

UNIVERSIDAD DE GRANADA

# Análisis de Eficiencia de Algoritmos

---

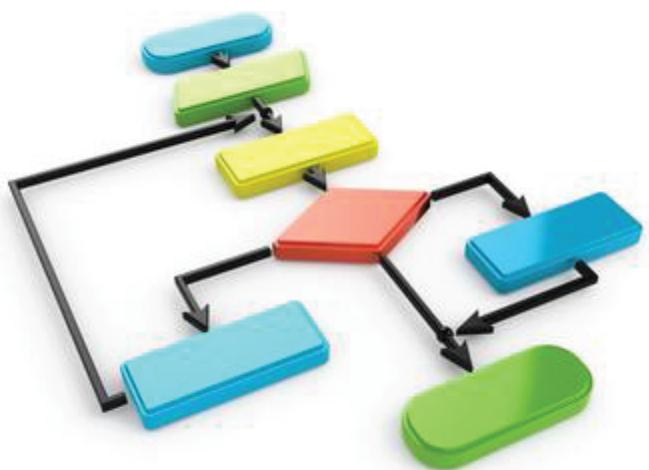
## Algorítmica

**Aarón Bueno Rodríguez  
Bryan Moreno Picamán  
Miguel Ángel Rodríguez Serrano**



# Algoritmos

## Eficiencia $n^2$



---

## Algoritmo

---

## Burbuja

---

### Explicación del Algoritmo

---

Es un algoritmo de ordenación de orden  $O(n^2)$ .  $n^2$

Este algoritmo utiliza la recursividad para ir dividiendo en dos el vector dado, e ir ordenando cada parte.

No es uno de los algoritmos de ordenación más rápidos, pero tampoco es de los más lentos.

---

### Tiempos

---

Se midió el tiempo con la librería ctime antes y después de llamar al algoritmo (no fue necesario utilizar un bucle para el algoritmo por números demasiado pequeños).

Se creó un script para llamar al ejecutable en un bucle donde la entrada inicial era 1000 y se van aumentando las entradas de 1000 en 1000 y que redirige la salida del número de entradas más los tiempos para esa entrada a un fichero .dat

Se utilizó en gnuplot como función para calcular las constantes mediante regresión:

$$f(x) = a_1 * x * x + a_2 * x + a_3$$

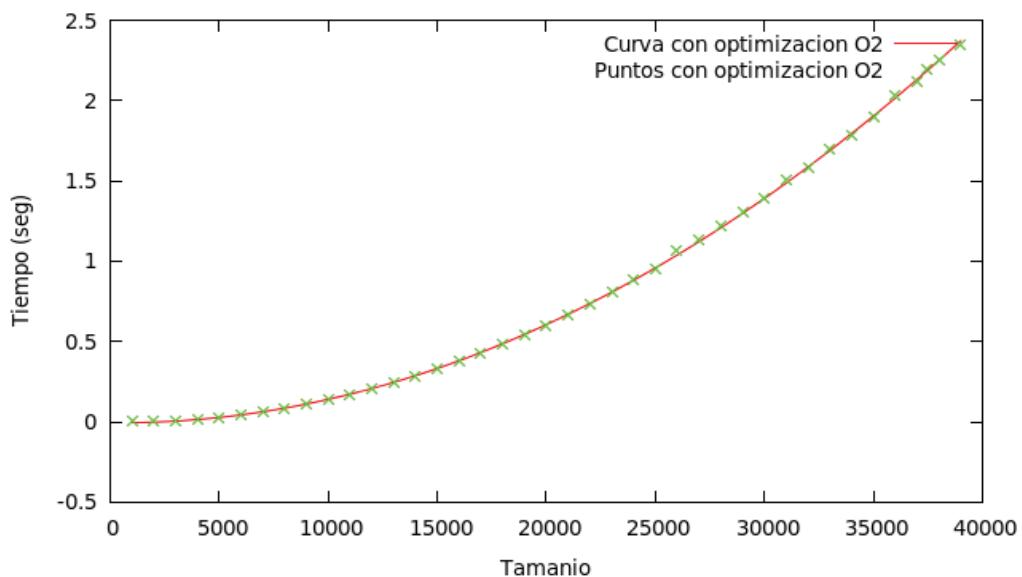
El resultado fue el siguiente:

a1	= 1.60174e-09	+/- 1.21e-11	(0.7555%)
a2	= -1.51444e-06	+/- 4.991e-07	(32.96%)
a3	= -0.00557868	+/- 0.004329	(77.6%)

---

### Representación Gráfica

---



---

## Algoritmo

---

## Inserción

---

### Explicación del Algoritmo

---

Se trata de un algoritmo de ordenación que nos permite realizar de forma recursiva la ordenación de un vector, usa un algoritmo de orden  $O(n^2)$ . Que produce una recurrencia que va aumentando bastante, si no tanto como en algunas de un orden superior, si bastante conforme el valor de 'n' crece.

---

### Tiempos

---

Para la medición de los tiempos, se ha usado un método que mide el tiempo antes y después de cada una de las ejecuciones con diferentes tamaños para el 'n'.

Para el cálculo de grafica han sido tomadas una serie de medidas de tiempo, concretamente unas medidas para n entre 1000 y 51000.

