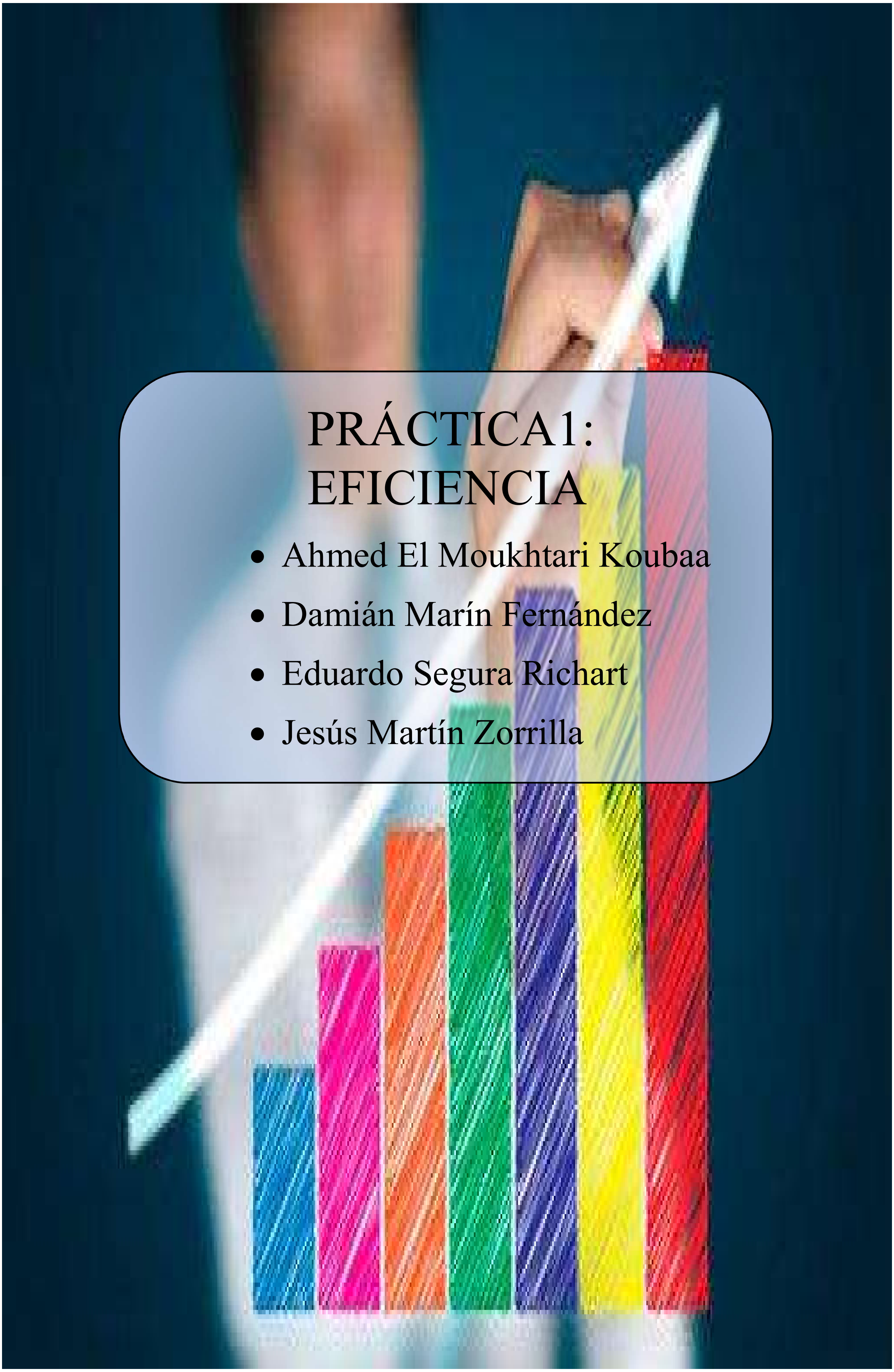


PRÁCTICA1: EFICIENCIA

- Ahmed El Moukhtari Koubaa
- Damián Marín Fernández
- Eduardo Segura Richart
- Jesús Martín Zorrilla

ÍNDICE

1. Descripción del problema.

En esta práctica se nos plantea realizar el “análisis” de varios algoritmos. Entre estos se encuentran los algoritmos de ordenación básicos (burbuja, inserción, selección), algunos de los algoritmos de ordenación más eficientes (quicksort, heapsort, mergesort) y dos algoritmos menos eficientes (Floyd y Hanói).

Para realizar este análisis se nos pide llevar a cabo varias ejecuciones de los algoritmos con diferentes tamaños de entrada, es decir, aplicarlos en diferentes casos, a esto lo conocemos como el cálculo de la eficiencia empírica.

Una vez calculadas las eficiencias empíricas y hechas las tablas debemos realizar gráficas comparando los resultados obtenidos, esto podemos hacerlo con la herramienta que prefiramos (Gnuplot, Xmgrace, Excel, etc).

Sobre las gráficas se nos pide calcular la eficiencia híbrida, la cual se nos indica realizar con alguna de las herramientas las que disponemos.

Se cita en el guion que un aspecto importante es el efecto de los parámetros externos, de los cuales debemos hablar y sugerir un estudio de este tipo consultándolo con el profesor y llevándolo a cabo.

2. Implementación del Código.

Todo el código que usamos en esta práctica ha sido aportado en la documentación de la misma, aunque cabe destacar ciertas modificaciones que hemos realizado por diversas razones como comodidad o seguir una medida única.

Así pues, se han editado los códigos para que todos los programas acepten la entrada como un parámetro pasado como argumento por línea de comandos al ejecutar el programa y se ha usado en todos el "high_resolution_clock" para que no haya diferencia en los tiempos a la hora de llevar a cabo las ejecuciones y las mediciones.

La siguiente es la estructura de los algoritmos de ordenación básicos, representada en las imágenes por el algoritmo de burbuja. La única diferencia con otros algoritmos es la llamada a la función de ordenación. No olvidamos cambiar las bibliotecas incluidas y la especificación de "namespace" usada.

```cpp
#include <iostream>
#include <chrono>
#include <ctime>
#include <cstdlib>
#include <climits>
#include <cassert>

using namespace std;
using namespace std::chrono;
```

```cpp
int main(int argc, char ** argv)
{
   if (argc < 2){
      cerr << "Error: numero de parametros incorrectos" << endl;
      cerr << "USO: " << argv[0] << " <numero elementos>" << endl;
      exit(EXIT_FAILURE);
   }

  high_resolution_clock::time_point tantes, tdespues;
  duration<double> transcurrido;

  int n = atoi(argv[1]);
  int * T = new int[n];
  assert(T);

  srandom(time(0));

  for (int i = 0; i < n; i++)
    {
      T[i] = random();
    };

  tantes = high_resolution_clock::now();
  burbuja(T, n);
  tdespues = high_resolution_clock::now();
  transcurrido = duration_cast<duration<double>>(tdespues - tantes);
  cout << n << " " << transcurrido.count() << endl;

  delete [] T;

  return 0;
};
```

La estructura que se sigue es simple, obtener todos los datos a tratar, guardar el instante de tiempo antes de aplicar el algoritmo y una vez aplicado guardar el posterior. Finalmente se muestra por pantalla el tamaño al que se ha aplicado el algoritmo y la diferencia entre instantes que hace referencia al tiempo de trabajo o duración del algoritmo. Evidentemente esta estructura es la que se sugiere en el guion. Para otros algoritmos también se han producido cambios, conviene ver los algoritmos de Floyd y de Hanói han sido modificados con el mismo objetivo como se muestra a continuación.

```cpp
int main (int argc, char **argv)
{
  clock_t tantes;    // Valor del reloj antes de la ejecución
  clock_t tdespues;  // Valor del reloj después de la ejecución
  int dim;           // Dimensión de la matriz

  //Lectura de los parametros de entrada
  if (argc != 2){
    cerr << "Número de parámetros incorrecto" << endl;
    cerr << "USO: " << argv[0] << "num elementos" << endl;
    exit(EXIT_FAILURE);
  }

  dim = atoi(argv[1]);
  int ** M = ReservaMatriz(dim);

  RellenaMatriz(M,dim);


  // Empieza el algoritmo de floyd
  tantes = clock();
  Floyd(M,dim);
  tdespues = clock();
  cout << dim << " " << ((double)(tdespues-tantes))/CLOCKS_PER_SEC << endl;
  LiberaMatriz(M,dim);

  return 0;
}
```

```cpp
int main(int argc, char ** argv)
{
   if (argc < 2){
      cerr << "Error: numero de parametros incorrectos" << endl;
      cerr << "USO: " << argv[0] << " <numero elementos>" << endl;
      exit(EXIT_FAILURE);
   }

   high_resolution_clock::time_point tantes, tdespues;
   duration<double> transcurrido;

   int num_discos = atoi(argv[1]);

   tantes = high_resolution_clock::now();
   hanoi(num_discos, 1, 2);
   tdespues = high_resolution_clock::now();
   transcurrido = duration_cast<duration<double>>(tdespues - tantes);
   cout << num_discos << " " << transcurrido.count() << endl;

   return 0;
}
```

Cabe destacar que se ha usado un script propio en vez del script "mimacro" proporcionado en la documentación de la práctica. La estructura de este script es bastante simple, realiza un bucle incrementado la variable contadora que es pasada como argumento a la ejecución del programa, hay un bucle para cada programa y para cada serie de programas se actualizan las variables INICIO, FINAL E INCREMENTO, para que el tamaño de los datos sea el adecuado dependiendo del programa o conjunto de programas.

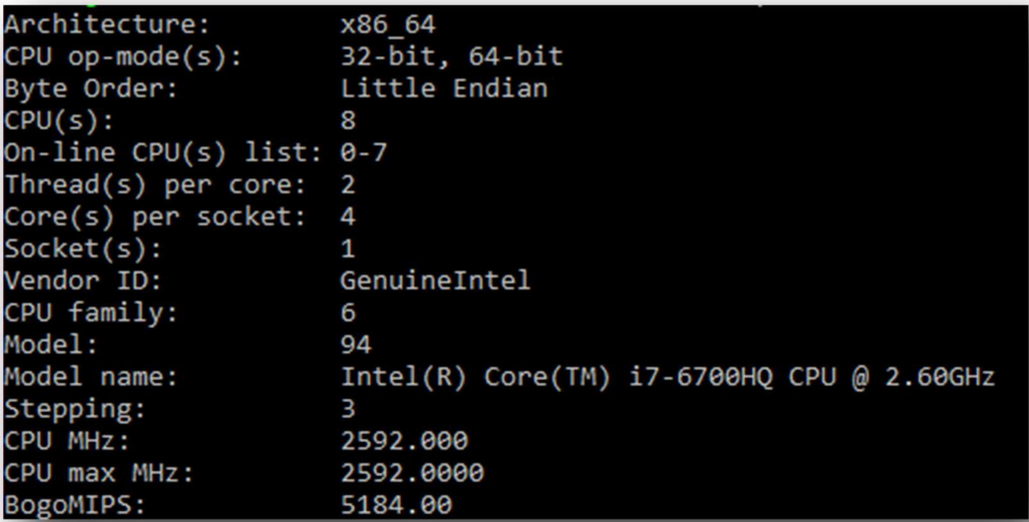
```bash
#!/bin/bash

#-------------------------------------------------#
# Para los algoritmos de ordenacion cuadraticos
INICIO=10000 # Tamanio inicial
FINAL=200000 # Tamanio final
INCRE=7600   # Tamanio de incremento
#-------------------------------------------------#

#-----------------------------------------------------------#
# EJECUCIÓN ALGORITMO DE ORDENACIÓN BURBUJA
rm res/burbuja.dat
for ((tam=$INICIO; tam<=$FINAL; tam+=$INCRE)) do
  ./burbuja $tam >> res/burbuja.dat
done
#-----------------------------------------------------------#
```

3. Mediciones empíricas.

Las especificaciones del equipo que ha llevado a cabo estas mediciones son las siguientes:



```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                8
On-line CPU(s) list:   0-7
Thread(s) per core:    2
Core(s) per socket:    4
Socket(s):             1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 94
Model name:            Intel(R) Core(TM) i7-6700HQ CPU @ 2.60GHz
Stepping:              3
CPU MHz:               2592.000
CPU max MHz:           2592.0000
BogoMIPS:              5184.00
```

Para mostrar todos los datos de manera más clara, usaremos una tabla. En la primera tabla podemos ver una comparación general de los tiempos de todos los algoritmos de ordenación, los algoritmos n*log(n) son claramente más eficientes De izquierda a derecha aparecen ordenados de menos a más eficientes.

Tamaños	Burbuja	Selección	Inserción	Mergesort	Heapsort	Quicksort
10000	0.255053	0.117256	0.0938404	0.00155668	0.00151286	0.0010819
17600	0.864844	0.359074	0.290766	0.002891	0.00273124	0.00209875
25200	1.8235	0.737175	0.599571	0.004963	0.00400816	0.00305498
32800	3.15053	1.24206	1.01231	0.005487	0.00539753	0.0042423
40400	4.83774	1.88588	1.5128	0.007352	0.00670754	0.00497331
48000	7.03347	2.66334	2.14481	0.009391	0.00810732	0.00609304
55600	9.56826	3.57109	2.88994	0.009507	0.00955758	0.00710712
63200	11.9939	4.61608	3.70183	0.011131	0.0110666	0.00814578
70800	15.1629	5.78523	4.6777	0.013099	0.0123721	0.00908153
78400	18.6327	7.09463	5.73385	0.014929	0.0137789	0.0102033
86000	22.5184	8.53547	6.94295	0.017229	0.0155898	0.011261
93600	26.7134	10.1179	8.2323	0.019244	0.017011	0.0127163
101200	31.315	11.8065	9.62902	0.021434	0.0184207	0.0136462
108800	38.3838	13.6546	11.16	0.019412	0.0200903	0.014416
116400	42.2404	15.6192	13.4192	0.021278	0.0217059	0.0155649
124000	48.3924	17.7472	15.8072	0.023359	0.0231133	0.0167578
131600	56.3024	19.9885	17.5316	0.024953	0.0245736	0.0178021
139200	60.4578	22.3559	19.7099	0.02679	0.0277268	0.0187858
146800	69.8402	24.8439	21.6786	0.028996	0.0281662	0.0203

154400	73.3324	27.5282	24.6134	0.030815	0.0294643	0.0211196
162000	83.1672	30.2532	26.6298	0.033063	0.0309397	0.0220494
169600	92.124	35.2392	27.8852	0.035161	0.0329503	0.0232497
177200	96.8594	36.2317	31.4581	0.037214	0.0344862	0.0248
184800	105.273	39.4081	35.0845	0.039439	0.0359864	0.0257361
192400	114.52	42.7032	34.6138	0.042133	0.0381247	0.0267367
200000	123.941	46.1832	37.6923	0.044077	0.0392406	0.0279626

La siguiente muestra una comparación de los algoritmos de ordenación más eficientes para una entrada bastante más grande. Podemos ver que heapsort para 200 millones de elementos tarda casi lo mismo que burbuja para 200 mil, un tamaño mil veces menor esto se debe a la diferencia de ordenes (O(n^2) con O(n*log(n))). Podemos observar que mergesort es prácticamente el doble de rápido que heapsort. Quicksort es el algoritmo más rápido (como su nombre indica).

