UNIVERSIDAD
DE GRANADA

GRADO EN INGENIERÍA INFORMÁTICA

Práctica 1: Análisis de Eficiencia de Algoritmos

Autores:
Yunkai Lin Pan: 20 %
Alfonso Jesús Piñera Herrera: 20 %
Álvaro Hernández Coronel: 20 %
Jaime Castillo Uclés: 20 %
Yeray López Ramírez: 20 %

Curso:2ºC
Asignatura: Algorítmica
Fecha: 22 de Marzo de 2022
Grupo de prácticas: C2
Número de grupo: 3

Índice

1. Descripción del problema

En esta práctica analizaremos de forma **empírica** e **híbrida** distintos algorítmos.

2. Algoritmos de Ordenación

En esta parte trataremos cuatro de los algoritmos de ordenación más conocidos. Los dividiremos en *cuadráticos* y *logarítmicos*:

2.1. Eficiencia n^2

Los algoritmos a calcular su eficiencia empírica que tienen este tipo de eficiencia son el algoritmo de la **burbuja** y de la **inserción**. Empezamos con el algoritmo de la *burbuja*.

2.1.1. Ordenación por Burbuja

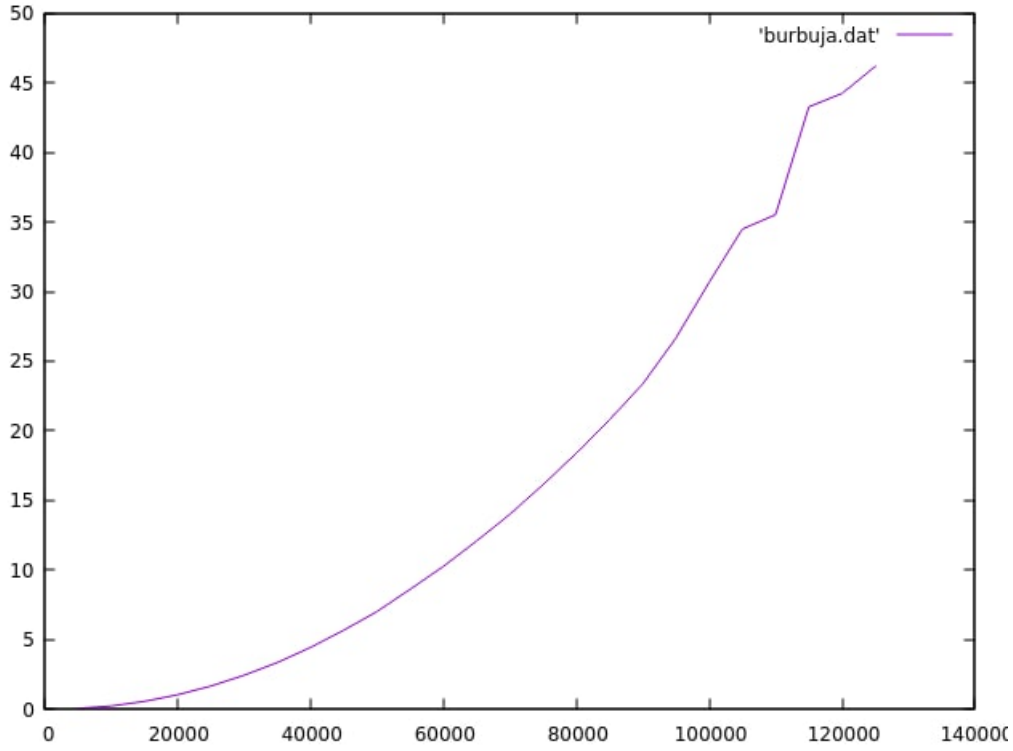
La función proporcionada para este algoritmo es la siguiente:

```
 inline void burbuja(int T[], int num_elem)
{
 burbuja_lims(T, 0, num_elem);
}

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;
 }
}
```

Al ejecutar `./burbuja` con los valores predeterminados del compilador:

Tamaño	Tiempo(seg)
5000	0.0858651
10000	0.247879
15000	0.585624
20000	1.05817
25000	1.67296
30000	2.45572
35000	3.37203
40000	4.44497
45000	5.69015
50000	7.0206
55000	8.62408
60000	10.2721
65000	12.0943
70000	14.0043
75000	16.1324
80000	18.3781
85000	20.7872
90000	23.3628
95000	26.6501
100000	30.6623
105000	34.4759
110000	35.5128
115000	43.256
120000	44.2151
125000	46.153

Al usar gnuplot para graficar los datos anteriores, se crea la siguiente gráfica:



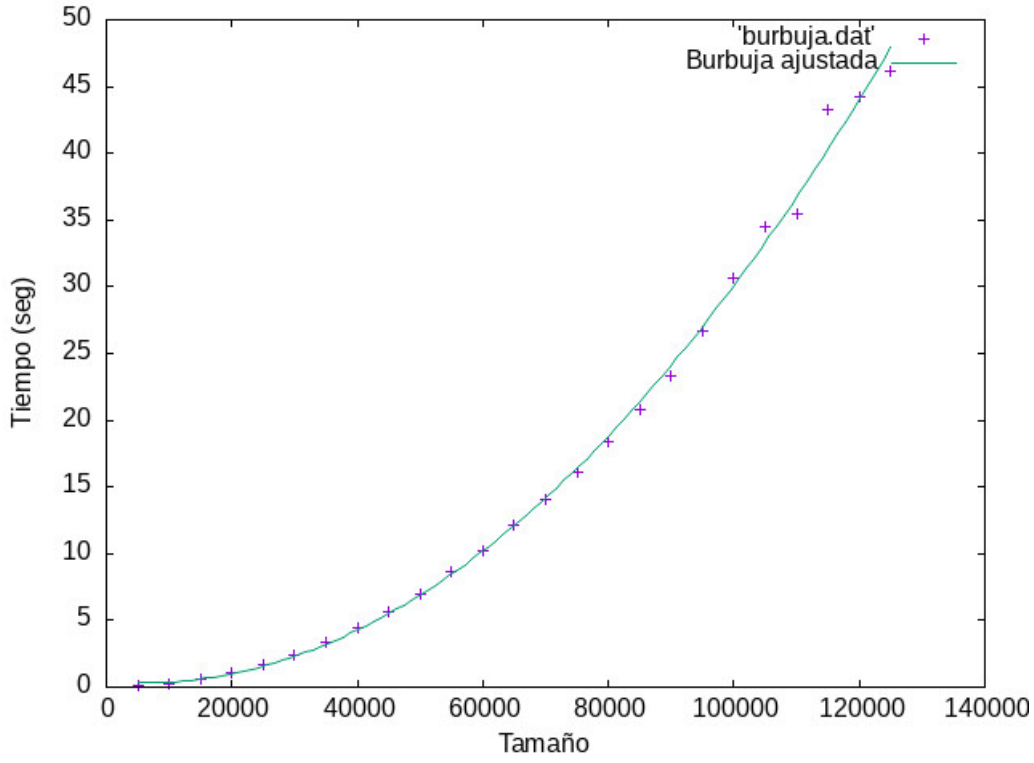
La eficiencia híbrida calculada mediante gnuplot da como resultado el siguiente fichero `fit.log`:

```
FIT:    data read from 'burbuja.dat'
       #datapoints = 25
function used for fitting: f(x)
       f(x) = a0*x*x+a1*x+a2
iter      chisq       delta/lim  lambda   a0             a1             a2
  0 1.3460545235e+21   0.00e+00  4.24e+09    1.000000e+00   1.000000e+00
 13 1.6690141492e+01  -1.14e-05  4.24e-04    3.336682e-09  -3.675681e-05
After 13 iterations the fit converged.
Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a0              = 3.33668e-09      +/- 1.502e-10    (4.501%)
a1              = -3.67568e-05     +/- 2.011e-05    (54.72%)
a2              = 0.402602         +/- 0.5674       (140.9%)
correlation matrix of the fit parameters:
                a0     a1     a2
a0              1.000
a1             -0.971  1.000
a2              0.774 -0.884  1.000
```

De aquí concluimos que la formula ajustada es:

$$f(x) = 3{,}336\,68 \times 10^{-9}x^2 - 3{,}675\,68 \times 10^{-5}x + 0{,}402602$$

Tras representar la función ajustada anterior en la gráfica de puntos podemos ver que se ajustan perfectamente:



2.1.2. Ordenación por Inserción

La función del algoritmo de inserción analizado es la siguiente:

```
static void insercion_lims(int T[], int inicial, int final){
  int i, j, 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--;
        }
  }
}
```

La tabla de datos resultante en el PC de nuestro compañero Álvaro es:

Tamaño	**Tiempo(seg)**
5000	0.0398803
10000	0.107249
15000	0.245123
20000	0.419207
25000	0.630739
30000	0.911298
35000	1.27238
40000	1.60618
45000	2.05534
50000	2.51724
55000	3.06278
60000	3.63798
65000	4.22758
70000	9.06372
75000	9.3657
80000	10.3132
85000	7.27995
90000	8.11229
95000	9.09415
100000	10.0099
105000	11.0659
110000	14.0415
115000	13.2668
120000	14.474
125000	15.6717

Como podemos entrever, hay algunos picos inesperados para ciertos tamaños.

Al graficar la tabla de datos, efectivamente vemos los picos más claramente:

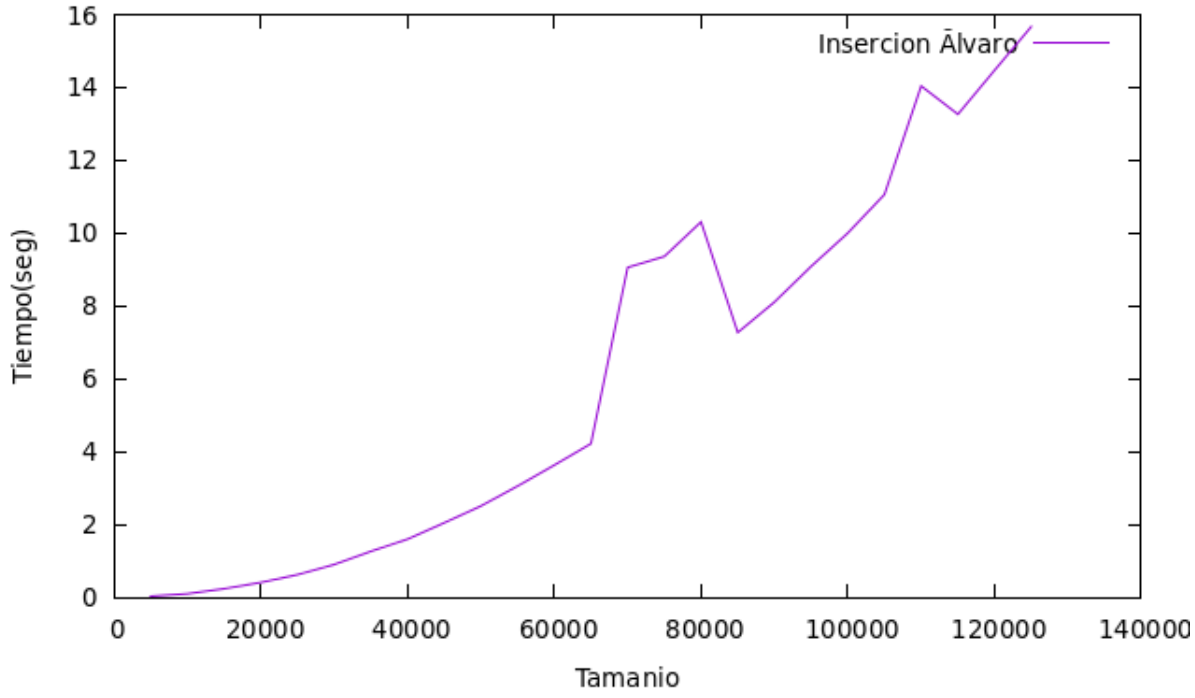
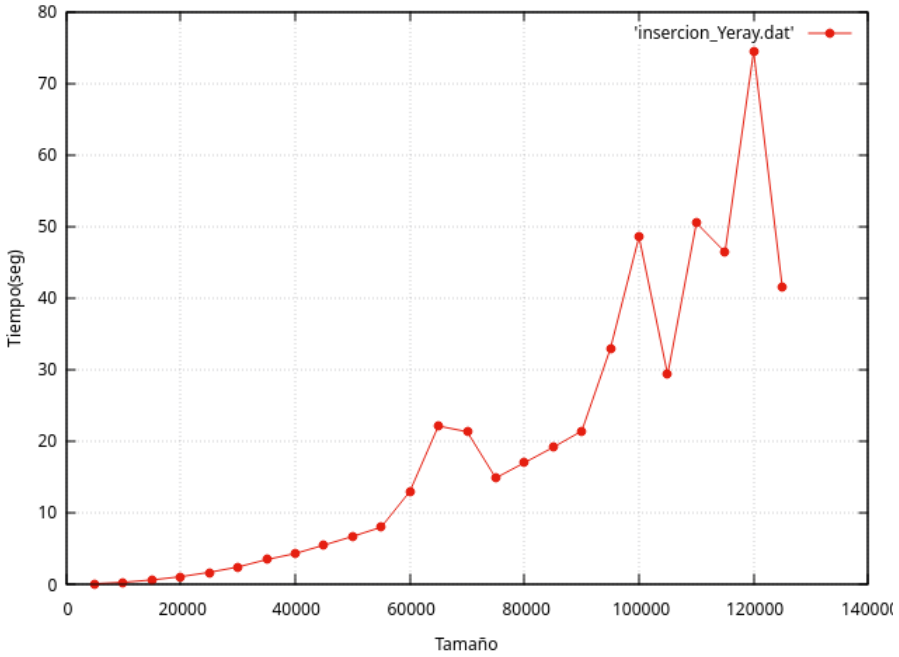


