UNIVERSIDAD
DE GRANADA

Escuela Técnica Superior de Ingeniería Informática y
Telecomunicaciones

Práctica 1:

ANÁLISIS DE EFICIENCIA DE ALGORITMOS

Titulación: DGIIM

Asignatura: Algorítmica

Curso: 2022/23

Autor: Antonio Javier Rodríguez Romero

Correo: antoniojrr@correo.ugr.es

ÍNDICE:

1. OBJETIVOS

Esta práctica se basa en el análisis de eficiencia de distintos algoritmos de ordenación de *arrays* de distintos órdenes. La meta de esta será observar las diferencias entre cada uno de estos, comparar su eficacia trabajando en vectores de distintos tamaños mediante su análisis teórico, empírico e híbrido y concluir en cada caso cual sería el óptimo.

Complementariamente, observaremos cómo afecta el medio de ejecución del programa (Hardware de la máquina) y la compilación (Niveles de optimización del compilador) a los tiempos de cada uno de los algoritmos.

2. DISEÑO DEL ESTUDIO

2.1. PLANTEAMIENTO

Con el objetivo de estudiar cada uno de estos algoritmos por separado, se ha llevado acabo la implementación de estos en lenguaje *C++*, de forma que, con la utilización de bibliotecas de la *STL* como *chrono* o *clock*, podremos automatizar la toma de tiempos en la ejecución de estos en *arrays* de longitud variable (Utilización de memoria dinámica).

Además, con la meta de facilitar aún más estos trabajos, se ha trabajado en un sistema de operativo *Linux*, que nos da la posibilidad de crear *scripts* que, aun siendo muy sencillos, realizan tareas como:

- Ejecución repetida los algoritmos con distintos argumentos:

```
#!/bin/bash

i=<tamaño inicial>
while [ "$i" -le <tamaño final> ]
do
    printf "$i " >> $2
    ./$1 $i >> $2
    i=$(( $i + 5000 ))
    printf "\n" >> $2
done
```

Ilustración 1: Script "mide-tiempos.sh"

- Creación de gráficas y ajuste por regresión empleando mínimos cuadrados (Para lo que también ha sido necesario *gnuplot*, software de representación gráfica):

```
#!/bin/bash
#
# Uso: gnuplot dibuja.sh

f(x) = <función a ajustar>
fit f(x) '<datos>' via <parámetros>
set terminal png

set output 'ajuste.png'
set xlabel "Tamaño"
set ylabel "Tiempo (segundos)"
plot '<datos>' title '<título>', f(x) title 'Curva ajustada'

set output 'grafica.png'
set xlabel "Tamaño"
set ylabel "Tiempo (segundos)"
plot '<datos1>' with lines title '<título1>', '<datos2>' with lines title '<título2>', '<datos3>'with lines title '<título3>', '<datos4>' with lines title '<título4>'
```

Ilustración 2: Script "dibuja.sh"

Por otro lado, para el correcto desarrollo de la práctica primero se ha tenido que realizar un estudio teórico de los algoritmos, de forma que hemos podido clasificarlos en dos grupos bien diferenciados.

Encontramos los algoritmos de búsqueda de orden $O(n^2)$, de los cuales estudiaremos:

- Algoritmo de ordenación Burbuja.
- Algoritmo de ordenación por Inserción.
- Algoritmo de ordenación por Selección.

Para estos, tomaremos los tiempos de cada uno en ordenar *arrays* de 5000 elementos hasta 125000, aumentando su tamaño de 5000 en 5000.

A su vez, encontramos los algoritmos de búsqueda de orden O(nlogn). De estos, escogeremos:

- Algoritmo *Heapsort*.
- Algoritmo *Mergesort*.
- Algoritmo *Quicksort*.

Como por su orden entendemos que serán más rápidos que los anteriores, en estos casos tomaremos *arrays* desde 5000000 hasta 125000000, aumentando su tamaño con saltos de 5000000 (A excepción del estudio de *Quicksort*, el cual, debido a que se producía *core dump*, se ha tenido que reducir el tamaño al de los *arrays* de los casos anteriores).

En cada uno de los algoritmos y para cada tamaño de *array* se ha realizado una ejecución del algoritmo en el caso de que el vector este rellenado aleatoriamente, que este ordenado inversamente y que ya se encuentre ordenado, para también comparar entre estos. En concreto, la secuencia de instrucciones que se ha llevado acabo para la medición ha sido:

```
#include <chrono>
…
t_antes = std::chrono::high_resolution_clock::now();
<Ejecución VECES veces del algoritmo>
t_despues =  std::chrono::high_resolution_clock::now();
transcurrido = std::chrono::duration_cast<std::chrono::duration<double>>(t_despues
- t_antes);
```

Como vemos, cada una de las medidas tomadas es el resultado de hacer la media de "*VECES*" tomas de datos reales (Con *VECES* igual a 5 en la mayoría de algoritmos y 10 en el caso de *Quicksort,* debido al disminuido tamaño de los *arrays,* y de algún caso de inestabilidad mayor.

2.2. HARDWARE

Por último, el entorno de estudio ha sido principalmente un portátil de características:

- Nombre: *Asus TUF Dash F15*
- Procesador: *11th Gen Intel Core i7-11370H @ 3.30GHz   3.30 GHz*
- *CPUs*: 8
- 16 GB *RAM*
- Cachés:
  - ***L1d:*** *192 KiB (4 instances)*
  - ***L1i:*** *128 KiB (4 instances)*
  - ***L2:*** *5 MiB (4 instances)*
  - ***L3:*** *12 MiB (1 instance)*

Para la comparativa entre este (Lo llamaremos $\mathbf{PC_1}$) y otros entornos, se ha realizado un estudio similar en dos portátiles más. Por un lado, las características del que llamaremos $\mathbf{PC_2}$ son:

- Nombre: *HP Pavilion 15-ec2003ns*
- Procesador: *AMD Ryzen 7 5800H with Radeon*
- *CPUs*: 16
- 16 GB *RAM*
- Cachés:
  - ***L1d:*** *256 KiB (4 instances)*
  - ***L1i:*** *256 KiB (4 instances)*
  - ***L2:*** *4 MiB (4 instances)*
  - ***L3:*** *16 MiB (1 instance)*

Por otro lado, el $\mathbf{PC_3}$ será:

- Nombre: *HP ENVY Laptop 13-ba1xxx*
- Procesador: *11th Gen Intel Core i7-1165G7 @ 2.80GHz, ~2.8GHz*
- *CPUs*: 8
- 16 GB *RAM*
- Cachés:
  - ***L1d:*** *192 KiB (4 instances)*
  - ***L1i:*** *128 KiB (4 instances)*
  - ***L2:*** *5 MiB (4 instances)*
  - ***L3:*** *12 MiB (1 instance)*

3. ALGORITMOS O(N^2)

3.1. ALGORITMO BURBUJA

3.1.1. Funcionamiento

```
void burbuja_lims(long double T[], int inicial, int final)
{
  int i, j;
  long double 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;
        }
}
```

El funcionamiento de este consiste en la comparación de dos elementos adyacentes en el *array* de forma que, si el primero es mayor que el segundo, su posición en el vector es intercambiada.

De esta forma, por cada iteración del primer bucle *for*, un elemento quedará colocado en su posición, comenzando por el más pequeño hasta colocar en la última iteración el más grande.

Las medidas en este algoritmo las hemos tomado con tamaño de *array* desde 5000 hasta 125000, con un orden inicial aleatorio, inverso y directo, y con optimización 0, 1, 2 y 3.

3.1.2. Tablas de datos

Tamaño	*Aleatorio*	*Inverso*	*Ordenado*
5000	0.0760818	0.0981694	0.0267685
10000	0.261763	0.396873	0.108281
15000	0.669641	0.904569	0.257206
20000	1.30734	1.65898	0.438897
25000	2.16812	2.48901	0.685232
30000	3.14369	3.58993	1.01485
35000	4.41686	5.13425	1.34234
40000	5.89326	6.42785	1.79147
45000	7.39177	8.25347	2.27659
50000	9.36771	10.1059	2.78869
55000	11.0047	11.9253	3.29298
60000	13.2624	14.3186	3.92974
65000	15.6777	16.9041	4.6151
70000	18.2346	19.5721	5.57548
75000	21.2536	22.834	6.14559
80000	24.065	25.5545	6.994
85000	27.1762	28.8506	7.90432
90000	30.8576	32.4438	8.88307
95000	34.0088	36.0713	9.87591
100000	38.4724	40.4878	10.9735
105000	41.8815	44.4782	13.2953
110000	47.0856	48.2291	13.188
115000	49.5649	52.1733	14.3569
120000	54.2223	57.2089	15.5923
125000	57.9138	60.9695	16.9283

Tabla 1: Burbuja Optimización 0

Tamaño	*Aleatorio*	*Inverso*	*Ordenado*
5000	0.0430727	0.0543675	0.0113763
10000	0.137902	0.213679	0.0428119
15000	0.397842	0.485541	0.09871
20000	0.828999	0.858209	0.171148
25000	1.34369	1.34023	0.267621
30000	1.99853	1.92736	0.386526
35000	2.8279	2.74624	0.534089
40000	3.82557	3.56418	0.691429
45000	4.90772	4.56164	0.958453
50000	6.29302	5.55551	1.15589
55000	7.51085	6.55679	1.33441
60000	9.0895	8.05677	1.56616
65000	10.5841	9.56917	1.90617
70000	12.5842	11.2773	2.25727
75000	14.2838	12.6524	2.42484
80000	16.395	14.501	2.88824
85000	18.824	16.6623	3.10834
90000	20.8159	18.1491	3.51009
95000	23.1523	20.513	3.9613
100000	25.5626	22.2996	4.42306
105000	28.3236	25.2198	4.78616
110000	32.0443	28.1172	5.28729
115000	35.0877	29.4414	5.7637
120000	38.4165	32.8884	6.30712
125000	41.8835	34.753	6.82249

Tabla 2: Burbuja Optimización 1

Tamaño	*Aleatorio*	*Inverso*	*Ordenado*
5000	0.0425151	0.0622688	0.0116888
10000	0.175281	0.246047	0.0428053
15000	0.486959	0.554755	0.0962719
20000	0.995468	1.02655	0.180396
25000	1.62196	1.62369	0.279922
30000	2.26417	2.21673	0.385708
35000	3.13376	3.0629	0.523599
40000	4.14824	3.94096	0.685474
45000	5.29867	4.9801	0.86564
50000	6.60282	6.15089	1.06877
55000	8.00764	7.43878	1.29637
60000	9.56976	8.8682	1.56448
65000	11.2924	10.3906	1.81706
70000	13.1711	12.0516	2.10785
75000	15.1279	13.8598	2.41036
80000	17.2574	15.7485	2.76831
85000	19.5865	17.9453	3.09911
90000	21.9318	19.9281	3.4652
95000	24.493	22.2864	3.87809
100000	27.2905	24.8109	4.29659
105000	30.0986	28.0562	4.7171
110000	32.9883	29.9786	5.22329
115000	36.2543	32.5392	5.66491
120000	39.4243	37.6109	6.36608
125000	43.5637	38.8413	6.7893

Tabla 3: Burbuja Optimización 2

Tamaño	*Aleatorio*	*Inverso*	*Ordenado*
5000	0.0394778	0.0638237	0.0116946
10000	0.1631	0.248356	0.0446288
15000	0.454211	0.550119	0.0962575
20000	0.866263	0.997553	0.172024
25000	1.41067	1.53235	0.266949
30000	2.1423	2.23163	0.395058
35000	2.95304	3.08227	0.526085
40000	3.93412	4.18612	0.717593
45000	5.03048	5.00173	0.864977
50000	6.06147	6.1628	1.06817
55000	7.59855	7.80281	1.31087
60000	9.00045	9.30324	1.59765
65000	10.7263	10.6237	1.86741
70000	12.199	12.0049	2.15412
75000	14.2894	14.0668	2.54758
80000	16.3639	15.6823	2.73971
85000	18.3124	17.6973	3.09415
90000	20.3023	20.5464	3.62782
95000	23.4633	22.7647	4.0003
100000	25.9332	25.0904	4.37126
105000	28.3414	28.3194	4.95453
110000	31.2285	29.6148	5.24206
115000	34.2961	33.7355	5.8444
120000	36.7498	35.3661	6.45919
125000	39.8595	38.5829	6.92337

