Práctica 1:
Eficiencia de los algorítmos.

Bedmar López, Pedro
Escalona García, Alejandro
España Sánchez, Joaquín Alejandro
García Venegas, Joaquín

GRUPO G2-C1

Índice:

1. Planteamiento del problema

En esta práctica se pretende abordar la comparación de eficiencia entre algoritmos. Ya que en la parte teórica de la asignatura se está estudiando eficiencia teórica, en esta parte de prácticas nos centraremos en mayor medida en la empírica.

Para ello, vamos a comparar la eficiencia de una serie de algoritmos implementados en C++ utilizando entradas de diferentes tamaños. Calcularemos los tiempos de ejecución de cada uno de los algoritmos para cada tamaño y también seguiremos los demás pasos descritos en el índice, comprobando que la eficiencia empírica se ajusta a la teórica.

2. Tablas con eficiencia empírica de algunos algoritmos

Utilizando la implementación de los algoritmos proporcionada por los profesores, calcularemos el tiempo que tardan en ejecutarse gracias al reloj del sistema. Este reloj es capaz de capturar en una variable el momento temporal actual, y si realizamos una medición al inicio y otra al final de la ejecución del algoritmo y restamos la final menos la inicial, conseguimos la duración. Para transformarla a segundos la dividimos entre la constante CLOCKS_PER_SEC.

Hemos elegido los algoritmos ordenación por burbuja, mergesort, Floyd y el de las torres de Hanoi para calcular sus tiempos de ejecución. Para facilitar el trabajo, hemos compilado todos los ficheros .cpp donde se implementan los algoritmos y hemos utilizado la macro de terminal facilitada por los profesores para generar todos los datos de cada tabla con una sola sentencia.

En algunos casos, vamos a realizar varios cálculos de tiempo, para así obtener un tiempo promedio y eliminar posibles valores atípicos en las gráficas. Estos valores pueden producirse porque en la máquina se ejecutan múltiples procesos al mismo tiempo, y quizás en un momento dado se ha colado algún proceso con más prioridad y esto ha provocado una bajada en el rendimiento del algoritmo. En otros casos puede deberse a factores relacionados con la aleatoriedad de los números: hay algoritmos que tardan más o menos según la entropía de los datos, y eso puede afectar al tiempo de ejecución. Promediando la solución reducimos el impacto de estos factores.

Burbuja -> O(n^2)

Es un sencillo algoritmo de ordenamiento. Funciona revisando cada elemento de la lista que va a ser ordenada con el siguiente, intercambiándolos de posición si están en el orden equivocado. Es necesario revisar varias veces toda la lista hasta que no se necesiten más intercambios, lo cual significa que la lista está ordenada. Este algoritmo obtiene su nombre de la forma con la que suben por la lista los elementos durante los intercambios, como si fueran pequeñas burbujas. También es conocido como el método del intercambio directo. Dado que solo usa comparaciones para operar elementos, se lo considera un algoritmo de comparación, siendo el más sencillo de implementar.

```
static void burbuja_lims(int T[], int inicial, int final)
{
  int i, j;
  int aux;
  for (i = inicial; i < final - 1; i++)
    for (j = final - 1; j > i; j--)
      if (T[j] < T[j-1])
  {
    aux = T[j];
    T[j] = T[j-1];
    T[j-1] = aux;
  }
}
```

Como se observa, al existir dos bucles anidados para llevar a cabo la ejecución del algoritmo, se deduce fácilmente que la eficiencia es **O(n^2)**.

En este ejercicio hemos aplicado la técnica mencionada en el apartado 1 para promediar el resultado. Este promedio lo hemos llevado a cabo mediante la varianza y la media, para analizar cual se ajusta mejor al resultado que queremos obtener.

EJECUCIONES DEL ALGORITMO DE BURBUJA

TAMAÑO	EJEC 1	EJEC 2	EJEC 3	EJEC 4	EJEC 5	TAMAÑO	MEDIANA	TAMAÑO	MEDIA
1000	0,003897	0,002495	0,002466	0,001909	0,002518	1000	0.0025065	1000	0.002657
5500	0,066756	0,086558	0,092713	0,065804	0,087999	5500	0.0872785	5500	0.079966
10000	0,241772	0,333284	0,333682	0,252711	0,247448	10000	0.252711	10000	0.2817794
14500	0,533285	0,751523	0,735426	0,560312	0,554625	14500	0.560312	14500	0.6270342
19000	0,973972	0,970548	0,986573	1,00458	0,981246	19000	0.992913	19000	0.9833838
23500	1,47246	1,52793	1,49613	1,54711	1,53942	23500	1.543265	23500	1.51661
28000	2,1841	2,26283	2,16052	2,23302	2,19255	28000	2.212785	28000	2.206604
32500	2,89517	2,97743	3,04642	3,08804	2,96499	32500	3.032735	32500	2.99441
37000	3,9351	4,02055	3,84711	5,04225	3,93137	37000	4.488675	37000	4.155276
41500	4,82877	4,90773	4,88288	6,4826	4,93924	41500	5.71092	41500	5.208244
46000	6,26376	6,47019	6,11051	6,00533	6,5968	46000	6.43028	46000	6.289318
50500	9,61476	9,29909	9,16529	7,60487	9,71111	50500	9.5051	50500	9.079024
55000	8,65288	8,72626	11,5101	9,1077	8,80501	55000	8.956355	55000	9.36039
59500	10,1817	10,3084	10,2574	13,4606	10,4011	59500	11.93085	59500	10.92184
64000	14,1593	14,53	12,3087	11,8177	14,6062	64000	14.38275	64000	13.48438
68500	13,8723	13,7134	16,1902	14,5502	13,7359	68500	14.21125	68500	14.4124
73000	17,3367	17,601	15,5853	20,5519	18,017	73000	20.5519	73000	17.81838
77500	18,9314	22,3095	22,3952	17,4516	17,7727	77500	18.35205	77500	19.77208
82000	23,239	20,0476	19,6693	24,267	24,0724	82000	24.0724	82000	22.25906
86500	27,5842	26,3786	26,6651	22,5402	22,4587	86500	24.4594	86500	25.12536
91000	26,2324	27,2209	25,0588	29,2891	28,3277	91000	28.8084	91000	27.22578
95500	28,8818	28,3599	30,496	31,1191	30,2422	95500	30.68065	95500	29.8198
100000	31,2147	39,0352	33,669	32,3103	34,2918	100000	34.2918	100000	34.1042

*Mergesort -> O(n*log(n))*

En este algoritmo de ordenación que aplica la técnica divide y vencerás para mejorar la eficiencia respecto a otros algoritmos de ordenación convencionales. Es un algoritmo recursivo donde el vector se divide en dos partes del mismo tamaño (generalmente), hasta llegar al caso base, que es aquel donde tenemos vectores de tamaño 0 o 1. Dentro del proceso de recursividad, estos subvectores se van recomponiendo de forma ordenada.

Teóricamente es el algoritmo con mejor eficiencia, pero en la práctica quicksort lo supera en muchas ocasiones, ya que las constantes ocultas que multiplican la función son mayores en mergesort. Aquí también aplicamos la mediana a las cinco ejecuciones para conseguir un resultado fiable.

EJECUCIONES DEL ALGORITMO MERGESORT

TAMAÑO	EJEC1	EJEC2	EJEC3	EJEC4	EJEC5			MEDIANA
2000000	0,399695	0,38922	0,400356	0,399327	0,403966			0,399695
2400000	0,47836	0,473798	0,480041	0,497695	0,478826			0,478826
2800000	0,557549	0,562462	0,567306	0,57065	0,570346			0,567306
3200000	0,635717	0,635606	0,645827	0,651084	0,645531			0,645531
3600000	0,743541	0,729556	0,741262	0,74999	0,740124			0,741262
4000000	0,837808	0,819012	0,838032	0,83159	0,855543			0,837808
4400000	0,910536	0,905608	0,921594	0,927977	0,919751			0,919751
4800000	0,990218	0,990681	1,00633	1,00482	1,00988			1,00482
5200000	1,08162	1,07886	1,11533	1,0934	1,12622			1,0934
5600000	1,16114	1,1618	1,19904	1,17755	1,20845			1,17755
6000000	1,24976	1,24896	1,26899	1,26993	1,2782			1,26899
6400000	1,33655	1,34441	1,35792	1,35756	1,36047			1,35756
6800000	1,42352	1,42647	1,45938	1,45551	1,45232			1,45232
7200000	1,50682	1,50862	1,53848	1,54225	1,53589			1,53589
7600000	1,5959	1,60217	1,62606	1,63941	1,63123			1,62606
8000000	1,70719	1,70925	1,74421	1,74045	1,73594			1,73594
8400000	1,79933	1,7969	1,82427	1,85895	1,82937			1,82427
8800000	1,88649	1,88247	1,92883	1,91643	1,91715			1,91643
9200000	1,97947	1,97907	2,01476	2,02809	2,01895			2,01476
9600000	2,05702	2,06119	2,09006	2,08746	2,09123			2,08746
10000000	2,14083	2,14826	2,18145	2,18768	2,1792			2,1792
10400000	2,23828	2,24015	2,28392	2,27977	2,28423			2,27977
10800000	2,32267	2,32079	2,36422	2,369	2,36419			2,36419
11200000	2,41209	2,40523	2,4558	2,45703	2,45832			2,4558
11600000	2,49989	2,50219	2,55268	2,5456	2,54651			2,5456

