Análisis de eficiencia de algoritmos

Introducción

Hemos repartido los algoritmos de la siguiente manera para agilizar el trabajo:

- Burbuja e inserción: Ana García Muñoz (intel core i-7 8ª Gen)
- Fibonacci y Hanoi: José María Ramírez González (intel core i-7 8ª Gen)
- Quicksort y Mergesort: Manuel Sánchez Pérez (intel core i-5 8ª Gen)
- Floyd y Dijkstra: Víctor Gutiérrez Rubiño. (AMD Ryzen 5 2600X)

El trabajo realizado ha sido repartido equitativamente entre los cuatro miembros del equipo (25% cada uno).

Ejecución de la práctica

Vamos a tratar los algoritmos en el siguiente orden:

1. Algoritmo de Burbuja.
2. Algoritmo de Inserción.
3. Algoritmo de Fibonacci.
4. Algoritmo de Hanoi.
5. Algoritmo de ordenación Quicksort.
6. Algoritmo de ordenación Mergesort.
7. Algoritmo de Floyd.
8. Algoritmo de Dijkstra.
9. Comparación de los algoritmos de ordenación.
10. Comparación de todo el conjunto de algoritmos.

Eficiencia Algoritmo Burbuja

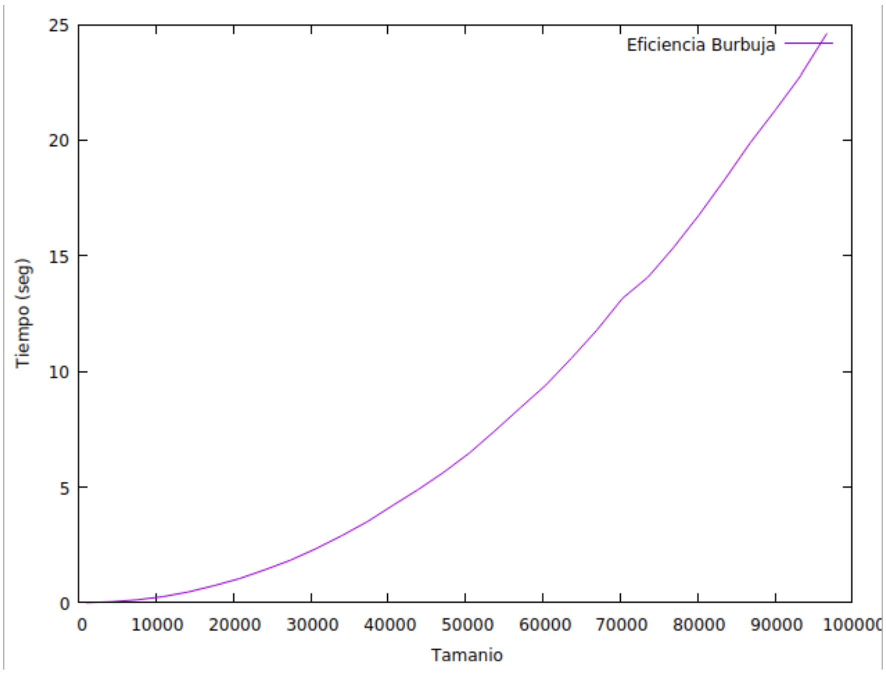
```
static void burbuja_lims(int T[], int inicial, int final)
{
  int i, j;
  int aux;
  for (i = inicial; i < final - 1; i++)
    for (j = final - 1; j > i; j--)
      if (T[j] < T[j - 1])
      {
        aux = T[j];
        T[j] = T[j - 1];
        T[j - 1] = aux;
      }
}
```

Eficiencia empírica

Tras ejecutar el algoritmo variando el tamaño del vector T desde 1000 hasta 100000, en tramos de 3300, he obtenido los siguientes datos:

Algoritmo Ordenación por Burbuja
O(n^2)

Tamaño	Tiempo
1000	0.0039
4300	0.040911
7600	0.127747
10900	0.263351
14200	0.46848
17500	0.738022
20800	1.04364
24100	1.42484
27400	1.84068
30700	2.33643
34000	2.89388
37300	3.50475
40600	4.2061
43900	4.89083
47200	5.64233
50500	6.46958
53800	7.42745
57100	8.42741
60400	9.41499
63700	10.5693
67000	11.7918
70300	13.1632
73600	14.0783
76900	15.3581
80200	16.7648
83500	18.2932
86800	19.8774
90100	21.319
93400	22.8115
96700	24.581



Como se puede ver en la gráfica, la función crece cuadráticamente, tal y como se esperaba.

Estudio con factores externos

Los datos expuestos en el apartado anterior se han obtenido sin especificar optimización, es decir, compilando con el siguiente comando:

g++ -o burbuja burbuja.cpp

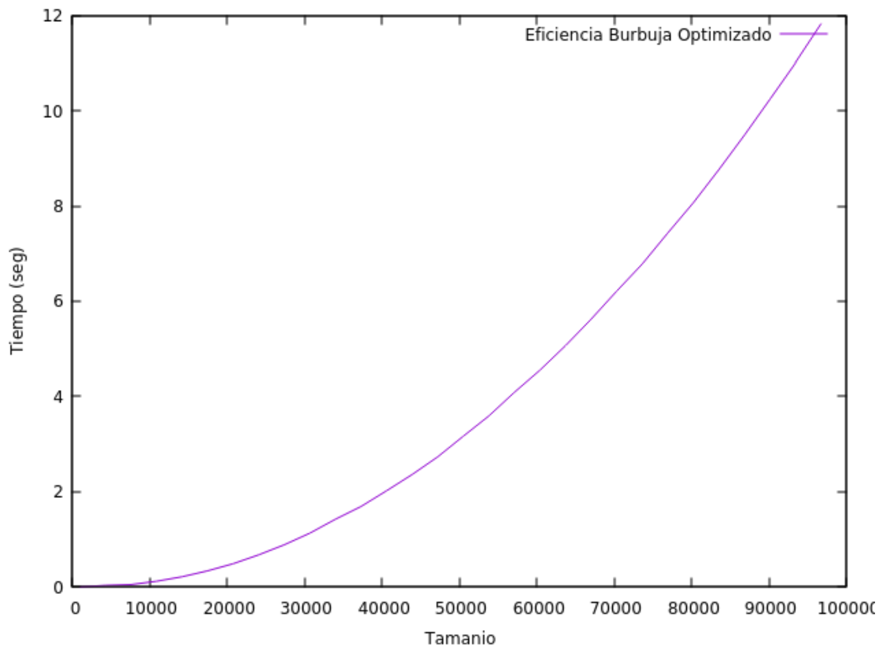
La pregunta es : ¿ Cambiará la eficiencia empírica si cambiamos la optimización con -O2? Vamos a comprobarlo. Compilaré el mismo código con el comando

g++ -O2 -o burbuja burbuja.cpp

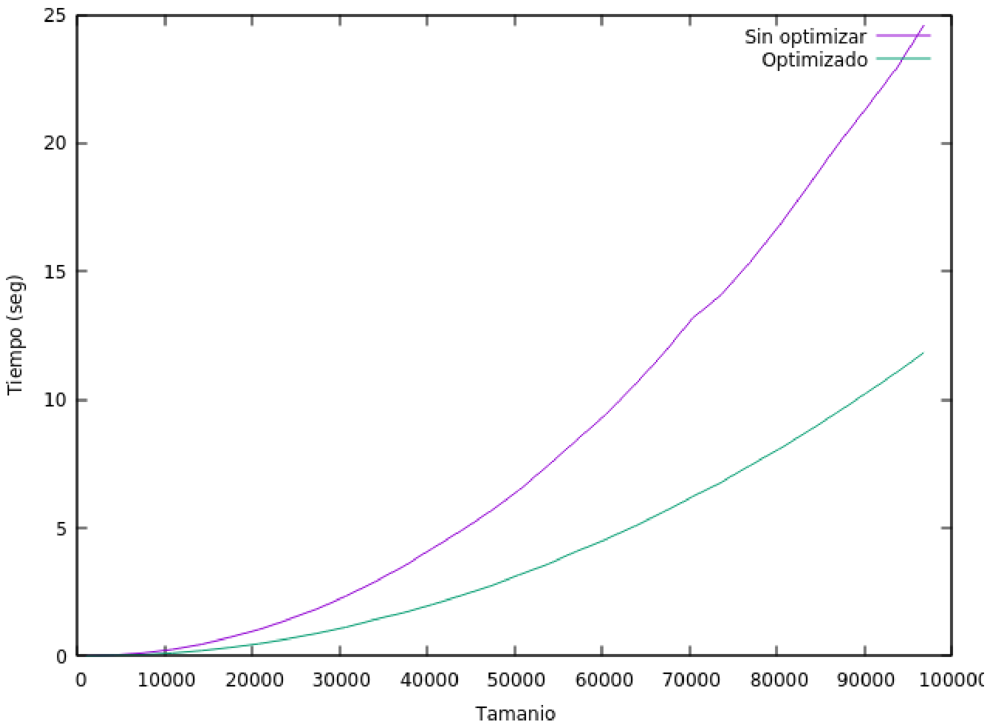
y lo ejecutaré con los mismos datos. De esta ejecución, he obtenido lo siguiente:

Burbuja optimizado

Tamaño	Tiempo
1000	0.002694
4300	0.037059
7600	0.055436
10900	0.122644
14200	0.217846
17500	0.341717
20800	0.492288
24100	0.680431
27400	0.889332
30700	1.13397
34000	1.4245
37300	1.69241
40600	2.02283
43900	2.36392
47200	2.73446
50500	3.1671
53800	3.593
57100	4.08738
60400	4.55384
63700	5.06978
67000	5.62142
70300	6.21166
73600	6.77735
76900	7.43339
80200	8.06921
83500	8.76242
86800	9.48719
90100	10.2386
93400	11.0023
96700	11.8091



Si comparamos las dos gráficas, podemos ver que, al compilar el código optimizándolo , hemos conseguido reducir a la mitad el tiempo de ejecución:



Eficiencia híbrida

Vamos a comprobar que el algoritmo es O(n^2). Para ello, definiremos una función para ajustar a los datos. En este caso, dicha función es cuadrática, así que la definiremos de la siguiente manera:

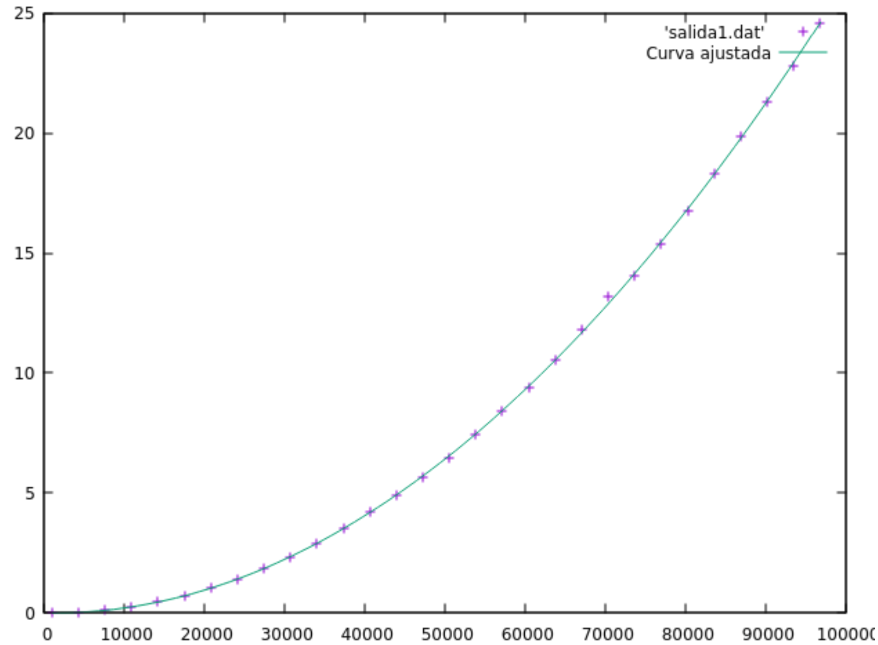
$$T(n) = a_0 \times n^2 + a_1 \times n + a_2$$

