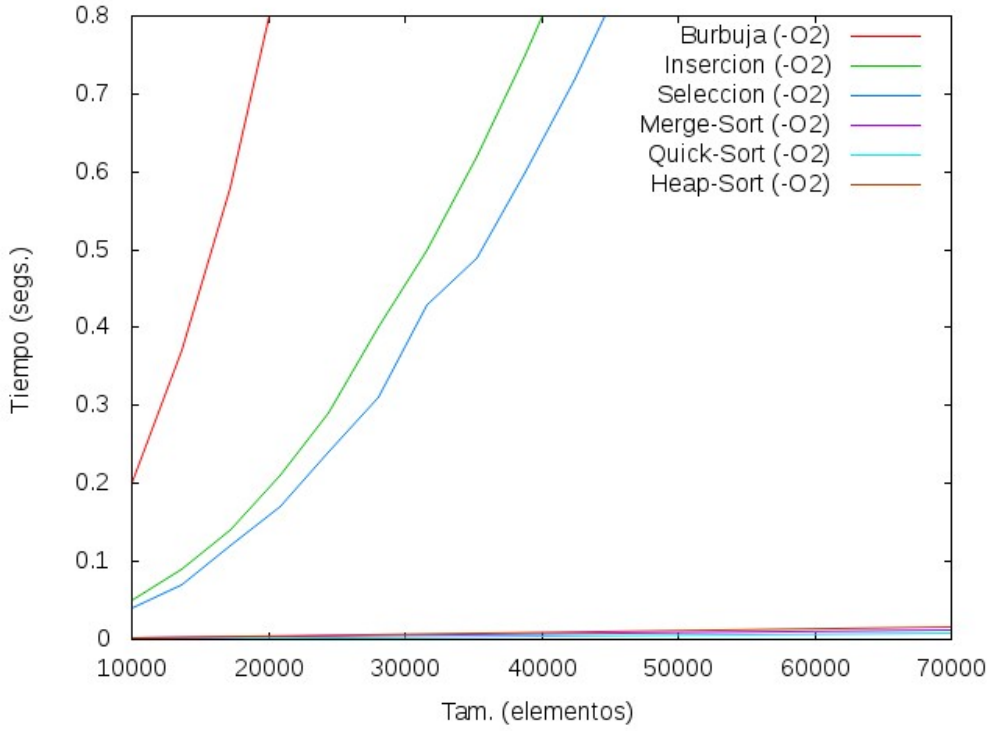
Tabla de resultados de eficiencia empírica

Tamaño	Merge-Sort	Quick-Sort	Heap-Sort
10000	0.001327	0.000977	0.001337
109800	0.019603	0.013088	0.024683
209600	0.039138	0.027347	0.048020
309400	0.059717	0.039729	0.074922
409200	0.078414	0.053883	0.118198
509000	0.099300	0.068902	0.157687
608800	0.123325	0.081956	0.192630
708600	0.148113	0.096455	0.235000
808400	0.176119	0.110701	0.274389
908200	0.180272	0.125627	0.311454
1008000	0.206591	0.141467	0.363121
1107800	0.228793	0.156301	0.423965
1207600	0.256883	0.170729	0.490211
1307400	0.281103	0.185917	0.504463
1407200	0.309763	0.202339	0.581250
1507000	0.339500	0.216542	0.610936
1606800	0.368010	0.230469	0.685213
1706600	0.353934	0.248052	0.743007
1806400	0.380326	0.261371	0.805761
1906200	0.402022	0.277369	0.847026
2006000	0.430930	0.293183	0.912391
2105800	0.451173	0.305356	0.968117
2205600	0.480629	0.323580	1.035170
2305400	0.504747	0.337295	1.092640
2405200	0.534223	0.359377	1.121840
2505000	0.558312	0.371952	1.216200
2604800	0.588402	0.391270	1.243600
2704600	0.613907	0.402472	1.326250
2804400	0.647271	0.419786	1.398240
2904200	0.670900	0.435857	1.431920
3004000	0.702285	0.449628	1.527870
3103800	0.729837	0.464104	1.534230
3203600	0.765512	0.488097	1.621890
3303400	0.716479	0.497329	1.668150
3403200	0.743978	0.516030	1.747850
3503000	0.767855	0.529033	1.782230
3602800	0.796204	0.546855	1.885810