Figura 1: Algoritmo de Inserción en el PC de Álvaro

Desconocemos el porqué de este resultado errático, también sucede en el ordenador personal del compañero Yeray:



Al intentar ajustar los datos de antes vemos que el porcentaje de error es inusualmente alto:

```
 ******************************************************************************
Mon Mar 21 16:02:10 2022


FIT:        data read from "insercion.dat"
            format = z
            #datapoints = 25
            residuals are weighted equally (unit weight)

function used for fitting: f(x)
    f(x) = a0*x*x+a1*x+a2
fitted parameters initialized with current variable values

iter          chisq   delta/lim  lambda   a0                a1
   0 1.3460545288e+21   0.00e+00  4.24e+09        1.000000e+00
  13 3.6180941721e+01  -5.50e-05  4.24e-04        6.294092e-10

After 13 iterations the fit converged.
final sum of squares of residuals : 36.1809
rel. change during last iteration : -5.50109e-10

degrees of freedom        (FIT_NDF)                        : 22
rms of residuals          (FIT_STDFIT) = sqrt(WSSR/ndf)        : 1.28242
variance of residuals (reduced chisquare) = WSSR/ndf    : 1.64459

Final set of parameters               Asymptotic Standard Error
=======================               ==========================
a0                  = 6.29409e-10         +/- 2.211e-10        (35.13%)
a1                  = 5.46974e-05         +/- 2.961e-05        (54.14%)
a2                  = -0.933148           +/- 0.8354           (89.52%)

correlation matrix of the fit parameters:
                a0     a1     a2
a0              1.000
a1             -0.971  1.000
a2              0.774 -0.884  1.000
```

La función ajustada que nos genera es:

$$f(x) = 6{,}294\,09 \times 10^{-10}x^2 + 5{,}469\,74 \times 10^{-5}x - 0{,}933148$$

Al ajustarla vemos que la función pasa por el medio de los puntos:

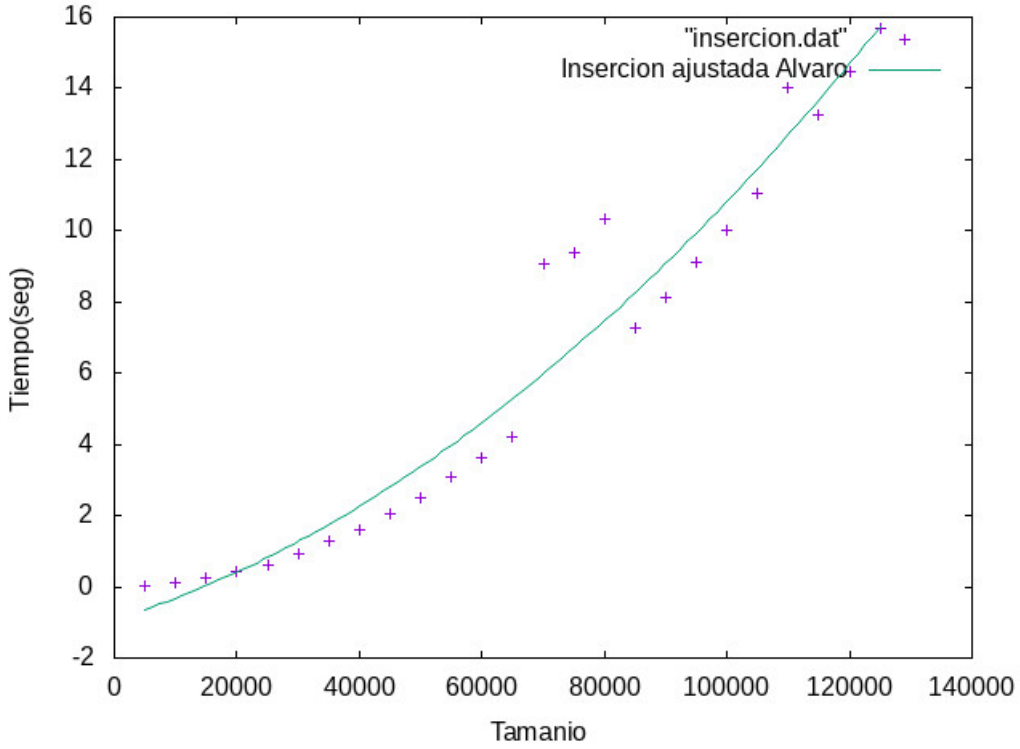


Figura 2: Ajuste en el PC de Álvaro

Sin embargo, tras ejecutar exactamente el mismo fichero fuente del algoritmo en el ordenador de Jaime obtenemos una tabla sin sorpresas:

Tamaño	**Tiempo(seg)**
5000	0.0286526
10000	0.107456
15000	0.208767
20000	0.351672
25000	0.552823
30000	0.772061
35000	1.08757
40000	1.41086
45000	1.62846
50000	1.95397
55000	2.31595

60000	2.75238
65000	3.25217
70000	3.82004
75000	4.35694
80000	4.95242
85000	5.61813
90000	6.40346
95000	7.37948
100000	7.86567
105000	8.78106
110000	9.40774
115000	10.3115
120000	12.5462
125000	12.1462

Efectivamente al graficarla ésta se ajusta a una función cuadrática convencional a excepción de la última ejecución que sufre una bajada de medio segundo respecto a la anterior ejecución:

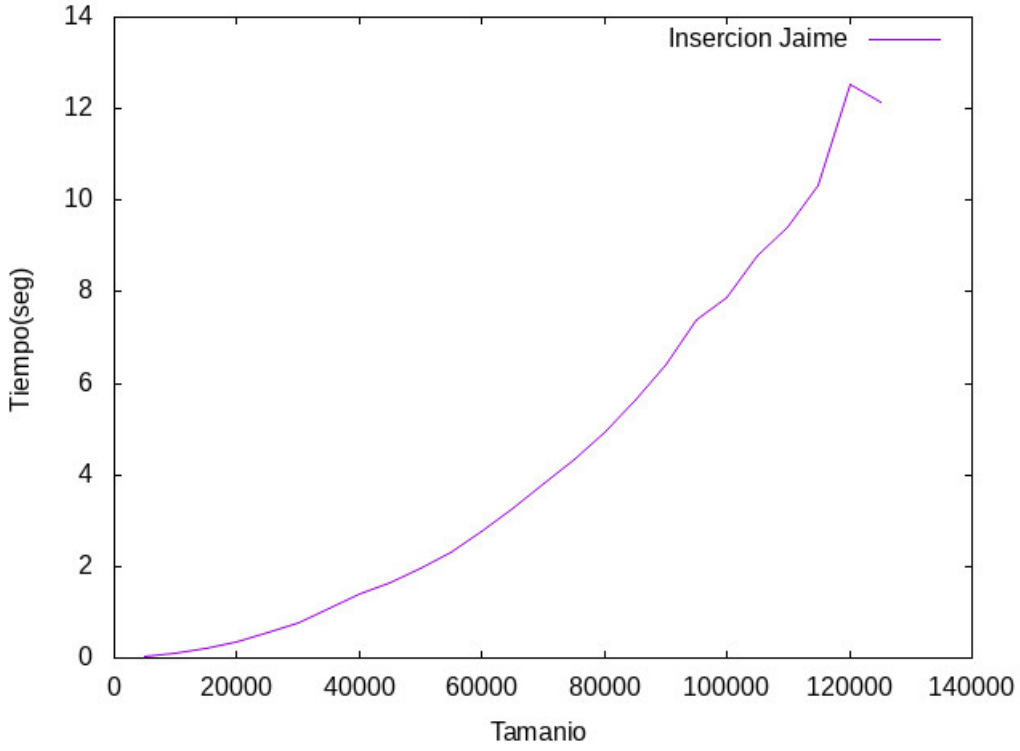


Figura 3: Algoritmo de Inserción en el PC de Jaime

El ajuste queda finalmente:

```
*******************************************************************************
Mon Mar 21 16:14:53 2022


FIT:        data read from "insercion2.dat"
            format = z
            #datapoints = 25
            residuals are weighted equally (unit weight)

function used for fitting: f(x)
    f(x) = a0*x*x+a1*x+a2
fitted parameters initialized with current variable values

iter          chisq          delta/lim  lambda    a0
   0 1.1054819544e+02   0.00e+00  3.59e+00        6.294092e-10
   4 1.4550871377e+00  -1.09e-03  3.59e-04        8.460509e-10

After 4 iterations the fit converged.
final sum of squares of residuals : 1.45509
rel. change during last iteration : -1.08883e-08

degrees of freedom        (FIT_NDF)                              : 22
rms of residuals          (FIT_STDFIT) = sqrt(WSSR/ndf)          : 0.257178
variance of residuals (reduced chisquare) = WSSR/ndf    : 0.0661403

Final set of parameters                Asymptotic Standard Error
=======================                ==========================
a0                  = 8.46051e-10          +/- 4.434e-11        (5.241%)
a1                  = -6.43769e-06         +/- 5.938e-06        (92.24%)
a2                  = 0.144484             +/- 0.1675           (116%)

correlation matrix of the fit parameters:
                    a0   a1         a2
a0                  1.000
a1                 -0.971  1.000
a2                  0.774 -0.884  1.000
```

La función resultante del ajuste es similar a la anterior:

$$f(x) = 8{,}460\,51 \times 10^{-10}x^2 - 6{,}437\,69 \times 10^{-6}x + 0{,}144484$$

Sin embargo, esta vez si pasa por los puntos de forma correcta:

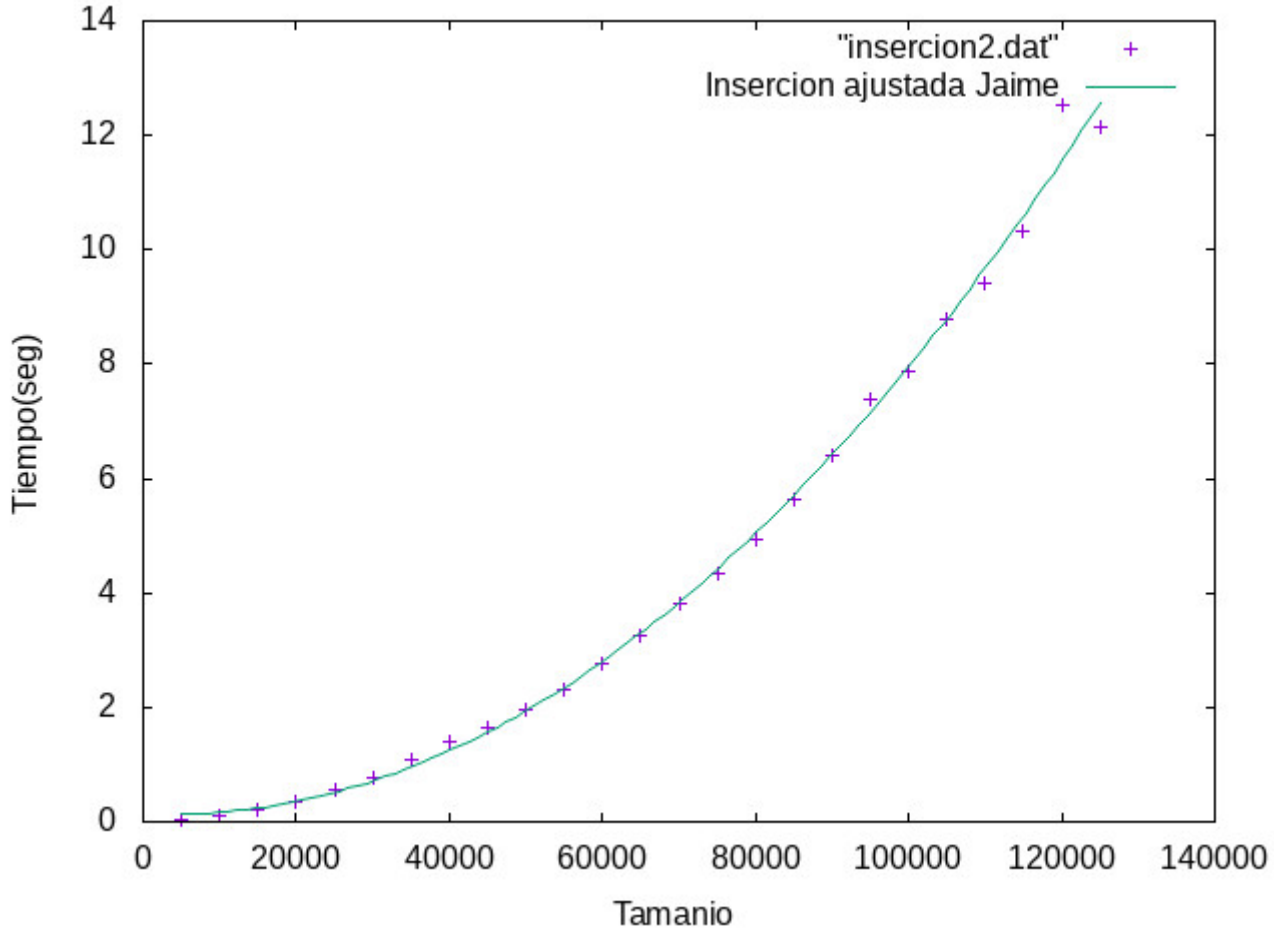


Figura 4: Ajuste del algoritmo de Inserción en el PC de Jaime

2.2. Eficiencia n log(n)

Los algoritmos de orden $n\ log(n)$ son el algoritmo de **mergesort** y el **quicksort** que son algoritmos de ordenación de vectores. Los explicamos a continuación:

2.2.1. Ordenación por Mergesort

Es uno de los algoritmos más eficientes de la práctica junto al Quicksort. Como veremos después, el set de datos no pasa del segundo.

