# Análisis de la eficiencia de algoritmos

Algorítmica

Doble Grado en Ingeniería Informática y Matemáticas

Yábir García Benchakhtir
yabirgb@correo.ugr.es

7 de marzo de 2018

# Índice

# 1. Análisis de los algoritmos

Queda por realizar un análisis teórico de los algoritmos de ordenación pero por falta de tiempo lo entregaré en la siguiente fecha de entrega.

## 1.1. Algoritmo burbuja

Este algoritmo funciona como 'las burbujas en un refresco'. Lo que hace es comparar de izquierda a derecha del vector cada elemento con los adyacentes de modo que los más grandes se desplazan a la derecha.

Como se explica en el guión la eficiencia de este algoritmo es:

$$\frac{a}{2}n^2 - \frac{3a}{2}n + a$$

## 1.2. Algoritmo de ordenación por insercción

Busca en un vector de elementos el menor y lo coloca a la izquierda, después repetimos esta operación para los $n-1$ elementos restantes.

## 1.3. Algoritmo de ordenación por selección

Partimos de una lista de elementos ordenados con un solo elemento y de otra de elementos desordenados con $n-1$ elementos. En cada paso elegimos un elemento de la lista de elementos desordenados y lo insertamos de manera ordenada en la lista de elementos ordenados.

```
1  static void seleccion_lims(int T[], int inicial, int final)
2  {
3    int i, j, indice_menor;
4    int menor, aux;
5    for (i = inicial; i < final - 1; i++) {
6      indice_menor = i;
7      menor = T[i];
8      for (j = i; j < final; j++)
9        if (T[j] < menor) {
10         indice_menor = j;
11         menor = T[j];
12       }
13     aux = T[i];
14     T[i] = T[indice_menor];
15     T[indice_menor] = aux;
16   };
17 }
```

## 1.4. Algoritmo de ordenación mergesort

Creamos listas de un solo elementos y las vamos combinando en distintos pasos de manera ordenada. Estamos haciendo algoritmo que consiste en dividir la complejidad en tareas más pequeñas y después juntar el resultado ya más sencillo.

La complejidad de este algoritmo es $O(n \cdot log(n)$ más concretamente:

$$T(n) = c_1 n + c_2 n \cdot log(n)$$

## 1.5. Algoritmo de ordenación quicksort

En este algoritmo tomamos un punto que hará de pivote. Movemos los elementos de manera que a la derecha del pivote queden los que son mayores que él y la izquierda los que son menores. Una vez hecho esto repetimos el proceso tomando pivotes en los conjuntos de los elementos mayores y menores.

## 1.6. Algoritmo de ordenación heapsort

Disponemos todos los elementos en un *heap*. Buscamos el mayor de los elementos y lo quitamos insertandolo al final. Funciona de manera parecida a la ordenación burbuja.

## 1.7. Algoritmo floyd

Este algoritmo realiza todas las sumas posibles y devuelve la mayor de ellas.

```
1  void Floyd(int **M, int dim)
2  {
3          for (int k = 0; k < dim; k++)
4            for (int i = 0; i < dim;i++)
5              for (int j = 0; j < dim;j++)
6                {
7                  int sum = M[i][k] + M[k][j];
8                  M[i][j] = (M[i][j] > sum) ? sum : M[i][j];
9                }
10 }
```

Este algoritmo es de complejidad $O(n^3)$, más concretamente:

$$\sum_{k=0}^{n}\sum_{i=0}^{n}\sum_{j=0}^{n} a = an^3$$

## 1.8. Algoritmo de las torres de Hanoi

```
1  void hanoi (int M, int i, int j)
2  {
3    if (M > 0)
4      {
5        hanoi(M-1, i, 6-i-j);
6        //cout << i << " -> " << j << endl;
7        hanoi (M-1, 6-i-j, j);
8    }
9  }
```

En este algoritmo aplicamos la siguiente recurrencia:

$$h(n) = 2h(n-1)$$

que se corresponde con la ecuación en recurrencias homogenea:

$$x_n = 2x_{n-1}$$

Cuya solución es a partir de un termino incial $x_0$:

$$x_n = 2^n x_0$$

# 2. Cálculo de la eficiencia empírica

## 2.1. Procedimiento

Para el cálculo de la eficiencia empírica se ha automatizado el proceso. Para ellos se han hecho 2 scripts de bash y un archivo makefile para realizar las siguientes tareas:

- Crear archivos ejecutables para todos los algoritmos programados en *C++* con distintas opciones de optimización a saber *O1*, *O2* y *O3*.
- Ejecutar los distintos tests para los intervalos programados y almacenar los resultados en archivos de datos.
- Crear las respectivas gráficas para cada tabla de datos obtenida usando la herramienta gnuplot.

Para los algoritmos de medición se ha elegido un rango de datos común en el intervalo [1000, 25000] de manera que se toman 25 medidas haciendo incrementos de 1000 en 1000 para tomar las medidas.

$$D = \{x \in [1000, 25000] : x = 1000k, k \in \mathbb{N}\}$$

Junto a este documento se encuentran las gráficas creadas y los datos que proporciona el programa *gnuplot* sobre las mediciones.

A la hora de hacer un ajuste de mínimos cuadrados se han usado las siguientes funciones de ajuste:

| Complejidad (*Big O*) | Función de ajuste |
|---|---|
| $O(n^2)$ | $f(x) = ax^2 + bx + c$ |
| $O(n^3)$ | $f(x) = ax^3 + bx^2 + cx + d$ |
| $O(nlog(n))$ | $f(x) = a \cdot log(x + b) + c$ |
| $O(2^n)$ | $f(x) = a2^x$ |

Cuadro 1: Funciones de ajuste

## 2.2. Condiciones de las mediciones

Para llevar a cabo las mediciones se ha utilizado un ordenador con las siguientes características:

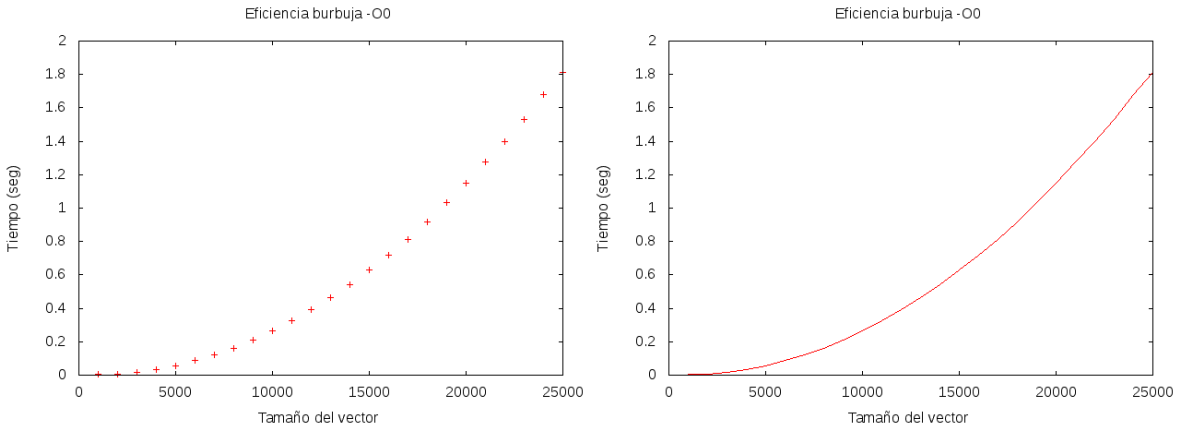
- CPU: Intel Pentium G3258 (2) @ 3.200GHz
- Memoria RAM: 7876MiB
- Kernel: 4.13.0-36-generic
- OS: Linux Mint 18.3 Sylvia x86_64

A la hora de realizar los tests se ha tenido la precaución de minimizar el uso de *CPU* para no interferir en las mediciones.

