Práctica 1: Análisis de Eficiencia de Algoritmos

Salvador Romero Cortés
Abel Ríos González
Raúl Durán Racero
Alberto Palomo Campos
Manuel Contreras Orge

Índice

Introducción

En esta práctica hemos estudiado la importancia del análisis de la eficiencia de los algoritmos, calculando y comparando los tiempos de los distintos algoritmos sin optimizar y optimizados con la opción -O2 de g++.

Algoritmos

Orden cuadrático (n^2)

Número de elementos	Tiempo	
	Burbuja	Inserción
1000	0,001013	0,000641
2000	0,005496	0,003507
3000	0,008341	0,005762
4000	0,016454	0,01085
5000	0,020168	0,010594
6000	0,023315	0,0194
7000	0,042037	0,024319
8000	0,046724	0,0278
9000	0,060599	0,034871
10000	0,080852	0,049935
11000	0,103205	0,054327
12000	0,118171	0,065861
13000	0,148478	0,071711
14000	0,180319	0,087327
15000	0,208435	0,101118
16000	0,244259	0,11346
17000	0,282936	0,123755
18000	0,327441	0,143577
19000	0,389044	0,156507
20000	0,422813	0,174984
21000	0,470729	0,194062
22000	0,529972	0,211922
23000	0,586939	0,230065
24000	0,647461	0,245232
25000	0,704676	0,27198
26000	0,777404	0,290884
27000	0,863583	0,313948
28000	0,930515	0,342312
29000	0,995931	0,361499
30000	1,09278	0,399943
31000	1,16948	0,417477
32000	1,25182	0,436098
33000	1,33835	0,4685
34000	1,42794	0,495719
35000	1,51265	0,531717
36000	1,63462	0,555547
37000	1,72527	0,591001
38000	1,86006	0,620326
39000	1,93487	0,654127
40000	2,04198	0,684257
41000	2,18398	0,728092
42000	2,28484	0,766185
43000	2,42763	0,798637
44000	2,51036	0,832944
45000	2,66067	0,862375
46000	2,75778	0,903285
47000	2,88913	0,941095
48000	3,02064	0,992055
49000	3,15038	1,04113
50000	3,26571	1,0585

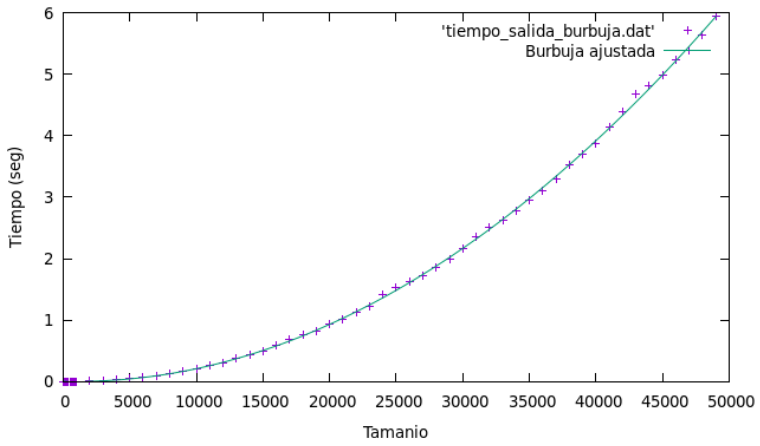
Burbuja

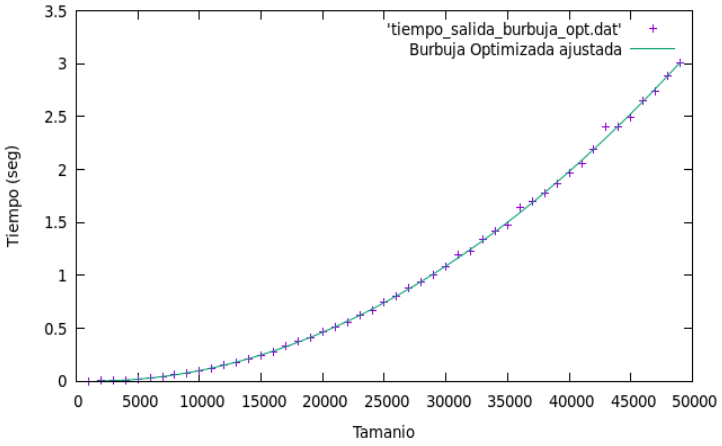
El algoritmo de la burbuja es un sencillo y conocido algoritmo de ordenación. Consiste en revisar cada elemento de una lista y compararlo con el siguiente, y si están en orden incorrecto, los intercambia. La lista se revisa hasta que quede ordenada.

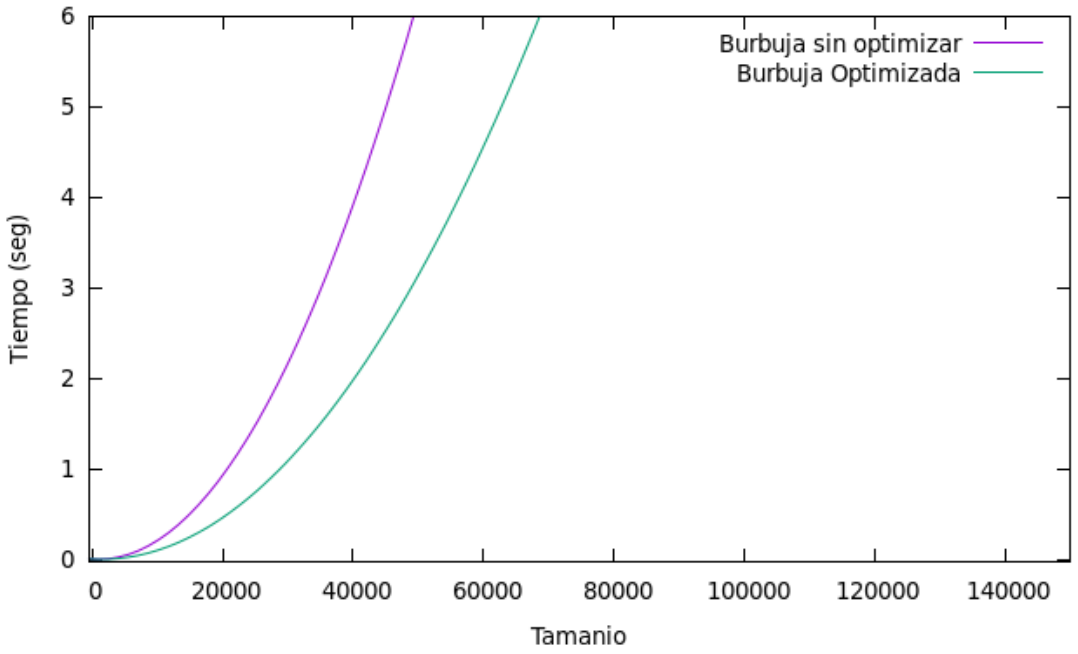
Para estudiar la eficiencia de este algoritmo, hemos llamado a un script para ejecutar este algoritmos con distintos tamaños y almacenar sus tiempos de ejecución:

```
#!/bin/csh -vx
echo "" >> tiempo_salida_burbuja.dat
@ i = 1000
while ( $i < 10000 )
./burbuja $i >> tiempo_salida_burbuja.dat
@ i += 1000
end
```