Para obtener las constantes ocultas, utilizaremos gnuplot para realizar la regresión, obteniendo los siguientes datos:

```
gnuplot> fit f(x) 'salida1.dat' via a0,a1,a2
iter      chisq       delta/lim  lambda   a0            a1            a2
   0 5.5717465220e+20   0.00e+00  2.49e+09    1.000000e+00   1.000000e+00   1.000000e+00
   1 6.7283504630e+16  -8.28e+08  2.49e+08    1.097644e-02   9.999874e-01   1.000000e+00
   2 6.8263503842e+09  -9.86e+11  2.49e+07   -1.149100e-05   9.999872e-01   1.000000e+00
   3 5.9957583356e+09  -1.39e+04  2.49e+06   -1.271175e-05   9.999775e-01   1.000000e+00
   4 5.9841617883e+09  -1.94e+02  2.49e+05   -1.269945e-05   9.990099e-01   1.000000e+00
   5 4.9740644353e+09  -2.03e+04  2.49e+04   -1.157788e-05   9.107976e-01   9.999964e-01
   6 4.3570110661e+07  -1.13e+07  2.49e+03   -1.080842e-06   8.520084e-02   9.999628e-01
   7 4.9949070148e+01  -8.72e+10  2.49e+02    1.921182e-09   4.104221e-05   9.999587e-01
   8 3.5909738193e+00  -1.29e+06  2.49e+01    3.039168e-09  -4.688712e-05   9.999027e-01
   9 3.5524812458e+00  -1.08e+03  2.49e+00    3.037257e-09  -4.666135e-05   9.943348e-01
  10 1.5430943267e+00  -1.30e+05  2.49e-01    2.914465e-09  -3.217997e-05   6.386268e-01
  11 1.5788967152e-01  -8.77e+05  2.49e-02    2.701025e-09  -7.008158e-06   2.032810e-02
  12 1.5747101832e-01  -2.66e+02  2.49e-03    2.697250e-09  -6.562956e-06   9.392528e-03
  13 1.5747101831e-01  -8.32e-06  2.49e-04    2.697249e-09  -6.562877e-06   9.390594e-03
iter      chisq       delta/lim  lambda   a0            a1            a2

After 13 iterations the fit converged.
final sum of squares of residuals : 0.157471
rel. change during last iteration : -8.31537e-11

degrees of freedom    (FIT_NDF)                        : 27
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 0.0763692
variance of residuals (reduced chisquare) = WSSR/ndf   : 0.00583226

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a0              = 2.69725e-09      +/- 1.914e-11    (0.7096%)
a1              = -6.56288e-06     +/- 1.933e-06    (29.45%)
a2              = 0.00939059       +/- 0.04082      (434.7%)

correlation matrix of the fit parameters:
                a0     a1     a2
a0              1.000
a1             -0.968  1.000
a2              0.736 -0.860  1.000
gnuplot>
```

Por lo que la función ajustada, que podemos usar para calcular el tiempo de ejecución del algoritmo para una entrada de tamaño n, nos queda:

$$T(n) = 2.697 \times 10^{-9} \times n^2 + (-6.56) \times 10^{-6} \times n + 0.00939$$



Esta gráfica nos muestra el ajuste de datos.

Eficiencia Inserción

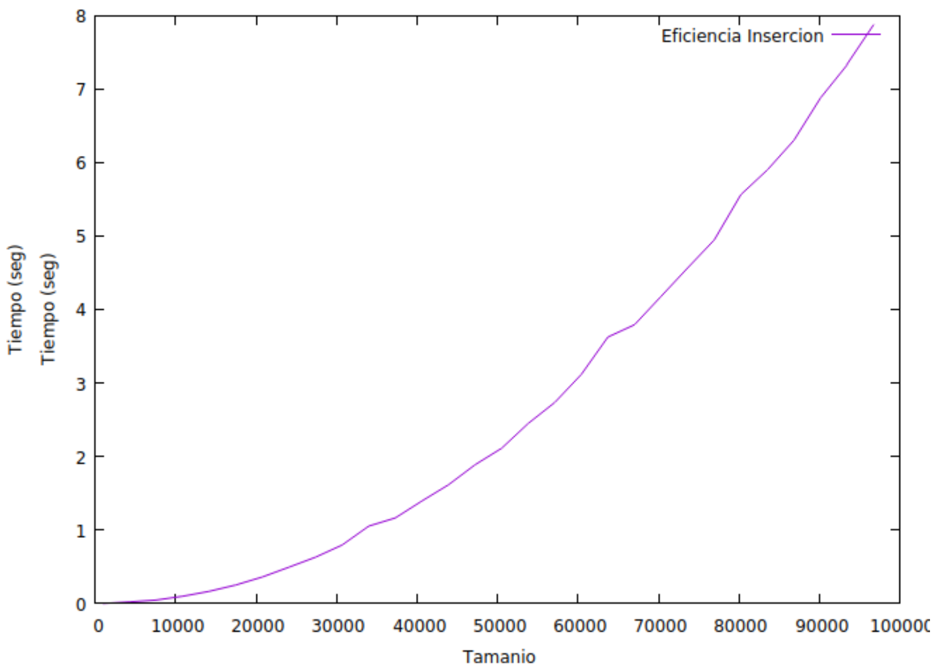
```
static void insercion_lims(int T[], int inicial, int final)
{
  int i, j;
  int aux;
  for (i = inicial + 1; i < final; i++)
  {
    j = i;
    while ((T[j] < T[j - 1]) && (j > 0))
    {
      aux = T[j];
      T[j] = T[j - 1];
      T[j - 1] = aux;
      j--;
    };
  };
}
```

Eficiencia empírica

Tras ejecutar el algoritmo variando el tamaño del vector T desde 1000 hasta 100000, en tramos de 3300, he obtenido los siguientes datos:

Algoritmo Ordenación por Inserción
O(n^2)

Tamaño	Tiempo
1000	0.00213
4300	0.022229
7600	0.048676
10900	0.099888
14200	0.166685
17500	0.252983
20800	0.363227
24100	0.494423
27400	0.630553
30700	0.796718
34000	1.05285
37300	1.16478
40600	1.39561
43900	1.61658
47200	1.88819
50500	2.11187
53800	2.45055
57100	2.73477
60400	3.11672
63700	3.62391
67000	3.79234
70300	4.17776
73600	4.56448
76900	4.94545
80200	5.56191
83500	5.89947
86800	6.29926
90100	6.87548
93400	7.32517
96700	7.86383



Como se puede ver en la gráfica, la función crece cuadráticamente, tal y como se esperaba.

Estudio con factores externos

Los datos expuestos en el apartado anterior se han obtenido sin especificar optimización, es decir, compilando con el siguiente comando:

g++ -o insercion insercion.cpp

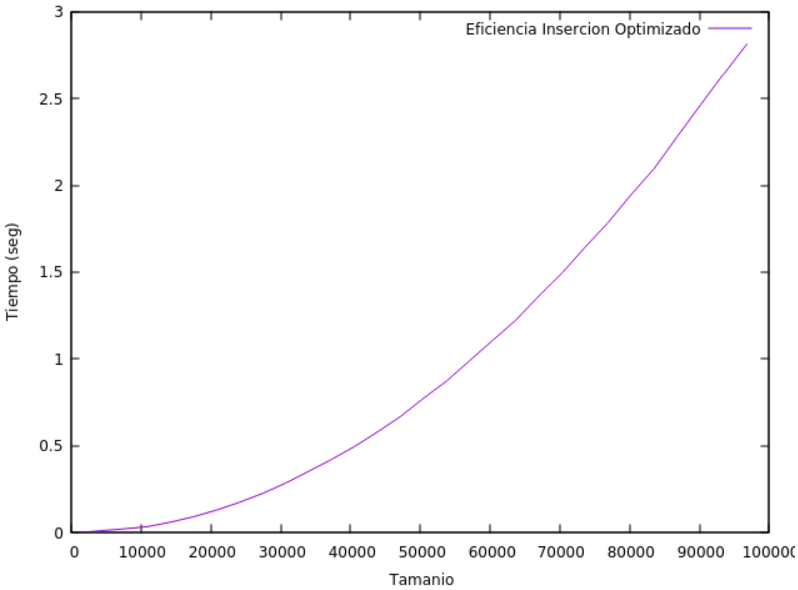
La pregunta es : ¿ Cambiará la eficiencia empírica si cambiamos la optimización con -O2? Vamos a comprobarlo. Compilaré el mismo código con el comando

g++ -O2 -o insercion insercion.cpp

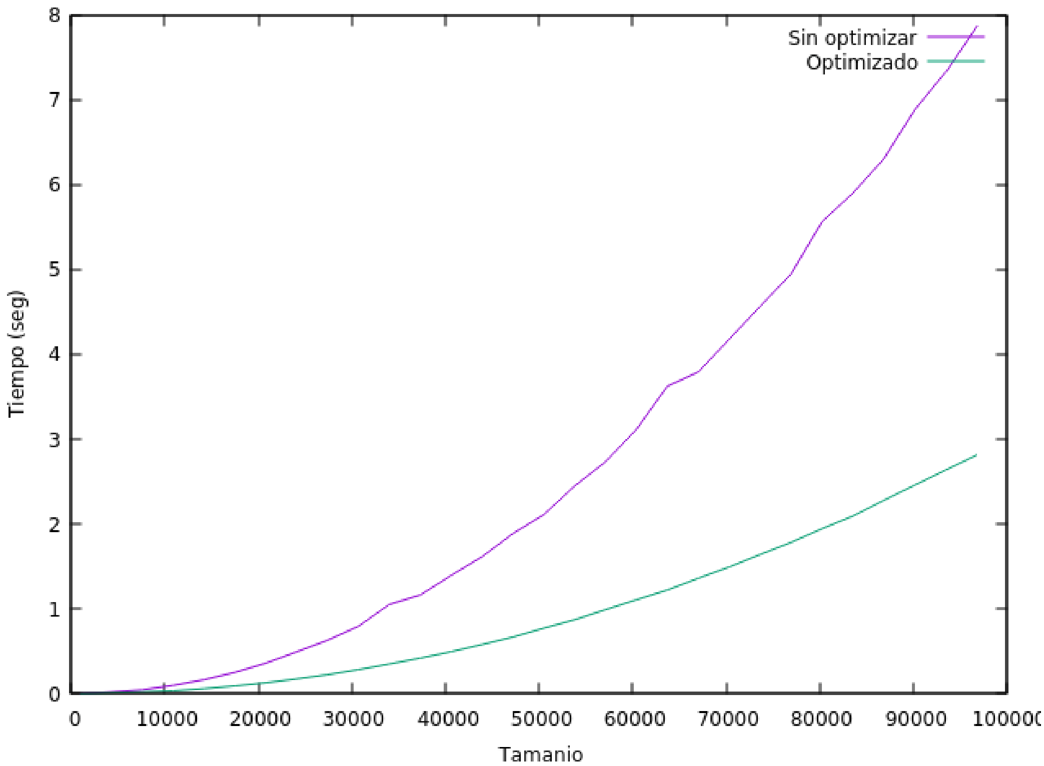
y lo ejecutaré con los mismos datos. De esta ejecución, he obtenido lo siguiente:

Insercion Optimizado

Tamaño	Tiempo
1000	0.000696
4300	0.013002
7600	0.023829
10900	0.036121
14200	0.061574
17500	0.093172
20800	0.131264
24100	0.176258
27400	0.22669
30700	0.286746
34000	0.353721
37300	0.42296
40600	0.497315
43900	0.581564
47200	0.670613
50500	0.776019
53800	0.874429
57100	0.991637
60400	1.10871
63700	1.22381
67000	1.36313
70300	1.49434
73600	1.64282
76900	1.78487
80200	1.94501
83500	2.09582
86800	2.27893
90100	2.45872
93400	2.63732
96700	2.81077



Esta vez, si comparamos las dos gráficas, podemos ver que, al compilar el código optimizándolo , hemos conseguido reducir casi tres veces el tiempo de ejecución:



Eficiencia híbrida

Vamos a comprobar que el algoritmo es O(n^2). Para ello, definiremos una función para ajustar a los datos. En este caso, dicha función es cuadrática, así que la definiremos de la siguiente manera:

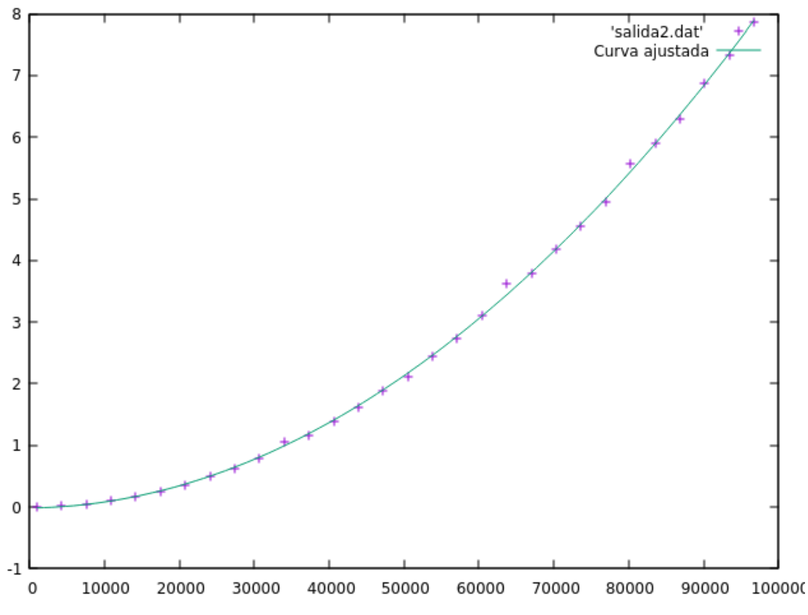
$$T(n) = a_0 \times n^2 + a_1 \times n + a_2$$

Para obtener las constantes ocultas, utilizaremos gnuplot para realizar la regresión, obteniendo los siguientes datos:

```
gnuplot> fit f(x) 'salida2.dat' via a0,a1,a2
iter      chisq       delta/lim  lambda   a0            a1            a2
   0 5.5717465417e+20   0.00e+00  2.49e+09    1.000000e+00   1.000000e+00   1.000000e+00
   1 6.7283504868e+16  -8.28e+08  2.49e+08    1.097644e-02   9.999874e-01   1.000000e+00
   2 6.8262672176e+09  -9.86e+11  2.49e+07   -1.149277e-05   9.999872e-01   1.000000e+00
   3 5.9956751662e+09  -1.39e+04  2.49e+06   -1.271352e-05   9.999775e-01   1.000000e+00
   4 5.9840787798e+09  -1.94e+02  2.49e+05   -1.270122e-05   9.990100e-01   1.000000e+00
   5 4.9739954383e+09  -2.03e+04  2.49e+04   -1.157966e-05   9.107982e-01   9.999964e-01
   6 4.3569506368e+07  -1.13e+07  2.49e+03   -1.082692e-06   8.520719e-02   9.999628e-01
   7 5.0032928545e+01  -8.71e+10  2.49e+02    6.290890e-11   4.797755e-05   9.999587e-01
   8 3.6754559068e+00  -1.26e+06  2.49e+01    1.180886e-09  -3.995111e-05   9.999013e-01
   9 3.6350441936e+00  -1.11e+03  2.49e+00    1.178928e-09  -3.971976e-05   9.941963e-01
  10 1.5254744515e+00  -1.38e+05  2.49e-01    1.053112e-09  -2.488177e-05   6.297288e-01
  11 7.1207083004e-02  -2.04e+06  2.49e-02    8.344158e-10   9.099113e-07  -3.795804e-03
  12 7.0767556840e-02  -6.21e+02  2.49e-03    8.305478e-10   1.366077e-06  -1.500067e-02
  13 7.0767556827e-02  -1.94e-05  2.49e-04    8.305471e-10   1.366158e-06  -1.500265e-02
iter      chisq       delta/lim  lambda   a0            a1            a2

After 13 iterations the fit converged.
final sum of squares of residuals : 0.0707676
rel. change during last iteration : -1.94286e-10

degrees of freedom    (FIT_NDF)                        : 27
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 0.0511959
variance of residuals (reduced chisquare) = WSSR/ndf   : 0.00262102

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a0              = 8.30547e-10      +/- 1.283e-11    (1.545%)
a1              = 1.36616e-06      +/- 1.296e-06    (94.83%)
a2              = -0.0150027       +/- 0.02737      (182.4%)

correlation matrix of the fit parameters:
                a0     a1     a2
a0              1.000
a1             -0.968  1.000
a2              0.736 -0.860  1.000
gnuplot> 
```

Por lo que la función ajustada, que podemos usar para calcular el tiempo de ejecución del algoritmo para una entrada de tamaño n, nos queda:

$$T(n) = 8.36255 \times 10^{-10} \times n^2 + 1.1366 \times 10^{-6} \times n - 0.015$$



Esta gráfica nos muestra el ajuste de datos.

Eficiencia Algoritmo Fibonacci

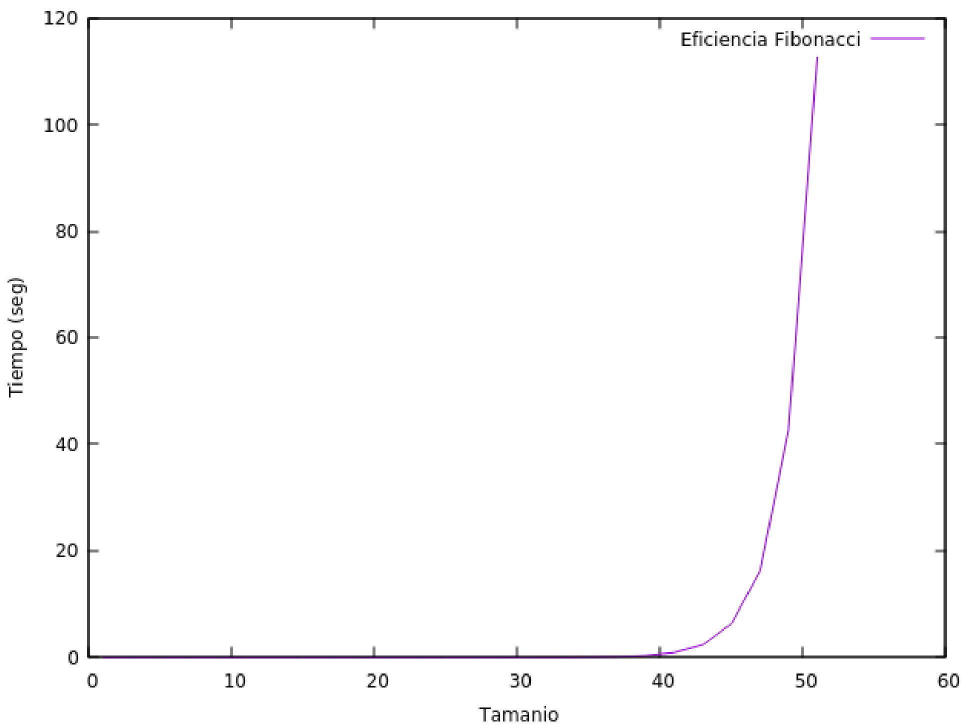
```
int fibo(int n)
{
  if (n < 2)
    return 1;
  else
    return fibo(n-1) + fibo(n-2);
}
```

Eficiencia empírica

Iteramos el programa obteniendo los tiempos de ejecución para distintos tamaños, en este caso, la tabla resultante es la siguiente, donde la 1ª columna se corresponde con el tamaño y la siguiente con el tiempo.

1	6.93e-07
3	6.91e-07
5	9.9e-07
7	1.655e-06
9	2.932e-06
11	3.353e-06
13	6.123e-06
15	1.202e-05
17	2.1755e-05
19	6.5504e-05
21	0.000137366
23	0.00037396
25	0.000949206
27	0.00245546
29	0.00705438
31	0.0252347
33	0.026479
35	0.055805
37	0.131359
39	0.359563
41	0.916804
43	2.38045
45	6.33156
47	16.2703
49	42.7859
51	112.571

De aquí, con la ayuda de gnuplot podemos realizar la gráfica, que queda tal que



Como vemos, el tiempo crece de forma exponencial, tal y como debería según lo calculado teóricamente (O(1.68^n)), la curva no queda muy ajustada, pero con el próximo cálculo de la eficiencia híbrida quedará más suavizada.

Eficiencia híbrida

A la hora de ajustar la curva obtenida para encontrar las constantes ocultas, usamos una función exponencial, que es la que más se asemeja a la obtenida mediante el cálculo de la eficiencia empírica.

La fórmula usada es tal que

$$f(n) = a_0 \times 1.68^n$$

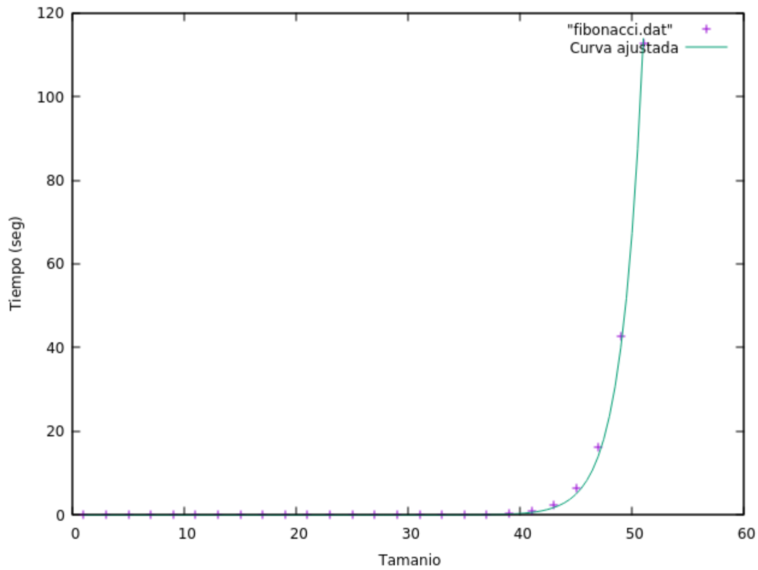
Donde a_0 es la constante oculta y n el tamaño de la muestra.