Tamaño	Heapsort	Mergesort	Quicksort
10000	0.00161667	0.00158699	0.0017712
8009600	3.27196	1.9879	1.50504
16009200	7.18963	4.17131	3.26161
24008800	11.2261	6.84359	5.2607
32008400	15.5649	8.69923	6.90691
40008000	20.3385	11.381	8.62958
48007600	24.7774	14.3598	10.3923
56007200	29.8923	15.3611	12.5513
64006800	33.8078	18.1841	14.5707
72006400	38.8179	22.0983	16.206
80006000	44.0496	24.2928	18.4465
88005600	48.9127	26.7032	20.1092
96005200	53.9727	30.0474	22.1808
104004800	59.0224	33.9665	23.2099
112004400	64.6553	31.8874	25.0839
120004000	67.8531	34.508	27.477
128003600	72.916	41.8445	28.9501
136003200	78.2451	43.8871	31.6132
144002800	83.5939	46.235	33.5762
152002400	91.0787	46.7064	34.7801
160002000	96.7645	51.0482	38.6511
168001600	103.096	54.4572	39.9755
176001200	107.919	57.7655	43.089
184000800	114.762	61.5062	44.4947
192000400	119.793	64.9649	44.4324
200000000	125.749	65.2368	47.6911

Para los algoritmos de la última serie una comparación con los anteriores no tiene ningún sentido pues no llevan a cabo una tarea parecida y los tamaños son muy

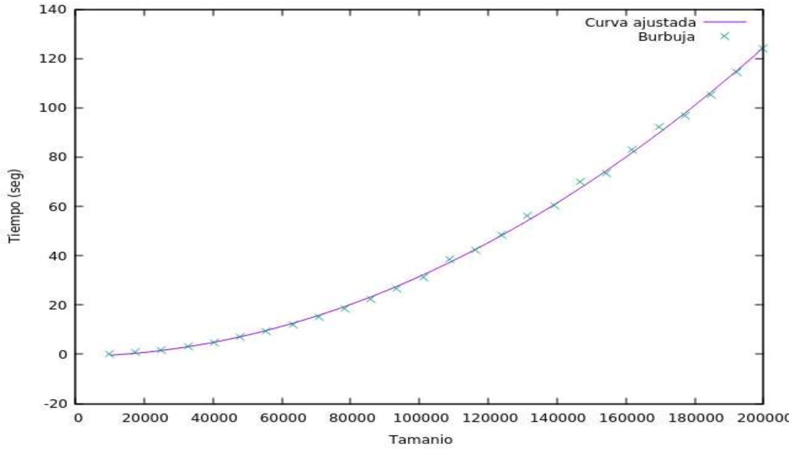
diferentes. Veamos una tabla de cada uno de estos algoritmos. Ambos algoritmos son bastante lentos y necesitamos reducir el tamaño. Podemos apreciar como el algoritmo de Floyd es bastante más lento que los de ordenación, pero infinitamente más rápido pues para 300 elementos Hanói tardaría miles de millones de años.

Tamaño	Floyd
300	0.139019
400	0.339344
500	0.649067
600	1.05678
700	1.98994
800	2.73719
900	3.8193
1000	5.36488
1100	6.64076
1200	9.08283
1300	10.9458
1400	13.5521
1500	16.7836
1600	20.8549
1700	24.3712
1800	29.501
1900	33.903
2000	40.6969
2100	48.2477
2200	57.174
2300	64.0366
2400	69.48
2500	80.1489
2600	88.8821
2700	102.33
2800	112.898

Tamaño	Hanói
10	0.0000081
11	0.0000135
12	0.0000255
13	0.0000483
14	0.0000978
15	0.0001873
16	0.0003733
17	0.0007873
18	0.0014877
19	0.0033287
20	0.00597016
21	0.0129736
22	0.0248079
23	0.0478088
24	0.0971039
25	0.188399
26	0.376515
27	0.775119
28	1.46569
29	2.89376
30	5.91028
31	11.7179
32	23.8177
33	47.3056
34	92.7409
35	187.881

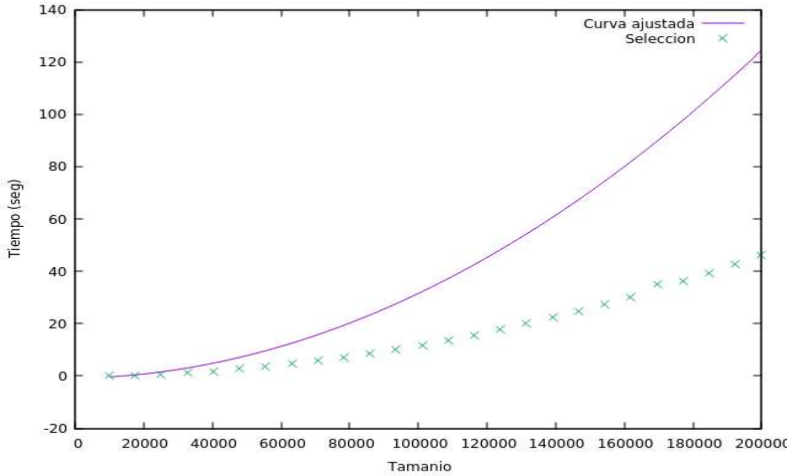
4. Gráficas y órdenes de eficiencia.

Veamos las representaciones gráficas de las ejecuciones de todos los algoritmos. Cada gráfica es la representación de los tiempos de ejecución con la curva de regresión. Los algoritmos de ordenación de orden cuadrático.



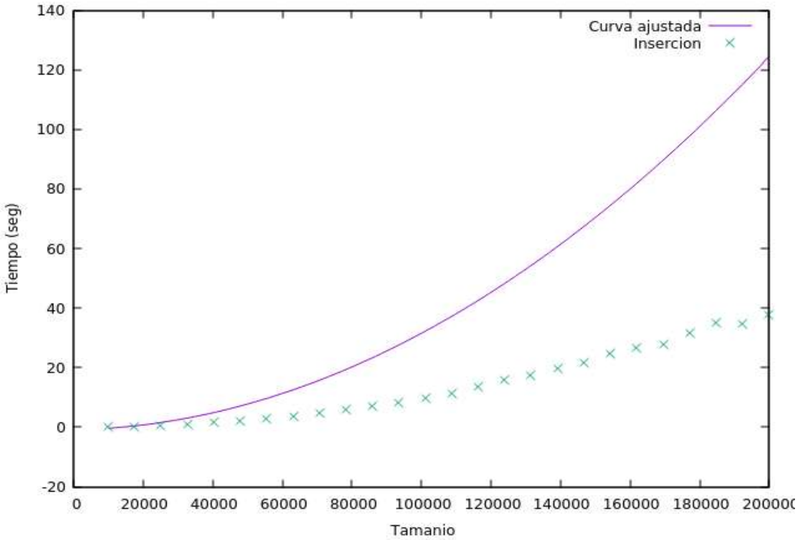
Burbuja: a0*x*x+a1*x+a2

El más lento de los cuadráticos



Selección: a0*x*x+a1*x+a2

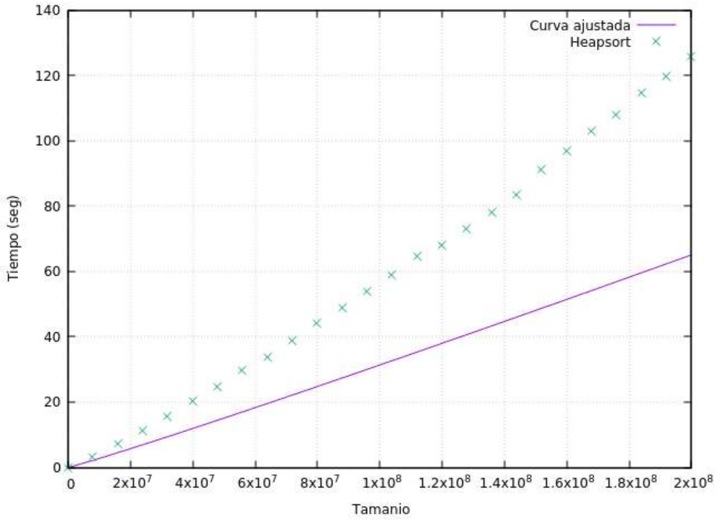
Mejor que burbuja, casi tan bueno como inserción.



Inserción: a0*x*x+a1*x+a2

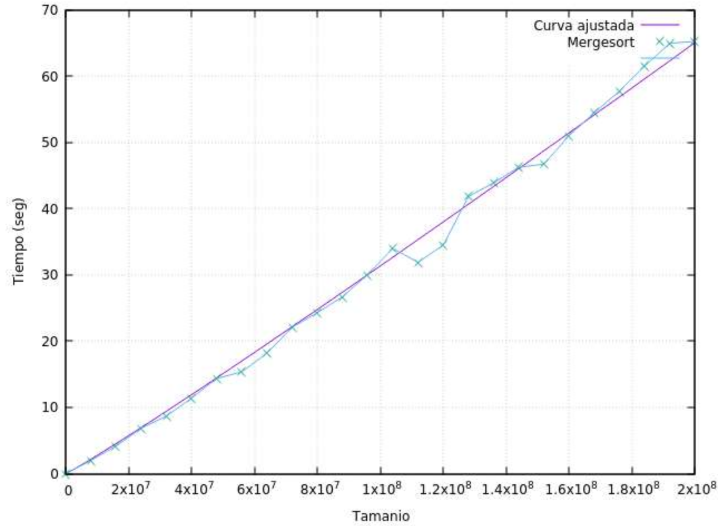
El mejor de los algoritmos de ordenación de orden cuadrático vistos en esta práctica.

Los algoritmos de ordenación de orden n*log(n).



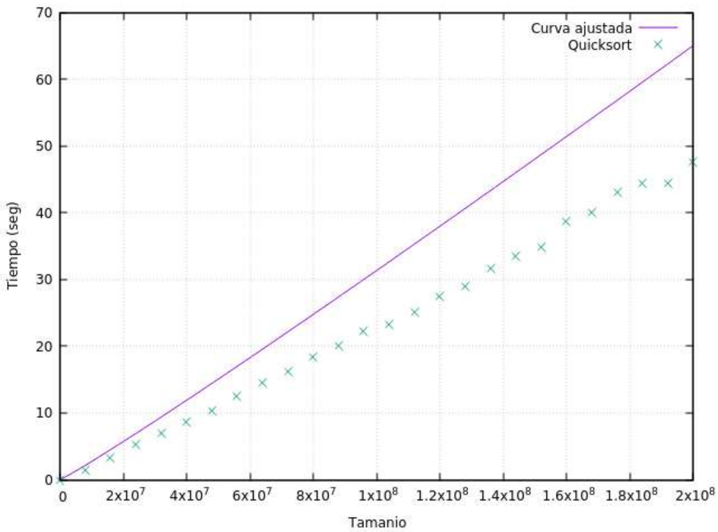
Heapsort: a0*n*log(n) + a1

Más ineficiente que el mergesort



Mergesort: a0*n*log(n) + a1

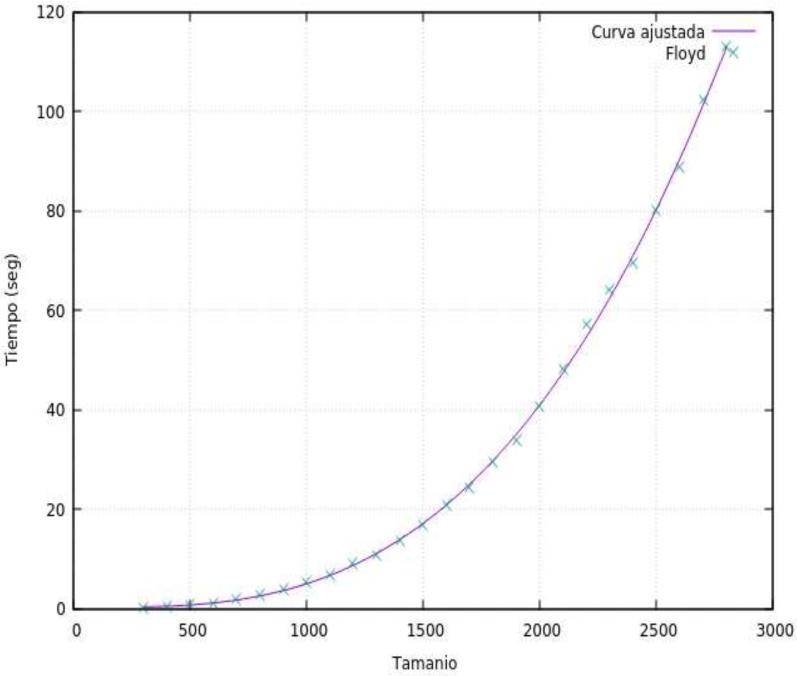
El mergesort va dando una especie de saltos cada x tamaños. Más ineficiente que el quicksort. Su consumo de memoria es elevado.



Heapsort: a0*n*log(n) + a1

El más rápido de los algoritmos de ordenación.

Los algoritmos de mayor orden de eficiencia.



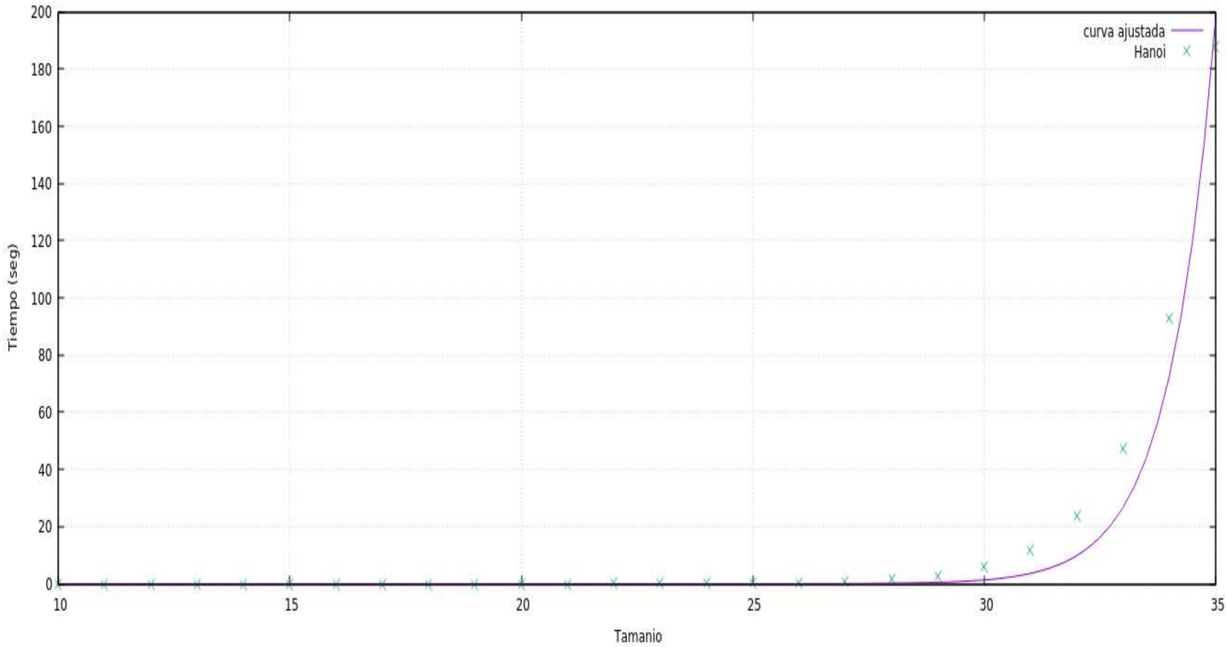
Floyd:

a0*x*x*x+a1*x*x+a2*x+a3.