Este mismo script lo usaremos para los tiempos del algoritmo sin utilizar la opción -O2 de gcc y usándolo.
Una vez obtenidos los tiempos, mediante gnuplot generamos unas gráficas que nos permita comparar fácilmente la diferencia de tiempos entre el algoritmo sin optimizar y el optimizado:







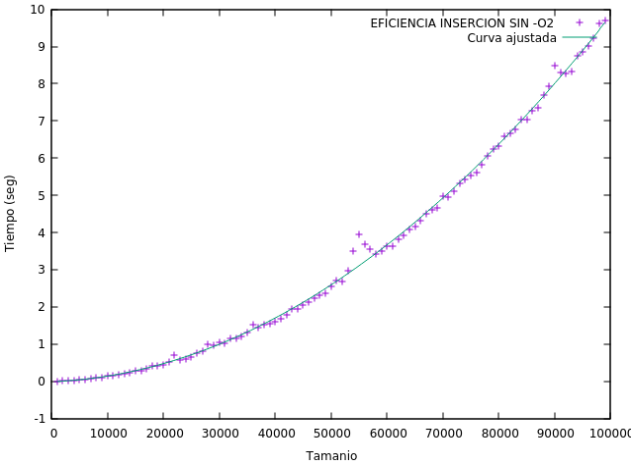
Se ajusta perfectamente

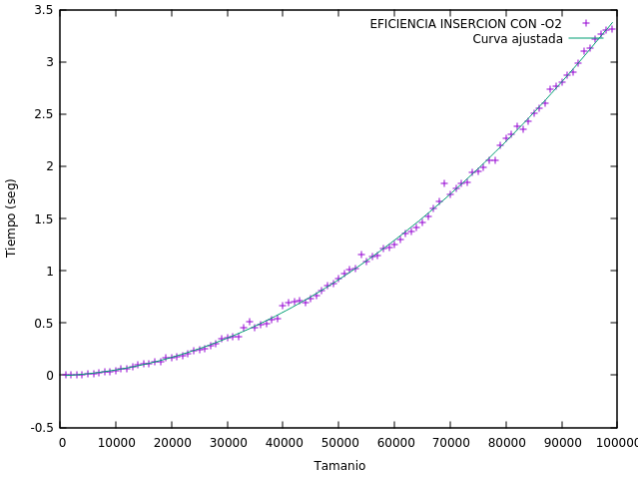
En la gráfica comparativa, podemos ver la gran diferencia en el tiempo de ejecución cuando se optimiza el algoritmo con la orden -O2, y cuando no se hace con esa orden.

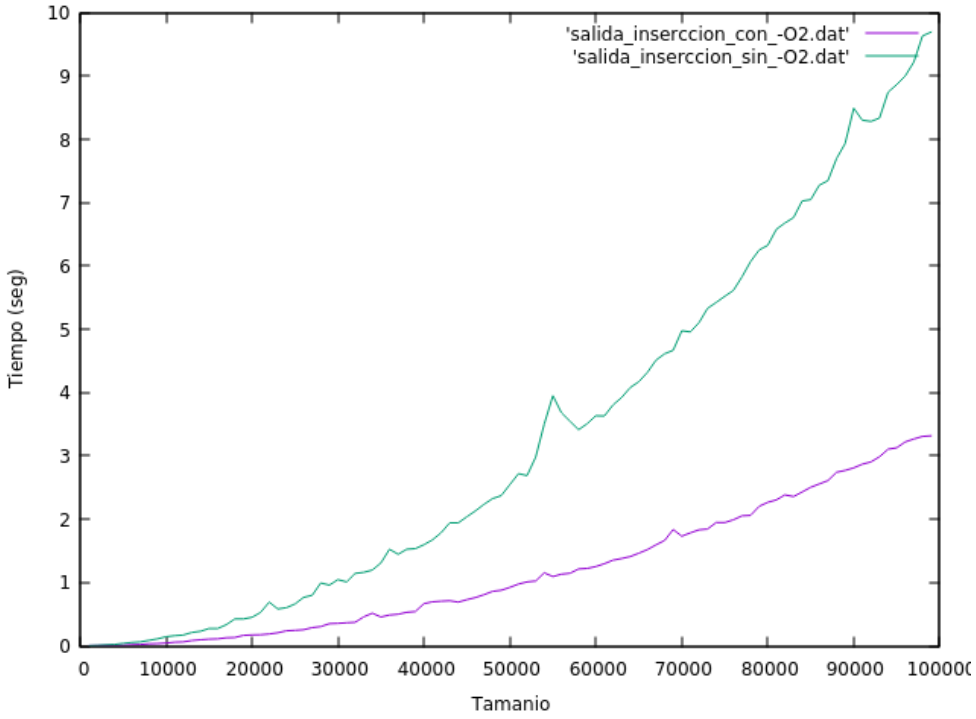
Inserción

El algoritmo de inserción es un algoritmo de ordenación que resulta natural para el ser humano. Consiste en partir de una lista con un solo elemento, la cual está obviamente ordenada. Después, cuando hay *k* elementos ordenados de menor a mayor, se toma el elemento *k+1* y se compara con todos los elementos ya ordenados, deteniéndose cuando se encuentra un elemento menor (habiendo desplazado todos los elementos mayores una posición a la derecha) o cuando ya no se encuentran elementos (todos los elementos fueron desplazados y este es el más pequeño). En este punto se inserta el elemento *k+1* debiendo desplazarse los demás elementos.

```
#!/bin/csh -vx
echo "" >> tiempo_salida_insercion.dat
@ i = 1000
while ( $i < 100000 )
./insercion $i >> tiempo_salida_insercion.dat
@ i += 1000
end
```







