Práctica 1 - Análisis Empírico e Híbrido de Eficiencia de Algoritmos

Memoria de práctica

Germán Castilla López
Jorge Gangoso Klöck
Pedro Morales Leyva
Clara Mª Romero Lara

Índice de Contenidos

1. Descripción general de la solución

Para realizar esta tarea hemos desarrollado un programa específico para el análisis de tiempos de los distintos algoritmos propuestos.

Dicho programa emplea `QueryPerformanceCounter(&LARGE_INTEGER)` de la librería de Windows antes y después de las llamadas a los algoritmos de ordenación para calcular el tiempo transcurrido durante las mismas.
La generación y destrucción de datos necesaria para emplear el algoritmo se realiza fuera de la llamada para así evitar contabilizarla junto al tiempo de ejecución del algoritmo en sí mismo.

Finalmente mencionar que el programa incorpora una función para automatizar la creación de scripts de GNUplot para exportar automáticamente gráficas ajustadas a sus respectivas funciones en formato pdf.

Los datos obtenidos son la media de 100 ejecuciones en cada punto para maximizar la precisión del cálculo y que la gráfica de eficiencia no se vea afectada por motivos externos a la complejidad del algoritmo (p.ej. un vector particularmente complicado, rendimiento puntual del ordenador...).

2. Cálculo de la eficiencia empírica e híbrida

El cálculo de la eficiencia empírica consiste en realizar pruebas experimentales de los algoritmos, generar ficheros de datos con los resultados de dichas pruebas y para analizarlo de forma más visual, generar gráficas empleando los ficheros de datos para comprobar la capacidad del algoritmo ante distintos tamaños de entrada de datos.

En este caso, y como se ha mencionado anteriormente, el cálculo de tiempo se realiza mediante la clase `Mtime`, que permite la creación de un objeto `Mtime` y de marcas temporales haciendo uso de la función `QueryPerformanceCounter(&large_integer)`. Finalmente aplicamos el método de clase `performancecounter_diff(&t1,&t2)` para obtener la diferencia de tiempo entre las dos marcas temporales que habremos situado antes y después de las llamadas a los algoritmos.

Para la eficiencia híbrida, nos apoyaremos en los conocimientos teóricos sobre los algoritmos utilizados y trataremos de ajustar las gráficas generadas a gráficas teóricas para comprobar la cantidad de desajuste que pueda aparecer, obteniendo en el proceso el valor de las constantes ocultas de nuestro caso particular.

Un buen ajuste implica un respaldo práctico al conocimiento teórico obtenido mediante el análisis del propio código (cantidad de bucles, bucles anidados, condiciones...) mientras que un mal ajuste podría significar un error en el cálculo teórico o bien unos valores de constantes ocultas demasiado grandes para ser despreciadas.

Estos cálculos se realizaron en un sistema *Windows 10 64 bits* con un procesador *AMD Ryzen 5 1600X Six-Core Processor 3.60 GHz.*

2.1. Algoritmos de ordenación

Los algoritmos de ordenación propuestos para estudio son: de orden O(n^2), el de Burbuja, Inserción y Selección; y de orden O(nlog(n)), Mergesort, Quicksort y Heapsort.

Para maximizar la precisión de las medidas temporales sobre todo en algoritmos más rápidos y algoritmos con una alta dependencia del caso inicial (orden inicial de los elementos del vector), las llamadas se ejecutan `NUMREPETICIONES` (valor 100, constante almacenada en *constantes.h*) veces, y después se realiza la media.

2.1.1. Complejidad O(n²)

Algoritmo de la burbuja

El algoritmo de Burbuja es un método muy sencillo de comprender e implementar pero con un amplio margen de mejora en cuanto a eficiencia.

La premisa es tan sencilla como comparar cada elemento con el siguiente e intercambiarlos en caso de que sea necesario, hacer esto para todo el vector y repetir hasta que el vector esté ordenado.

La cantidad de comparaciones depende únicamente del número de elementos y no varía con el orden inicial del vector, aunque sí lo hará la cantidad de intercambios necesarios.

Nº elementos	**Tiempo**	**Nº elementos**	**Tiempo**	**Nº elementos**	**Tiempo**
100	0.018406	**900**	1.58169	**1700**	5.6963
200	0.077334	**1000**	1.96444	**1800**	6.43445
300	0.172528	**1100**	2.37714	**1900**	7.11589
400	0.310599	**1200**	2.81228	**2000**	7.89364
500	0.480879	**1300**	3.31978	**2100**	8.81611
600	0.703714	**1400**	3.90306	**2200**	9.6025
700	0.96251	**1500**	4.4237	**2300**	10.5587
800	1.23816	**1600**	5.06018	**2400**	11.5164
				2500	12.394

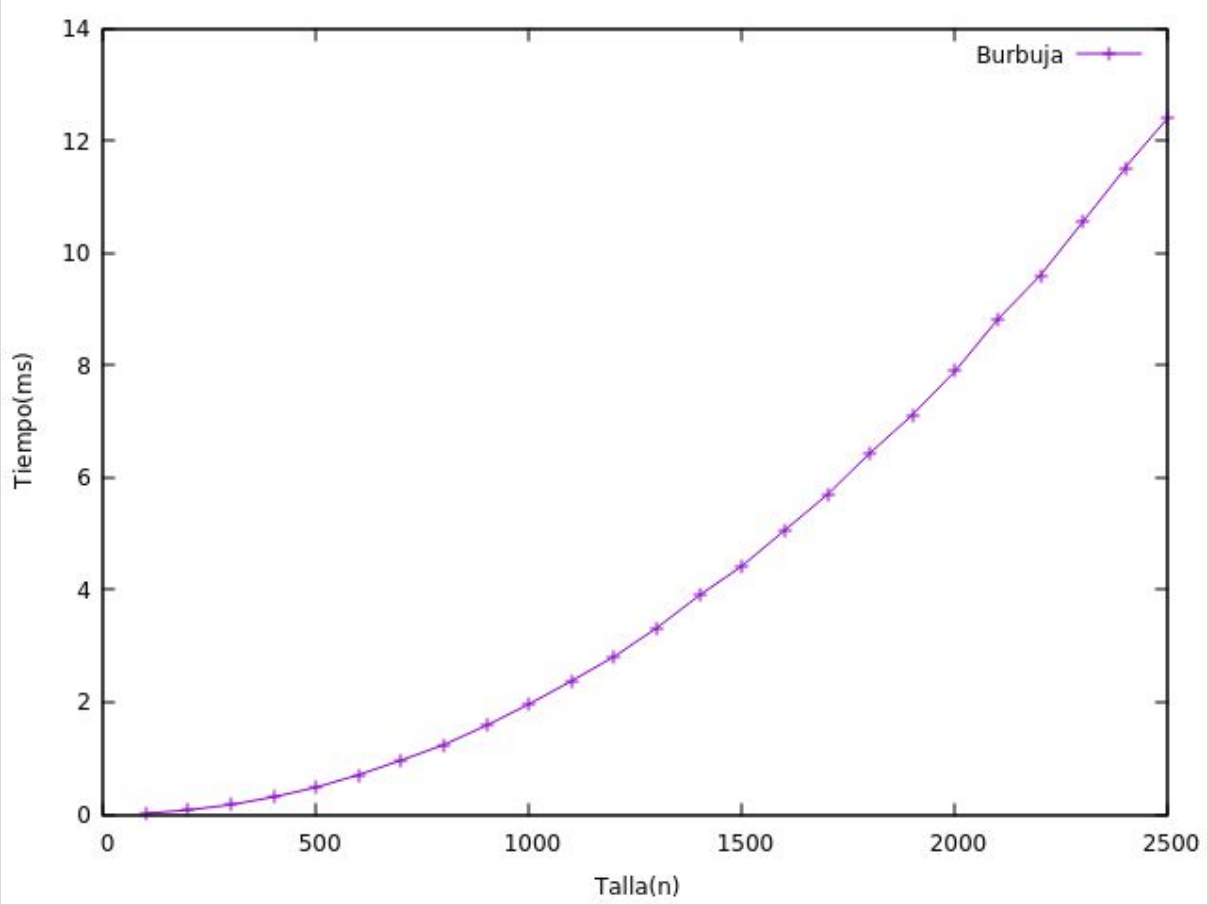
Una vez ajustamos la gráfica a $T(n) = an^2+bx+c$, obtenemos las siguientes constantes ocultas:

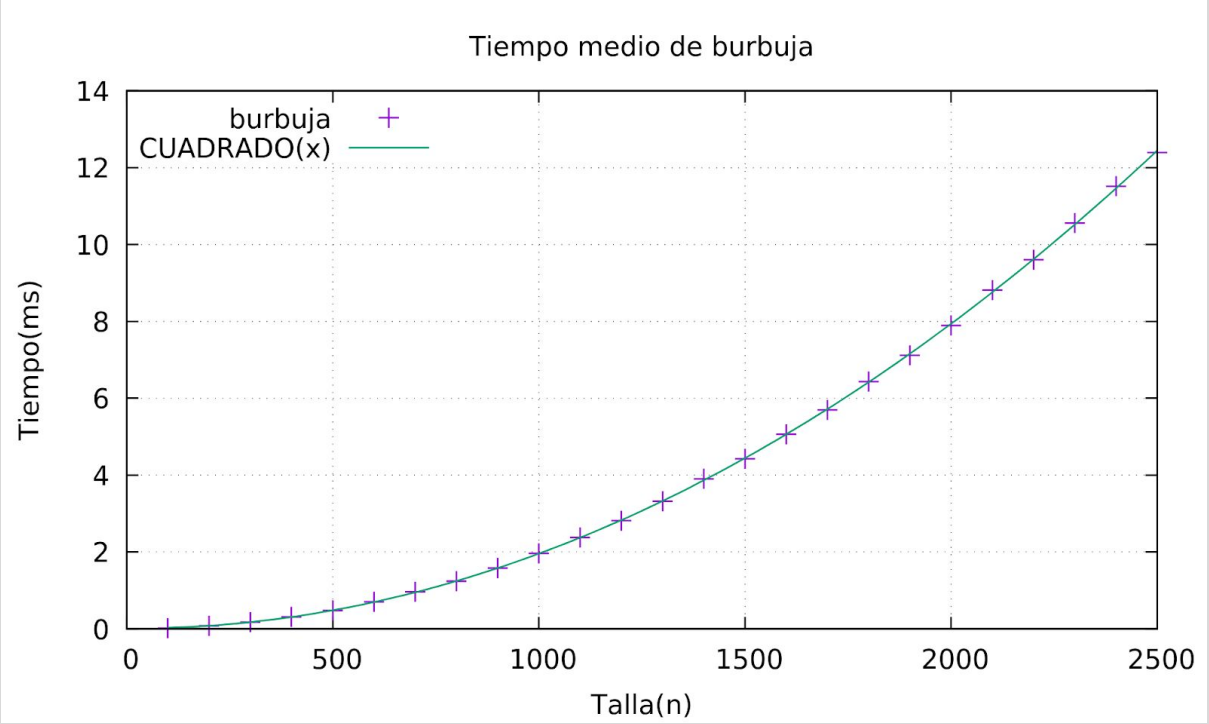
$$a = 1.89967e\text{-}06$$
$$b = 4.55868e\text{-}05$$
$$c = -0.00779279$$

Lo cual nos da una una función ajustada tal que:

$$f(x) = 1.89967e\text{-}06*x^2 + 4.55868e\text{-}05*x - 0.00779279$$

Con un porcentaje de error de la constante de mayor peso del 0.9923%


Burbuja
Tiempo(ms)
Talla(n)
0
2
4
6
8
10
12
14
0
500
1000
1500
2000
2500


Tiempo medio de burbuja
burbuja
CUADRADO(x)
Tiempo(ms)
Talla(n)
0
2
4
6
8
10
12
14
0
500
1000
1500
2000
2500

Algoritmo de inserción

El algoritmo de inserción es parecido al de burbuja en cuanto a simplicidad. Se seleccionan los elementos uno a uno de principio a fin de vector y se compara con los elementos anteriores, dejándolo en el sitio si es el mayor o poniéndose donde corresponda si no.

La mejora que presenta este algoritmo respecto al de la Burbuja es que en el caso de un vector inicialmente ordenado únicamente se compara cada elemento con su anterior, siendo así su caso mejor de orden O(n).

En el caso medio y peor sin embargo sigue siendo un algoritmo de orden O(n^2).