Tabla 4: Burbuja Optimización 3

3.1.3. Gráficas

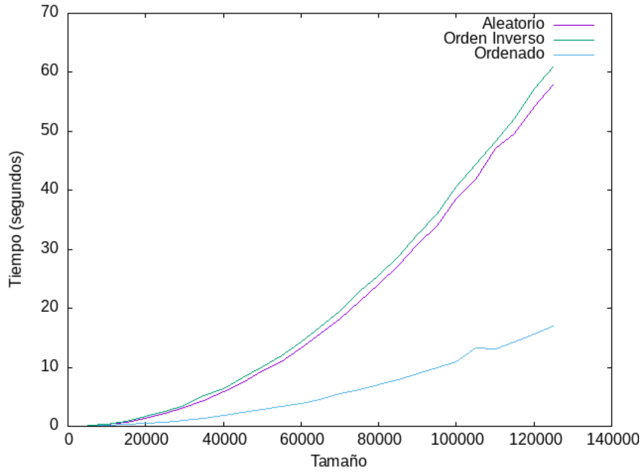

Ilustración 3: Burbuja Optimización 0

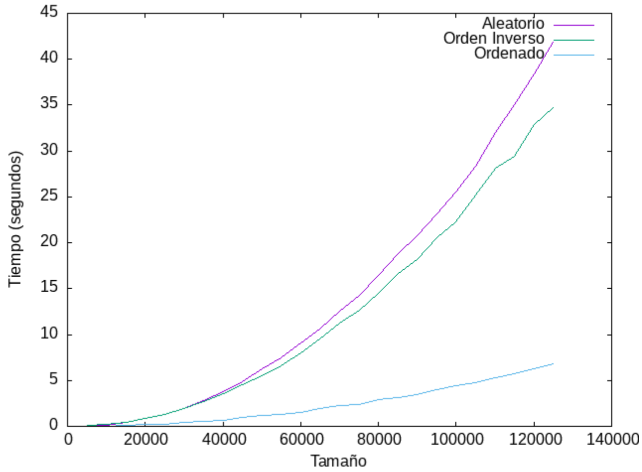

Ilustración 4: Burbuja Optimización 1

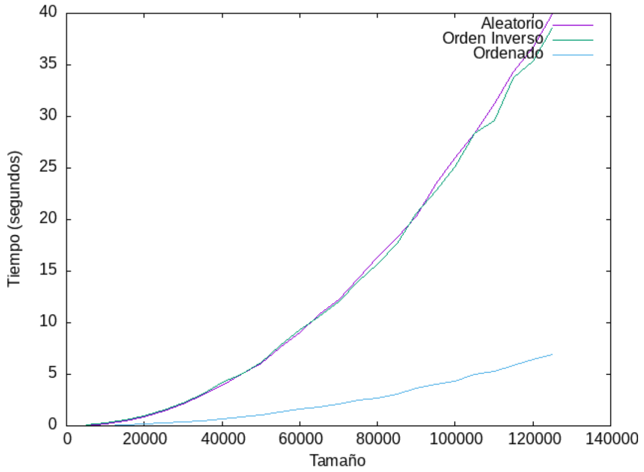

Ilustración 3: Burbuja Optimización 2

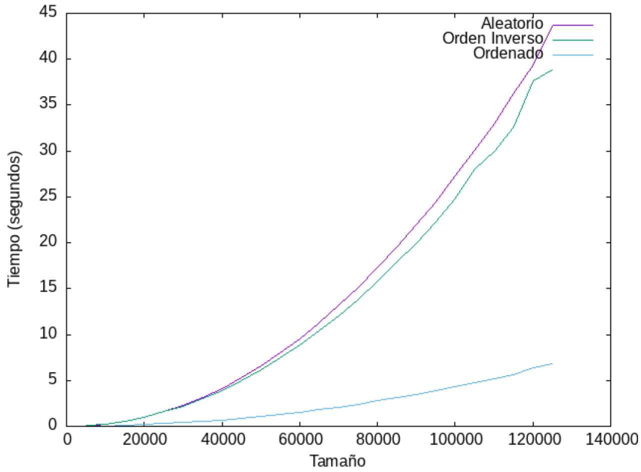

Ilustración 6: Burbuja Optimización 3

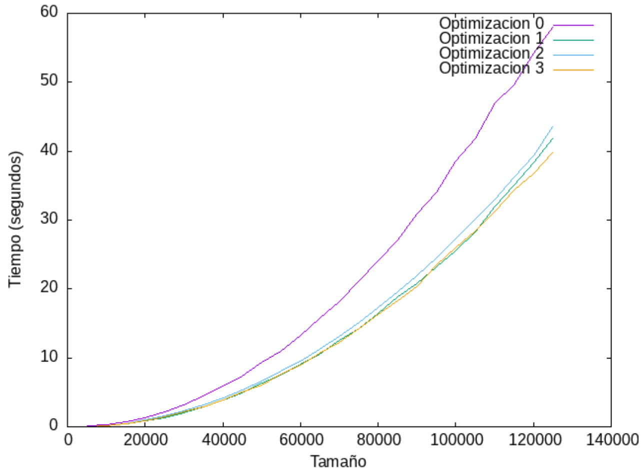

Ilustración 7: Burbuja Comparativa Optimizaciones

Los datos tomados para la comparativa de optimizaciones han sido obtenidos de los tiempos de ordenación del array aleatorio.

3.1.4. Estudio Híbrido

Función de aproximación por regresión mediante mínimos cuadrados:

$$f(x) = a_2x^2 + a_1x + a_0$$

$$a_2 = 2.84998 * 10^{-9}$$

$$a_1 = -2.56231 * 10^{-5}$$

$$a_0 = 0.205844$$

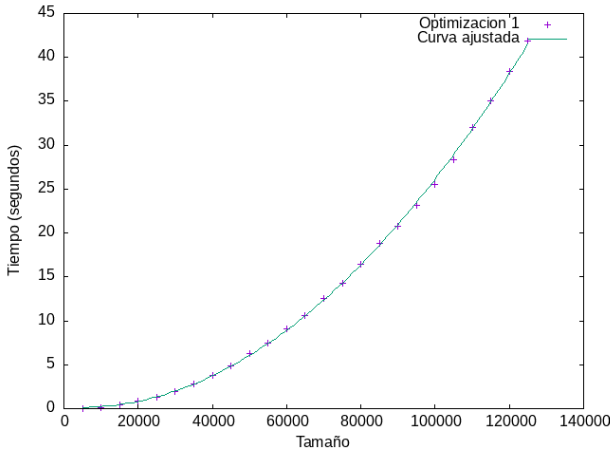

Ilustración 8: Ajuste Burbuja

Este cálculo ha sido realizado y representado por el software *gnuplot.* Los datos para los puntos han sido tomados del caso promedio (Vector Aleatorio) en optimización de nivel 1, tomada como representativa del resto de casos.

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

iter      chisq       delta/lim  lambda   a2            a1            a0
   0 1.3460545246e+21   0.00e+00  4.24e+09    1.000000e+00   1.000000e+00   1.000000e+00
  13 1.3078863703e+00  -2.57e-04  4.24e-04    2.849981e-09  -2.562315e-05   2.058435e-01

After 13 iterations the fit converged.
final sum of squares of residuals : 1.30789
rel. change during last iteration : -2.56814e-09

degrees of freedom    (FIT_NDF)                        : 22
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 0.243822
variance of residuals (reduced chisquare) = WSSR/ndf   : 0.0594494

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a2              = 2.84998e-09      +/- 4.204e-11    (1.475%)
a1              = -2.56231e-05     +/- 5.63e-06     (21.97%)
a0              = 0.205844         +/- 0.1588       (77.16%)

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

Ilustración 9: Informe del ajuste

Vemos como la función se ajusta fielmente a las mediciones tomadas además de obtener una varianza residual muy próxima a 0, luego podemos considerarlo un ajuste correcto.

3.1.5. Variaciones por el entorno

Para comprobar como afectaba el entorno de trabajo a los tiempos, tomamos medidas en distintos equipos (En optimización 0 y con caso promedio), obteniendo como resultados:

Tamaño	*PC_1*	*PC_2*	*PC_3*
5000	0.0760818	0.0771543	0.0430364
10000	0.261763	0.294051	0.184282
15000	0.669641	0.692832	0.583024
20000	1.30734	1.23732	1.28276
25000	2.16812	1.98864	2.01361
30000	3.14369	3.03978	2.9054
35000	4.41686	4.53378	4.0952
40000	5.89326	6.12376	5.42807
45000	7.39177	7.95191	7.30065
50000	9.36771	10.1649	9.21703
55000	11.0047	12.6919	11.6271
60000	13.2624	15.1954	13.1357
65000	15.6777	17.9696	16.147
70000	18.2346	21.1322	18.7456
75000	21.2536	25.0659	21.8062
80000	24.065	28.5086	25.0042
85000	27.1762	32.1271	28.2209
90000	30.8576	36.3825	31.7833
95000	34.0088	40.3417	35.9
100000	38.4724	45.364	40.1009
105000	41.8815	49.9794	44.1694
110000	47.0856	54.6643	55.6845
115000	49.5649	60.2659	64.8587
120000	54.2223	65.7783	66.7701
125000	57.9138	71.303	68.6423

Tabla 5: Burbuja Comparativa PCs

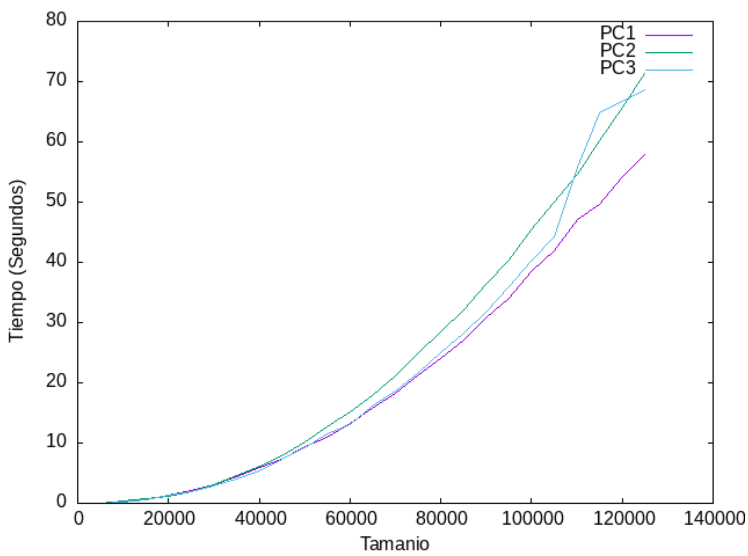

Ilustración 11: Burbuja Comparativa PCs

Vemos como en el PC_1 el algoritmo tarda menos en realizar la ordenación. Esto se puede deber a una mayor frecuencia de trabajo del procesador con respecto a los demás.

3.1.6. Conclusiones

En el algoritmo Burbuja vemos que en el caso promedio, vector aleatorio, tarda prácticamente el mismo tiempo en completarse la ordenación que en el peor caso, orden inverso. Claramente, cuando el vector ya se encuentra ordenado, vemos que los tiempos son bastante menores al resto de los casos.

En cuanto a las distintas condiciones de ejecución, notamos gran diferencia entre el nivel 0 de optimización y el 1, aunque ya este se asemeja mucho al 2 y al 3. Con respecto a la ejecución en los distintos equipos, los tiempos son muy parecidos.

3.2. ALGORITMO DE INSERCIÓN

3.2.1. Funcionamiento

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

El algoritmo de ordenación por inserción intercambia dentro del vector elementos adyacentes desde un punto dentro de este hasta llegar al principio, iterando este punto desde la posición 0 hasta el final del vector.

Las medidas en este algoritmo las hemos tomado con tamaño de *array* desde 5000 hasta 125000, con un orden inicial aleatorio, inverso y directo, y con optimización 0, 1, 2 y 3.

3.2.2. Tablas de datos

Tamaño	*Aleatorio*	*Inverso*	*Ordenado*
5000	0.0673577	0.130629	1.2162e-05
10000	0.263559	0.526625	2.4146e-05
15000	0.592082	1.17271	3.5558e-05
20000	1.06394	2.05688	4.7311e-05
25000	1.59932	3.26541	9.8456e-05
30000	2.83534	4.76974	7.2236e-05
35000	3.34404	6.30003	8.2622e-05
40000	4.30833	8.52164	9.617e-05
45000	5.5427	11.0907	0.000106238
50000	6.62608	13.4863	0.000124932
55000	8.17483	16.719	0.000248021
60000	9.69711	18.8389	0.000144566
65000	11.221	22.054	0.00015348
70000	12.9613	28.3362	0.000168195
75000	16.6684	34.273	0.000177071
80000	16.9381	34.5155	0.000229816
85000	19.0926	37.2857	0.000207024
90000	21.1028	42.0716	0.000212557
95000	24.0913	47.9039	0.000224663
100000	28.553	62.562	0.000236298
105000	28.4127	58.2402	0.000252663
110000	32.2996	63.6549	0.000268949
115000	35.4064	69.1222	0.000306032
120000	37.0341	74.6254	0.000294545
125000	41.3078	80.9875	0.000759228

Tabla 6: Inserción Optimización 0

Tamaño	*Aleatorio*	*Inverso*	*Ordenado*
5000	0.0381453	0.0773585	7.704e-06
10000	0.151744	0.316029	2.759e-05
15000	0.36568	0.732746	2.288e-05
20000	0.613791	1.23059	3.5829e-05
25000	1.0669	1.92794	3.8088e-05
30000	1.40313	2.81776	4.5692e-05
35000	2.05854	3.85498	5.3413e-05
40000	2.54247	5.04074	6.0867e-05
45000	3.23313	6.58667	6.8447e-05
50000	3.92025	7.96441	7.6203e-05
55000	4.82667	9.74306	8.3757e-05
60000	5.66463	11.4022	9.1243e-05
65000	6.98166	13.1523	9.895e-05
70000	7.66246	15.3019	0.000106498
75000	8.90926	17.6578	0.000114158
80000	9.96837	20.0911	0.00012176
85000	11.2535	22.7533	0.000129481
90000	12.7644	25.6878	0.000137806
95000	14.2856	28.399	0.00014509
100000	15.6472	31.448	0.000125516
105000	15.6072	33.9745	0.000162303
110000	18.5386	37.2921	0.000167336
115000	19.8135	42.084	0.000174823
120000	22.8284	45.2623	0.000368618
125000	24.151	48.9488	0.000196084

Tabla 7: Inserción Optimización 1

Tamaño	*Aleatorio*	*Inverso*	*Ordenado*
5000	0.0380846	0.0769616	7.712e-06
10000	0.153339	0.313923	1.5291e-05
15000	0.349472	0.6918	2.2921e-05
20000	0.605555	1.15854	3.0482e-05
25000	0.934463	1.82446	3.5911e-05
30000	1.27499	2.70351	7.4274e-05
35000	1.71059	3.47616	4.883e-05
40000	2.39306	4.65021	5.5789e-05
45000	3.00063	6.06235	6.455e-05
50000	3.66336	7.49765	8.435e-05
55000	4.38118	8.83538	7.8833e-05
60000	5.33961	10.7152	8.8789e-05
65000	6.56116	12.8537	0.000105155
70000	7.38273	14.9508	9.7555e-05
75000	8.15558	15.944	0.000104537
80000	9.0596	18.5926	0.000118152
85000	10.9542	21.0645	0.000129216
90000	12.05	23.3161	0.000125922
95000	13.1709	26.4164	0.000142508
100000	14.7272	29.2221	0.000143503
105000	15.5198	31.4567	0.000146544
110000	16.9872	35.2467	0.000157667
115000	19.1195	37.9686	0.000156054
120000	21.2531	41.6568	0.000167401
125000	21.782	42.6936	0.000179383

Tabla 8: Inserción Optimización 2

Tamaño	*Aleatorio*	*Inverso*	*Ordenado*
5000	0.0324245	0.0653299	7.281e-06
10000	0.128437	0.259974	1.4436e-05
15000	0.296193	0.584307	2.1575e-05
20000	0.513704	1.00103	2.7947e-05
25000	0.762724	1.5895	3.4903e-05
30000	1.11559	2.21077	4.0737e-05
35000	1.47089	3.02769	4.7602e-05
40000	1.96689	3.95952	5.4306e-05
45000	2.47166	4.94701	6.1055e-05
50000	3.04926	6.18141	6.7822e-05
55000	3.75229	7.57688	7.6687e-05
60000	4.41147	8.96899	8.1483e-05
65000	5.26999	10.5552	9.3194e-05
70000	6.1177	12.228	9.496e-05
75000	6.88584	13.9678	0.000104545
80000	8.12243	15.827	0.00010632
85000	8.9639	17.7705	0.000118466
90000	9.87041	20.0405	0.000125432
95000	11.1665	22.2525	0.000128959
100000	12.3772	23.99	0.000132092
105000	13.427	27.6778	0.00013878
110000	14.6818	29.1069	0.000141802
115000	15.885	31.9773	0.000151948
120000	16.5803	34.5185	0.000158413
125000	19.1131	37.927	0.000165015