Floyd -> $O(n^3)$

Rendimiento del algoritmo de Floyd	
Número de nodos	**Runtime**
100	0.009918
200	0.048741
300	0.160568
400	0.325063
500	0.616948
600	1.04604
700	1.6858
800	2.50732
900	3.51525
1000	4.87015
1100	6.58778
1200	8.53515
1300	10.947
1400	13.4565
1500	16.5788
1600	20.1431
1700	24.2614
1800	28.7446
1900	33.7018
2000	39.4295
2100	45.5004
2200	52.4182
2300	60.2006
2400	68.5271
2500	77.4754

Se trata de un algoritmo de análisis de grafos ponderados para encontrar el camino mínimo entre dos vértices cualesquiera.

El algoritmo representa al grafo con una matriz cuadrada de orden n (número de nodos del grafo). El valor de un elemento, por ejemplo, M[i][j] representa el coste de ir desde el nodo i hasta el j, si no están relacionados el valor será infinito. La matriz tendrá una diagonal con ceros, que sería la distancia de un nodo a él mismo.

```
void Floyd(int **M, int dim)
{
  for (int k = 0; k < dim; k++)
    for (int i = 0; i < dim;i++)
      for (int j = 0; j < dim;j++)
        {
    int sum = M[i][k] + M[k][j];
    M[i][j] = (M[i][j] > sum) ? sum : M[i][j];
        }
}
```

Como se observa claramente en el código del algoritmo de Floyd, su eficiencia teórica es **$O(n^3)$** debido a que hay tres bucles anidados.

Torres de Hanoi -> O(2^n)

En este algoritmo el tamaño del problema viene representado por el número de discos con los que se juega. El juego consiste en desplazar todos los discos desde una torre a otra siguiendo las siguientes reglas:

- En cada movimiento solo puede desplazarse un disco y el resto deben estar colocados en las torres.
- Nunca puede haber un disco de mayor tamaño sobre uno de menor tamaño.
- Solo será posible mover el disco que esté arriba en cada torre.

Este algoritmo tiene una eficiencia 0(2^n) porque para mover un disco es necesario realizar n-1 desplazamientos. Por eso, por ejemplo, para resolver un problema con 3 discos sería necesario realizar 2^n-1, cuya eficiencia es O(2^n).

Este algoritmo tiene una eficiencia 0(2^n) porque para mover un disco es necesario realizar n-1 desplazamientos. Por eso, por ejemplo, para resolver un problema con 3 discos sería necesario realizar 2^n-1, cuya eficiencia es O(2^n).

En este algoritmo vemos que la implementación es bastante sencilla, tenemos M que es el número de discos, i es el número de columna en el que están los discos y j es el número de columna al que se van a mover los discos. Tanto i como j pueden tener un valor del conjunto {1, 2, 3}.

```
void hanoi (int M, int i, int j)
{
  if (M > 0)
    {
      hanoi(M-1, i, 6-i-j);
      cout << i << " -> " << j << endl;
      hanoi (M-1, 6-i-j, j);
  }
}
```

Podemos hacer un breve análisis para averiguar qué ecuación de recurrencia está relacionada con este algoritmo, si lo hacemos en relación al número de movimientos (t) en términos del número de discos(n), tenemos:

1. Considerando que con un disco se produce un movimiento tendríamos nuestra primera condición inicial, T(1)=1.
2. Si hacemos el juego con 2 discos, tendríamos un movimiento para pasar el disco menor a una torre, otro para pasar el disco mayor a otra torre, y por último un tercer movimiento para colocar el menor encima del mayor, por lo que T(2)=3. Si lo hacemos para 3 discos tendríamos un total de 8 movimientos, por lo que T(3)=7.
3. Teniendo estas tres condiciones, podemos deducir que T(n) podría tener como solución no recurrente a 2^n - 1.

4. Sabiendo que se producen n-1 movimientos para poder mover un disco inferior a otra torre, la ecuación de recurrencia asociada tiene que venir en términos de T(n-1), teniendo de condición inicial T(1)=1 y que T(2)=3 y T(3)=7, se deduce que T(n)= 2T(n-1)+1.

$$T_n = 2T_{n-1} + 1$$

A la hora de generar los datos, para calcular la eficiencia empírica lógicamente hay que eliminar la salida que genera dicha función ya que solo nos interesa generar un fichero con dos columnas, una para la el tamaño del problema y otra para el tiempo de ejecución. Si no eliminamos esa salida el sistema de archivos no respondería al crearse un fichero de salida con tantos MB, ya que se genera un gran número de llamadas recursivas cuando el tamaño del problema aumenta.

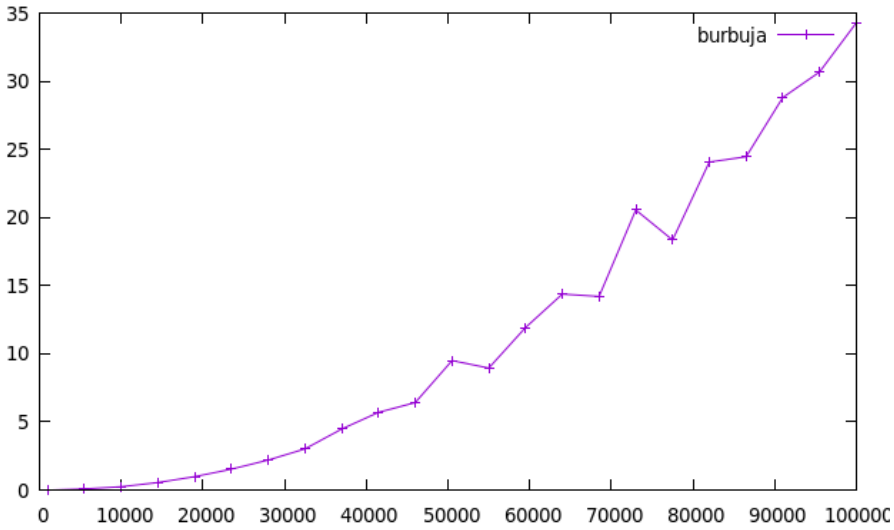
Para hacer una representación lo más exacta posible hemos optado por hacer cinco ejecuciones del mismo algoritmo, de los datos obtenidos de estas cinco ejecuciones obtenemos la media y la mediana para después en el apartado siguiente decidir cuál de las dos es más representativa de los datos, en este caso, como veremos, escogeremos la mediana.

			EJECUCIONES DEL ALGORITMO DE LAS TORRES DE HANOI							
TAMAÑO	EJEC1	EJEC2	EJEC3	EJEC4	EJEC5	TAMAÑO	MEDIANA	TAMAÑO	MEDIA
10	2,20E-05	6,00E-06	6,00E-06	6,00E-06	2,30E-05	10	0,000006	10	0,0000126
11	4,00E-05	1,10E-05	1,10E-05	1,10E-05	3,90E-05	11	0,000011	11	0,0000224
12	7,50E-05	2,10E-05	2,00E-05	2,00E-05	7,50E-05	12	0,000021	12	0,0000422
13	0,000146	3,90E-05	3,90E-05	4,00E-05	0,000146	13	0,00004	13	0,000082
14	0,000286	7,70E-05	7,60E-05	7,80E-05	0,000284	14	0,000078	14	0,0001602
15	0,000565	0,000151	0,000152	0,000155	0,000564	15	0,000155	15	0,0003174
16	0,001124	0,000301	0,000301	0,000308	0,001152	16	0,000308	16	0,0006372
17	0,00226	0,000658	0,000658	0,000615	0,002261	17	0,000658	17	0,0012904
18	0,003011	0,001565	0,001432	0,001285	0,001238	18	0,001432	18	0,0017062
19	0,002517	0,002452	0,002933	0,002561	0,00254	19	0,00254	19	0,0026006
20	0,00497	0,004939	0,004843	0,004967	0,004966	20	0,004966	20	0,004937
21	0,009996	0,010223	0,010143	0,01	0,009839	21	0,01	21	0,0100402
22	0,020016	0,019664	0,020124	0,01986	0,020013	22	0,020013	22	0,0199354
23	0,040545	0,039832	0,040185	0,039734	0,04143	23	0,040185	23	0,0403452
24	0,078939	0,080111	0,079269	0,079215	0,07973	24	0,079269	24	0,0794528
25	0,157349	0,157992	0,156192	0,158049	0,157028	25	0,157349	25	0,157322
26	0,309683	0,310422	0,31277	0,315409	0,309651	26	0,310422	26	0,311587
27	0,616788	0,616525	0,616255	0,621047	0,616951	27	0,616788	27	0,6175132
28	1,2316	1,2336	1,23016	1,23411	1,23152	28	1,2316	28	1,232198
29	2,46073	2,45913	2,4579	2,45957	2,45937	29	2,45937	29	2,45934
30	4,91282	4,91525	4,91524	4,91208	5.01361s	30	4,91403	30	4,9138475
31	9,81909	9,81618	9,81531	9,8167	9,82725	31	9,8167	31	9,818906
32	19,7957	19,6306	19,6338	19,6313	19,6789	32	19,6338	32	19,67406
33	39,2607	39,2569	39,25	39,4691	39,2641	33	39,2607	33	39,30016
34	78,5641	78,6553	78,9661	78,5665	78,4996	34	78,5665	34	78,65032