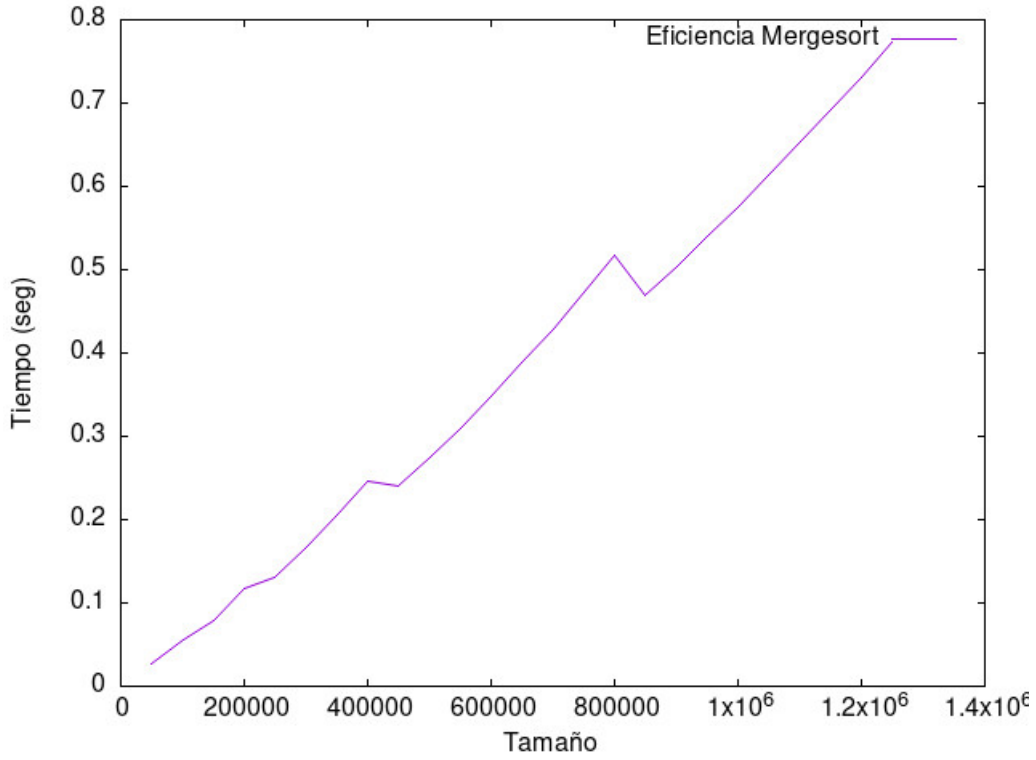
El código utlizado para el algoritmo Mergesort es:

```
const int UMBRAL_MS = 100;

void mergesort(int T[], int num_elem)
{
  mergesort_lims(T, 0, num_elem);
}

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;
        };
}
```

La tabla de datos obtenida es:

Tamaño	Tiempo(seg)
50000	0.00877253
100000	0.0184002
150000	0.0257687
200000	0.0387505
250000	0.0427258
300000	0.0545412
350000	0.0674265
400000	0.0810115
450000	0.0785888
500000	0.0901341
550000	0.102542
600000	0.114784
650000	0.127968
700000	0.141309
750000	0.155169
800000	0.169367
850000	0.154049
900000	0.165465
950000	0.177047
1000000	0.189419
1050000	0.202066
1100000	0.214444
1150000	0.227425
1200000	0.240154
1250000	0.253565

La gráfica resultante de la tabla calculada antes es:



Al calcular la eficiencia híbrida de Mergesort en gnuplot, obtenemos:

```
Ajustada a la fórmula a0*x*log(x) + a1
iter          chisq           delta/lim  lambda   a0
   0 2.6033979525e+15   0.00e+00  1.02e+07        1.000000e+00
   * 9.3808481861e-04   5.78e-11  1.02e+04        1.418029e-08
iter          chisq           delta/lim  lambda   a0

After 5 iterations the fit converged.
final sum of squares of residuals : 0.000938085
rel. change during last iteration : -5.77881e-16

degrees of freedom        (FIT_NDF)                       : 24
rms of residuals          (FIT_STDFIT) = sqrt(WSSR/ndf)   : 0.00625195
variance of residuals (reduced chisquare) = WSSR/ndf    : 3.90869e-05

Final set of parameters                Asymptotic Standard Error
=======================                ==========================
a0                  = 1.41803e-08          +/- 1.225e-10        (0.8641%)
a1                = 0.001932       +/- 0.0006271    (32.68%)
```

La función ajustada al algoritmo Mergesort para esta ejecución es:

$$1{,}418\,03 \times 10^{-8} n \log(n) + 0{,}001932$$

La gráfica del ajuste queda:

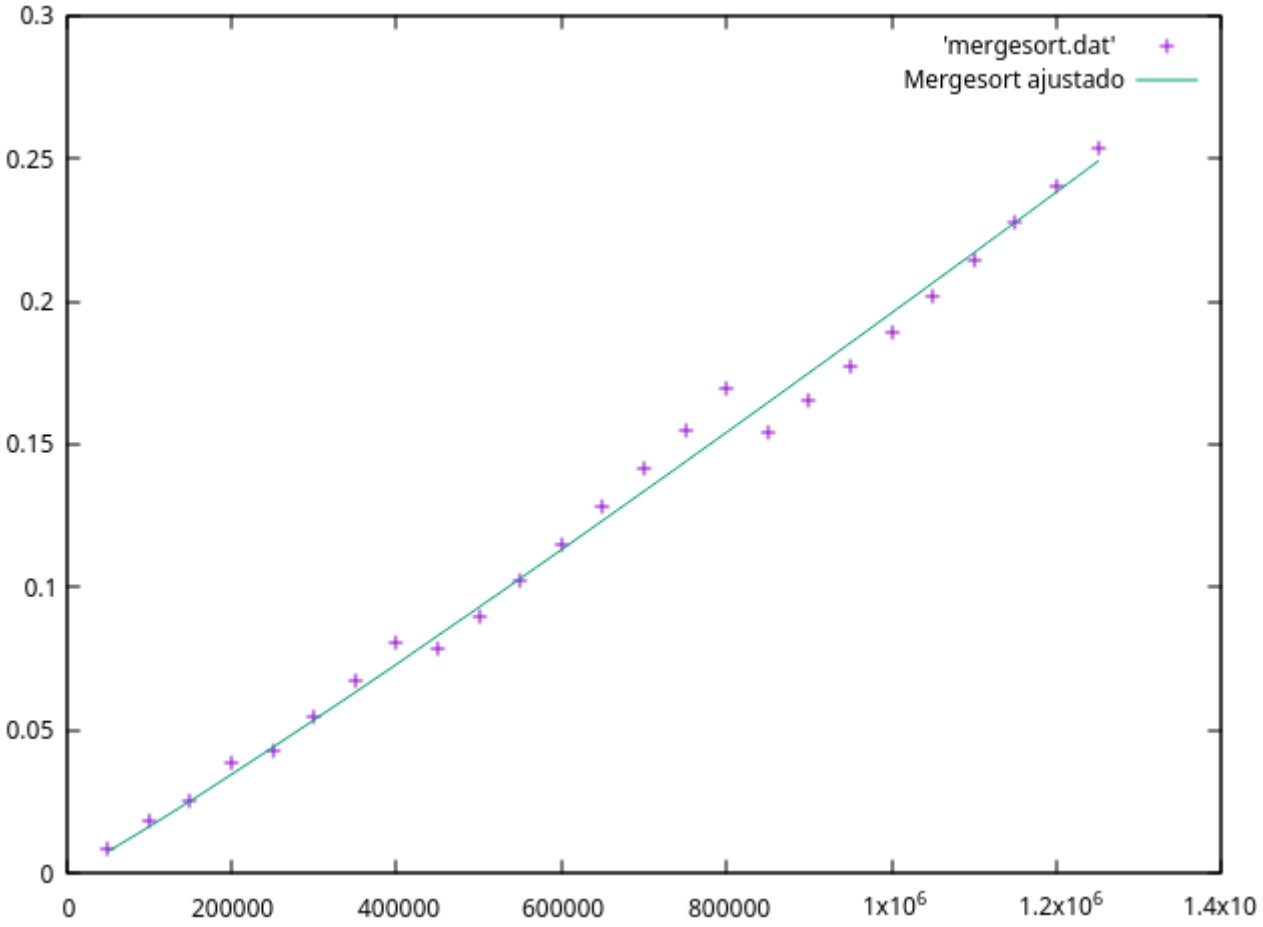


Figura 5: Ajuste de Mergesort

2.2.2. Ordenación por Quicksort

Junto a MergeSort, es el algoritmo más rápido de los 8 que hemos visto.

La función utilizada es la siguiente:

```
inline void quicksort(int T[], int num_elem)
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);
  };
}
```

Tras compilar y ejecutar `./quicksort`:

Tamaño	Tiempo(seg)
50000	0.0157172
100000	0.0325483
150000	0.0496276
200000	0.0683057
250000	0.0857514
300000	0.105036
350000	0.123697
400000	0.142723
450000	0.160988
500000	0.178935
550000	0.19783
600000	0.218152
650000	0.237553
700000	0.256666
750000	0.281156
800000	0.296435
850000	0.315857
900000	0.337722
950000	0.360285
1000000	0.376351

1050000	0.398741
1100000	0.417686
1150000	0.4358
1200000	0.455506
1250000	0.473686

Al graficar los datos vemos lo siguiente:

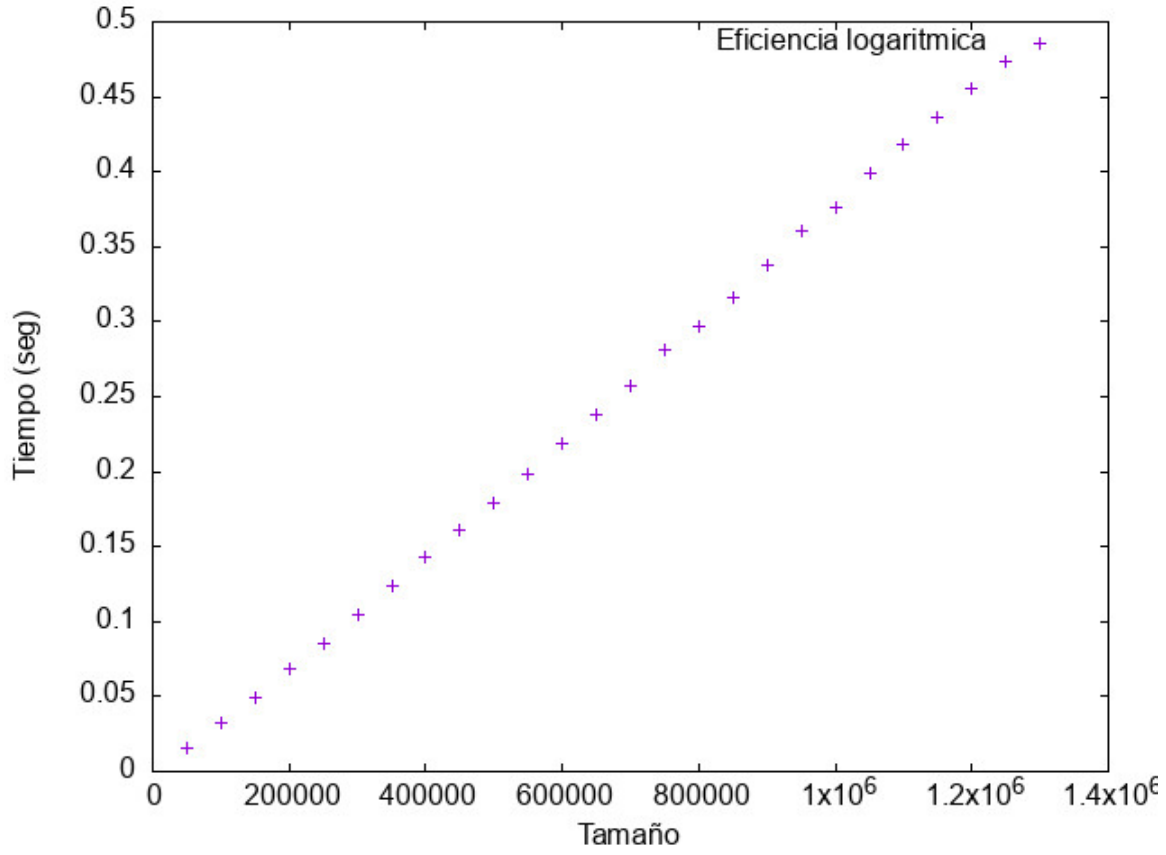


Figura 6: Eficiencia Algoritmo QuickSort

A pesar de parecer lineal, en realidad es logaritmica pues lo vamos a comprobar ahora mismo:

```
FIT:    data read from 'quicksort.dat'
        format = z
        #datapoints = 25
        residuals are weighted equally (unit weight)

function used for fitting: f(x)
        f(x) = a*x*log(x) + b
fitted parameters initialized with current variable values

iter      chisq       delta/lim  lambda   a             b
   0 6.6724879056e-05   0.00e+00  1.96e-01    2.710439e-08   2.021386e-03
   1 6.6724879056e-05   0.00e+00  1.96e+00    2.710439e-08   2.021386e-03

After 1 iterations the fit converged.
final sum of squares of residuals : 6.67249e-05
rel. change during last iteration : 0

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a               = 2.71044e-08      +/- 6.623e-11    (0.2444%)
b               = 0.00202139       +/- 0.0006759    (33.44%)

correlation matrix of the fit parameters:
                a      b
a               1.000
b              -0.864  1.000
```

