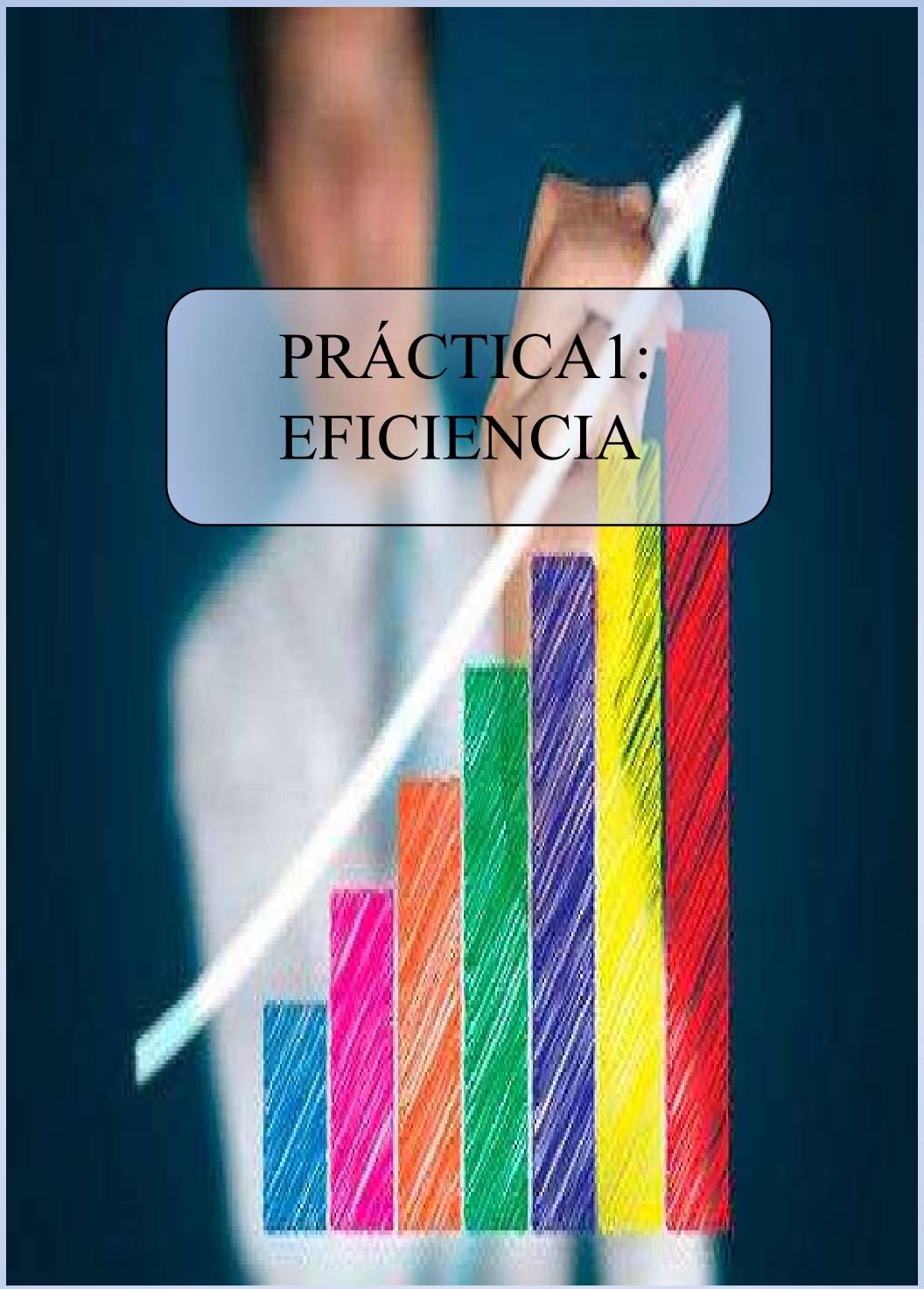


PRÁCTICA1: EFICIENCIA

Práctica 1: eficiencia.

Versión individual de la primera práctica de la asignatura de algorítmica

Autor: **Ahmed El Moukhtari Koubaa**

Asignatura: **Algorítmica**

Fecha: **10 de febrero de 2020**

Curso: **2019/2020**

Profesora: **María Teresa Lamata Jiménez**

ÍNDICE

1. Descripción del problema.
2. Implementación del código.
3. Mediciones empíricas.
4. Gráficas y órdenes de eficiencia.
5. Efecto de los parámetros externos.
6. Conclusiones.

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.

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

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

```
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.

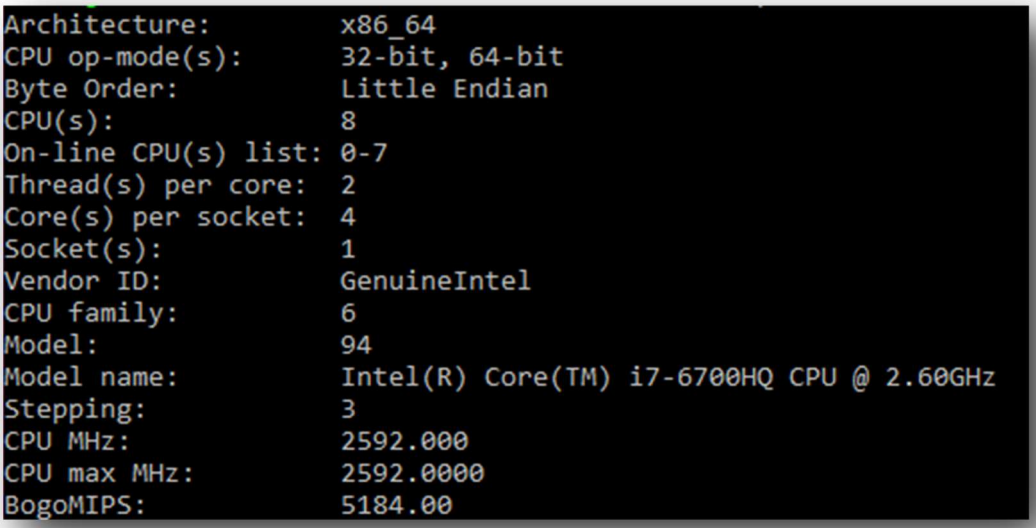
```bash
#!/bin/bash