Nº elementos	**Tiempo**	**Nº elementos**	**Tiempo**	**Nº elementos**	**Tiempo**
100	0.008128	**900**	0.552689	**1700**	1.95341
200	0.028364	**1000**	0.673295	**1800**	2.19376
300	0.064762	**1100**	0.820795	**1900**	2.42875
400	0.115956	**1200**	0.961247	**2000**	2.68068
500	0.17762	**1300**	1.13852	**2100**	3.00767
600	0.250471	**1400**	1.33962	**2200**	3.3597
700	0.341318	**1500**	1.52159	**2300**	3.6529
800	0.439213	**1600**	1.7349	**2400**	3.98343
				2500	4.24379

Una vez ajustamos la gráfica a $T(n) = an^2+bx+c$, obtenemos las siguientes constantes ocultas:

$$a = 6.66867e\text{-}07$$

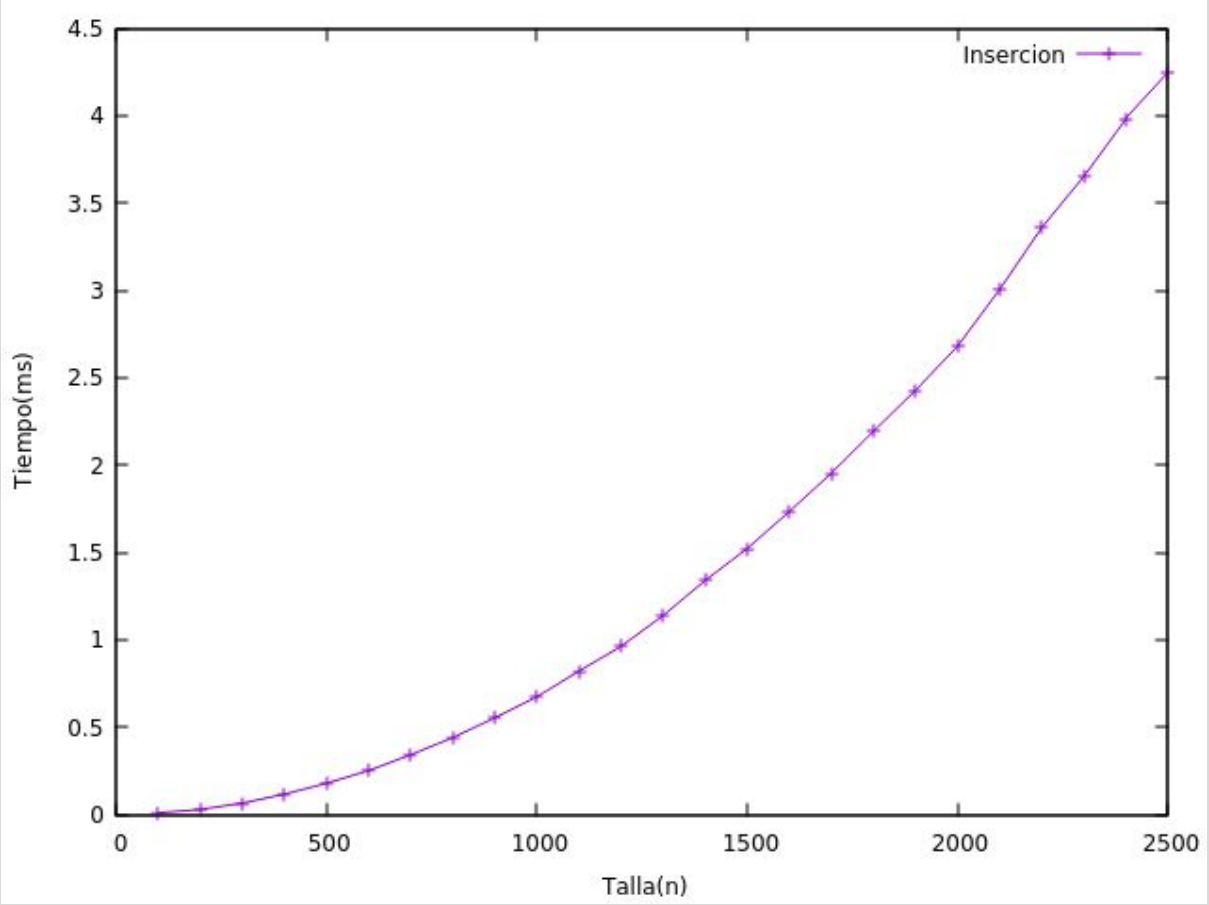
$$b = 1.93102e\text{-}05$$

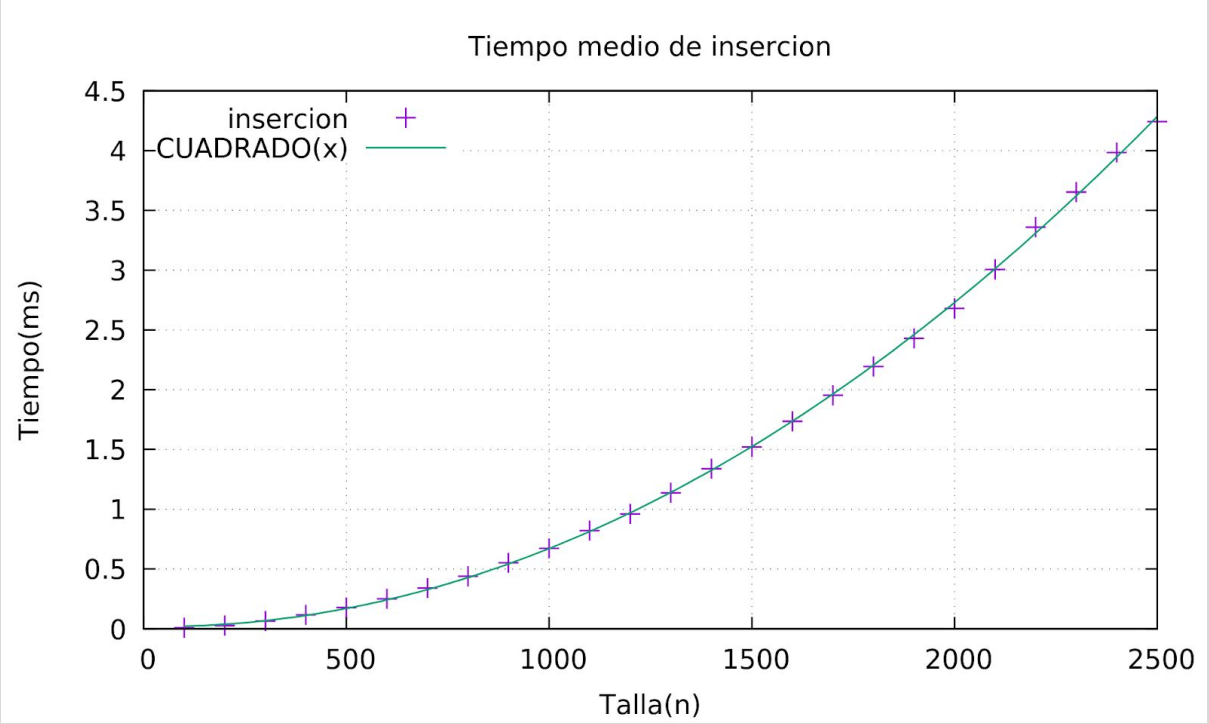
$$c = 0.000608212$$

Lo cual nos da una una función ajustada tal que:

$$f(x) = 6.66867e\text{-}07*x^2 + 1.93102e\text{-}05*x + 0.000608212$$

Con un porcentaje de error de la constante de mayor peso del 1.993%


Insercion
Tiempo(ms)
Talla(n)
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
0
500
1000
1500
2000
2500


Tiempo medio de insercion
insercion
CUADRADO(x)
Tiempo(ms)
Talla(n)
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
0
500
1000
1500
2000
2500

Algoritmo de selección

Éste algoritmo es uno muy utilizado por el ser humano ya que consiste en buscar en cada iteración el menor elemento del vector y fijarlo en la menor posición. A partir de ese momento, el valor ya está ordenado así que se procede a repetir tantas veces como elementos haya en el vector.

Una vez se sabe el funcionamiento es claro por qué la eficiencia es O(n^2), puesto que se busca entre los n elementos del vector el menor y esto se repite n veces.

Nuevamente se trata de un algoritmo cuyo número de iteraciones no varía según la ordenación inicial del vector (no hay diferenciación de casos).

Nº elementos	**Tiempo**	**Nº elementos**	**Tiempo**	**Nº elementos**	**Tiempo**
100	0.016588	**900**	1.13689	**1700**	3.89668
200	0.061348	**1000**	1.36049	**1800**	4.37472
300	0.132183	**1100**	1.64235	**1900**	4.87033
400	0.230774	**1200**	1.96355	**2000**	5.38332
500	0.353644	**1300**	2.29655	**2100**	5.92274
600	0.504102	**1400**	2.66066	**2200**	6.50515
700	0.683031	**1500**	3.04805	**2300**	7.10051
800	0.910039	**1600**	3.46187	**2400**	7.71828
				2500	8.37727

Una vez ajustamos la gráfica a $T(n) = an^2+bx+c$, obtenemos las siguientes constantes ocultas:

$$a = 1.29979e\text{-}06$$

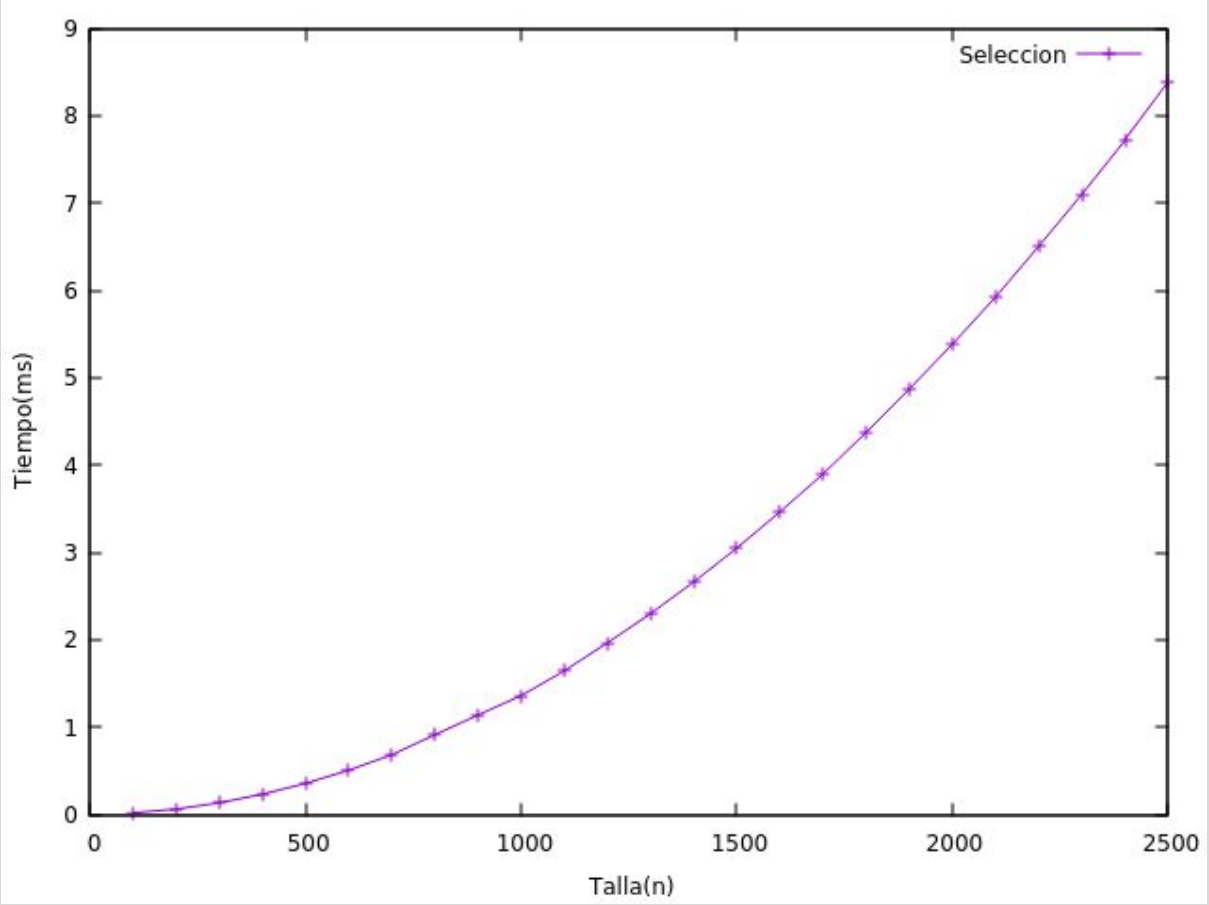
$$b = 6.42225e\text{-}05$$

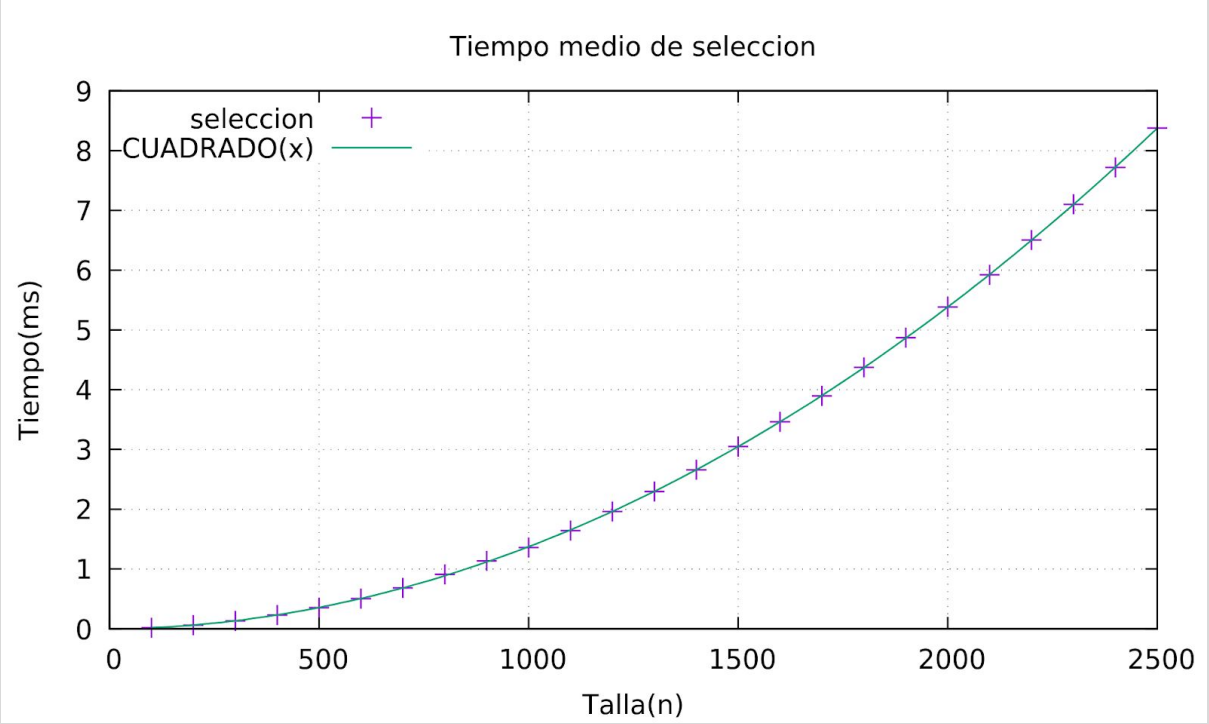
$$c = -0.0037786$$

Lo cual nos da una una función ajustada tal que:

$$f(x) = 1.29979e\text{-}06*x^2 + 6.42225e\text{-}05*x - 0.0037786$$

Con un porcentaje de error de la constante de mayor peso del 0.3513%


Seleccion
Tiempo(ms)
Talla(n)
0
1
2
3
4
5
6
7
8
9
0
500
1000
1500
2000
2500


Tiempo medio de seleccion
seleccion
CUADRADO(x)
Tiempo(ms)
Talla(n)
0
1
2
3
4
5
6
7
8
9
0
500
1000
1500
2000
2500

2.1.2. Complejidad O(nlog(n))

Algoritmo heapsort

El método heapsort es un método más potente de ordenación si bien también es más complejo de comprender y de implementar.

A grandes rasgos, este algoritmo coloca los elementos en una estructura de árbol o equivalente, y se establece una norma (en nuestro caso de heapsort máximo) por la cual un "nodo" padre nunca debe ser menor que los hijos. Por lo tanto el árbol generado inicialmente se redistribuye sistemáticamente para que se cumpla dicha norma.
Una vez hecho ésto podemos asegurar que el elemento por encima de todos, el nodo raíz, es el mayor elemento y podemos almacenarlo en la mayor posición del vector.
Esta reordenación (reajuste) se realizará, por lo tanto, n veces; necesitando para cada iteración un tiempo de orden log(n) para reajustarse.

Este algoritmo se encuentra casi a la par con la capacidad de quicksort, el cual es considerado el algoritmo de ordenación más rápido hasta el momento. Suele ser algo más lento de media pero carece del peor caso de orden O(n^2) que tiene quicksort, si bien hay alternativas que permiten evadir dicho peor caso.

Nº elementos	**Tiempo**	**Nº elementos**	**Tiempo**	**Nº elementos**	**Tiempo**
100	0.003988	**900**	0.058385	**1700**	0.118929
200	0.010331	**1000**	0.063153	**1800**	0.138901
300	0.01512	**1100**	0.073915	**1900**	0.140219
400	0.020632	**1200**	0.080904	**2000**	0.143234
500	0.029845	**1300**	0.084862	**2100**	0.156013
600	0.037363	**1400**	0.090613	**2200**	0.16988
700	0.041316	**1500**	0.101735	**2300**	0.169518
800	0.047769	**1600**	0.111671	**2400**	0.179607
				2500	0.187783

Una vez ajustamos la gráfica a T(n) = an*log(n) + b, obtenemos las siguientes constantes ocultas:

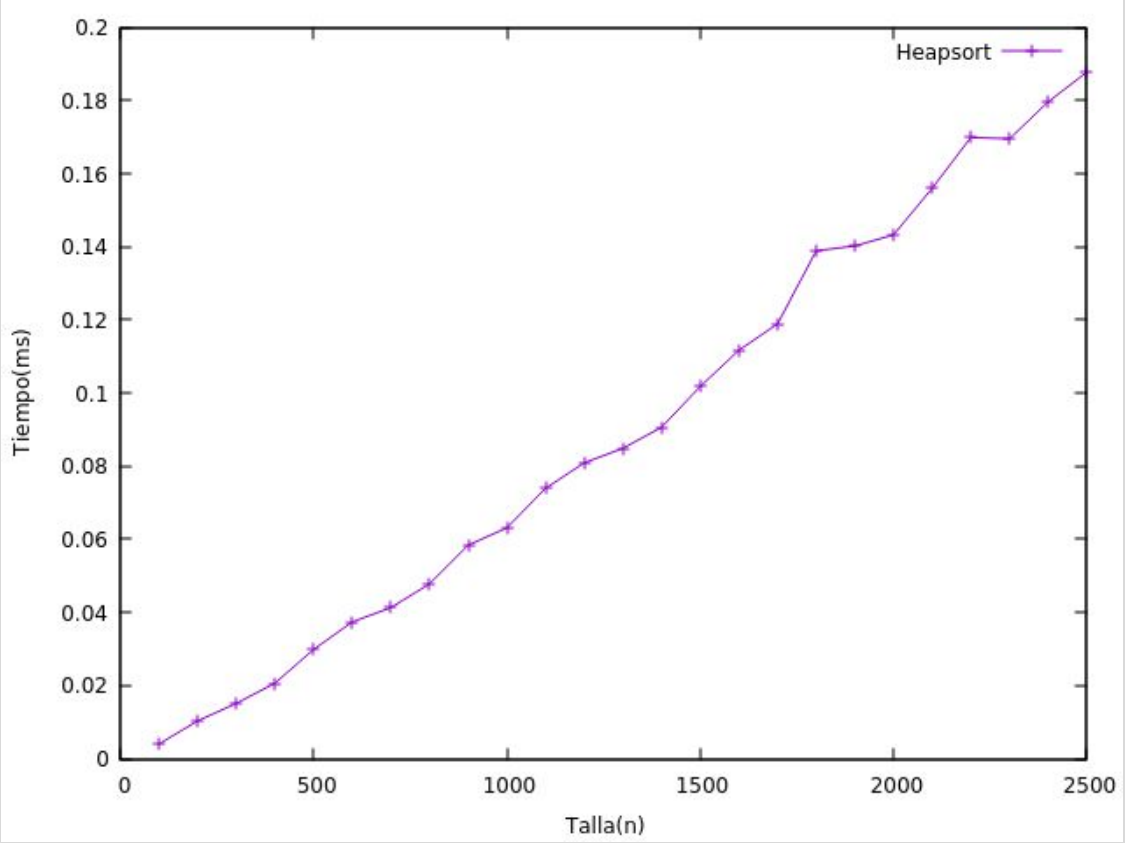
$$a = 9.74125e\text{-}06$$

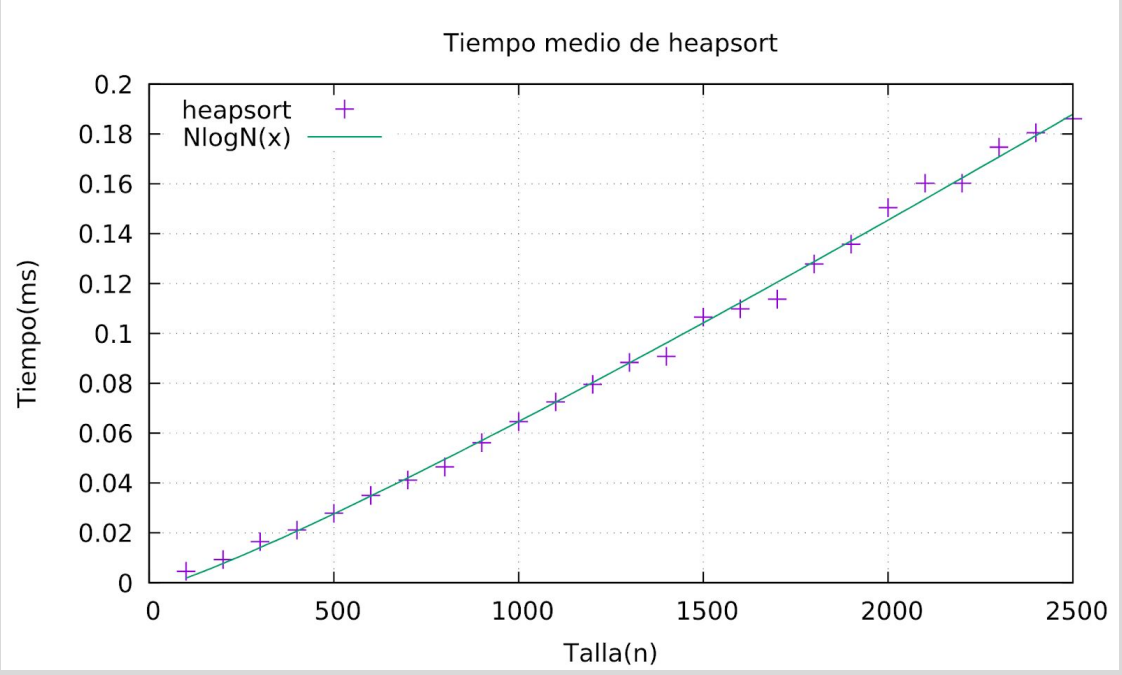
$$b = -0.00262128$$

Lo cual nos da una una función ajustada tal que:

$$f(x) = 9.74125e\text{-}06 * x * \log(x) + -0.00262128$$

Con un porcentaje de error de la constante de mayor peso del 1.062%





Algoritmo mergesort

El funcionamiento de mergesort, si bien no tan sencillo como los primeros que hemos estudiado, tampoco presenta especial complejidad.

El vector inicial se divide en dos subvectores de igual tamaño y se llama a la propia función de ordenación de forma recursiva con estos dos subvectores. Ésta división se realiza, en nuestro caso, hasta que los subvectores alcanzan un tamaño `UMBRAL_MS` (de valor 100) a partir del cual se aplica el algoritmo de inserción para ordenarse.

Una vez los subvectores de tamaño menor o igual que 100 están ordenados se llama a una función de merge (fusión) que va comparando los primeros elementos de cada subvector y colocándolos en su posición, aumentando el puntero y continuando hasta que todos los subvectores se han unido formando el vector final ordenado.