# 3. Resultados obtenidos

En esta sección de la practica me concentro en mostrar los resultados obtenidos tras haber completado el proceso de pruebas de los distintos algoritmos. En este caso muestro los datos con una eficiencia al compilar *O0*.

## 3.1. Algoritmo de ordenación burbuja




(a) Nube de puntos (b) Función continua

Figura 1: Resultados experimentales representados mediante una nube de puntos y la linea que los une

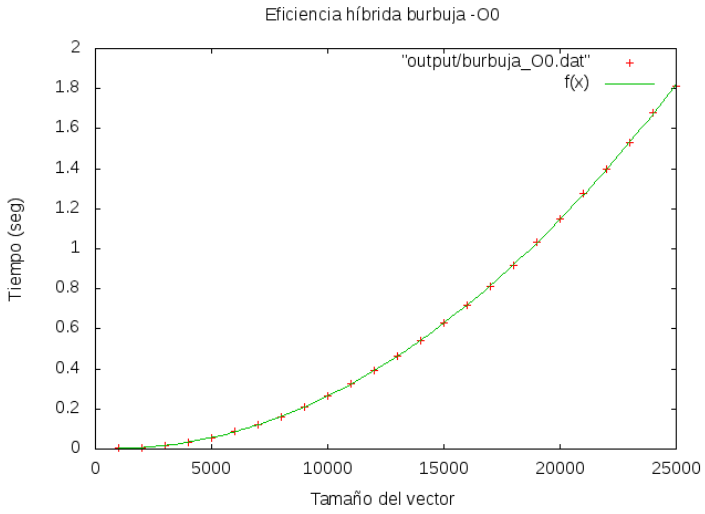
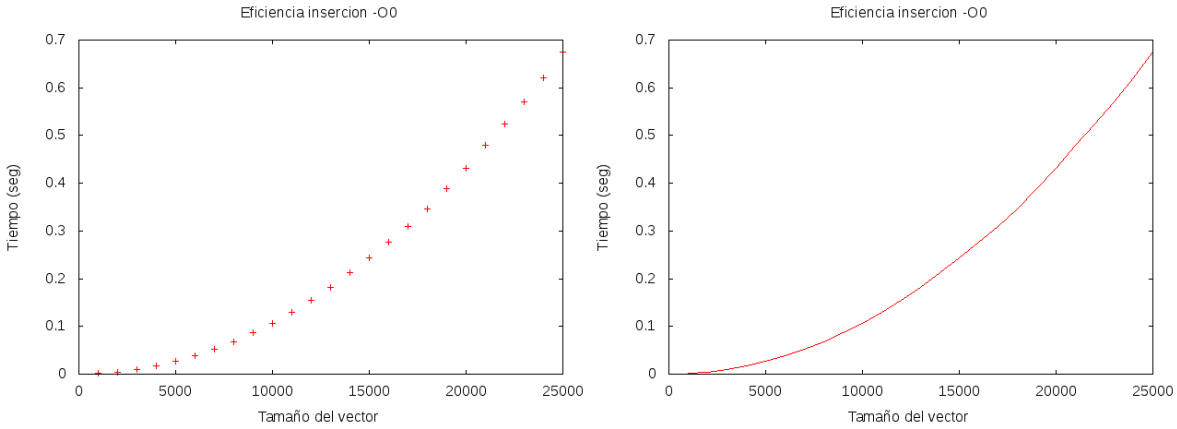



Figura 2: Ajuste para: ordenación usando el algoritmo de burbuja

```
Final set of parameters            Asymptotic Standard Error
=======================            ==========================

a                 = 3.11154e-09    +/- 1.128e-11    (0.3624%)
b                 = -5.17483e-06   +/- 3.02e-07     (5.837%)
c                 = 0.00563209     +/- 0.001704     (30.26%)
```

## 3.2. Algoritmo de ordenación por inserción




(a) Nube de puntos (b) Función continua

Figura 3: Resultados experimentales representados mediante una nube de puntos y la linea que los une

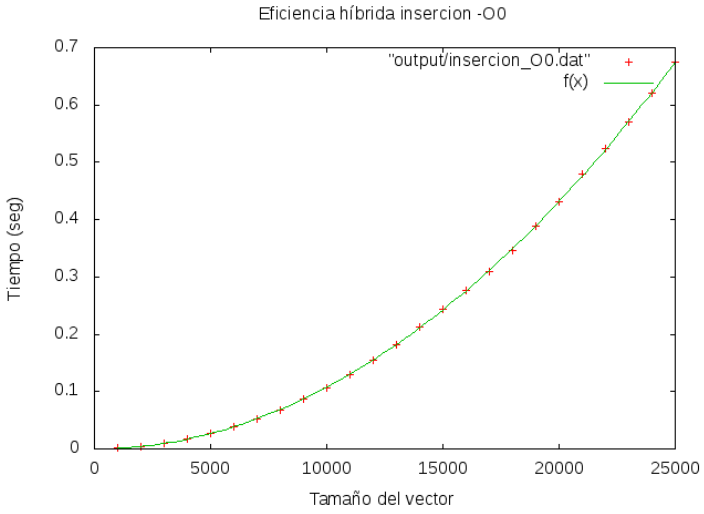
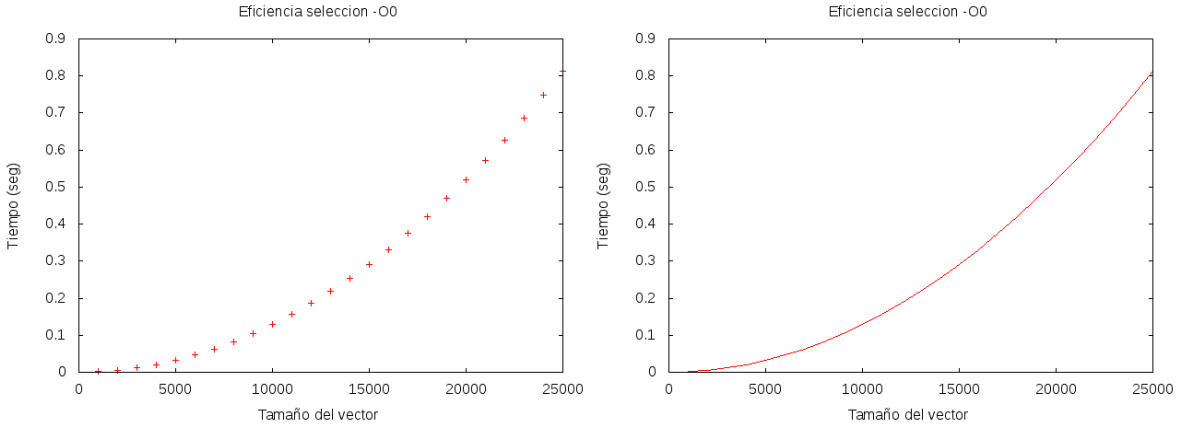