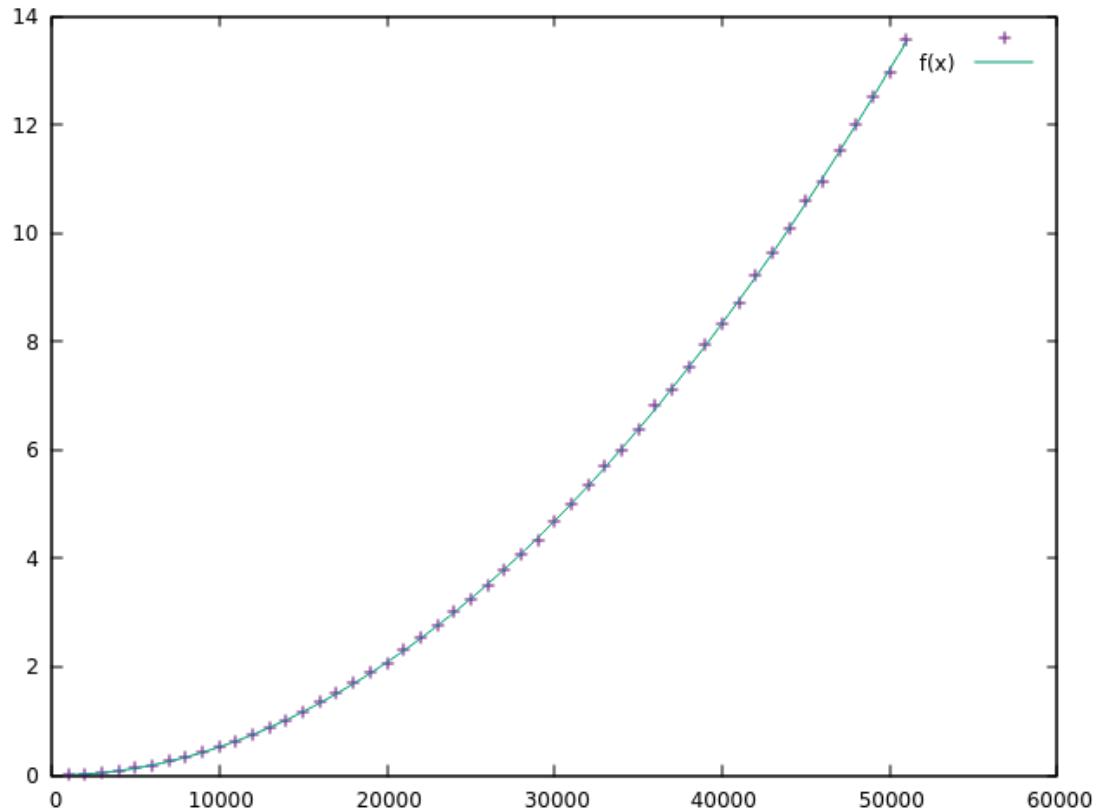
Para esto se ha usado una función  $f(x) = a_0 * (n^2)$  obteniendo un error bastante bajo de:

$$a_0 = 5.20917e-09 \quad +/- 3.027e-12 \quad (0.05811\%)$$

---

### Representación Gráfica

---



Como se puede observar en la gráfica, la función se adapta bastante bien a la crecida de los datos pero con algunas pequeñas variaciones, pero no tan pronunciadas como en otros algoritmos.

---

## Algoritmo

---

## Selección

---

### Explicación del Algoritmo

---

Recorre una lista de  $n$  elementos, una vez que localiza al menor elemento, lo intercambia con el primer elemento de la lista, dejando de esta manera en el inicio de la lista los elementos ordenados de menor a mayor.

Con el fin de ordenar dicha lista, este algoritmo tiene que realizar siempre el mismo número de comparaciones:

$$c(n) = \frac{n^2 - n}{2}$$

Por tanto, al ser  $c(n)$  el número de comparaciones y no depender del orden de los términos, sino del tamaño de los mismos, se puede afirmar que la cota ajustada asintótica del número de comparaciones pertenece al orden  $n^2$

$$\theta(c(n)) = n^2$$

---

### Tiempos

---

Con el fin de conocer de la manera más objetiva y fiel posible el tiempo de ejecución del algoritmo, se pasa a usar la biblioteca **ctime**, gracias a la cual, guardaremos en dos variables (tantes y tdespues) el valor del reloj antes y después respectivamente de la ejecución del algoritmo. Posteriormente, realizando el cociente entre la diferencia de ambas variables y la constante CLOCKS\_PER\_SEC, obtendremos el tiempo en segundos que tardó el algoritmo en ejecutarse.