Gráfica comparativa de algoritmos logarítmicos



Gráfica comparativa de algoritmos de ordenación



3.5. Eficiencia híbrida

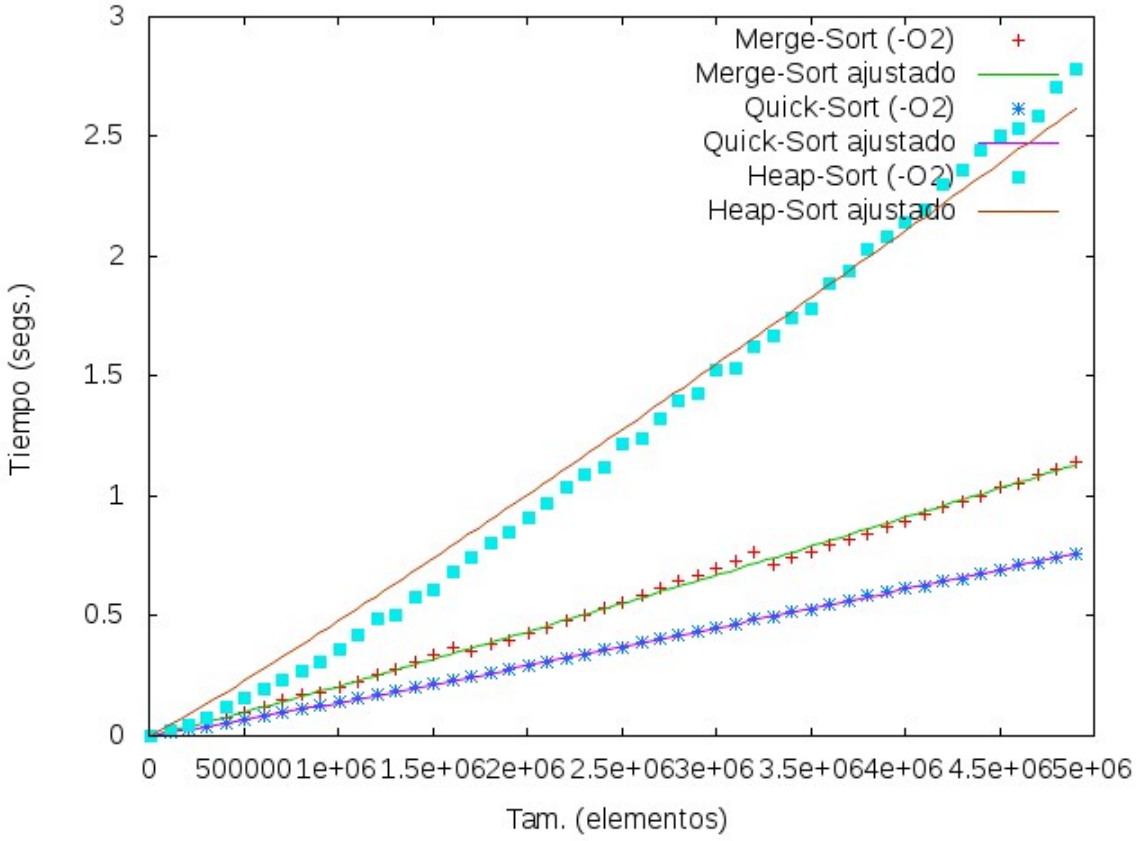
Ajuste de la función

mergesort, $f(x) = a * x * log(x)$

insercion, $g(x) = b * x * log(x)$

seleccion, $h(x) = c * x * log(x)$

Final set of parameters	Asymptotic Standard Error
$a = 1{,}49854e^{-08}$	$+/- 5{,}042e^{-11} (0{,}3365\,\%)$
$b = 1{,}00635e^{-08}$	$+/- 7{,}407e^{-12} (0{,}0736\,\%)$
$c = 3{,}46568e^{-08}$	$+/- 2{,}856e^{-10} (0{,}8239\,\%)$