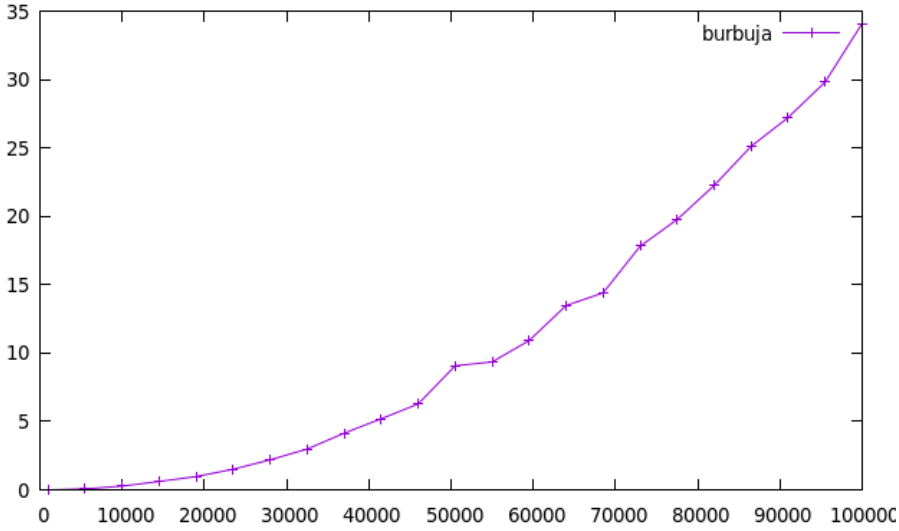
3. Gráficas resultantes

Algoritmo de Burbuja

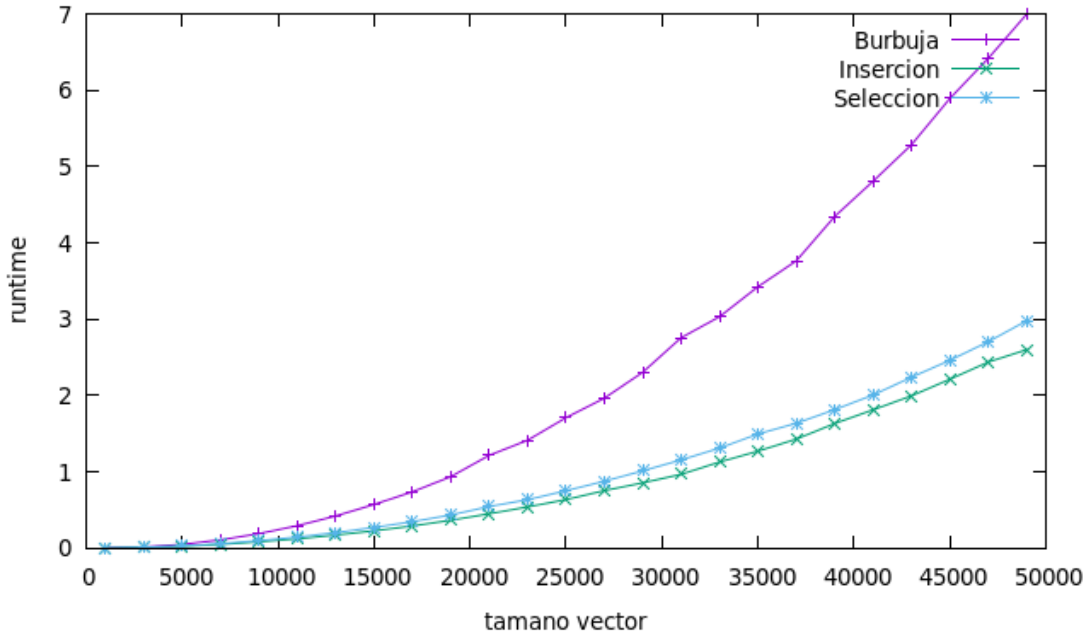
Primeramente, representamos los datos obtenidos con la mediana de las cinco ejecuciones. Comprobamos como aparentan crecer de forma cuadrática, pero observamos que aparecen valores atípicos en exceso.



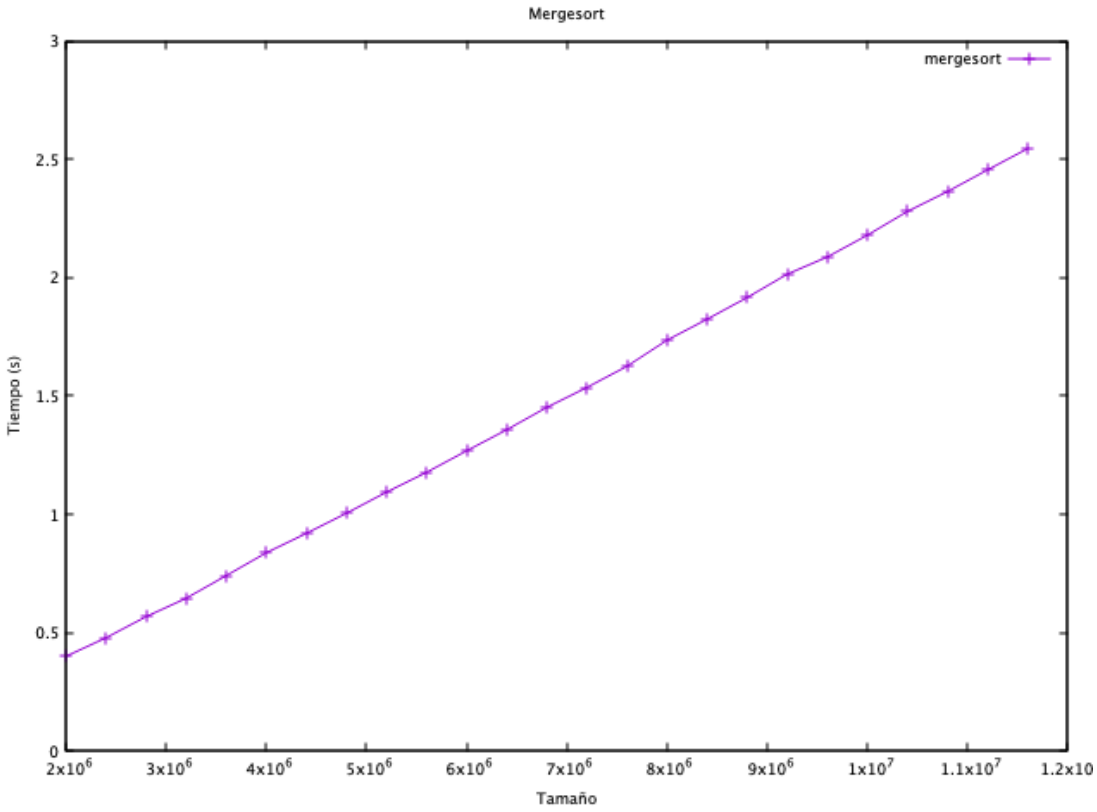
Por tanto, decidimos calcular la media en lugar de la mediana, y obtenemos un resultado mucho más preciso y continuo.



En la siguiente gráfica se puede apreciar la comparación de todos los algoritmos de orden cuadrático, en el que salta a la vista la gran diferencia entre burbuja y los otros dos algoritmos.