### Representación Gráfica

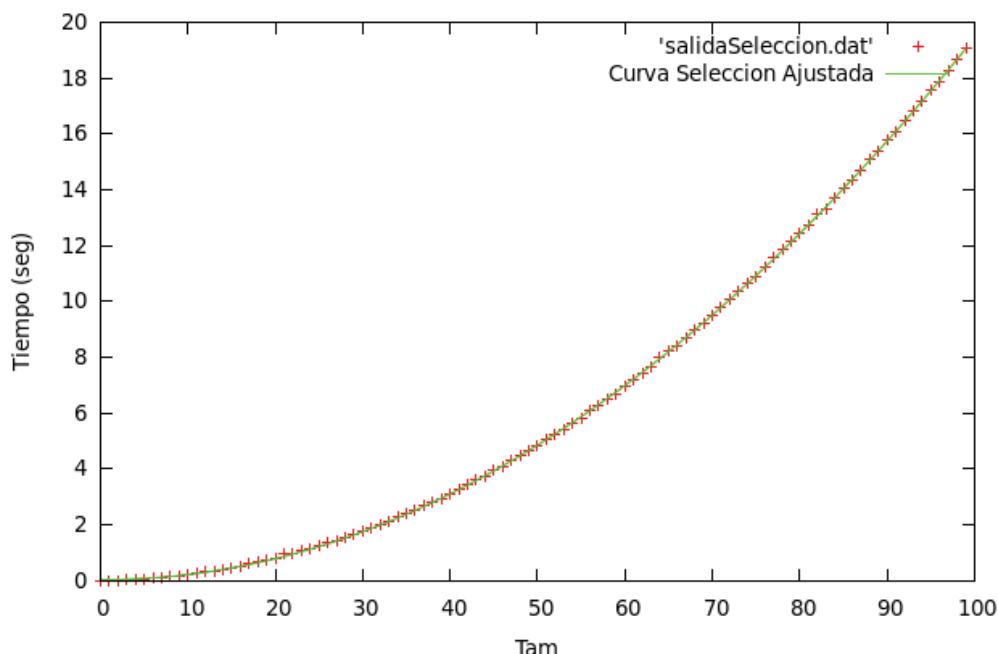
---

Para la obtención de la representación gráfica, se ha tomado una muestra de 100 medidas de tiempo. Ello ha sido posible usando la función:

$$f(x) = a_0 \cdot x^2 + a_1 \cdot x + a_2$$

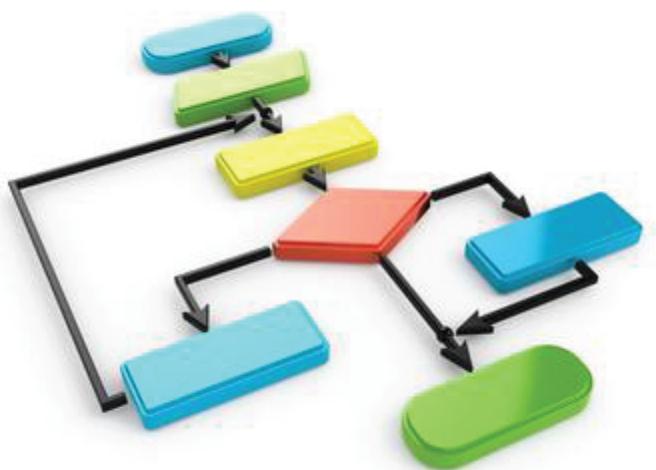
Para los parámetros  $a$  se obtuvieron los siguientes valores con su correspondiente margen de error:

<b>a0</b>	= 0.00195562	+/- 3.53e-06	(00.1805%)
<b>a1</b>	= -0.00148445	+/- 0.0003612	(24.3300%)
<b>a2</b>	= 0.0400999	+/- 0.007737	(19.2900%)



# Algoritmos

## Eficiencia $n \cdot \log(n)$



Algoritmo	Mergesort
-----------	-----------

### Explicación del Algoritmo

Es un algoritmo de ordenación de orden  $O(n \cdot \log(n))$ .

Es un algoritmo de ordenamiento externo estable que está basado en la técnica “Divide y vencerás”.

La eficiencia de este algoritmo es mucho mayor que la de cualquiera de orden  $O(n^2)$

### Tiempos

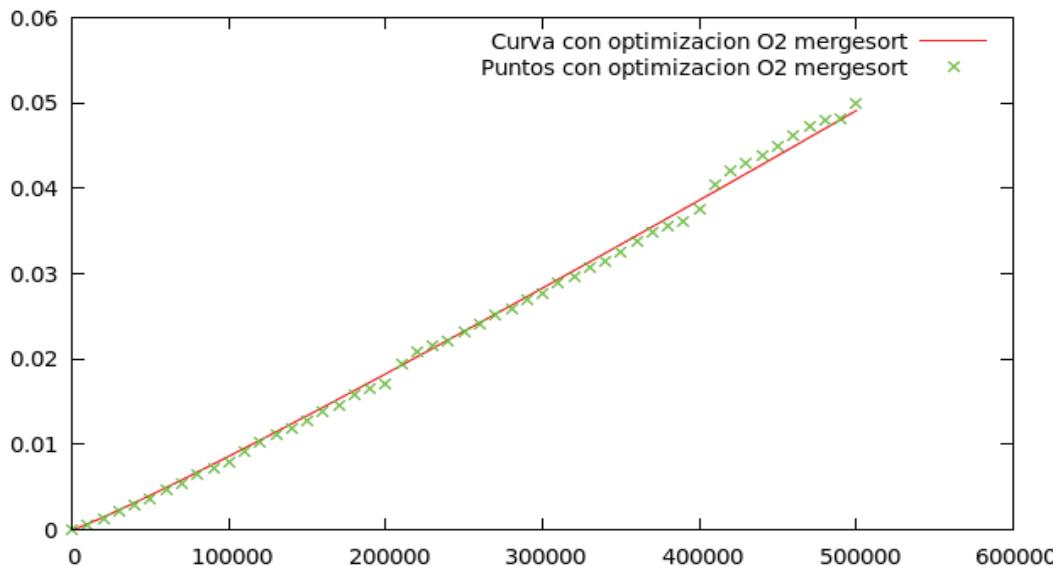
Se midió el tiempo con la librería ctime antes y después de llamar al algoritmo. Se utilizó un bucle para llamar al algoritmo por intentar suavizar los puntos que salían dispares (y después dividir la salida, claro), además de salir números pequeños debido a la gran eficiencia que tiene.