#-------------------------------------------------#
# 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:



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

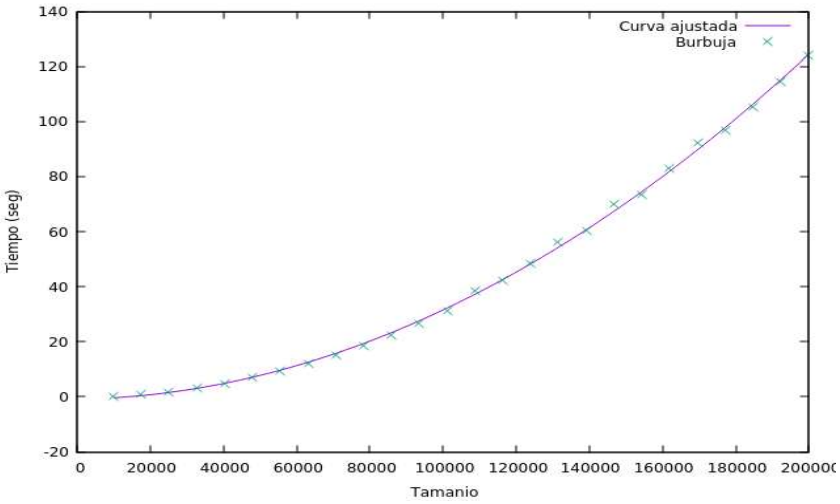
Para los algoritmos de la última serie una comparación con los anteriores no tiene ningún sentido pues no llevan a cabo una tarea parecida y los tamaños son muy diferentes. Veamos una tabla de cada uno de estos algoritmos. Ambos algoritmos son bastante lentos y necesitamos reducir el tamaño. Podemos 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

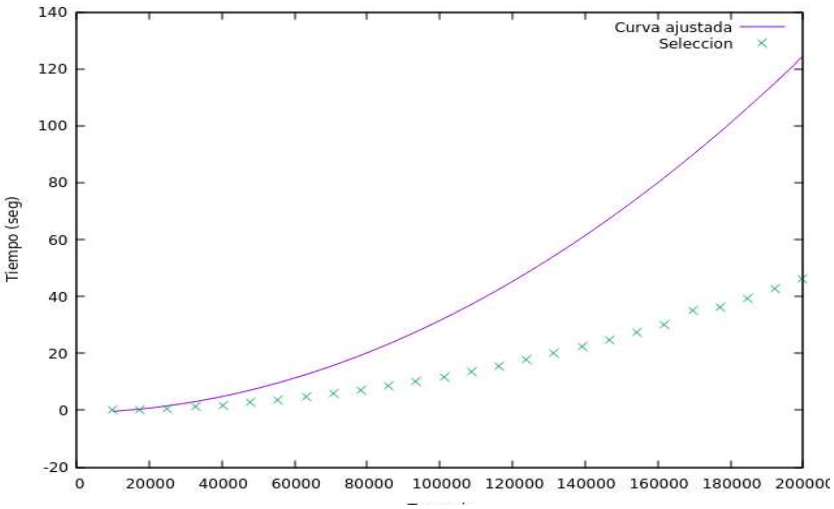
4. Gráficas y órdenes de eficiencia.

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.