Bastante más lento que los algoritmos de ordenación básicos.

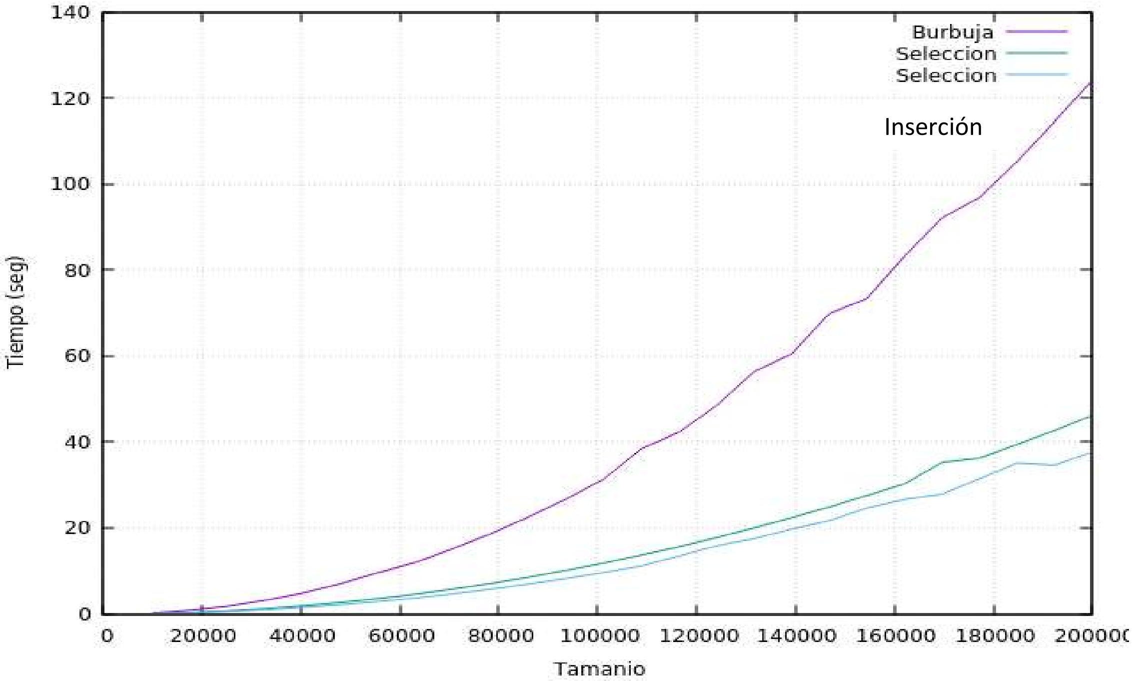
Hanói: a0*2*exp(x)

Algoritmo exponencial no polinomial, tremendamente lento para 1000 elementos tarda miles de millones de años. El más lento con diferencia.

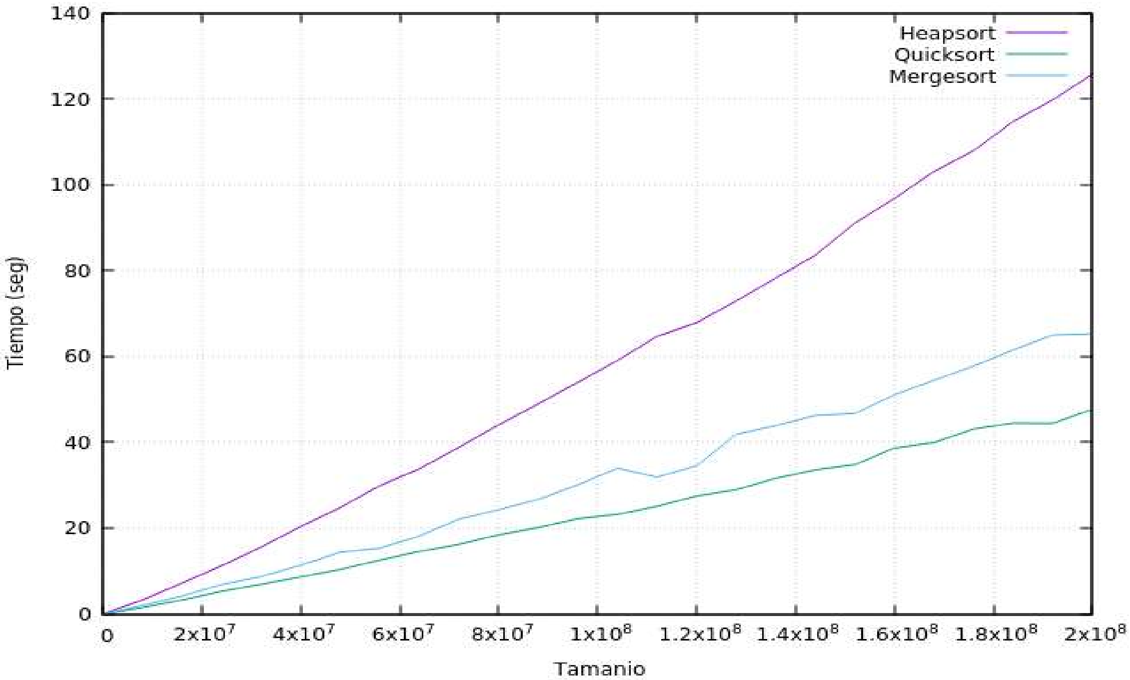


Vamos a comparar todos los algoritmos gráficamente para ver sus diferencias y semejanzas más claramente.

Podemos ver para las ordenaciones de órdenes cuadráticos que el más lento es el Burbuja y el más rápido es el de inserción, hasta 3 veces más rápido que burbuja.

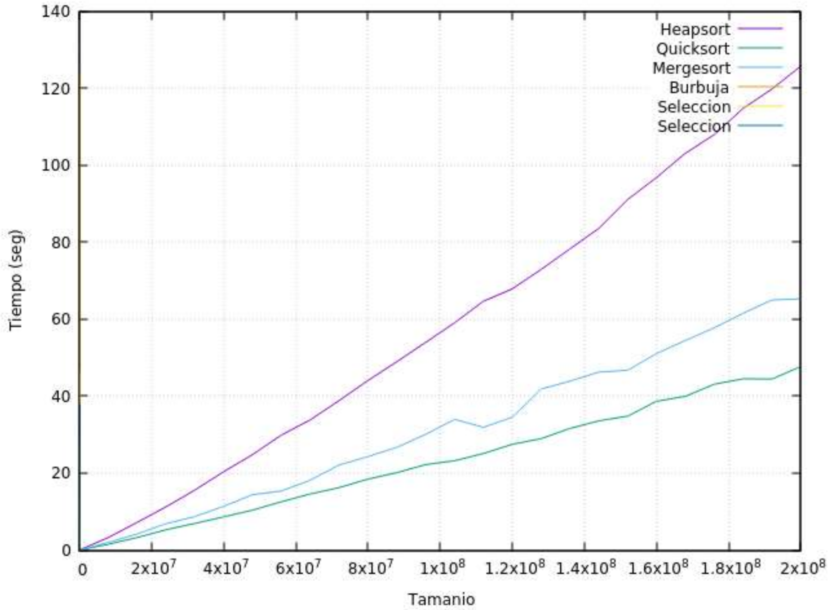


Para los algoritmos de ordenación más eficientes tenemos la siguiente comparación. Se observa que heapsort es el más lento y quicksort el más rápido como esperábamos, casi 3 veces más rápido que heapsort y 1/3 más rápido que mergesort el cual no es nada lento.

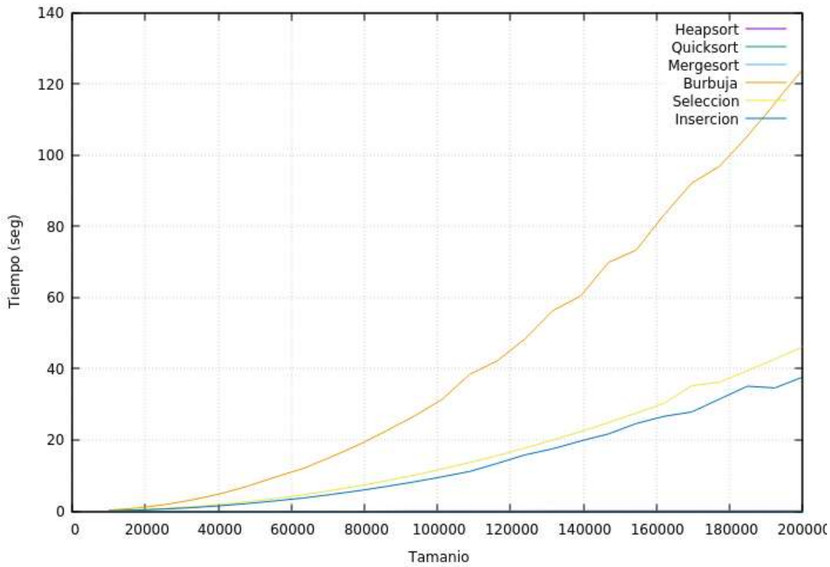


Comparemos ahora todos los algoritmos de ordenación, tenemos hechas 2 mediciones para los algoritmos más rápidos. Una con el mismo n que los básicos y otra con entradas 1000 veces más grandes.

NOTA: debido a la diferencia de tiempos algunas líneas no se aprecian.

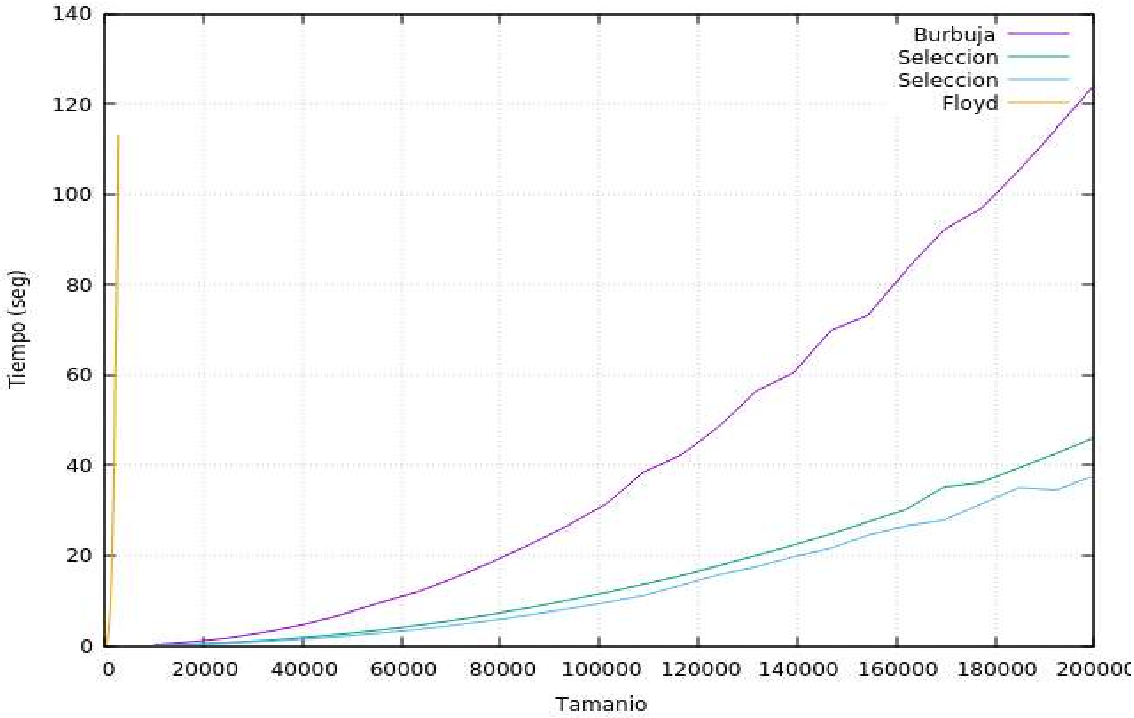


En esta podemos ver que en lo que los básicos ordenan 200 mil elementos los rápidos ordenan 200 millones.

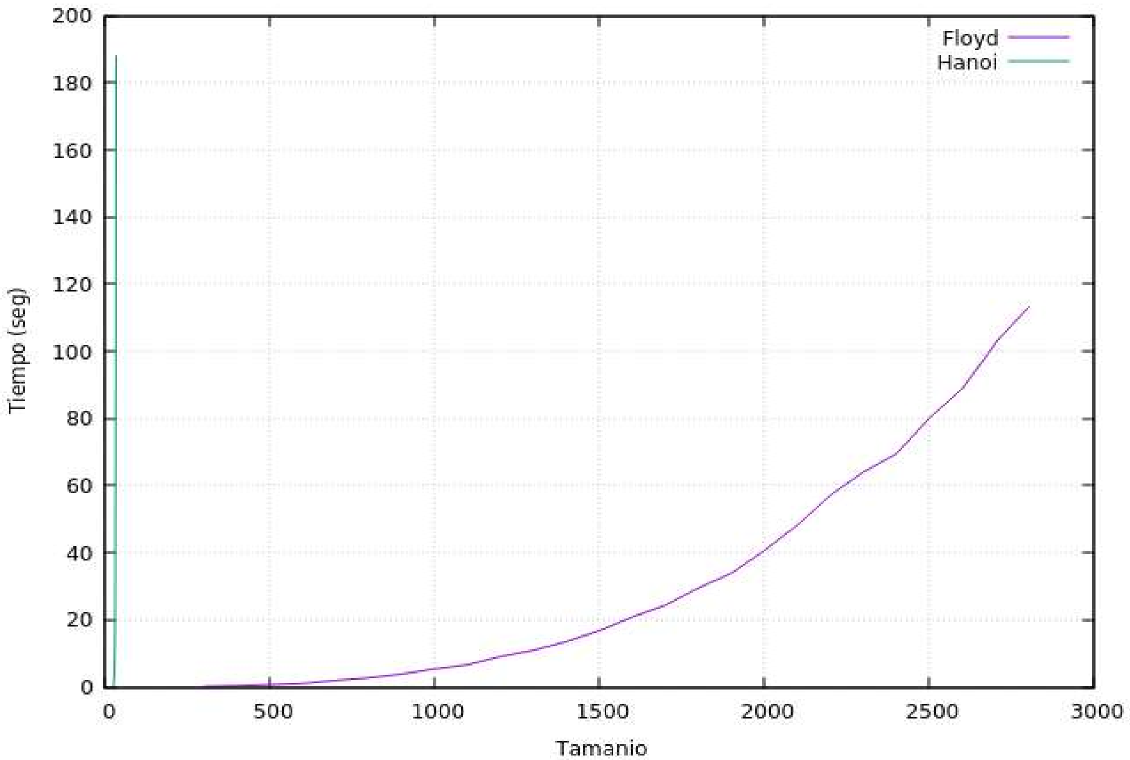


En esta otra comparativa vemos que literalmente no hay punto de comparación entre los rápidos y los lentos, pues los rápidos ni se aprecian.