Se ajusta perfectamente excepto algunos errores de caché

Orden logarítmico (n*log(n))

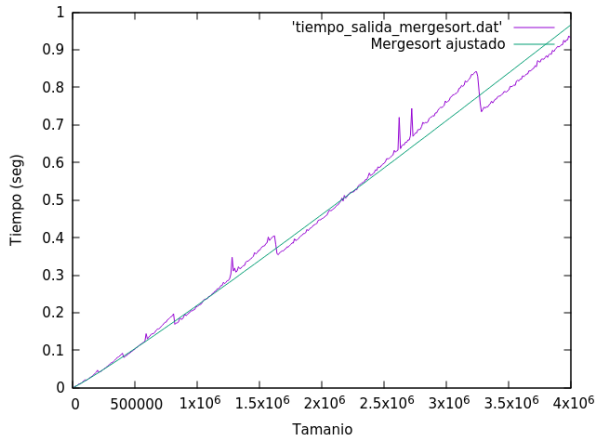
Número de elementos	Tiempo	
	Mergesort	Quicksort
1000	0.00159	8.1e-05
2000	0.003853	0.000173
3000	0.004801	0.000273
4000	0.007434	0.000371
5000	0.010831	0.000477
6000	0.010454	0.000663
7000	0.012628	0.000697
8000	0.015597	0.000813
9000	0.018461	0.000958
10000	0.021302	0.001046
11000	0.02003	0.001145
12000	0.021795	0.00125
13000	0.025249	0.001413
14000	0.027152	0.00149
15000	0.029487	0.0016
16000	0.032473	0.001727
17000	0.035593	0.001873
18000	0.037992	0.002145
19000	0.041586	0.002086
20000	0.047022	0.002212
21000	0.041473	0.002638
22000	0.042989	0.002498
23000	0.044395	0.002537
24000	0.047262	0.002665
25000	0.050109	0.002885
26000	0.052023	0.002937
27000	0.055405	0.003076
28000	0.057376	0.00323
29000	0.060138	0.003272
30000	0.063448	0.003444
31000	0.066566	0.003986
32000	0.06772	0.003818
33000	0.070641	0.003773
34000	0.073957	0.003903
35000	0.077884	0.004101
36000	0.081177	0.004214
37000	0.082362	0.004326
38000	0.086033	0.004524
39000	0.088338	0.00479
40000	0.092144	0.004842
41000	0.080907	0.005071
42000	0.08301	0.004938
43000	0.085539	0.005052
44000	0.08754	0.005102
45000	0.091294	0.005337
46000	0.092391	0.005441
47000	0.096134	0.005542
48000	0.098112	0.005733
49000	0.100854	0.005862
50000	0.102628	0.005984

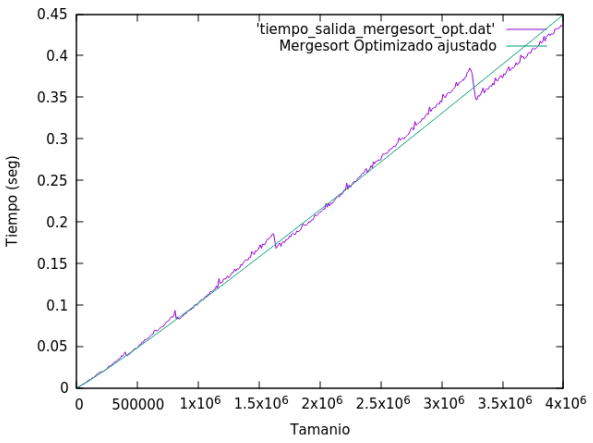
Mergesort

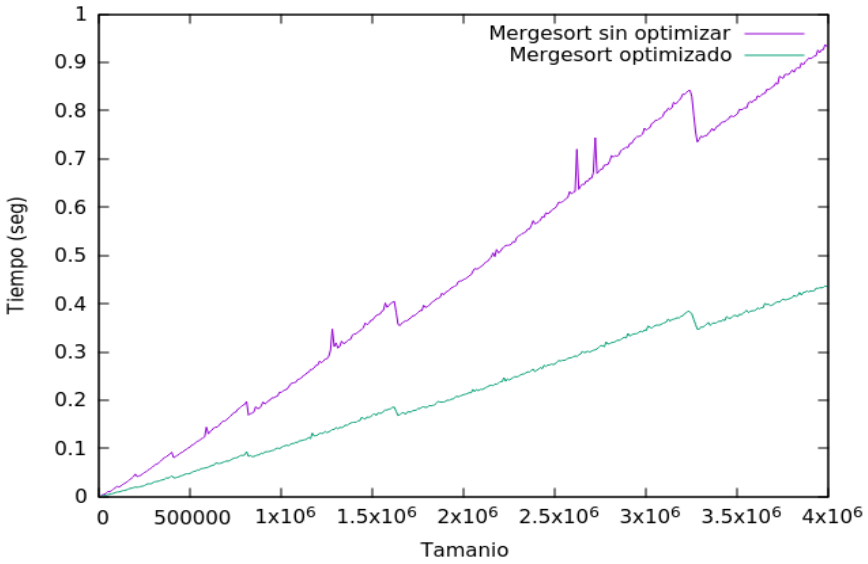
El algoritmo de ordenación por mezcla (o mergesort) es un algoritmo de ordenamiento externo que se basa en la técnica *Divide y Vencerás*.
Consiste en dividir una lista desordenada en dos sublistas del mismo tamaño aproximadamente, para luego ordenar cada sublista recursivamente aplicando el ordenamiento por mezcla. Finalmente, se mezclan las dos sublistas en una sola lista ordenada.