Figura 4: Ajuste para: ordenación usando el algoritmo de inserción

```
Final set of parameters            Asymptotic Standard Error
=======================            ==========================

a                 = 1.08271e-09    +/- 6.279e-12    (0.5799%)
b                 = -7.7687e-08    +/- 1.682e-07    (216.5%)
c                 = 0.000123925    +/- 0.0009489    (765.7%)
```

### 3.3. Algoritmo de ordenación por selección




(a) Nube de puntos (b) Función continua

Figura 5: Resultados experimentales representados mediante una nube de puntos y la linea que los une

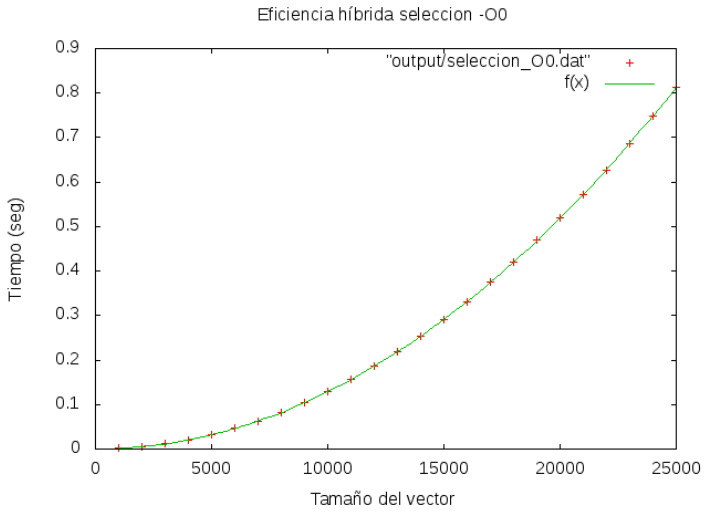


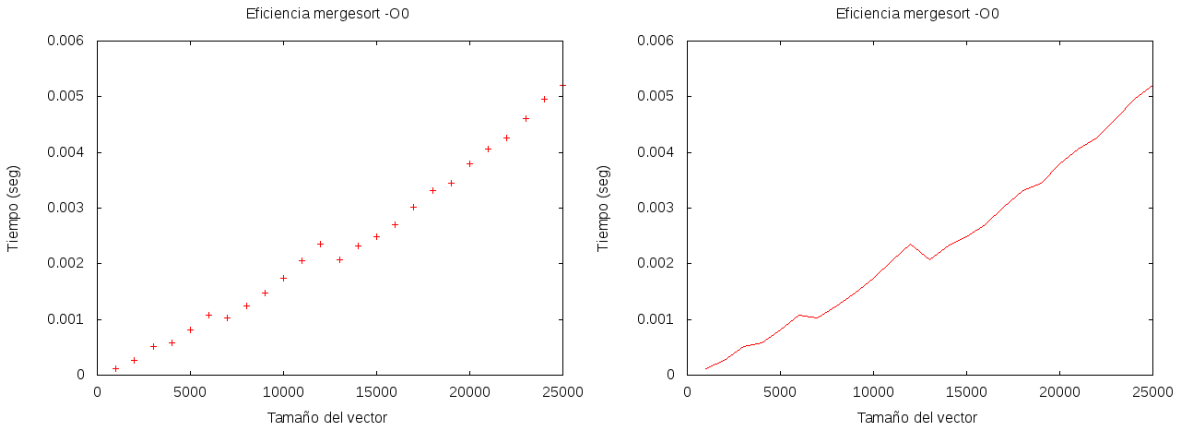

Figura 6: Ajuste para: ordenación usando el algoritmo de selección

```
Final set of parameters            Asymptotic Standard Error
=======================            ==========================

a                 = 1.30761e-09    +/- 1.868e-12    (0.1429%)
b                 = -2.41748e-07   +/- 5.005e-08    (20.7%)
c                 = 0.000572432    +/- 0.0002824    (49.33%)
```

## 3.4. Comparativa de los algoritmos cuadráticos de ordenación

## 3.5. Algoritmo de ordenación mergesort




(a) Nube de puntos (b) Función continua

Figura 7: Resultados experimentales representados mediante una nube de puntos y la linea que los une

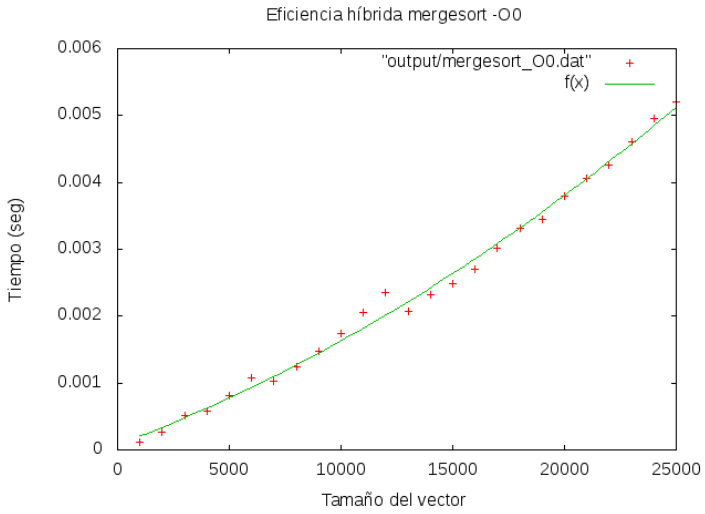
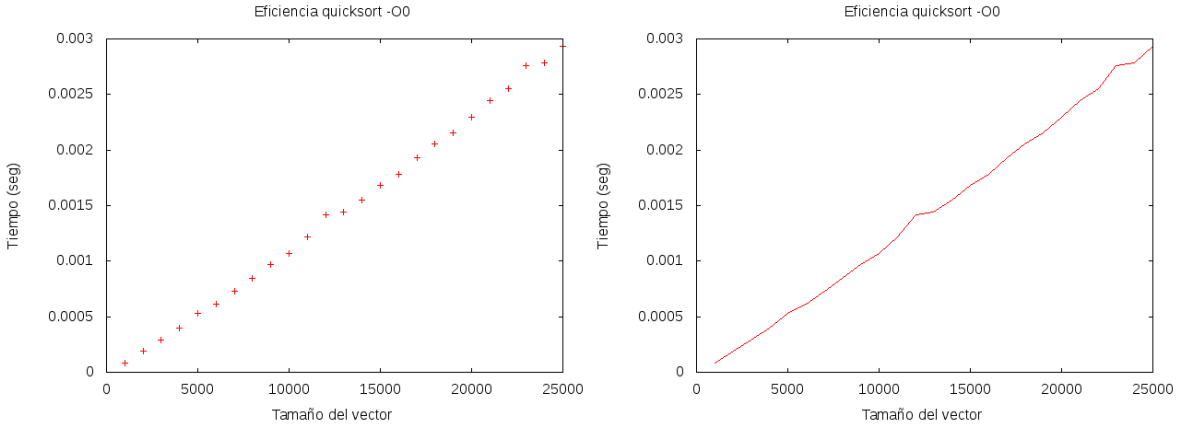



Figura 8: Ajuste para: ordenación usando el algoritmo mergesort

```
Final set of parameters            Asymptotic Standard Error
=======================            ==========================

a                 = 0.00162555     +/- 0.0003876    (23.85%)
b                 = 1.00697        +/- 964.1        (9.575e+04%)
c                 = -0.0126195     +/- 0.003794     (30.07%)
```

## 3.6. Algoritmo de ordenación quicksort




(a) Nube de puntos (b) Función continua

Figura 9: Resultados experimentales representados mediante una nube de puntos y la linea que los une