Nº elementos	**Tiempo**	**Nº elementos**	**Tiempo**	**Nº elementos**	**Tiempo**
100	0.007811	**900**	0.108175	**1700**	0.212182
200	0.017811	**1000**	0.125988	**1800**	0.240307
300	0.033066	**1100**	0.145682	**1900**	0.25549
400	0.041359	**1200**	0.159881	**2000**	0.269847
500	0.058122	**1300**	0.181561	**2100**	0.291372
600	0.075106	**1400**	0.20546	**2200**	0.300174
700	0.093216	**1500**	0.232792	**2300**	0.325909
800	0.089971	**1600**	0.195653	**2400**	0.341492
				2500	0.361577

Una vez ajustamos la gráfica a T(n) = an*log(n) + b, obtenemos las siguientes constantes ocultas:

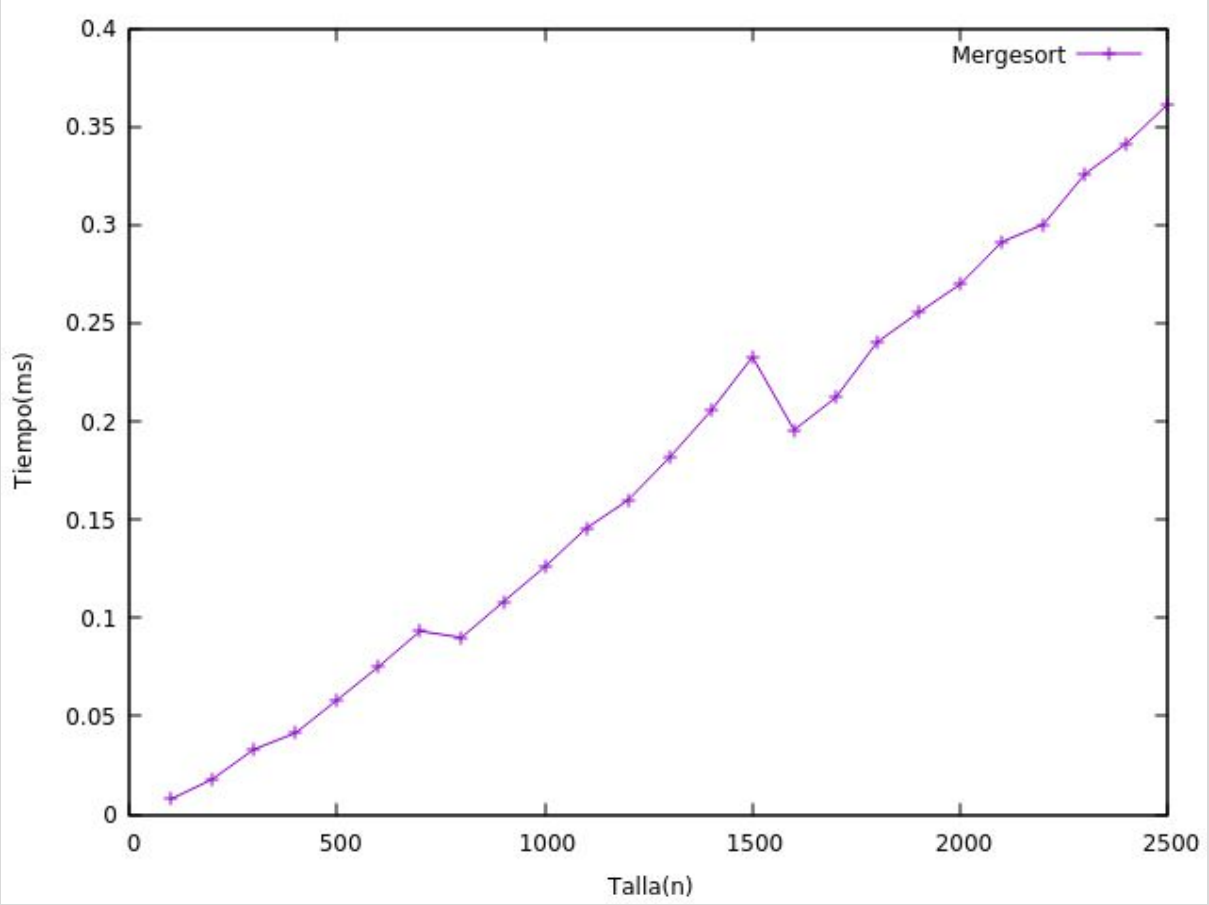
a = 1.77511e-05

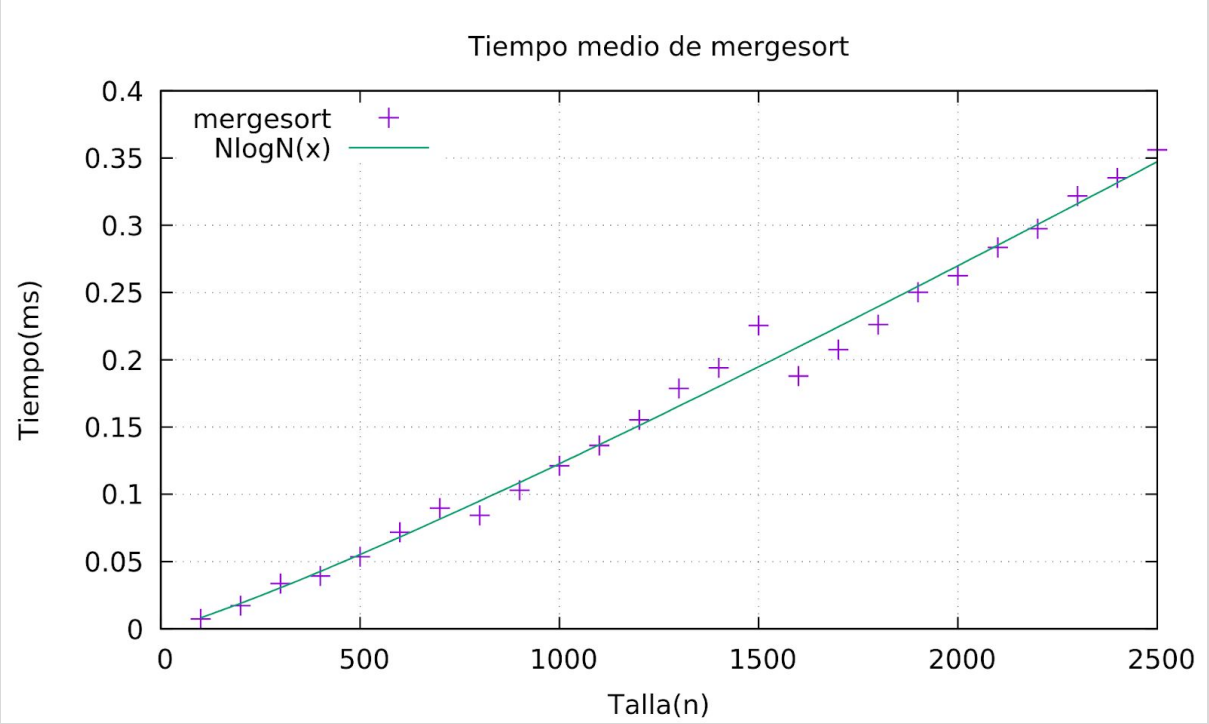
b = 0.000102283

Lo cual nos da una una función ajustada tal que:

f(x) = 1.77511e-05 * x * log(x) + 0.000102283

Con un porcentaje de error de la constante de mayor peso del 2.112%


Mergesort
Tiempo(ms)
Talla(n)
0
0.05
0.1
0.15
0.2
0.25
0.3
0.35
0.4
0
500
1000
1500
2000
2500


Tiempo medio de mergesort
mergesort
NlogN(x)
Tiempo(ms)
Talla(n)
0
0.05
0.1
0.15
0.2
0.25
0.3
0.35
0.4
0
500
1000
1500
2000
2500

Algoritmo quicksort

El algoritmo de Quicksort es, a efectos prácticos, el más rápido. Teóricamente es menos deseable que mergesort y heapsort, ya que en el peor caso tiene una complejidad de O(n^2). Sin embargo, la diferencia en constantes ocultas y la probabilidad de que se dé el peor caso, hace que éste algoritmo proporcione mejores resultados.

El funcionamiento es algo complejo y hay numerosas variaciones dependiendo del factor clave de éste algoritmo: el pivote. Para ordenar el vector decidimos mediante un criterio un elemento pivote (en nuestro caso el primer elemento del vector, aunque el sistema con mejores resultados hasta el momento consiste en escoger el mayor de los dos primeros elementos), a partir del cual deberemos cumplir la regla de que todo elemento a la izquierda del pivote debe ser menor y todo elemento a la derecha debe ser mayor a éste.

Esto se realiza mediante llamadas recursivas al propio algoritmo Quicksort. Podríamos decir que es parecido al algoritmo mergesort, pero que en lugar de escoger la mitad del vector para dividir, elegimos un elemento pivote.

Nº elementos	**Tiempo**	**Nº elementos**	**Tiempo**	**Nº elementos**	**Tiempo**
100	0.004242	**900**	0.056861	**1700**	0.13482
200	0.009852	**1000**	0.081431	**1800**	0.13098
300	0.015511	**1100**	0.070252	**1900**	0.134696
400	0.021037	**1200**	0.078668	**2000**	0.145841
500	0.027884	**1300**	0.086316	**2100**	0.158092
600	0.035644	**1400**	0.095945	**2200**	0.163137
700	0.04113	**1500**	0.104298	**2300**	0.172933
800	0.051028	**1600**	0.121297	**2400**	0.176786
				2500	0.189538

Una vez ajustamos la gráfica a T(n) = an*log(n) + bn, obtenemos las siguientes constantes ocultas:

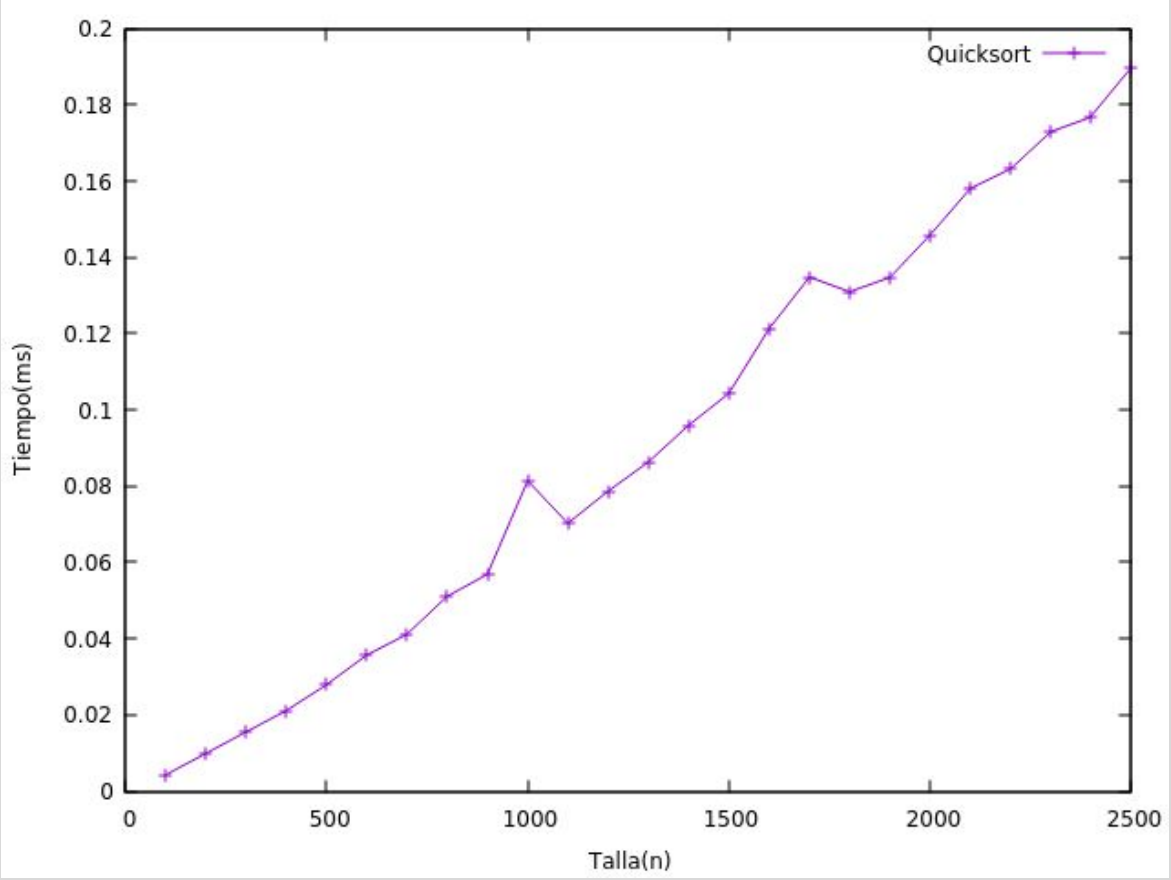
$$a = 9.66191e\text{-}06$$

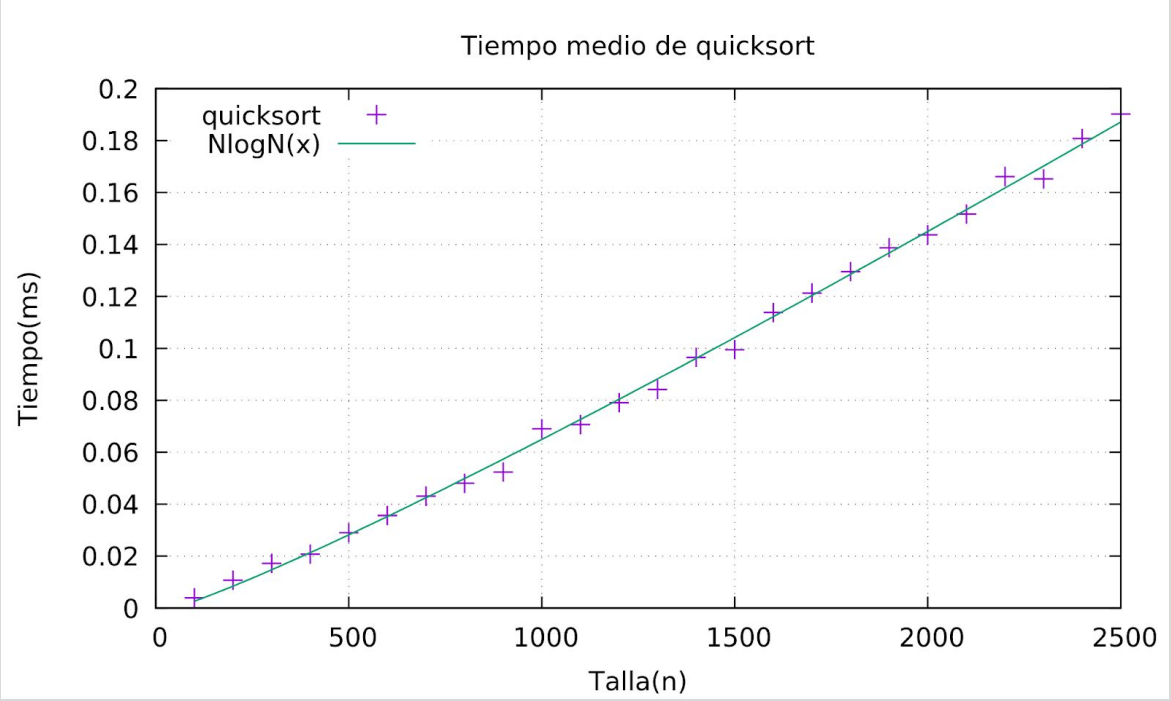
$$b = -0.00182033$$

Lo cual nos da una una función ajustada tal que:

$$f(x) = 9.66191e\text{-}06 * x * \log(x) - 0.00182033$$

Con un porcentaje de error de la constante de mayor peso del 0.9785%





2.2. Algoritmo de Floyd

Este algoritmo encuentra el camino mínimo entre dos vértices de un grafo ponderado sea dirigido o no.