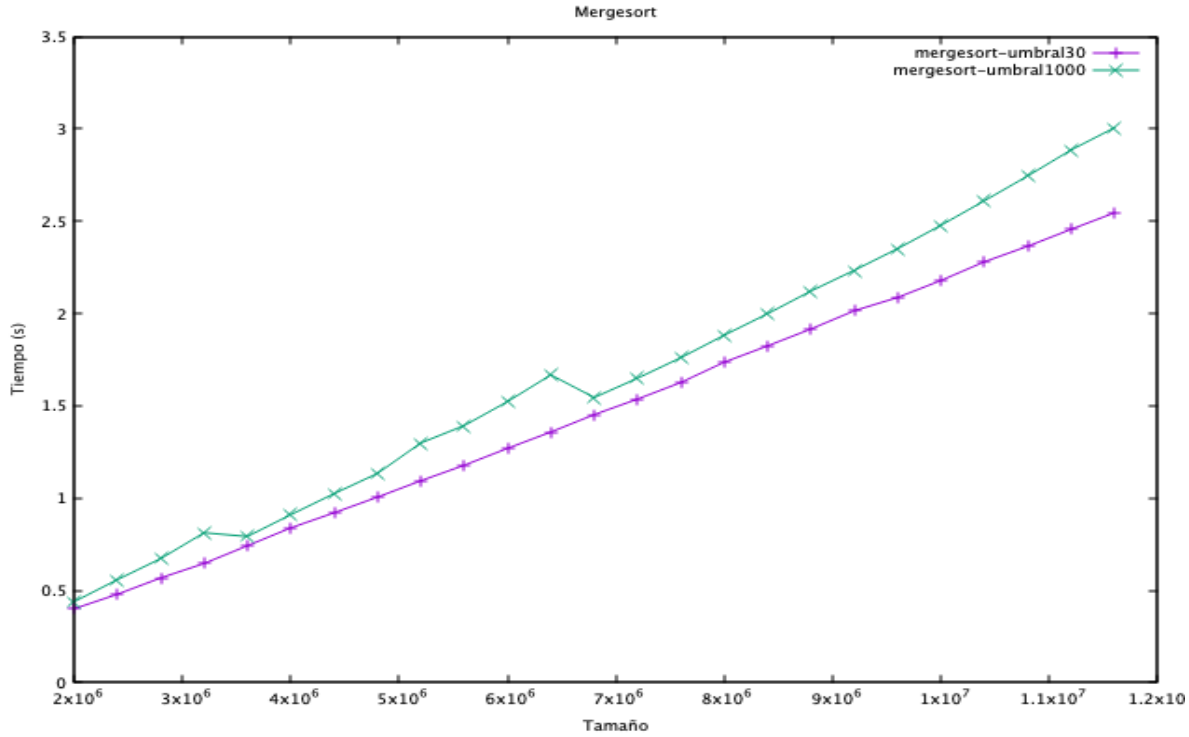
Algoritmo de Mergesort



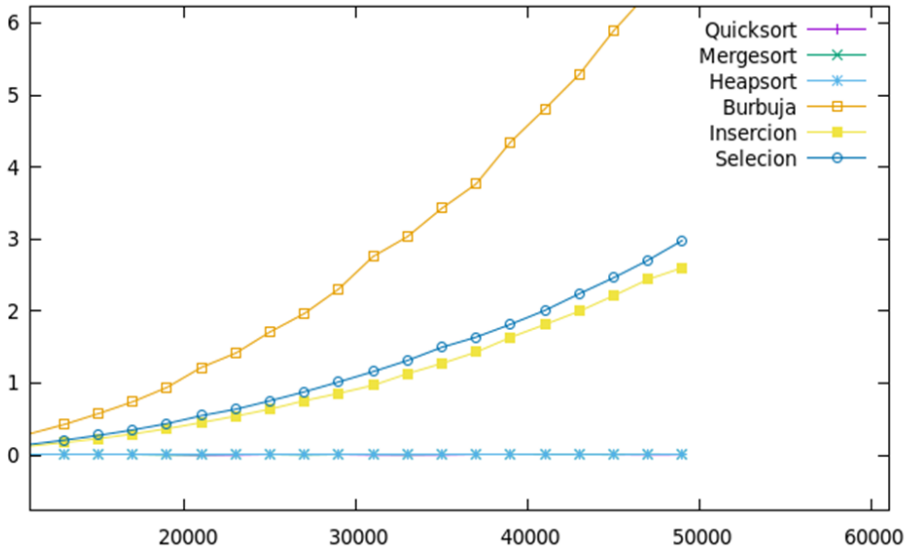
Como podemos observar mergesort es muy rápido, siendo capaz de ordenar vectores de tamaño muy grande en muy poco tiempo.

También es importante comentar que esta gráfica no es la que hemos obtenido directamente. Después de realizar la mediana de los datos, observamos cómo se seguían produciendo saltos en la gráfica, y

debido a la serie de precauciones que estábamos tomando, sospechamos que debía haber algún factor que estuviese afectando a los cálculos del que no nos habíamos percatado. Descartamos la opción de que la aleatoriedad de los números estaba afectando, ya que no produce impacto en este algoritmo. Investigando descubrimos en el código de mergesort una constante de tipo entero (const int UMBRAL_MS=1000), que determina en que momento mergesort deja de ejecutarse recursivamente y pasa a ordenar el subvector por inserción. Y es reduciendo esta constante a un número más bajo (30 en nuestro ejemplo) como eliminamos el problema de los saltos, ya que inserción se activa en tamaños más pequeños y no perjudica el rendimiento de mergesort, que funciona mejor en tamaños grandes. Por tanto, realmente no estamos ejecutando un mergesort puro, sino un mergesort-inserción:

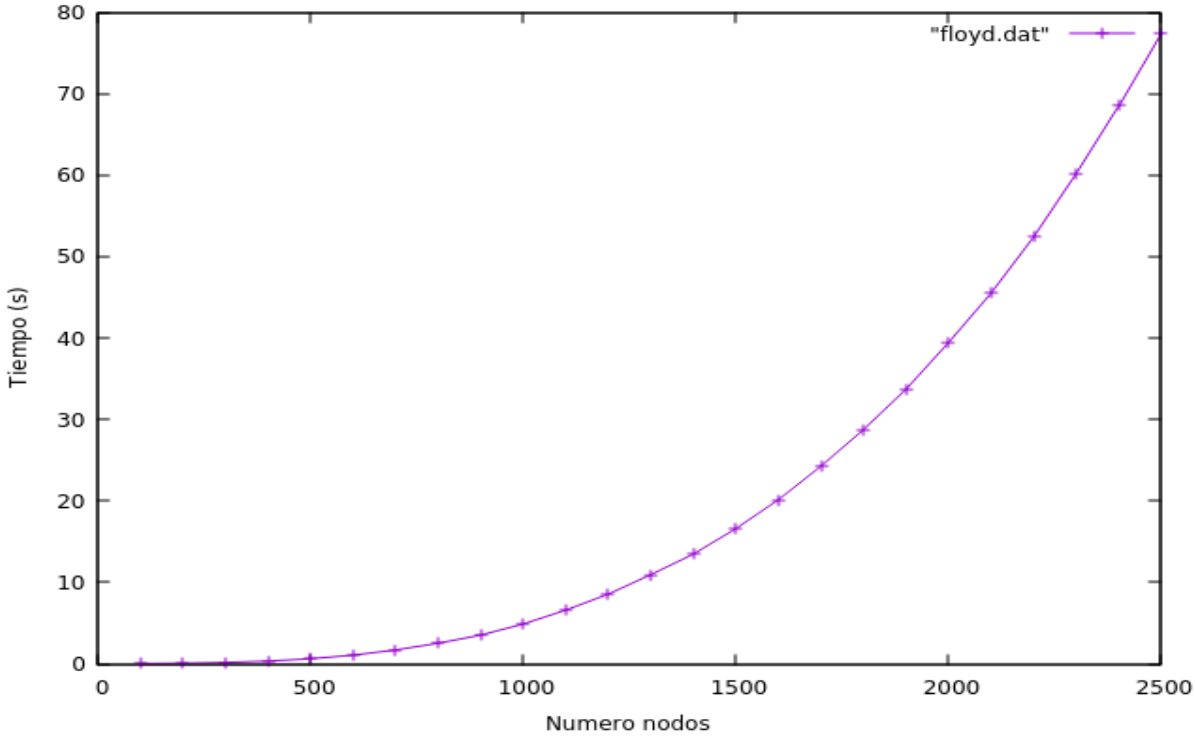


Una vez obtenidas todas las gráficas de los algoritmos de ordenación, vamos a realizar una gráfica comparativa de todos ellos:

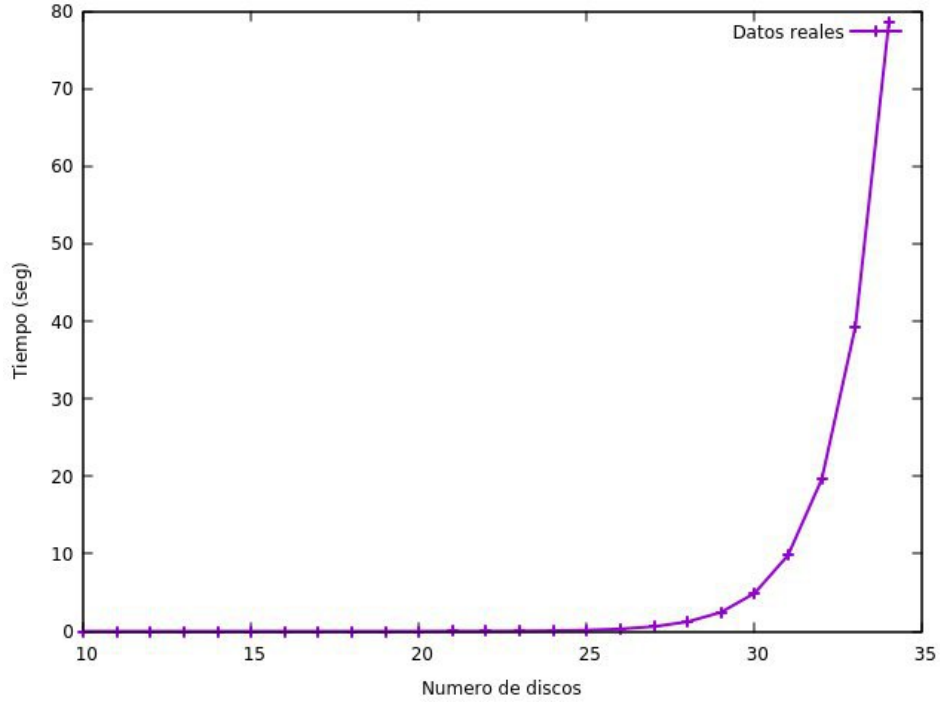


Como podemos ver, el algoritmo de burbuja sería el que peor rendimiento ha dado, después estarían inserción y selección. Y por último los algoritmos de Orden n*log(n).

Algoritmo de Floyd:



Algoritmo de las Torres de Hanoi:



En este caso para la representación de los datos hemos elegido la mediana ya que es realmente la más representativa de los datos. Por ejemplo, en el siguiente conjunto de números: {1 1 1 1 10} la media nos dice que el valor medio es 2,8, sin embargo, si hacemos la mediana obtenemos que el valor que deja al 50% de los datos a sus lados es el 1, el cual representa mejor al conjunto de datos obtenido, ya que el 10 ha sido un caso aislado.

Tras la representación del conjunto de datos que tenemos en el archivo generado por mimacro.sh llamado hanoi.dat, podemos observar como la gráfica describe una curva exponencial que se corresponde efectivamente con su eficiencia teórica O(2^n).

4. Eficiencia híbrida de algunos algoritmos

En este ejercicio lo que nos piden es calcular la eficiencia híbrida de cada algoritmo. Para ello, debemos seguir los siguientes pasos:

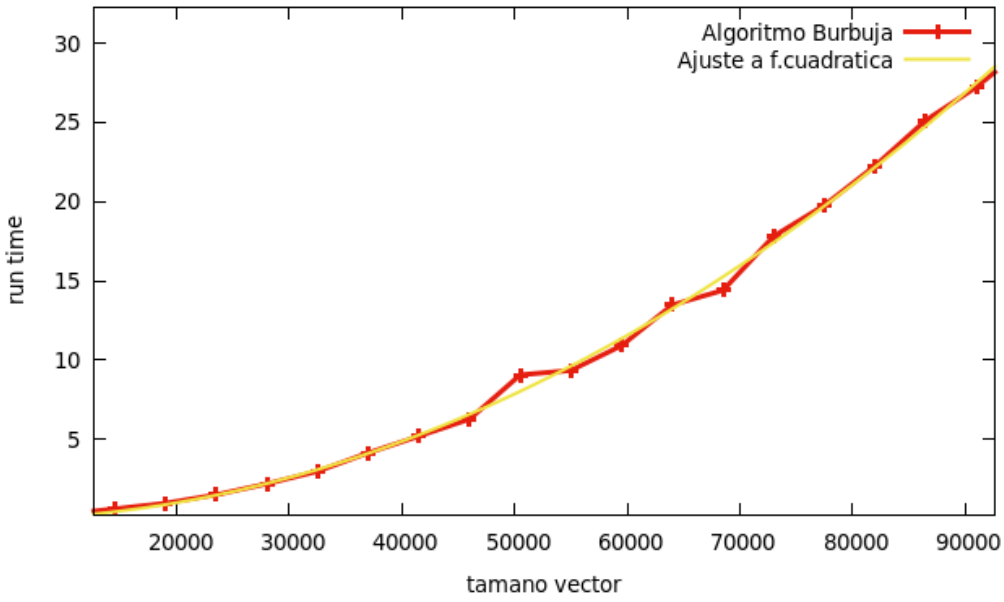
1. Definir la función que queremos ajustar a nuestros datos. Según la eficiencia teórica de nuestro algoritmo definimos la ecuación general de la función que queremos ajustar.
2. Indicarle a gnuplot que haga regresión, es decir, que estime la relación entre nuestros datos y la función que hemos definido.
3. Dibujar nuestros datos y la función definida para poder ver cómo se ajusta esta función a los datos.

-> Ajustes que se corresponden con su eficiencia teórica:

- ***Burbuja ajustada a $f(x)=a*x^2+b*x+c$:***
  - gnuplot> f(x) = a*x*x+b*x+c
  - gnuplot> fit f(x) 'salida_burbuja.dat' via a,b,c. Salida:

Final set of parameters	**Asymptotic Standard Error**	
==========	==========	
a = 3.51929e-09	+/- 2.041e-10	(5.8%)
b = -1.78002e-05	+/- 2.195e-05	(123.3%)
c = -0.0333484	+/- 0.5182	(1554%)

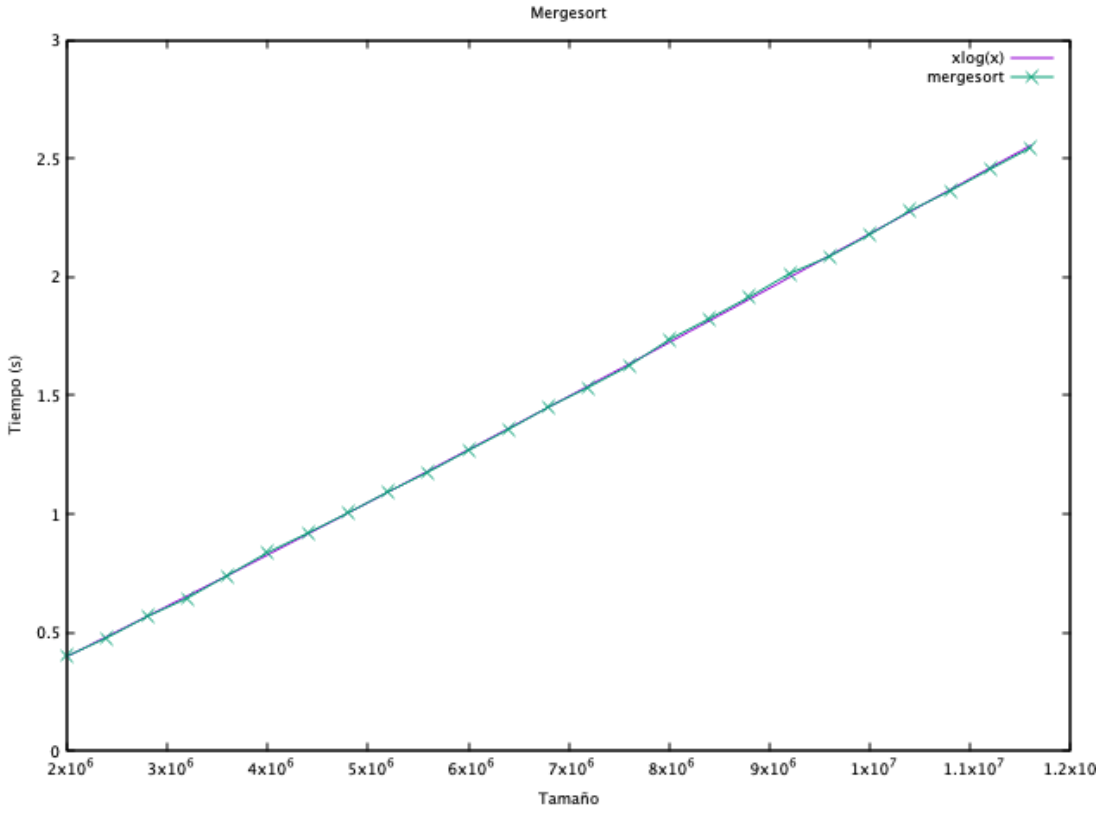
  - gnuplot> plot f(x) 'salida_burbuja.dat' title 'Curva ajustada'