Al ajustar la curva, nos sale el siguiente resultado

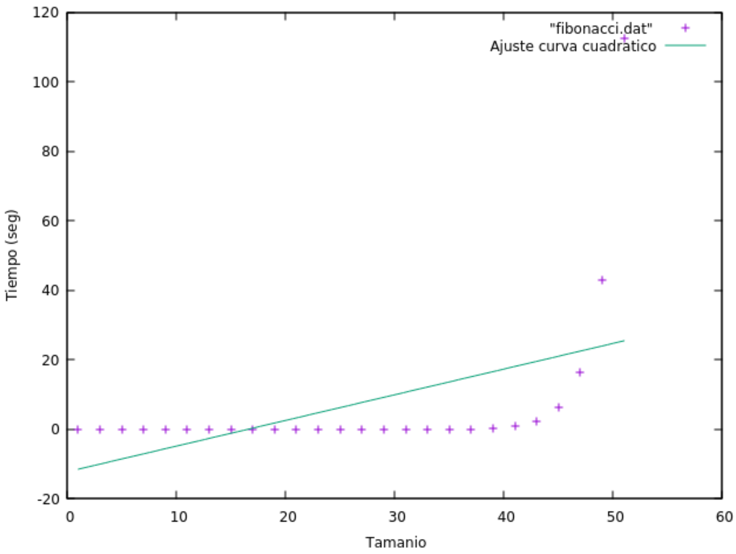
```
Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a0              = 3.67485e-10      +/- 2.227e-12    (0.6059%)
```

De aquí deducimos que a_0 está bastante bien ajustada, ya que el error es bastante pequeño y también podemos ver el valor que tomaría a_0.

Al realizar la gráfica con el ajuste obtenemos lo siguiente.



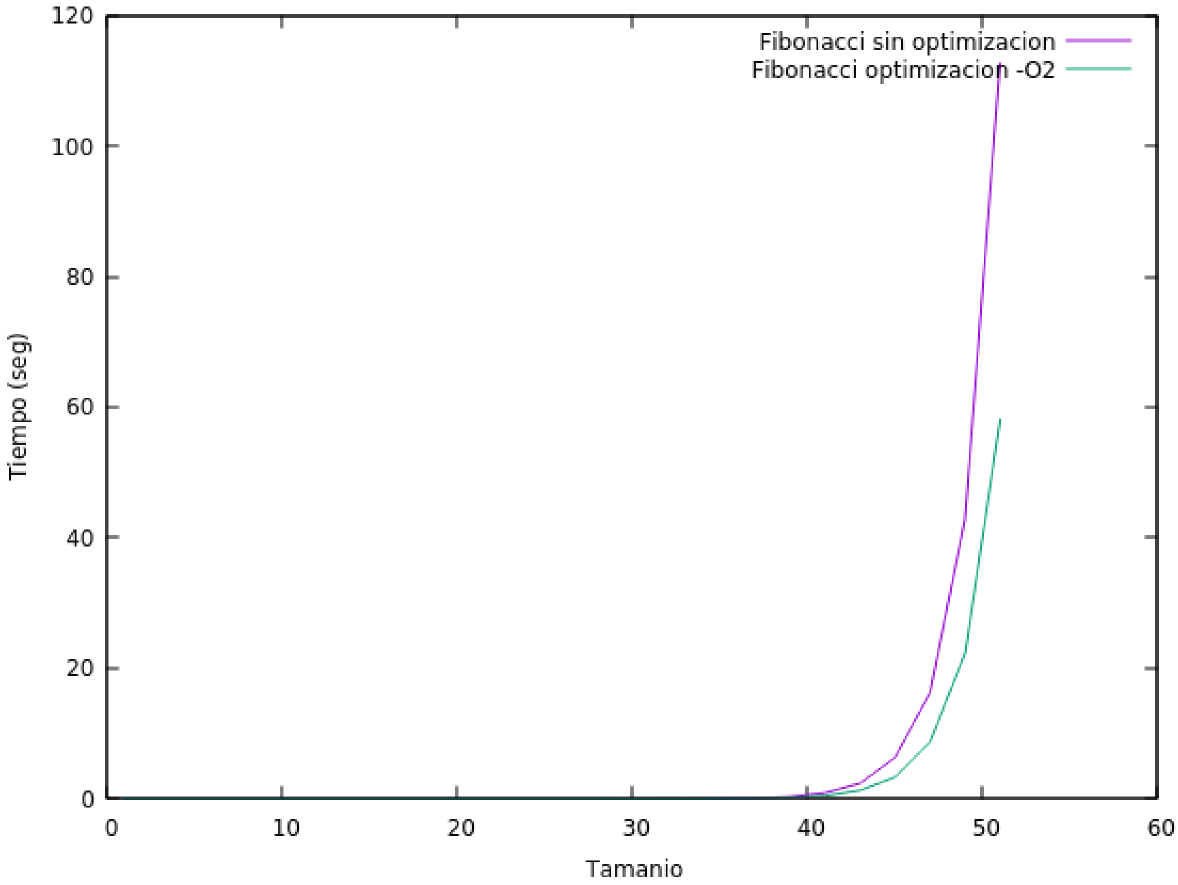
Como vemos, los datos encajan bastante bien y el ajuste es bastante preciso. Esto supone que podríamos usar el valor de la constante oculta para suponer tiempos de ejecución para un tamaño cualquiera dado.



Si intentamos realizar un ajuste cuadrático, por ejemplo, que no se corresponde tanto con la naturaleza de los datos obtenidos en la eficiencia empírica, obtendríamos un ajuste pésimo, que no podría usarse para suponer tiempos de ejecución, como se observa en la siguiente figura.

Efecto de factores externos

Para comprobar cómo afectan los factores externos a la eficiencia de este algoritmo, hemos compilado con la opción de optimización -O2, lo que generará un código más optimizado que si no tuviéramos esa opción activada, esto se puede ver claramente en la siguiente gráfica, que compara las ejecuciones del programa compilado con y sin esa opción.



Como se puede observar, la velocidad de ejecución mejora notablemente cuando realizamos la optimización del programa. No obstante, el crecimiento y la función asociada a la gráfica optimizada seguiría siendo la misma, lo único que cambiaría sería la constante oculta de la misma.

Eficiencia algoritmo Hanoi

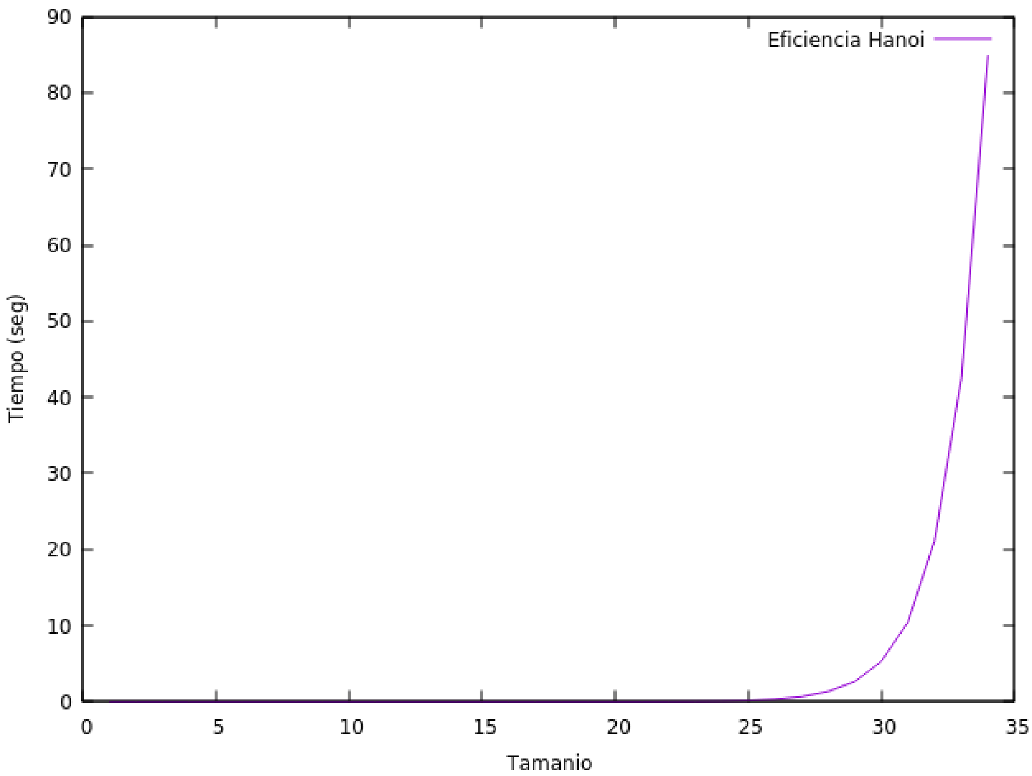
```
void hanoi (int M, int i, int j)
{
  if (M > 0)
   {
    hanoi(M-1, i, 6-i-j);
    //cout << i << " -> " << j << endl;
    hanoi (M-1, 6-i-j, j);
   }
}
```

Eficiencia empírica

Iteramos el programa obteniendo los tiempos de ejecución para distintos tamaños, en este caso, la tabla resultante es la siguiente, donde la 1ª columna se corresponde con el tamaño y la siguiente con el tiempo.

1	1.5e-07
2	2.13e-07
3	2.34e-07
4	3.45e-07
5	5.24e-07
6	8.07e-07
7	1.14e-06
8	1.878e-06
9	3.082e-06
10	8.237e-06
11	1.0599e-05
12	2.0767e-05
13	4.0564e-05
14	8.0061e-05
15	0.0001589
16	0.00033752
17	0.000644336
18	0.0013213
19	0.00254448
20	0.00517501
21	0.0106172
22	0.0208471
23	0.0411362
24	0.0822029
25	0.166658
26	0.325503
27	0.662341
28	1.33403
29	2.64471
30	5.28941
31	10.4877
32	21.17
33	42.4427
34	84.7686

Haciendo uso de esos datos y gnuplot, podemos ver la gráfica de este algoritmo



Como vemos, se produce un salto exponencial a partir de 27 elementos, haciéndose casi imposible el tomar muestras en un tiempo razonable a partir de los 34 elementos.

Eficiencia híbrida

A la hora de ajustar la curva obtenida para encontrar las constantes ocultas, usamos una función exponencial, que es la que más se asemeja a la obtenida mediante el cálculo de la eficiencia empírica.
La fórmula usada es tal que

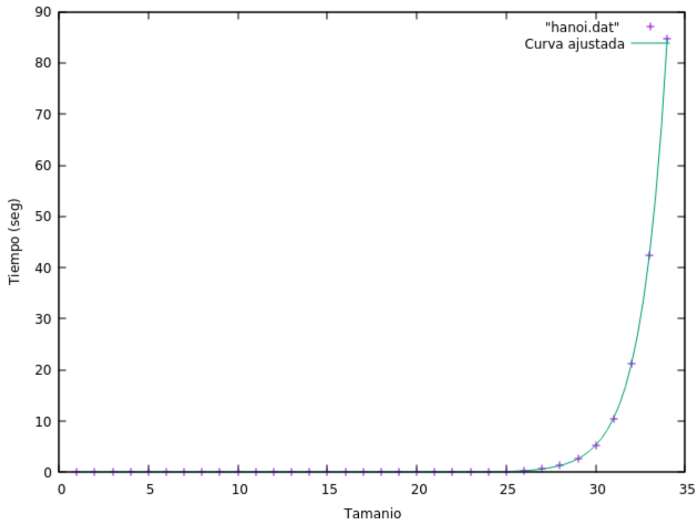
$$f(n) = a_0 \times 2^n$$

Donde a_0 es la constante oculta y n el tamaño de la muestra.