Tabla 9: Inserción Optimización 3

3.2.3. Gráficas

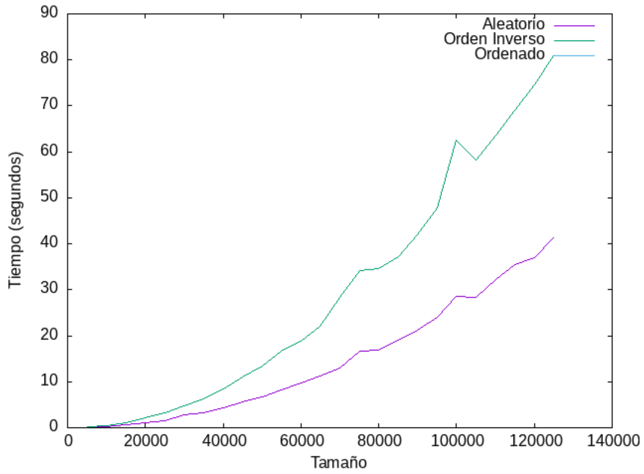

Ilustración 12: Inserción Optimización 0

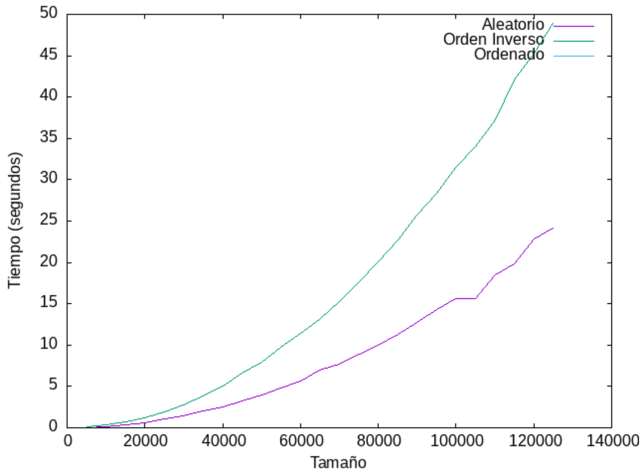

Ilustración 13: Inserción Optimización 1

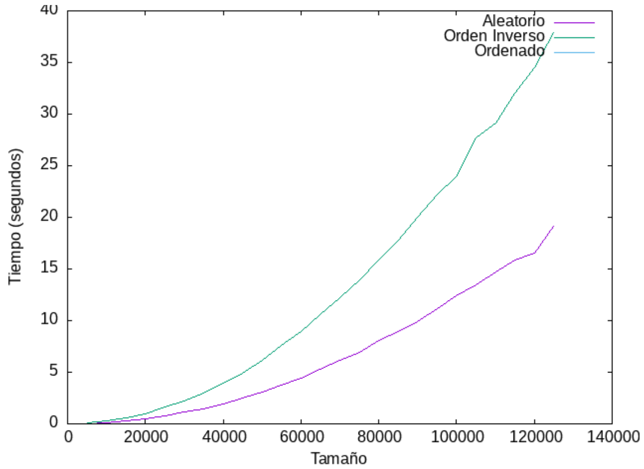

Ilustración 14: Inserción Optimización 2

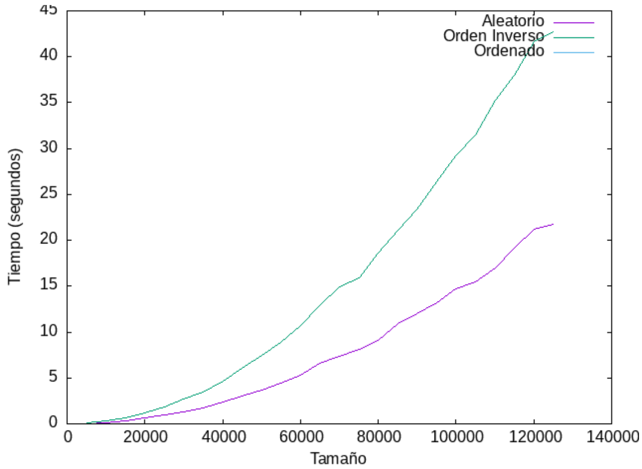

Ilustración 15: Inserción Optimización 3

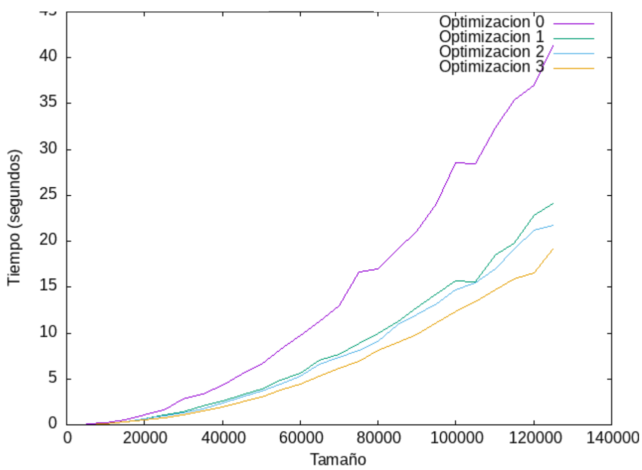

Ilustración 16: Inserción Comparativa Optimizaciones

Los datos tomados para la comparativa de optimizaciones han sido obtenidos de los tiempos de ordenación del array aleatorio.

3.2.4. Estudio Híbrido

Función de aproximación por regresión mediante mínimos cuadrados

$$f(x) = a_2x^2 + a_1x + a_0$$

$$a_2 = 1.47783 * 10^{-9}$$

$$a_1 = 7.18879 * 10^{-6}$$

$$a_0 = -0.0604146$$

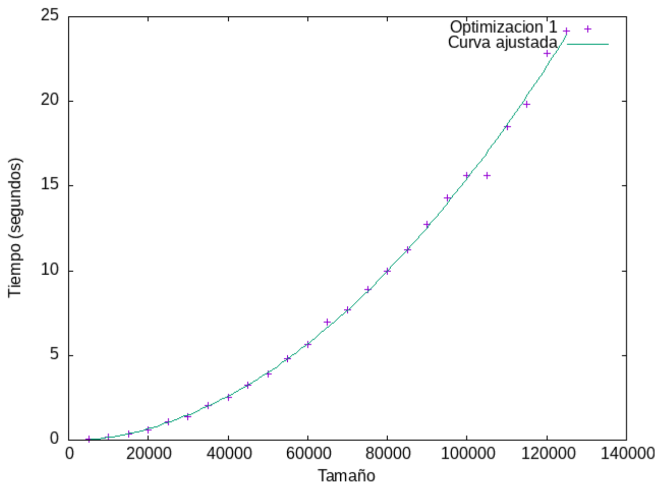

Ilustración 17: Inserción Ajuste

Este cálculo ha sido realizado y representado por el software *gnuplot.* Los datos para los puntos han sido tomados del caso promedio (Vector Aleatorio) en optimización de nivel 1, tomada como representativa del resto de casos.

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

iter      chisq       delta/lim  lambda   a2            a1            a0
   0 1.3460545275e+21   0.00e+00  4.24e+09    1.000000e+00   1.000000e+00   1.000000e+00
  13 3.1050243886e+00  -1.93e-04  4.24e-04    1.477827e-09   7.188786e-06  -6.041462e-02

After 13 iterations the fit converged.
final sum of squares of residuals : 3.10502
rel. change during last iteration : -1.92874e-09

degrees of freedom    (FIT_NDF)                        : 22
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 0.375683
variance of residuals (reduced chisquare) = WSSR/ndf   : 0.141137

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a2              = 1.47783e-09      +/- 6.478e-11    (4.383%)
a1              = 7.18879e-06      +/- 8.675e-06    (120.7%)
a0              = -0.0604146       +/- 0.2447       (405.1%)

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

Ilustración 18: Informe del ajuste

Vemos como la función se ajusta fielmente a las mediciones tomadas además de obtener una varianza residual muy próxima a 0, luego podemos considerarlo un ajuste correcto.

3.2.5. Variaciones por el entorno

Para comprobar como afectaba el entorno de trabajo a los tiempos, tomamos medidas en distintos equipos (En optimización 0 y con caso promedio), obteniendo como resultados:

Tamaño	*PC_1*	*PC_2*	*PC_3*
5000	0.0673577	0.0571364	0.0490437
10000	0.263559	0.229088	0.132032
15000	0.592082	0.508476	0.338985
20000	1.06394	0.896883	0.60957
25000	1.59932	1.42484	1.13757
30000	2.83534	2.2176	1.69049
35000	3.34404	2.78634	2.18683
40000	4.30833	3.61037	3.10263
45000	5.5427	4.65293	3.46879
50000	6.62608	5.70683	4.3163
55000	8.17483	6.83322	5.60247
60000	9.69711	8.38875	6.50437
65000	11.221	9.66845	8.36227
70000	12.9613	11.3949	9.94199
75000	16.6684	12.9792	11.0073
80000	16.9381	14.6035	12.8113
85000	19.0926	16.7615	14.6667
90000	21.1028	18.6198	16.3277
95000	24.0913	20.9888	18.1402
100000	28.553	22.8567	18.7708
105000	28.4127	25.1165	20.6123
110000	32.2996	27.6038	23.4941
115000	35.4064	29.9538	25.5847
120000	37.0341	32.8653	29.8773
125000	41.3078	35.6197	32.4186

Tabla 10: Inserción Comparativa PCs

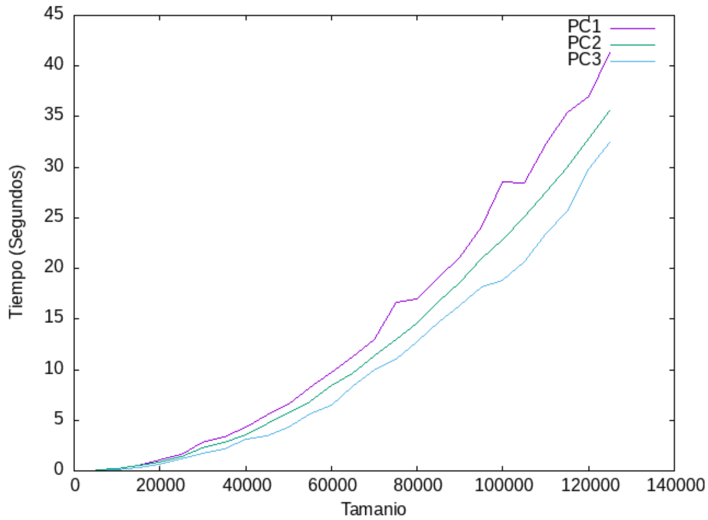

Ilustración 19: Inserción Comparativa PCs

Vemos como en el PC_1 el algoritmo tarda menos en realizar la ordenación. Esto se puede deber a una mayor frecuencia de trabajo del procesador con respecto a los demás.

3.2.6. Conclusiones

En este algoritmo, observamos que los tiempos de ordenación en el caso de que el vector ya se encuentre ordenado son prácticamente nulos, quedando muy claro que esta es la situación óptima. Por otro lado, podemos apreciar que existe una notable diferencia entre los tiempos de ordenación entre el caso de un *array* aleatorio y ordenado inversamente, demostrando que este último es el peor caso posible.

Sobre los efectos de los niveles de optimización en los tiempos, al igual que en el algoritmo anterior vemos que existe gran diferencia entre el nivel 0 y el 1. Sin embargo, este último con respecto a los 2 superiores no provoca una variación muy notable, llegando a mediciones en las que el nivel 1 consigue tiempos por debajo del nivel 2.

En cuanto al efecto del *hardware* en los tiempos, aunque quedan bien distinguidas las tres tendencias distintas seguidas por cada uno de los equipos, tampoco existe gran diferencia entre los resultados conseguidos con cada uno.

3.3. Algoritmo de Selección

3.3.1. Funcionamiento

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

El algoritmo de ordenación por selección trabaja dando tantas iteraciones como elementos a ordenar menos uno, en las cuales busca entre los elementos que restan por ordenar el menor, para al terminar ponerlo el primero de entre estos.

Las medidas en este algoritmo las hemos tomado con tamaño de *array* desde 5000 hasta 125000, con un orden inicial aleatorio, inverso y directo, y con optimización 0, 1, 2 y 3.

3.3.2. Tablas de datos

Tamaño	*Aleatorio*	*Inverso*	*Ordenado*
5000	0.032361	0.026605	0.018091
10000	0.074488	0.105648	0.072414
15000	0.164915	0.2377	0.16311
20000	0.291612	0.422546	0.290619
25000	0.457656	0.670484	0.472765
30000	0.6722	0.998005	0.709869
35000	0.961544	1.79805	0.922557
40000	1.21988	1.72965	1.23057
45000	1.55356	2.24453	1.47859
50000	1.84865	2.73864	1.84334
55000	2.30097	3.29225	2.22437
60000	2.67406	3.83745	2.71247
65000	3.15777	4.48472	3.14858
70000	3.5868	5.91433	3.66746
75000	4.30502	6.16093	4.31885
80000	4.96642	7.32207	4.97218
85000	5.838	7.99891	5.41323
90000	6.26267	9.22997	6.02862
95000	6.98271	10.5714	7.03682
100000	7.4498	11.1717	7.56368
105000	8.83641	12.0719	8.02582
110000	9.11824	13.3631	9.0508
115000	9.99222	14.1861	9.8971
120000	10.4566	15.8986	10.9143
125000	12.2159	18.0949	12.083

Tabla 11: Inserción Optimización 0

Tamaño	*Aleatorio*	*Inverso*	*Ordenado*
5000	0.008665	0.017281	0.008331
10000	0.034776	0.080437	0.032363
15000	0.07642	0.152679	0.073434
20000	0.144906	0.270956	0.141074
25000	0.222441	0.472865	0.208605
30000	0.296651	0.621114	0.305193
35000	0.42301	0.824799	0.429626
40000	0.561028	1.08618	0.552914
45000	0.747739	1.43376	0.675141
50000	0.8197	1.69702	0.835982
55000	1.04089	2.0827	1.0014
60000	1.17841	2.47914	1.29176
65000	1.56307	2.98587	1.36636
70000	1.6393	3.32946	1.75161
75000	1.99714	4.17878	1.8525
80000	2.3589	4.60504	2.60125
85000	2.53203	5.03534	2.45208
90000	2.75904	5.82143	2.69043
95000	2.9734	6.03833	2.93386
100000	3.42224	6.74369	3.29113
105000	3.96374	7.66148	3.68769
110000	4.00597	8.49443	3.92852
115000	4.40004	9.06123	4.31539
120000	4.65579	9.62683	4.69736
125000	5.40392	10.2729	5.05662

Tabla 12: Inserción Optimización 1

Tamaño	*Aleatorio*	*Inverso*	*Ordenado*
5000	0.010645	0.017917	0.008693
10000	0.040651	0.071207	0.033834
15000	0.084491	0.154781	0.074073
20000	0.144154	0.26818	0.128417
25000	0.222476	0.444307	0.226442
30000	0.316887	0.596677	0.316916
35000	0.450891	0.809678	0.392737
40000	0.563015	1.09676	0.553366
45000	0.687642	1.32735	0.648974
50000	0.840833	1.65012	0.840123
55000	1.02832	2.05318	1.00772
60000	1.25197	2.45403	1.22459
65000	1.46007	2.81113	1.38289
70000	1.62491	3.29278	1.72256
75000	1.88597	3.71099	1.80672
80000	2.2277	4.28892	2.18393
85000	2.44788	4.82883	2.36026
90000	2.79057	5.40475	2.72223
95000	3.05588	6.02442	2.8958
100000	3.29206	6.77276	3.39427
105000	3.73728	7.62951	3.61388
110000	4.0541	8.01878	4.05729
115000	4.6666	8.7784	4.27851
120000	4.77594	9.98307	4.8294
125000	5.59639	10.6391	5.03243