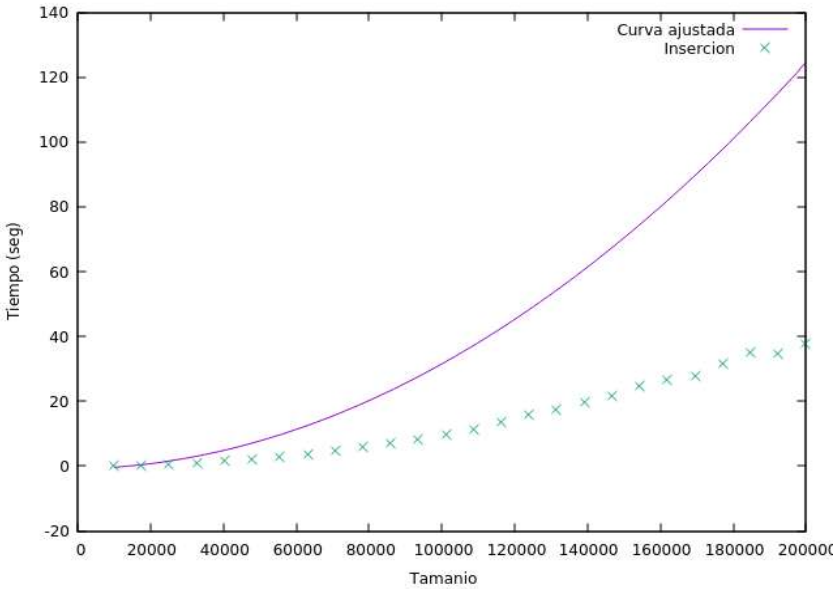
Burbuja: a0*x*x+a1*x+a2

El más lento de los cuadráticos



Selección: a0*x*x+a1*x+a2

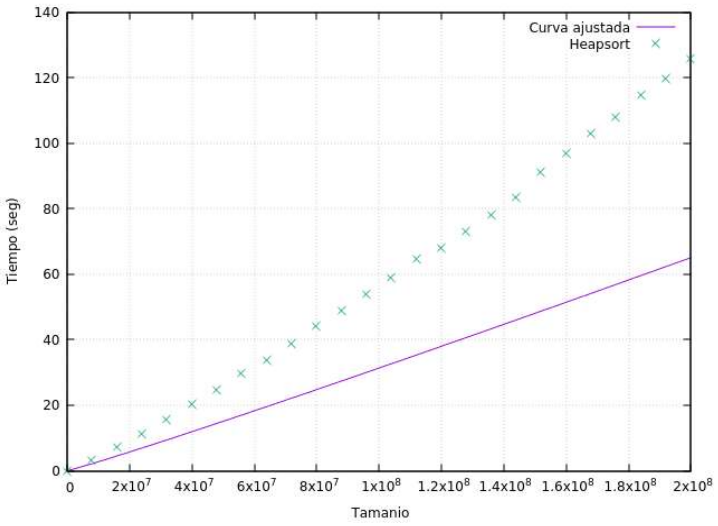
Mejor que burbuja, casi tan bueno como inserción.



Inserción: a0*x*x+a1*x+a2

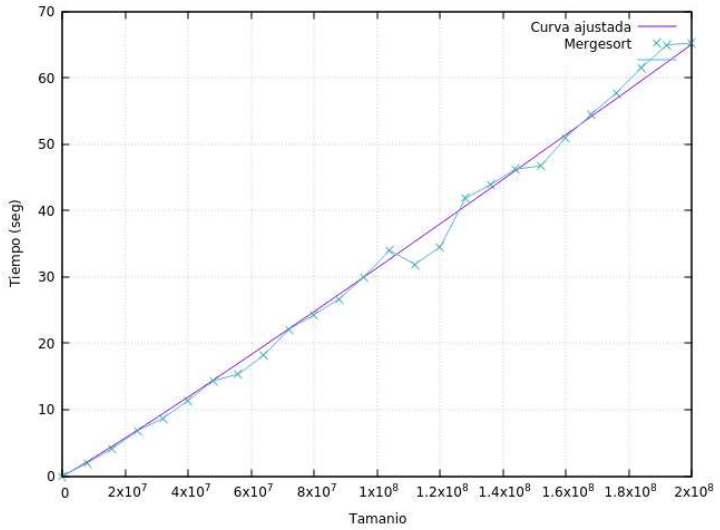
El mejor de los algoritmos de ordenación de orden cuadrático vistos en esta práctica.

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



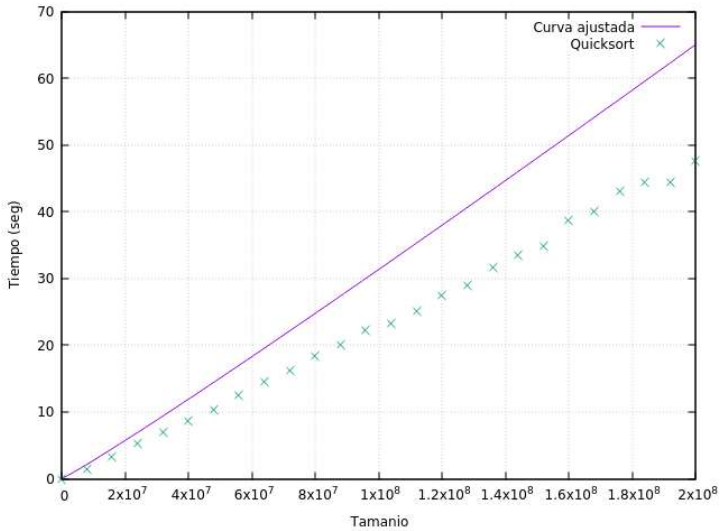
Heapsort: a0*n*log(n) + a1

Más ineficiente que el mergesort



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

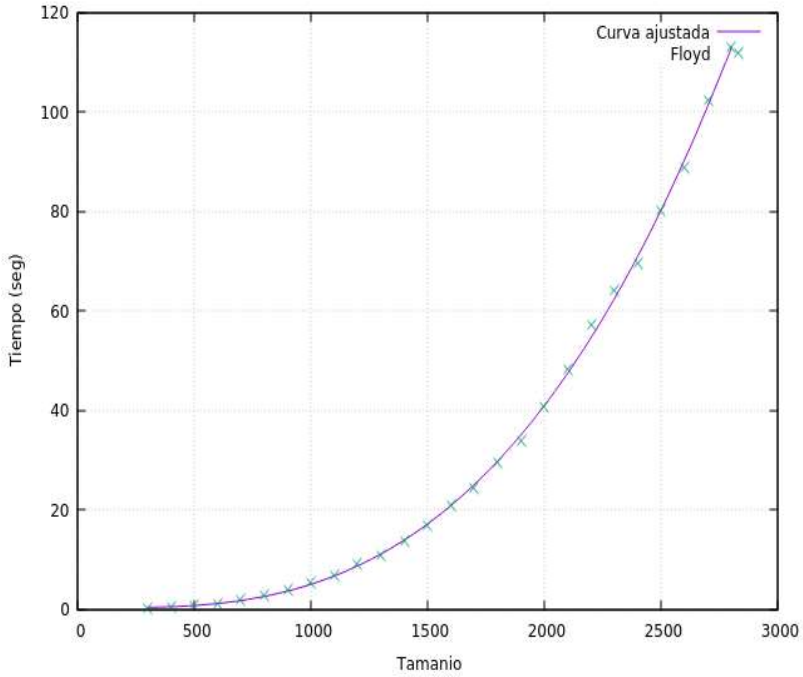
El mergesort va dando una especie de saltos cada x tamaños. Más ineficiente que el quicksort. Su consumo de memoria es elevado.