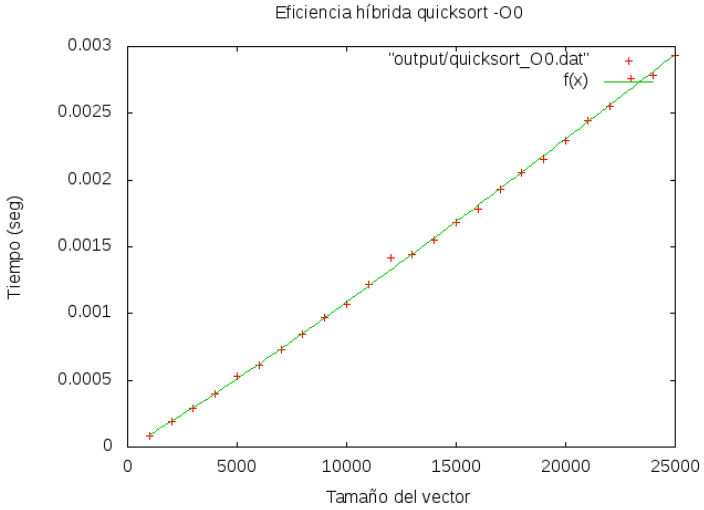
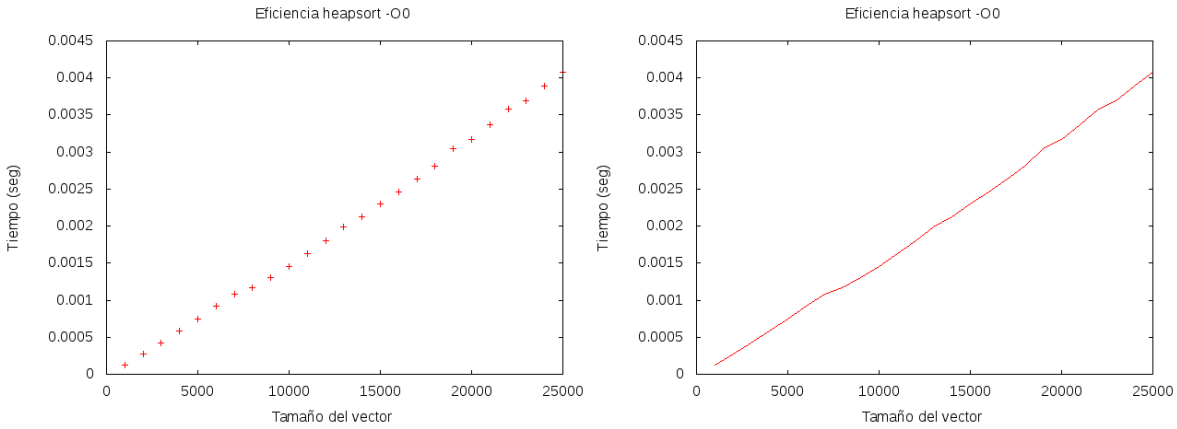



Figura 10: Ajuste para: ordenación usando el algoritmo quicksort

```
Final set of parameters            Asymptotic Standard Error
=======================            ==========================

a                 = 0.000967764    +/- 0.0001992    (20.58%)
b                 = 1.00236        +/- 832.3        (8.303e+04%)
c                 = -0.00746142    +/- 0.00195      (26.13%)
```

## 3.7. Algoritmo de ordenación heapsort




(a) Nube de puntos

(b) Función continua

Figura 11: Resultados experimentales representados mediante una nube de puntos y la linea que los une

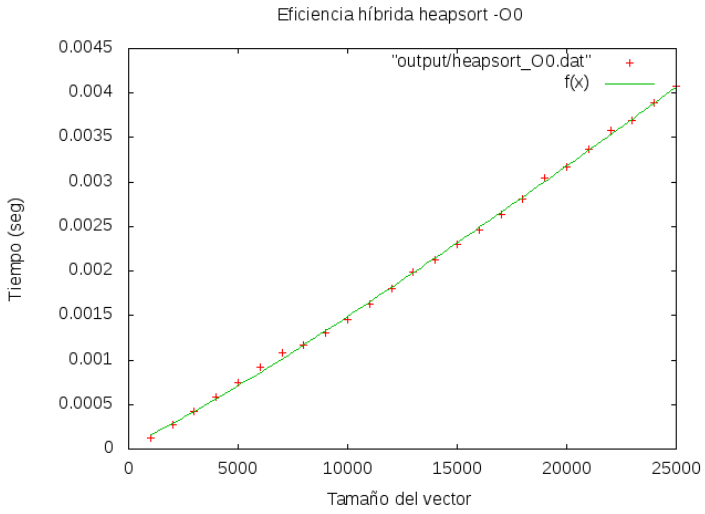
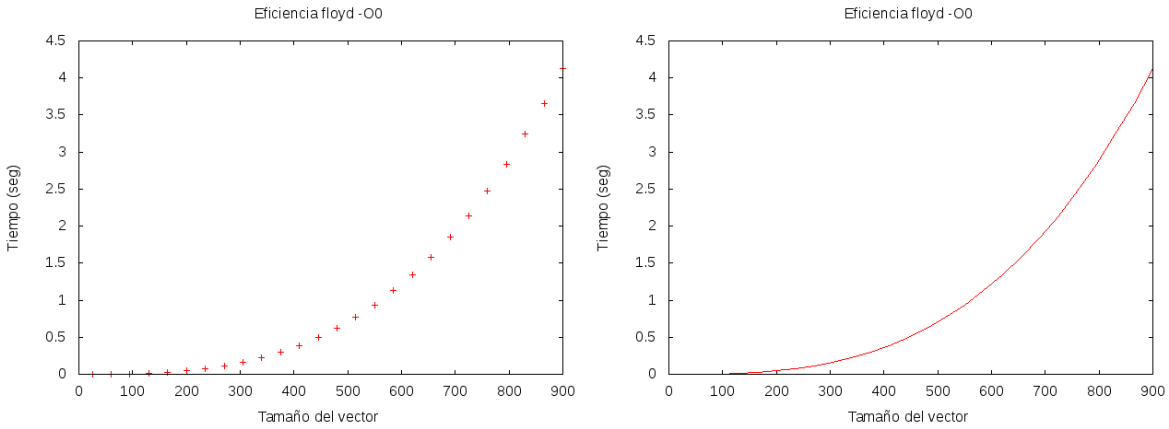



Figura 12: Ajuste para: ordenación usando el algoritmo heapsort

```
Final set of parameters            Asymptotic Standard Error
=======================            ==========================

a                 = 0.00132491     +/- 0.0002776    (20.95%)
b                 = 1.00436        +/- 847.2        (8.436e+04%)
c                 = -0.0102003     +/- 0.002718     (26.64%)
```

## 3.8. Algoritmo floyd




(a) Nube de puntos (b) Función continua

Figura 13: Resultados experimentales representados mediante una nube de puntos y la linea que los une