Tabla 13: Inserción Optimización 2

Tamaño	*Aleatorio*	*Inverso*	*Ordenado*
5000	0.011056	0.017443	0.008089
10000	0.043306	0.067241	0.032646
15000	0.093555	0.151258	0.072444
20000	0.174789	0.28262	0.134173
25000	0.265553	0.461164	0.202685
30000	0.38067	0.665985	0.340084
35000	0.517218	0.892789	0.422763
40000	0.675711	1.17424	0.531352
45000	0.834972	1.36297	0.695902
50000	1.03277	1.7932	0.865184
55000	1.20413	2.12558	1.05623
60000	1.4753	2.46032	1.20223
65000	1.73993	2.97958	1.46595
70000	2.10904	3.50351	1.74149
75000	2.26959	3.91643	1.87681
80000	2.5843	4.35726	2.19187
85000	2.9258	4.88557	2.40528
90000	3.49419	5.68792	2.77488
95000	3.57655	6.14637	3.0906
100000	4.0049	7.06806	3.33198
105000	4.47093	7.93429	3.78251
110000	4.9406	8.39477	4.2745
115000	5.8399	9.27117	4.8454
120000	6.44988	10.0604	4.85334
125000	6.45372	10.8695	5.35744

Tabla 14: Inserción Optimización 3

3.3.3. Gráficas

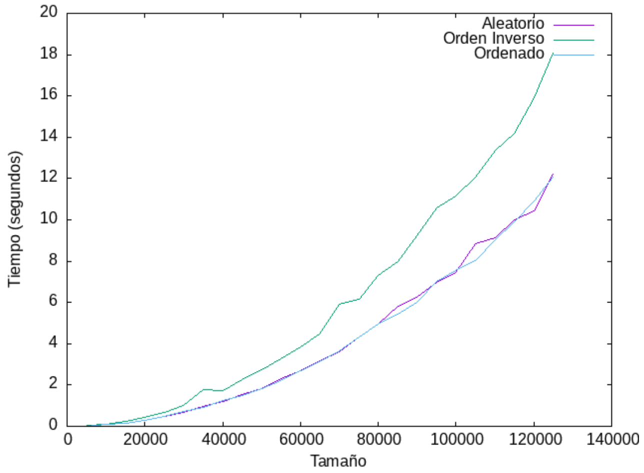

Ilustración 20: Selección Optimización 0

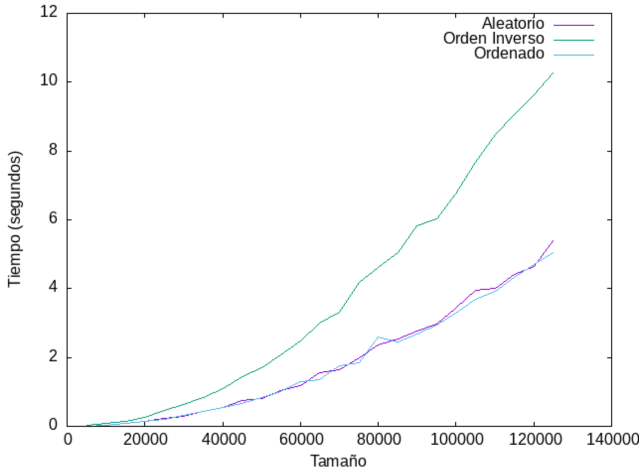

Ilustración 21: Selección Optimización 1

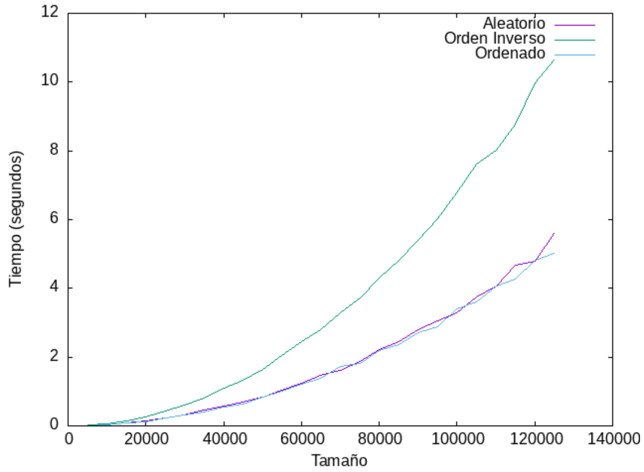

Ilustración 22: Selección Optimización 2

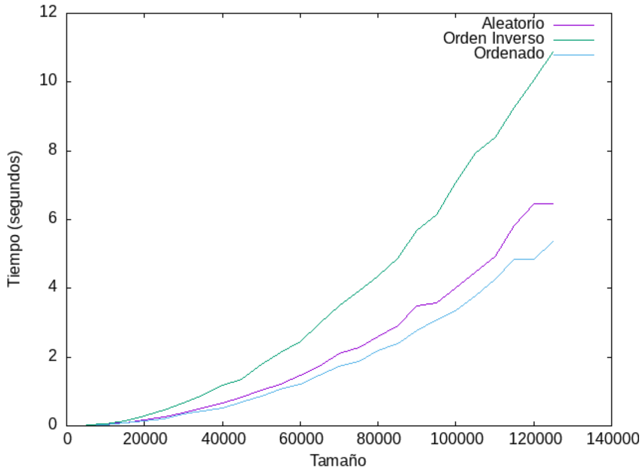

Ilustración 23: Selección Optimización 3

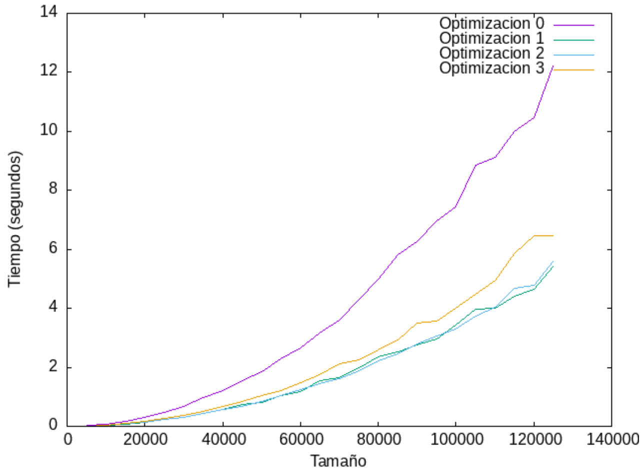

Ilustración 24: Selección Comparativa Optimizaciones

Los datos tomados para la comparativa de optimizaciones han sido obtenidos de los tiempos de ordenación del array aleatorio.

3.3.4. Estudio Híbrido

Función de aproximación por regresión mediante mínimos cuadrados

$$f(x) = a_2x^2 + a_1x + a_0$$

$$a_2 = 3.16898 * 10^{-10}$$

$$a_1 = 2.77102 * 10^{-6}$$

$$a_0 = -0.0418065$$

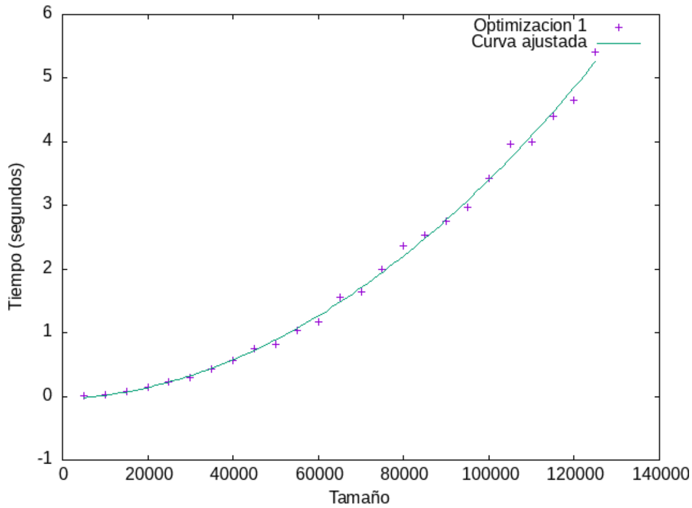

Ilustración 25: Selección Ajuste

Este cálculo ha sido realizado y representado por el software *gnuplot.* Los datos para los puntos han sido tomados del caso promedio (Vector Aleatorio) en optimización de nivel 1, tomada como representativa del resto de casos.

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

iter      chisq       delta/lim  lambda   a2            a1            a0
   0 1.3460545307e+21   0.00e+00  4.24e+09    1.000000e+00   1.000000e+00   1.000000e+00
  13 1.9068519898e-01  -3.03e-03  4.24e-04    3.168976e-10   2.771017e-06  -4.180654e-02

After 13 iterations the fit converged.
final sum of squares of residuals : 0.190685
rel. change during last iteration : -3.0314e-08

degrees of freedom    (FIT_NDF)                        : 22
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 0.0930995
variance of residuals (reduced chisquare) = WSSR/ndf   : 0.00866751

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a2              = 3.16898e-10      +/- 1.605e-11    (5.065%)
a1              = 2.77102e-06      +/- 2.15e-06     (77.58%)
a0              = -0.0418065       +/- 0.06065      (145.1%)

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

Ilustración 26: Informe del ajuste

Vemos como la función se ajusta fielmente a las mediciones tomadas además de obtener una varianza residual muy próxima a 0, luego podemos considerarlo un ajuste correcto.

3.3.5. Variaciones por el entorno

Para comprobar como afectaba el entorno de trabajo a los tiempos, tomamos medidas en distintos equipos (En optimización 0 y con caso promedio), obteniendo como resultados:

Tamaño	*PC_1*	*PC_2*	*PC_3*
5000	5000	0.02	0.021
10000	10000	0.07	0.08
15000	15000	0.16	0.18
20000	20000	0.29	0.313
25000	25000	0.45	0.533
30000	30000	0.62	0.701
35000	35000	0.87	0.966
40000	40000	1.14	1.272
45000	45000	1.45	1.607
50000	50000	1.96	1.945
55000	55000	2.13	2.344
60000	60000	2.53	2.86
65000	65000	2.97	3.313
70000	70000	3.45	3.882
75000	75000	4	4.464
80000	80000	4.73	5.078
85000	85000	5.35	5.745
90000	90000	6.55	6.532
95000	95000	6.43	7.215
100000	100000	7.09	8.052
105000	105000	7.95	8.847
110000	110000	8.64	9.719
115000	115000	9.48	10.674
120000	120000	10.28	11.657
125000	125000	11.13	12.696

Tabla 15: Selección Comparativa PCs

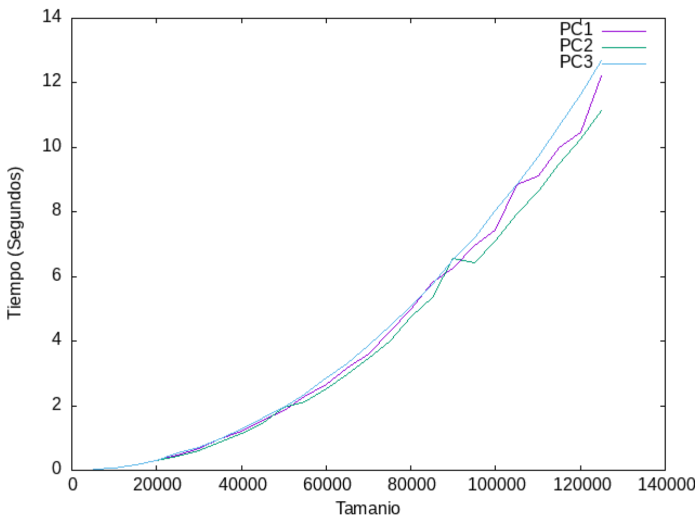

Ilustración 27: Selección Comparativa PCs

Vemos que son bastante parecidas las tendencias de las 3 máquinas.

3.3.6. Conclusiones

Al igual que los anteriores, de los datos obtenidos podemos deducir que el peor caso para este algoritmo es partir de un *array* ordenado inversamente, superando los tiempos obtenidos con este punto de partida a los obtenidos con *array* aleatorio y ya ordenado por bastante. A su vez, es curioso observar que los tiempos del caso promedio, *array* aleatorio, son muy similares a los obtenidos con este ya ordenado, de forma que se vuelve un algoritmo de ordenación muy poco eficiente si este último caso ocurre.

Como segundo punto, es fácilmente apreciable que en tamaños de *array* a partir de orden 10^6 se hace notar la diferencia de tiempos entre la compilación sin optimización y con nivel 1, 2 o 3.

Por último, vemos que en cuanto a la variación derivada de la máquina donde se ejecute el algoritmo no encontramos que sea relevante, siguiendo tendencias de crecimiento del tiempo bastante parecidas.

4. ALGORITMOS O(NLOG(N))

4.1. ALGORITMO *HEAPSORT*

4.1.1. Funcionamiento

```
static void heapsort(long duble T[], int num_elem)
{
  int i;
 for (i = num_elem/2; i >= 0; i--)
   reajustar(T, num_elem, i);
 for (i = num_elem - 1; i >= 1; i--)
   {
    Long double aux = T[0];
    T[0] = T[i];
    T[i] = aux;
    reajustar(T, i, 0);
   }
}
```

El algoritmo de ordenación *Heapsort* implementa el aprovechamiento de las propiedades de los *APO* para lograr su objetivo. Para ello, utiliza el *array* como representación del *APO*, de forma que conociendo su índice en este sabríamos la posición en la que se encuentra dentro del árbol.

Sin embargo, esto requiere de una función extra que nos ayude a modificar el *array* de forma que tenga la estructura correcta como representación del *APO.* Esta será *reajustar(long double*,int,int)*.

Las medidas en este algoritmo las hemos tomado con tamaño de *array* desde 50000 hasta 1250000, con un orden inicial aleatorio, inverso y directo, y con optimización 0, 1, 2 y 3.