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

El más rápido de los algoritmos de ordenación.

Los algoritmos de mayor orden de eficiencia.



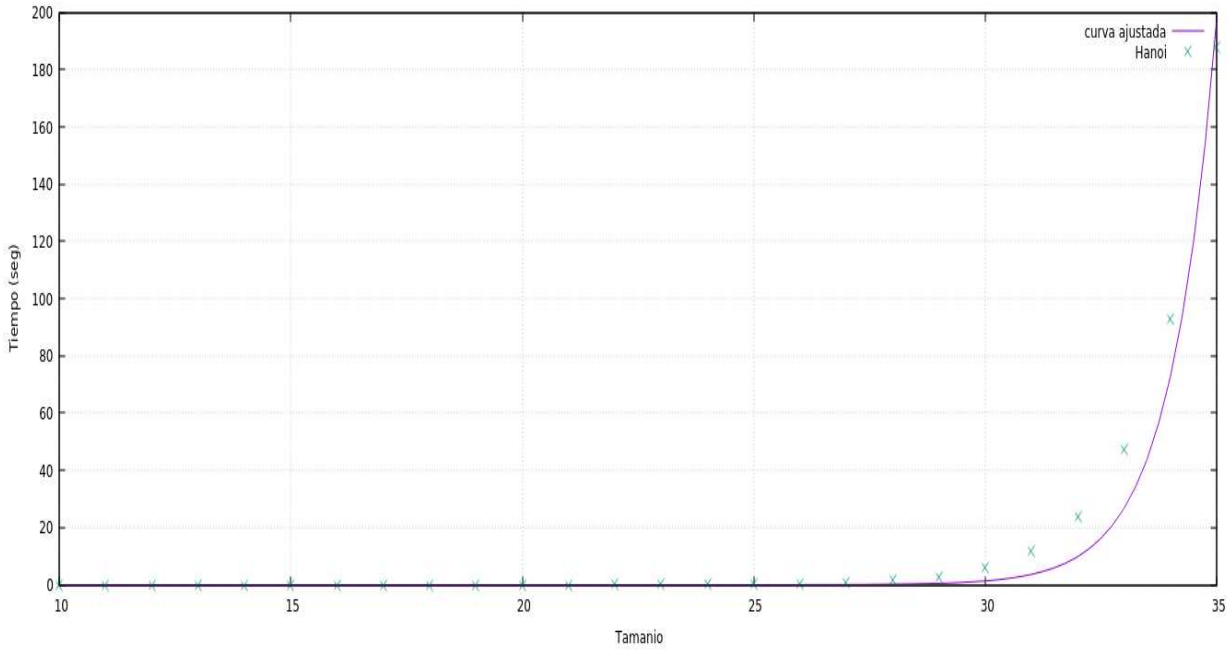
Floyd:

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

Bastante más lento que los algoritmos de ordenación básicos.

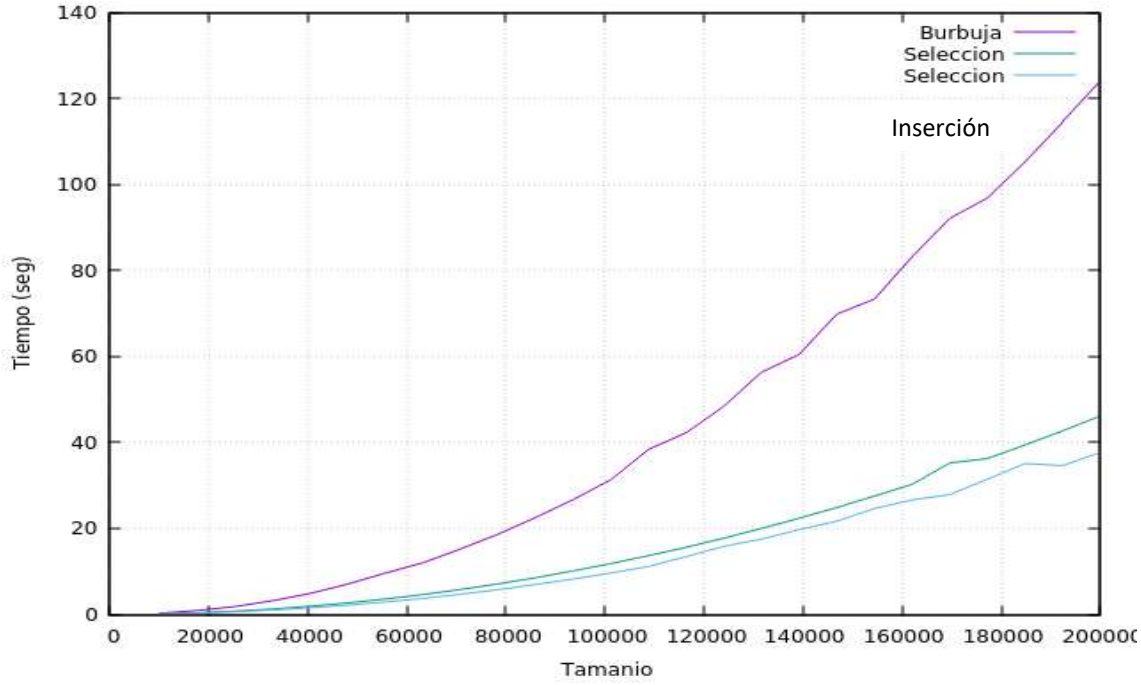
Hanói: a0*2*exp(x)