Se creó un script para llamar al ejecutable en un bucle donde la entrada inicial era 100 y se van aumentando las entradas de 10000 en 10000 hasta obtener 100 mediciones (y aún así hace las mediciones bastante rápido) y que redirige la salida del número de entradas más los tiempos para esa entrada a un fichero .dat

Se utilizó en gnuplot como función para calcular las constantes mediante regresión:

$$f(x) = a_1 \cdot x \cdot \log(x)$$

### Representación Gráfica



---

## Algoritmo

---

## QuickSort

---

---

### Explicación del Algoritmo

---

Se trata de un algoritmo de ordenación basado en la ecuación:

$$T(n) = T(1) + T(n - 1) + c * n$$

Permite realizar de forma recursiva la ordenación de un vector, por medio de un algoritmo de orden  $O(n \log n)$ . Esto produce una recurrencia que aumenta pero de una forma bastante menos agresiva si la comparamos con otros algoritmos.

---

## Tiempos

---

Para la medición de los tiempos, se ha usado un método que mide el tiempo antes y después de cada una de las ejecuciones con diferentes tamaños para el 'n'.

Para el cálculo de grafica han sido tomadas una serie de medidas de tiempo, concretamente unas medidas para n entre 100000 y 5100000.

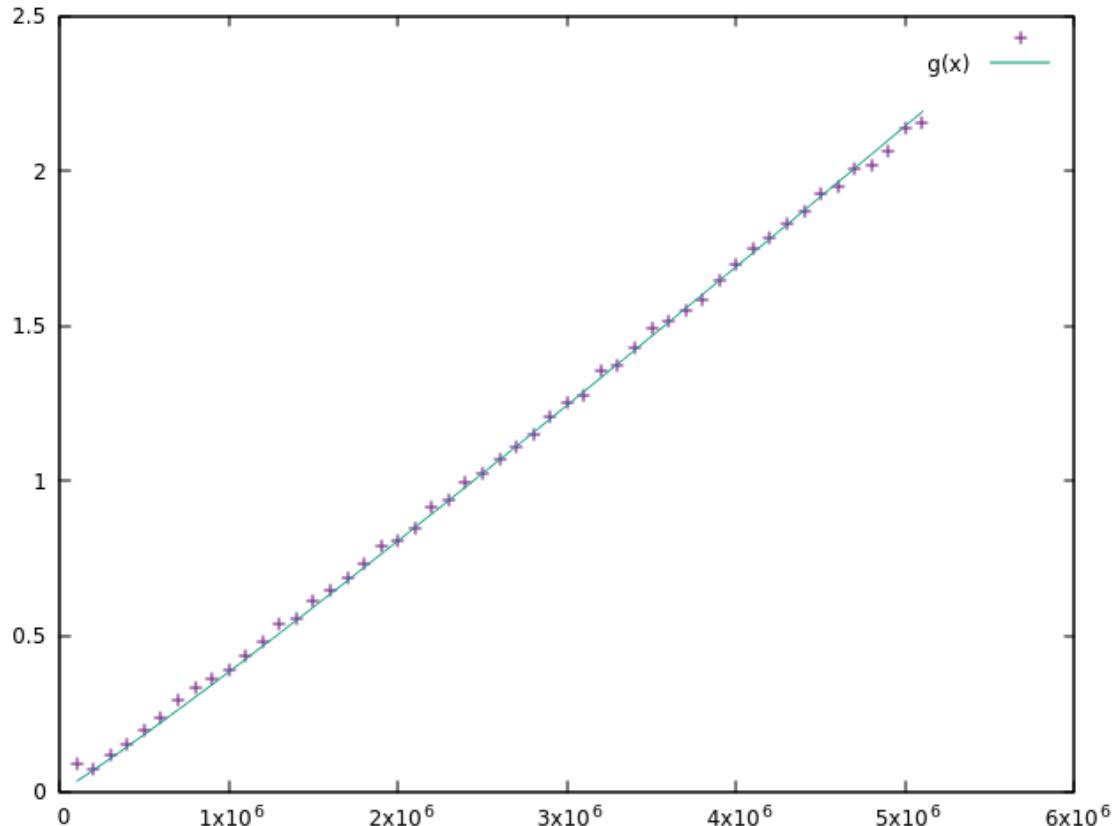
Para esto se ha usado una función  $g(x)=a_0*(n \log n)$ . obteniendo un error bastante bajo de:

$$a_0 = 2.78128e-08 \quad +/- 5.689e-11 \quad (0.2046\%)$$

---

## Representación Gráfica

---



---

Como se puede observar en la gráfica, la función se adapta bastante bien a la crecida de los datos pero con algunas pequeñas variaciones.

---

Algoritmo	Heapsort
-----------	----------

### Explicación del Algoritmo

Almacena todos los elementos del vector a ordenar en un *montículo* (heap), y luego de manera recursiva extrae el nodo que queda como raíz del montículo (cima) obteniendo así el conjunto ordenado.

### Tiempos

Con el fin de conocer de la manera más objetiva y fiel posible el tiempo de ejecución del algoritmo, se pasa a usar la biblioteca ctime, gracias a la cual, guardaremos en dos variables (tantes y tdespues) el valor del reloj antes y después respectivamente de la ejecución del algoritmo. Posteriormente, realizando el cociente entre la diferencia de ambas variables y la constante CLOCKS\_PER\_SEC, obtendremos el tiempo en segundos que tardó el algoritmo en ejecutarse.