4.1.2. Tablas de datos

Tamaño	*Aleatorio*	*Inverso*	*Ordenado*
50000	0.0082210	0.0055925	0.0052615
100000	0.0175497	0.0133284	0.0136809
150000	0.0278097	0.0215011	0.0204397
200000	0.0417435	0.0306006	0.028143
250000	0.0493009	0.0349704	0.0363883
300000	0.0679123	0.0456708	0.0426966
350000	0.0736743	0.0536993	0.0497608
400000	0.0970621	0.0657197	0.0582736
450000	0.102434	0.0694625	0.0642368
500000	0.117715	0.0774899	0.07187
550000	0.130632	0.0859162	0.0792039
600000	0.146503	0.0922167	0.0904423
650000	0.160031	0.102229	0.0993682
700000	0.175512	0.109806	0.104082
750000	0.190379	0.119423	0.11315
800000	0.203501	0.1282	0.124074
850000	0.222643	0.136501	0.128663
900000	0.234881	0.14677	0.137691
950000	0.260219	0.168596	0.156868
1000000	0.269484	0.164776	0.154124
1050000	0.282923	0.173648	0.161675
1100000	0.298297	0.182443	0.169039
1150000	0.317501	0.228499	0.184207
1200000	0.341845	0.204686	0.188271
1250000	0.359864	0.210935	0.196975

Tabla 16: Heapsort Optimización 0

Tamaño	*Aleatorio*	*Inverso*	*Ordenado*
50000	0.0040914	0.0036410	0.0036938
100000	0.0098428	0.0082014	0.0088365
150000	0.0149683	0.0124682	0.0122888
200000	0.0226329	0.0169146	0.0166328
250000	0.0274746	0.020309	0.0205016
300000	0.0341684	0.0250807	0.0247955
350000	0.0431581	0.0323457	0.0327017
400000	0.0559675	0.0401556	0.0413045
450000	0.0689216	0.0456963	0.047337
500000	0.0815028	0.0540672	0.0529973
550000	0.0839523	0.0556759	0.0538966
600000	0.0943883	0.0614382	0.0607467
650000	0.102136	0.066035	0.0632949
700000	0.123741	0.0738209	0.0708066
750000	0.127712	0.0784578	0.0740016
800000	0.140136	0.0851311	0.0820935
850000	0.16561	0.0896921	0.0882962
900000	0.163068	0.0935865	0.0885334
950000	0.181972	0.10326	0.0925883
1000000	0.192427	0.111753	0.111831
1050000	0.265718	0.124163	0.117601
1100000	0.243568	0.137064	0.123957
1150000	0.304103	0.135307	0.125176
1200000	0.2629	0.13635	0.12688
1250000	0.295027	0.144477	0.128414

Tabla 17: Heapsort Optimización 1

Tamaño	*Aleatorio*	*Inverso*	*Ordenado*
50000	0.0059825	0.0043091	0.0040897
100000	0.0123357	0.0090721	0.0084686
150000	0.0195808	0.0134743	0.0151682
200000	0.0285532	0.0182818	0.0172077
250000	0.0340705	0.0232152	0.0222094
300000	0.041782	0.0284683	0.0268857
350000	0.0500551	0.033842	0.0340554
400000	0.0690226	0.0399177	0.0398079
450000	0.0699629	0.0440694	0.0411309
500000	0.0760728	0.0569612	0.0492703
550000	0.0882897	0.0545628	0.0519469
600000	0.098794	0.0772514	0.0821304
650000	0.157545	0.0718827	0.0626194
700000	0.117442	0.0726376	0.0671639
750000	0.127074	0.0819397	0.074051
800000	0.138636	0.0911679	0.0784054
850000	0.155017	0.0982719	0.0866663
900000	0.168263	0.100262	0.0918154
950000	0.192503	0.112497	0.102051
1000000	0.20081	0.115894	0.106264
1050000	0.202288	0.119432	0.110261
1100000	0.211444	0.129257	0.130355
1150000	0.284289	0.149852	0.136757
1200000	0.263846	0.145101	0.138046
1250000	0.265511	0.157338	0.140004

Tabla 18: Heapsort Optimización 2

Tamaño	*Aleatorio*	*Inverso*	*Ordenado*
50000	0.0063082	0.0045009	0.0041119
100000	0.0151296	0.0108154	0.0089873
150000	0.0223347	0.0140449	0.0131021
200000	0.0282467	0.0188807	0.0168651
250000	0.0357594	0.0248455	0.0224481
300000	0.0507926	0.0354346	0.0320723
350000	0.0603104	0.0353668	0.0313904
400000	0.0620867	0.041292	0.038205
450000	0.079959	0.0482152	0.0424512
500000	0.0816131	0.0522703	0.0493969
550000	0.0846391	0.0594915	0.0543841
600000	0.103017	0.0642602	0.0559226
650000	0.114557	0.0707841	0.0629198
700000	0.131439	0.079091	0.0752991
750000	0.132689	0.0990821	0.075793
800000	0.135617	0.0830962	0.0774106
850000	0.179852	0.102238	0.0862644
900000	0.169838	0.10581	0.0938928
950000	0.180533	0.107767	0.0995909
1000000	0.1999	0.12748	0.116432
1050000	0.243233	0.147271	0.111728
1100000	0.216644	0.139114	0.123383
1150000	0.233636	0.146909	0.132733
1200000	0.245145	0.161337	0.128754
1250000	0.251515	0.156486	0.133015

Tabla 19: Heapsort Optimización 3

4.1.3. Gráficas

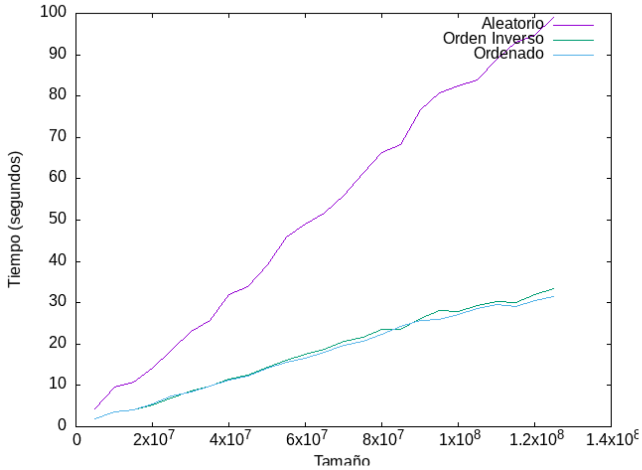

Ilustración 28: Heapsort Optimización 0

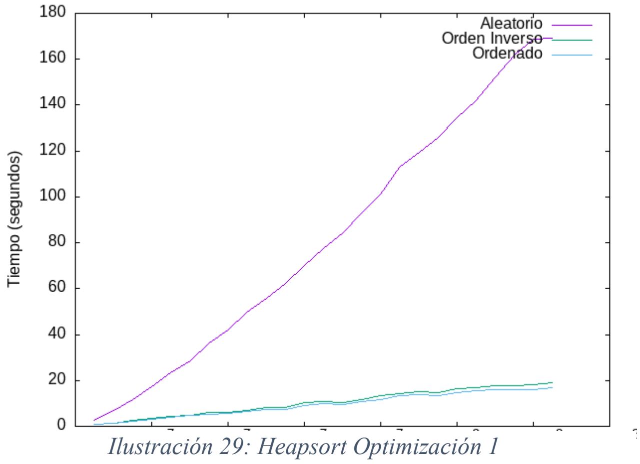

Ilustración 29: Heapsort Optimización 1

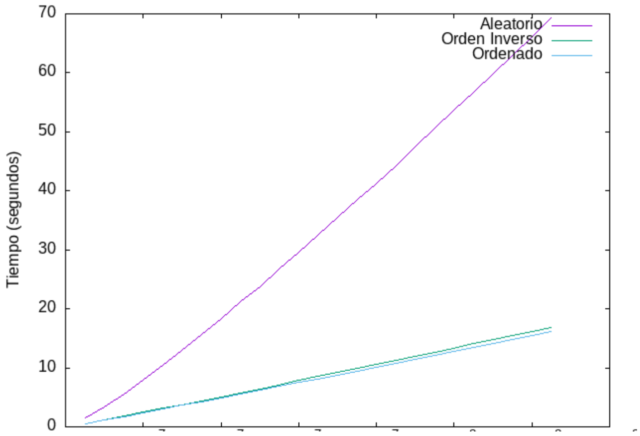

Ilustración 30: Heapsort Optimización 2

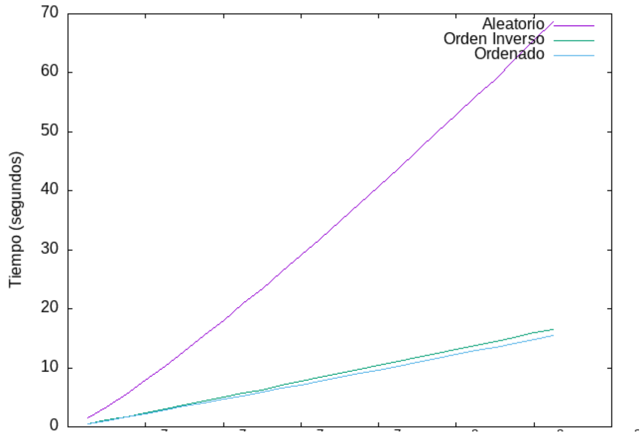

Ilustración 31: Heapsort Optimización 3

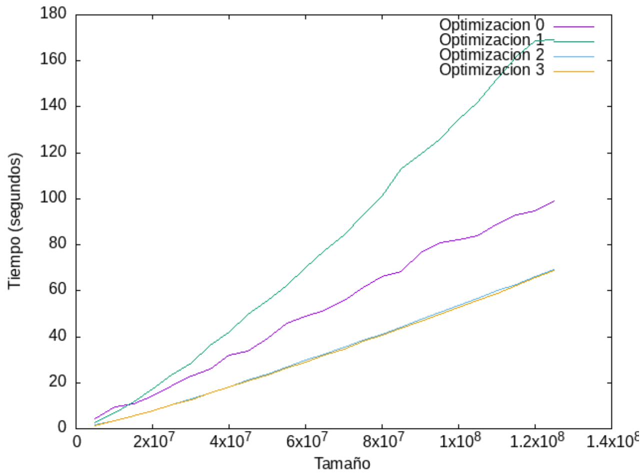

Ilustración 32: Heapsort Comparativa Optimizaciones

Los datos tomados para la comparativa de optimizaciones han sido obtenidos de los tiempos de ordenación del array aleatorio.

4.1.4. Estudio Híbrido

Función de aproximación por regresión mediante mínimos cuadrados:

$$f(x) = a_1 x log(x) + a_0$$

$$a_1 = 7.05655 * 10^{-8}$$

$$a_0 = 1$$

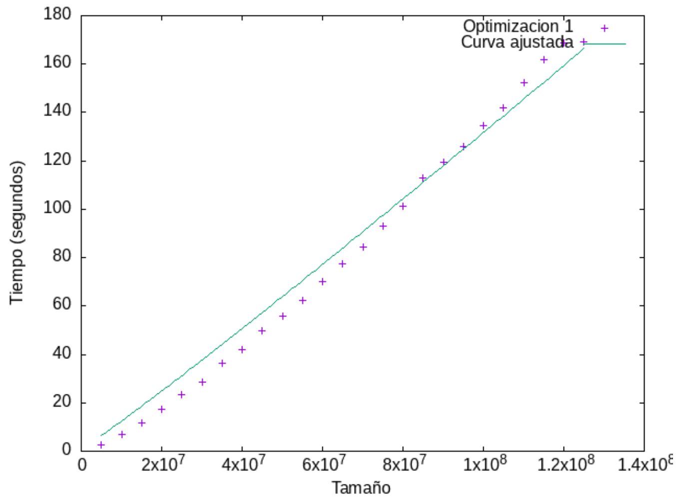

Ilustración 33: Heapsort ajuste

Este cálculo ha sido realizado y representado por el software *gnuplot.* Los datos para los puntos han sido tomados del caso promedio (Vector Aleatorio) en optimización de nivel 1, tomada como representativa del resto de casos.

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

iter      chisq       delta/lim  lambda   a1            a0
   0 4.6423675364e+19   0.00e+00  9.64e+08    1.000000e+00   1.000000e+00
   4 1.0512594684e+03  -2.72e-01  9.64e+04    7.056551e-08   1.000000e+00

After 4 iterations the fit converged.
final sum of squares of residuals : 1051.26
rel. change during last iteration : -2.71541e-06

degrees of freedom    (FIT_NDF)                        : 23
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 6.76069
variance of residuals (reduced chisquare) = WSSR/ndf   : 45.7069

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a1              = 7.05655e-08      +/- 1.987e-09    (2.816%)
a0              = 1                +/- 2.708        (270.8%)

correlation matrix of the fit parameters:
                a1     a0
a1              1.000
a0             -0.866  1.000
```

Ilustración 34: Informe del ajuste

Vemos que la función no ajusta del todo bien los datos tomados en las mediciones, y menos con una función logarítmica, asimilándose más bien a una lineal. Aunque cada medida sea resultado de hacer la media de 5 datos, se da una incongruencia entre el resultado de la gráfica y el estudio teórico del algoritmo, con lo cual se deberá a factores que no hemos controlado durante las ejecuciones.

4.1.5. Variaciones por el entorno

Para comprobar como afectaba el entorno de trabajo a los tiempos, tomamos medidas en distintos equipos (En optimización 0 y con caso promedio), obteniendo como resultados:

Tamaño	*PC_1*	*PC_2*	*PC_3*
5000	0.0082210	0.0117554	0.0089992
10000	0.0175497	0.0253525	0.0159986
15000	0.0278097	0.0390139	0.0260387
20000	0.0417435	0.0533736	0.0350326
25000	0.0493009	0.0677763	0.0440385
30000	0.0679123	0.084044	0.0660378
35000	0.0736743	0.0983905	0.0660436
40000	0.0970621	0.115259	0.0750301
45000	0.102434	0.131474	0.0879459
50000	0.117715	0.146794	0.0989956
55000	0.130632	0.162912	0.111043
60000	0.146503	0.17928	0.146847
65000	0.160031	0.195917	0.195013
70000	0.175512	0.212789	0.170999
75000	0.190379	0.235449	0.208384
80000	0.203501	0.246528	0.190754
85000	0.222643	0.263232	0.214377
90000	0.234881	0.285555	0.204795
95000	0.260219	0.299172	0.232564
100000	0.269484	0.318874	0.235178
105000	0.282923	0.339873	0.273845
110000	0.298297	0.361981	0.273997
115000	0.317501	0.374983	0.323564
120000	0.341845	0.404426	0.306433
125000	0.359864	0.418618	0.330006

Tabla 20: Heapsort Comparativa PCs

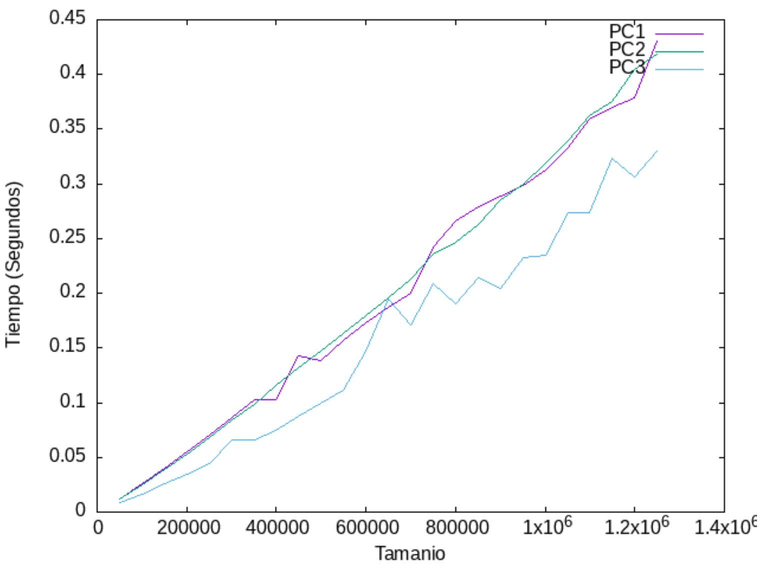

Ilustración 35: Heapsort Comparativa PCs

4.1.6. Conclusiones

Encontramos como novedad que con este algoritmo el peor caso pasa a ser el *array* aleatorio, consiguiendo tiempos mucho más altos el código en estas situaciones que con el vector ya ordenado, ya sea inversamente o directamente. Además, es curioso como estos dos últimos consiguen resultados muy similares.

Se da a su vez otro hecho inusual con este algoritmo y es la obtención de tiempos bastante superiores con optimización 1 que sin optimización. Aunque dependiendo de cómo el compilador haya generado el código de ensamblado y el ejecutable podría llegar a ser factible que ocurriera eso, también puede deberse a la ejecución de algún proceso muy demandante simultaneo a la toma de tiempos del nivel 1 de optimización.

Por otro lado, en cuanto a los efectos del *hardware*, vemos que se obtienen resultados muy parecidos en los 3 equipos, aún con una pequeña diferencia entre el ordenador 1 y 2 con el 3, consiguiendo este último bajar un poco los tiempos de ejecución.

4.2. ALGORITMO MERGESORT

4.2.1. Funcionamiento