Eficiencia: O(nlogn)

```
#!/bin/csh -vx
echo "" >> tiempo_salida_mergesort.dat
@ i = 10000
while ( $i < 4000000 )
./mergesort $i >> tiempo_salida_mergesort.dat
@ i += 10000
end
```







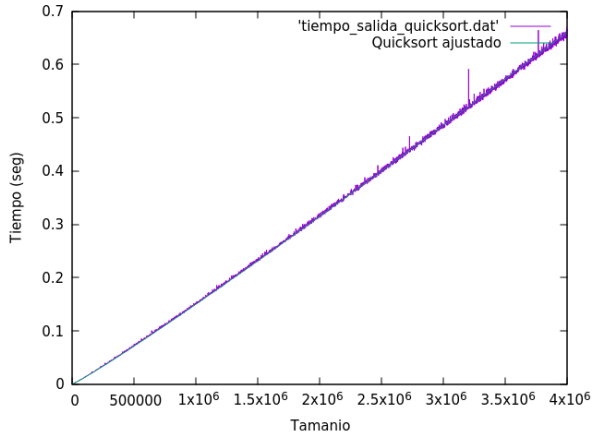
Como podemos ver no se ajusta tanto ya que va haciendo forma escalonada

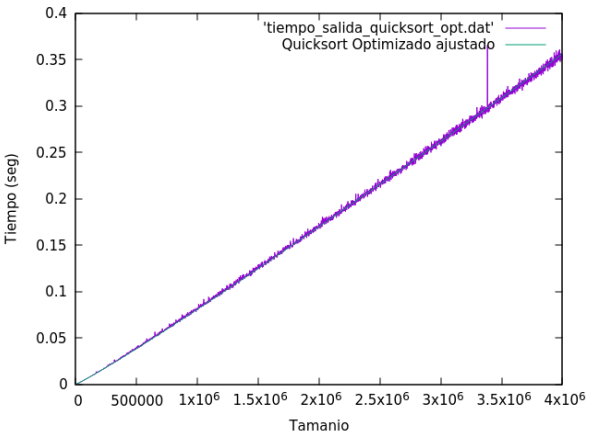
Quicksort

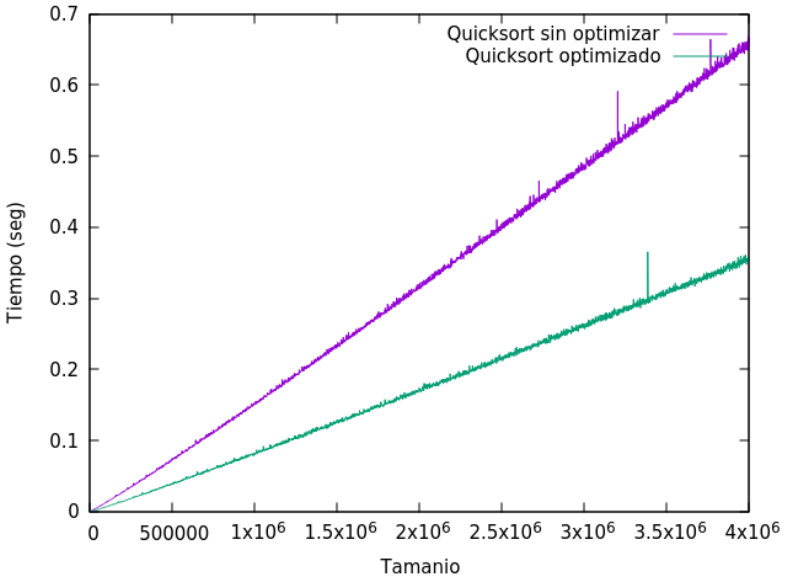
El algoritmo de ordenación quicksort (ordenamiento rápido) consiste en elegir un elemento del conjunto que queremos ordenar, al que llamaremos *pivote*, y recolocar el resto de elementos en torno a este pivote, de manera que a su derecha queden elementos menores que él, y a su izquierda sean mayores. De esta forma, la lista de elementos queda dividida en dos sublistas separadas por el pivote. Se repite este proceso recursivamente mientras cada sublista tenga más de un elemento. Cuando acabe el proceso, tendremos una lista totalmente ordenada.

Eficiencia: *O(nlogn)* (Caso medio).
Complejidad en el peor de los casos: *O(n^2).*
Esta última complejidad se puede evitar fácilmente con alta probabilidad eligiendo el pivote correcto.

```
#!/bin/csh -vx
echo "" >> tiempo_salida_quicksort.dat
@ i = 10000
while ( $i < 4000000 )
./quicksort $i >> tiempo_salida_quicksort.dat
@ i += 10000
end
```







Como podemos ver se ajusta perfectamente, mucho más que el mergesort.

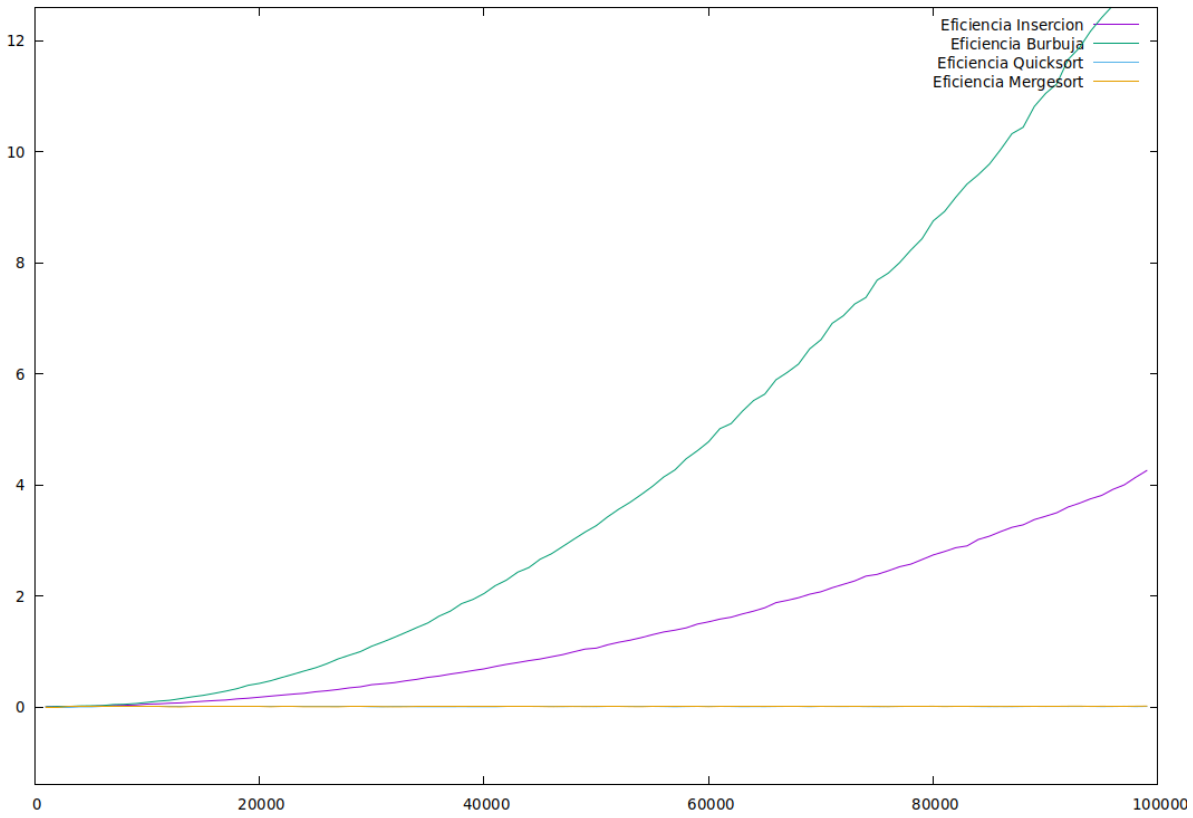
Comparación entre algoritmos de ordenación