3.6. Comparación de ajustes para el algoritmo Quicksort

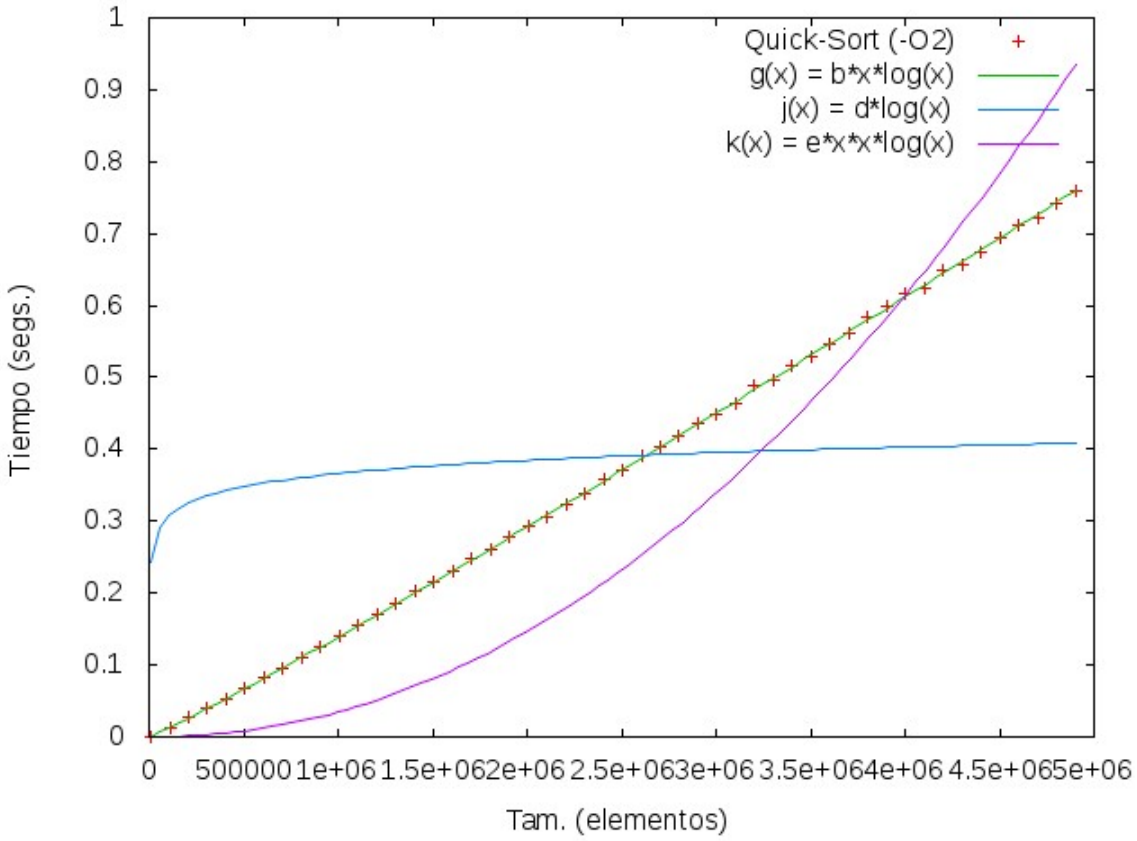
$f(x) = a * x * log(x)$

$j(x) = d * log(x)$

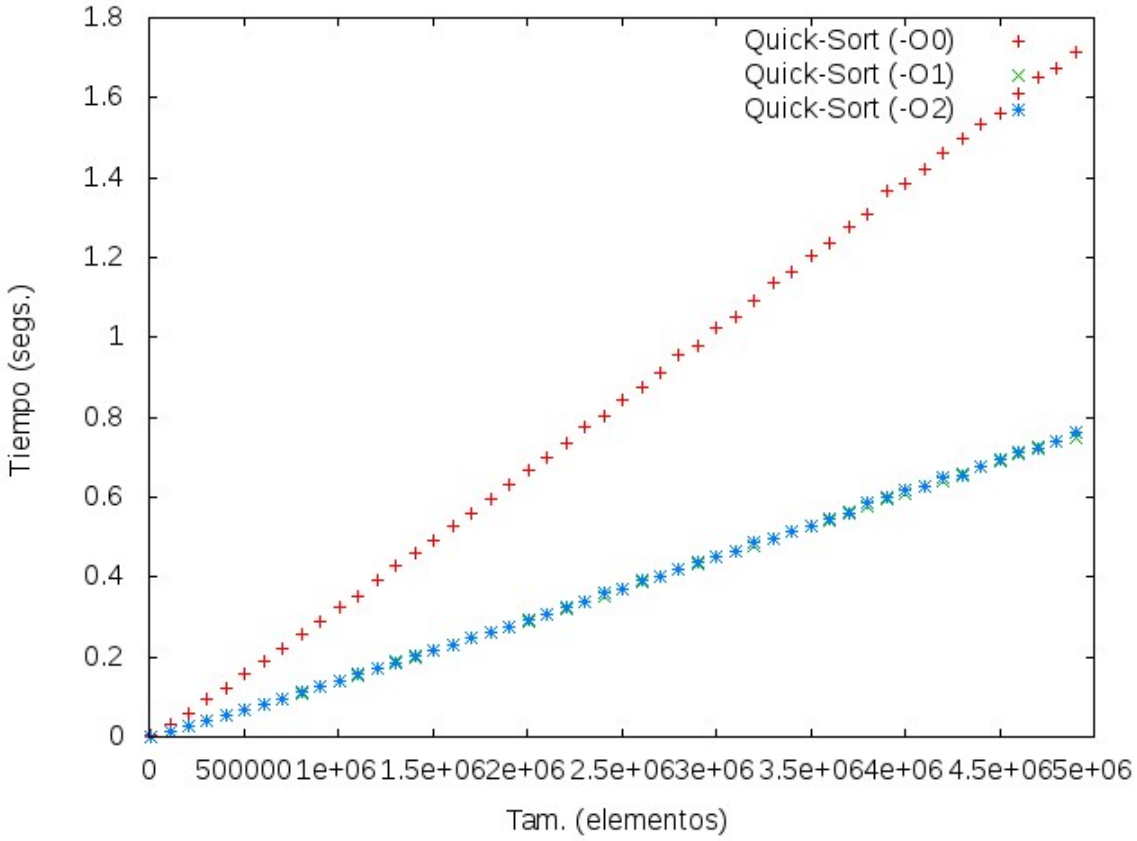
$k(x) = e * x^2 * log(x)$

Final set of parameters	Asymptotic Standard Error
$a = 1{,}49854e^{-08}$	$+/- 5{,}042e^{-11} (0{,}3365\,\%)$
$d = 0{,}0265147$	$+/- 0{,}002001 (7{,}548\,\%)$
$e = 2{,}5246e^{-15}$	$+/- 9{,}02e^{-17} (3{,}573\,\%)$

Se puede observar gráficamente el crecimiento de un algoritmo con un orden de eficiencia O(n log n) en comparación con el crecimiento de una función logarítmica (log n) y una función n^2 log n.