```
static void mergesort_lims(long double T[], int inicial, int final){
    if (final - inicial < UMBRAL_MS) {
            insercion_lims(T, inicial, final);
    }
    else {
            int k = (final - inicial)/2;
            long double * U = new long double [k - inicial + 1];
            assert(U);
            int l, l2;
            for (l = 0, l2 = inicial; l < k; l++, l2++)
                    U[l] = T[l2];
            U[l] = LLONG_MAX;
            long double * V = new long double [final - k + 1];
            assert(V);
            for (l = 0, l2 = k; l < final - k; l++, l2++)
                    V[l] = T[l2];
            V[l] = LLONG_MAX;
            mergesort_lims(U, 0, k);
            mergesort_lims(V, 0, final - k);
            fusion(T, inicial, final, U, V);
            delete [] U;
            delete [] V;
    }
}
```

Este algoritmo de ordenación resulta de mejorar uno de los más rápidos de los de orden *O(n^2)*, el de inserción. Gracias a llamadas recursivas a si mismo, el código divide el *array* en dos conjuntos más pequeños donde vuelve a ejecutarse hasta llegar a vectores de tamaño por debajo de *UMBRAL_MS*, que por defecto está establecido en 50. Una vez llegado a este tamaño, se ejecuta el algoritmo de inserción para ordenarlo.

Para el correcto funcionamiento de este algoritmo es necesaria una función que, tomando dos *arrays* ordenados, sea capaz de juntarlos manteniendo el orden. Esta será *fusion()*.

Como las gráficas con este algoritmo salían demasiado inestables, hemos aumentado el tamaño de los *arrays* a ordenar, de manera que llegan desde $5*10^6$ elementos hasta $1.25*10^8$ elementos.

4.2.2. Tablas de datos

Tamaño	*Aleatorio*	*Inverso*	*Ordenado*
5000000	1.92359	2.66085	0.754551
10000000	3.83117	5.08973	1.55595
15000000	5.151	6.97505	2.16223
20000000	7.93284	9.69437	2.90908
25000000	10.5608	14.8246	3.95038
30000000	11.0974	14.2811	5.08943
35000000	14.6724	15.9306	5.7597
40000000	16.6203	20.4292	6.31771
45000000	20.0223	24.7834	7.09398
50000000	21.8151	29.1064	7.97203
55000000	21.0519	25.6	9.27994
60000000	22.7875	28.8784	10.0935
65000000	26.2845	32.2974	10.8572
70000000	27.7851	32.1386	13.4971
75000000	32.6026	39.5873	13.6118
80000000	34.895	45.025	14.4823
85000000	38.4307	47.0186	14.9797
90000000	39.1796	52.1172	16.8499
95000000	43.2019	55.7575	16.3085
100000000	46.6023	60.5634	17.5507
105000000	40.9785	49.768	18.7538
110000000	42.7985	53.3108	20.1956
115000000	48.059	57.722	20.8898
120000000	48.95	61.3017	20.6237
125000000	50.6052	64.6533	21.998

Tabla 21: Mergesort Optimización 0

Tamaño	*Aleatorio*	*Inverso*	*Ordenado*
5000000	1.24394	1.36077	0.485799
10000000	2.97105	2.80461	1.19792
15000000	3.66706	3.32507	1.49397
20000000	5.37249	5.46495	2.09574
25000000	7.31048	7.7336	2.61284
30000000	8.11314	7.31765	3.45118
35000000	9.93158	10.4195	4.1545
40000000	11.3261	11.2568	4.44263
45000000	12.8885	14.5799	5.44262
50000000	14.8934	16.5428	6.11078
55000000	15.1264	12.6353	6.36185
60000000	16.7589	15.3987	6.89499
65000000	17.4735	16.7677	7.90003
70000000	20.3064	18.9939	8.88659
75000000	22.2123	22.3415	10.3401
80000000	25.9045	25.2557	11.1468
85000000	27.3573	26.7582	11.1586
90000000	27.5074	28.4312	11.8732
95000000	30.3136	32.9707	12.2064
100000000	32.8419	34.8087	13.0554
105000000	29.2555	24.4822	12.6386
110000000	30.266	26.5004	13.3713
115000000	31.1737	28.0288	14.6062
120000000	33.9927	30.5976	14.8325
125000000	35.9769	32.9396	16.0397

Tabla 22: Mergesort Optimización 1

Tamaño	*Aleatorio*	*Inverso*	*Ordenado*
5000000	1.25002	1.2718	0.391688
10000000	2.40067	2.43171	0.799922
15000000	3.10947	2.91441	1.16022
20000000	4.68318	4.97822	1.82664
25000000	7.13813	7.41515	2.66499
30000000	6.91207	6.94842	2.86176
35000000	8.3462	9.00806	3.51257
40000000	10.1828	10.3293	3.68739
45000000	11.719	12.7701	4.31912
50000000	13.2626	14.8492	4.82287
55000000	12.8813	11.2679	5.1885
60000000	13.8684	13.013	5.81655
65000000	16.1849	15.6284	6.5867
70000000	17.5537	16.6777	7.65251
75000000	18.8133	19.726	7.57288
80000000	19.8614	21.2453	8.4427
85000000	22.7342	23.402	8.70722
90000000	25.6019	26.3575	8.6587
95000000	25.3848	29.7044	10.1216
100000000	27.1072	29.6944	10.8008
105000000	24.5695	21.5564	10.2261
110000000	25.5209	23.9947	10.6785
115000000	27.645	24.8991	11.3382
120000000	29.9953	26.9573	12.1346
125000000	33.7605	30.4166	12.4616

Tabla 23: Mergesort Optimización 2

Tamaño	*Aleatorio*	*Inverso*	*Ordenado*
5000000	1.28159	1.26733	0.371798
10000000	2.4144	2.64686	0.88913
15000000	3.48101	3.12561	1.53575
20000000	4.91212	4.91365	1.90577
25000000	6.63271	7.22963	2.61396
30000000	7.49428	6.60679	3.67215
35000000	9.87558	8.99558	3.44247
40000000	10.1538	10.9519	3.9936
45000000	11.7001	12.5811	4.46932
50000000	14.4018	17.0572	6.19588
55000000	14.009	11.4765	5.48955
60000000	14.9218	13.4192	5.65816
65000000	15.889	14.3196	6.15055
70000000	18.5144	16.8097	7.17176
75000000	19.0213	18.7786	7.39044
80000000	20.7822	21.0382	7.83692
85000000	23.6139	23.1267	9.11984
90000000	25.9093	25.7515	9.32877
95000000	27.8298	28.535	11.4109
100000000	30.1752	30.915	10.4102
105000000	25.3352	22.2698	11.0183
110000000	26.6716	22.6964	10.9248
115000000	28.2778	24.4814	11.6201
120000000	29.0945	26.3045	12.2932
125000000	31.0393	28.3343	12.3744

Tabla 24: Mergesort Optimización 3

4.2.3. Gráficas

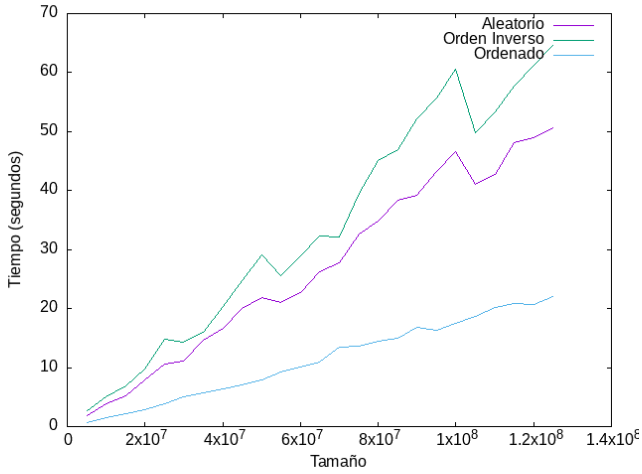

Ilustración 36: Mergesort Optimización 0

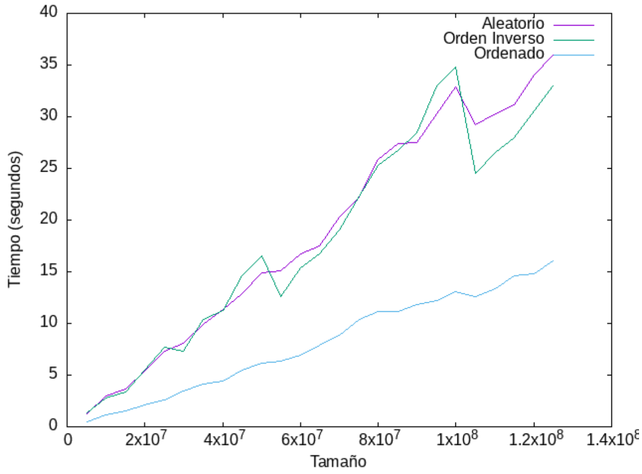

Ilustración 37: Mergesort Optimización 1

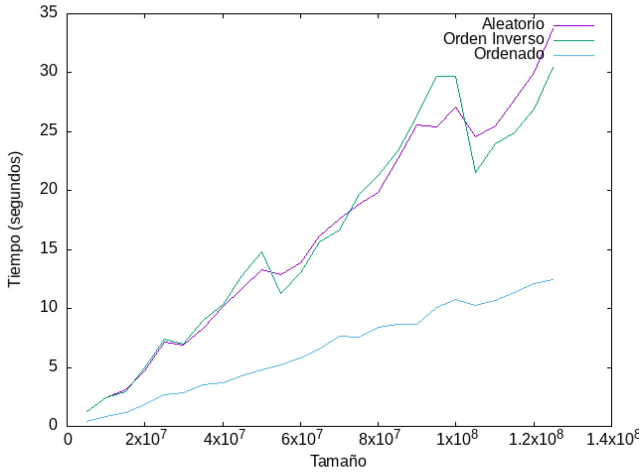

Ilustración 38: Mergesort Optimización 2

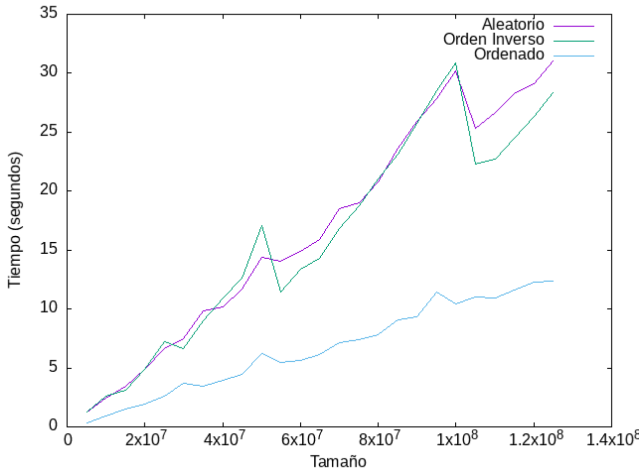

Ilustración 39: Mergesort Optimización 3

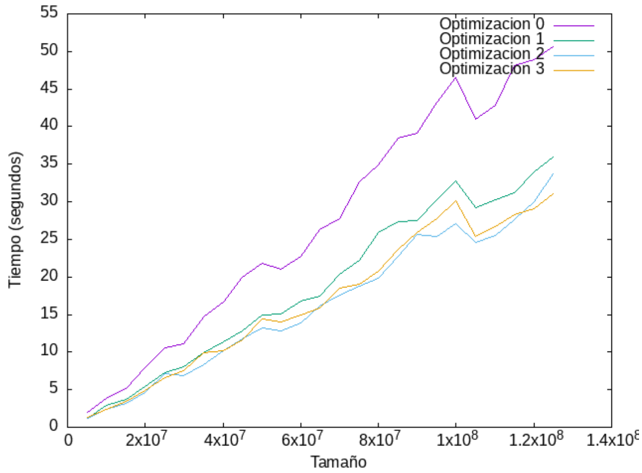

Ilustración 40: Mergesort Comparativa Optimizaciones

Los datos tomados para la comparativa de optimizaciones han sido obtenidos de los tiempos de ordenación del array aleatorio.

4.2.4. Estudio Híbrido

Función de aproximación por regresión mediante mínimos cuadrados:

$$f(x) = a_1 x log(x) + a_0$$

$$a_1 = 1.53192 * 10^{-8}$$

$$a_0 = 1$$

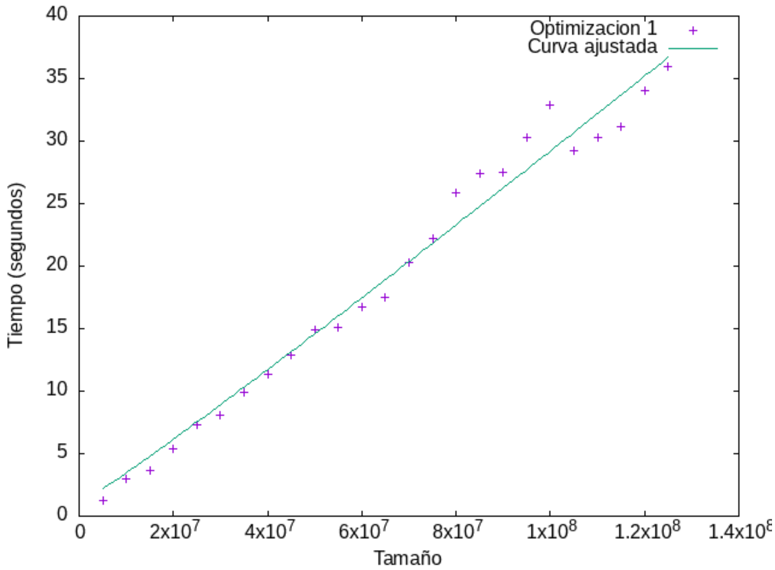

Ilustración 41: Mergesort ajuste

Este cálculo ha sido realizado y representado por el software *gnuplot.* Los datos para los puntos han sido tomados del caso promedio (Vector Aleatorio) en optimización de nivel 1, tomada como representativa del resto de casos.

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

iter      chisq       delta/lim  lambda   a1            a0
   0 4.6423680493e+19   0.00e+00  9.64e+08    1.000000e+00   1.000000e+00
   5 5.6684910218e+01  -3.42e-06  9.64e+03    1.531923e-08   1.000000e+00

After 5 iterations the fit converged.
final sum of squares of residuals : 56.6849
rel. change during last iteration : -3.42311e-11

degrees of freedom    (FIT_NDF)                        : 23
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 1.56989
variance of residuals (reduced chisquare) = WSSR/ndf   : 2.46456

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a1              = 1.53192e-08      +/- 4.615e-10    (3.012%)
a0              = 1                +/- 0.6289       (62.89%)

correlation matrix of the fit parameters:
                a1     a0
a1              1.000
a0             -0.866  1.000
```

Ilustración 42: Mergesort ajuste

Podemos observar cómo, debido a la inestabilidad del algoritmo a la hora de la obtención de los tiempos de ejecución, no se consigue un buen ajuste con esta función. Además, al igual que con *Heapsort*, parece que sería óptimo un ajuste lineal.