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

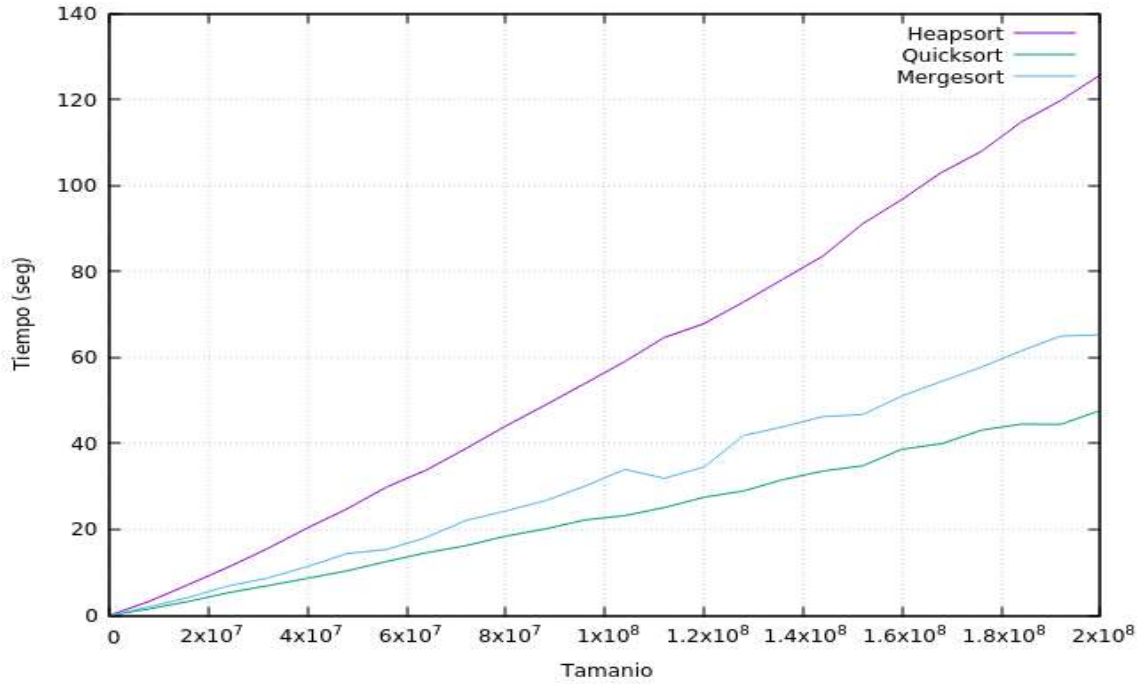


Vamos a comparar todos los algoritmos gráficamente para ver sus diferencias y semejanzas más claramente.

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.

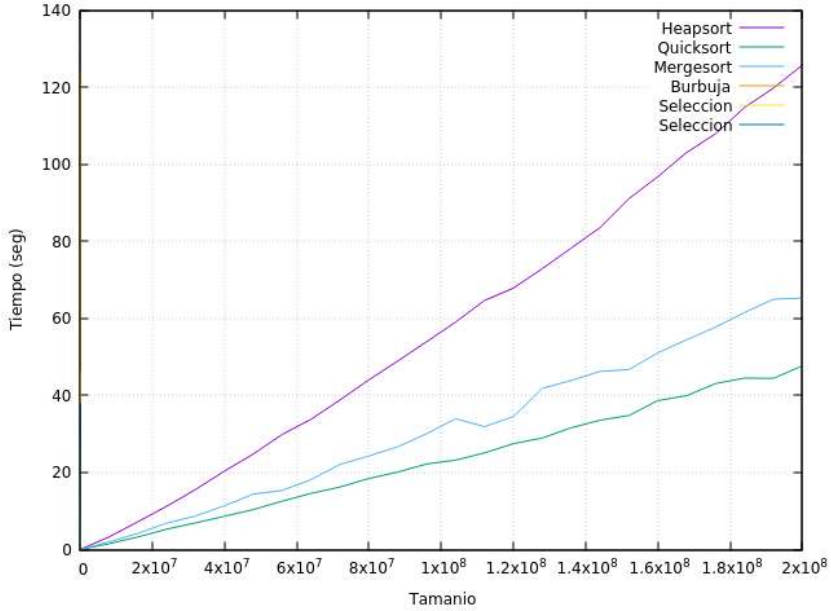


Para los algoritmos de ordenación más eficientes tenemos la siguiente comparación. Se observa que heapsort es el más lento y quicksort el más rápido como esperábamos, casi 3 veces más rápido que heapsort y 1/3 más rápido que mergesort el cual no es nada lento.