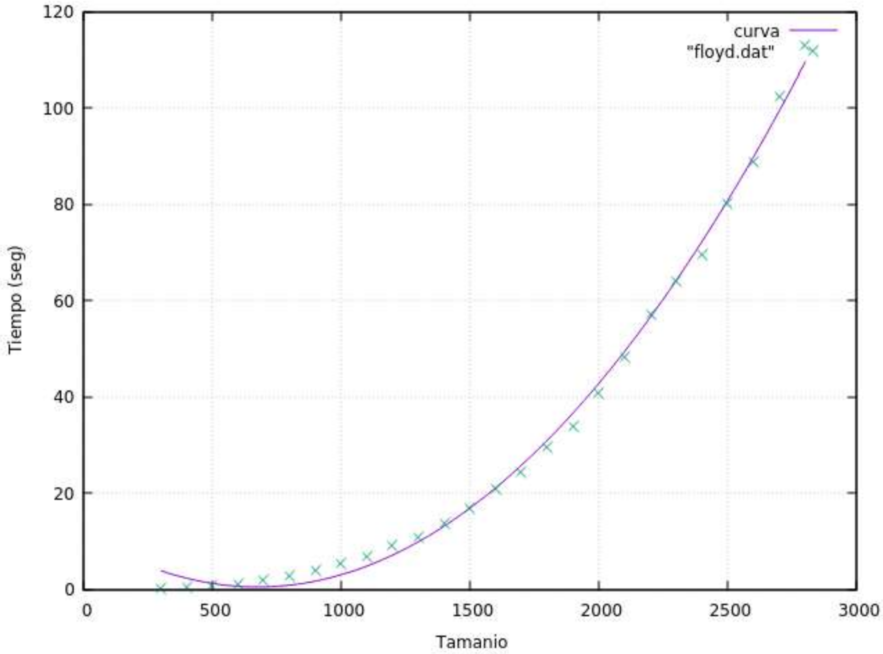
Comparemos ahora el algoritmo de Floyd con los de orden cuadrado. Podemos ver que dicho algoritmo es realmente lento en comparación a estos a pesar de ser polinomial. Para el tamaño mínimo de los cuadrados Floyd tardaría muchísimo tiempo, aunque es un tiempo finito.



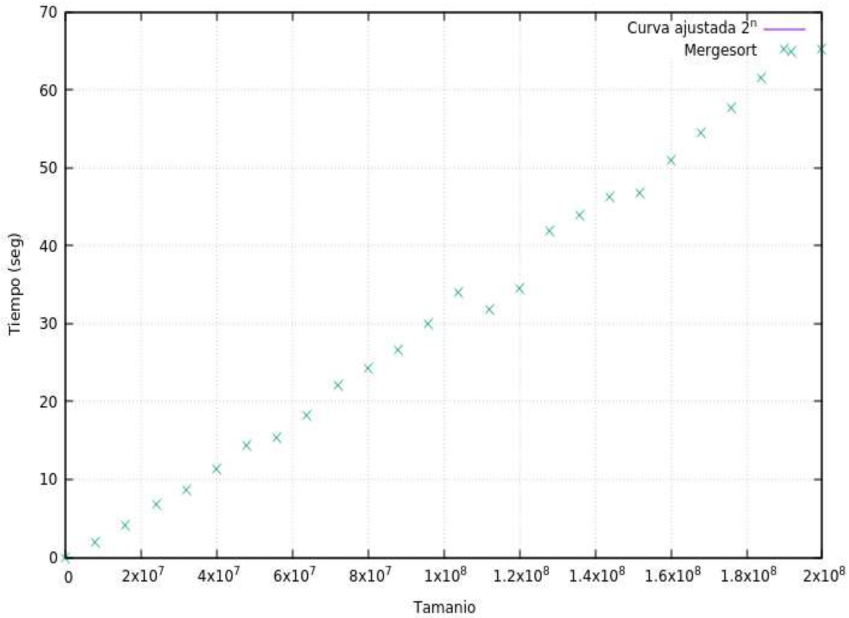
Al comparar Floyd con Hanói, nos damos cuenta de que este es una bendición pues Hanói es prácticamente una línea vertical debido a que es no polinomial.



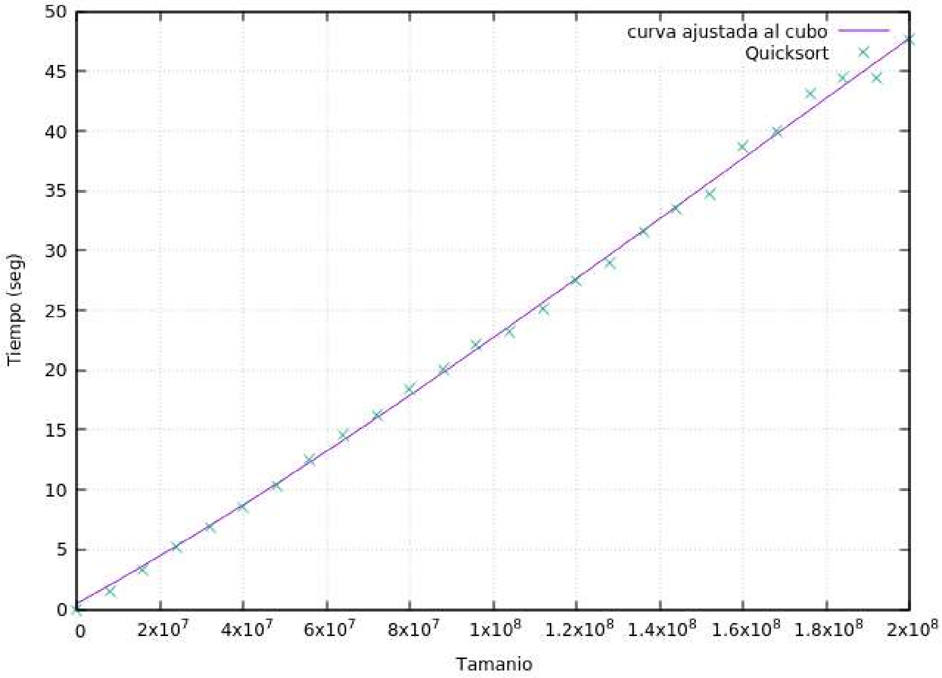
Probemos a calcular la eficiencia híbrida con ajustes diferentes:



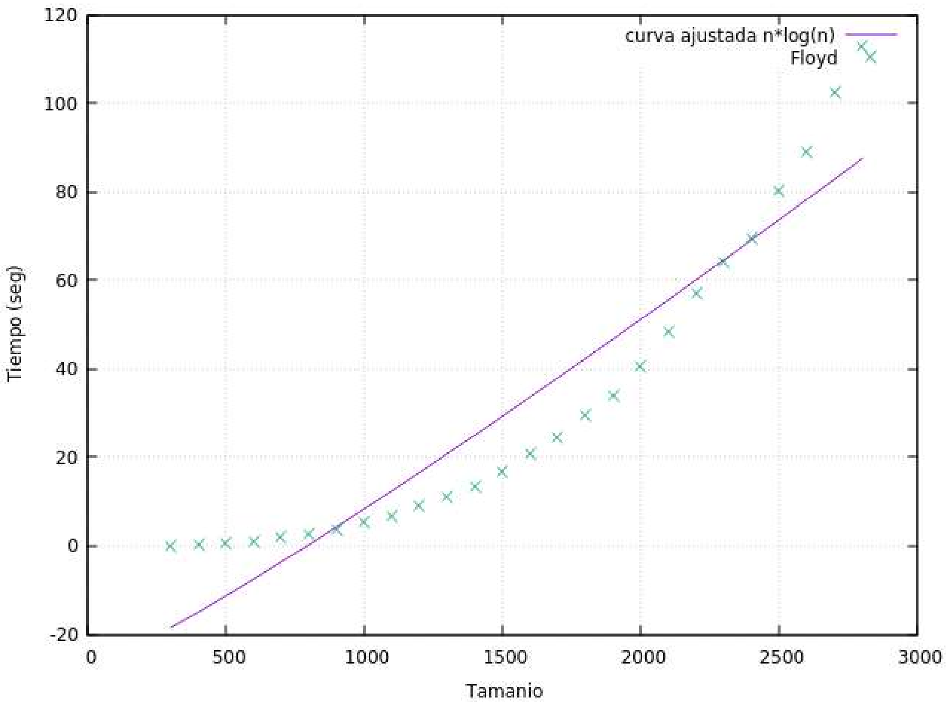
Floyd con un ajuste cuadrático. Apenas hay diferencias.



Mergesort con ajuste 2^n, no hay punto de comparación son ordenes muy dispares.



Quicksort con ajuste cúbico, órdenes muy dispares, pero perfectamente ajustables.



Floyd ajustado a n*log(n), hay punto de comparación, pero no parecen estar muy ajustados.

5. Comparaciones entre optimizar (o no)

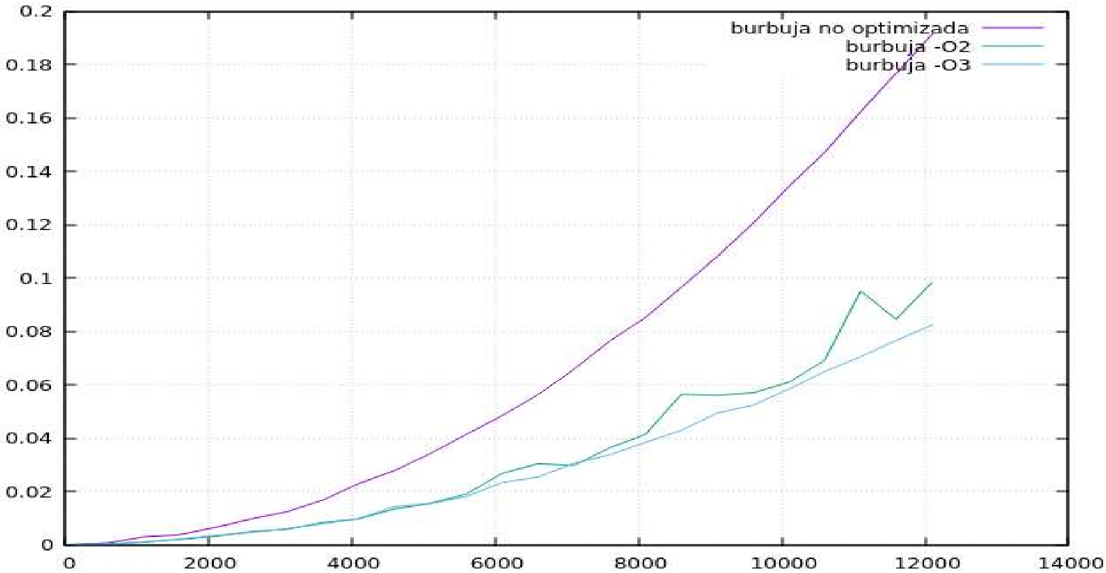
Compararemos los algoritmos en base a las opciones usadas para la compilación de los programas que los implementaban. Se han usado las opciones –O2 y –O3, con 10 ejecuciones por tamaño, estos son los resultados:

BURBUJA

Tamaño	Burbuja	Burbuja –O2	Burbuja –O3
100	0,0000559	0,0000473	0,0000087
600	0,0006684	0,0003325	0,0002791
1100	0,0029876	0,0009438	0,0010236
1600	0,0037486	0,0020146	0,0021509
2100	0,0065223	0,0031525	0,0035466
2600	0,0096896	0,0049387	0,0046334
3100	0,0124138	0,0057985	0,0060024
3600	0,0167894	0,0083532	0,0078898
4100	0,0229371	0,0096899	0,0098328
4600	0,0278419	0,0134595	0,0143341
5100	0,0342268	0,0155387	0,0153186
5600	0,0414072	0,0190226	0,0181444
6100	0,0484069	0,0267483	0,0232942
6600	0,0562251	0,0303629	0,025407
7100	0,0657105	0,0297015	0,0305231
7600	0,0763762	0,0363119	0,0337031
8100	0,0852099	0,0413743	0,0383944
8600	0,0964816	0,0563764	0,0428817
9100	0,1080832	0,0560221	0,0493888
9600	0,1205084	0,0569847	0,0523568
10100	0,134272	0,0608712	0,0583358
10600	0,1470804	0,0691447	0,0648991
11100	0,1622753	0,095087	0,0704927
11600	0,1768598	0,0843858	0,0766336
12100	0,1916143	0,0981963	0,0824231

Podemos observar que de la opción normal sin optimización a la versión O2 hay de media un 195% de mejora aproximadamente, es decir, es el doble de rápido esta implementación del algoritmo a nivel máquina, lo cual es una ganancia considerable, sin embargo, es mucho más considerable la ganancia con -03 que es de un 232% aproximadamente, esto es más del doble y para unos tamaños considerables nos da una ganancia considerable.

Veamos la representación gráfica de estos datos.

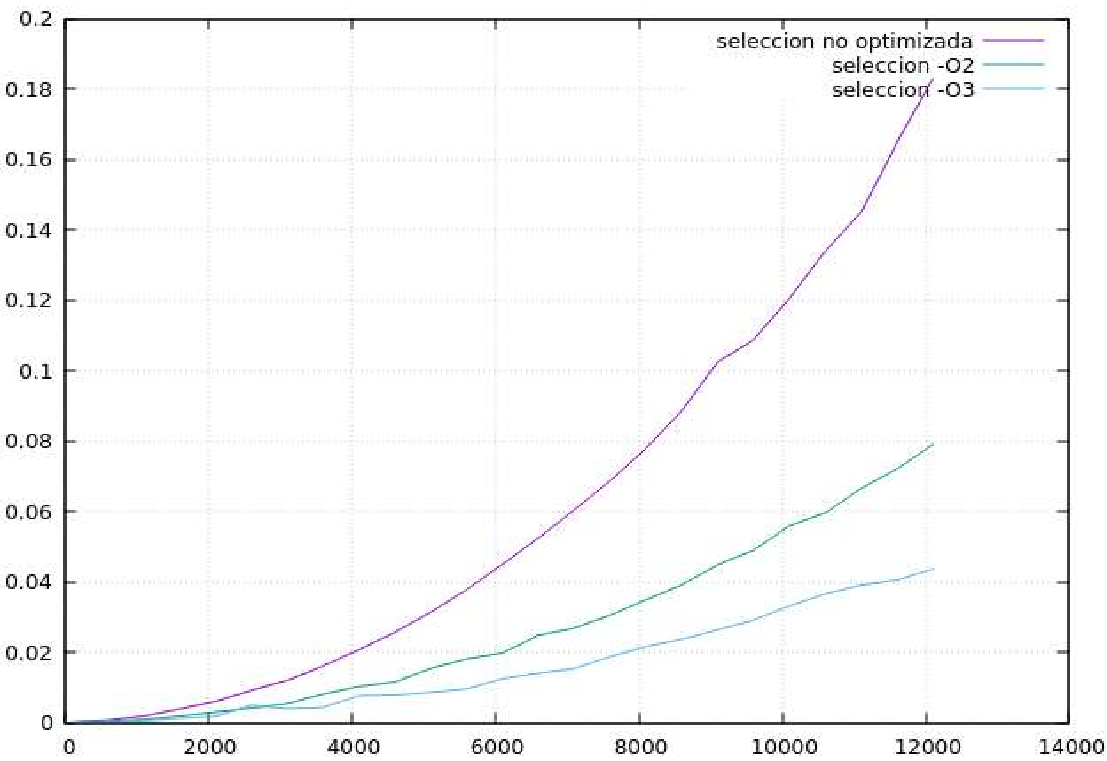


SELECCIÓN

Tamaño	Selección	Selección –O2	Selección –O3
100	0,0000596	0,0000261	0,0000064
600	0,0005887	0,0003193	0,0001512
1100	0,0018454	0,0008208	0,0005192
1600	0,0038098	0,0017711	0,0011496
2100	0,0059764	0,0029407	0,0017492
2600	0,0091048	0,004062	0,0049395
3100	0,0119054	0,0053513	0,0039308
3600	0,0160292	0,0080671	0,0042811
4100	0,0206625	0,0102315	0,0075745
4600	0,0256193	0,011445	0,007808
5100	0,031304	0,0153622	0,0085409
5600	0,037732	0,0180298	0,0095306
6100	0,0449446	0,0197088	0,0124278
6600	0,0523792	0,024751	0,0140282
7100	0,0603189	0,0268011	0,0153027
7600	0,0686638	0,0304405	0,0186496
8100	0,07782	0,0348375	0,0215654
8600	0,0885429	0,0390699	0,0236186
9100	0,1024313	0,0448014	0,026356
9600	0,1086669	0,0489325	0,0290662
10100	0,1204882	0,0558592	0,0331244
10600	0,1338082	0,0595169	0,0365306
11100	0,1449901	0,0665083	0,0390315
11600	0,1646252	0,0720068	0,0404008
12100	0,1826251	0,0789297	0,0436513

Observamos ahora que con la versión –O2 obtenemos una ganancia del 231% aproximadamente, más del doble de eficiente que la versión normal y el caso realmente extremo es la versión con –O3 es del 418% aproximadamente, lo que significa un tiempo de cálculo 4 veces menor que la versión normal.