Algoritmo	Eficiencia
burbuja	$O(n^2)$
insercion	$O(n^2)$
mergesort	$O(n \log n)$
quicksort	$O(n \log n)$

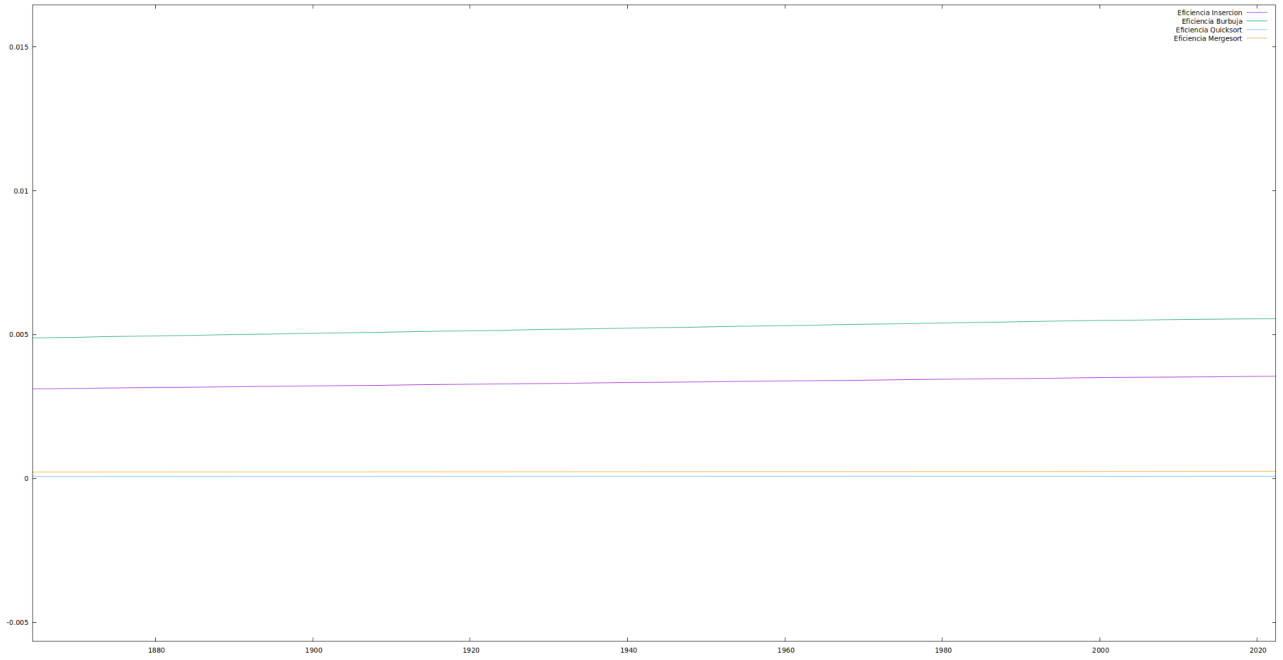
Burbuja e Inserción son menos recomendables por la eficiencia cuadrática.

Merge y quick son los más óptimos.

Mergesort	Quicksort
Vectores	Listas enlazadas
Adaptabilidad a cualquier tamaño	Tamaños pequeños



Como vemos a la hora de comparar los 4 algoritmos de ordenación, la propia escala de la gráfica muestra la gran diferencia entre el orden cuadrático y los de orden nlogn.
Solo empezamos notar la diferencia cuando nos acercamos mucho en el gráfico.



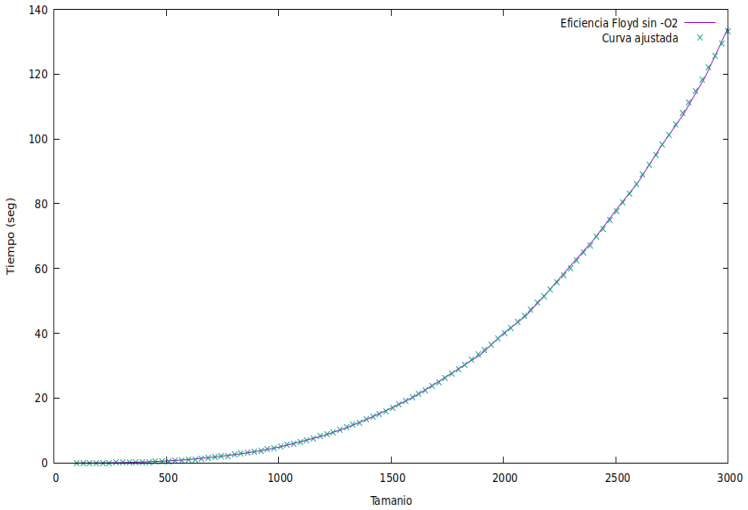
Sin embargo, seguimos viendo como la diferencia entre mergesort y quicksort es mínima, mientras que sigue siendo abismal con respecto a los de orden cuadrático.