Quedando la fórmula ajustada como:

$$2{,}710\,44 \times 10^{-8}\, n \log(n) + 0{,}00202139$$

La gráfica ajustada resultante es:

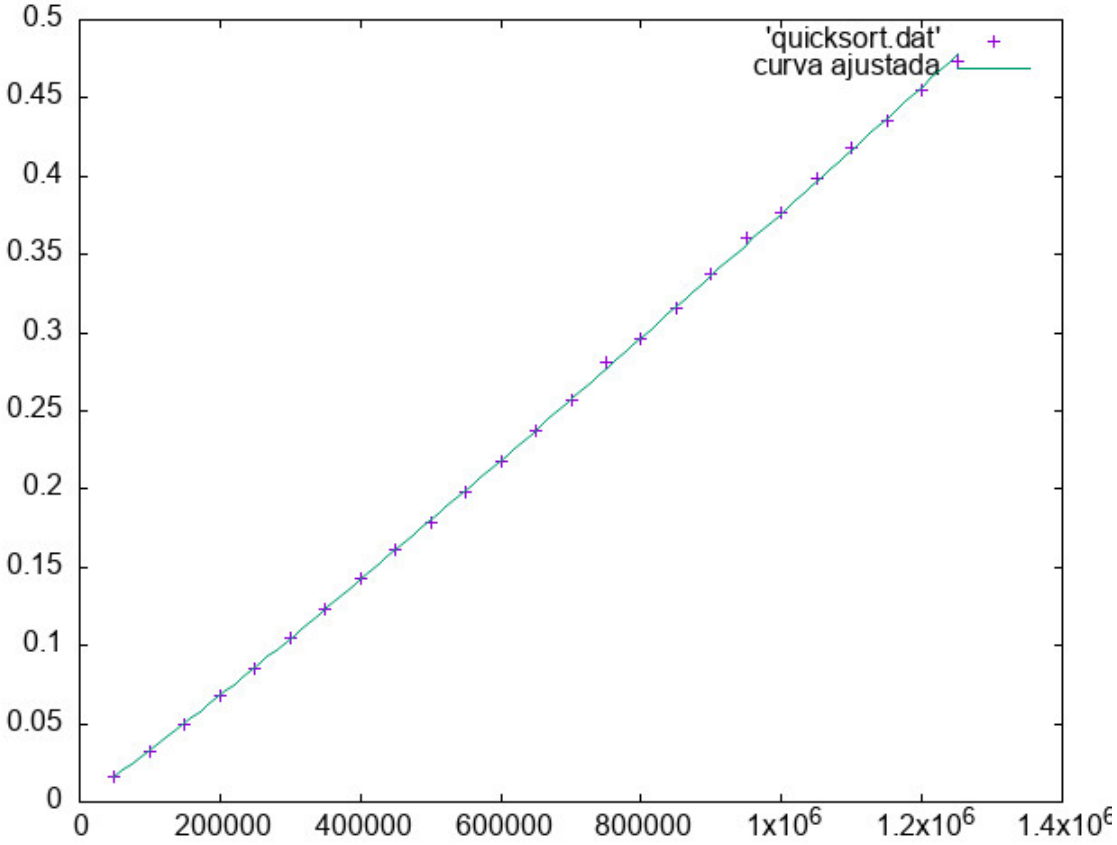


Figura 7: Ajuste de Quicksort

3. Otros Algoritmos

3.1. Eficiencia n^3

3.1.1. Algoritmo de Floyd

A continuación, a vemos un algoritmo que tiene la finalidad de calcular el costo del mínimo camino de un grafo dirigido.

Para calcularlo siguiendo las instrucciones del guión hemos ejecutado el algoritmo con la macro del guión, variando su tamaño de 50 a 50 hasta 1250. Mostramos la siguiente tabla con los datos del archivo generado `.dat`.

El código de Floyd utilizado es:

```
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];
              }
}
```

La tabla de datos del floyd es:

Tamaño	Tiempo(seg)
50	0.00189273
100	0.0112866
150	0.0175568
200	0.0525859
250	0.0895145
300	0.191243
350	0.217314
400	0.330062
450	0.517893
500	0.697055
550	0.933725
600	1.27572
650	1.49303
700	2.02039
750	2.46124
800	2.96186
850	3.5521
900	4.10995
950	4.81961
1000	5.70665
1050	6.56104
1100	7.47344

1150	8.87397
1200	9.59515
1250	10.9697

Tras modificar el código fuente y compilar, obtenemos los siguientes datos del `.dat`:

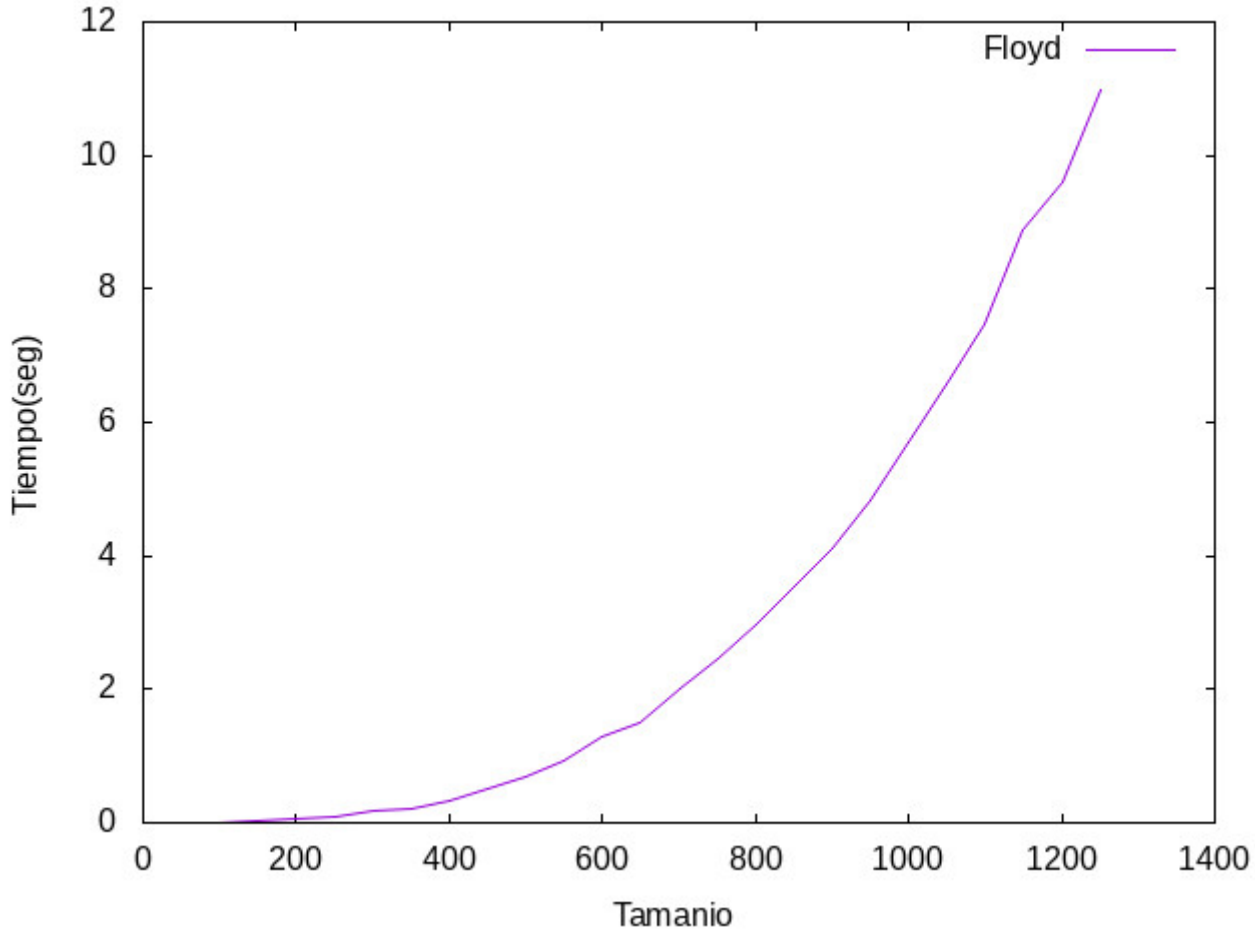


Figura 8: Algoritmo Floyd

El ajuste/búsqueda de variables ocultas genera el siguiente log:

```
******************************************************************************
Mon Mar 21 15:30:04 2022


FIT:        data read from "floyd.dat"
            format = z
            #datapoints = 25
            residuals are weighted equally (unit weight)

function used for fitting: f(x)
    f(x) = a0*x*x*x+a1*x*x+a2*x+a3
fitted parameters initialized with current variable values

iter          chisq           delta/lim  lambda   a0
   0 1.5636145726e+19   0.00e+00  3.95e+08        1.000000e+00
  12 1.3210841975e-01  -1.85e-02  3.95e-04        5.093144e-09

After 12 iterations the fit converged.
final sum of squares of residuals : 0.132108
rel. change during last iteration : -1.85173e-07

degrees of freedom        (FIT_NDF)                              : 21
rms of residuals          (FIT_STDFIT) = sqrt(WSSR/ndf)          : 0.079315
variance of residuals (reduced chisquare) = WSSR/ndf    : 0.00629088

Final set of parameters                Asymptotic Standard Error
=======================                ==========================
a0                  = 5.09314e-09          +/- 4.346e-10        (8.534%)
a1                  = 9.66697e-07          +/- 8.585e-07        (88.81%)
a2                  = -0.000409019         +/- 0.0004853        (118.6%)
a3                  = 0.0393051            +/- 0.0743           (189%)

correlation matrix of the fit parameters:
                    a0     a1      a2      a3
a0                   1.000
a1                  -0.987  1.000
a2                   0.926 -0.973  1.000
a3                  -0.719  0.795 -0.898  1.000
```

La función resultado del ajuste:

$$f(x) = 5{,}093\,14 \times 10^{-9}\, x^3 + 9{,}666\,97 \times 10^{-7}\, x^2 - 0{,}000409019x + 0{,}0393051$$

La gráfica ajustada queda:

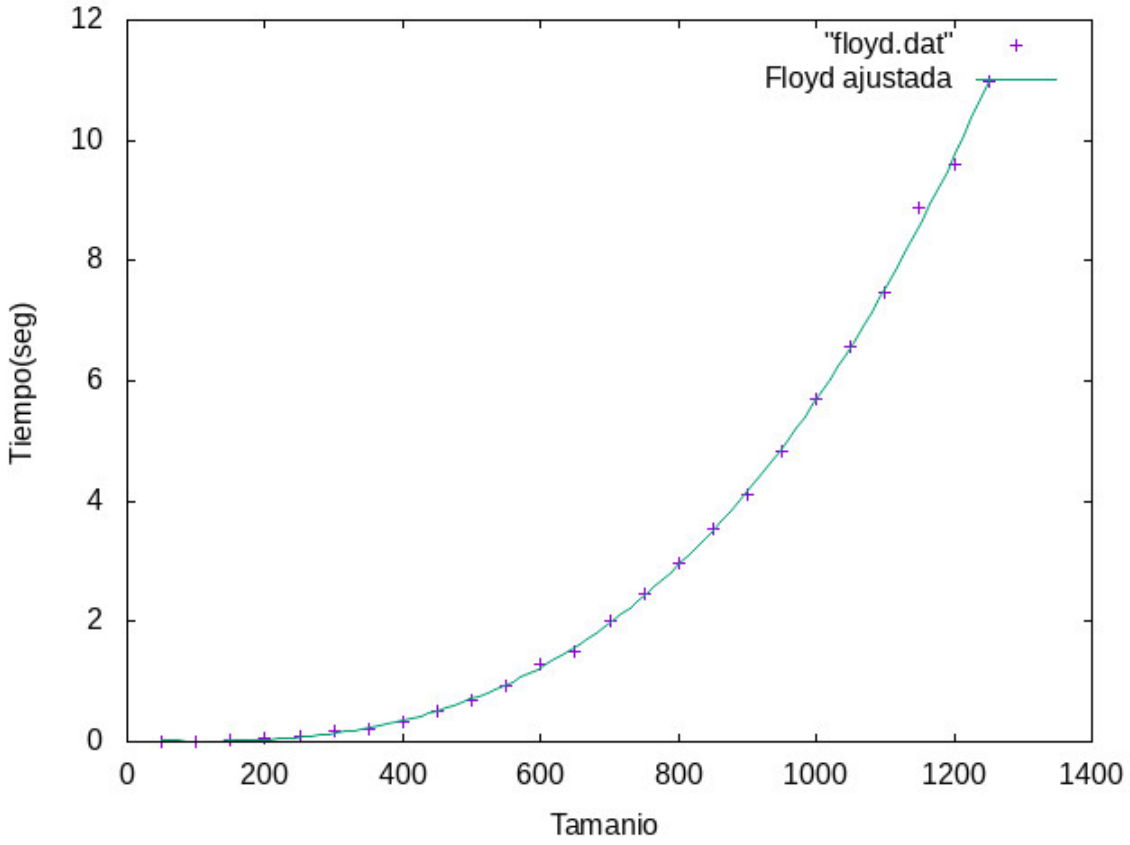


Figura 9: Floyd Ajustado

3.1.2. Algoritmo de dijkstra

A continuación, vemos el algoritmo recursivo de Dijkstra un algoritmo que tiene el mismo fin pero utiliza un procedimiento recursivo.