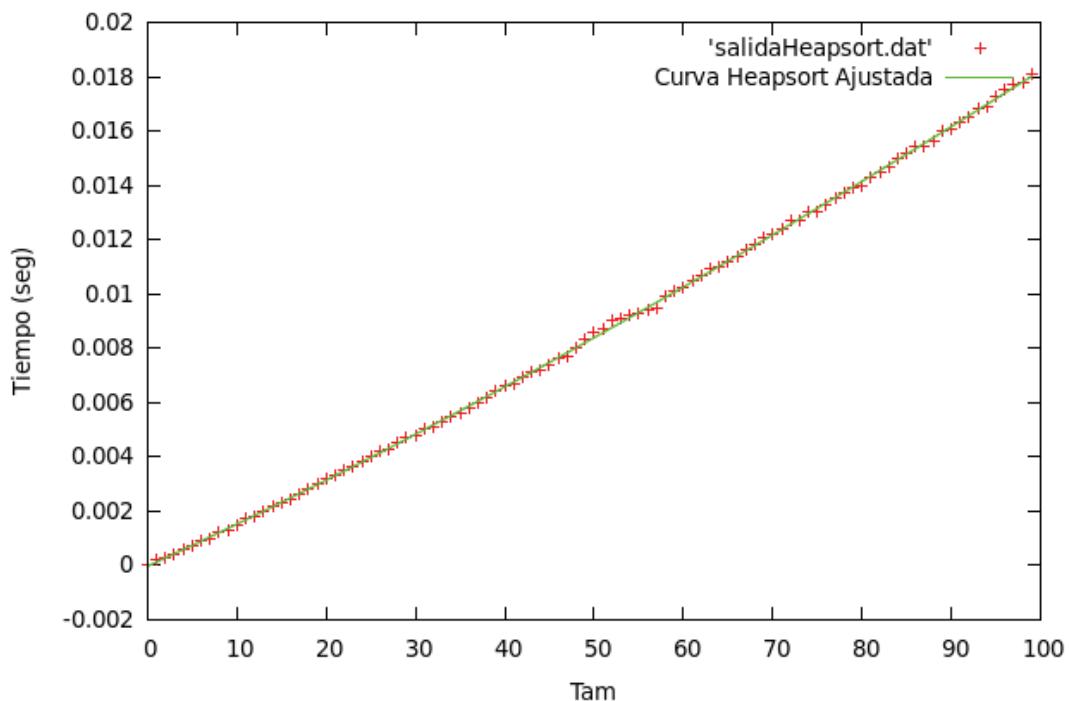
### Representación Gráfica

Para la obtención de la representación gráfica, se ha tomado una muestra de 100 medidas de tiempo. Ello ha sido posible usando la función:

$$f(x) = n \cdot \log(n)$$

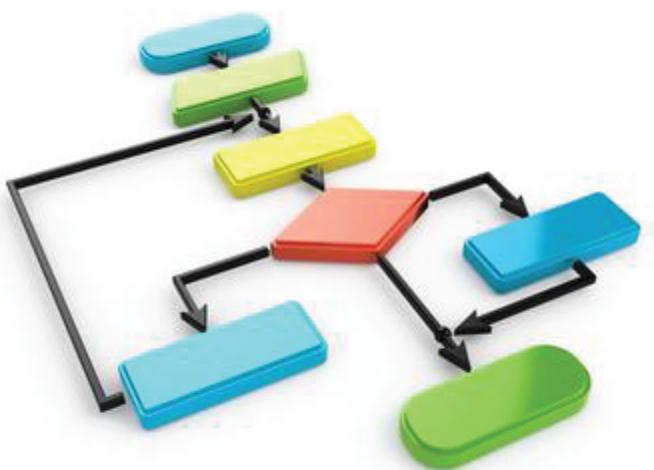
Para los parámetros  $a$  se obtuvieron los siguientes valores con su correspondiente margen de error:

<b>a0</b>	= -5.53906e-10	+/- 4.015e-10	(72.49%)
<b>a1</b>	= 3.67307e-07	+/- 6.049e-08	(16.47%)
<b>a2</b>	= 0.00015136	+/- 2.57e-06	(1.698%)
<b>a3</b>	= -3.36408e-05	+/- 2.923e-05	(86.89%)



# Algoritmos

## Eficiencia $n^3$



---

<b>Algoritmo</b>	<b>Floyd</b>
------------------	--------------

---

### Explicación del Algoritmo

Es un algoritmo de análisis sobre grafos de orden  $O(n^3)$  para encontrar el camino en grafos dirigidos ponderados. Es un ejemplo de programación dinámica.

Es un algoritmo muy poco eficiente, pero no existen otros algoritmos mejores para el uso que se le da a este algoritmo.

---

### Tiempos

---

Se midió el tiempo con la librería ctime antes y después de llamar al algoritmo.