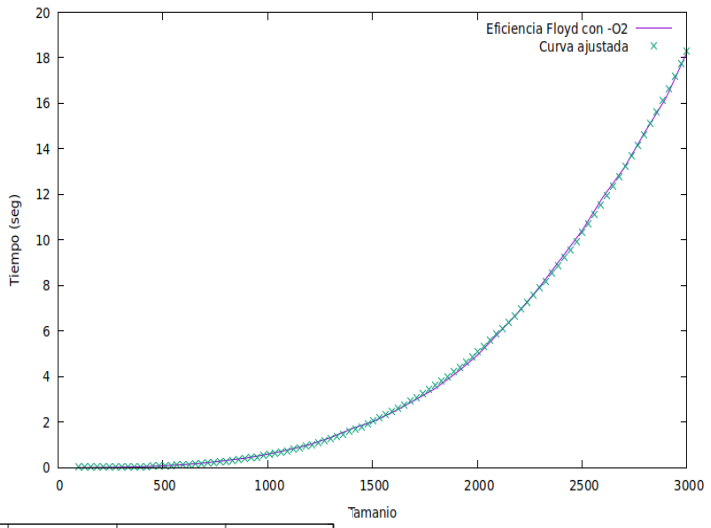
Orden cúbico (n^3)

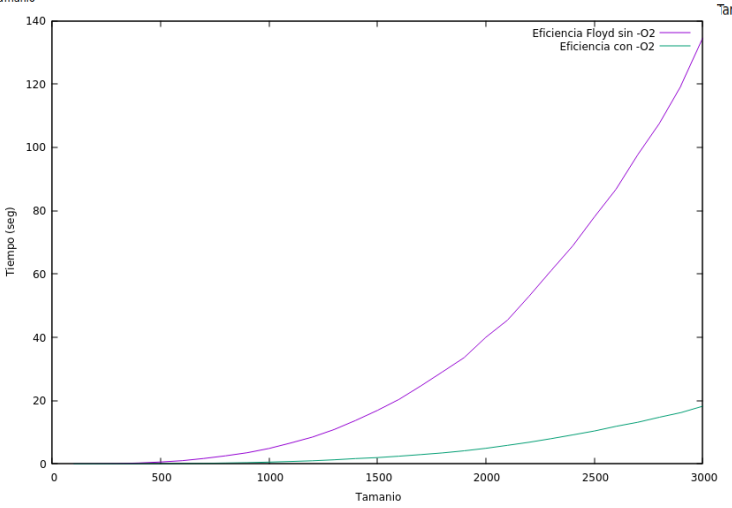
Número de elementos	Tiempo	
	Floyd	Dijkstra
100	0,009388	0,011119
200	0,041783	0,045359
300	0,122609	0,143941
400	0,296711	0,324782
500	0,609276	0,611432
600	1,04486	1,01846
700	1,73139	1,64084
800	2,55977	2,35729
900	3,54449	3,26669
1000	4,88647	4,61268
1100	6,61795	6,08456
1200	8,47532	7,87791
1300	10,8398	10,0183
1400	13,7483	12,3527
1500	16,9078	15,0617
1600	20,3766	18,3661
1700	24,6196	21,8605
1800	29,0723	25,7688
1900	33,5809	30,2394
2000	40,0054	35,0632
2100	45,4465	40,4438
2200	53,0286	46,4513
2300	61,0151	52,5049
2400	68,817	59,5358
2500	78,0031	66,0396
2600	86,7588	74,4151
2700	97,6658	82,7934
2800	10,7572	90,5696
2900	11,9511	10,1935
3000	134,458	111,446

Floyd

El algoritmo de Floyd es un algoritmo que analiza grafos para encontrar el camino mínimo en grafos dirigidos ponderados. La función de dicho algoritmo consiste en encontrar el camino más corto entre todos los pares de nodos o vértices del grafo en una única ejecución. Como resultado, el algoritmo te da el peso mínimo entre los dos pares de nodos que se quiere saber.



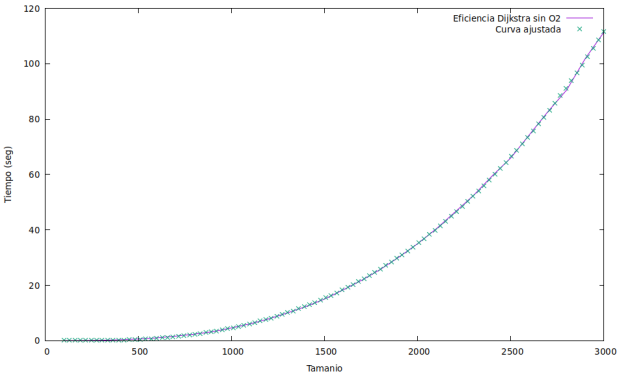


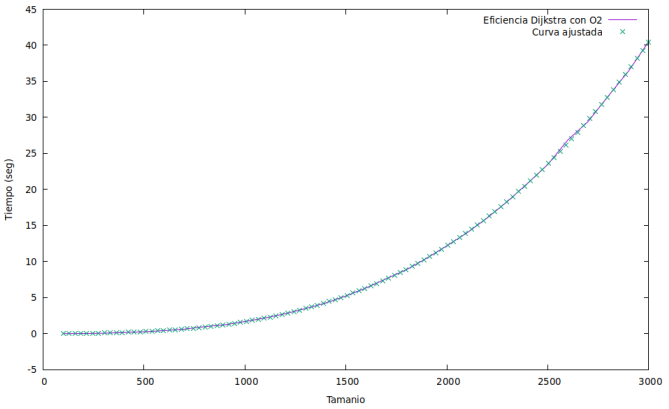


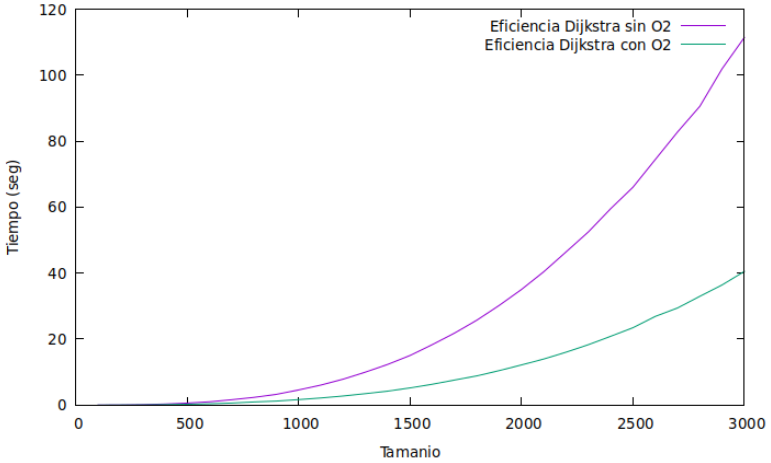
Se ajusta bien

Dijkstra

El algoritmo de Dijkstra, o de caminos mínimos, es un algoritmo para la determinación del camino más corto, dado un vértice origen hacia el resto de vértices del grafo, dependiendo del peso de cada arista.