3.7. Comparativa de Optimizaciones del Compilador



4. Análisis del Algoritmo de Floyd con eficiencia O(n^3)

4.1. Características de la máquina utilizada

Hardware

Procesador: Intel Core i5-2430M CPU @ 2.40GHz x4

Memoria: 6 GiB

Sistema Operativo: Ubuntu 14.04 LTS 64-bit

4.2. Compilador usado y opciones de compilación

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

g++ -O0 -o floyd floyd.cpp -std=gnu++0x

g++ -O1 -o floyd floyd.cpp -std=gnu++0x

g++ -O2 -o floyd floyd.cpp -std=gnu++0x

4.3. Script de ejecución

```
#!/bin/bash

echo "Tiempos de ejecucion" > salida.dat

for((i=100; i <= 3000; i=i+100))
    do
        printf "$i " >> salida.dat ; ./floyd $i >> salida.dat
    done
```

El código de floyd.cpp se ha modificado ligeramente introduciendo iteraciones de 100 para cada ejecución del programa. Después se multiplican los ciclos por segundo por esta variable para obtener unos resultados más precisos cuando hacemos mediciones de tiempos muy rápidos.
Cada ejecución del programa se hace a partir de 100 nodos de un grafo hasta llegar a 3000 de 100 en 100.

4.4. Eficiencia empírica

Tabla de resultados de eficiencia empírica

Entrada (nº de nodos de un grafo)	Tiempo Algoritmo de Floyd (s)
100	0.004903
200	0.015850
300	0.032081
400	0.072025
500	0.137821
600	0.234690
700	0.383734
800	0.587289
900	0.872558
1000	1.224310
1100	1.654030
1200	2.161490
1300	2.720190
1400	3.378910
1500	4.157390
1600	4.997010
1700	5.985960
1800	7.079230
1900	8.347460
2000	9.665220
2100	11.17560
2200	12.80760
2300	14.71100
2400	16.54150
2500	18.69650
2600	20.97660
2700	23.58870
2800	27.07430
2900	30.38790
3000	32.80490

4.5. Eficiencia híbrida

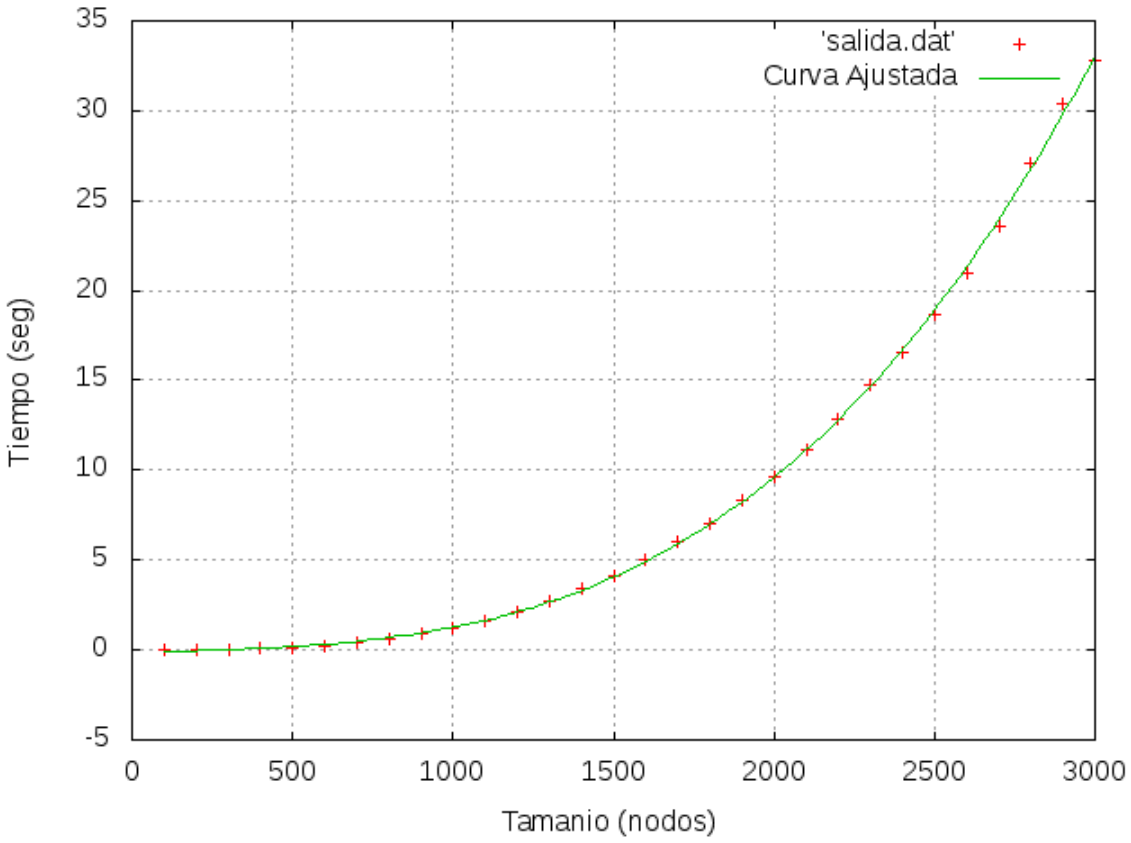
Ajuste de la función

Floyd, $f(x) = a_0 * x^3 + a_1 * x^2 + a_2 * x + a_3$

Final set of parameters	Asymptotic Standard Error
$a_0 = 1{,}36468e^{-09}$	$+/- 6{,}953e^{-11}(5{,}095\,\%)$
$a_1 = -6{,}4347e^{-07}$	$+/- 3{,}276e^{-07}\ (50{,}9\,\%)$
$a_2 = 0{,}000718562$	$+/- 0{,}0004408(61{,}35\,\%)$
$a_3 = -0{,}172498$	$+/- 0{,}1604(92{,}99\,\%)$