- ***Mergesort ajustado a $f(x)=a*x*\log_2(x)+b$***
  - gnuplot> f(x) = **a***x*(log(x)/log(2))+**b**
  - gnuplot> fit f(x) 'salida_mergesort.dat' via a,b. Salida:

Final set of parameters	**Asymptotic Standard Error**	
==========	==========	
a = 9.35851e-09	+/- 1.973e-11	(0.2108%)
b = 0.0072172	+/- 0.003356	(46.5%)

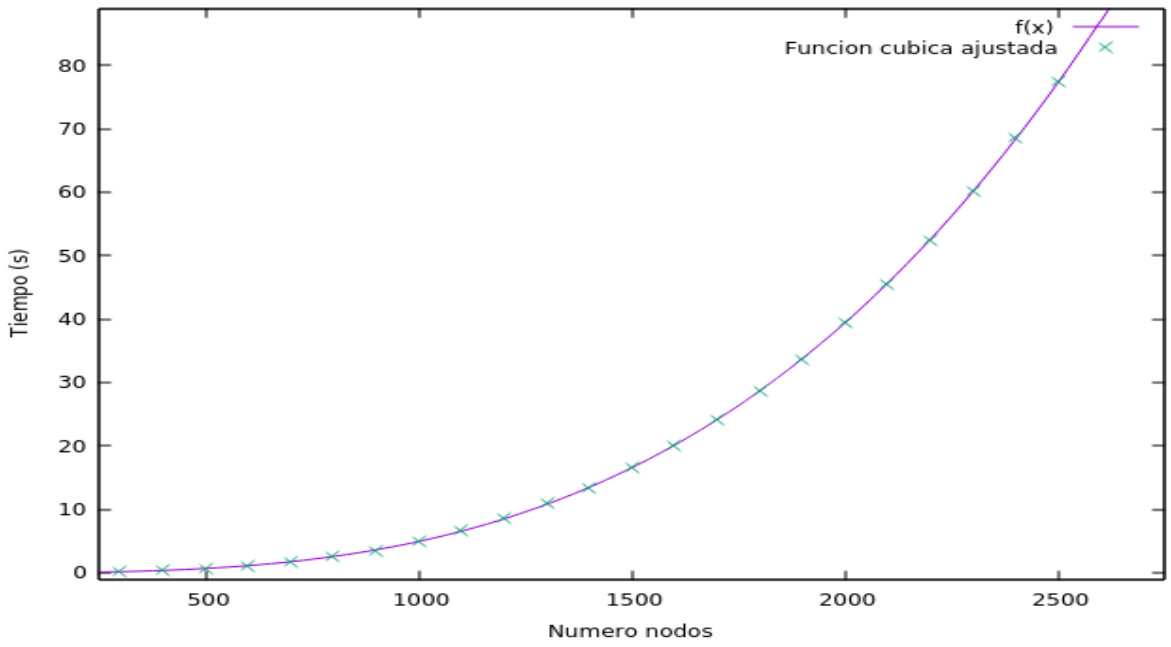
  - gnuplot> plot f(x) title “xlog(x)”, 'salida_mergesort.dat' title “mergesort”



- ***Floyd ajustada a f(x)=ax3+bx2+cx+d***
  - gnuplot> f(x) = a*x*x*x+b*x*x+c*x+d
  - gnuplot> fit f(x) 'floyd.dat' via a,b,c,d

Final set of parameters	**Asymptotic Standard Error**	
==========	==========	
a = 5.14211e-09	+/- 6.163e-11	(1.199%)
b = -6.73147e-07	+/- 2.435e-07	(36.17%)
c = 0.000544829	+/- 0.0002753	(50.52%)
d = -0.0943319	+/- 0.08428	(89.35%)

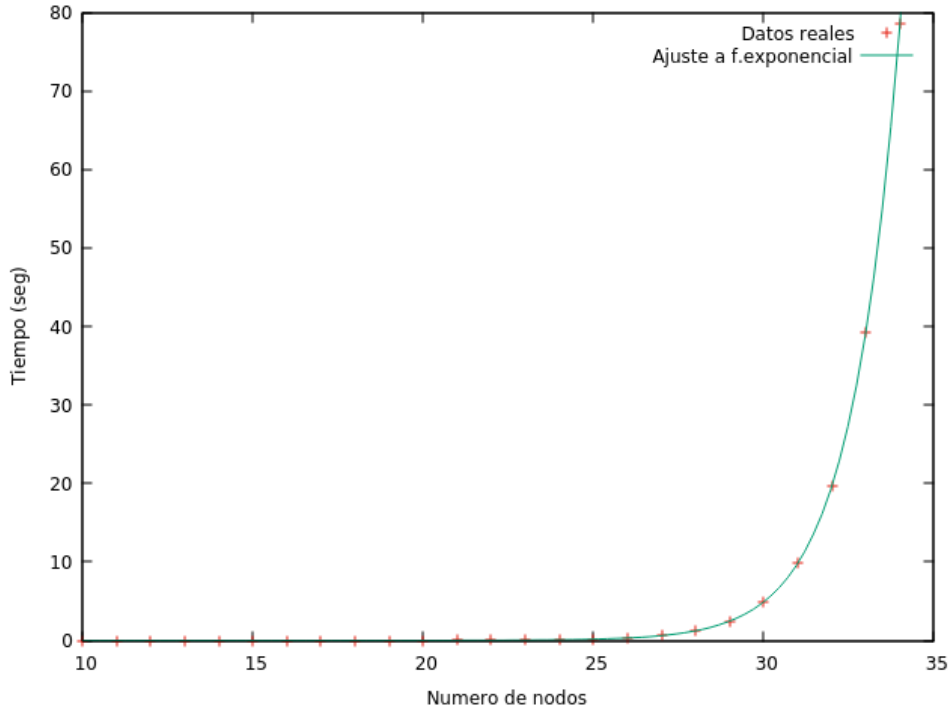
  - gnuplot> plot f(x), 'floyd.dat' title 'Funcion cubica ajustada'



- ***Hanoi ajustada a f(x)=a*b^x***
  - gnuplot> f(x)=a*b**x
  - gnuplot> fit f(x) 'hanoi.dat' via a,b

Final set of parameters		**Asymptotic Standard Error**	
==========		==========	
a	= 6.3623e-09	+/- 3.945e-10	(6.2%)
b	= 1.98379	+/- 0.003489	(0.1759%)

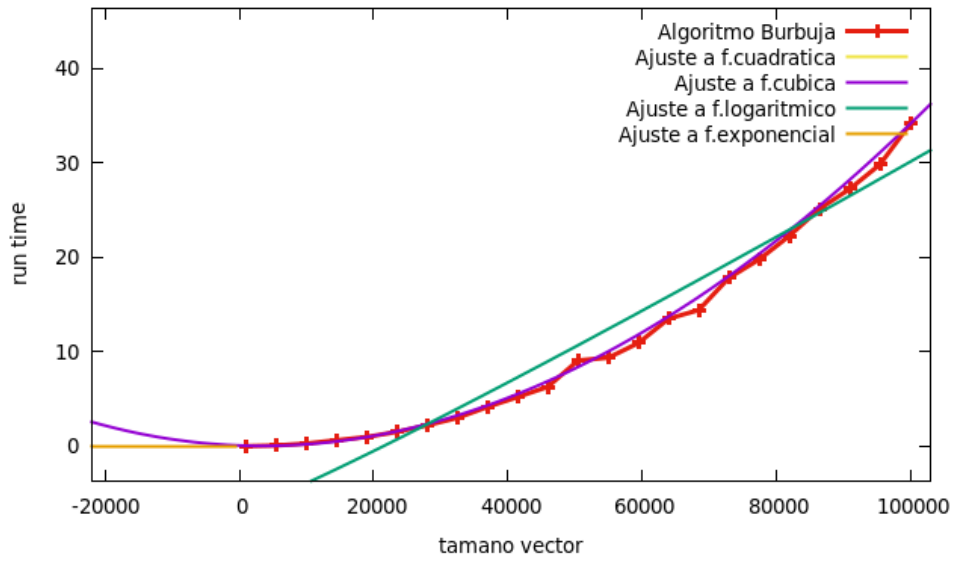
gnuplot> plot f(x), 'hanoi.dat' via a, b



El ajuste realizado se corresponde con los datos generados.