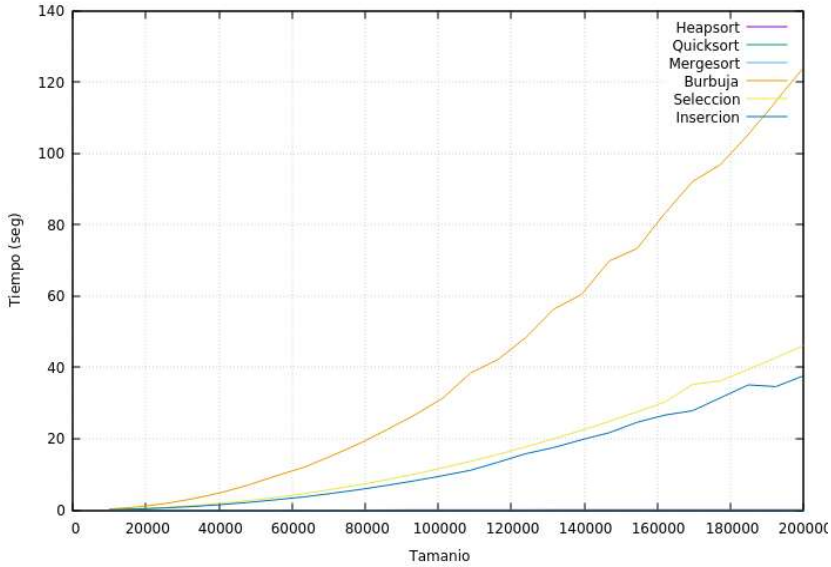


Comparemos ahora todos los algoritmos de ordenación, tenemos hechas 2 mediciones para los algoritmos más rápidos. Una con el mismo n que los básicos y otra con entradas 1000 veces más grandes.

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

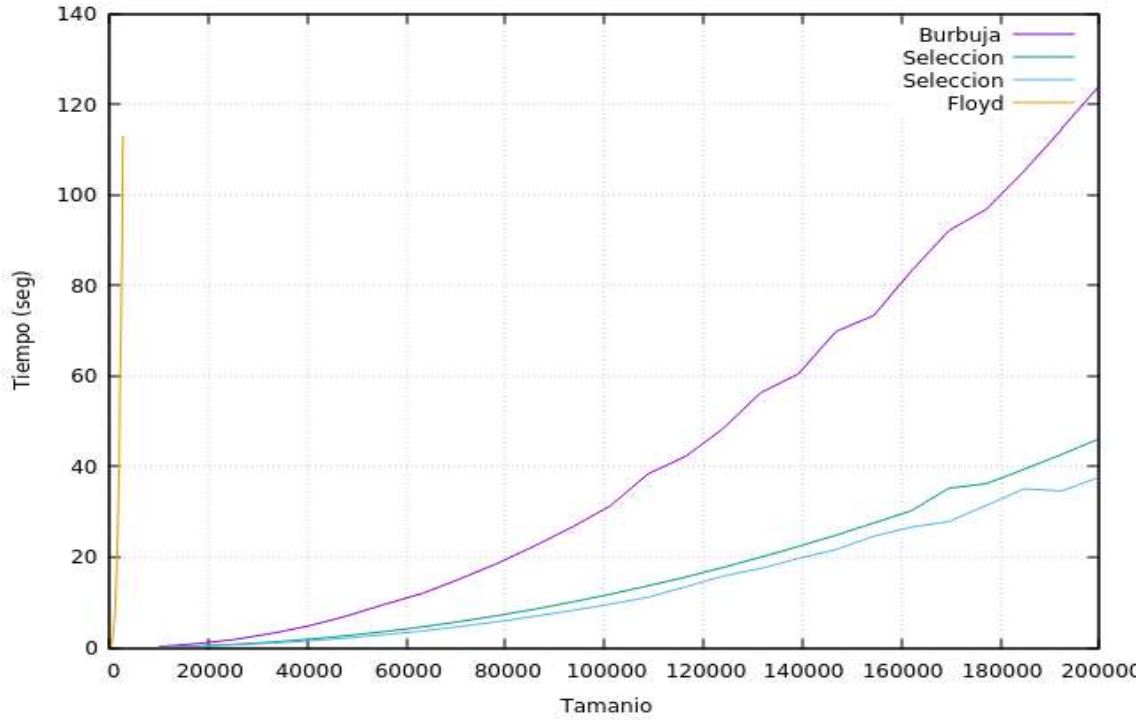


En esta podemos ver que en lo que los básicos ordenan 200 mil elementos los rápidos ordenan 200 millones.