En primer lugar crea una matriz que representa los distintos vértices y los valores de ponderación que los separan si tienen una arista de unión directa.

Una vez la matriz está rellena, se suman los valores de ponderación de realizar un determinado camino y se compara con el valor que se tenía anteriormente almacenado (el valor de realizar el camino directo, que a veces puede ser imposible de realizar). Si este valor es más favorable, se almacena en la matriz y se almacenaría también en una matriz el vértice por el que pasamos durante este recorrido, que podríamos usar más adelante para comprobar qué camino tomar si fuésemos a aplicar el algoritmo en un caso real, pero que en éste caso no es necesario y con la propia matriz de valores es suficiente.

Cabe mencionar que el algoritmo de Floyd no calcula sólo el camino mínimo entre dos vértices concretos, sino que calcula todos los caminos mínimos entre los posibles vértices de un grafo. Una vez realizado se puede utilizar el resultado para saber el camino mínimo entre cualquier par de vértices.

Nº elementos	**Tiempo**	**Nº elementos**	**Tiempo**	**Nº elementos**	**Tiempo**
50	0.0006345	**250**	0.0749062	**450**	0.518842
75	0.0020867	**275**	0.10045	**475**	0.513772
100	0.0049522	**300**	0.142798	**500**	0.709699
125	0.0095836	**325**	0.163229	**525**	0.684459
150	0.0163957	**350**	0.204542	**550**	0.870056
175	0.0258959	**375**	0.292163	**575**	1.05451
200	0.0381568	**400**	0.309632	**600**	1.22651
225	0.0607625	**425**	0.422578	**625**	1.36603
				650	1.52189
				675	1.73552

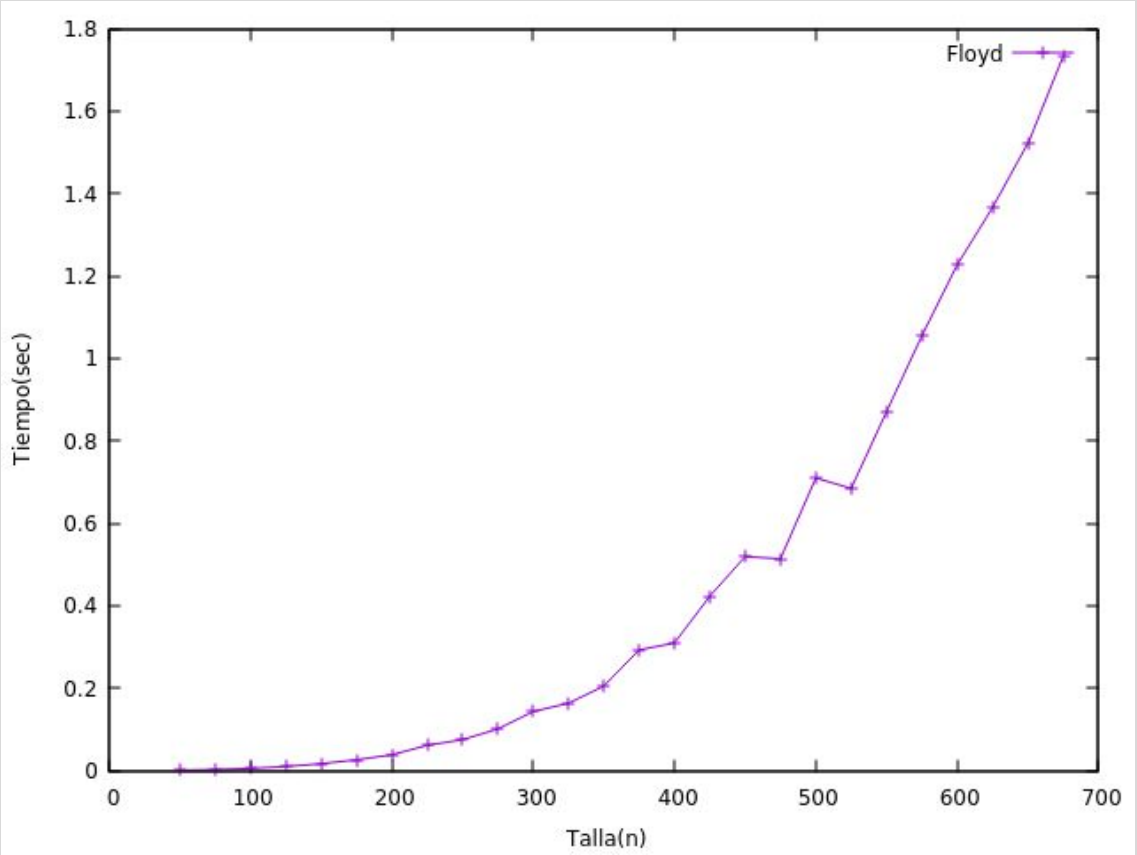
Una vez ajustamos la gráfica a $T(n) = an^3+bn^2+cn+d$, obtenemos estas constantes ocultas:

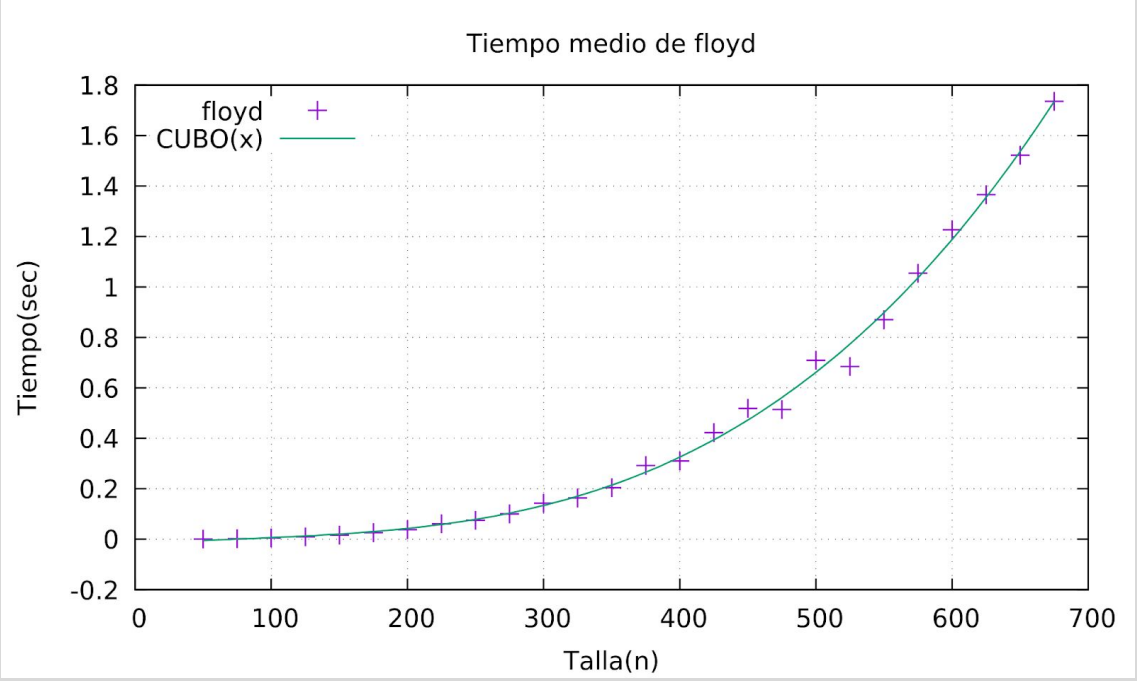
a = 5.57373e-09
b =1.80543e-07
c = -0.00010309
d = 0.00654602

Lo cual nos da una una función ajustada tal que:

$$f(x) = 5.57373e\text{-}09*x^3 + 1.80543e\text{-}07*x^2 - 0.00010309*x + 0.00654602$$

Con un porcentaje de error de la constante de mayor peso del 20.11%





2.3. Algoritmo de Hanoi

El algoritmo de Hanoi resuelve el juego o problema de las torres de Hanoi, en el que, teniendo una pila de discos sobre una columna, se desea pasar la totalidad de la pila a otra torre sin que en ningún momento un disco tenga encima uno mayor y moviendo los discos de uno en uno.

En nuestro caso este algoritmo resuelve el problema de forma recursiva. La función se llama a sí misma proveyendo el número de discos que se desean mover la columna en la que están dichos discos y la columna a la cual se desean llevar.

De esta manera se baja en complejidad hasta llegar a un caso simple en el que se aplica siempre la misma regla. La cantidad mínima de movimientos necesarios es siempre (2^n)-1, es decir: para mover una pila de 5 discos, necesitaremos (2^5)-1 movimientos = 31.

Nº elementos	**Tiempo**	**Nº elementos**	**Tiempo**	**Nº elementos**	**Tiempo**
5	2.1e-06	**15**	0.0015055	**25**	1.62005
6	3.1e-06	**16**	0.0030212	**26**	3.32738
7	6e-06	**17**	0.0061903	**27**	6.67983
8	1.16e-05	**18**	0.0122282	**28**	12.6251
9	2.32e-05	**19**	0.0248997	**29**	25.1894
10	4.92e-05	**20**	0.0500386	**30**	49.8541
11	0.0001033	**21**	0.101272		
12	0.0001853	**22**	0.206804		
13	0.0003857	**23**	0.398619		
14	0.0007541	**24**	0.817935		

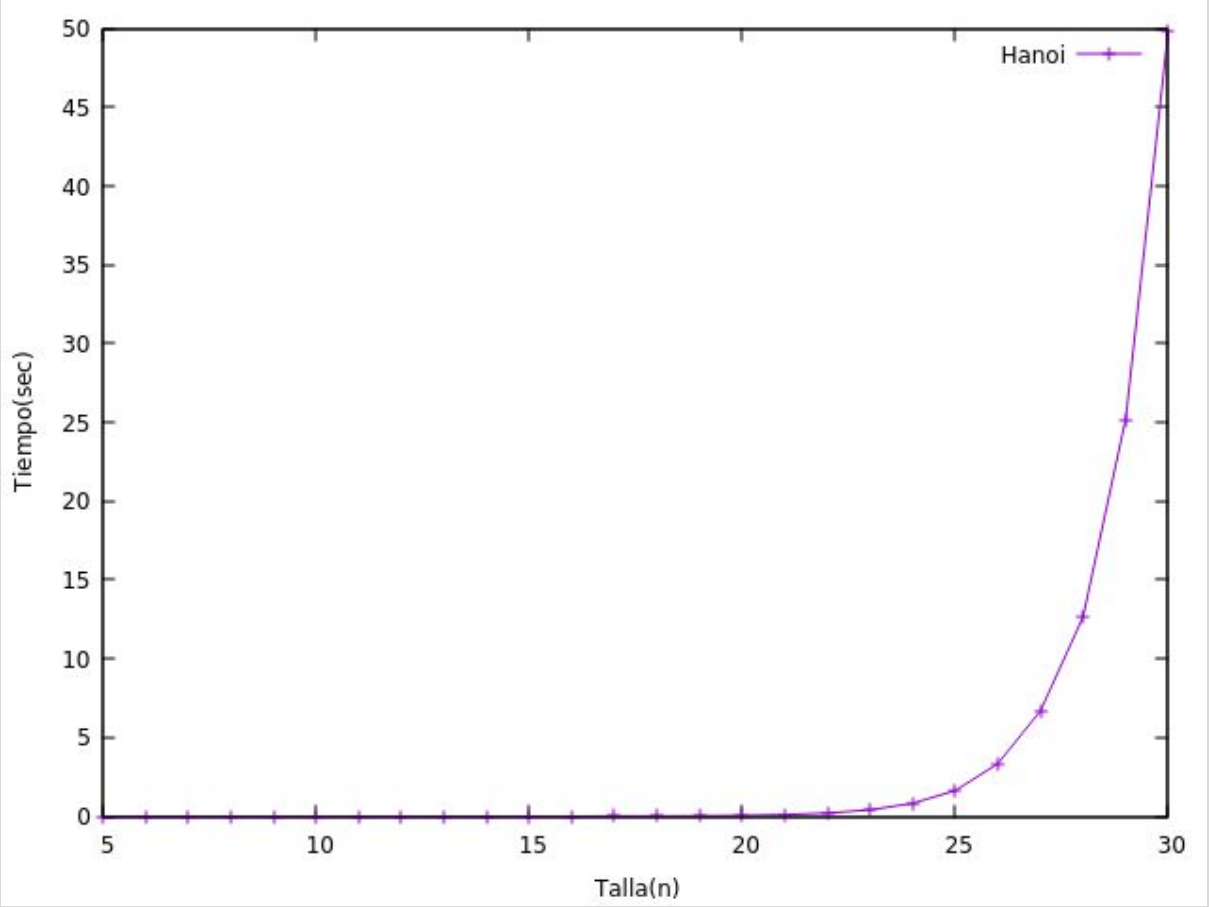
Una vez ajustamos la gráfica a $T(n) = a*2^n$, obtenemos la variable oculta:

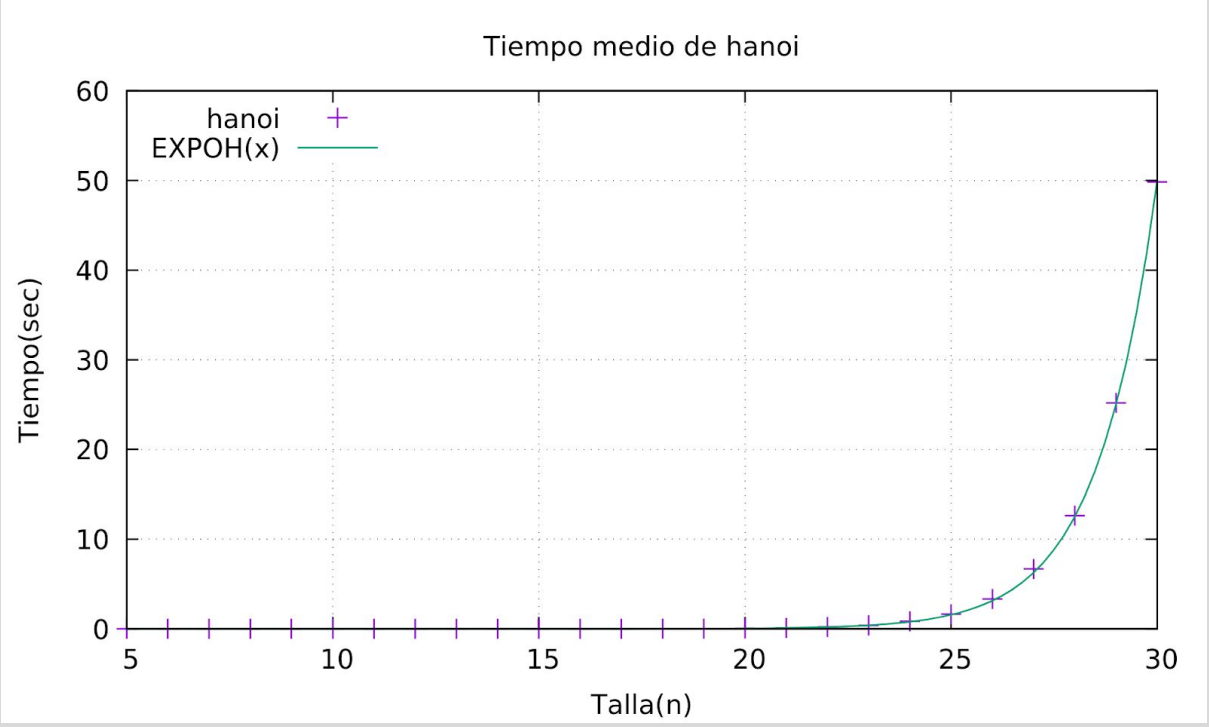
$$a = 4.61553e\text{-}08$$

Lo cual nos da una una función ajustada tal que:

$$f(x) = 4.61553e\text{-}08 * 2^n$$

Con un porcentaje de error de la constante de mayor peso del 0.04097%


Hanoi
Tiempo(sec)
Talla(n)
0
5
10
15
20
25
30
35
40
45
50
5
10
15
20
25
30


Tiempo medio de hanoi
hanoi
EXPOH(x)
Tiempo(sec)
Talla(n)
0
10
20
30
40
50
60
5
10
15
20
25
30

3. Análisis de optimización

Otro ejercicio propuesto en esta práctica era la medición de la eficiencia empírica afectada por parámetros externos, como la optimización en la compilación del código. Al compilar código C o C++ con gcc o g++, podemos hacer uso de la flag -O, que afecta a cómo se genera el código ensamblador derivado de nuestro código original.

Para este ejercicio, se ha hecho la prueba con las siguientes flags de compilación:

- ➔ `g++ codigo.cpp -o codigoobj`
  Sin optimización alguna

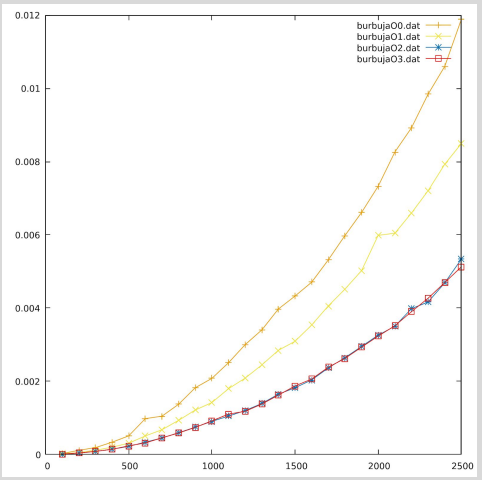
- ➔ `g++ -O1 codigo.cpp -o codigoobj`
  El nivel más básico de optimización. Se intenta producir un código rápido y pequeño sin tomar mucho tiempo de compilación

- ➔ `g++ -O2 codigo.cpp -o codigoobj`
  El nivel recomendado de optimización. Se aumenta el rendimiento del código (respecto a -O1) sin comprometer el tamaño y sin tomar mucho más tiempo de compilación

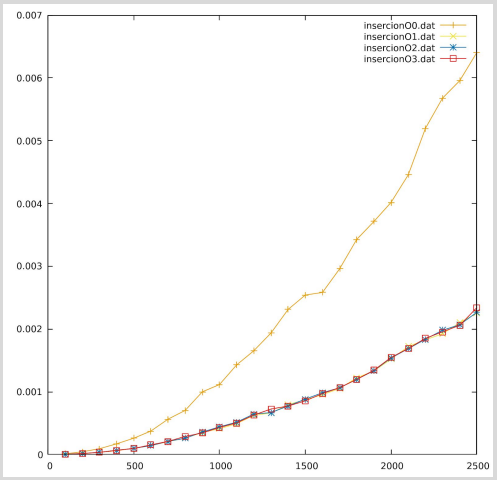
- ➔ `g++ -O3 codigo.cpp -o codigoobj`
  El nivel más alto de optimización posible. Emplea optimizaciones que son caras en términos de tiempo de compilación y uso de memoria.
  El hecho de compilar con -O3 no garantiza una mejora del rendimiento ya que (en muchos casos) los ficheros son tan grandes que ralentizan el sistema

Cada algoritmo se compiló con las anteriores opciones de optimización en un sistema *Arch Linux 64 bit* con un procesador *Intel Core i5-8250U CPU @ 1.60GHz.*

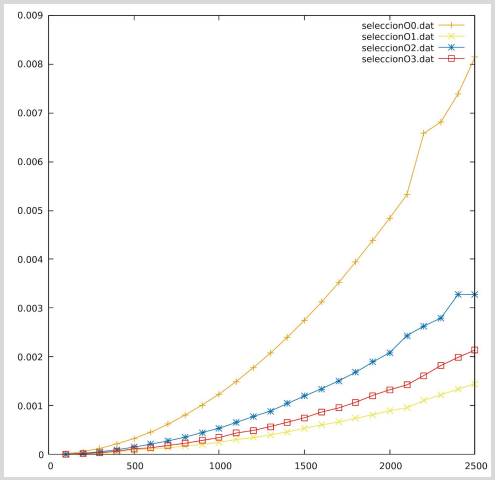
3.1. Algoritmos O(n²)



Burbuja



Inserción

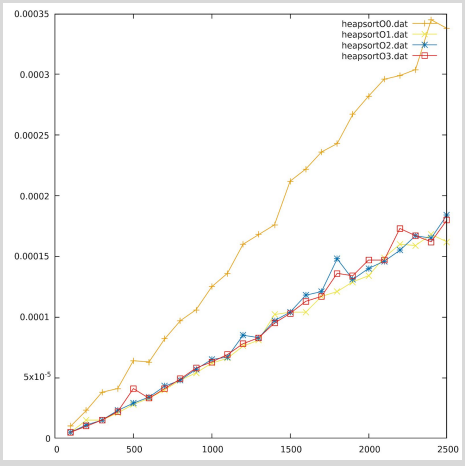


Selección

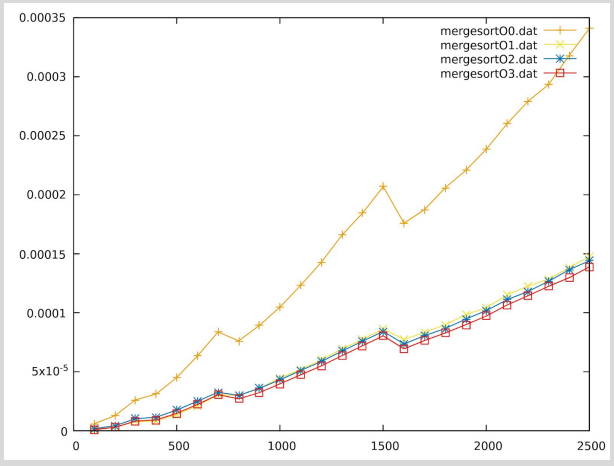
Las gráficas obtenidas se corresponden por lo general bastante bien con la descripción teórica dada anteriormente: es obvio que la flag -O0 es la que nos ofrece peor rendimiento, pero la diferencia entre las flag -O2 y -O3 suele ser pequeña, si no inexistente. Una mayor optimización del código ensamblador no tiene por qué mejorar considerablemente los tiempos de ejecución.

Cabe destacar el caso del algoritmo de selección, en el cual la optimización -O1 es la superior: esto nos lleva a pensar que la forma en la que se optimiza el código ensamblador en -O1 es, simplemente, más adecuada que otras optimizaciones para el código de selección. Recordemos que la optimización es una solución **generalista** a la hora de traducir a ensamblador nuestro código de alto nivel, no nos garantiza nada.

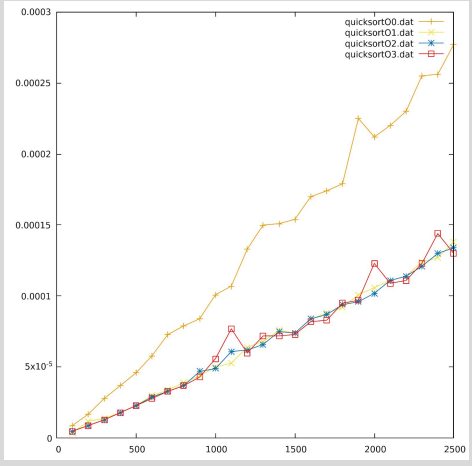
3.2. Algoritmos O(nlog(n))



Heapsort



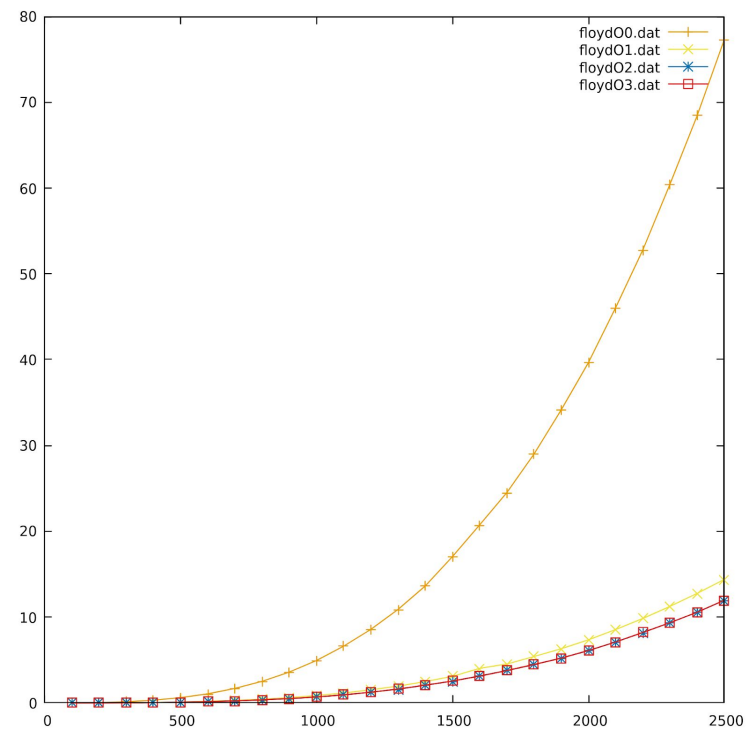
Mergesort



Quicksort

En los algoritmos logarítmicos nos encontramos que, de nuevo, la flag -O0 nos da los peores tiempos; pero que la diferencia entre los distintos niveles de optimización es mucho menor que en los cuadráticos.