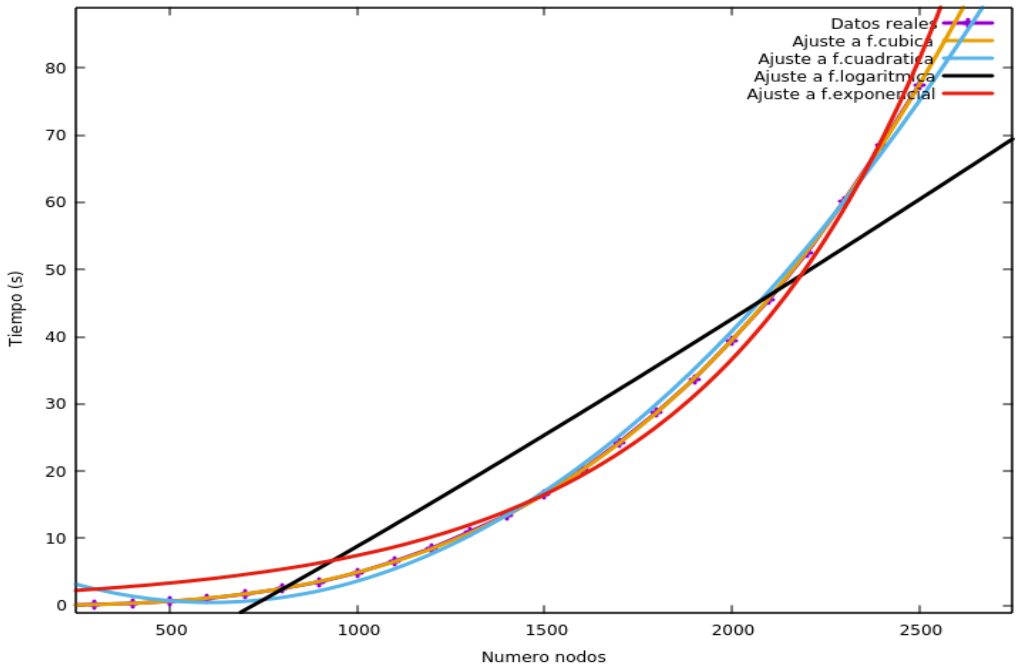
-> Ahora vamos a realizar una comparación de algunos algoritmos con diferentes ajustes para compararlos con el original:

- ***Burbuja ajustada al resto de funciones:***



Observamos que la que mejor se ajusta es la función cuadrática, produciendo las demás un peor ajuste.

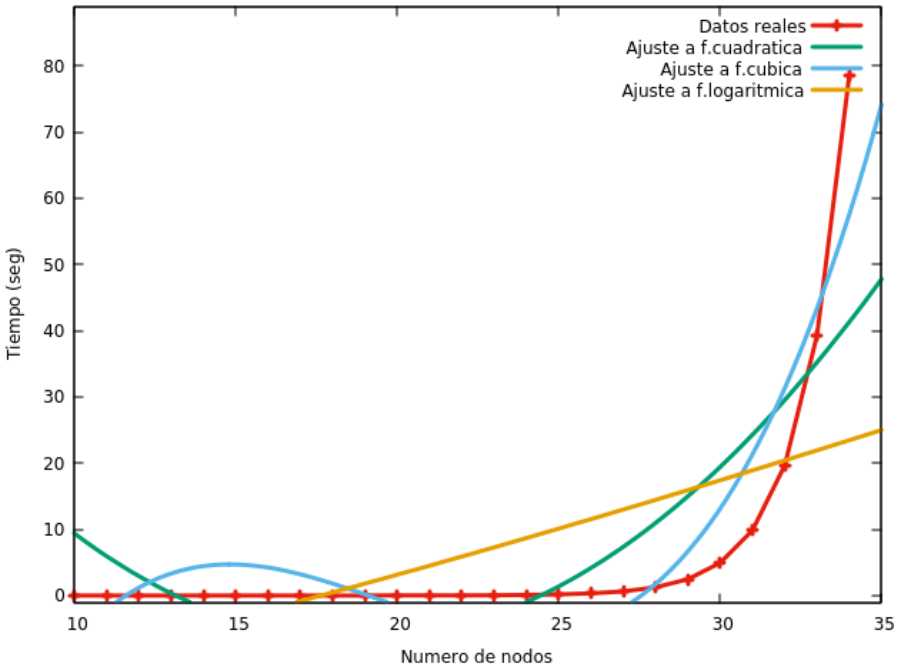
- ***Floyd ajustada al resto de funciones:***



Como se observa en la imagen, la función que mejor ajusta a los datos reales es la función cúbica, ya que es la que coincide perfectamente con los datos reales. Las demás funciones no ajustan muy bien, la función exponencial y la cuadrática, al final se acercan un poco a los datos reales, pero al inicio no coinciden.

NOTA: La gráfica de los datos reales no se aprecia bien al estar solapada por la función cúbica.

- ***Hanoi ajustada al resto de funciones:***



Como se observa en la gráfica, realmente no hay ninguna función que se ajuste bien a los datos reales de nuestro algoritmo, excepto la función exponencial (la hemos omitido porque arriba ya hemos hecho su ajuste).

5. Eficiencia empírica y parámetros externos

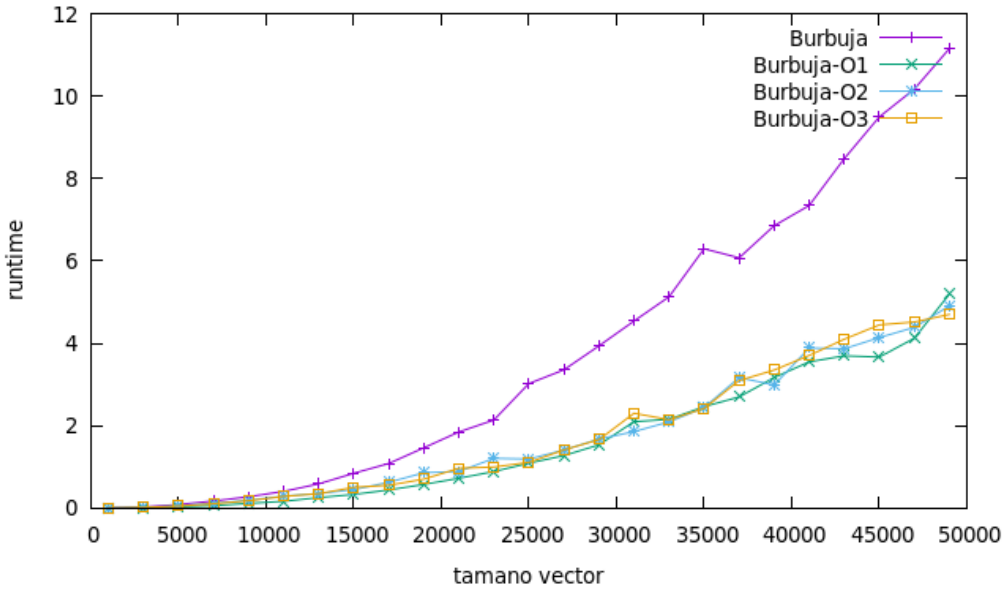
En este ejercicio se pide que hagamos un estudio de la eficiencia en función de otros parámetros externos, ya que la eficiencia no sólo depende de cómo sea el algoritmo, sino que hay otros factores que pueden influenciar en la ejecución de este tales como las opciones de compilación usadas para crear el ejecutable, el sistema operativo sobre el que se ejecuta, el ordenador sobre el que se ejecuta, etc. En este caso, se va a realizar un estudio mediante las diferentes opciones de optimización en la compilación. Los flags son los siguientes, y cada uno de ellos activa y desactiva unos subflags de más bajo nivel:

1. -O0: El código no se optimizará.
2. -O1: El compilador trata de reducir el tamaño del código y el tiempo de ejecución sin efectuar optimizaciones que conlleven una gran cantidad de tiempo de compilación.
3. -O2: Aplica casi todas las optimizaciones posibles que no implican un sacrificio entre tiempo de ejecución y memoria. Comparado con -O1 aumenta el tiempo de compilación y el rendimiento.
4. -O3: Este nivel de optimización es el más alto. En algunas ocasiones puede provocar que la ejecución sea más lenta, debido a un flag al que llama (-ftree-vectorize) que intenta vectorizar los bucles dando lugar a este enlentecimiento. Suele acarrear un código más largo.