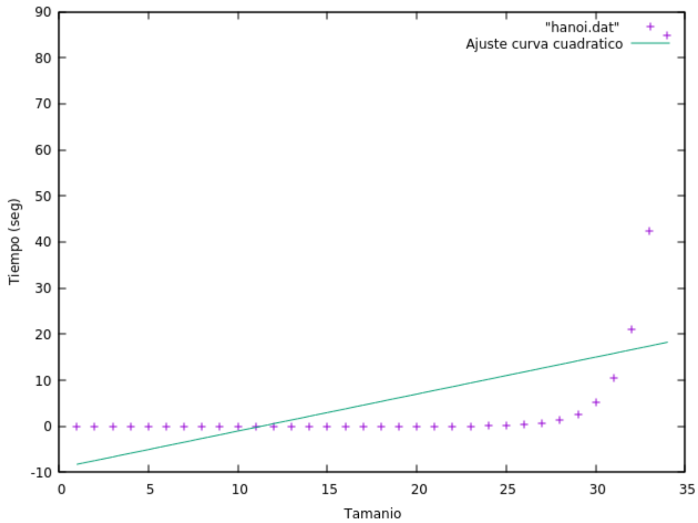
Al ajustar la curva, nos sale el siguiente resultado

```
Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a0              = 4.9346e-09       +/- 1.103e-12    (0.02235%)
```

Aquí podemos ver el valor de a_0, que es del orden de 10^{-9} y el error asociado, que es bastante pequeño, lo que supone un buen ajuste de la función a los datos obtenidos.
Al realizar una gráfica del ajuste y los datos obtenidos, obtenemos lo siguiente



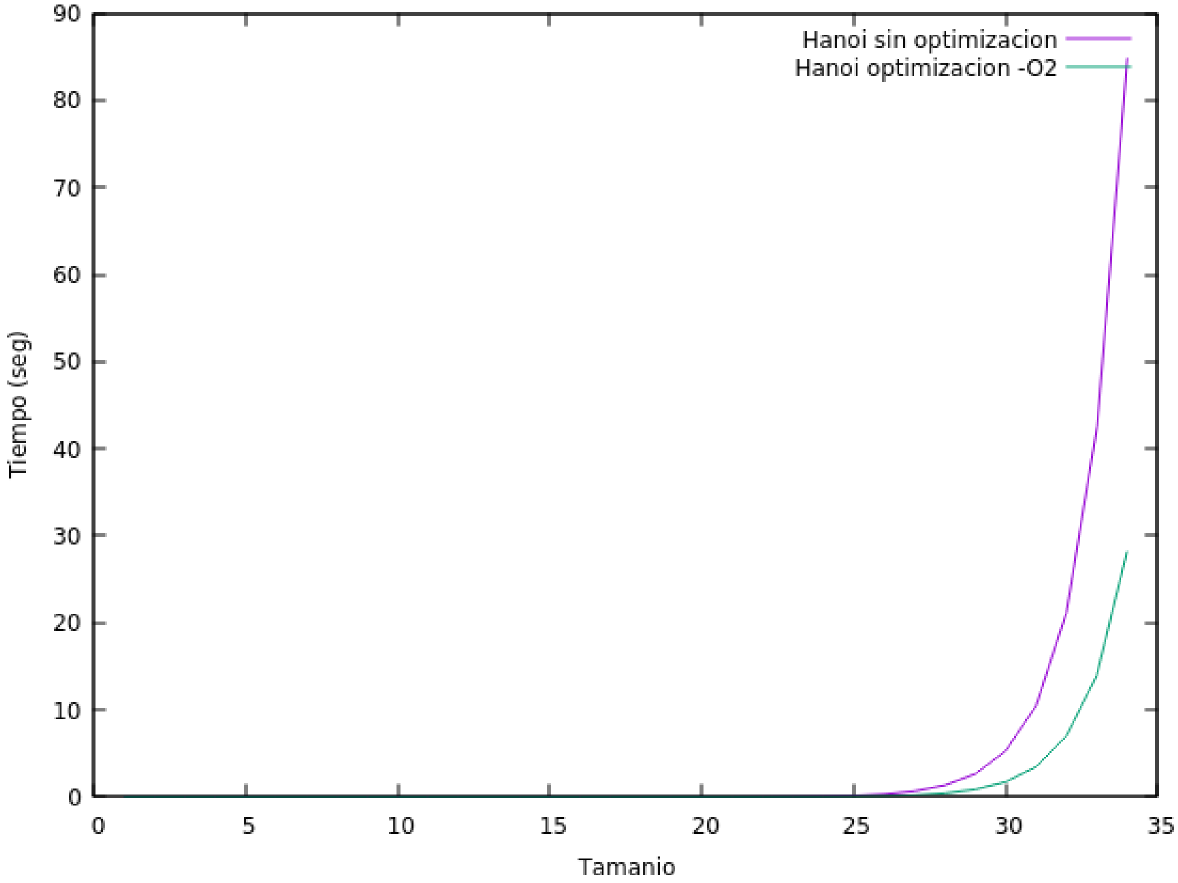
Como vemos, se ajusta bastante bien con los datos, lo que significa que podríamos usar el valor de la constante oculta para suponer tiempos de ejecución para cualquier tamaño de entrada.



En cambio, si intentamos realizar un ajuste a una función cuadrática, por ejemplo, obtendríamos un ajuste pésimo, que se ve bastante claro en la siguiente gráfica. Esto no podría usarse para suponer tiempos de ejecución, debido al mal ajuste de la función cuadrática.

Efecto de factores externos

Para comprobar cómo afectan los factores externos a la eficiencia de este algoritmo, hemos compilado con la opción de optimización -O2, lo que generará un código más optimizado que si no tuviéramos esa opción activada, esto se puede ver claramente en la siguiente gráfica, que compara las ejecuciones del programa compilado con y sin esa opción.



Como se puede observar, la velocidad de ejecución mejora notablemente cuando realizamos la optimización del programa. No obstante, el crecimiento y la función asociada a la gráfica optimizada seguiría siendo la misma, lo único que cambiaría sería la constante oculta de la misma.

Eficiencia Algoritmo Quicksort

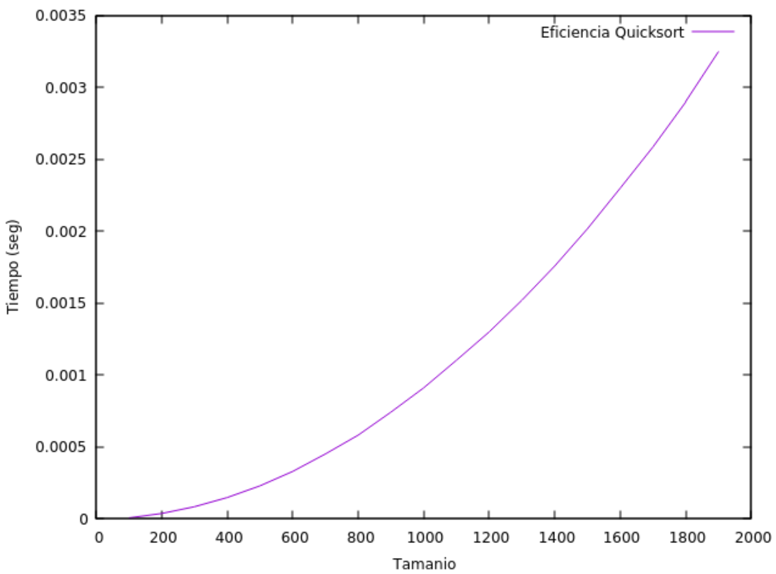
```
static void quicksort_lims(int T[], int inicial, int final)
{
  int k;
  if (final - inicial < UMBRAL_QS) {
    insercion_lims(T, inicial, final);
  } else {
    dividir_qs(T, inicial, final, k);
    quicksort_lims(T, inicial, k);
    quicksort_lims(T, k + 1, final);
  };
}
```

Eficiencia empírica

Ejecutamos el programa obteniendo como resultado la siguiente tabla de tamaños y tiempos.

```
100 7.8125e-06
200 3.75e-05
300 8.4375e-05
400 0.000148437
500 0.000229687
600 0.000329688
700 0.000451562
800 0.000582812
900 0.000742188
1000 0.000910937
1100 0.00110312
1200 0.0013
1300 0.00152031
1400 0.00175781
1500 0.00201719
1600 0.00229844
1700 0.00258594
1800 0.00289844
1900 0.00324375
```

Usamos gnuplot para generar la siguiente gráfica:



Eficiencia híbrida

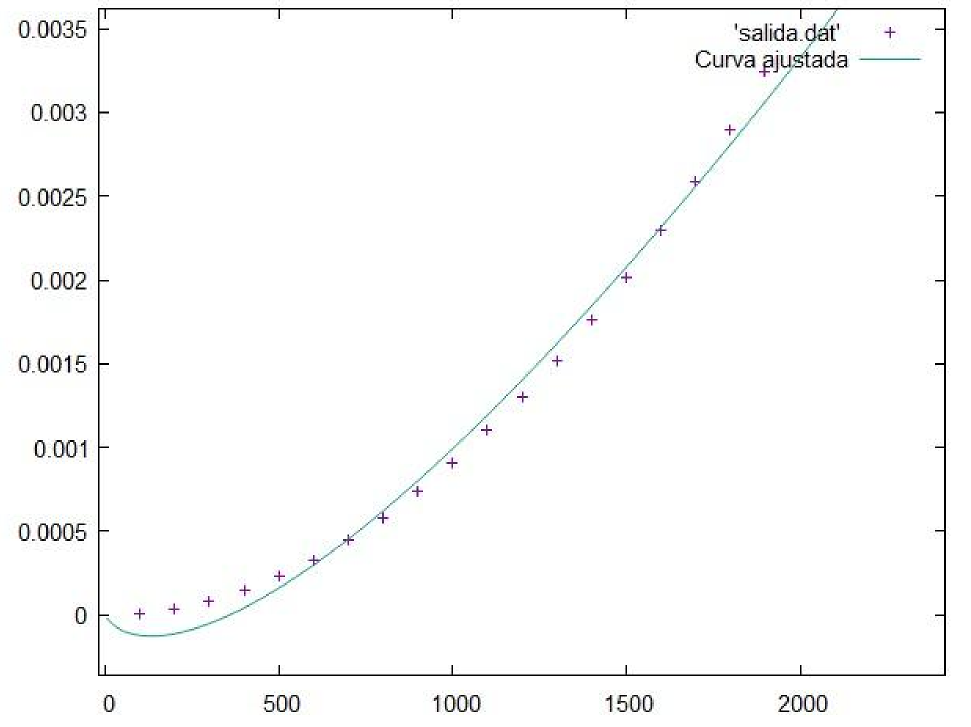
Ahora vamos a ajustar la curva obtenida usando la siguiente función logarítmica que es la que más se asemeja a la curva:

*f(n) = a0 * n * log(a1*n)*

donde n es el tamaño de la muestra y a0, a1 son las constantes ocultas.
Al ajustar la curva nos sale el siguiente resultado:

```
Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a0              = 9.68674e-07      +/- 5.996e-08    (6.19%)
a1              = 0.00278041       +/- 0.0002405    (8.651%)
```

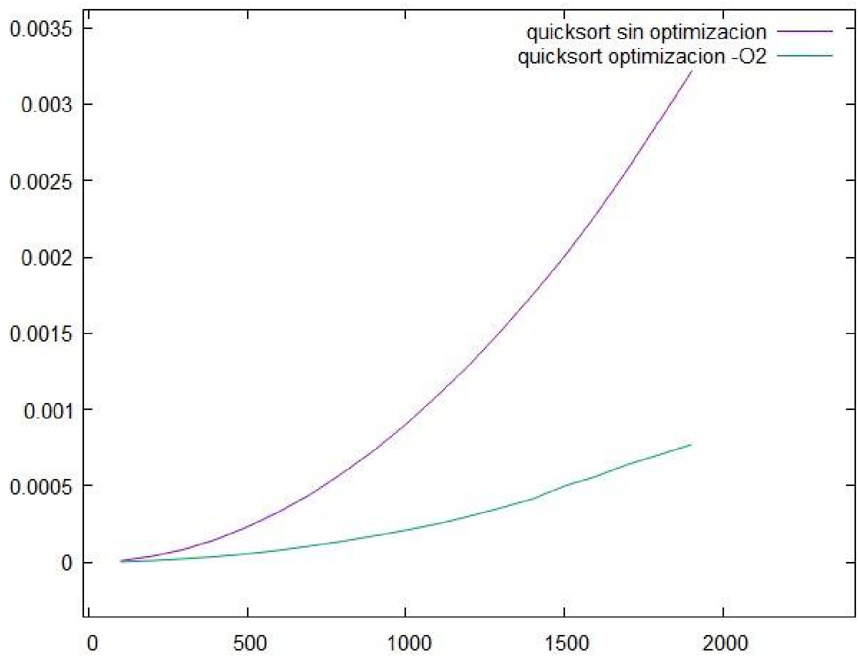
Por tanto la función que nos queda para una entrada n es
*f(n) = 9.68674e-07 * n * log(0.00278041 * n)*



Efecto de factores externos

Ahora compilamos el código con la opción **-O2** para optimizarlo de la siguiente manera:
g++ -O2 quicksort.cpp

La siguiente gráfica muestra las diferencias entre ejecutarlo con y sin la opción de optimización.