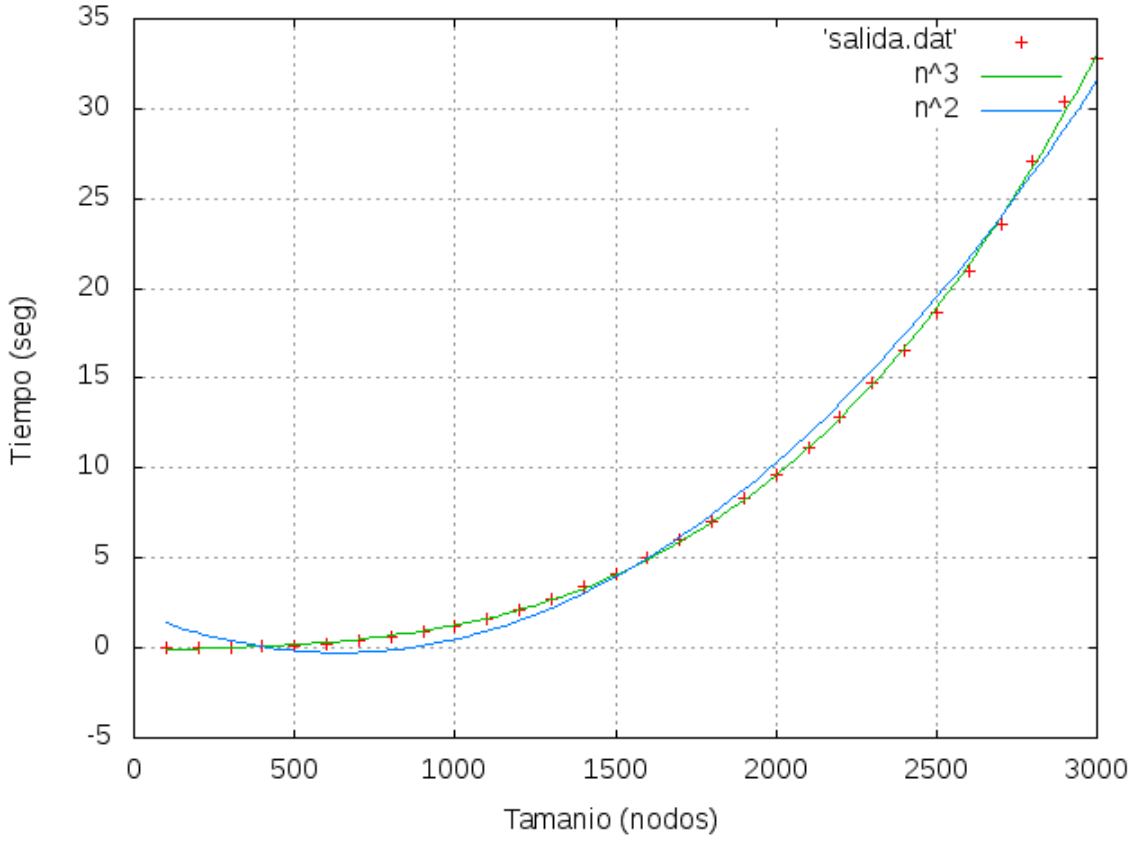
Como podemos observar, el error estándar es de sólo 0.5 cuando se intenta ajustar la curva a una variable de tipo cúbico. No obstante, el error aumenta de manera considerable con las otras variables del polinomio, es decir, con el cuadrado, la lineal y el término independiente.
Tomamos este ajuste como bueno y procedemos a representar gráficamente la curva ajustada y la curva de los datos originales.

Tomamos este ajuste como bueno y procedemos a representar gráficamente la curva ajustada y la curva de los datos originales.



4.6. Comparación con otros ajustes

He aquí una comparativa con un polinomio de segundo grado.



La gráfica se ajusta mejor a un polinomio de tercer grado. No obstante, aunque el error estándarpara una función cuadrática sea menor (3 % frente al 5 % del cúbico) las constantes ocultas aumentan.

$f(x) = b_0 * x^2 + b_1 * x + b_2$

Final set of parameters	**Asymptotic Standard Error**
$b_0 = 5{,}70231e^{-06}$	$+/- 6{,}953e^{-11} (5{,}095\,\%)$
$b_1 = -0{,}00727985$	$+/- 0{,}0006561 (9{,}013\,\%)$
$b_2 = 2{,}06122$	$+/- 0{,}4412 (21{,}41\,\%)$