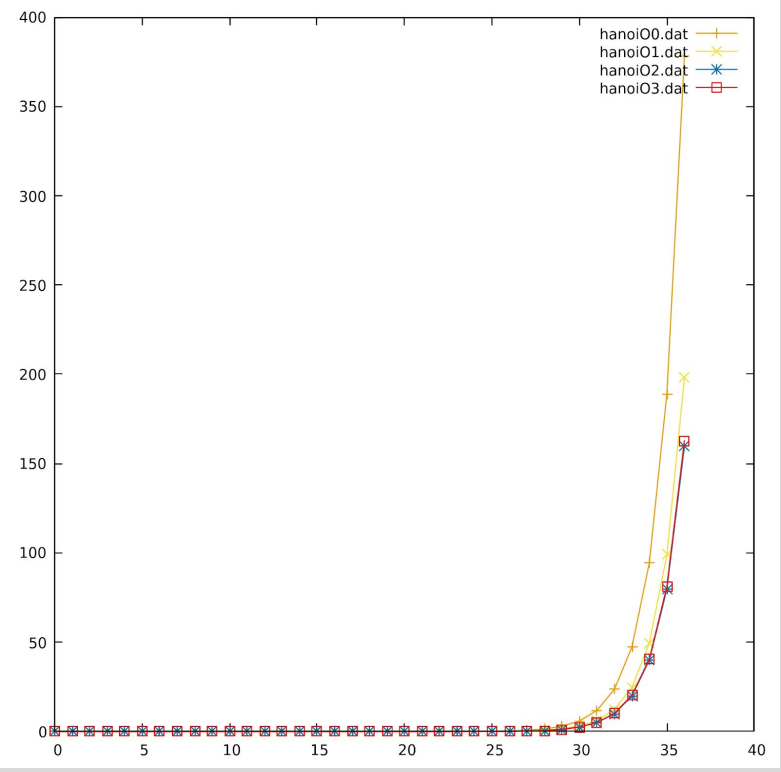
3.3. Algoritmo O(n³)



En el caso del algoritmo de Floyd, la diferencia es extremadamente notable respecto a con-sin optimización.

La versión de -O0 prácticamente multiplica por 8 el tiempo de ejecución

3.4. Algoritmo O(2^n)

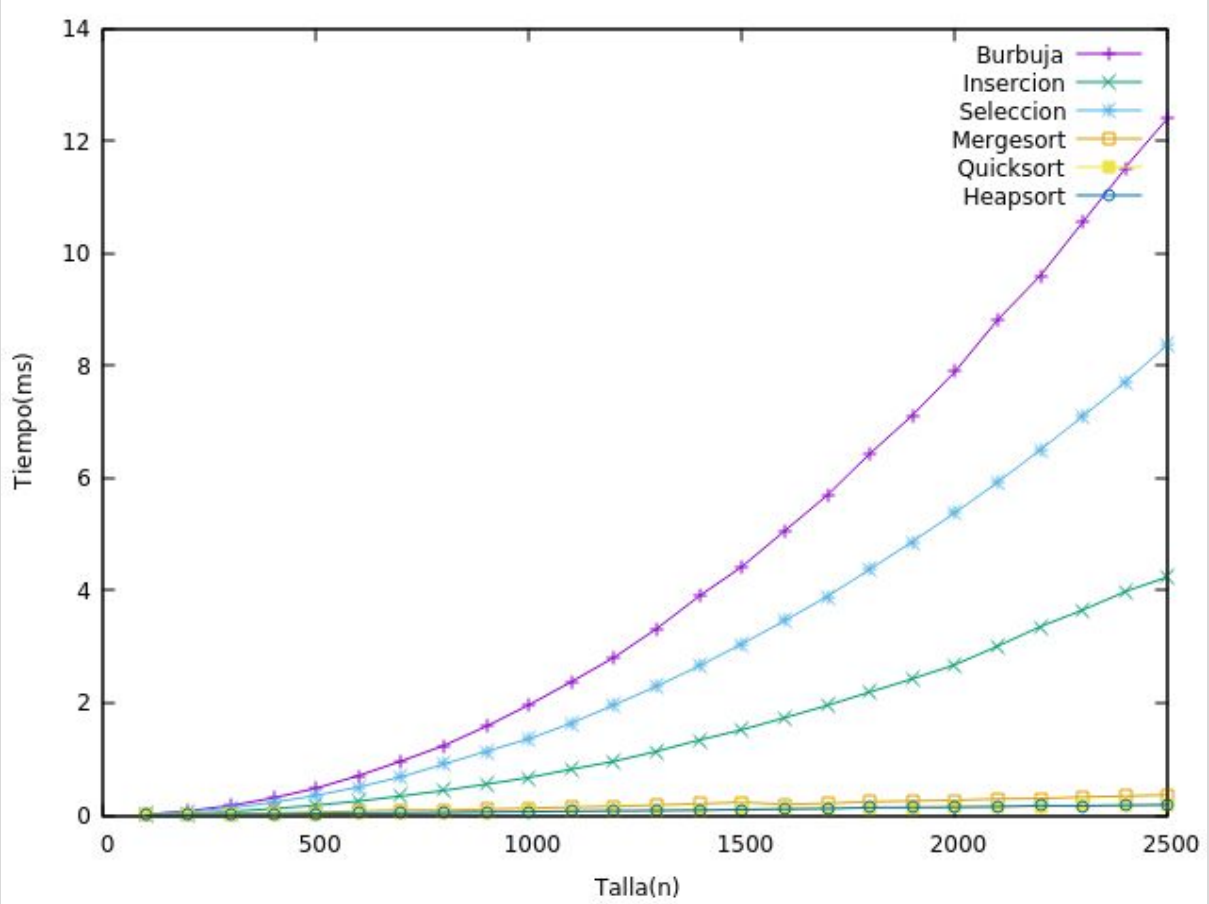


En Hanoi, aunque la diferencia no sea tan acentuada inicialmente, la propia naturaleza del algoritmo 2^n hace que el tiempo de ejecución se duplique en las últimas iteraciones

4. Comparativas

Estos cálculos se realizaron en un sistema *Windows 10 64 bits* con un procesador *AMD Ryzen 5 1600X Six-Core Processor 3.60 GHz.*

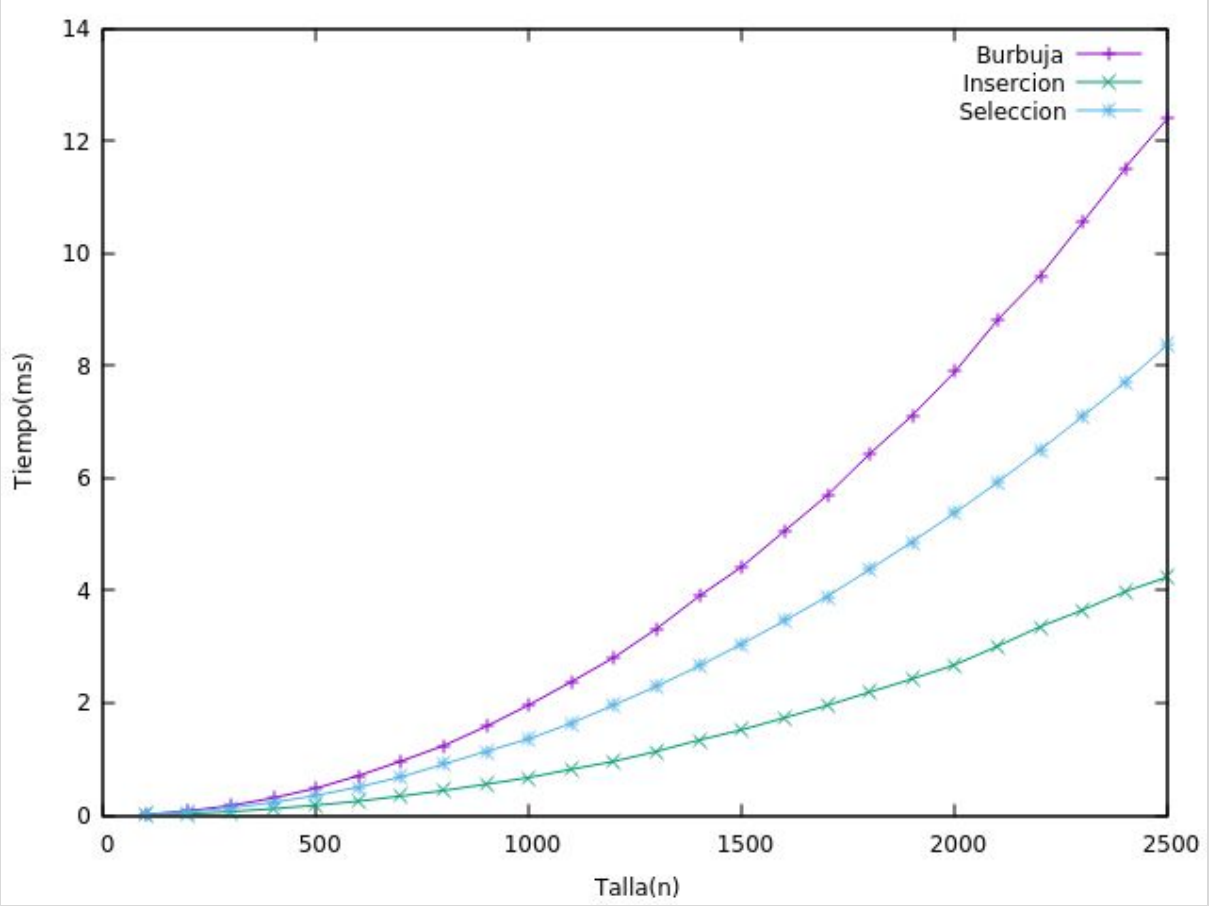
4.1. Algoritmos de ordenación



En la gráfica están presentes los seis algoritmos de ordenación: los tres de complejidad cuadrática primero (burbuja, inserción y selección) y los tres de complejidad logarítmica después (mergesort, quicksort y heapsort).

Podemos ver claramente como los tres algoritmos cuadráticos despuntan según el tamaño aumenta, mientras que los algoritmos logarítmicos siguen muy pegados al eje x, casi sin cambios en el tiempo aunque el tamaño de la prueba vaya aumentando.