Como se puede ver en la gráfica, hay una gran diferencia en los tiempos de ejecución.

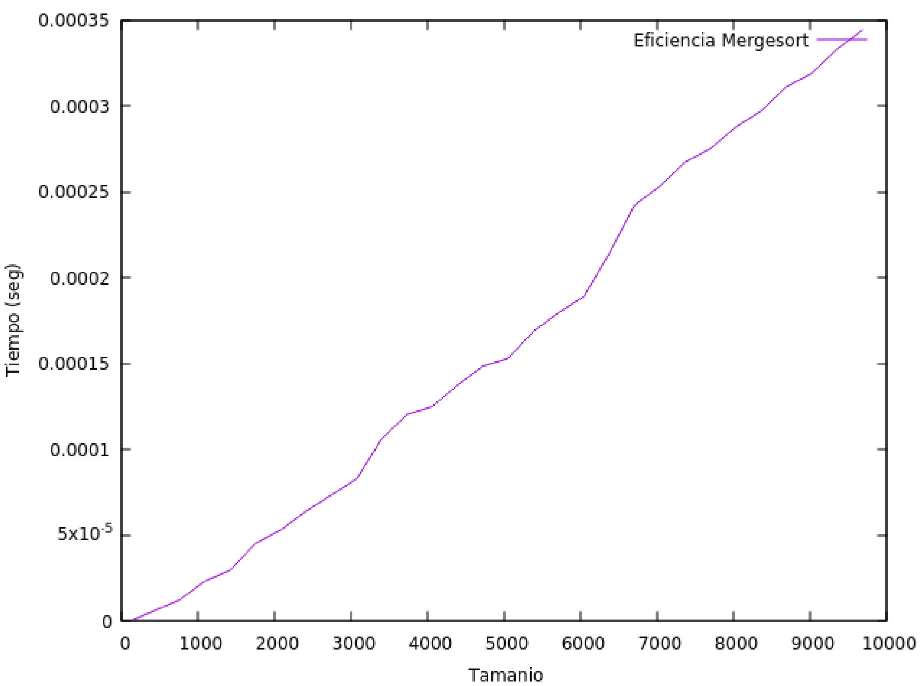
Eficiencia Algoritmo Mergesort

```
static void mergesort_lims(int T[], int inicial, int final)
{
  if (final - inicial < UMBRAL_MS)
    {
      insercion_lims(T, inicial, final);
    } else {
      int k = (final - inicial)/2;

      int * U = new int [k - inicial + 1];
      assert(U);
      int l, l2;
      for (l = 0, l2 = inicial; l < k; l++, l2++)
  U[l] = T[l2];
      U[l] = INT_MAX;

      int * V = new int [final - k + 1];
      assert(V);
      for (l = 0, l2 = k; l < final - k; l++, l2++)
  V[l] = T[l2];
      V[l] = INT_MAX;

      mergesort_lims(U, 0, k);
      mergesort_lims(V, 0, final - k);
      fusion(T, inicial, final, U, V);
      delete [] U;
      delete [] V;
    };
}
```

Eficiencia empírica

Ejecutando el programa con este algoritmo obtenemos la siguiente tabla y usando gnuplot generamos una gráfica

100	0
430	6.25e-06
760	1.25e-05
1090	2.34375e-05
1420	2.96875e-05
1750	4.53125e-05
2080	5.3125e-05
2410	6.40625e-05
2740	7.34375e-05
3070	8.28125e-05
3400	0.00010625
3730	0.000120313
4060	0.000125
4390	0.0001375
4720	0.000148437
5050	0.000153125
5380	0.00016875
5710	0.000179687
6040	0.000189062
6370	0.000214063
6700	0.000242187
7030	0.000253125
7360	0.000267188
7690	0.000275
8020	0.0002875
8350	0.000296875
8680	0.000310938
9010	0.00031875
9340	0.000332812
9670	0.00034375



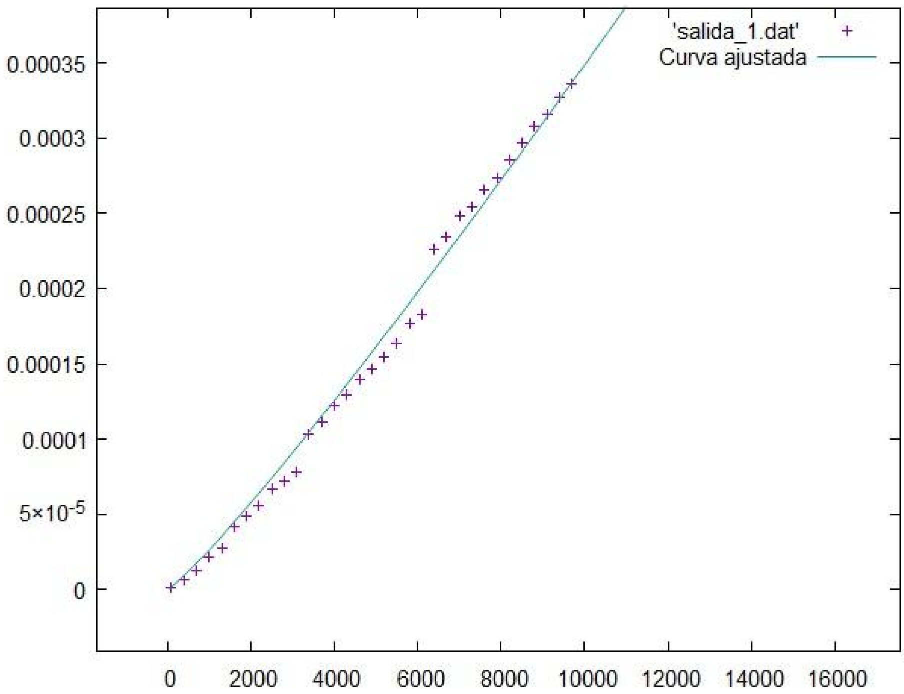
Eficiencia híbrida

Ahora vamos a ajustar la curva obtenida usando la siguiente función logarítmica que es la que más se asemeja a la curva:

*f(n) = a0 * n * log(n)*
donde n es el tamaño de la muestra y a0 es la constante oculta.
Al ajustar la curva nos sale el siguiente resultado:

```
Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a0              = 3.78301e-09      +/- 3.155e-11    (0.8339%)
```

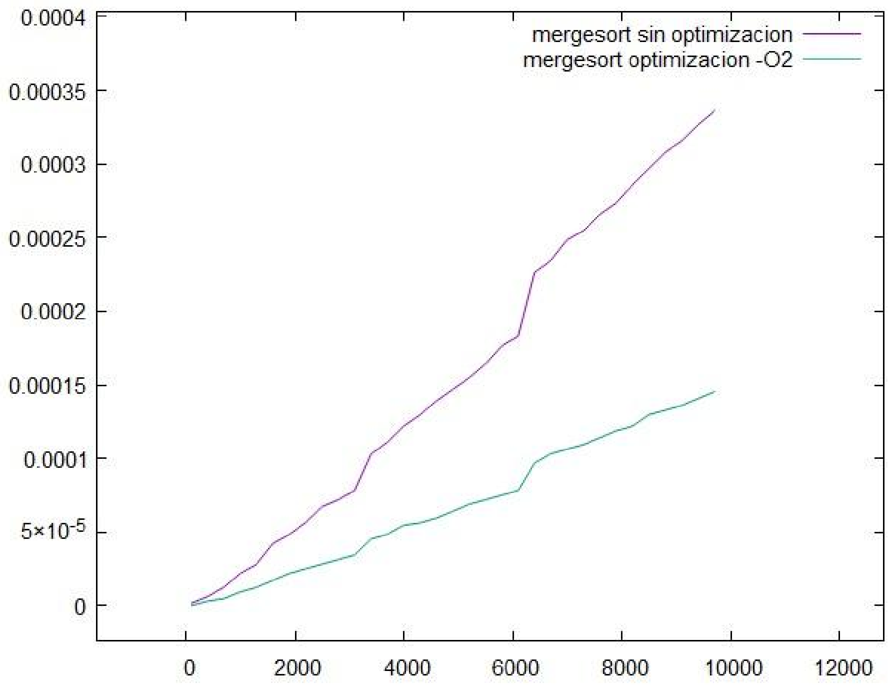
Por tanto la función que nos queda para una entrada n es
*f(n) = 3.78301e-09 * n * log(n)*



Efecto de factores externos

Ahora compilamos el código con la opción **-O2** para optimizarlo de la siguiente manera:
g++ -O2 mergesort.cpp

La siguiente gráfica muestra las diferencias entre ejecutarlo con y sin la opción de optimización.



Como se puede ver en la gráfica, hay una gran diferencia en los tiempos de ejecución.

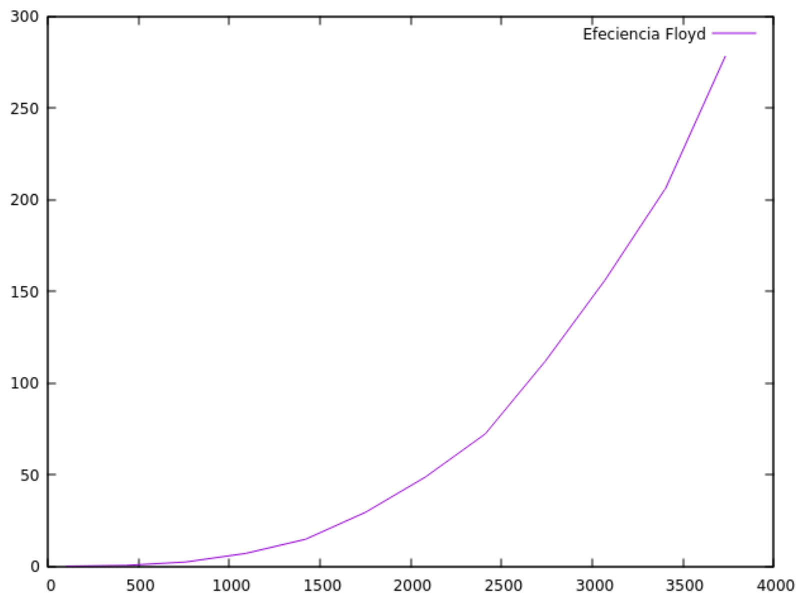
Eficiencia Algoritmo Floyd

```
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];
        }
}
```

Eficiencia empírica

Después de ejecutar el algoritmo variando el tamaño desde 100 hasta 5000, en iteraciones de 330, los resultados obtenidos son:

```
100 0.005373
430 0.418749
760 2.24407
1090 6.95294
1420 14.6757
1750 29.3725
2080 48.6111
2410 72.1713
2740 111.744
3070 156.092
3400 205.609
3730 277.991
```



La primera fila son las iteraciones y la segunda el tiempo en segundos.