Para ambos algoritmos de or*d*en de eficiencia cúbico se ha usado un script similar.

```
#!/bin/csh -vx
echo "" >> dijkstra_o2.dat
@ i = 100
while ( $i < 3000 )
./dijkstra_o2 $i >> dijkstra_o2.dat
@ i += 100
end
```

Orden Exponencial

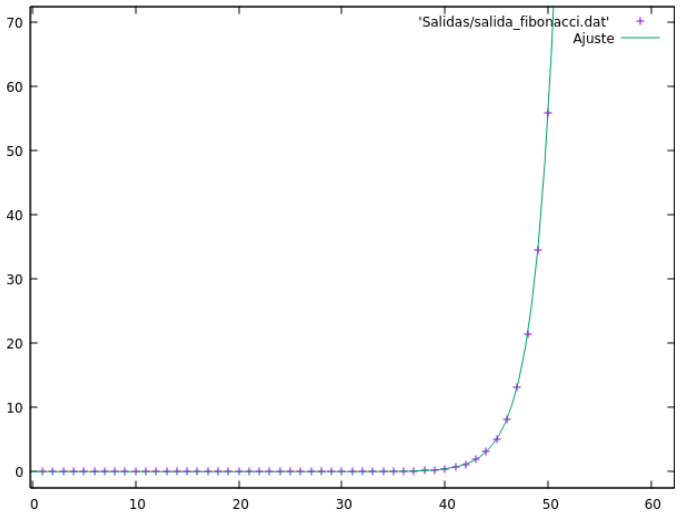
Número de elementos	Tiempo	
	Fibonacci	Hanoi
1	4E-06	3E-06
2	4E-06	4E-06
3	5E-06	6E-06
4	4E-06	6E-06
5	4E-06	5E-06
6	4E-06	9E-06
7	4E-06	6E-06
8	4E-06	8E-06
9	3E-06	1,20E-05
10	4E-06	2,10E-05
11	4E-06	5E-05
12	3E-06	6,80E-05
13	4E-06	0,000132
14	6E-06	0,000258
15	6E-06	0,000386
16	9E-06	0,000615
17	1,40E-05	0,001458
18	2,10E-05	0,002056
19	4,30E-05	0,00353
20	7,10E-05	0,006086
21	7,30E-05	0,010141
22	0,000104	0,017016
23	0,000168	0,033674
24	0,000274	0,067192
25	0,000435	0,133353
26	0,000642	0,266804
27	0,001023	0,53589
28	0,001597	1,06839
29	0,00254	2,15397
30	0,003715	4,28788
31	0,006096	8,55063
32	0,009902	17,1855
33	0,015747	34,3141
34	0,025647	68,6025

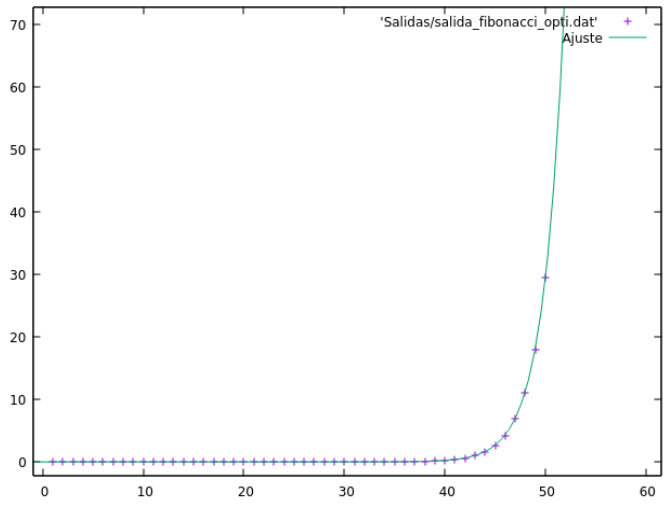
Fibonacci (φ^n)

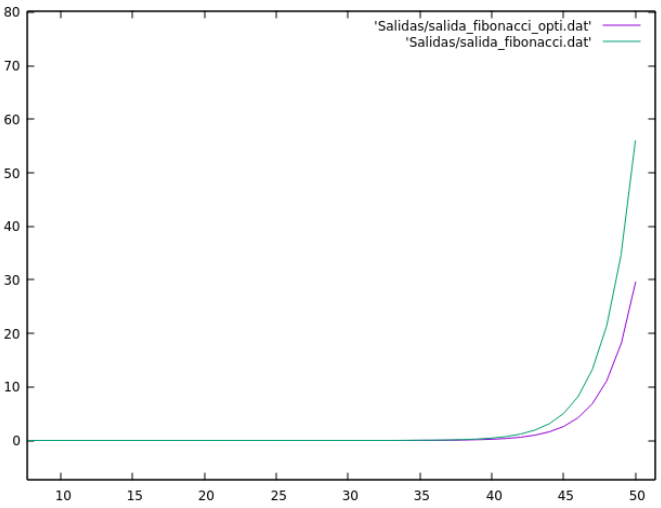
El algoritmo proporcionado que calcula la serie o sucesión de Fibonacci consiste en para tamaños menores que 2 devolver un 1, y para tamaños iguales que dos o mayor sumamos el término anterior y el anterior al anterior. Con este algoritmo podemos obtener el término de la serie que queramos de forma recursiva.

Eficiencia: *$O(\varphi^n)$. φ = número áureo.*

```
#!/bin/bash -vx
i=1
while [ $i -le 50 ]
do
    z=$(./Algoritmos/fibonacci $i)
    echo $i" "$z >> Salidas/salida_fibonacci.dat
    (( i += 1))
done
```





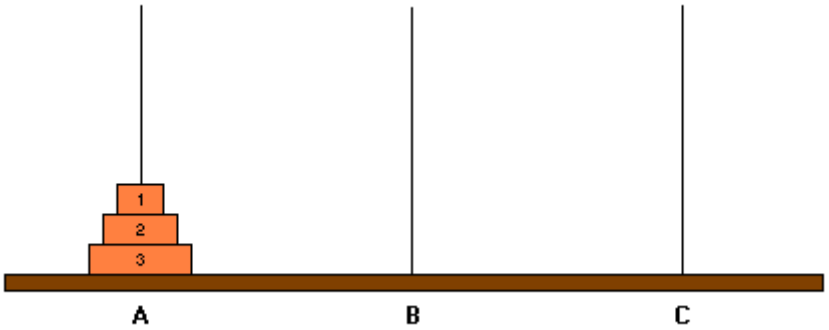


Para poder ejecutar el algoritmo a partir de ciertos tamaños (que tienen asociados un número que supera el límite de las variables int), tuvimos que cambiar el tipo de la variable que almacena el resultado de int a long long int.