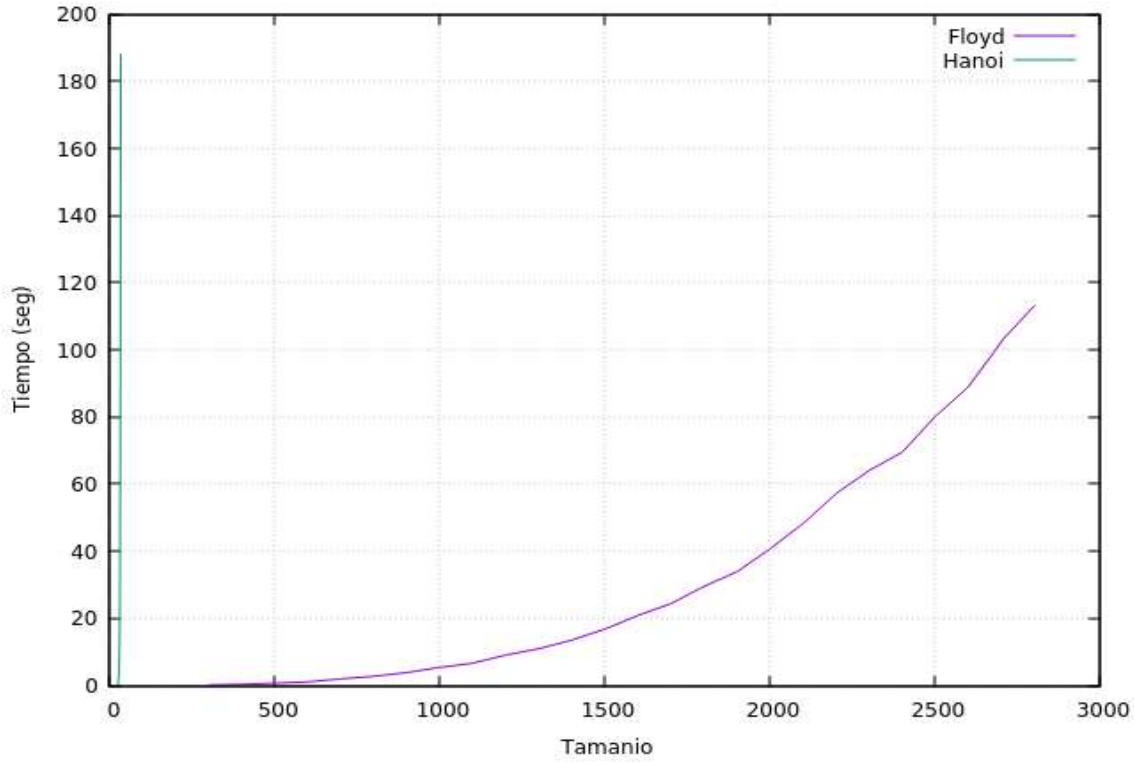


En esta otra comparativa vemos que literalmente no hay punto de comparación entre los rápidos y los lentos, pues los rápidos ni se aprecian.

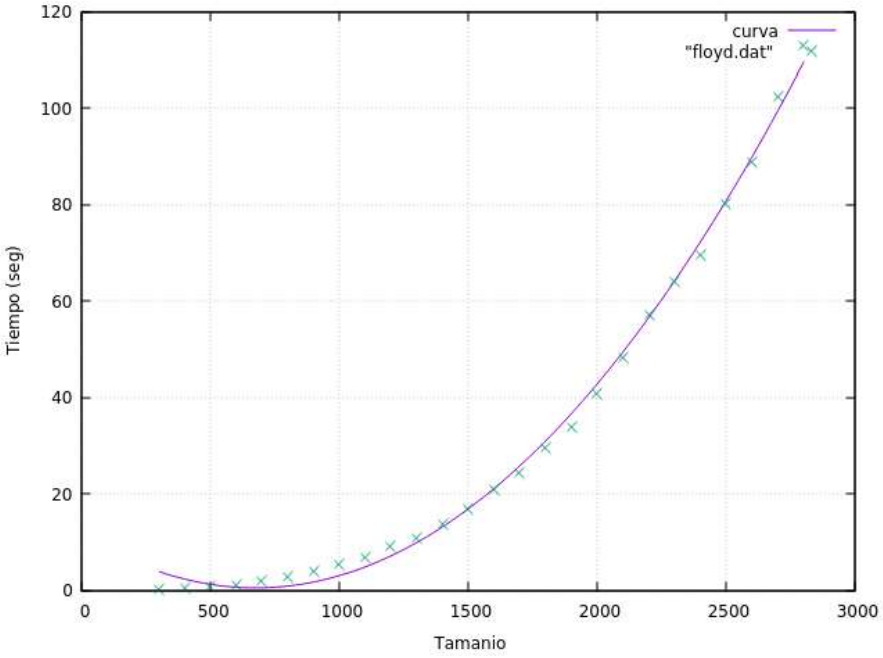
Comparemos ahora el algoritmo de Floyd con los de orden cuadrado. Podemos ver que dicho algoritmo es realmente lento en comparación a estos a pesar de ser polinomial. Para el tamaño mínimo de los cuadrados Floyd tardaría muchísimo tiempo, aunque es un tiempo finito.



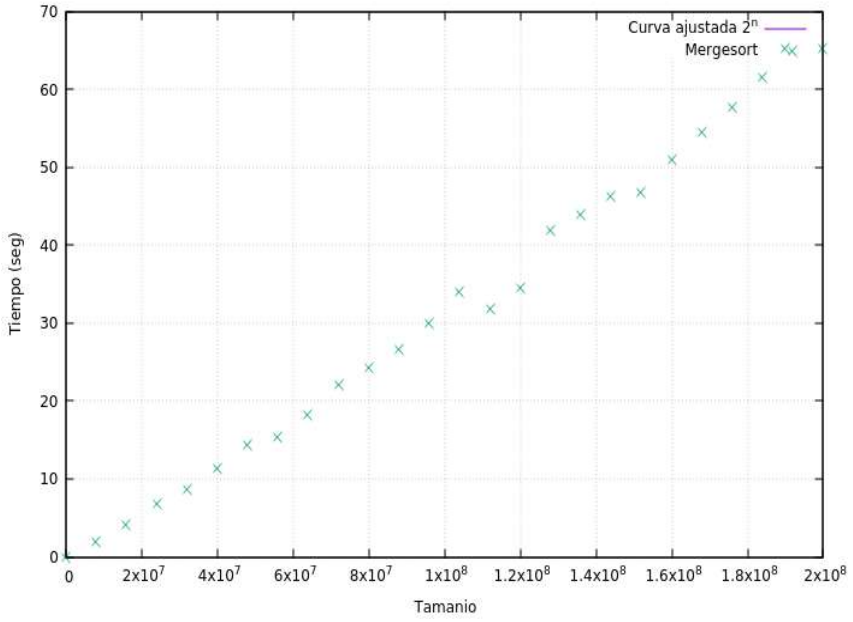
Al comparar Floyd con Hanói, nos damos cuenta de que este es una bendición pues Hanói es prácticamente una línea vertical debido a que es no polinomial.