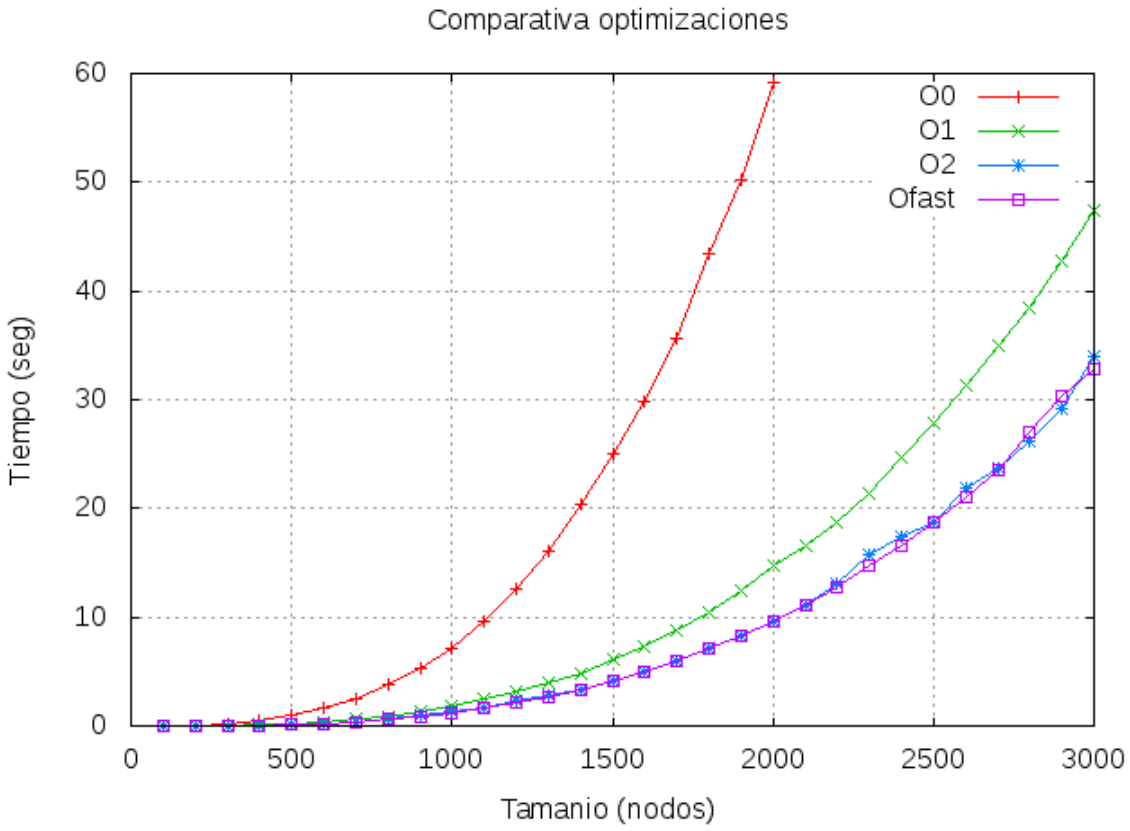
¿Se podría decir que en este caso se ha comportado como una función cuadrática?

Sí, podría afirmase dada la pequeña muestra en las mediciones en los que los tiempos de ejecución ya se disparaban. Esto ha hecho que el ajuste a una curva ligeramente mejor a una funcióncuadrática.

Incluso así, la teoría nos dice que se comporta como un algoritmo de orden cúbico y es así su comportamiento cuando trabajamos con nodos de entradas superiores donde los tiempos de ejecución son inmanejables.

4.7. Comparativa de Optimizaciones del Compilador

Por último mostramos una gráfica comparativa con los distintos tipos de optimización y los tiempos de ejecución.



De la optimización O0 (sin optimización) sólo he tomado 20 muestras hasta 2000 nodos porque se disparaban demasiado los tiempos.

Destacar la forma exponencial de la optimización O0.

5. Análisis de Algoritmo de Fibonacci con eficiencia O($((1+\sqrt{5})/2)^n$)

5.1. Características de la máquina utilizada

Hardware

Procesador: Intel Core i7-3630QM CPU @ 2.40GHz x 8

Memoria: 7,7 GiB

Sistema Operativo: Ubuntu 14.04 LTS 64-bit

5.2. Compilador usado y opciones de compilación

g++ -O2 -o fibonacci fibonacci.cpp -std=gnu++0x

5.3. Script de ejecución

```
#!/bin/bash

for((i = 0; i < 30; i++))
    do
        ./fibonacci >> salida.dat
    done
```