El código utilizado es el siguiente:

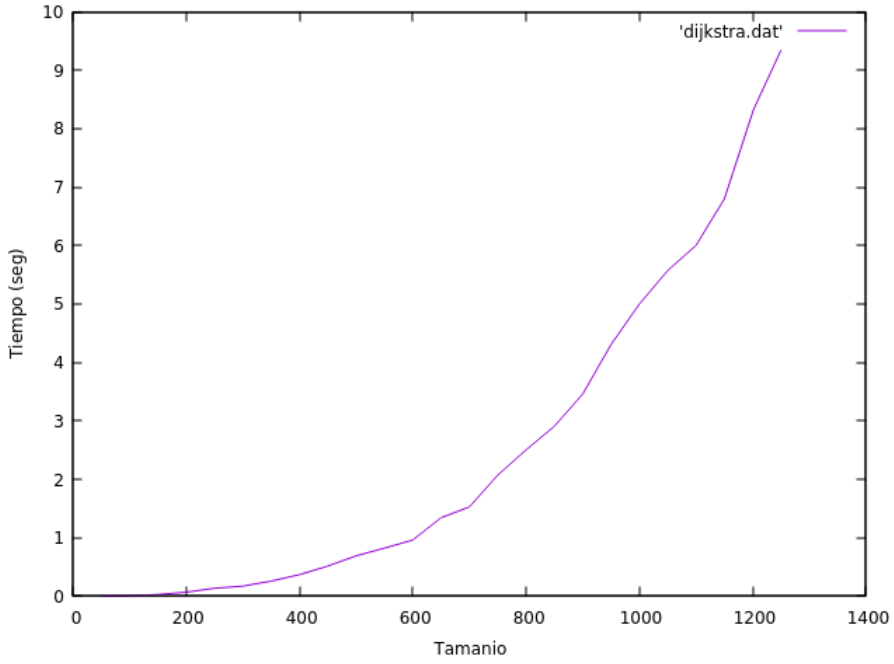
```
 void Dijkstra(int **M, int **Sal, int dim, int src) // adjacency matrix
{
        int dist[dim]; // integer array to calculate minimum distance
        bool Tset[dim];// boolean array to mark visted/unvisted

        // set the nodes with infinity distance
        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);
                Tset[m]=true;// m with minimum distance included in Tset.
                for(int i = 0; i<dim; i++)
                {
                        // Updating the minimum distance
                        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];
}
```

La tabla de datos de dijkstra:

Tamaño	Tiempo(seg)
50	0.00283111
100	0.00949102
150	0.0285824
200	0.0704248
250	0.136364
300	0.171888
350	0.259129

400	0.369816
450	0.515604
500	0.690241
550	0.822436
600	0.962625
650	1.34316
700	1.52782
750	2.07249
800	2.50367
850	2.90906
900	3.4613
950	4.30861
1000	5.00108
1050	5.57536
1100	6.00137
1150	6.79705
1200	8.28284
1250	9.33255

Cuya gráfica asociada es:



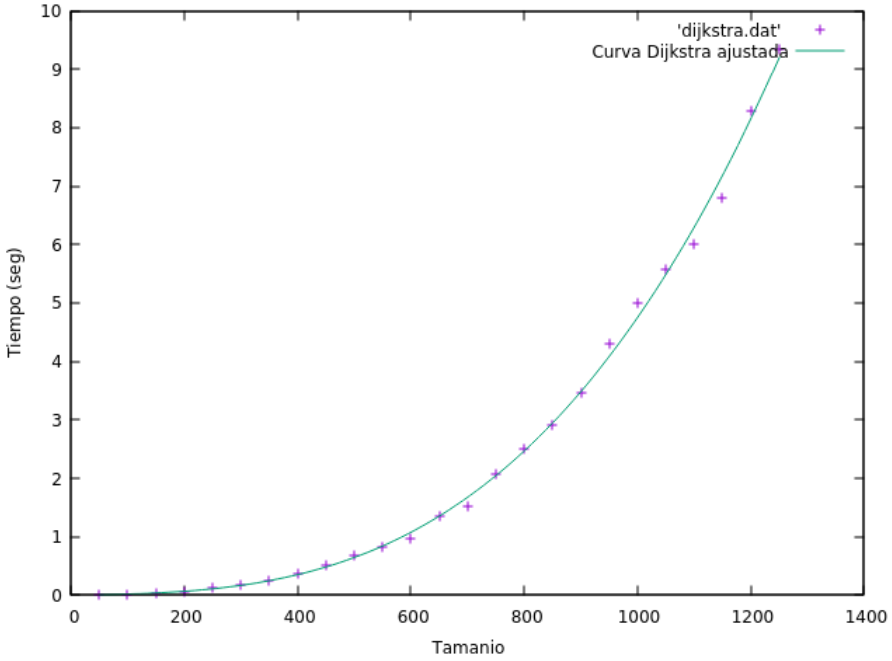
Calculamos el ajuste con gnuplot:

```
*******************************************************************************
Mon Mar 21 15:20:52 2022


FIT:    data read from 'dijkstra.dat'
        format = z
        #datapoints = 25
        residuals are weighted equally (unit weight)

function used for fitting: f(x)
        f(x)=a3*x*x*x+a2*x*x+a1*x+a0
fitted parameters initialized with current variable values

iter      chisq       delta/lim  lambda       a3             a2
   0 1.5636145755e+19   0.00e+00  3.95e+08    1.000000e+00   1.000000e+00
  12 4.3580634335e-01  -6.12e-03  3.95e-04    4.705614e-09  -1.473596e-07

After 12 iterations the fit converged.
final sum of squares of residuals : 0.435806
rel. change during last iteration : -6.11875e-08

degrees of freedom    (FIT_NDF)                        : 21
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 0.144058
variance of residuals (reduced chisquare) = WSSR/ndf   : 0.0207527

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a3              = 4.70561e-09      +/- 7.894e-10    (16.78%)
a2              = -1.4736e-07      +/- 1.559e-06    (1058%)
a1              = 0.000192697      +/- 0.0008814    (457.4%)
a0              = -0.00275768      +/- 0.135        (4894%)

correlation matrix of the fit parameters:
                a3     a2     a1     a0
a3              1.000
a2             -0.987  1.000
a1              0.926 -0.973  1.000
a0             -0.719  0.795 -0.898  1.000
```

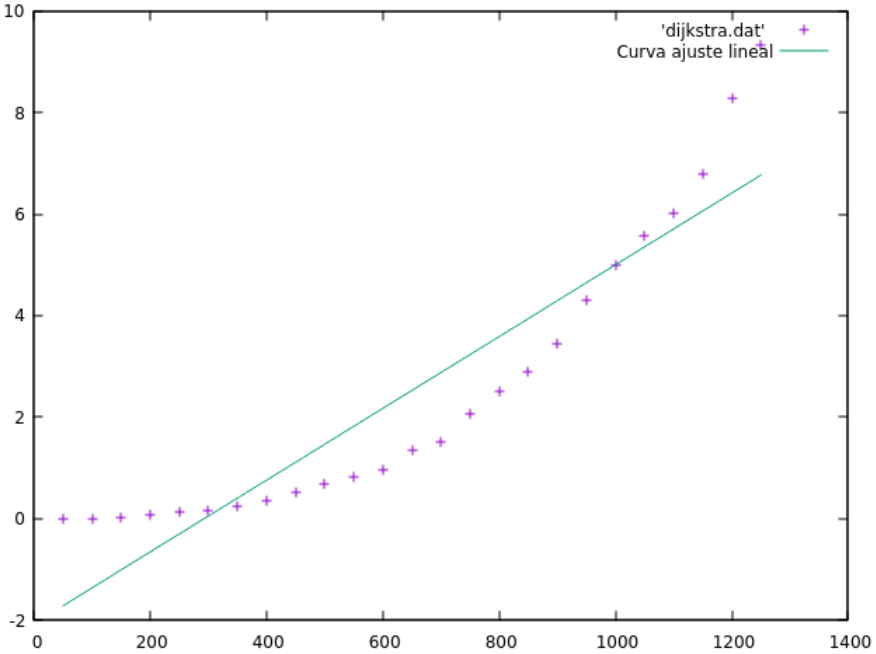
La formula ajustada es:

$$f(x) = 4{,}705\,61 \times 10^{-9}x^3 - 1{,}4736 \times 10^{-7}x^2 + 0{,}000192697x - 0{,}00275768$$

La gráfica con el ajuste queda:



A continuación, vamos a probar otros ajustes que no son buenos para el algoritmo de Dijkstra. Probamos un ajuste lineal.



Como podemos observar el ajuste es completamente malo, no es para nada la función real. Probamos con un ajuste con n^4:

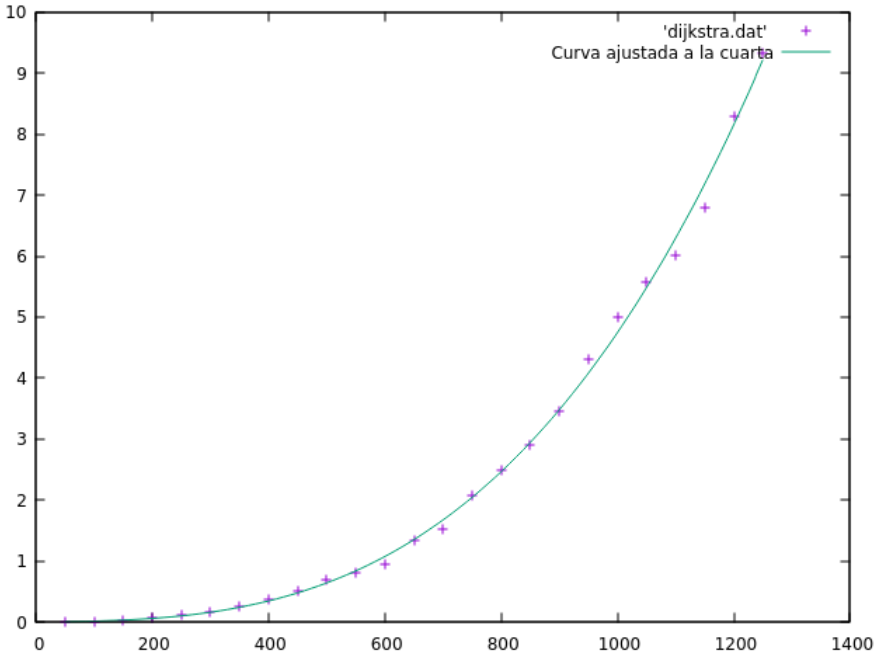


Figura 10: Ajuste n^4 de Dijkstra

Podemos observar que en ciertos tamaños pequeños del algoritmo de Dijkstra, con un ajuste superior al suyo propio, el ajuste queda bien.

3.2. Eficiencia a^n

Los algoritmos de orden a^n son el algoritmo de hanoi y el algoritmo recursivo de fibonacci que se utilizan respectivamente para resolver el famoso problema de las torres de Hanoi y calcular los números de la sucesión de Fibonacci.