Se creó un script para llamar al ejecutable en un bucle donde la entrada inicial era 150 y se van aumentando las entradas de 150 en 150 hasta 3750 (para tener 25 mediciones, era imposible obtener más debido a que el algoritmo es demasiado costoso)

Se utilizó en gnuplot como función para calcular las constantes mediante regresión:

$$f(x) = a_1 * x * x * x + a_2 * x * x + a_3 * x + a_4 * x$$

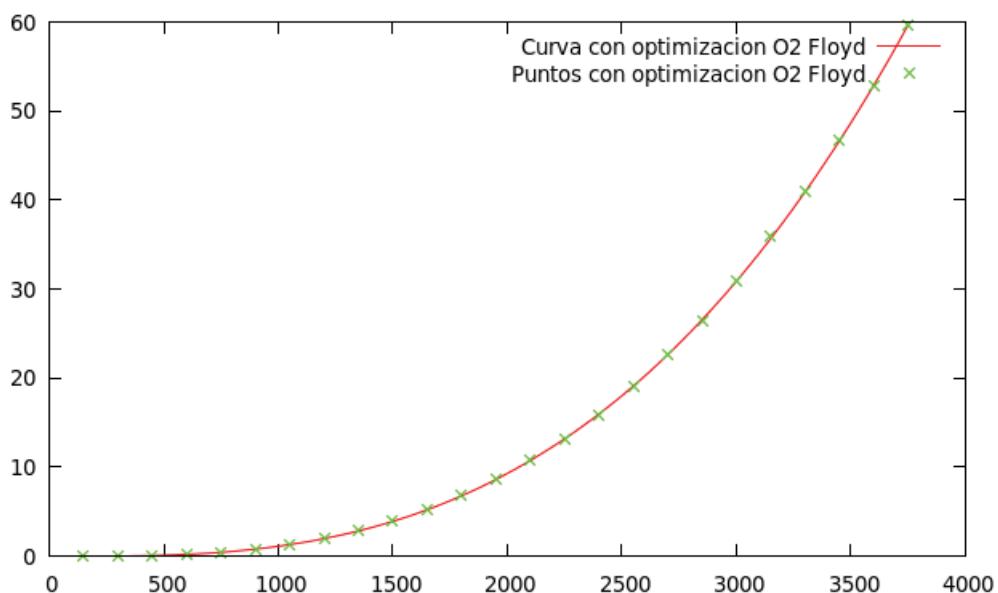
El resultado es el siguiente:

a1	= 1.04639e-09	+/- 1.202e-11	(1.149%)
a2	= 4.36133e-07	+/- 7.123e-08	(16.33%)
a3	= -0.00046293	+/- 0.0001208	(26.09%)
a4	= 0.094715	+/- 0.05548	(58.58%)

---

### Representación Gráfica

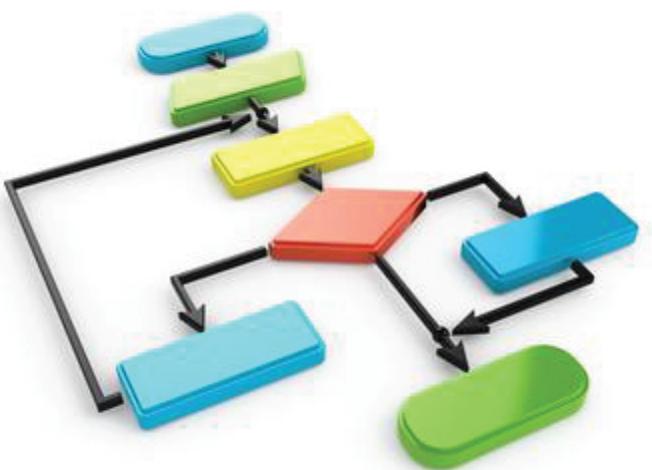
---



# Algoritmos

## Eficiencia $\varphi$

$$\frac{1 + \sqrt{5}}{2}$$



---

## Algoritmo

---

## Fibonacci

---

### Explicación del Algoritmo

---

Se trata de un algoritmo, basado en la ecuación:

$$f_n = f_{n-1} + f_{n-2}$$

Que nos permite calcular de forma recursiva la Sucesión de Fibonacci, se trata de un algoritmo de orden  $O((\frac{1+\sqrt{5}}{2})^n)$ . Esto produce una recurrencia que aumenta bastante en tiempo de ejecución conforme aumentan el tamaño de los numero (n)

---

## Tiempos

---

Para la medición de los tiempos, se ha usado un método que mide el tiempo antes y después de cada una de las ejecuciones con diferentes tamaños para el 'n'.

Para el cálculo de grafica han sido tomadas una serie de medidas de tiempo, concretamente unas medidas para n entre 0 y 50.

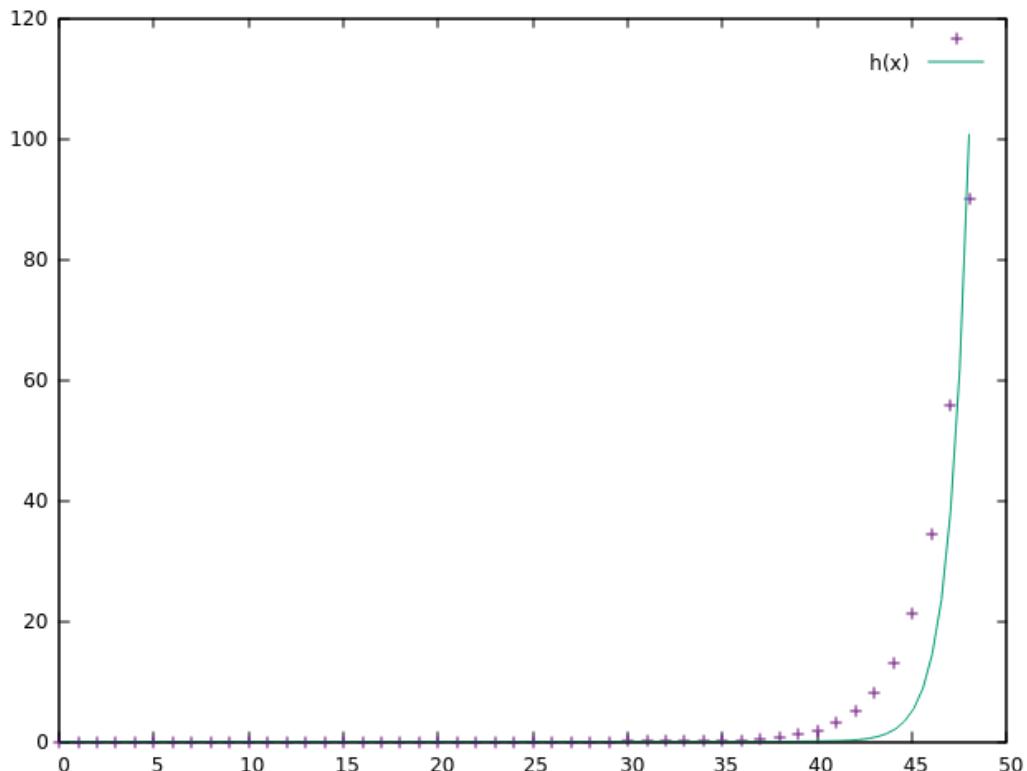
Para esto se ha usado una función  $h(x)=a_0*1.618$  obteniendo un error de:

$$a_0 = 8.86664e-20 \quad +/- 4.386e-21 \quad (4.947\%)$$

---

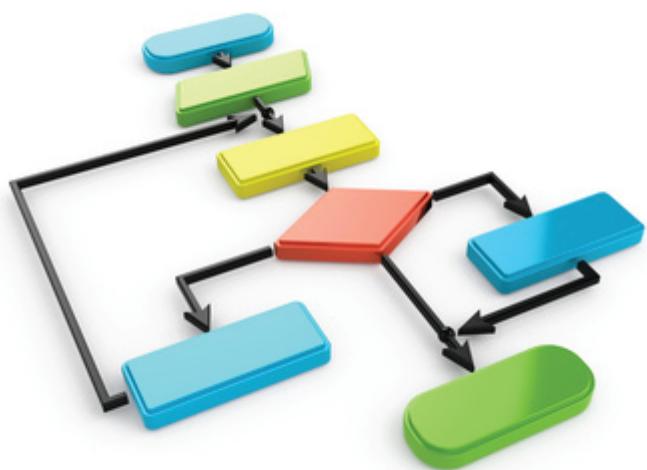
### Representación Gráfica

---



Como se puede observar en la grafica, la adaptación se mejora conforme el tamaño crece, pero no es tan buena para números relativamente bajos.

# Comparaciones



---

## Comparación Diferentes PCs

---

### Propiedades

---

- Mismo algoritmo.
- Misma optimización de código ( en este caso, hemos utilizado -O2 ).

#### Diferentes PCs:

- PC1: Intel(R) Core(TM) i5-2300 CPU @ 2.80GHz | Ram: 4Gb | Windows 8.1 x64
- PC2: Intel(R) Celeron(R) CPU 2.80GHz | Ram: 8 Gb | Ubuntu 10.04 x64

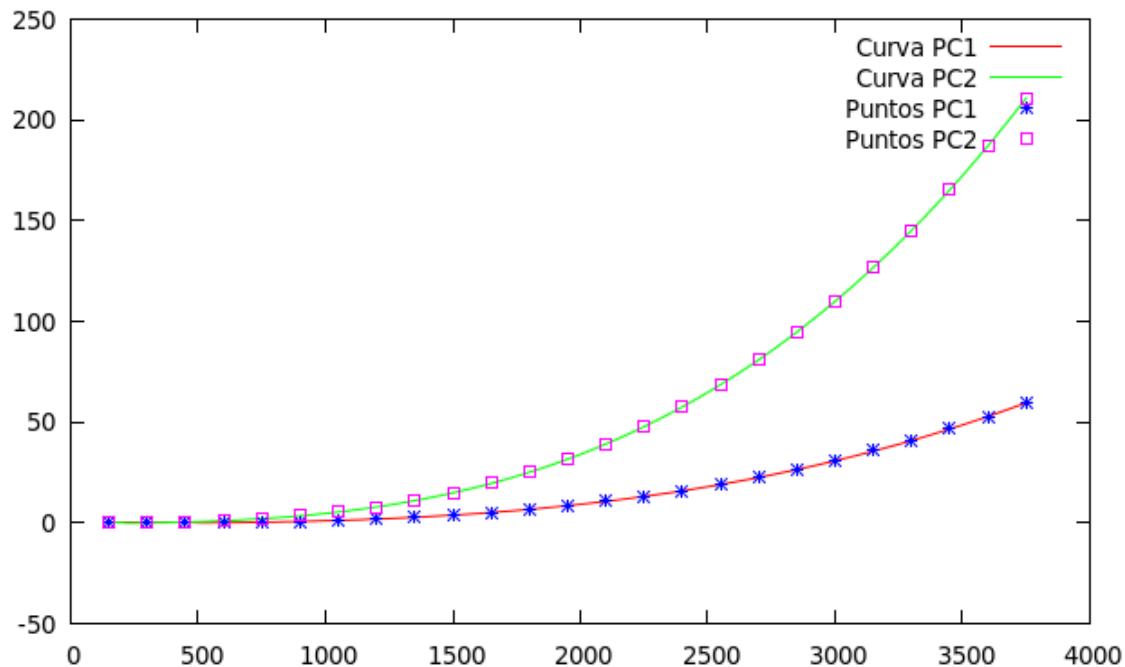
---

### Representación Gráfica

---

Para esta demostración utilizamos el algoritmo de Floyd.