4.2.5. Variaciones del entorno

Para comprobar como afectaba el entorno de trabajo a los tiempos, tomamos medidas en distintos equipos (En optimización 0 y con caso promedio), obteniendo como resultados:

Tamaño	*PC_1*	*PC_2*	*PC_3*
50000	0.019669	0.0181102	0.012
100000	0.042973	0.0380698	0.028
150000	0.059982	0.0524057	0.035
200000	0.09491	0.0789907	0.08
250000	0.100437	0.0805037	0.066
300000	0.133122	0.120991	0.073
350000	0.166416	0.133732	0.133
400000	0.202449	0.162229	0.117
450000	0.182156	0.145952	0.138
500000	0.213667	0.173797	0.184
550000	0.237676	0.19525	0.217
600000	0.314404	0.223726	0.217
650000	0.308198	0.284718	0.168
700000	0.335257	0.278119	0.243
750000	0.36244	0.325283	0.308
800000	0.407898	0.34033	0.282
850000	0.315183	0.328044	0.242
900000	0.350146	0.306435	0.251
950000	0.374838	0.329834	0.232
1000000	0.420032	0.353527	0.346
1050000	0.458685	0.384863	0.314
1100000	0.492307	0.407873	0.285
1150000	0.503965	0.441919	0.374
1200000	0.502014	0.461439	0.383
1250000	0.555117	0.49041	0.433

Tabla 25: Mergesort Comparativa PCs

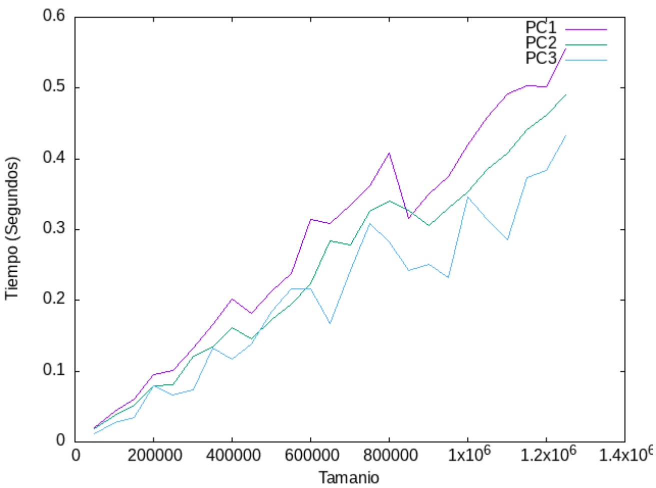

Ilustración 43: Mergesort Comparativa PCs

4.2.6. Conclusiones

Aparte de un comportamiento errático a la hora de realizar las mediciones de los tiempos, encontramos que no existe un caso peor como tal, ya que el promedio, *array* aleatorio, consigue unos resultados muy similares a los que obtenemos con el vector ordenado inversamente.

A su vez, se ha de recalcar que encontramos en todos los niveles de optimización picos de tiempo en las mismas zonas. En cuanto a la mejora obtenida, observamos que ocurre lo mismo que en el resto de algoritmos, existe un gran salto entre el nivel 0 y el 1, el cual no encontramos con el resto de niveles.

Por otro lado, vemos que el comportamiento en un entorno es muy distinto al obtenido con mediciones en otros, aunque no hay ninguno en el que destaque una eficiencia superior al resto.

4.3. ALGORITMO QUICKSORT

4.3.1. Funcionamiento

```
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);
        };
}
```

El funcionamiento del algoritmo de ordenación *Quicksort* es bastante similar al del anterior, el *mergesort*. Este realiza llamadas recursivas dividiendo a través de una función *dividir_qs()* el array en otros dos de menor tamaño, hasta que este sea menor a *UMBRAL_QS*.

Sin embargo, al contrario que el anterior, la división no es simplemente en 2 mitades, sino que la función mencionada previamente establece un pivote distribuyendo los elementos del vector a la derecha o izquierda de este dependiendo de su valor, de forma que no es necesaria una función *fusion()*.

Las medidas en este algoritmo las hemos tomado con tamaño de *array* desde 50000 hasta 1250000, con un orden inicial aleatorio, inverso y directo, y con optimización 0, 1, 2 y 3.

4.3.2. Tablas de datos

Tamaño	*Aleatorio*	*Inverso*	*Ordenado*
5000	0.0009643	0.0187074	0.018151
10000	0.0012594	0.0735475	0.0720688
15000	0.0019231	0.171295	0.170557
20000	0.0026292	0.291412	0.289818
25000	0.0033667	0.455046	0.452916
30000	0.0041156	0.659789	0.651346
35000	0.0047411	0.894152	0.885447
40000	0.0055854	1.26227	1.34967
45000	0.0097862	1.79564	1.58428
50000	0.0070746	2.08963	2.09506
55000	0.0088350	2.48403	2.50121
60000	0.0098944	3.04788	3.16141
65000	0.010847	3.59704	3.52255
70000	0.0122116	4.13348	4.40419
75000	0.0133129	5.00914	4.76617
80000	0.0152362	5.39126	5.36263
85000	0.0160539	5.81793	5.79938
90000	0.0146797	6.79386	6.50974
95000	0.0169083	7.41081	6.99148
100000	0.0155402	7.67485	7.72041
105000	0.0193042	9.15262	8.94402
110000	0.0198874	10.7739	9.74727
115000	0.020448	11.2443	11.2204
120000	0.0253869	13.0056	12.9639
125000	0.0232229	14.2235	13.4281

Tabla 26: Quicksort Optimización 0

Tamaño	*Aleatorio*	*Inverso*	*Ordenado*
5000	0.0004014	0.0085831	0.0064392
10000	0.0008162	0.0316827	0.0277128
15000	0.0015986	0.0763101	0.0582032
20000	0.0017637	0.132135	0.10682
25000	0.0024043	0.202263	0.164009
30000	0.0026783	0.288961	0.233932
35000	0.0031349	0.392875	0.331947
40000	0.0037301	0.50591	0.417702
45000	0.0040757	0.641784	0.515578
50000	0.0050667	0.803741	0.654774
55000	0.0053228	1.03626	0.806027
60000	0.0056853	1.27183	0.950578
65000	0.0060627	1.38025	1.40003
70000	0.0082912	1.78659	1.3882
75000	0.0073107	1.79523	1.9719
80000	0.0100994	2.60436	2.16641
85000	0.0101701	2.92104	2.42826
90000	0.0116366	2.88386	2.37631
95000	0.0104081	4.0399	2.73002
100000	0.0122518	3.90761	2.62791
105000	0.0144503	4.5542	3.60982
110000	0.0143105	4.96593	3.98132
115000	0.0137337	4.85609	3.93809
120000	0.0139178	5.37488	4.56271
125000	0.0143656	6.1594	5.16432

Tabla 27: Quicksort Optimización 1

Tamaño	*Aleatorio*	*Inverso*	*Ordenado*
5000	0.0004325	0.0095873	0.0091978
10000	0.0009194	0.034973	0.0360177
15000	0.0013538	0.0778615	0.0847158
20000	0.0021880	0.162964	0.14996
25000	0.0027741	0.262535	0.243448
30000	0.0031453	0.345232	0.356825
35000	0.0041262	0.430471	0.40997
40000	0.0037286	0.534541	0.532518
45000	0.0041799	0.723766	0.864285
50000	0.0060126	0.984187	0.963927
55000	0.0060457	1.36176	1.15484
60000	0.0062297	1.52096	1.46672
65000	0.0075155	1.80602	1.74902
70000	0.0084190	1.96782	1.86208
75000	0.0079765	2.27318	2.33886
80000	0.0097381	2.5942	2.66781
85000	0.0104373	2.69825	2.88836
90000	0.0110836	3.45507	3.56387
95000	0.012369	3.73467	3.60213
100000	0.0109517	4.35044	4.27565
105000	0.014915	4.26606	4.57864
110000	0.0123666	4.67641	5.08172
115000	0.0132008	5.46375	5.51603
120000	0.0134674	5.8896	5.96314
125000	0.0157187	6.51393	6.6073

Tabla 28: Quicksort Optimización 2

Tamaño	*Aleatorio*	*Inverso*	*Ordenado*
5000	0.0003528	0.0085768	0.0086524
10000	0.0008053	0.034345	0.0347281
15000	0.0012021	0.0803682	0.073807
20000	0.0016358	0.130214	0.132318
25000	0.0021439	0.209702	0.205914
30000	0.0025405	0.305281	0.322174
35000	0.0030502	0.408233	0.542973
40000	0.0049676	0.703842	0.624246
45000	0.0049778	0.786957	0.821366
50000	0.0057136	0.848208	0.848773
55000	0.0052243	1.14689	1.2304
60000	0.0064058	1.50841	1.5698
65000	0.0083339	1.68831	1.73499
70000	0.0078073	1.91434	1.86784
75000	0.0079710	2.1498	2.2027
80000	0.0088957	2.2018	2.52982
85000	0.0085162	3.00109	3.17743
90000	0.0104113	3.13471	3.27855
95000	0.0101371	3.76323	3.77768
100000	0.0114917	3.75453	3.86195
105000	0.01449	4.45041	4.28842
110000	0.0115776	4.55442	4.51609
115000	0.012617	5.32853	5.22468
120000	0.0115502	5.76735	5.94495
125000	0.0145356	6.3076	6.24313

Tabla 29: Quicksort Optimización 3

4.3.3. Gráficas

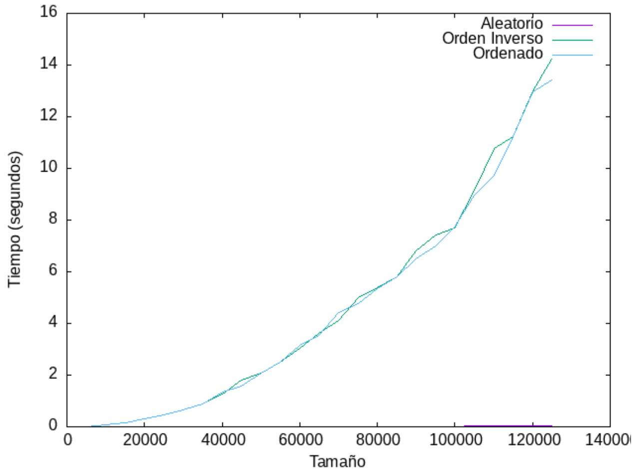

Ilustración 44: Quicksort Optimización 0

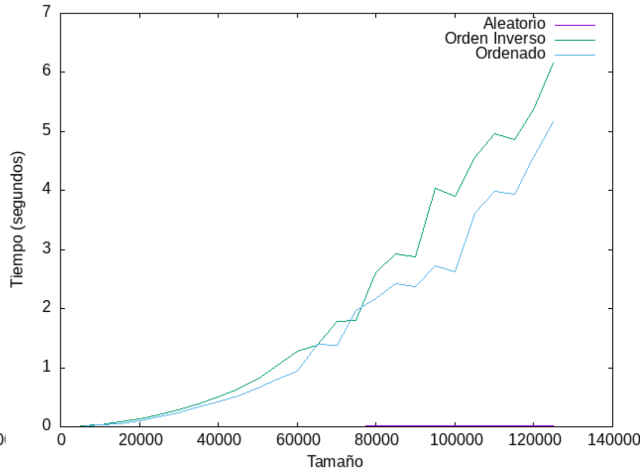

Ilustración 45: Quicksort Optimización 1

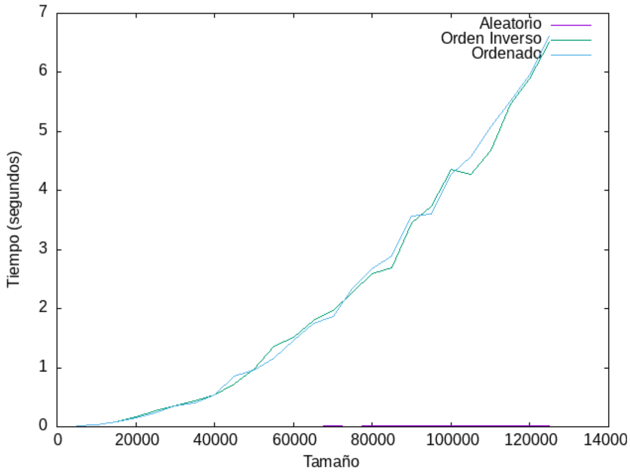

Ilustración 46: Quicksort Optimización 2

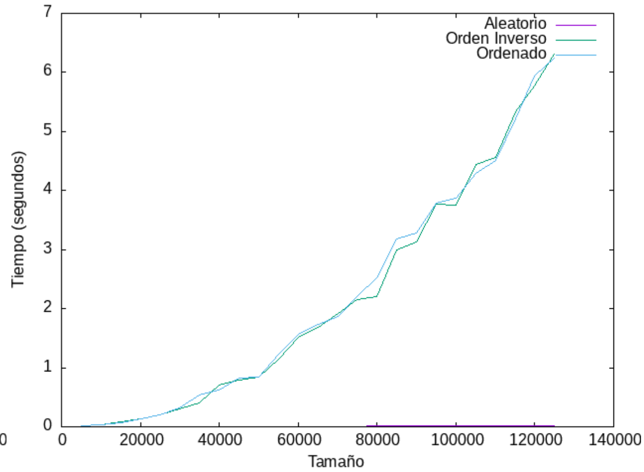

Ilustración 47: Quicksort Optimización 3

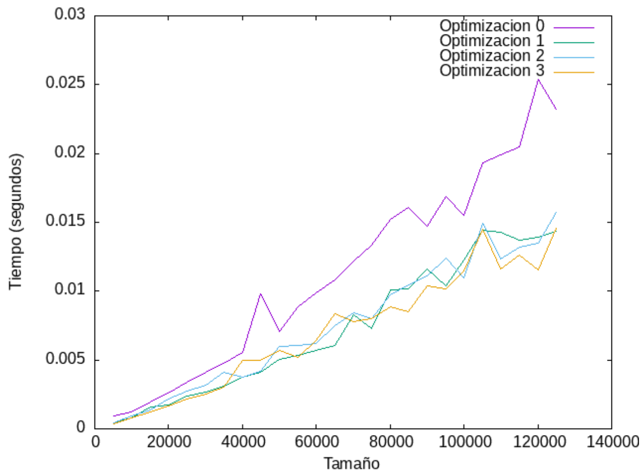

Ilustración 48: Quicksort Comparativa Optimizaciones

Los datos tomados para la comparativa de optimizaciones han sido obtenidos de los tiempos de ordenación del array aleatorio.