3.2.1. Algoritmo de fibonacci

El algoritmo de fibonacci es un clásico de la programación y de las matemáticas. Su código recursivo es bastante simple:

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

Con unos datos suficientemente altos podemos obtener su eficiencia, obteniendo la tabla:

Tamaño	Tiempo(seg)
2	3.5e-07
4	3.9e-07
6	7.31e-07
8	1,34E-03
10	2,58E-03
12	5,01E-03
14	1.0349e-05
16	2.4075e-05
18	5.6516e-05
20	0.000141576
22	0.000362882
24	0.000991363
26	0.00248774
28	0.00645781
30	0.0172408
32	0.045158
34	0.11677
36	0.304076
38	0.799934
40	2.08225
42	5.75886
44	14.5311
46	37.9627
48	98.5858
50	258313

La gráfica queda:

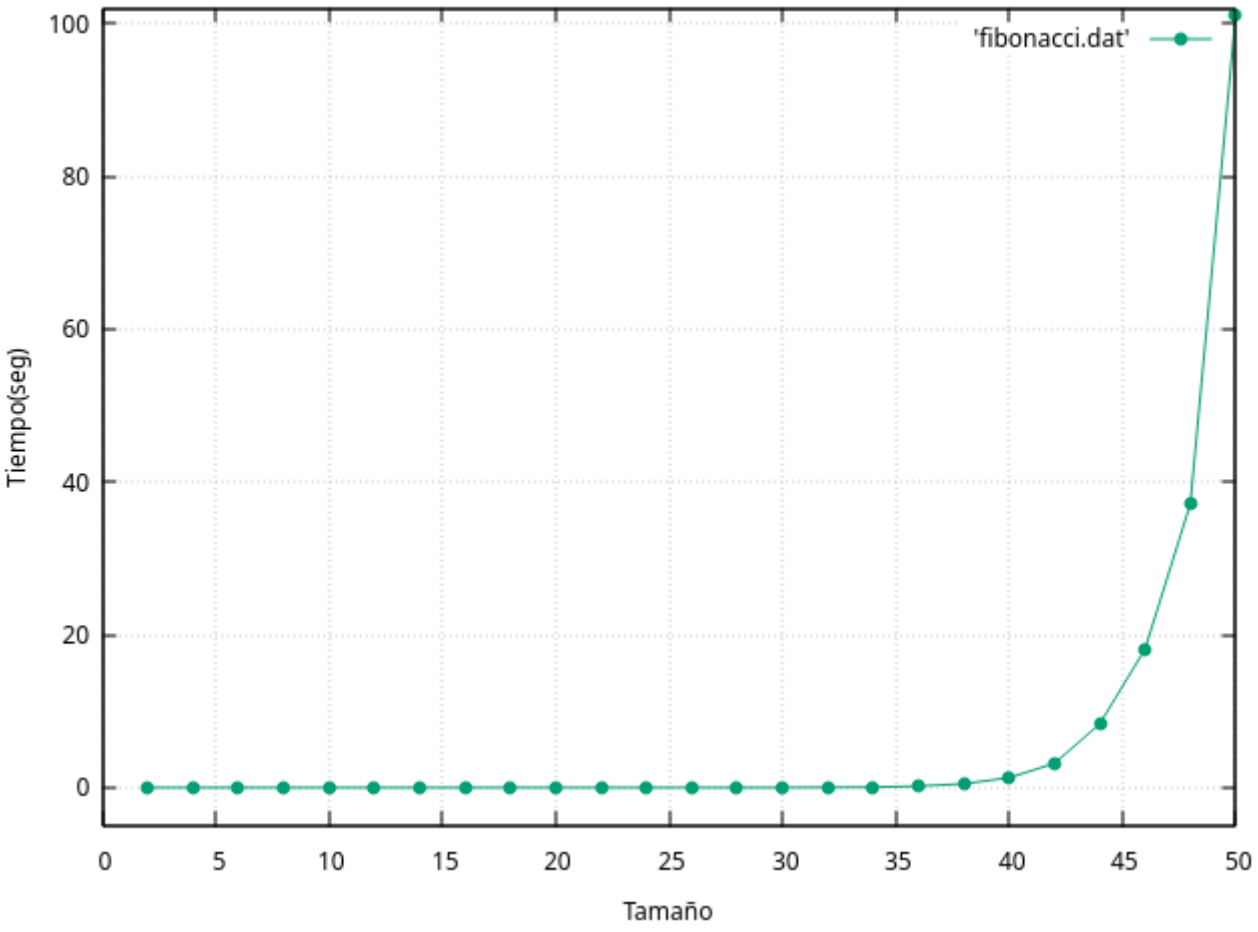


Figura 11: Algoritmo de Fibonacci

El ajuste del algoritmo en gnuplot es el siguiente:

```
*******************************************************************************
Sun Mar 20 21:44:24 2022


FIT:    data read from 'fibonacci.dat'
        format = z
        #datapoints = 25
        residuals are weighted equally (unit weight)

function used for fitting: f(x)
        f(x) = a*1.618**x + b
fitted parameters initialized with current variable values

iter      chisq       delta/lim  lambda   a             b
   0 9.2543992530e+20   0.00e+00  4.30e+09    1.000000e+00   1.000000e+00
   5 2.1871058673e+01  -2.44e-05  4.30e+04    9.139440e-09   1.000000e+00

After 5 iterations the fit converged.
final sum of squares of residuals : 21.8711
rel. change during last iteration : -2.44244e-10

degrees of freedom    (FIT_NDF)                        : 23
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 0.975149
variance of residuals (reduced chisquare) = WSSR/ndf   : 0.950916

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a               = 9.13944e-09      +/- 3.359e-11    (0.3676%)
b               = 1                +/- 0.2044       (20.44%)

correlation matrix of the fit parameters:
                a      b
a               1.000
b              -0.299  1.000
```

La función ajuste obtenida:

$$f(x) = 9{,}139\,44 \times 10^{-9} \left(\frac{1+\sqrt{5}}{2} \right)^{n} + 1$$

La gráfica ajustada es:

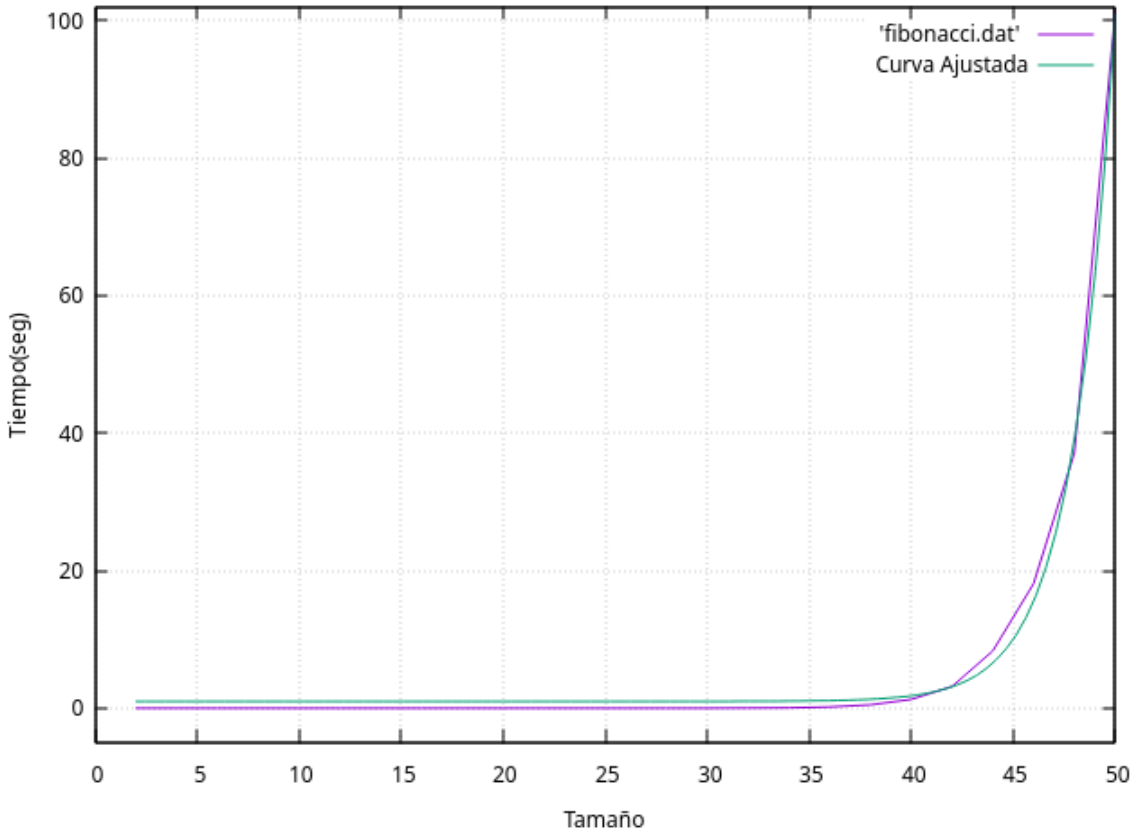


Figura 12: Ajuste de Fibonacci

3.2.2. Algoritmo de hanoi

El código utilizado para el algoritmo de hanoi es:

```
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);
  }
}
```

La tabla de datos resultante de este algoritmo es la siguiente:

Tamaño	Tiempo(seg)
3	3.81e-07
4	2.4e-07
5	3.4e-07
6	4.91e-07
7	8.12e-07
8	1.392e-06
9	5.701e-06
10	4.85e-06
11	9.338e-06
12	1.8375e-05
13	3.631e-05
14	7.5824e-05
15	0.000144055
16	0.000289803
17	0.000578454
18	0.00115254
19	0.00230918
20	0.00461556
21	0.00921158
22	0.0184436
23	0.0367877
24	0.0738795
25	0.147934
26	0.295861
27	0.592275
28	1.18338
29	2.36899
30	4.7301
31	9.49097
32	18.998
33	37.9925

Quedando la siguiente gráfica:

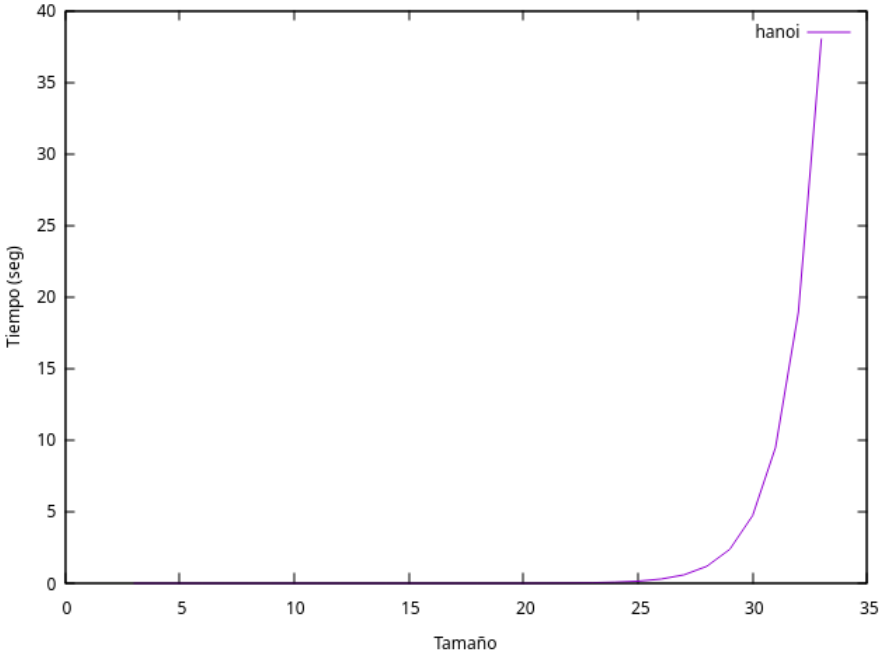


Figura 13: Algoritmo de hanoi

El ajuste y búsqueda de constantes ocultas en gnuplot queda:

```
Ajustada a la fórmula a0*2x
iter          chisq           delta/lim  lambda   a0
   0 9.3672982738e+03   0.00e+00  2.53e+01        1.418029e-08
   4 4.5210289275e-04  -2.18e-09  2.53e-03        4.422579e-09
iter          chisq           delta/lim  lambda   a0

After 4 iterations the fit converged.
final sum of squares of residuals : 0.000452103
rel. change during last iteration : -2.1823e-14

degrees of freedom        (FIT_NDF)                              : 30
rms of residuals          (FIT_STDFIT) = sqrt(WSSR/ndf)          : 0.00388202
variance of residuals (reduced chisquare) = WSSR/ndf    : 1.50701e-05

Final set of parameters                 Asymptotic Standard Error
=======================                 ==========================
a0                  = 4.42258e-09           +/- 3.914e-13        (0.00885%)
```

La función ajustada que se obtiene es la siguiente:

$$f(x) = 4{,}422\,58 \times 10^{-9} \times 2^{x}$$

La gráfica resultante del ajuste es:

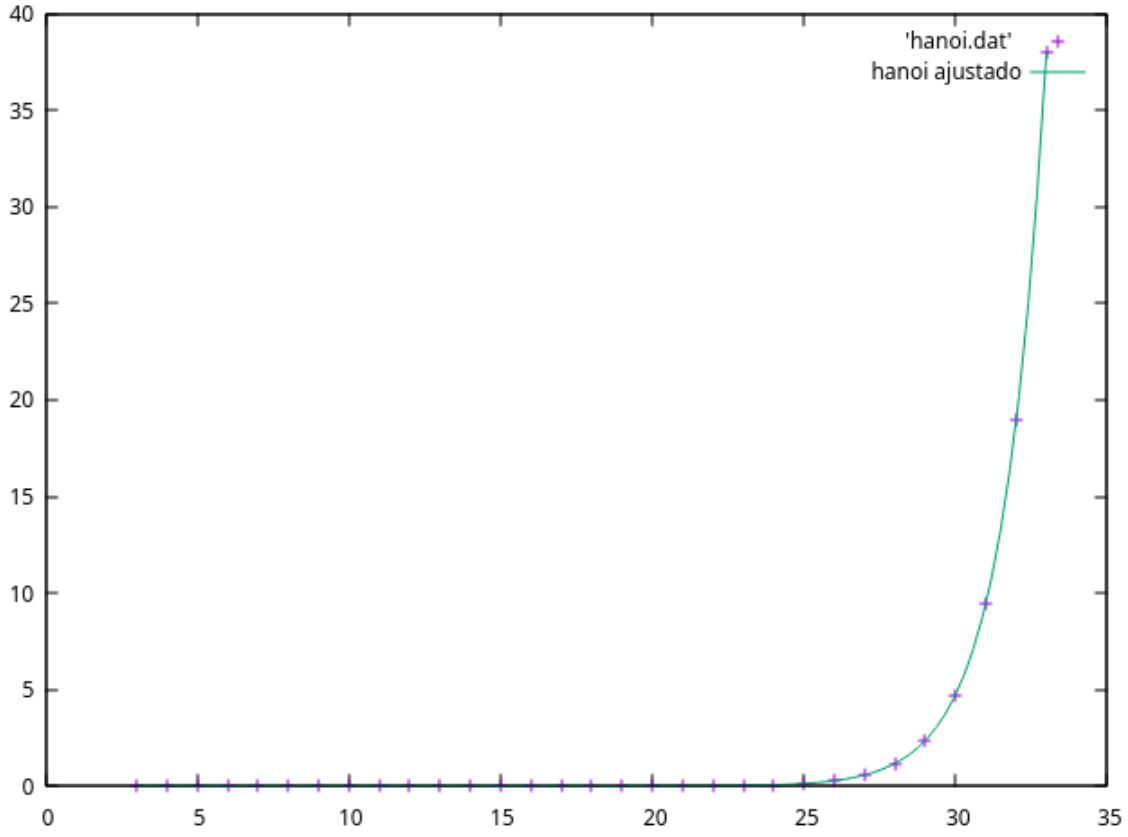


Figura 14: Ajuste Hanoi

4. Variación de la eficiencia empírica por factores externos

Hemos elegido el algoritmo de burbuja para ver como se comporta con diferentes grados de optimización, distintos SO e incluso modos de rendimiento en el mismo ordenador.