Veamos la representación gráfica de estos datos.

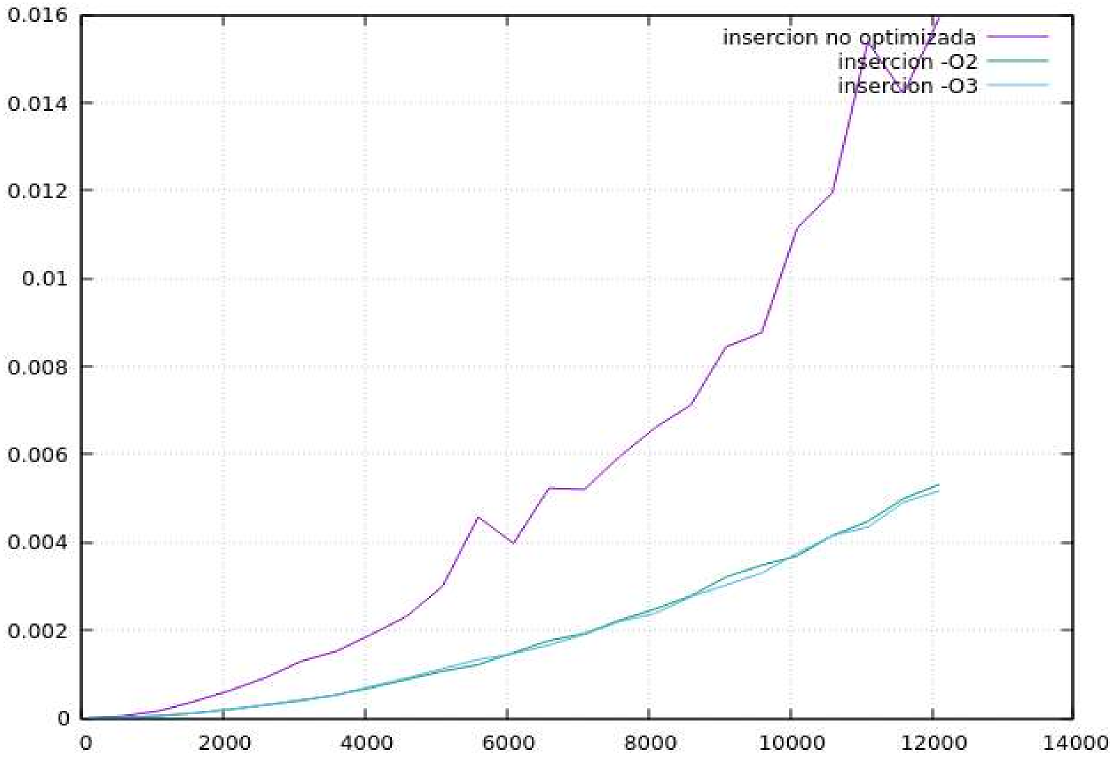


INSERCIÓN

Tamaño	Inserción	Inserción –O2	Inserción –O3
100	0,0000022	0,000004	0,0000057
600	0,0000499	0,0000221	0,0000178
1100	0,0001707	0,0000543	0,0000615
1600	0,0003872	0,0001162	0,0001157
2100	0,0006334	0,0002049	0,0001967
2600	0,0009217	0,0003006	0,0003077
3100	0,0012945	0,0004021	0,0004189
3600	0,0015269	0,0005335	0,0005273
4100	0,0019127	0,0006935	0,0007297
4600	0,0023292	0,0008849	0,0009202
5100	0,003011	0,0010703	0,001131
5600	0,004577	0,0012172	0,0013353
6100	0,0039784	0,0014949	0,0014711
6600	0,0052352	0,0017665	0,0016625
7100	0,0052042	0,0019275	0,0019077
7600	0,0059611	0,0022333	0,002204
8100	0,0066155	0,002488	0,0023868
8600	0,0071218	0,0027844	0,0027647
9100	0,0084494	0,0032117	0,0030325
9600	0,0087665	0,0034847	0,0032985
10100	0,011147	0,0036902	0,0037522
10600	0,0119517	0,0041575	0,004153
11100	0,0153873	0,0044782	0,0043484
11600	0,0142162	0,0049906	0,0049126
12100	0,0159281	0,0053133	0,005169

La versión –O2 obtiene una ganancia del 299% aproximadamente, es decir, para el mismo tamaño es 3 veces más rápido, para la versión –O3 también hay una gran mejora, pero solo con respecto a la versión base, esta ganancia es del 308% aproximadamente.

Veamos los datos representados en una gráfica para entenderlos mejor.

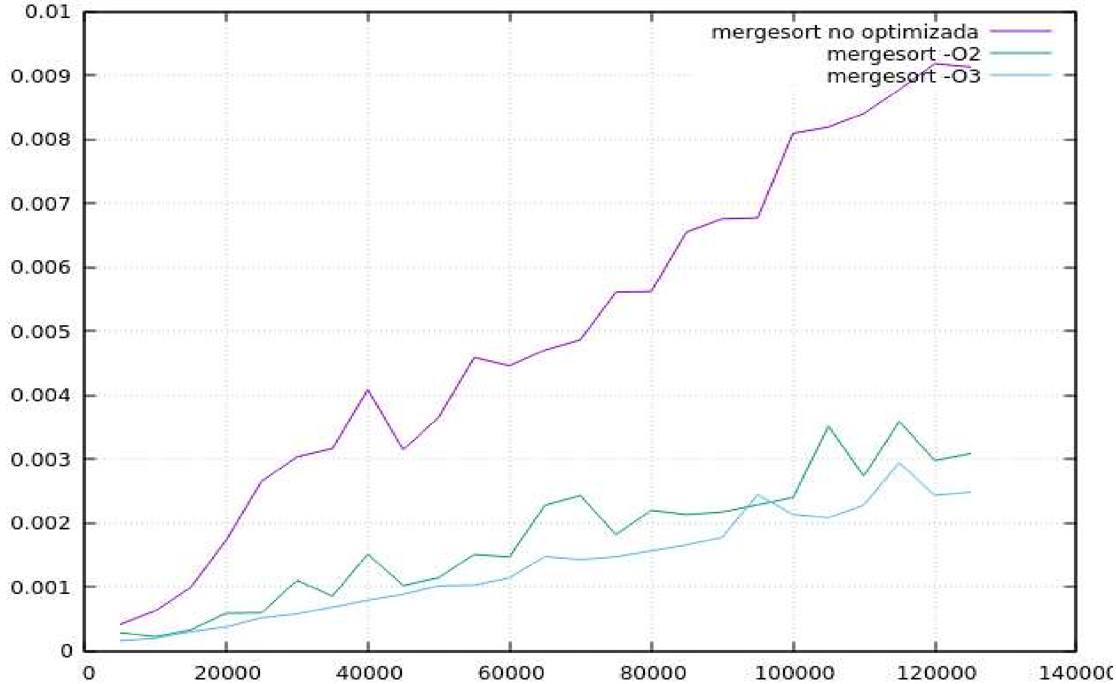


MERGESORT

Tamaño	Mergesort	Mergesort–O2	Mergesort –O3
5000	0,0004091	0,0002825	0,0001512
10000	0,0006235	0,000215	0,0001897
15000	0,000985	0,0003262	0,0002987
20000	0,0017243	0,0005842	0,0003716
25000	0,0026482	0,00059	0,0005136
30000	0,0030308	0,001098	0,0005749
35000	0,0031591	0,0008509	0,0006803
40000	0,0040864	0,0015085	0,0007876
45000	0,0031405	0,0010191	0,0008803
50000	0,0036518	0,0011425	0,0010148
55000	0,004584	0,0015045	0,0010261
60000	0,004457	0,0014676	0,0011384
65000	0,0047025	0,0022745	0,0014719
70000	0,0048602	0,0024243	0,001426
75000	0,0056086	0,0018164	0,0014673
80000	0,0056172	0,0021936	0,0015618
85000	0,0065508	0,002129	0,0016529
90000	0,0067528	0,0021694	0,0017739
95000	0,0067677	0,0022816	0,0024365
100000	0,0080944	0,0023944	0,0021322
105000	0,0081902	0,0035119	0,0020785
110000	0,0084037	0,0027291	0,0022758
115000	0,0087744	0,003588	0,0029418
120000	0,0091847	0,002973	0,0024285
125000	0,0091323	0,0030821	0,0024784

En este caso tenemos que la versión con O2 mejora aproximadamente un 296% casi 3 veces más rápido. La versión con O3 mejora hasta un 368% aproximadamente, estás mejoras son bastantes considerables, sobre todo para un algoritmo que de base ya era rápido.

La representación gráfica es la siguiente.

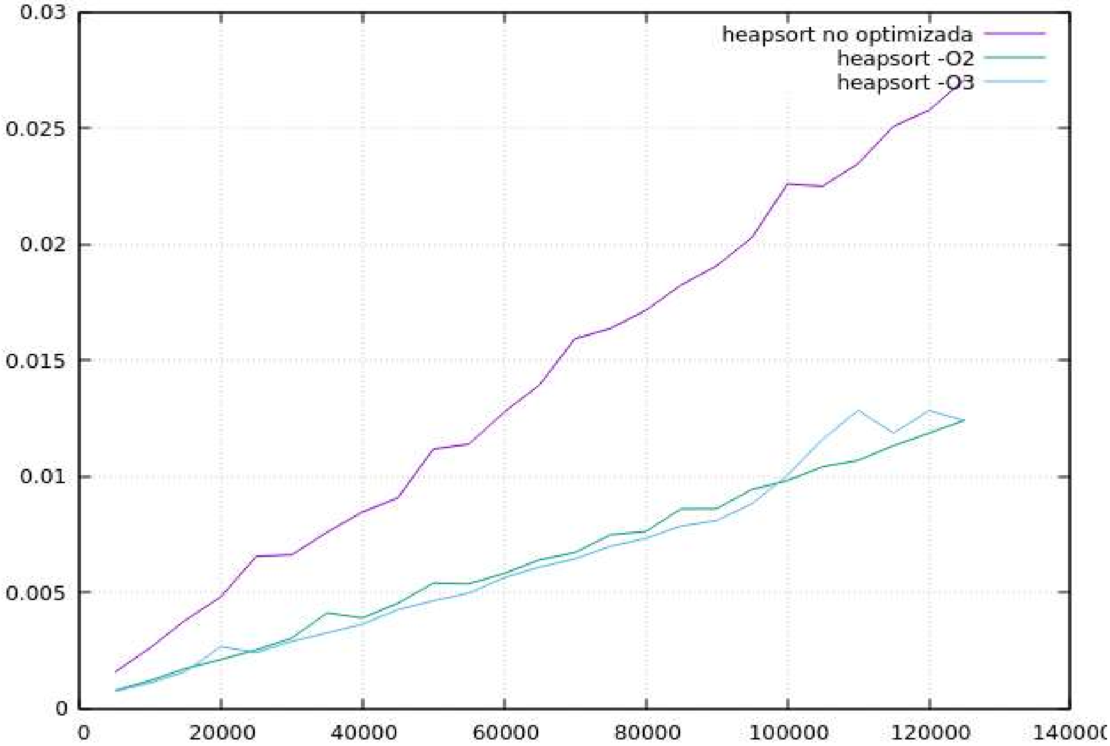


HEAPSORT

Tamaño	Heapsort	Heapsort –O2	Heapsort –O3
5000	0,0015455	0,0007614	0,00073
10000	0,0026069	0,0011956	0,0010908
15000	0,003792	0,0017137	0,0015811
20000	0,0048037	0,0020992	0,0026586
25000	0,0065436	0,0025166	0,0023974
30000	0,0066102	0,0030191	0,0028837
35000	0,0075866	0,0041055	0,0032484
40000	0,0084537	0,0038909	0,0036249
45000	0,009065	0,0045249	0,0042462
50000	0,0111661	0,0053964	0,0046301
55000	0,0113767	0,0053599	0,0049585
60000	0,0127524	0,0058061	0,0056297
65000	0,0139207	0,0063951	0,0060739
70000	0,0159221	0,0067052	0,0064385
75000	0,0163652	0,007469	0,0069745
80000	0,017139	0,0076174	0,0073222
85000	0,0182344	0,0085869	0,0078478
90000	0,0190584	0,0085935	0,0080832
95000	0,0202712	0,0094247	0,0088064
100000	0,0225849	0,009811	0,0100324
105000	0,0224915	0,0104005	0,0115915
110000	0,0234605	0,01068	0,0128384
115000	0,0250744	0,0113101	0,0118565
120000	0,0257634	0,0118528	0,0128237
125000	0,0270761	0,01241	0,0124041

Observamos una mejora de aproximadamente un 218% de O2 con respecto a la versión no optimizada, es decir, es más del doble de rápida. La versión O3 mejora en aproximadamente un 219% apenas hay diferencias con la versión O2.

Vemos la representación de los datos de manera gráfica.