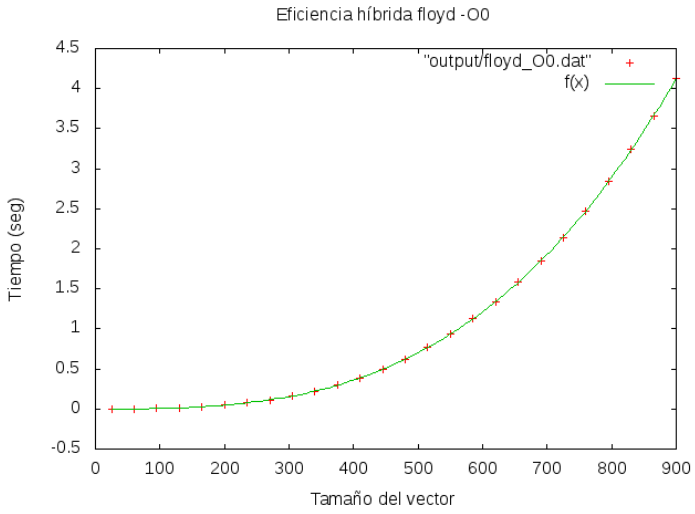
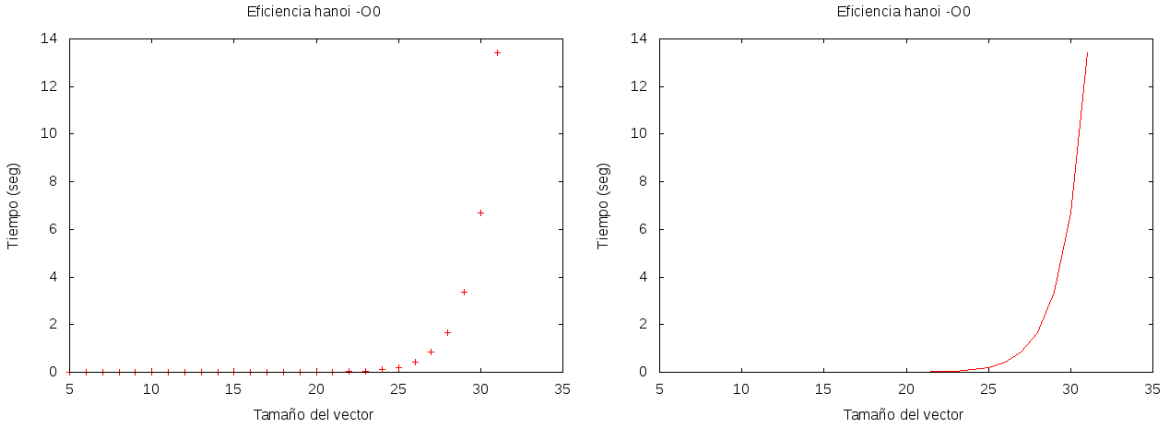



Figura 14: Ajuste para: algoritmo para calculo de costo floyd

## 3.9. Algoritmo de las torres de Hanoi




(a) Nube de puntos (b) Función continua

Figura 15: Resultados experimentales representados mediante una nube de puntos y la linea que los une

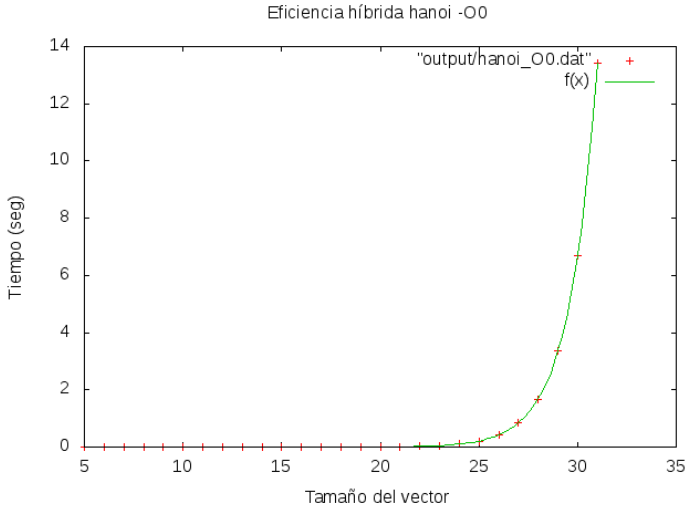



Figura 16: Ajuste para: calculo de los movimientos en las torres de hanoi

# 4. Analisis de los resultados

## 4.1. Comparativa de los algoritmos $O(n^2)$ de ordenación

| Tamaño | Burbuja | Insercción | Selección |
|---|---|---|---|
| 0 | 0.004246 | 0.00112 | 0.00135595 |
| 1000 | 0.008201 | 0.004292 | 0.00524505 |
| 2000 | 0.019307 | 0.009728 | 0.011705 |
| 3000 | 0.034225 | 0.017294 | 0.020711 |
| 4000 | 0.056845 | 0.026957 | 0.03227 |
| 5000 | 0.088076 | 0.038868 | 0.046363 |
| 6000 | 0.120856 | 0.052248 | 0.062962 |
| 7000 | 0.161862 | 0.068156 | 0.082298 |
| 8000 | 0.210286 | 0.086587 | 0.104299 |
| 9000 | 0.264146 | 0.107304 | 0.128745 |
| 10000 | 0.323677 | 0.130252 | 0.156038 |
| 11000 | 0.394902 | 0.15566 | 0.185971 |
| 12000 | 0.463354 | 0.182503 | 0.218305 |
| 13000 | 0.541129 | 0.212983 | 0.253628 |
| 14000 | 0.628463 | 0.243724 | 0.291057 |
| 15000 | 0.718559 | 0.2767 | 0.331193 |
| 16000 | 0.814468 | 0.309732 | 0.374365 |
| 17000 | 0.919391 | 0.346489 | 0.419854 |
| 18000 | 1.03422 | 0.387793 | 0.469411 |
| 19000 | 1.14783 | 0.432006 | 0.518467 |
| 20000 | 1.27545 | 0.479765 | 0.571629 |
| 21000 | 1.39817 | 0.524347 | 0.62718 |
| 22000 | 1.53151 | 0.569582 | 0.68739 |
| 23000 | 1.67702 | 0.620089 | 0.747746 |
| 24000 | 1.81403 | 0.675667 | 0.81211 |