4.3.4. Estudio Híbrido

Función de aproximación por regresión mediante mínimos cuadrados:

$$f(x) = a_1 x log(x) + a_0$$

$$a_1 = 1.24281 * 10^{-8}$$

$$a_0 = 1$$

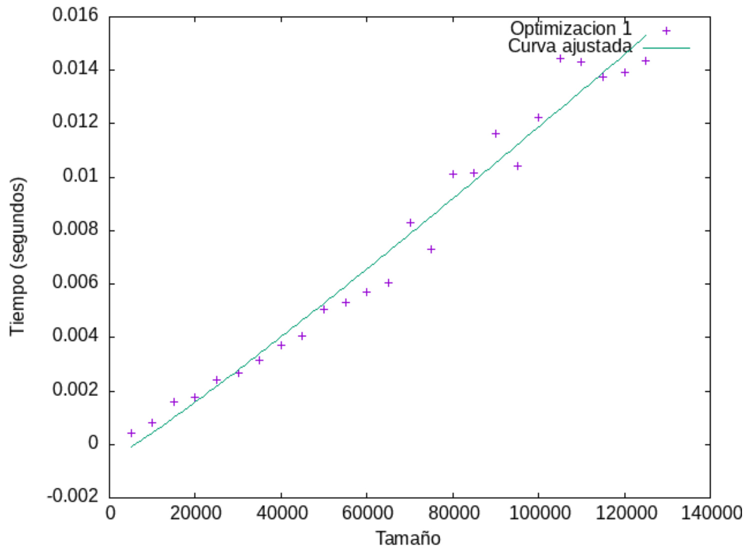

Ilustración 49: Quicksort ajuste

Este cálculo ha sido realizado y representado por el software *gnuplot.* Los datos para los puntos han sido tomados del caso promedio (Vector Aleatorio) en optimización de nivel 1, tomada como representativa del resto de casos.

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

iter      chisq       delta/lim  lambda   a1             a0
   0 4.6423680493e+19   0.00e+00  9.64e+08    1.000000e+00   1.000000e+00
   5 5.6684910218e+01  -3.42e-06  9.64e+03    1.531923e-08   1.000000e+00

After 5 iterations the fit converged.
final sum of squares of residuals : 56.6849
rel. change during last iteration : -3.42311e-11

degrees of freedom    (FIT_NDF)                        : 23
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 1.56989
variance of residuals (reduced chisquare) = WSSR/ndf   : 2.46456

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a1              = 1.53192e-08      +/- 4.615e-10    (3.012%)
a0              = 1                +/- 0.6289       (62.89%)

correlation matrix of the fit parameters:
                a1     a0
a1              1.000
a0             -0.866  1.000
```

Ilustración 50: Quicksort ajuste

Al igual que con *Heapsort* y *Mergesort*, el ajuste no es óptimo y la distribución de puntos se asemeja mayormente a una función lineal, aunque en las medidas tomadas estén levemente dispersas.

4.3.5. Variaciones del entorno

Para comprobar como afectaba el entorno de trabajo a los tiempos, tomamos medidas en distintos equipos (En optimización 0 y con caso promedio), obteniendo como resultados:

Tamaño	*PC_1*	*PC_2*	*PC_3*
5000000	1.92359	0.0181102	0.012
10000000	3.83117	0.0380698	0.028
15000000	5.151	0.0524057	0.035
20000000	7.93284	0.0789907	0.08
25000000	10.5608	0.0805037	0.066
30000000	11.0974	0.120991	0.073
35000000	14.6724	0.133732	0.133
40000000	16.6203	0.162229	0.117
45000000	20.0223	0.145952	0.138
50000000	21.8151	0.173797	0.184
55000000	21.0519	0.19525	0.217
60000000	22.7875	0.223726	0.217
65000000	26.2845	0.284718	0.168
70000000	27.7851	0.278119	0.243
75000000	32.6026	0.325283	0.308
80000000	34.895	0.34033	0.282
85000000	38.4307	0.328044	0.242
90000000	39.1796	0.306435	0.251
95000000	43.2019	0.329834	0.232
100000000	46.6023	0.353527	0.346
105000000	40.9785	0.384863	0.314
110000000	42.7985	0.407873	0.285
115000000	48.059	0.441919	0.374
120000000	48.95	0.461439	0.383
125000000	50.6052	0.49041	0.433

Tabla 30: Quicksort Comparativa PCs

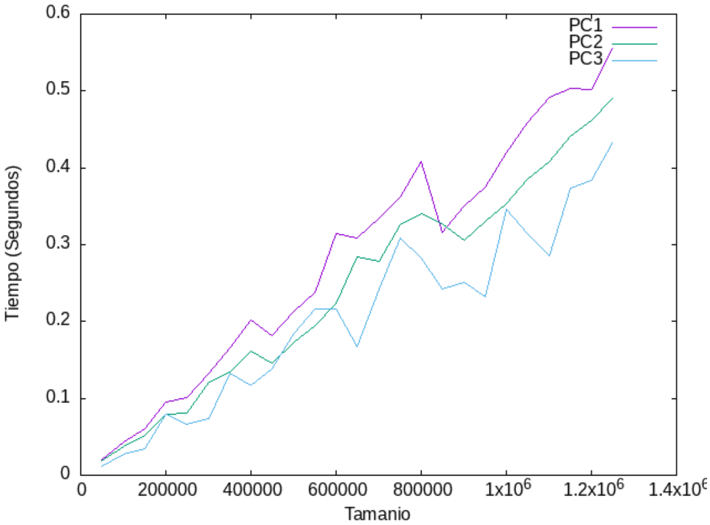

Ilustración 51: Quicksort Comparativa PCs

4.3.6. Conclusiones

Observamos que, por muy extraño que parezca, los tiempos de ordenación de *arrays* aleatorios son de varios ordenes menores a los de vectores ya ordenados e inversamente ordenado. Esto se puede deber en parte a la necesidad de organizar el conjunto según el pivote elegido, de forma que, el que esté ordenado directamente o inversamente retrase este proceso.

En cuanto a los niveles de optimización, no existe un cambio notable en cuanto a la tendencia que llevábamos con los algoritmos anteriores. Existe un gran salto entre el nivel 0 y el 1, cosa que no existe entre el 1, el 2 y el 3.

Por último, aparte de ver una gráfica con los puntos un tanto dispersos de su supuesta tendencia teórica, vemos que en los 3 equipos obtenemos resultados muy parecidos, tal vez consiguiendo tiempos mejores en el tercer equipo.

5. COMPARATIVA ENTRE ALGORITMOS

5.1. ALGORITMOS O(N^2)

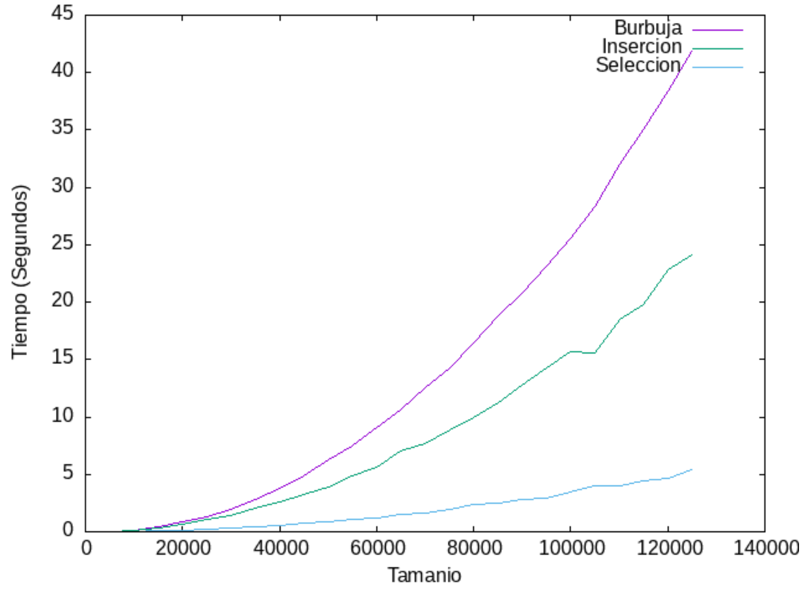

Ilustración 52: Comparativa Algoritmos O(n^2)

Estos datos han sido obtenidos con optimización nivel 1 y con el caso de array aleatorio.

Observamos claramente como el algoritmo burbuja obtiene los peores tiempos, siendo superado en gran medida por el de inserción y, sobre todo, por el de selección.

5.2. ALGORITMOS O(NLOG(N))

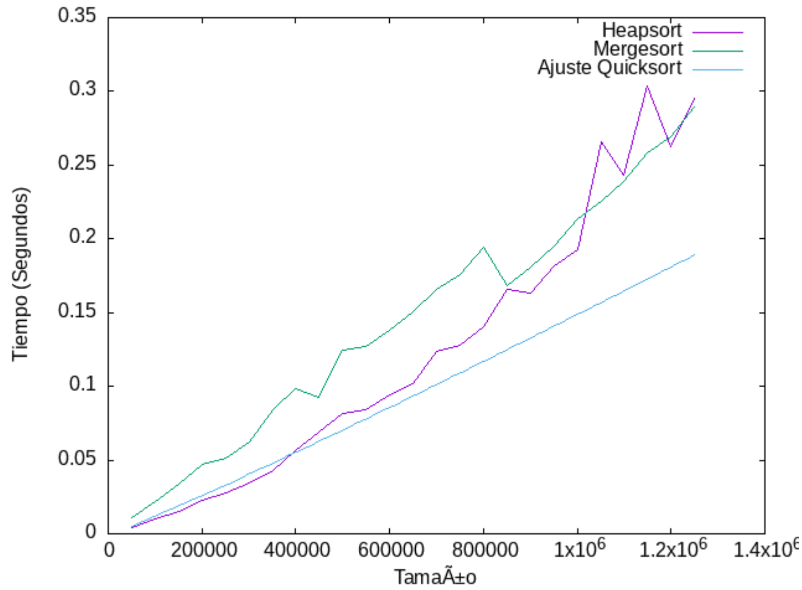

Ilustración 53: Comparativa Algoritmos O(nlog(n))

Estos datos han sido obtenidos con optimización nivel 1 y con el caso de array aleatorio.

Encontramos menos diferencia que con los algoritmos anteriores, aunque podemos ver perfectamente como el algoritmo *Quicksort* es el que mejores tiempos consigue, aunque es superado en tamaños más pequeños por *Heapsort.*

5.3. TODOS LOS ALGORITMOS

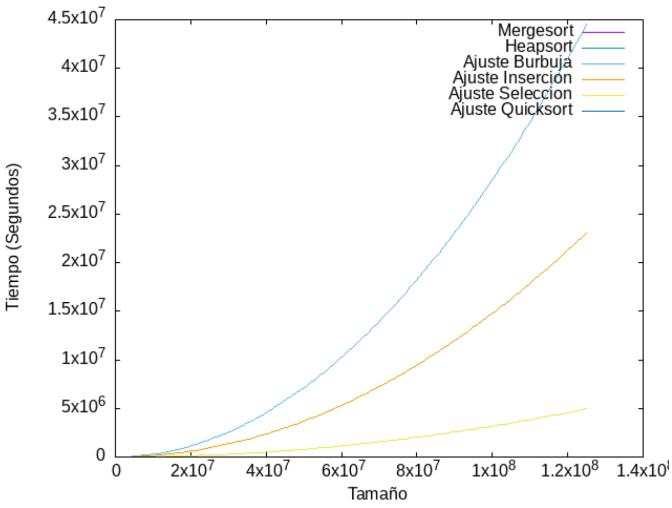

Ilustración 54: Con ajustes de los de O(n^2)

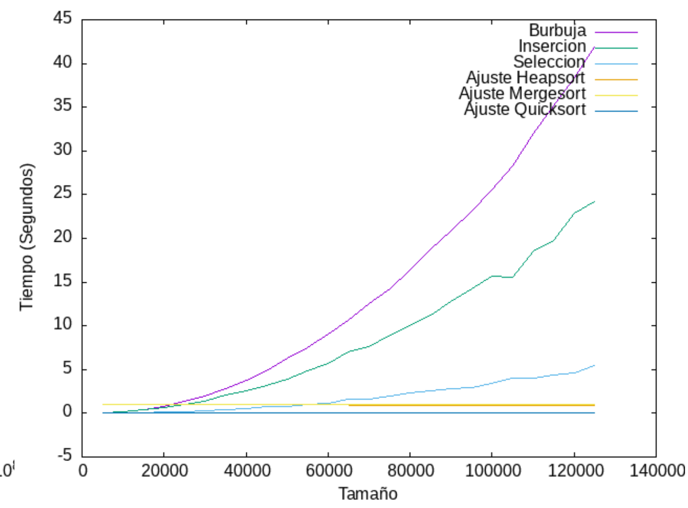

Ilustración 55: Con ajustes de los de O(nlog(n))

Como todo indicaba, vemos como los algoritmos de orden O(nlog(n)) consiguen tiempos muy por debajo de los obtenidos por el resto, tanto es así que apenas se aprecian en las gráficas. En la primera, al tener valores tan próximos a 0 en comparación a los obtenidos por los ajustes, la representación de estas ni aparece. En la segunda, vemos que su representación parece una recta horizontal debido a que prácticamente no crece con estos tamaños de *array*.

Vemos como, en conclusión, el más rápido de todos los algoritmos estudiados en esta práctica es el **algoritmo *Quicksort***.

Aunque cabe recalcar que realizando el estudio, este algoritmo ha provocado múltiples core dump con tamaños grandes de array, lo cual indica que está indicado sobre todo para conjuntos a ordenar pequeños.