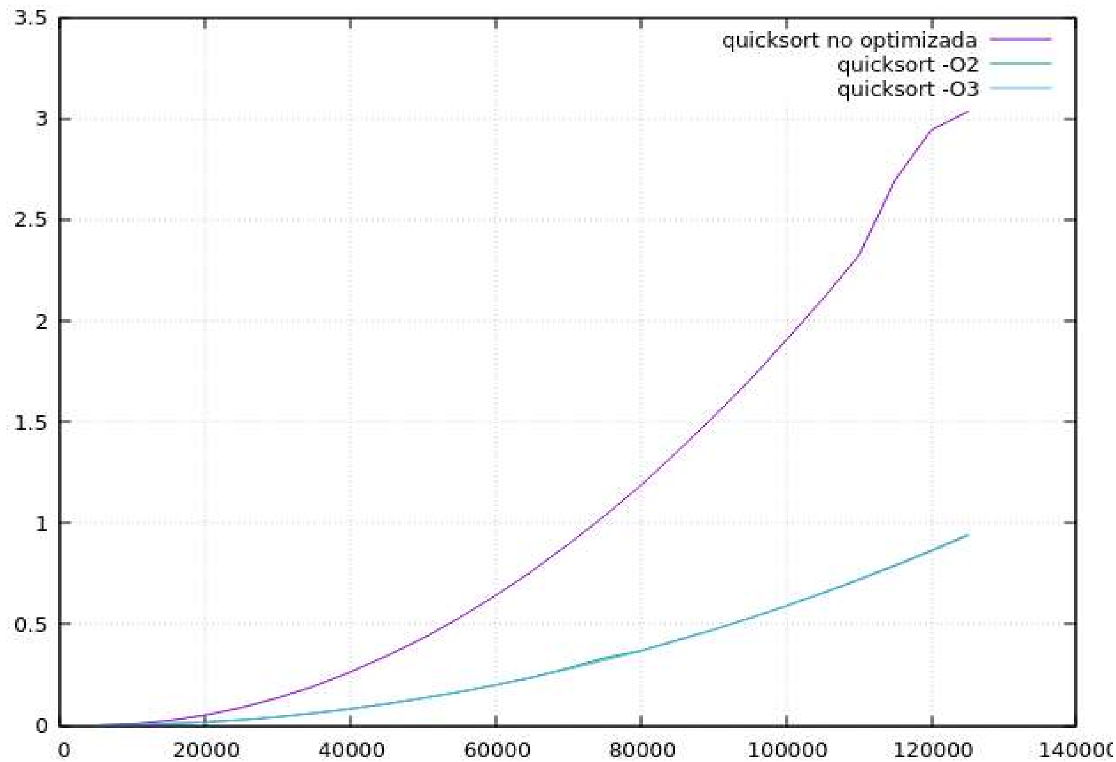


QUICKSORT

Tamaño	Quicksort	Quicksort –O2	Quicksort –O3
5000	0,0000375	0,0000136	0,0000041
10000	0,0071796	0,0025947	0,0024271
15000	0,0230884	0,0087712	0,0085491
20000	0,0504205	0,0169307	0,0168552
25000	0,0876771	0,0270184	0,0269169
30000	0,1359541	0,0420323	0,0415954
35000	0,1937896	0,0603196	0,0610687
40000	0,2630498	0,0821618	0,0813643
45000	0,3428754	0,1066589	0,1057963
50000	0,4305599	0,134994	0,1346822
55000	0,5309697	0,1654226	0,1656663
60000	0,6415025	0,2001093	0,1999097
65000	0,7614604	0,2374104	0,2374016
70000	0,8939114	0,2825545	0,2776458
75000	1,035278	0,333503	0,3219021
80000	1,1864927	0,3687767	0,3680694
85000	1,3514202	0,4206121	0,4181745
90000	1,5239524	0,4734026	0,4722091
95000	1,7055482	0,5302335	0,5293425
100000	1,9030063	0,5903492	0,5896261
105000	2,1055678	0,6539008	0,6522473
110000	2,3193433	0,7211567	0,7177817
115000	2,693555	0,7912608	0,7871012
120000	2,9426929	0,8646549	0,8600117
125000	3,0342886	0,9421234	0,936764

Se observa una mejora tremenda de O2 con respecto al programa sin optimizar, una mejora del 322% aproximadamente. Para la versión O3 la ganancia es de 323%, es decir, prácticamente no hay diferencia entre la versión O2 y la O3, pero ambas son muy buenas para este el algoritmo más rápido de los vistos en la práctica.

Veamos la representación gráfica de los datos ya mencionados.

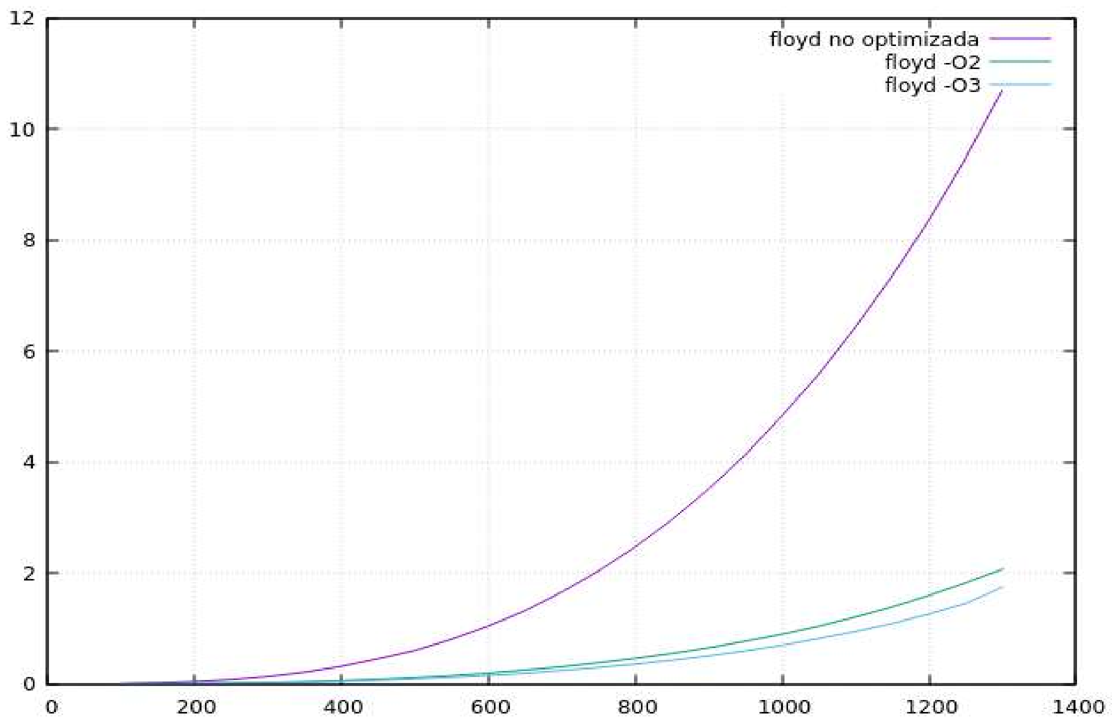


FLOYD

Tamaño	Floyd	Floyd –O2	Floyd –O3
100	0,0063228	0,0013452	0,0010345
150	0,0184564	0,0046397	0,0031549
200	0,0402106	0,01017	0,0066429
250	0,0768453	0,0144199	0,0116327
300	0,1315863	0,0246602	0,0190058
350	0,2102942	0,0384987	0,0302125
400	0,3112786	0,0586704	0,0451684
450	0,4503154	0,0817863	0,0639626
500	0,5998839	0,1129877	0,0873288
550	0,7994896	0,1503036	0,1162614
600	1,036431	0,1933897	0,1506564
650	1,316488	0,2474946	0,1916996
700	1,648679	0,3070108	0,2387066
750	2,026959	0,37768	0,2930467
800	2,464448	0,4574571	0,3524093
850	2,954062	0,5455299	0,4220259
900	3,50323	0,6477027	0,5029184
950	4,121488	0,7660497	0,5941524
1000	4,831181	0,8917415	0,6929853
1050	5,575247	1,036819	0,8154068
1100	6,428017	1,203815	0,9406228
1150	7,352457	1,384789	1,088146
1200	8,357931	1,592772	1,258927
1250	9,465997	1,819313	1,449779
1300	10,68651	2,064269	1,741153

Para este algoritmo tan ineficiente observamos una ganancia realmente importante. La versión O2 mejora en aproximadamente en un 517% y la O3 en 613%, es decir, más de 5 y 6 veces más rápido. Esta ganancia es realmente considerable pero no basta para hacer frente a la ineficiencia de este algoritmo cúbico.

Veamos la representación gráfica.

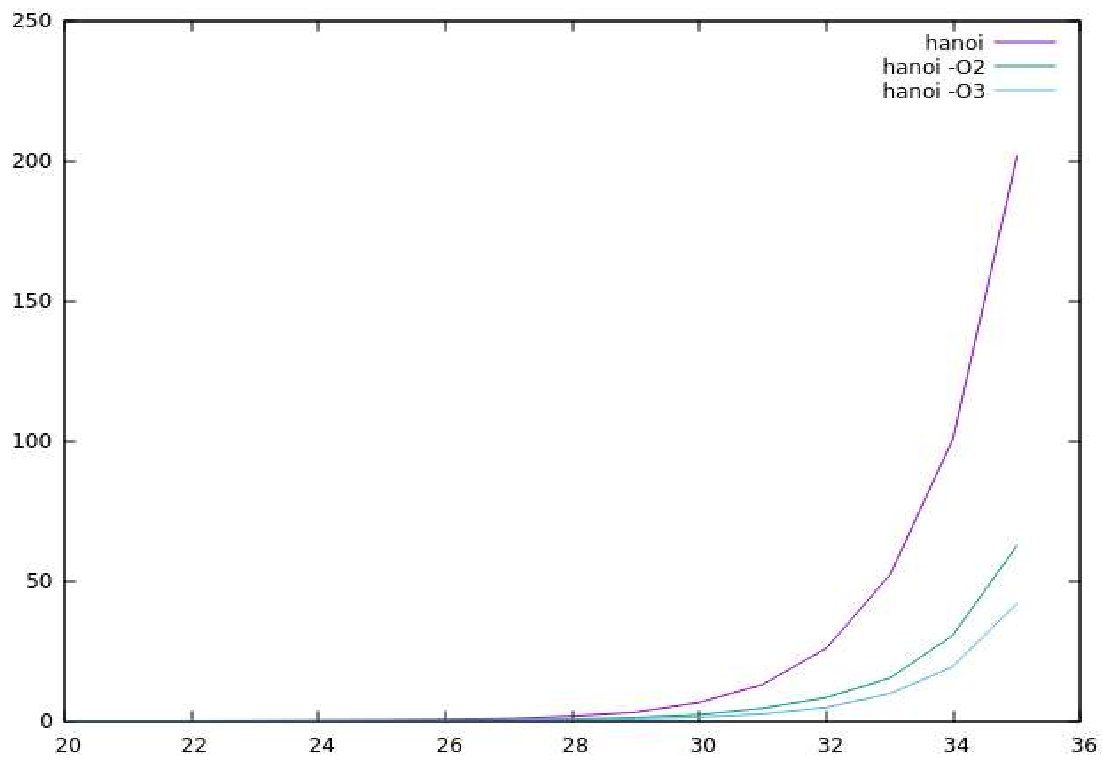


HANOI

Tamaño	Hanói	Hanói –O2	Hanói –O3
20	0.00571676	0.00212709	0.00133282
21	0.0127643	0.00483815	0.00291074
22	0.0243757	0.00936966	0.00570213
23	0.0459817	0.0204941	0.0108632
24	0.0904333	0.0337595	0.02027
25	0.181212	0.0732108	0.040555
26	0.367647	0.14608	0.0780421
27	0.730343	0.278897	0.15842
28	1.61666	0.496791	0.300499
29	3.01548	1.12118	0.605391
30	6.5334	2.08285	1.19072
31	12.9954	4.41869	2.38058
32	25.9695	8.33032	4.72635
33	52.1456	15.2898	9.74744
34	101.151	30.6212	19.3526
35	201.444	62.3394	41.6967

Para este algoritmo la mejora es bastante interesante, la versión O2 mejora un 323% aproximadamente mientras que la versión O3 mejora un 483%, lo cual implica que estas versiones son hasta 3 y 4 veces más rápidas.

Veamos la representación gráfica de estos datos para verlos más claramente.

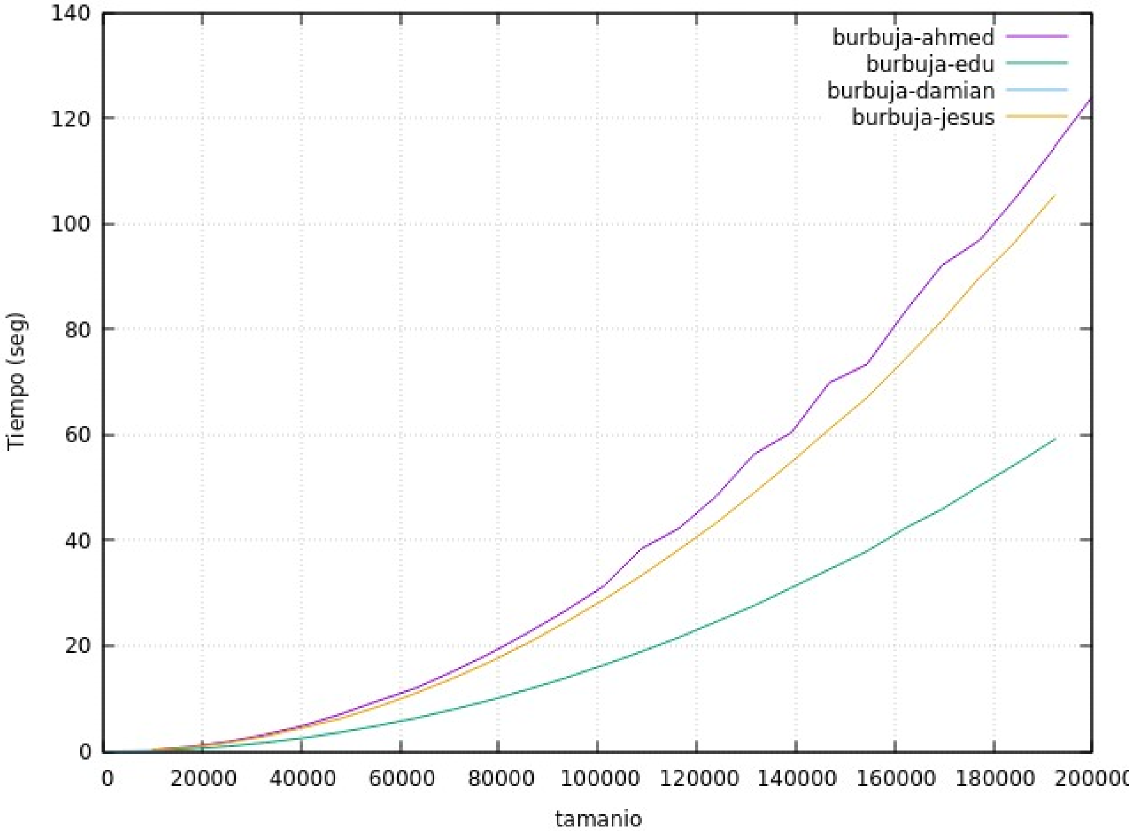


6.Comparación entre hardware.