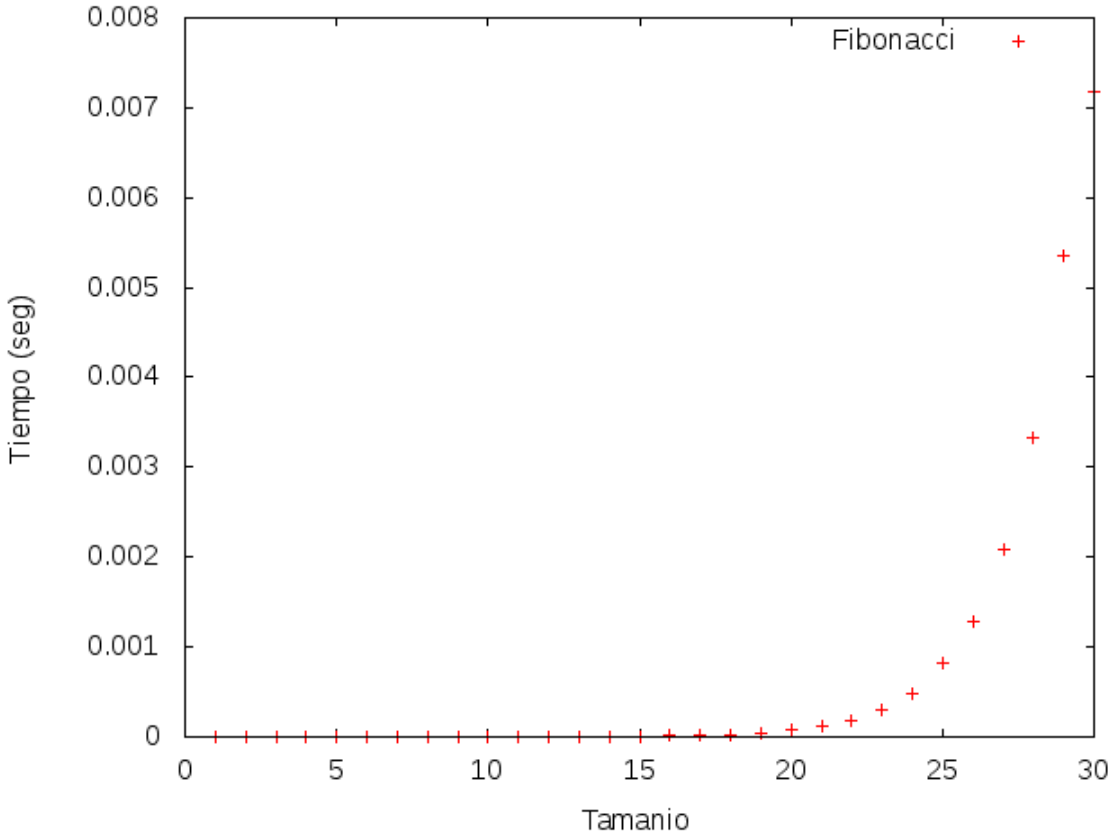
El código de fibonacci.cpp se ha modificado ligeramente introduciendo iteraciones para cada ejecución del programa. Después se multiplican los ciclos por segundo por esta variable para obtener unos resultados más precisos cuando hacemos mediciones de tiempos muy rápidos.
Cada ejecución del programa se hace a partir de 1 elemento hasta llegar a 30 de 1 en 1.

5.4. Eficiencia empírica

Tabla de resultados de eficiencia empírica

Tamaño (no de elementos)	Fibonacci
1	3.41e-07
2	4.04e-07
3	4.44-07
4	4.41e-07
5	6.32e-07
6	6.75e-07
7	6.66e-07
8	9.39e-07
9	1.627e-06
10	1.971e-06
11	1.707e-06
12	2.632e-06
13	3.502e-06
14	4.921e-06
15	7.51e-05
16	1.1402e05
17	1.7724e-05
18	1.0268e-05
19	4.5135e-05
20	7.207e-05
21	0.000115658
22	0.000186563
23	0.000302145
24	0.000486496
25	0.000826974
26	0.00127498
27	0.00208339
28	0.00333583
29	0.00535737
30	0.00718014