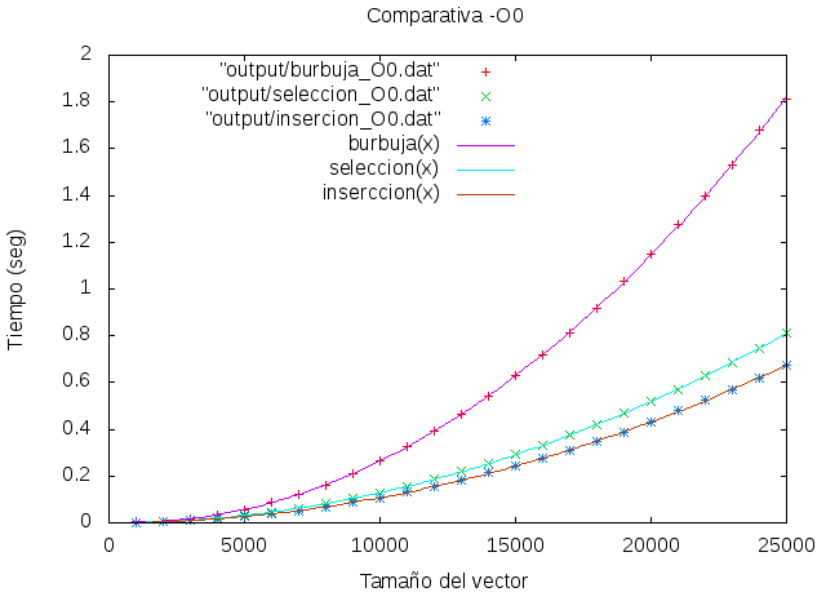



Figura 17: Comaparación entre los distintos algoritmos de ordenación cuadráticos

## 4.2. Comparativa de los algoritmos $nlog(n)$ de ordenación

| Tamaño | Mergesort | Quicksort | Heapesort |
|---|---|---|---|
| 0 | 0.000117755 | 8.7e-05 | 0.000124 |
| 1000 | 0.000265505 | 0.000187 | 0.000268 |
| 2000 | 0.000509884 | 0.000294 | 0.000422 |
| 3000 | 0.000580912 | 0.000394 | 0.000585 |
| 4000 | 0.000807877 | 0.000534 | 0.000748 |
| 5000 | 0.00107341 | 0.000616 | 0.00092 |
| 6000 | 0.00103389 | 0.000727 | 0.001085 |
| 7000 | 0.00124508 | 0.000843 | 0.001164 |
| 8000 | 0.00147816 | 0.00097 | 0.001303 |
| 9000 | 0.00174622 | 0.001065 | 0.00146 |
| 10000 | 0.002063 | 0.001219 | 0.001627 |
| 11000 | 0.002352 | 0.001415 | 0.001808 |
| 12000 | 0.002076 | 0.001446 | 0.001986 |
| 13000 | 0.002313 | 0.001549 | 0.00212 |
| 14000 | 0.002485 | 0.001683 | 0.002294 |
| 15000 | 0.002708 | 0.001781 | 0.002464 |
| 16000 | 0.003018 | 0.001935 | 0.002641 |
| 17000 | 0.00331 | 0.002054 | 0.002815 |
| 18000 | 0.003453 | 0.002157 | 0.00305 |
| 19000 | 0.00379 | 0.002295 | 0.003175 |
| 20000 | 0.004064 | 0.002443 | 0.003364 |
| 21000 | 0.004262 | 0.002554 | 0.003576 |
| 22000 | 0.004612 | 0.002758 | 0.003686 |
| 23000 | 0.004959 | 0.002788 | 0.003889 |
| 24000 | 0.005205 | 0.002935 | 0.004078 |

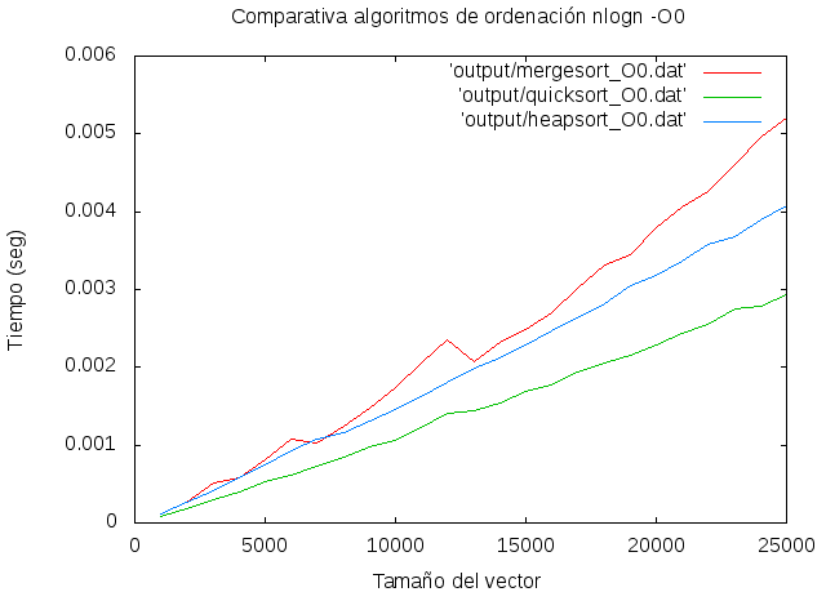



Figura 18: Comaparación entre los distintos algoritmos de ordenación *nlogn*

## 4.3. Comparativa de los algoritmos de ordenación

| Tamaño | Burbuja | Inserción | Selección | Mergesort | Quicksort | Heapesort |
|---|---|---|---|---|---|---|
| 0 | 0.004246 | 0.00112 | 0.00135595 | 0.000117755 | 8.7e-05 | 0.000124 |
| 1000 | 0.008201 | 0.004292 | 0.00524505 | 0.000265505 | 0.000187 | 0.000268 |
| 2000 | 0.019307 | 0.009728 | 0.011705 | 0.000509884 | 0.000294 | 0.000422 |
| 3000 | 0.034225 | 0.017294 | 0.020711 | 0.000580912 | 0.000394 | 0.000585 |
| 4000 | 0.056845 | 0.026957 | 0.03227 | 0.000807877 | 0.000534 | 0.000748 |
| 5000 | 0.088076 | 0.038868 | 0.046363 | 0.00107341 | 0.000616 | 0.00092 |
| 6000 | 0.120856 | 0.052248 | 0.062962 | 0.00103389 | 0.000727 | 0.001085 |
| 7000 | 0.161862 | 0.068156 | 0.082298 | 0.00124508 | 0.000843 | 0.001164 |
| 8000 | 0.210286 | 0.086587 | 0.104299 | 0.00147816 | 0.00097 | 0.001303 |
| 9000 | 0.264146 | 0.107304 | 0.128745 | 0.00174622 | 0.001065 | 0.00146 |
| 10000 | 0.323677 | 0.130252 | 0.156038 | 0.002063 | 0.001219 | 0.001627 |
| 11000 | 0.394902 | 0.15566 | 0.185971 | 0.002352 | 0.001415 | 0.001808 |
| 12000 | 0.463354 | 0.182503 | 0.218305 | 0.002076 | 0.001446 | 0.001986 |
| 13000 | 0.541129 | 0.212983 | 0.253628 | 0.002313 | 0.001549 | 0.00212 |
| 14000 | 0.628463 | 0.243724 | 0.291057 | 0.002485 | 0.001683 | 0.002294 |
| 15000 | 0.718559 | 0.2767 | 0.331193 | 0.002708 | 0.001781 | 0.002464 |
| 16000 | 0.814468 | 0.309732 | 0.374365 | 0.003018 | 0.001935 | 0.002641 |
| 17000 | 0.919391 | 0.346489 | 0.419854 | 0.00331 | 0.002054 | 0.002815 |
| 18000 | 1.03422 | 0.387793 | 0.469411 | 0.003453 | 0.002157 | 0.00305 |
| 19000 | 1.14783 | 0.432006 | 0.518467 | 0.00379 | 0.002295 | 0.003175 |
| 20000 | 1.27545 | 0.479765 | 0.571629 | 0.004064 | 0.002443 | 0.003364 |
| 21000 | 1.39817 | 0.524347 | 0.62718 | 0.004262 | 0.002554 | 0.003576 |
| 22000 | 1.53151 | 0.569582 | 0.68739 | 0.004612 | 0.002758 | 0.003686 |
| 23000 | 1.67702 | 0.620089 | 0.747746 | 0.004959 | 0.002788 | 0.003889 |
| 24000 | 1.81403 | 0.675667 | 0.81211 | 0.005205 | 0.002935 | 0.004078 |