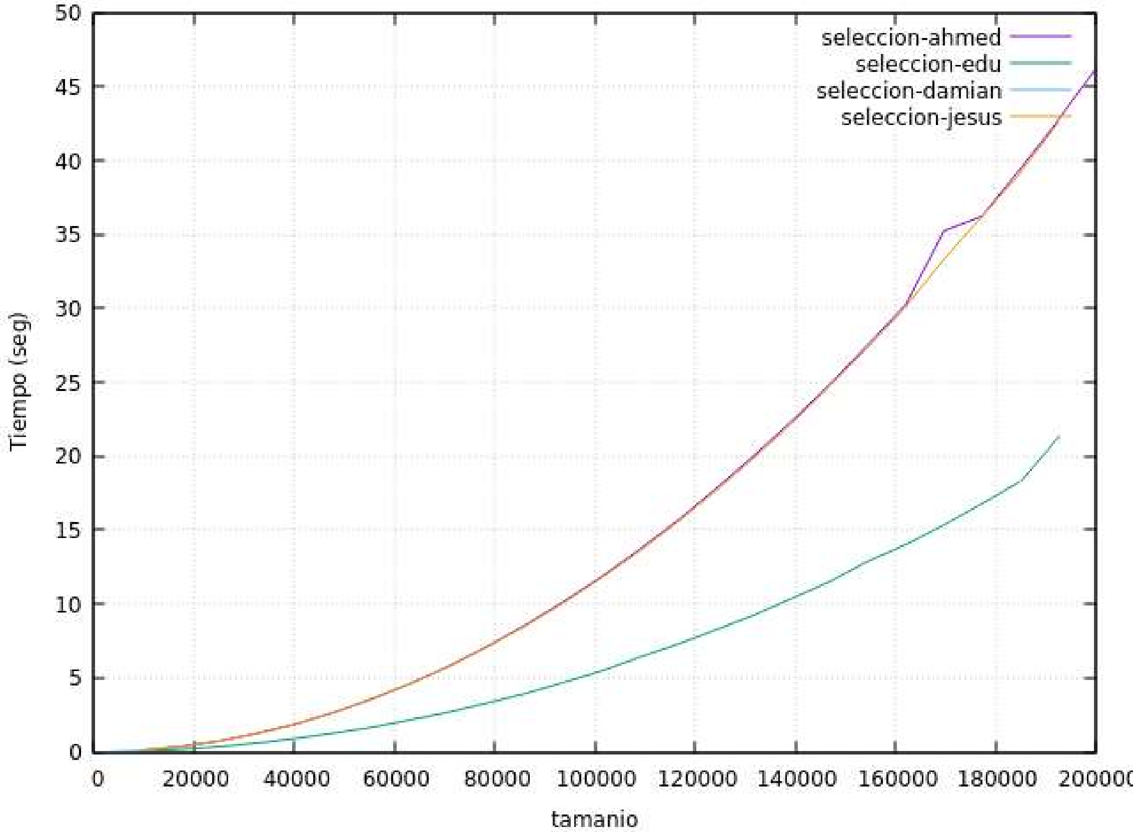
En este punto del guion vamos a ver qué ocurre y que resultados obtenemos al ejecutar estos algoritmos en diferentes equipos. En este caso los equipos serán 4, uno correspondiente a cada compañero, por eso y a partir de ahora cada equipo será identificado por el nombre de su propietario, así pues, tenemos los siguientes:

- Ahmed → I7 6700HQ 3.2 Ghz | 16 GB RAM
- Damián → I7 6700U 3.2 Ghz | 8 GB RAM
- Eduardo → I7 6500U 2.5 Ghz | 12 GB RAM
- Jesús → I7 7500U 2.6 Ghz | 8 GB RAM