Hanoi (2n)

El algoritmo proporcionado que sirve para resolver el problema de las torres de Hanoi con n discos. El problema consiste en pasar todos los discos desde la varilla **A** a la **C** apoyándote en la B cumpliendo dos reglas:

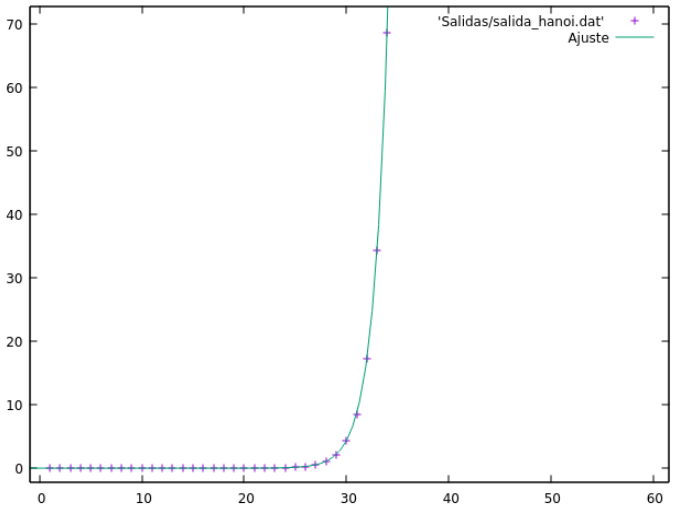
- Un disco de mayor tamaño (mayor número en este dibujo) no puede estar encima de otro con menos tamaño ni viceversa.
- Solo puedes mover un disco a la vez.

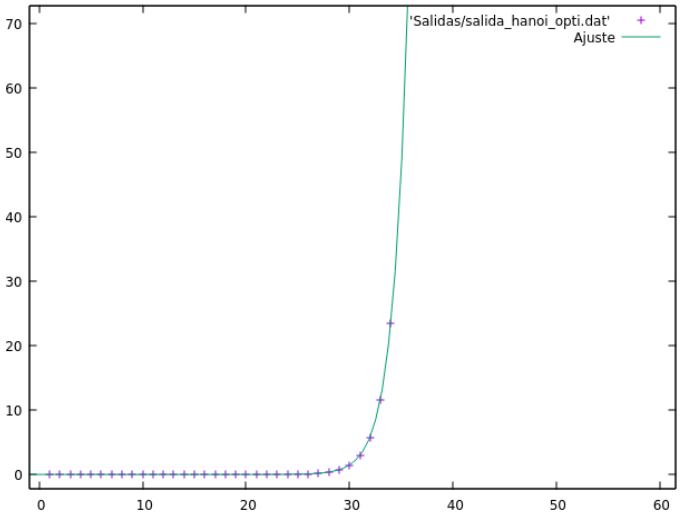


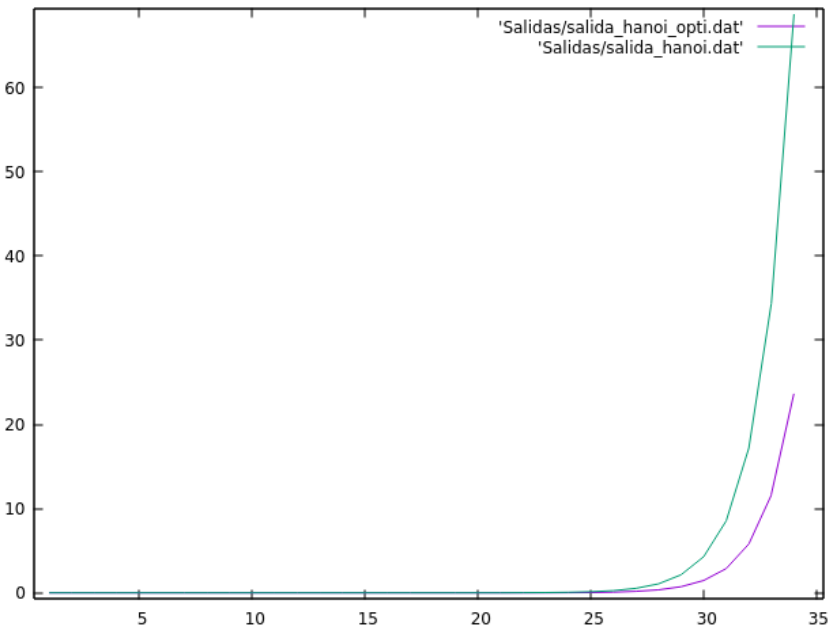
El algoritmo consiste en mover primero los n-1 primeros discos (los que están más arriba) a la varilla auxiliar para poder mover el último a la varilla destino y luego mover los n-1 discos de la varilla auxiliar a la varilla destino; esto siempre que el número de discos sea mayor. En el caso de que queramos mover 1 disco simplemente bastaría con moverlo a la varilla destino. Así definimos la recursividad y el caso base.

Eficiencia: $O(2^n)$.

```
#!/bin/bash -vx
i=1
while [ $i -le 34 ]
do
	z=$(./Algoritmos/hanoi $i)
	echo $i" "$z >> Salidas/salida_hanoi_opti.dat
	(( i += 1))
done
```


'Salidas/salida_hanoi.dat'
Ajuste
0
10
20
30
40
50
60
70


'Salidas/salida_hanoi_opti.dat'
Ajuste
0
10
20
30
40
50
60
70


'Salidas/salida_hanoi_opti.dat'
'Salidas/salida_hanoi.dat'
0
10
20
30
40
50
60
5
10
15
20
25
30
35

Conclusión

Como hemos podido observar en todos los algoritmos estudiados, la opción de optimización de g++ -O2 mejora considerablemente el tiempo de ejecución de cada uno de estos, sobre todo en los de Hanoi y Fibonacci.

Porcentaje de participación

Todos los alumnos del grupo estamos de acuerdo en que los 5 alumnos hemos tenido la siguiente participación en la práctica:

Salvador Romero Cortés	20%
Abel Ríos González	20%
Raúl Durán Racero	20%
Alberto Palomo Campos	20%
Manuel Contreras Orge	20%