4.1. Flags de Optimización

Tamaño	Tiempo(seg)
5000	0.0593889
10000	0.214639
15000	0.497034
20000	0.920987
25000	1.43304
30000	2.09295
35000	2.86473
40000	3.77359
45000	4.80492
50000	5.90436
55000	7.18508
60000	8.52722
65000	10.0494
70000	11.6697
75000	13.4947
80000	15.3079
85000	17.3202
90000	19.3166
95000	21.4845
100000	24.2024
105000	26.3043
110000	28.9074
115000	31.5045
120000	34.3248
125000	37.2732

Cuadro 10: -Ofast

Tamaño	Tiempo(seg)
5000	0.0353615
10000	0.12893
15000	0.279444
20000	0.519757
25000	0.848564
30000	1.22494
35000	1.69467
40000	2.22243
45000	2.86132
50000	3.54748
55000	4.28462
60000	5.1476
65000	6.03558
70000	7.023
75000	8.09728
80000	9.21789
85000	10.6009
90000	11.7605
95000	13.1332
100000	14.4467
105000	16.195
110000	17.6103
115000	19.6109
120000	21.3958
125000	37.8755

Cuadro 11: -Os

Tamaño	Tiempo(seg)
5000	0.040981
10000	0.121329
15000	0.263826
20000	0.514343
25000	0.794101
30000	1.17507
35000	1.61848
40000	2.21165
45000	2.89982
50000	3.46264
55000	4.26109
60000	5.10696
65000	6.01018
70000	7.32373
75000	8.25187
80000	9.10546
85000	10.8578
90000	11.9108
95000	13.0263
100000	14.8928
105000	15.9502
110000	17.4961
115000	19.4555
120000	21.0152
125000	22.5864

Cuadro 13: -O1

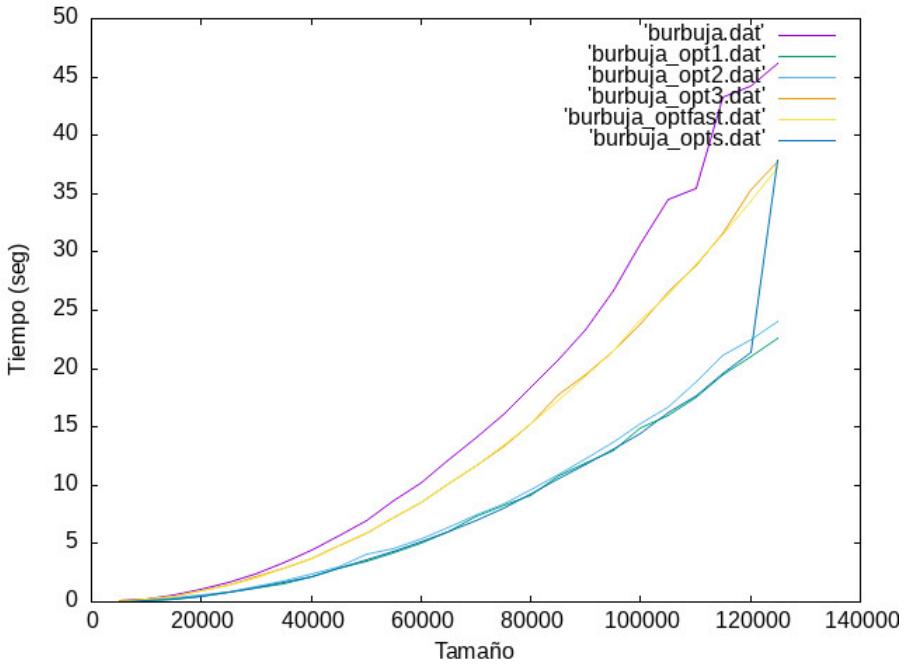
Tamaño	Tiempo(seg)
5000	0.0379419
10000	0.147648
15000	0.306699
20000	0.586954
25000	0.89294
30000	1.3054
35000	1.78492
40000	2.37671
45000	3.03008
50000	4.10658
55000	4.5193
60000	5.4013
65000	6.37231
70000	7.44308
75000	8.46187
80000	9.67369
85000	10.9855
90000	12.2241
95000	13.6452
100000	15.2075
105000	16.7299
110000	18.9098
115000	21.1148
120000	22.4663
125000	24.0422

Cuadro 15: -O2

Tamaño	Tiempo(seg)
5000	0.0810466
10000	0.22743
15000	0.515814
20000	0.909461
25000	1.43552
30000	2.11438
35000	2.88025
40000	3.76616
45000	4.825
50000	5.90919
55000	7.17914
60000	8.53562
65000	10.0452
70000	11.6576
75000	13.3829
80000	15.2273
85000	17.7326
90000	19.4547
95000	21.5236
100000	23.8546
105000	26.5079
110000	28.8266
115000	31.6179
120000	35.3284
125000	37.725

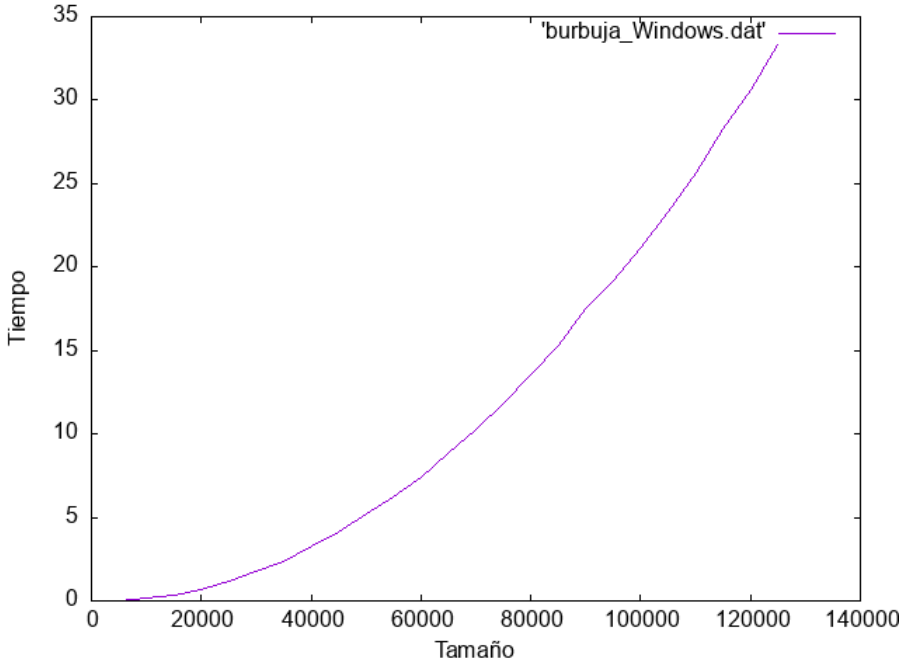
Cuadro 17: -O3

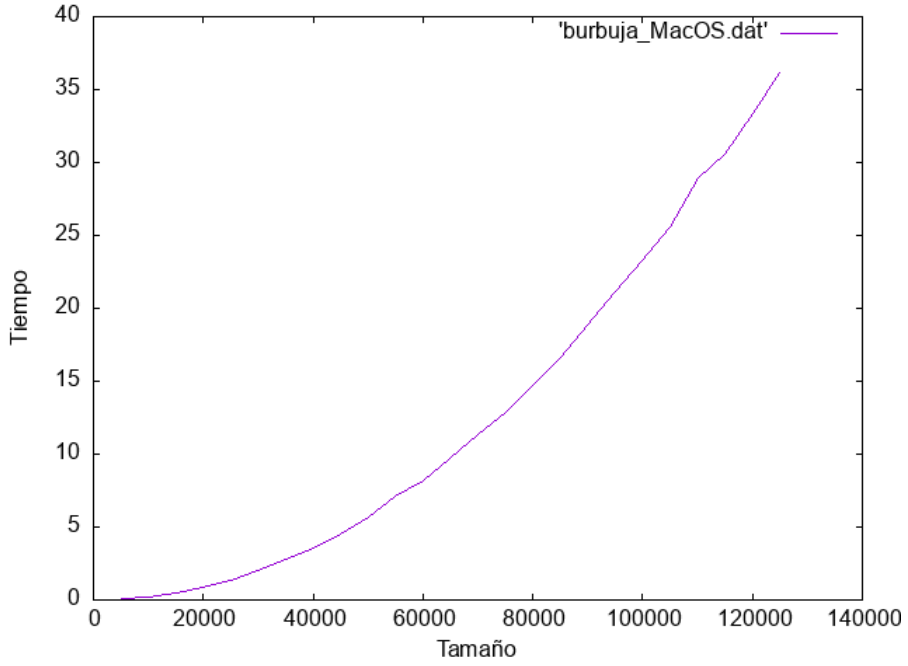
La grafica comparativa resultante es:



4.2. Distintos Sistemas Operativos

Además hemos probado el algoritmo de Burbuja en distinos Sistemas operativos como Windows 10 y MacOS:





Tras la comparativa final, sorprendentemente en windows tarda menos:

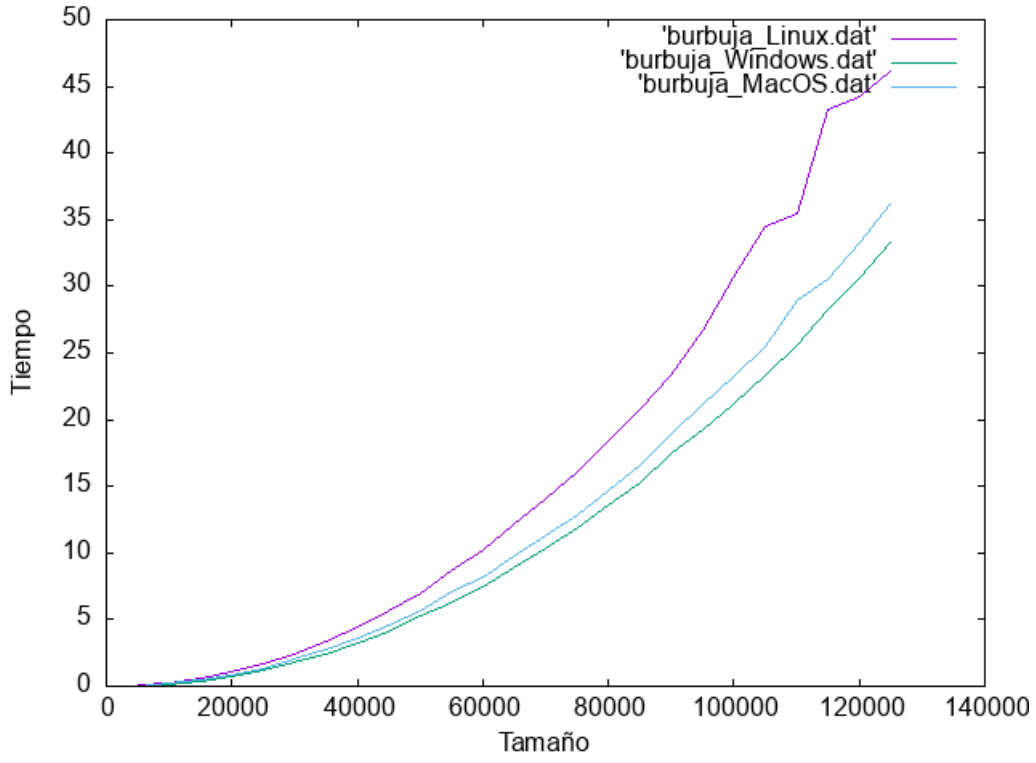


Figura 15: Comparación entre SO

5. Los tiempos de todos los algoritmos

En esta sección compararemos todos los algoritmos en la misma gráfica y veremos la diferencia de eficiencia entre los distintos órdenes de algoritmos que hemos medido.