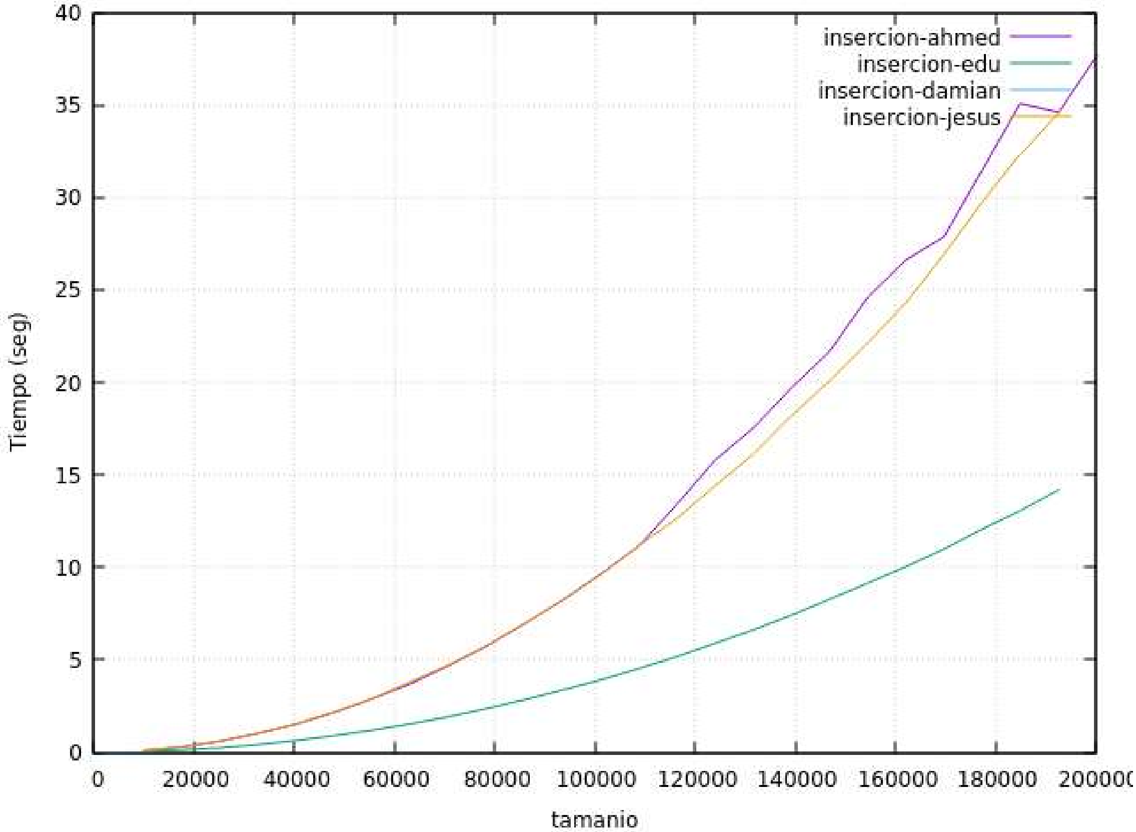
BURBUJA



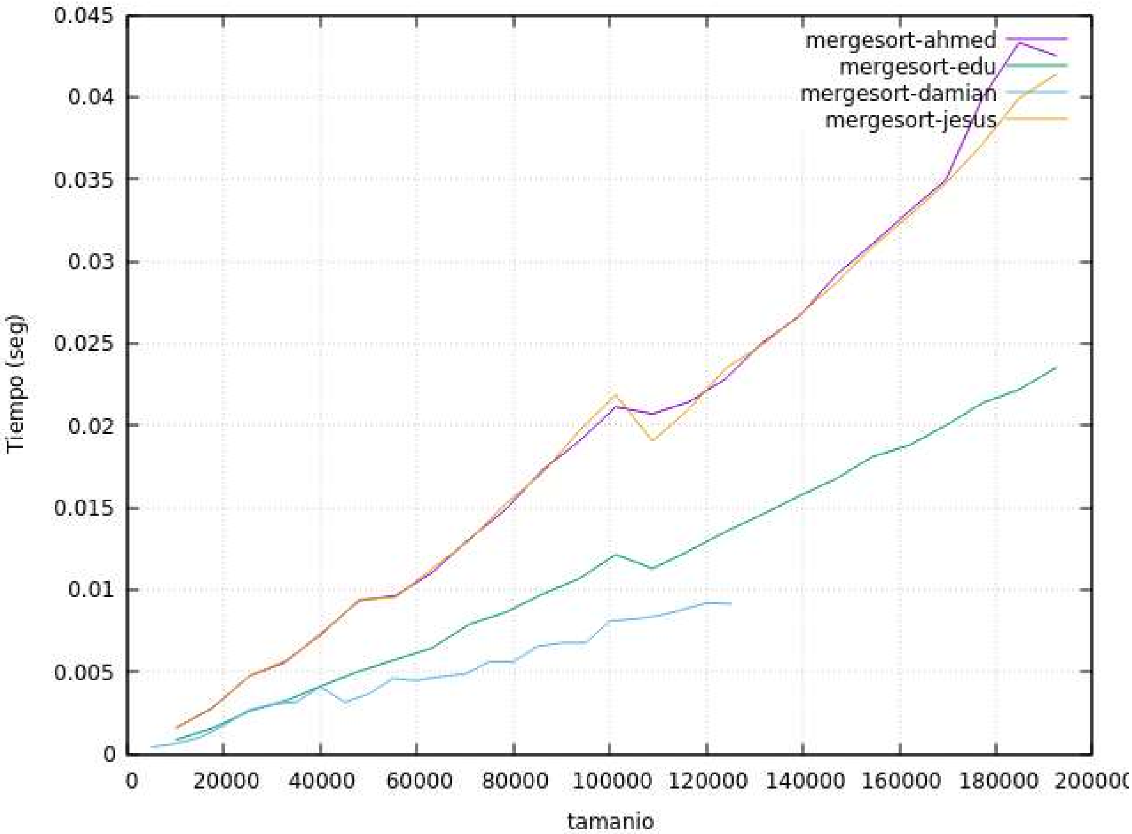
SELECCIÓN



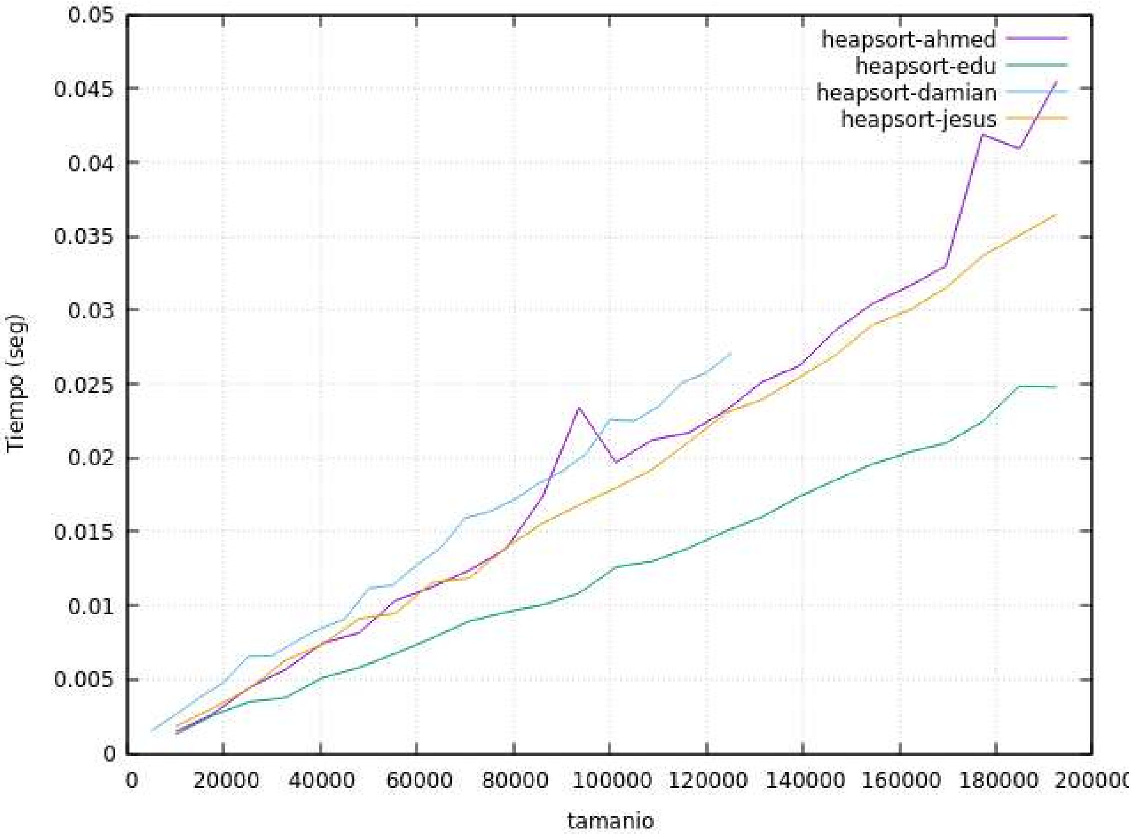
INSERCIÓN



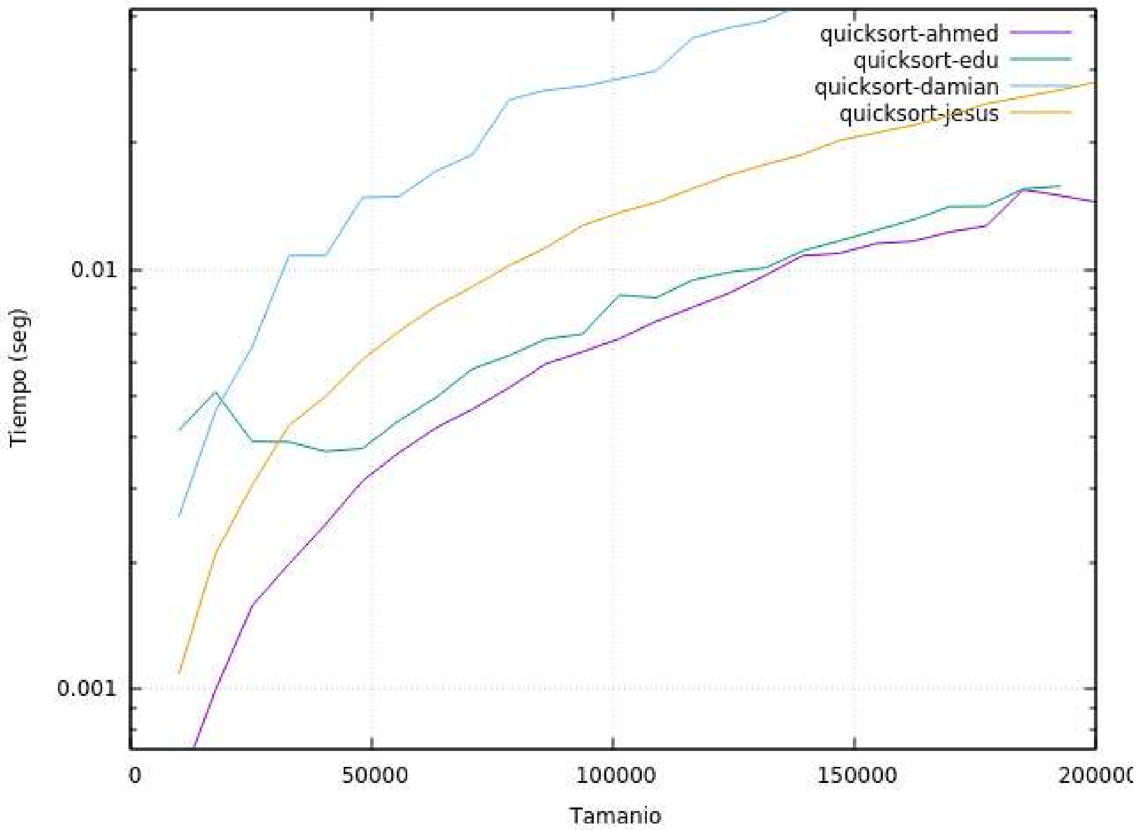
MERGESORT



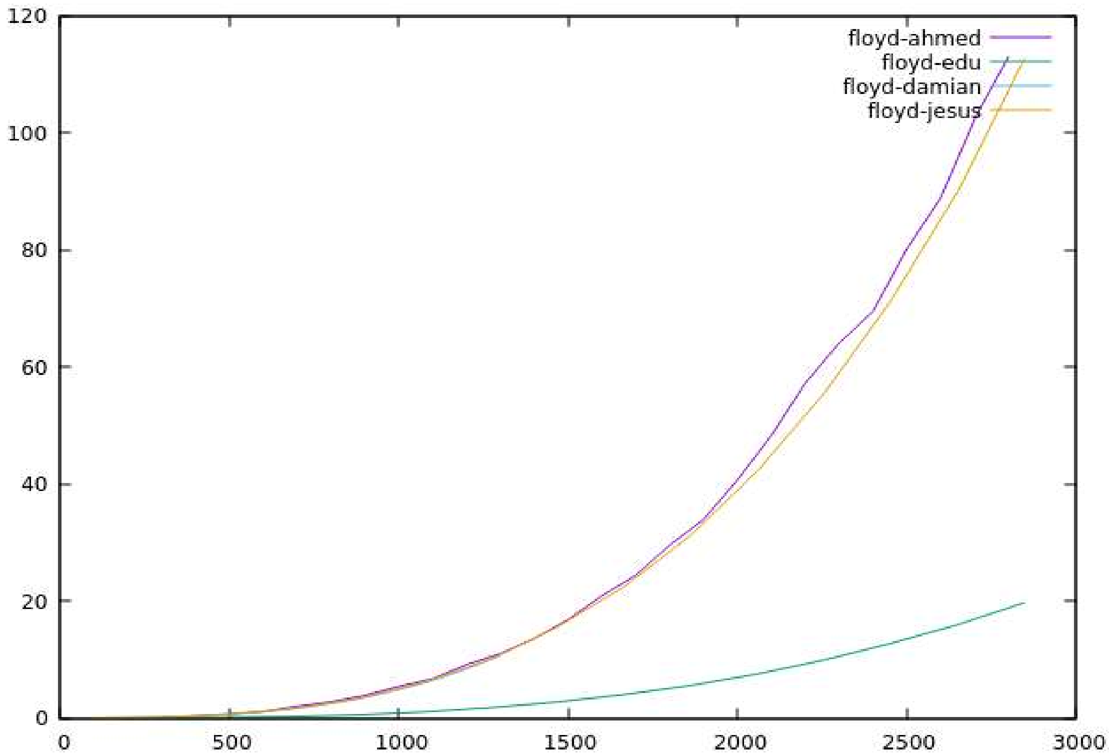
HEAPSORT



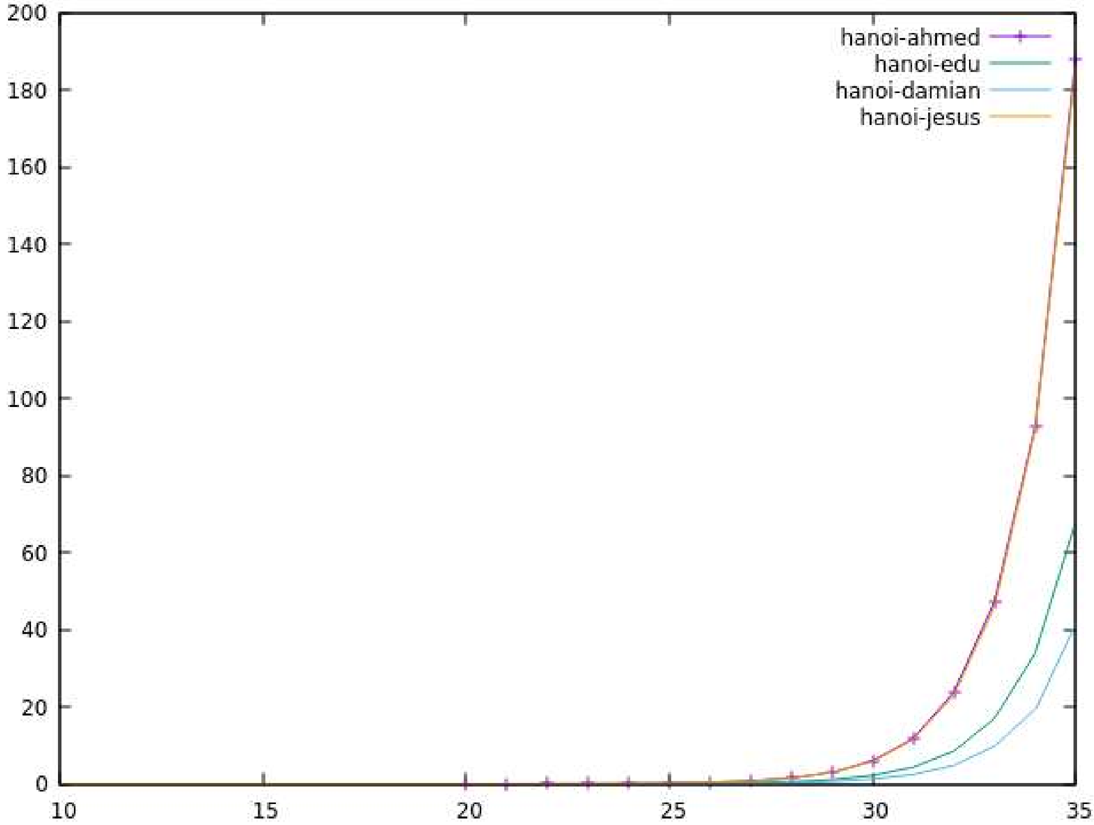
QUICKSORT



FLOYD



HANOI



Como podemos ver en las comparativas de absolutamente todos los algoritmos en los 4 equipos mencionados previamente, hay grandes diferencias de tiempo entre unos y otros, pero la curva se mantiene con la misma forma en todos ellos, esto quiere decir que, el algoritmo es el mismo y no variará dependiendo del hardware en eficiencia, aunque si en tiempo para un determinado tamaño.

7. Comparación entre sistemas.

En este apartado podemos ver como los resultados de la ejecución en dos sistemas operativos, realmente en dos distribuciones de Linux. La primera de estas dos es ampliamente conocida por todos nosotros, hablamos de Ubuntu en su versión 18.04 LTS, la segunda distribución también conocida (aunque menos que Ubuntu) es Linux Mint o simplemente Mint. Los resultados de esta comparativa son los siguientes:

BURBUJA

BURBUJA		
Tamaño	Ubuntu 18.04	Linux Mint
10000	0,135794	0,141672
17600	0,437113	0,456428
25200	0,9387	0,9678
32800	1,63414	1,68533
40400	2,51504	2,53282
48000	3,60458	3,64234
55600	4,86092	4,87653
63200	6,28132	6,29274
70800	7,92288	7,92872
78400	9,73844	9,74021
86000	11,7568	11,8895
93600	13,9396	14,0254
101200	16,3617	16,4972
108800	18,8811	18,8993
116400	21,5946	21,6287
124000	24,5695	24,5978
131600	27,6027	27,6813
139200	31,0196	31,0632
146800	34,4855	34,5928
154400	37,9288	37,9661
162000	42,2114	42,3002
169600	45,8808	45,9349
177200	50,3061	50,4662
184800	54,6127	54,7014
192400	59,1245	59,1912

INSERCIÓN

INSERCIÓN		
Tamaño	Ubuntu 18.04	Linux Mint
10000	0,0535	0,059234
17600	0,1213	0,145545
25200	0,2441	0,287675
32800	0,4155	0,402134
40400	0,6369	0,701659
48000	0,8930	0,987413
55600	1,1865	1,18764
63200	1,5362	1,58734
70800	1,9179	1,90127
78400	2,3572	2,38765
86000	2,8379	2,95675
93600	3,3718	3,12432
101200	3,9144	401.230
108800	4,5317	4,87124
116400	5,1818	5,24665
124000	5,8888	6,12564
131600	6,6235	6,98128
139200	7,4082	7,75239
146800	8,2684	8,43665
154400	9,1455	9,72187
162000	10,0362	10,1743
169600	10,9883	11,5467
177200	12,0652	12,9668
184800	13,0542	14,7886
192400	14,1706	15,0452

SELECCIÓN

SELECCIÓN		
Tamaño	Ubuntu 18.04	Linux Mint
10000	0,0615	0,057568
17600	0,1709	0,156234
25200	0,3459	0,313534
32800	0,5964	0,572359
40400	0,8955	0,866778
48000	1,2647	1,24353
55600	1,6597	1,85645
63200	2,1561	2,12439
70800	2,6686	2,56428
78400	3,2870	3,14235
86000	3,9219	3,84375
93600	4,6762	4,36568
101200	5,4492	5,1325
108800	6,3793	5,98342
116400	7,2670	6,54575
124000	8,2312	7,23456
131600	9,2279	8,99214
139200	10,3657	9,23418
146800	11,4979	10,9934
154400	12,8840	11,7565
162000	14,0157	13,0185
169600	15,3376	14,9819
177200	16,7884	15,6547

MERGESORT

MERGESORT		
Tamaño	Ubuntu 18.04	Linux Mint
10000	0,00084106	0,000123
17600	0,001551	0,001456
25200	0,002604	0,002345
32800	0,003229	0,003322
40400	0,004169	0,004234
48000	0,005028	0,004987
55600	0,005742	0,005456
63200	0,006456	0,006021
70800	0,00788	0,007345
78400	0,008629	0,008456
86000	0,00972	0,00954
93600	0,010669	0,010657
101200	0,012139	0,012234
108800	0,011284	0,011645
116400	0,012361	0,012867
124000	0,013532	0,014567
131600	0,014581	0,014613
139200	0,015701	0,015876
146800	0,016744	0,016435
154400	0,018101	0,018123
162000	0,018791	0,019123
169600	0,019999	0,019432
177200	0,021377	0,020456

HEAPSORT

HEAPSORT		
Tamaño	Ubuntu 18.04	Linux Mint
10000	0,001271	0,001234
17600	0,002556	0,002234
25200	0,003479	0,003234
32800	0,003776	0,003897
40400	0,005112	0,004987
48000	0,0058	0,0056
55600	0,00676	0,00656
63200	0,007805	0,007546
70800	0,008934	0,009879
78400	0,009548	0,01038
86000	0,01004	0,010588
93600	0,010846	0,010898
101200	0,012589	0,013435
108800	0,012995	0,013787
116400	0,013904	0,013998
124000	0,015027	0,015234
131600	0,016009	0,016564
139200	0,017362	0,018123
146800	0,018501	0,018786
154400	0,019572	0,019678
162000	0,020374	0,020465
169600	0,020997	0,021567
177200	0,022439	0,023245

QUICKSORT

QUICKSORT		
Tamaño	Ubuntu 18.04	Linux Mint
10000	0,004124	0,004233
17600	0,0051	0,005234
25200	0,003883	0,003765
32800	0,003871	0,003887
40400	0,003678	0,003653
48000	0,00374	0,003768
55600	0,004353	0,004275
63200	0,004942	0,005134
70800	0,005782	0,005893
78400	0,006221	0,006354
86000	0,006816	0,006912
93600	0,007008	0,007112
101200	0,008681	0,008975
108800	0,008559	0,008764
116400	0,009423	0,009767
124000	0,009839	0,009991
131600	0,010091	0,010145
139200	0,011066	0,011145
146800	0,01168	0,011877
154400	0,01239	0,01271
162000	0,013123	0,01465
169600	0,014106	0,014923
177200	0,014125	0,015124

FLOYD

FLOYD		
Tamaño	Ubuntu 18.04	Linux Mint
100	0,0042	0,0044
296	0,0279	0,0282
492	0,0969	0,0969
688	0,2508	0,2572
884	0,5339	0,5692
1080	1,0205	1,0247
1276	1,7671	1,7752
1472	2,7361	2,7425
1668	4,0023	4,0135
1864	5,5513	5,5621
2060	7,5116	7,5342
2256	9,8823	9,9312
2452	12,7115	13,8823
2648	15,8841	17,9232
2844	19,6339	19,7213
3040	23,8525	24,8892
3236	28,9473	31,0123
3432	34,5302	36,5986
3628	40,7294	41,7233
3824	47,6500	48,6534
4020	56,2221	57,0283
4216	65,6965	67,8723
4412	73,0946	75,5847

HANOI

HANOI		
Tamaño	Ubuntu 18.04	Linux Mint
15	0,000136	0,000141
16	0,000151	0,000156
17	0,000294	0,000291
18	0,000587	0,000601
19	0,001705	0,00182
20	0,002508	0,002702
21	0,005376	0,006237
22	0,010741	0,01128
23	0,020842	0,021498
24	0,034249	0,043298
25	0,067767	0,07654
26	0,134409	0,15891
27	0,268069	0,28754
28	0,530572	0,52092
29	1,06226	1,39849
30	2,12342	2,29842
31	4,26198	4,98283
32	8,47872	8,49866
33	16,9479	17,1234
34	33,9082	35,4534
35	67,7679	69,2323
36	135,539	139,3244
37	271,653	276,4545

No necesitamos una gráfica para darnos cuenta de que en todos los cálculos vemos que Mint es más lento que Ubuntu, pero por muy poco. Según los cálculos realizados Linux Mint es entre un 3% y un 8% más lento que Ubuntu.

Evidentemente estos resultados no alteran en nada los algoritmos, es decir, mantienen sus curvas y ordenes de eficiencia como en las anteriores representaciones.

6. Conclusiones.

Tras varias ejecuciones de todos los algoritmos, comparaciones, gráficas y teóricas deducimos que el algoritmo más eficiente para ordenación es el quicksort, lo cual no es una sorpresa (su propio nombre lo indica), este es un algoritmo de orden O(n*log(n)) no tan bueno como un algoritmo O(n) pero bastante bueno para la aplicación que tiene.

El algoritmo de ordenación más lento es el algoritmo de burbuja, siendo hasta 3 veces más lento que el siguiente cuadrático más lento. De los algoritmos de ordenación destaca mergesort por un gran consumo de memoria, de hecho, para un tamaño de entrada de 10.000.000.0000 ha consumido el total de memoria disponible en el pc donde se ejecutó (16 GB RAM) llevando a su cancelación.

Al comparar el algoritmo de Floyd con los cuadráticos (los polinomiales más lentos) vemos que es mucho más lento e ineficiente, sin embargo, basta con ver la gráfica de comparación con el algoritmo de Hanói para darse cuenta de que no es tan malo y de hecho comparándolo con el de Hanói es una bendición ya que esté (el de Hanói) es el más lento de los algoritmos vistos en esta práctica, es un algoritmo no polinomial, exponencial en base 2, realmente ineficiente. Para un tamaño de entrada 100 pasarán años hasta obtener una solución.

En cuanto a las diferencias observadas al ejecutar un programa que implementa un algoritmo u otro cabe destacar que la optimización es realmente útil y notoria, sobre todo si se van a tratar con tamaños de entrada realmente grandes. Así pues, vemos que el algoritmo que se ve más favorecido es el de Floyd (hasta 6 veces más rápido) mientras que el que menos aprovecha estas mejoras es el algoritmo de burbuja (llegando a ser más del doble de rápido, pero solo eso).

Al realizar las mediciones en sistemas operativos diferentes no apreciamos grandes diferencias, en este guion se ha documentado la diferencia entre Ubuntu y Mint, la cual no es nada de otro mundo ya que no se ha llegado a ver ni un 10% de diferencia. Aunque no se ha documentado, también se han realizado pruebas en Windows 10, obteniendo una diferencia similar a la del caso anterior, pero siendo esta vez Ubuntu menos eficiente.

Para terminar, las diferencias en el hardware han sido bastante notables, aunque no han degradado las curvas de los algoritmos ni mucho menos, se han obtenido los resultados esperados, es decir, mayor potencia de cómputo equivale a menor tiempo. Las capacidades de memoria RAM no han tenido un gran efecto pues no se han llegado a usar entradas de tamaño tan grande como la capacidad que tenemos en cada equipo (mínimo global 8 GB RAM).