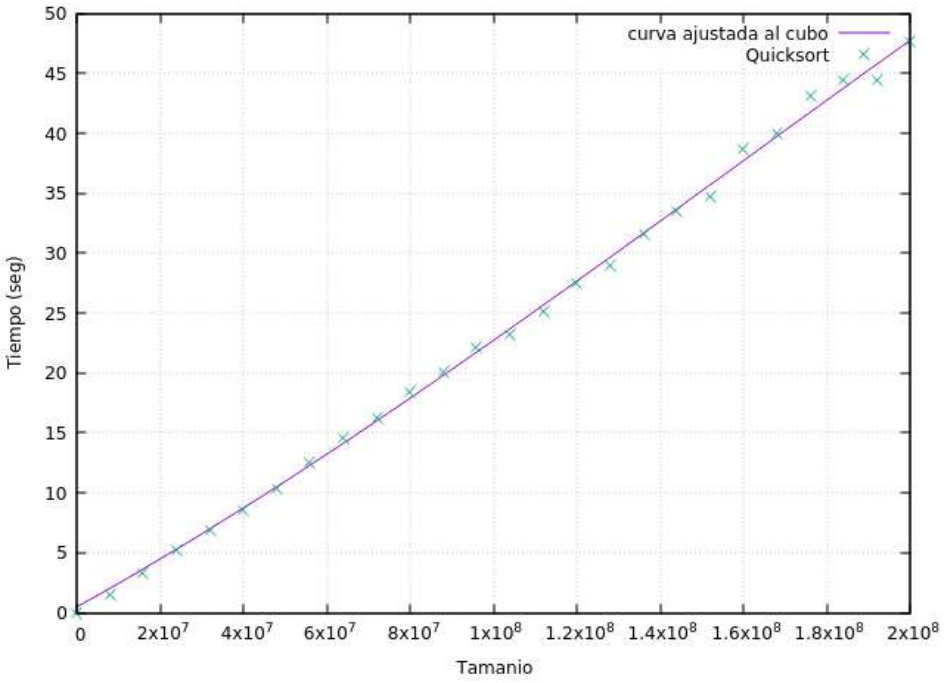
Probemos a calcular la eficiencia híbrida con ajustes diferentes:



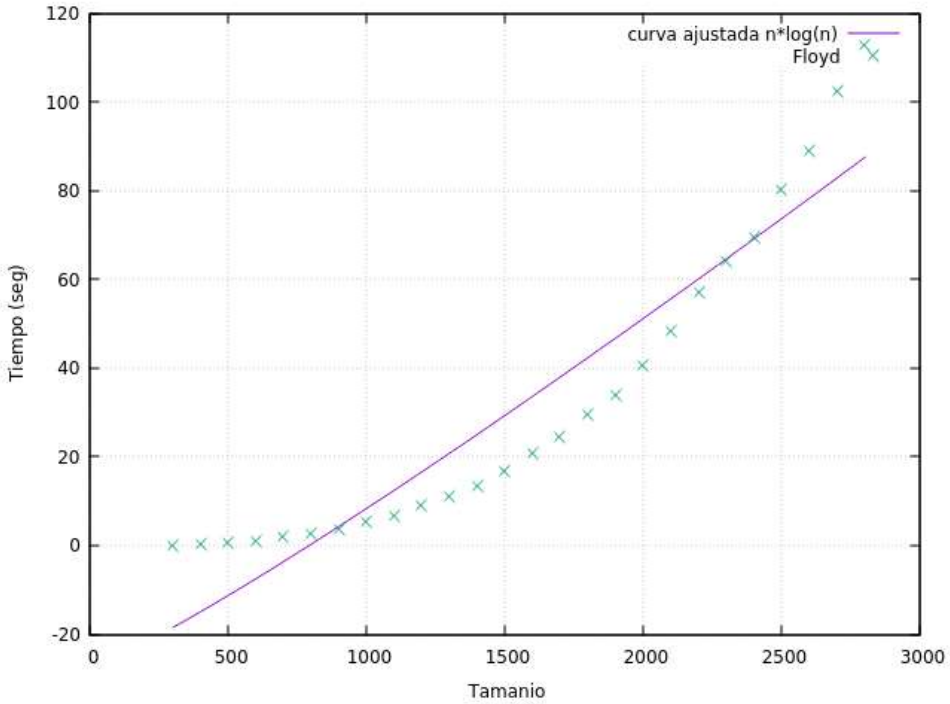
Floyd con un ajuste cuadrático. Apenas hay diferencias.



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. Efecto de los parámetros externos.

Todas las mediciones que se han llevado hasta ahora se han realizado en Ubuntu 18.04 lts, al ejecutar los programas en Windows 10 no se aprecian grandes cambios (principalmente porque fueron ejecutadas en la terminal de Ubuntu). Lo único que se aprecia es que en Windows se usa poco más de un 20% de CPU mientras que en Ubuntu es el 100%.

Todos los programas ejecutados que se han ejecutado fueron compilados sin ninguna optimización, es decir, g++ -o <nombre> <programa>.cpp –std=g++0x. Al compilar con –O3 se reducen muchísimo los tiempos de ejecuciones, estos son algunos ejemplos:

Burbuja	50.000 → 8.00659
BurbujaOpt	50.000 → 4.19916
Quicksort	10.000.000 → 1.95716
QuicksortOpt	10.000.000 → 0.960342
Floyd	1.000 → 5.75
FloydOpt	1.000 → 0.84375
Hanói	30 → 6.9188
HanóiOpt	30 → 2.2355

Como podemos ver la mejora es notable, para los algoritmos más rápidos es menos notable, aunque se reduce hasta a la mitad. Los algoritmos más lentos son los que más favorecidos se ven. Floyd reduce hasta 8 veces su tiempo de ejecución.

Esta mejora es debida a que el compilador es capaz de intuir las instrucciones que se están llevando a cabo, mejorar la predicción y sobre todo aprovechar al máximo el paralelismo a nivel de datos e instrucciones pues disponemos de un procesador multicore y multihebra.

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 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 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.