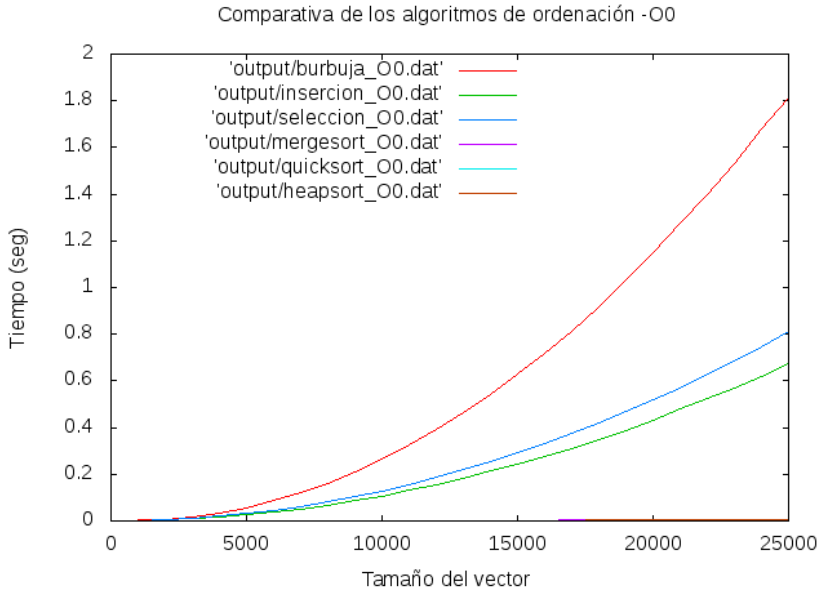



Figura 19: Comaparación entre los distintos algoritmos de ordenación *nlogn*

De los resultados obtenidos podemos ver de forma clara como el algoritmo de burbuja es el que desempeña la tarea de forma considerablemente más lenta.

Entre los algoritmos de selección e insercción no encontramos mucha diferencia aunque son notoriamente más eficientes que los de orden cuadrático como se observa en la comparativa.

Finalmente los algoritmos de orden $O(n \cdot log(n))$ son mucho más eficientes que el resto aunque podemos observar como el algoritmo quicksort ha sido el que ha obtenido mejores resultados pese a ser del mismo orden de eficiencia.

En el algoritmo de hanoi podemos observar como al ser $O(2^n)$ el tiempo que tarda en ejecutarse crece rapidamente a partir de un punto lo cual nos permite realizar únicamente pocas iteraciones.

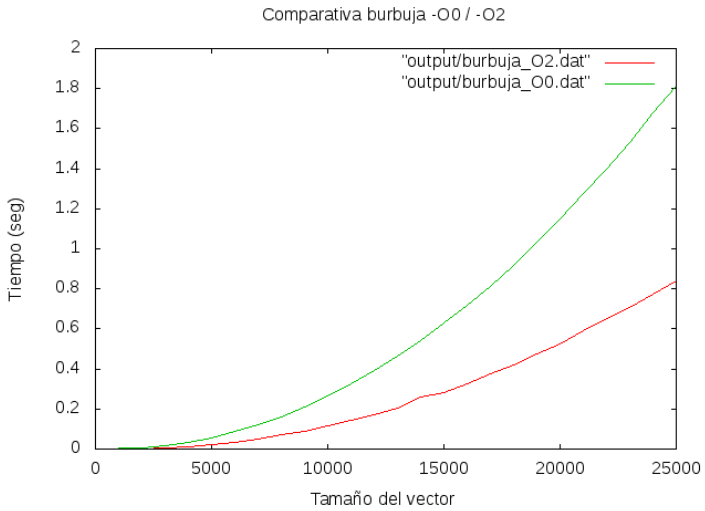
La suma de los cuadrados de los residuos de los ajustes realizados ha sido la siguiente:

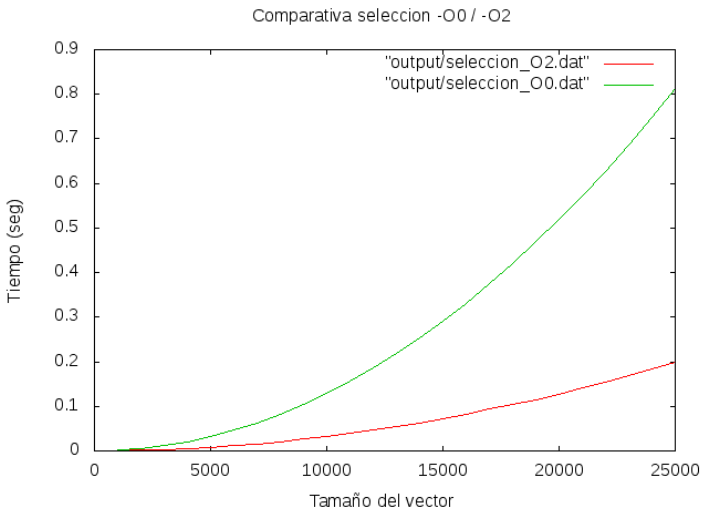
| Algoritmo | Suma residual |
|---|---|
| Burbuja | 9.84713e-05 |
| Inserción | 7.19712e-05 |
| Selección | 8.779e-06 |
| Mergesort | 1.54282e-05 |
| Heapsort | 7.24527e-06 |
| Quicksort | 2.17388e-06 |
| Floyd | 0.000654703 |
| Hanoi | 1.17475e-05 |

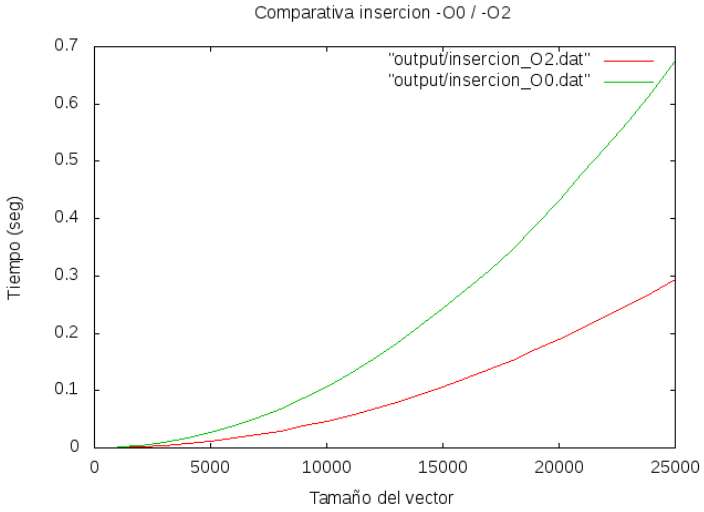
Cuadro 2: Ajuste los resultados

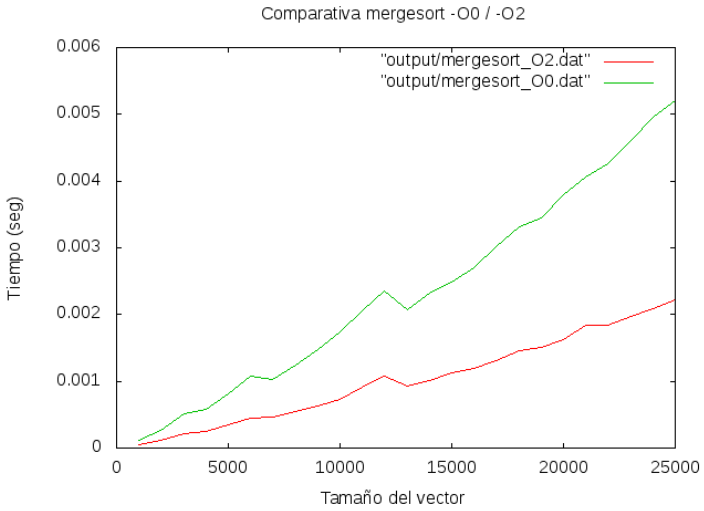
Lo cual nos indica que los resultados que hemos obtenido realemte se ajustan bien a lo esperado según la notación *Big O*