Gráfica de eficiencia de algoritmo de Fibonacci

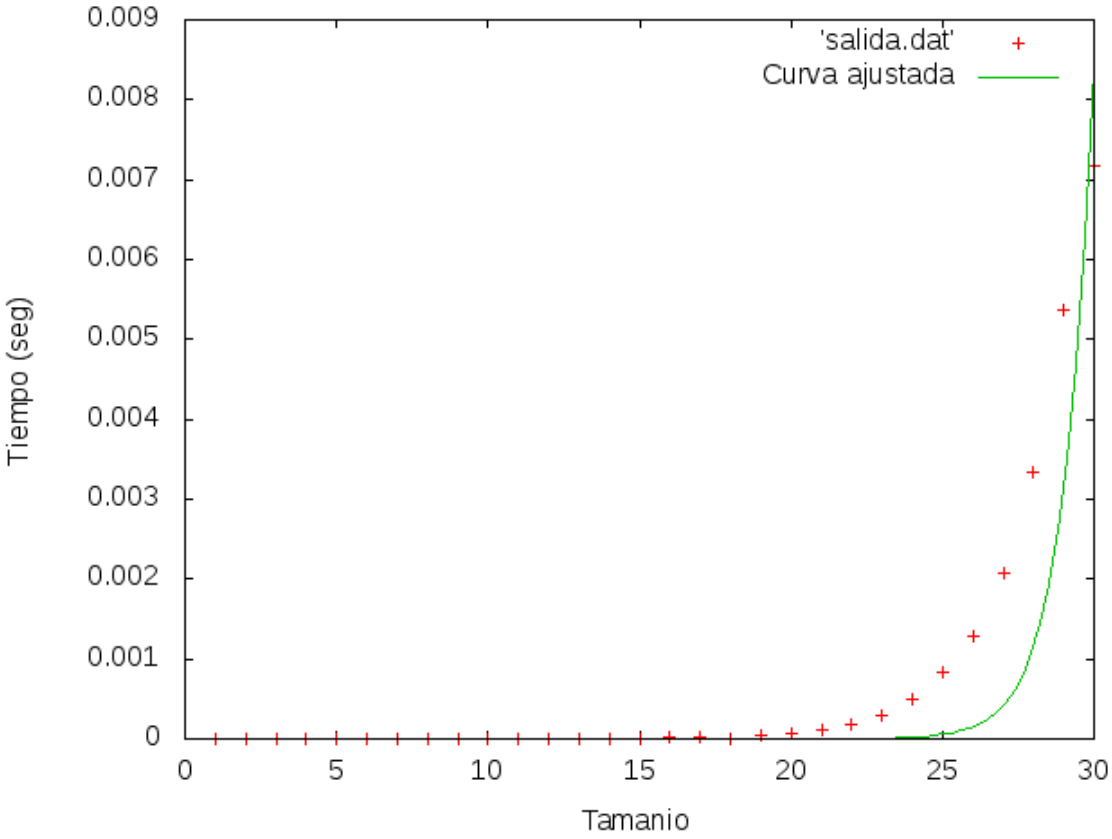


5.5. Eficiencia híbrida

Ajuste de la función

Fibonacci, $f(x) = b^x$

Final set of parameters	Asymptotic Standard Error
$b = 7{,}87814e^{-16}$	$+/- 6{,}552e^{-17} (8{,}317\,\%)$



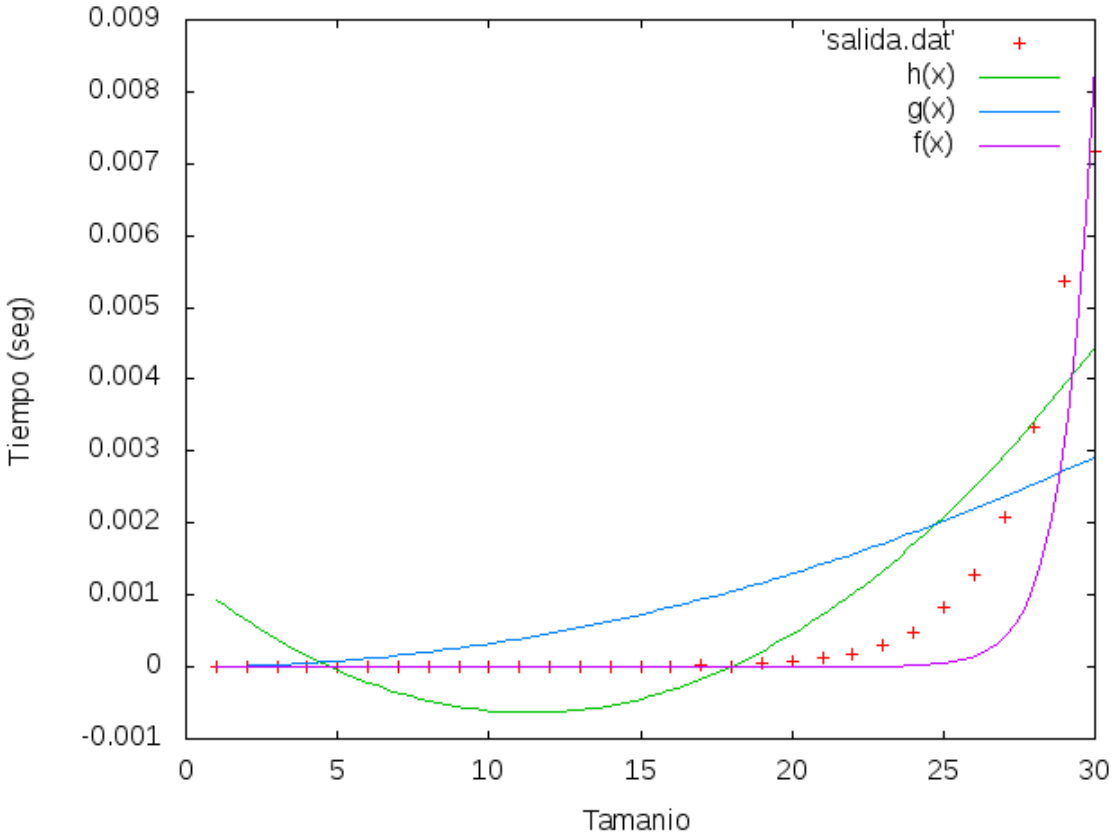
5.6. Comparación con otros ajustes.

$f(x) = b^x$

$g(x) = c * x^2$

$h(x) = a_0 * x^2 + a_1 * x + a_2$

Final set of parameters	Asymptotic Standard Error
$b = 7{,}87814e^{-16}$	$+/- 6{,}552e^{-17}(8{,}317\,\%)$
$c = 3{,}24662e^{-06}$	$+/- 5{,}292e^{-07}(16{,}3\,\%)$
$a_0 = 1{,}46444e^{-05}$	$+/- 2{,}437e^{-06}(16{,}64\,\%)$
$a_1 = -0{,}000332944$	$+/- 7{,}787e^{-05}(23{,}39\,\%)$
$a_2 = 0{,}00125635$	$+/- 0{,}0005236(41{,}68\,\%)$



Como podemos apreciar, aunque hay un momento en que el ajuste con la función f(x) se separa de los puntos que representan nuestros datos luego vuelve a juntarse con los puntos, en cambio g(x) y h(x) apenas se acercan a la representación de nuestros datos.