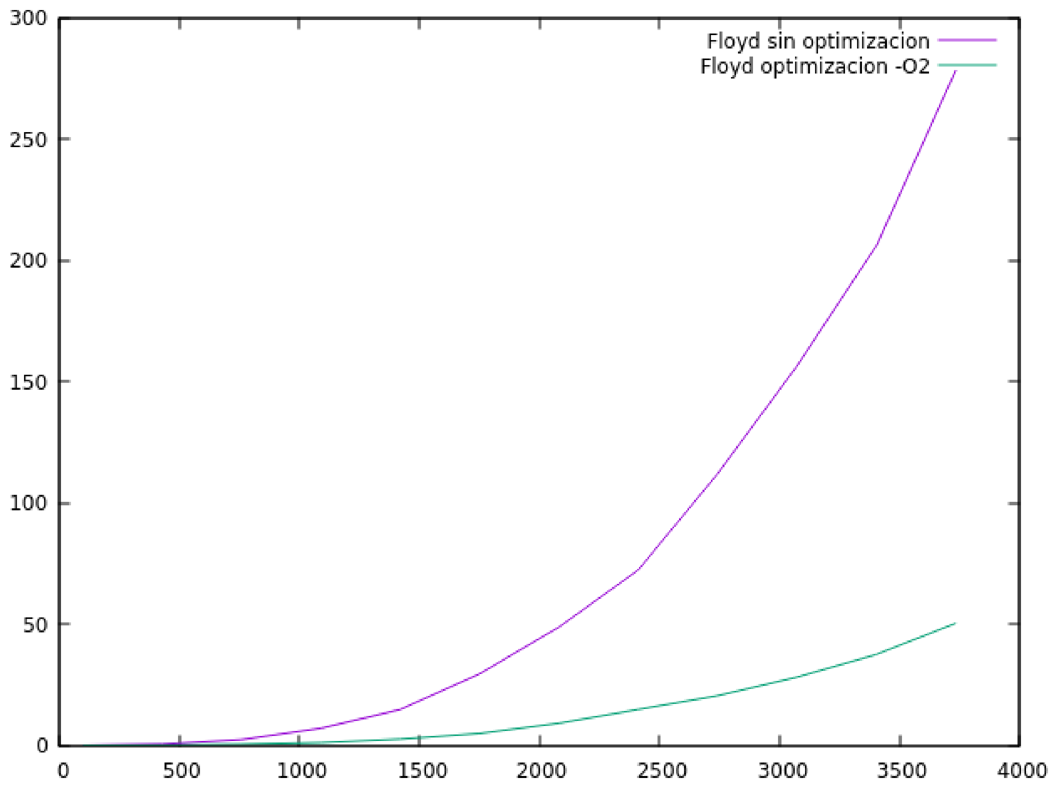
En dicha gráfica se puede observar un orden cúbico debido a su curva aunque dicha curva no esté muy definida

Estudios con factores externos.

Los datos presentados en el anterior punto se han obtenido sin optimización, es decir, compilando con *g++ -o floyd floyd.cpp*

Sin embargo en este apartado llevaremos a cabo el estudio de la gráfica resultante de la diferencia respecto a la gráfica anterior con el comando:
g++ -O2 -o floyd floyd.cpp

Y esta sería la gráfica resultante:



En esta gráfica podemos observar como el algoritmo de floyd optimizado es muchísimo más rápido que el que no lo está, incluso me atrevería a decir que es 6 veces más rápido

Eficiencia híbrida

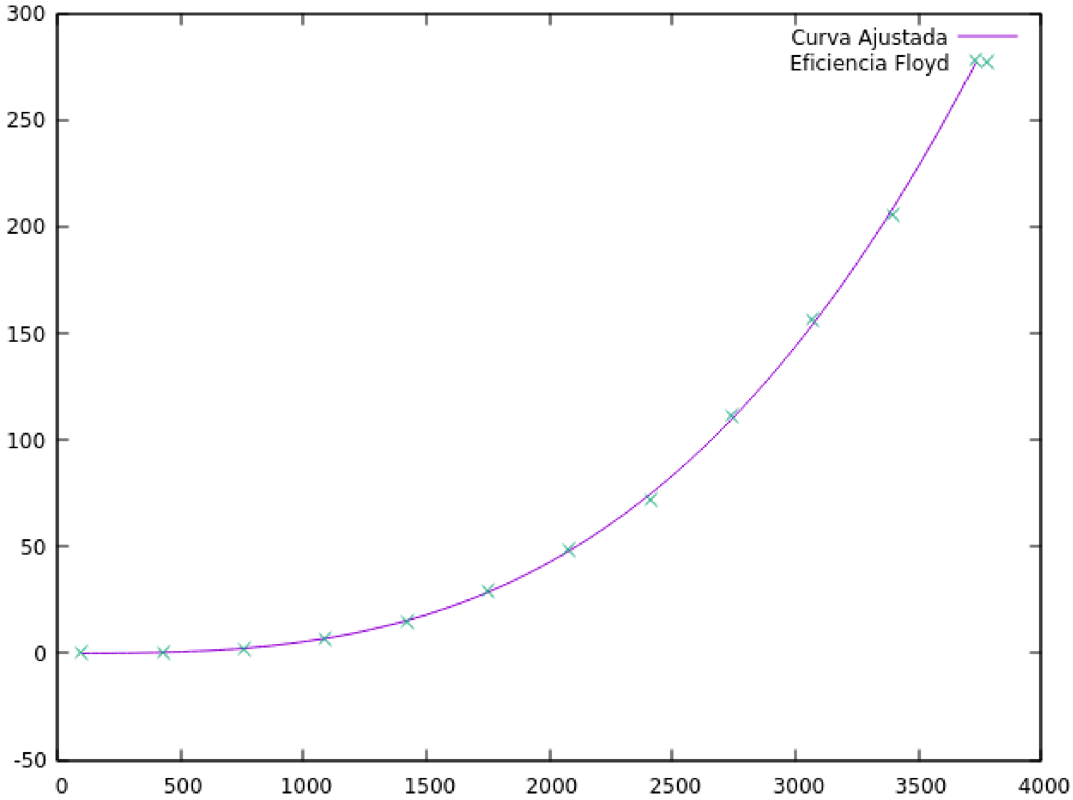
Después de todo esto ajustaremos la curva a través de la función:

F(n) = $a_0 \times n^3 + a_1 \times n^2 + a_2 \times n + a_3$

```
Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a0              = 5.33426e-09      +/- 5.19e-10     (9.73%)
a1              = -6.75333e-08     +/- 3.024e-06    (4478%)
a2              = 0.000188411      +/- 0.004932     (2618%)
a3              = -0.09843         +/- 2.127        (2161%)
```

por lo que la función seria, F(n) =

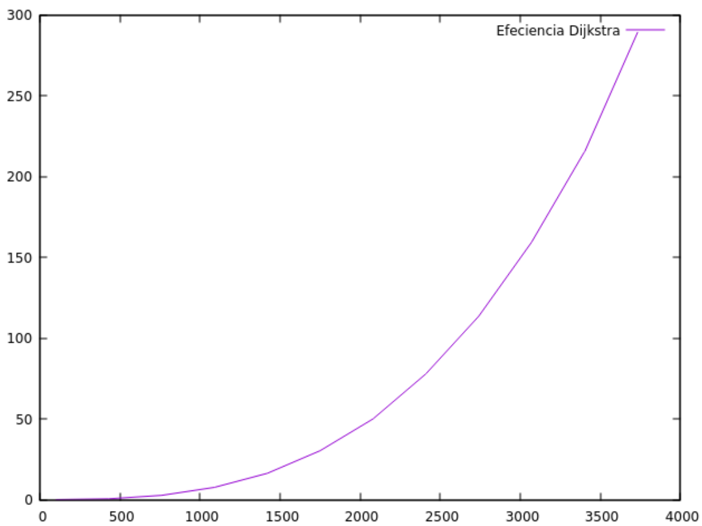
$5.33426e - 09 \times n^3 - 6.75333e - 08 \times n^2 + 1.88411e - 04 \times n - 0.09843$



Eficiencia Algoritmo Dijkstra

```
void Dijkstra(int **M, int **Sal, int dim, int src) // adjacency ma
{
  int dist[dim]; // integer array to calculate minimum distance fo
  bool Tset[dim];// boolean array to mark visted/unvisted for each

  // set the nodes with infinity distance
  // except for the initial node and mark
  // them unvisited.
  for(int i = 0; i<dim; i++)
  {
    dist[i] = INT_MAX;
    Tset[i] = false;
  }

  dist[src] = 0;   // Source vertex distance is set to zero.

  for(int k = 0; k<dim; k++)
  {
    int m=minimumDist(dist,Tset,dim); // vertex not yet included.
    Tset[m]=true;// m with minimum distance included in Tset.
    for(int i = 0; i<dim; i++)
    {
      // Updating the minimum distance for the particular node.
      //if(!Tset[i] && graph[m][i] && dist[m]!=INT_MAX && dist[m]+g
      if(!Tset[i] && dist[m]!=INT_MAX && dist[m]+M[m][i]<dist[i])
        dist[i]=dist[m]+M[m][i];
    }
  }
  for(int i = 0; i<dim; i++)
    Sal[src][i]=dist[i];
}
```

```
100 0.008208 s
430 0.515975 s
760 2.62699 s
1090 7.57769 s
1420 16.2595 s
1750 30.2673 s
2080 49.9125 s
2410 77.8444 s
2740 113.48 s
3070 159.26 s
3400 215.202 s
3730 289.028 s
```



Eficiencia empírica

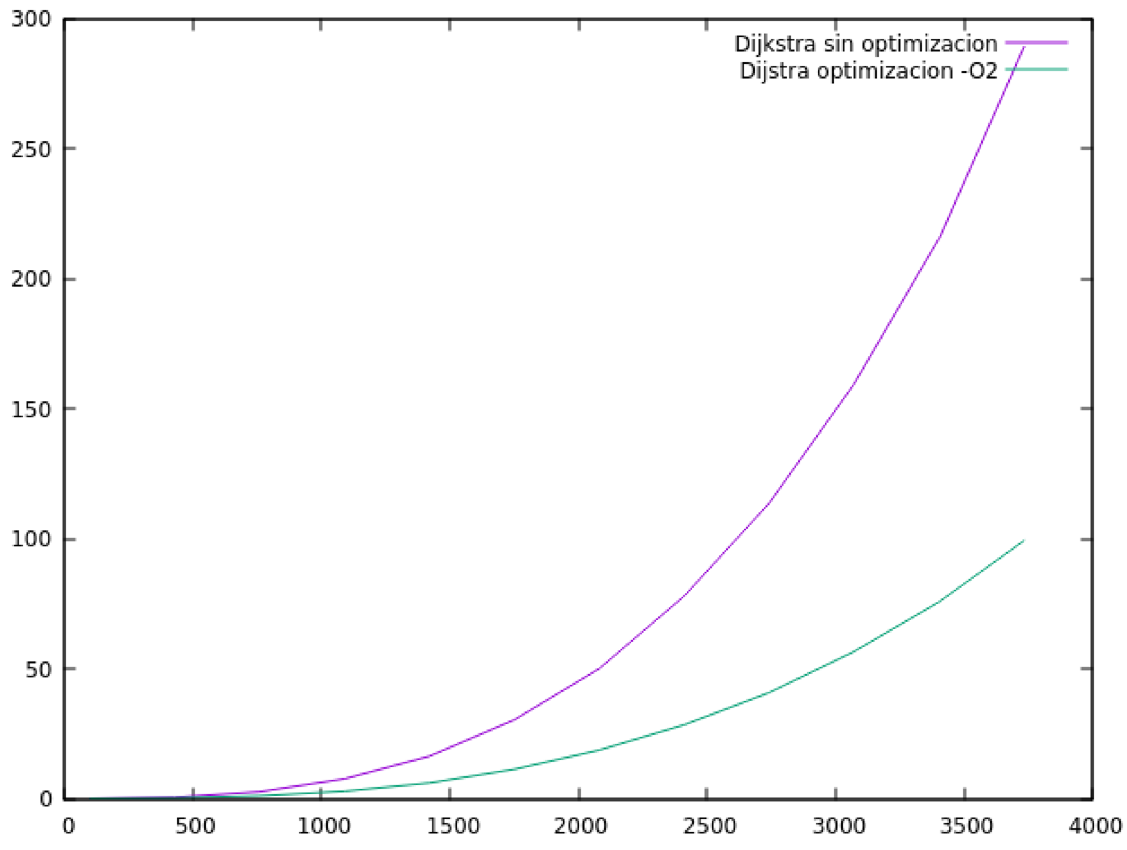
Como podemos observar tenemos que en las primeras iteraciones este algoritmo (Dijkstra) es mejor que el anterior (Floyd), aunque conforme las iteraciones van aumentando el tiempo se “iguala” en los dos algoritmos, dicha curva se puede denotar como cúbica y está un poco mejor definida que la gráfica del algoritmo anterior

Estudios con factores externos.