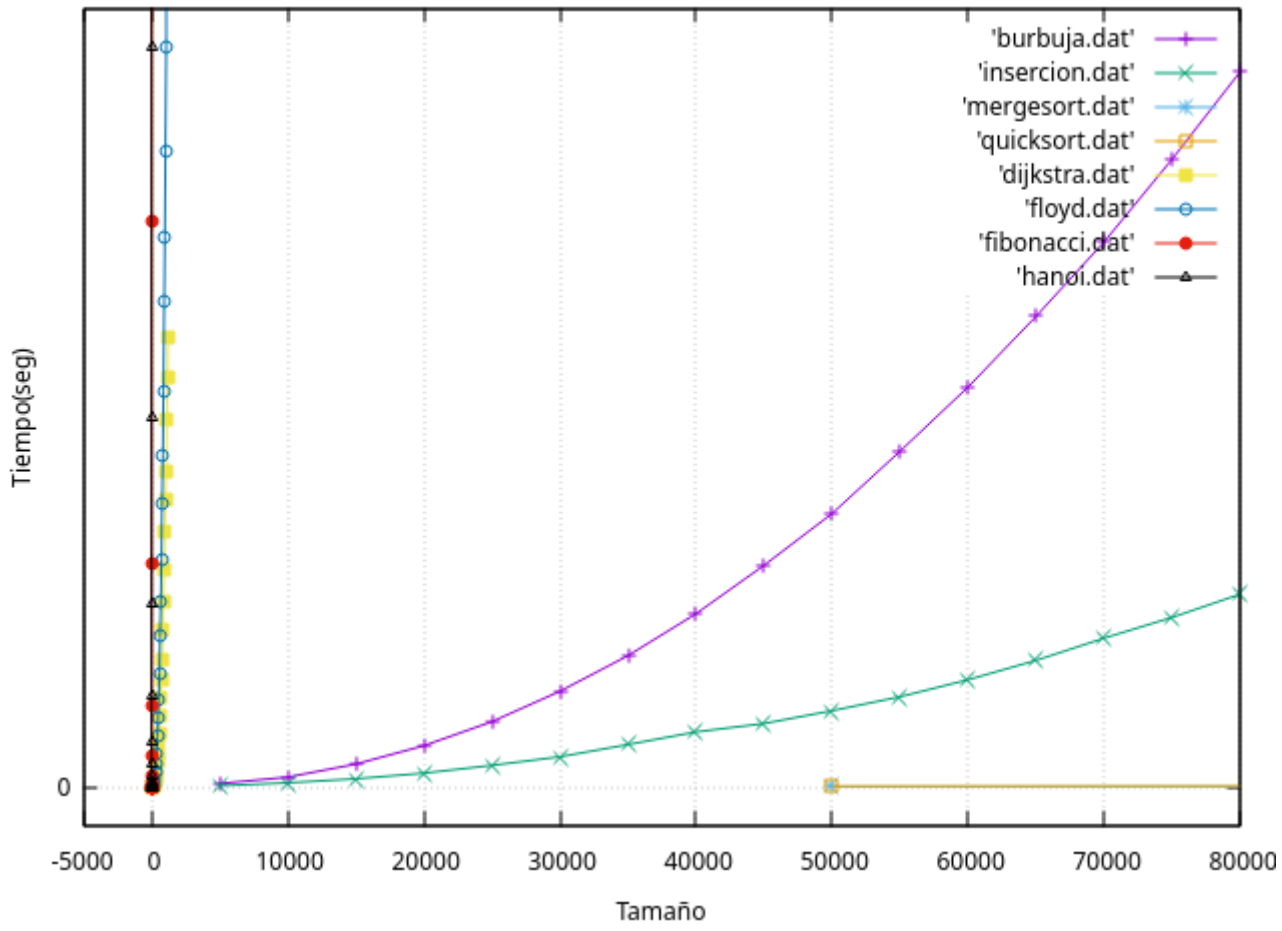


Figura 16: La eficiencia de todos los algoritmos

Ahora obtendremos específicamente podemos ver la diferencia en la gráfica de los algoritmos de ordenación.

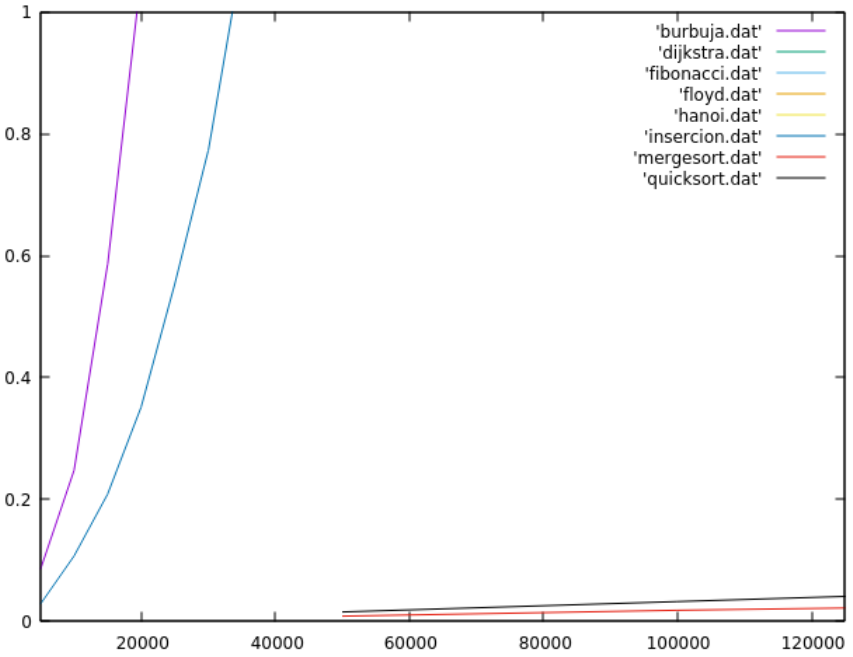


Figura 17: La eficiencia de los algorítmos de ordenación

Las gráficas por eficiencia de algoritmos son:

- Eficiencia n^2:

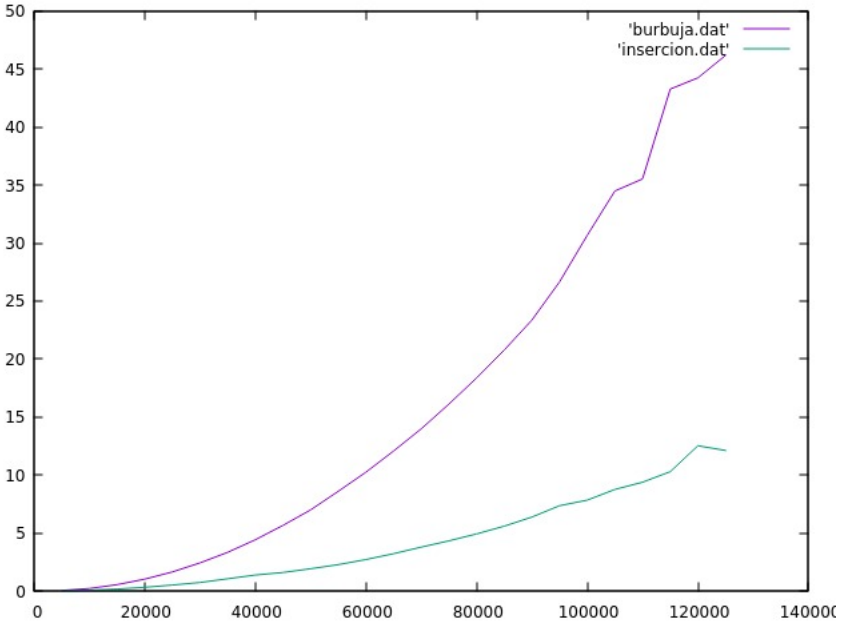
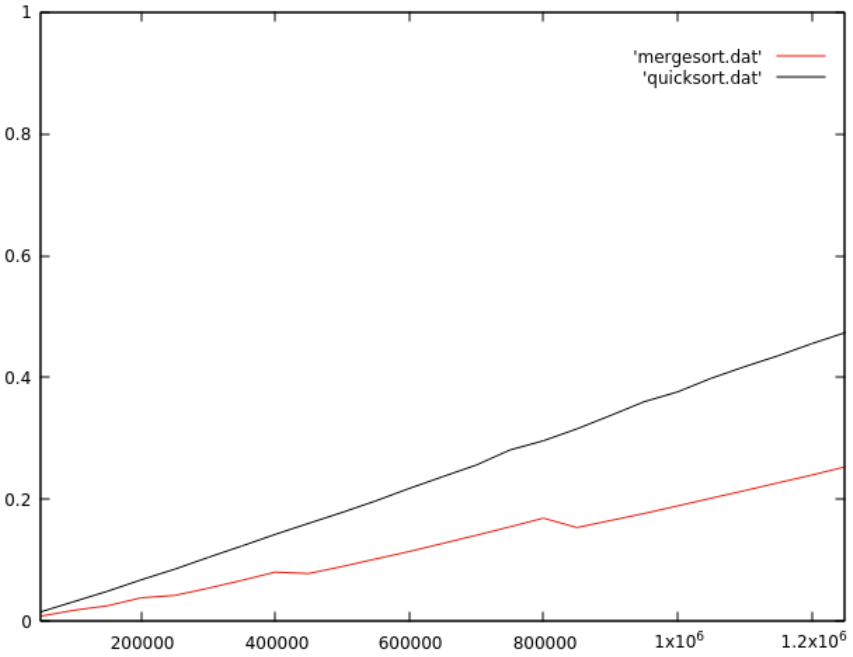


Figura 18: La eficiencia de los algorítmos n^2

- Eficiencia nlog(n)



- Eficiencia n^3

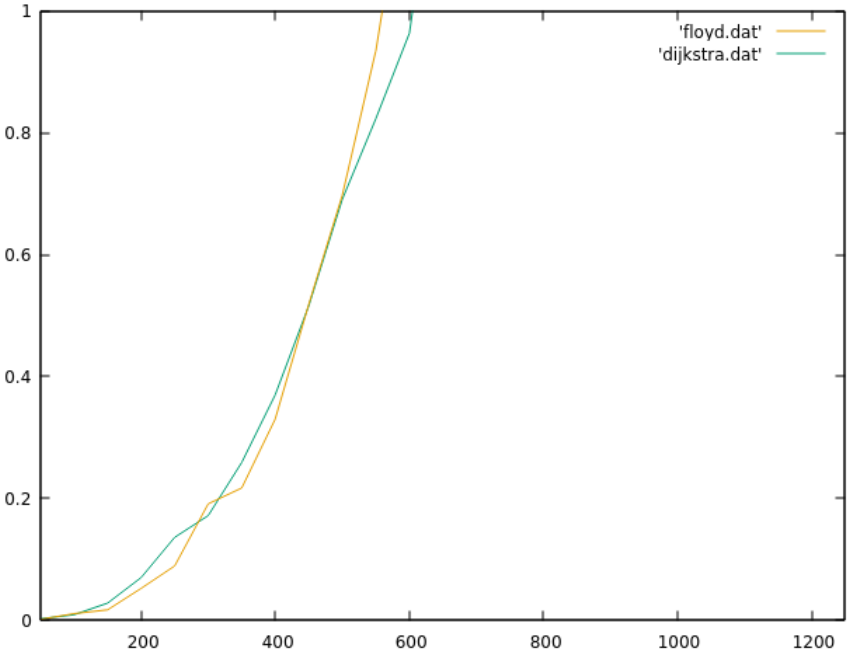


Figura 19: La eficiencia de los algorítmos n^3

- Eficiencia a^n

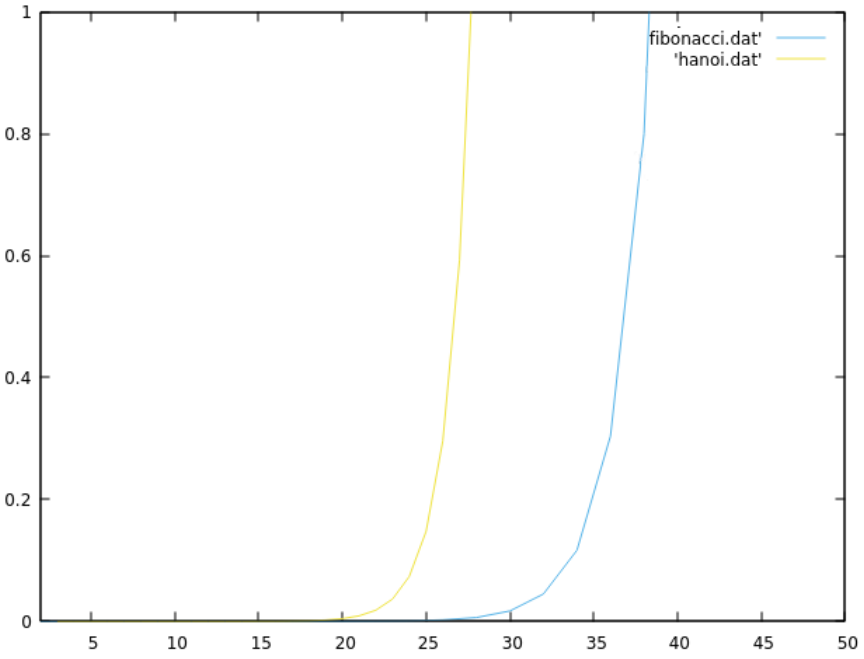


Figura 20: La eficiencia de los algorítmos a^n