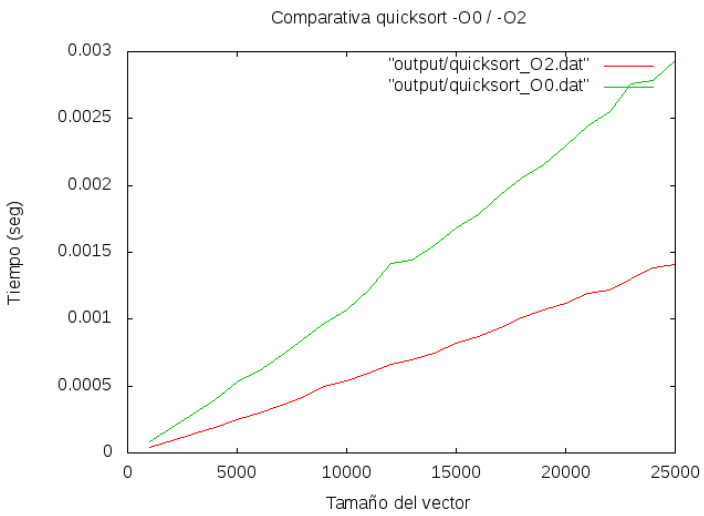
Tambíen he realizado medidas con otros ordenes de eficiciencia. En este caso expongo los resultados con la eficiencia $-O2$. Como se puede apreciar en las gráficas, los resultados son mucho mejores y destacan el caso de la ordenación por burbuja donde se consigue que, aun siendo cuadrática, el tiempo de ejecución se reduzca más de la mitad y el caso del algoritmo de Hanoi, donde pasamos de casi 14 segundos en los últimos valores a tan solo 5.

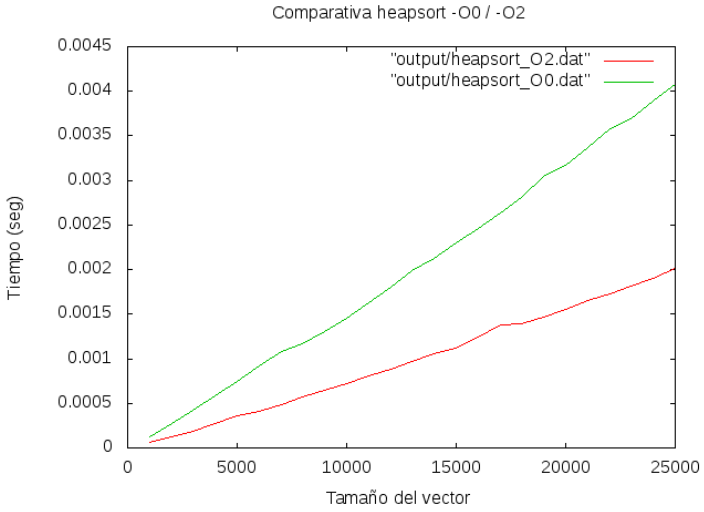


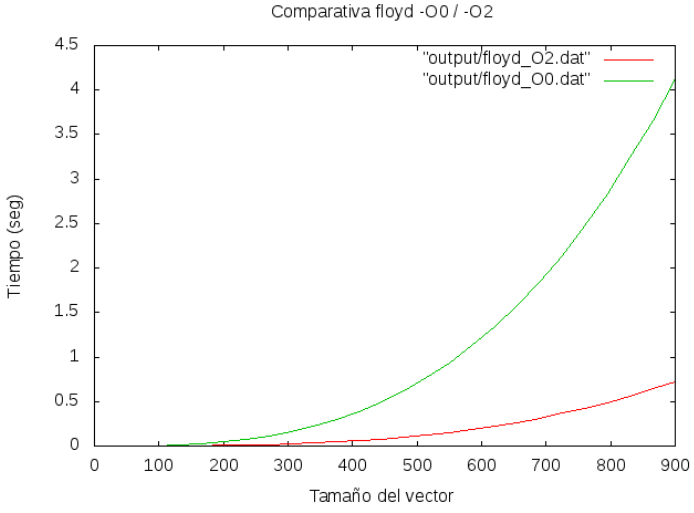

Comparativa seleccion -O0 / -O2
"output/seleccion_O2.dat"
"output/seleccion_O0.dat"
Tiempo (seg)
Tamaño del vector
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
0
5000
10000
15000
20000
25000

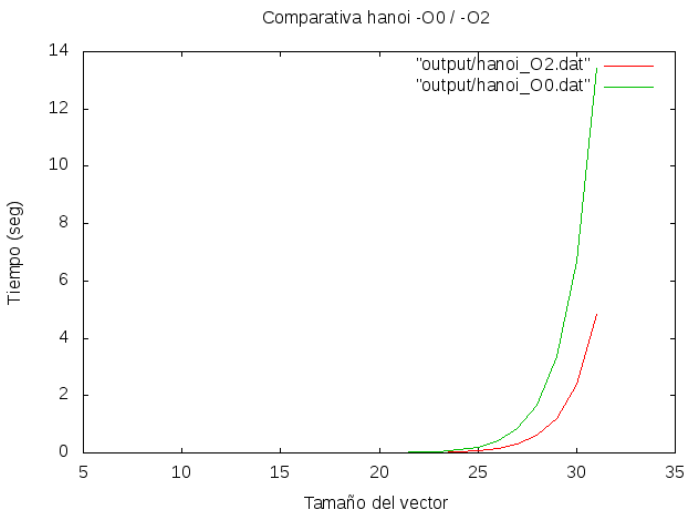

Comparativa insercion -O0 / -O2
"output/insercion_O2.dat"
"output/insercion_O0.dat"
Tiempo (seg)
Tamaño del vector
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0
5000
10000
15000
20000
25000


Comparativa mergesort -O0 / -O2
"output/mergesort_O2.dat"
"output/mergesort_O0.dat"
Tiempo (seg)
Tamaño del vector
0
0.001
0.002
0.003
0.004
0.005
0.006
0
5000
10000
15000
20000
25000


Comparativa quicksort -O0 / -O2
"output/quicksort_O2.dat"
"output/quicksort_O0.dat"
Tiempo (seg)
Tamaño del vector
0.003
0.0025
0.002
0.0015
0.001
0.0005
0
0
5000
10000
15000
20000
25000


Comparativa heapsort -O0 / -O2
"output/heapsort_O2.dat"
"output/heapsort_O0.dat"
Tiempo (seg)
Tamaño del vector
0.0045
0.004
0.0035
0.003
0.0025
0.002
0.0015
0.001
0.0005
0
0
5000
10000
15000
20000
25000


Comparativa floyd -O0 / -O2
"output/floyd_O2.dat"
"output/floyd_O0.dat"
Tiempo (seg)
Tamaño del vector
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
0
100
200
300
400
500
600
700
800
900


Comparativa hanoi -O0 / -O2
"output/hanoi_O2.dat"
"output/hanoi_O0.dat"
Tiempo (seg)
Tamaño del vector
0
2
4
6
8
10
12
14
5
10
15
20
25
30
35