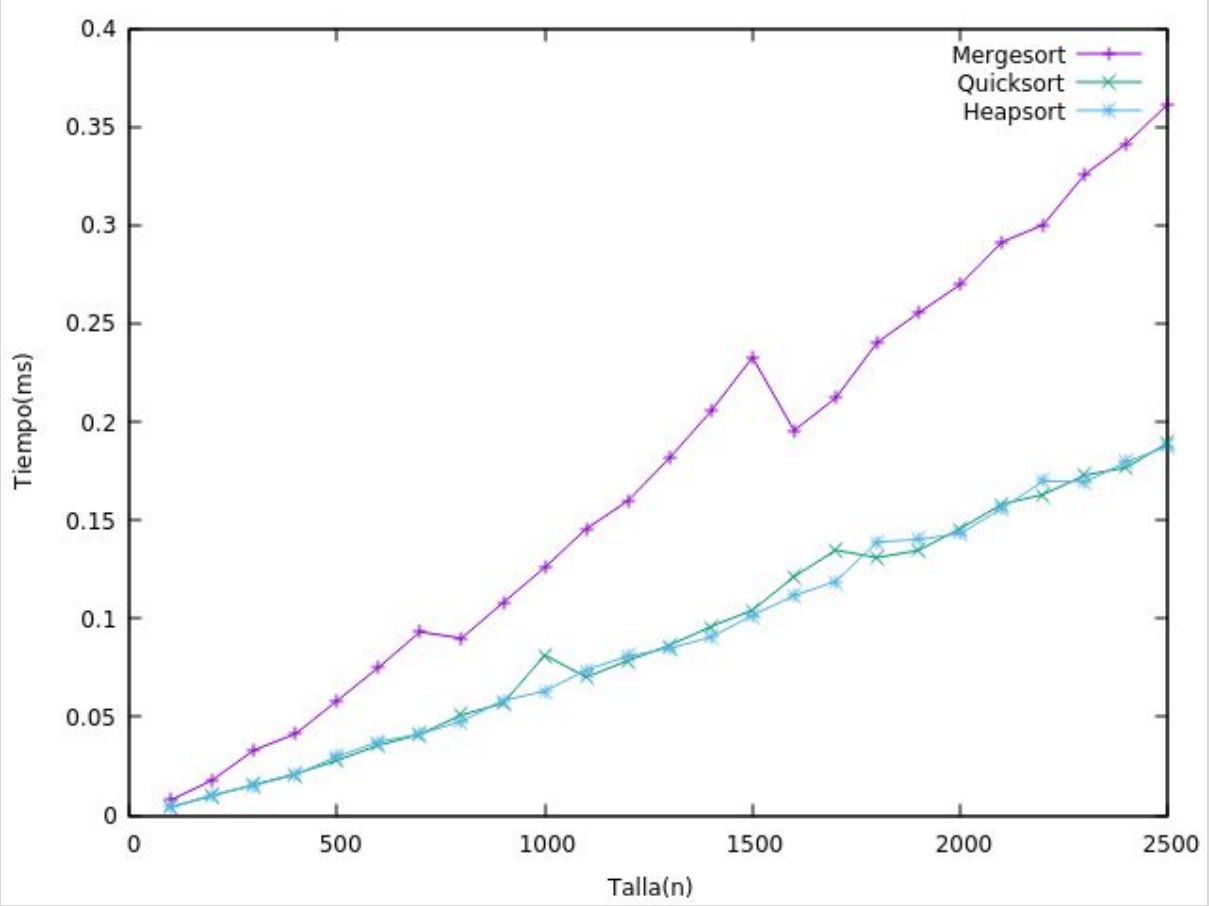
4.2. Algoritmos O(n²)



En la gráfica podemos ver los tres algoritmos de ordenación de complejidad cuadrática (burbuja, inserción y selección).

Es curioso ver que, aunque teóricamente los tres son de complejidad cuadrática, al hacer el estudio empírico hay diferencias notables entre ellos, siendo el peor algoritmo el de burbuja, muy lejos de la eficacia del de inserción.

4.3. Algoritmos O(nlog(n))



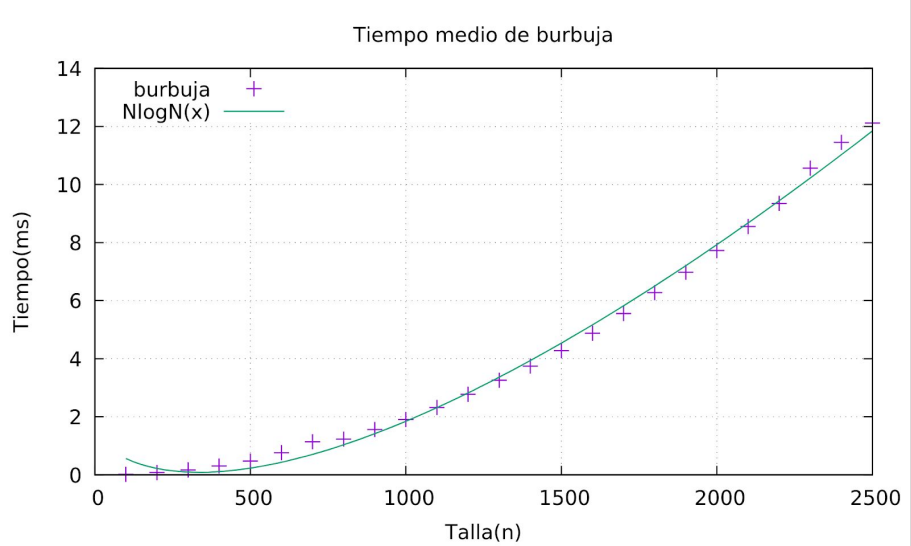
Aquí podemos ver algo parecido a lo que pasaba con anterioridad: tenemos los tres algoritmos de ordenación de complejidad n*log(n) (mergesort, quicksort y heapsort).

Aunque también teóricamente tienen la misma complejidad, podemos ver diferencias entre ellos, sobre todo al comparar empíricamente mergesort con los otros dos, el cual parece mucho menos eficiente en tiempo. Sin embargo, quicksort y heapsort prácticamente van a la par.

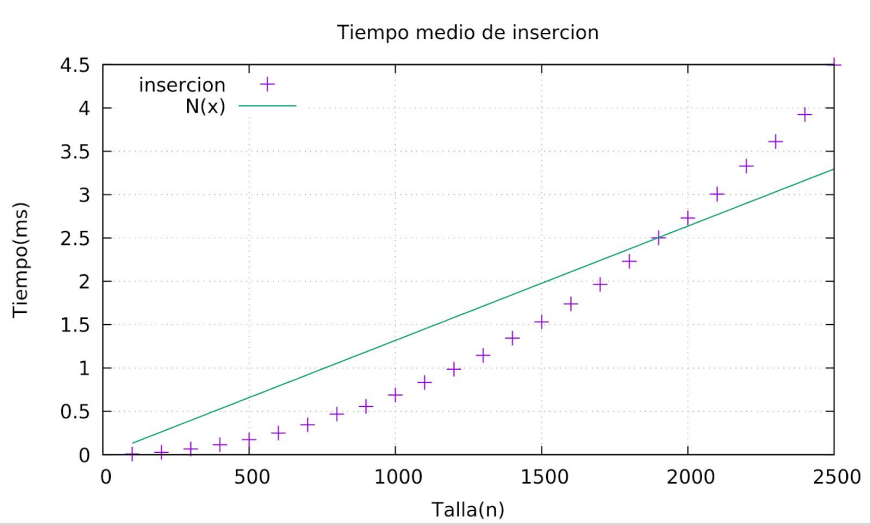
5. Ajuste erróneo

Otro ejercicio propuesto en el guión de prácticas era emplear la función de gnuplot `fit` con una función f(x) que no se correspondiera a la T(n) del algoritmo graficado.

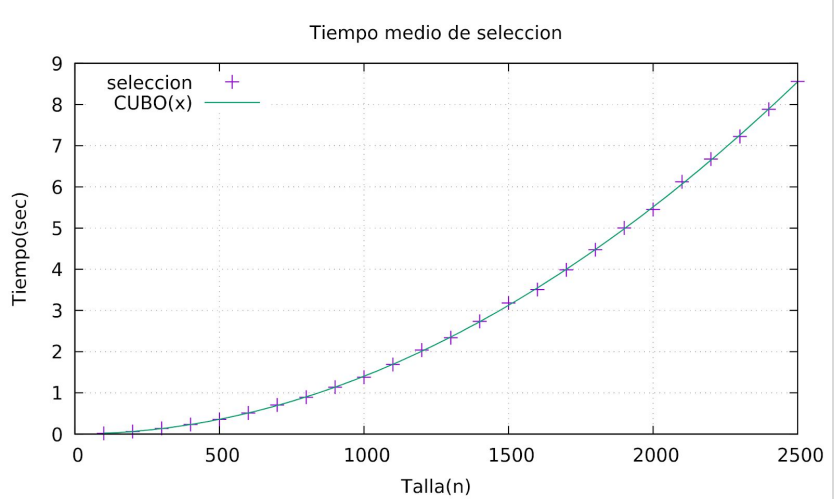
5.1. Burbuja con ajuste n*log(n)



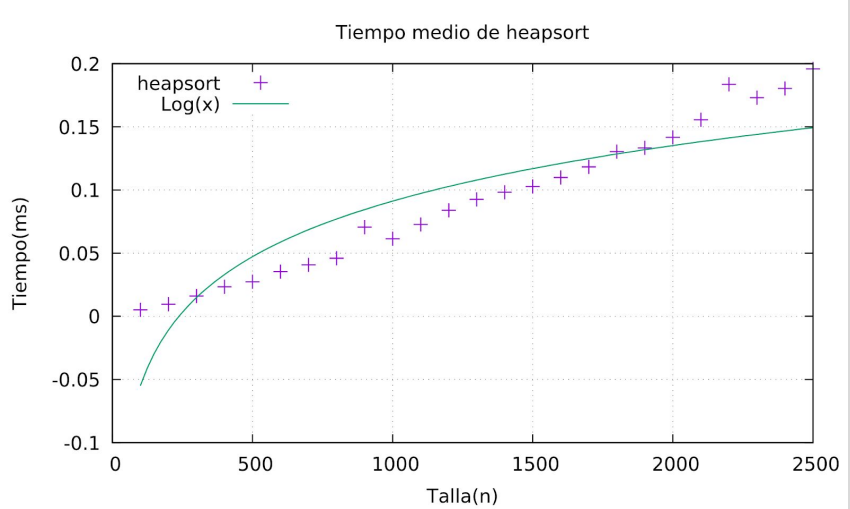
5.2.Inserción con ajuste n



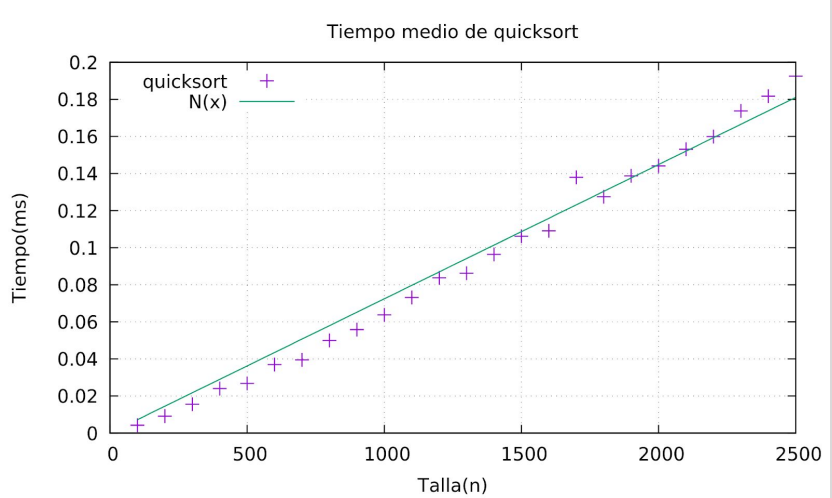
5.3. Selección con ajuste n^3



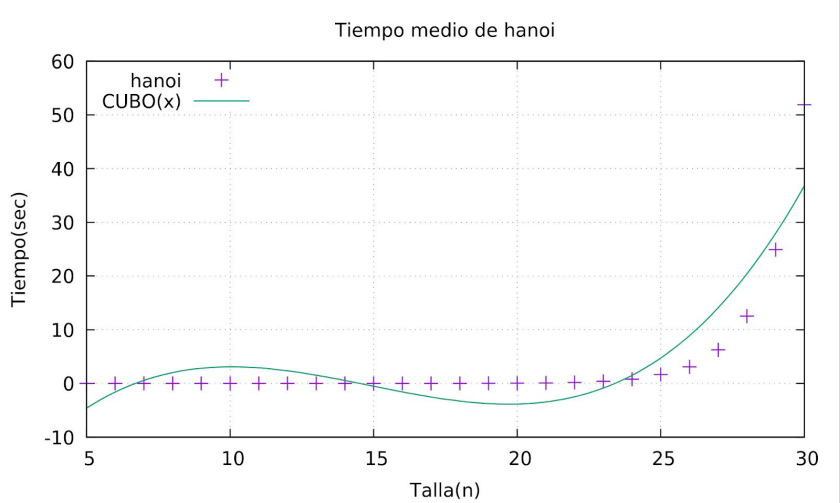
5.4. Heapsort con ajuste log(n)



5.4. Quicksort con ajuste n



5.5. Hanoi con ajuste n³



6. Conclusiones

Las conclusiones que inferimos de esta práctica son las siguientes:

- La eficiencia empírica y teórica se corresponden y es fácilmente comprobable. Al compararlas en forma de gráfica no existen diferencias notables más allá de perturbaciones externas a la eficiencia, como la distribución inicial del vector aleatorio, interrupciones ocasionadas por el sistema operativo o por la máquina empleada.
  Aún así, la diferencia que podemos apreciar entre distintas máquinas es limitada y siempre se ajusta a la eficiencia teórica

- La optimización de código proporcionada por gcc o g++ no deja de ser una forma de traducir código de alto nivel a ensamblador: mayores valores de optimización no garantizan un código más rápido, y en niveles muy altos (-O3) puede llegar a ser contraproducente.
  Hemos visto casos (como el algoritmo de selección) en los que optimizaciones menores mejoraron notablemente el tiempo de ejecución del código.

- Ajustar los datos de un algoritmo a una eficiencia que no es la suya, dependiendo del algoritmo y su eficiencia, puede tener un ajuste inesperadamente bueno. Esto es particularmente cierto en valores pequeños como los que hemos empleado en esta memoria de práctica (de 100 a 2500 en intervalos de 100).