Los datos presentados en el anterior punto se han obtenido sin optimización, es decir, compilando con *g++ -o dijkstra dijkstra-iterado.cpp*

Sin embargo en este apartado llevaremos a cabo el estudio de la gráfica resultante de la diferencia respecto a la gráfica anterior con el comando:
g++ -O2 -o dijkstra dijkstra-iterado.cpp

Y esta sería la gráfica resultante:



Se puede observar que la gráfica sin optimización es mucho más lenta que la optimizada.

Eficiencia híbrida

Después de todo esto ajustaremos la curva a través de la función:

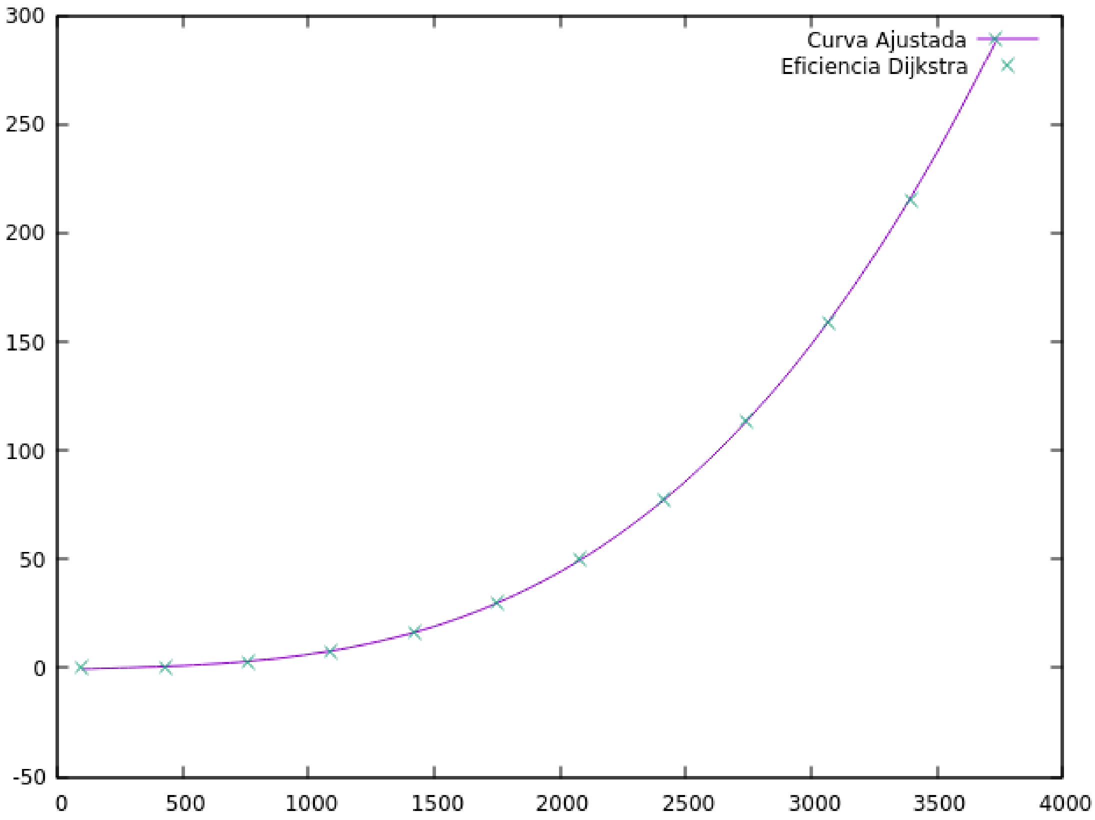
F(n) = $a_0 \times n^3 + a_1 \times n^2 + a_2 \times n + a_3$

```
Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a0              = 5.94264e-09      +/- 2.532e-10    (4.261%)
a1              = -2.3574e-06      +/- 1.475e-06    (62.59%)
a2              = 0.00344878       +/- 0.002406     (69.77%)
a3              = -0.773316        +/- 1.038        (134.2%)
```

por lo que la función seria, F(n) =

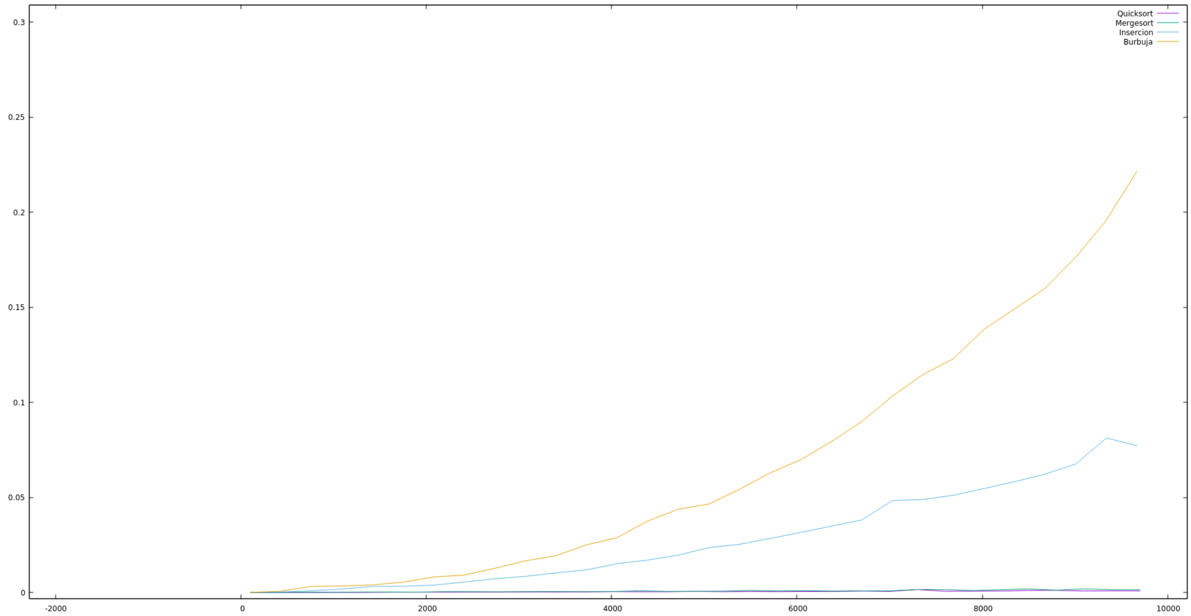
$5.94264e-09 \times n^3 - 2.3574e-06 \times n^2 + 3.44878e-03 \times n - 0.773316$

Luego la curva:



Comparación de los algoritmos de ordenación

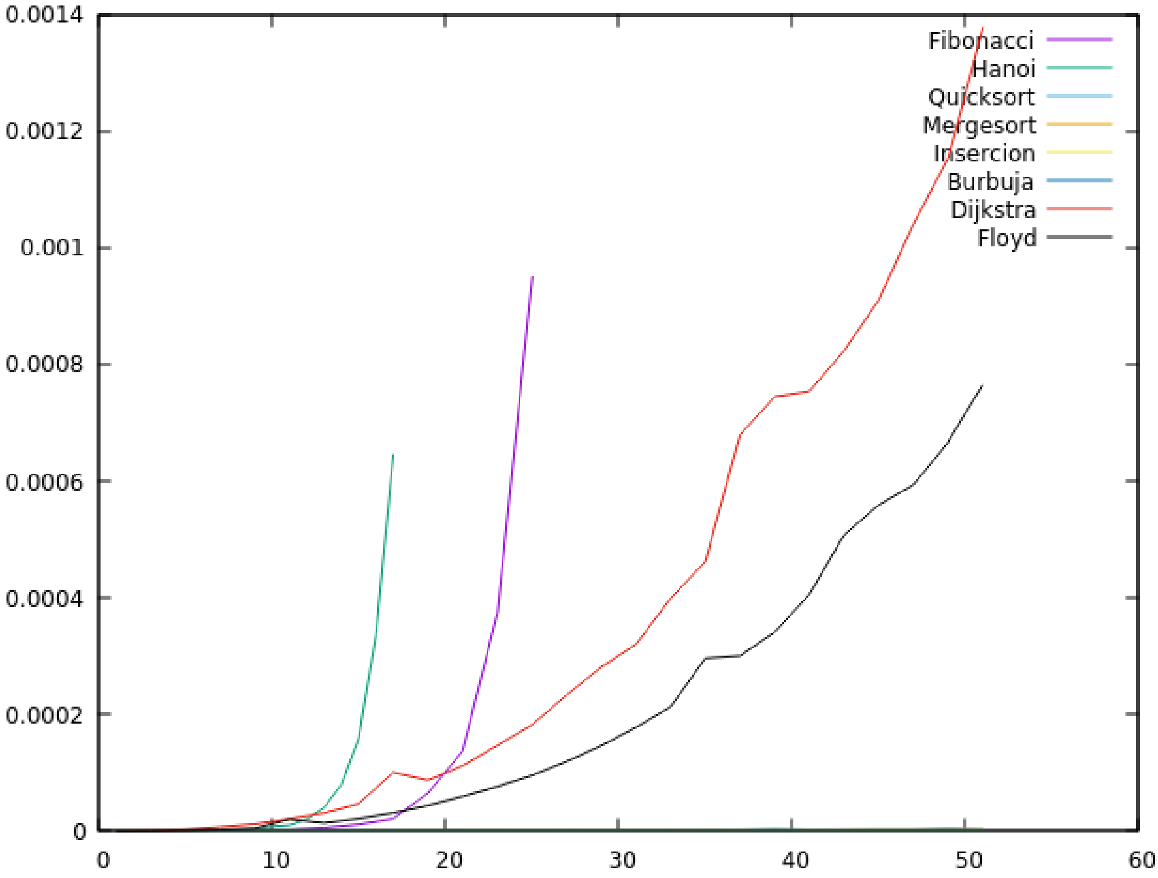
Los algoritmos de ordenación que tratamos en esta práctica son burbuja, inserción, mergesort y quicksort, los dos primeros son del orden 0(n^2) y los dos últimos del orden O($n \times log(n)$).



Como se observa, los de burbuja e inserción se disparan mientras los otros dos se quedan rozando el 0, viendo claramente cuales son los más eficientes.

Comparativa general de los algoritmos

Debido a las distintas duraciones de los algoritmos para los distintos tamaños, es complicado realizar una gráfica en la que se aprecien todos los algoritmos perfectamente. De lo mejor que hemos podido conseguir es lo siguiente:



Aquí se aprecia el salto exponencial de Fibonacci y Hanoi, seguidos de Dijkstra y Floyd y los de ordenación, que quedan pegados al 0 absoluto, debido a la corta duración de los mismos.