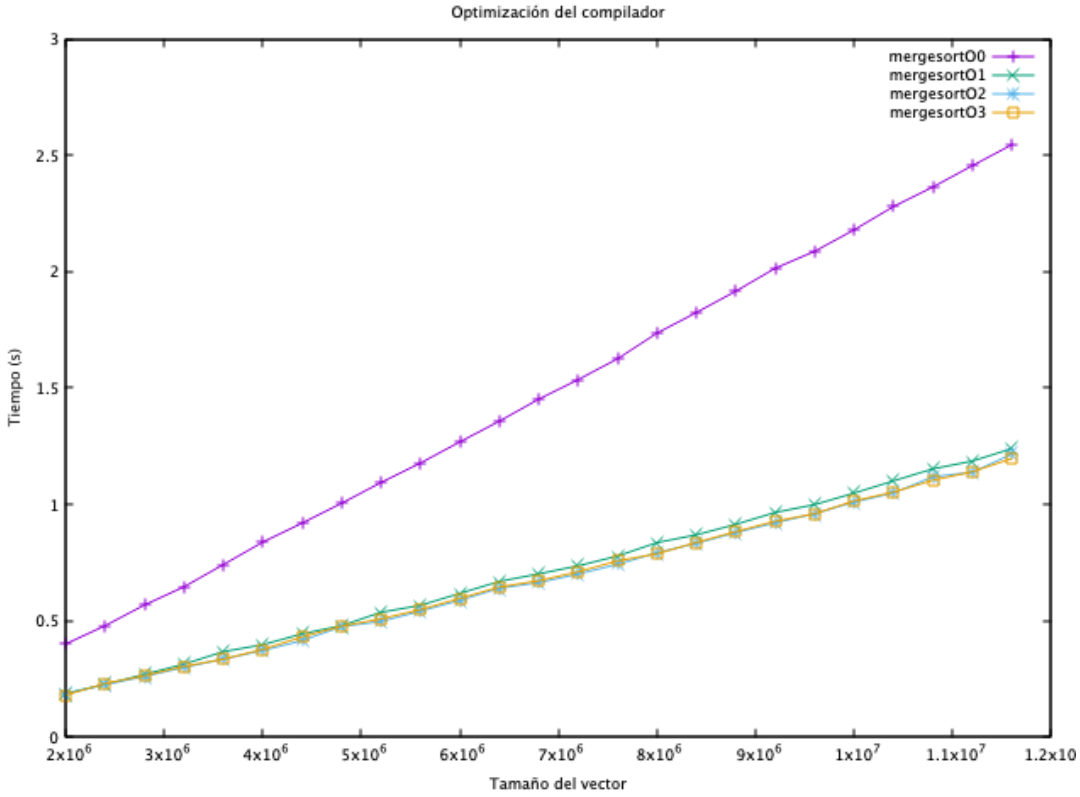
Ahora procedemos a mostrar una gráfica para cada algoritmo, para ello he tenido que compilarlos previamente con g++ y la opción de compilación deseada, muestro este ejemplo, para el resto de algoritmos se procede de igual forma:

1. g++ -O0 -o burbuja burbuja.cpp
2. g++ -O1 -o burbuja burbuja.cpp
3. g++ -O2 -o burbuja burbuja.cpp
4. g++ -O3 -o burbuja burbuja.cpp

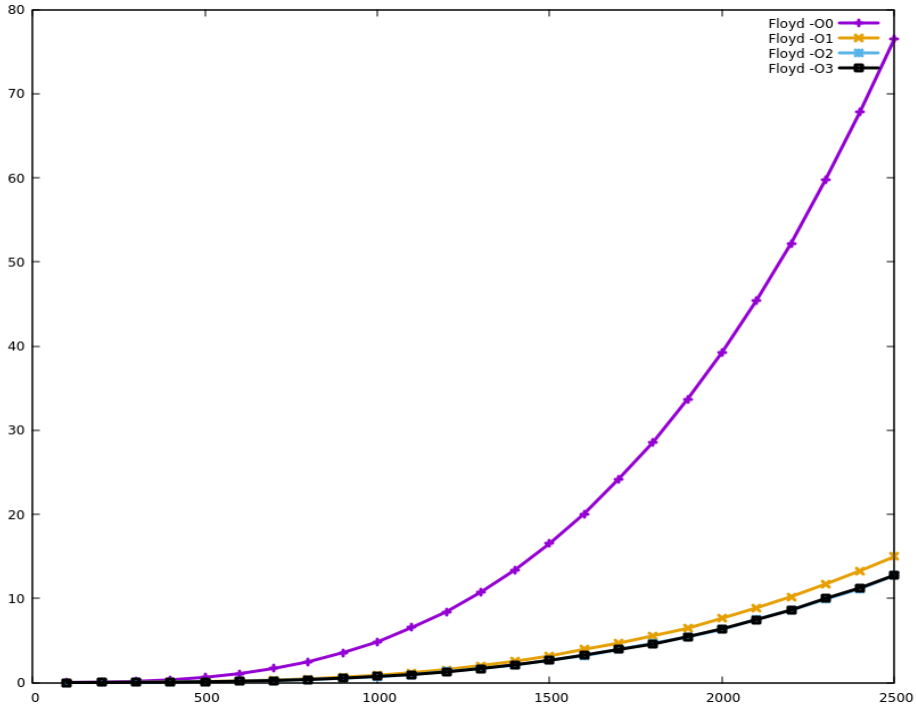
- ***Algoritmo de ordenación por Burbuja:***



- ***Algoritmo mergesort:***

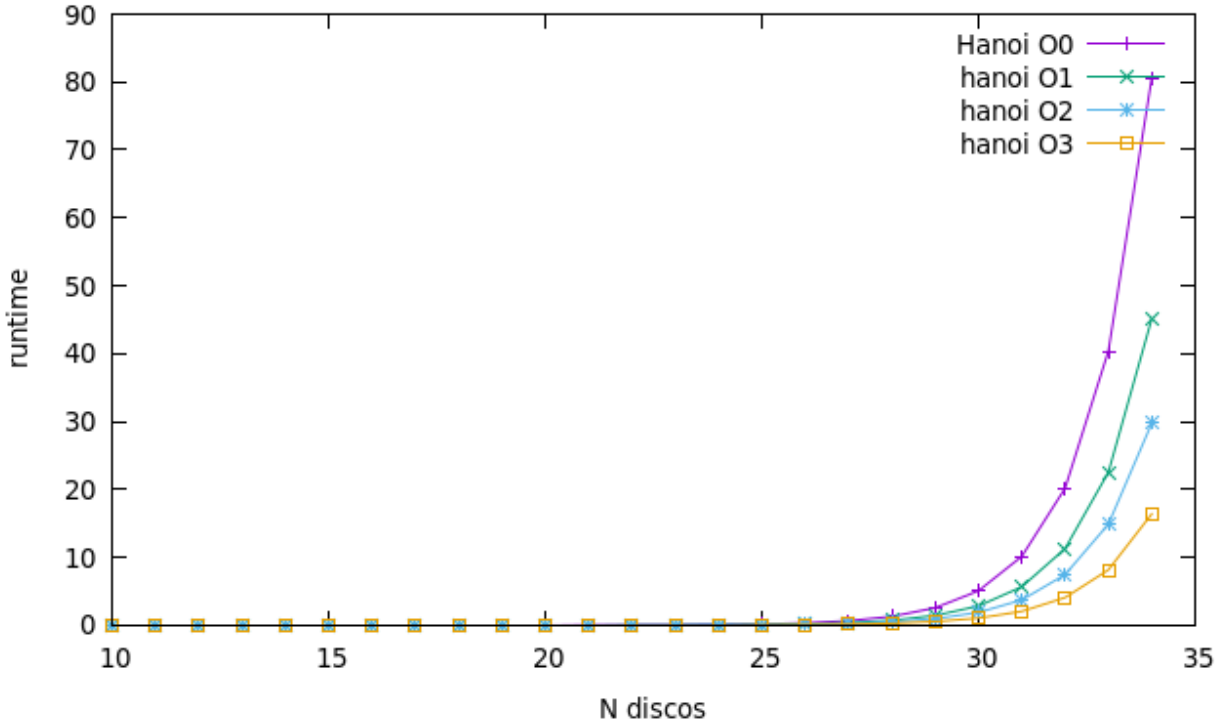


- ***Algoritmo de Floyd:***



NOTA: La optimización con –O2 no aprecia ve apenas porque está superpuesta con la optimización –O3, es decir, ambas optimizaciones son prácticamente idénticas para este algoritmo.

- ***Algoritmo de las Torres de Hanoi:***



En esta gráfica vemos que el tiempo que se ha tardado en la ejecución va reduciéndose conforme se va aumentando el nivel de optimización, en este algoritmo la opción de optimización -O3 produce mejor rendimiento que la opción -O2.

6. Conclusión:

En la práctica hemos demostrado como la eficiencia empírica se ajusta en gran medida a la eficiencia teórica. También hemos observado como factores del propio sistema donde se ejecutan los algoritmos afectan a esta eficiencia empírica, en lo que llamamos las constantes ocultas, y que aplicando optimizaciones a la hora de compilar se pueden reducir ampliamente los tiempos de ejecución.