El resultado es el siguiente:



---

## Comparación Diferente optimización

---

### Propiedades

---

- Mismo algoritmo.
- Distinta optimización de código:**
  - Sin optimización.
  - Optimización O2.
- Mismo PC.

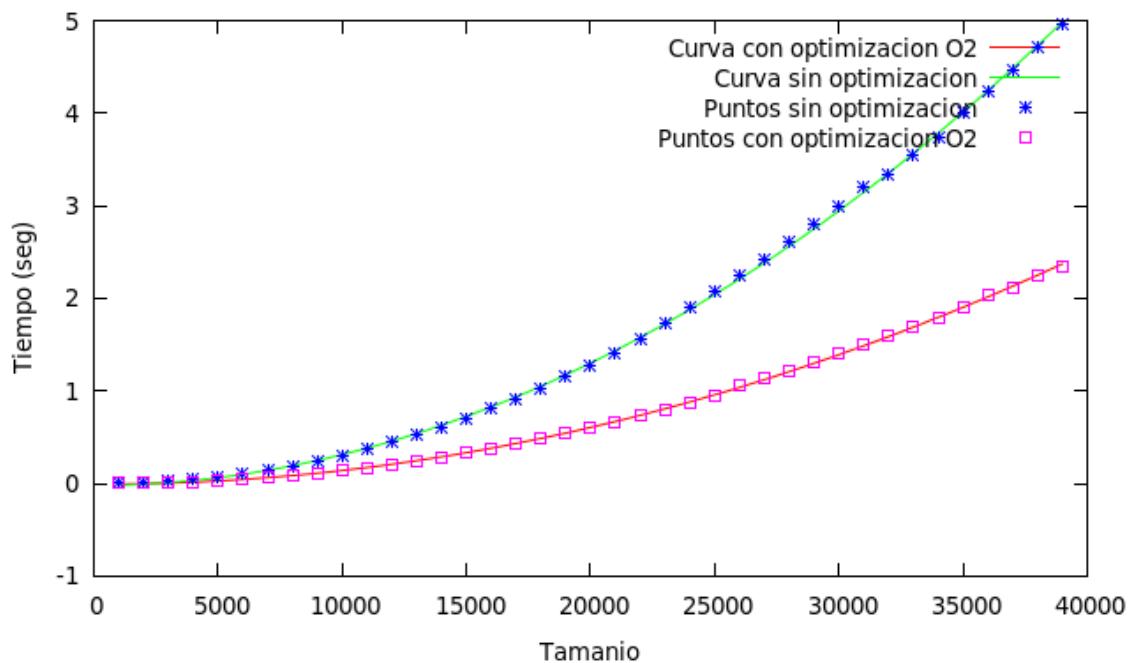
---

### Representación Gráfica

---

Para esta demostración utilizamos el algoritmo de burbuja.

El resultado es el siguiente:



---

## Comparación Algoritmos de ordenación

---

### Propiedades

---

#### -Distintos algoritmos de ordenación:

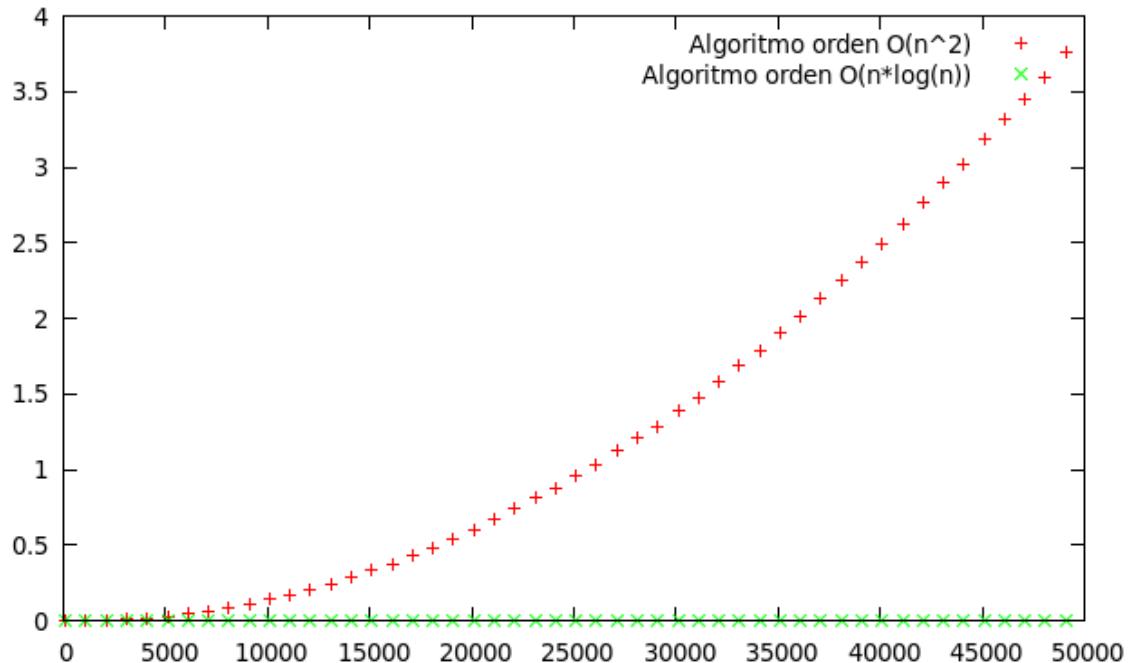
- Inserción.
- Selección.
- Burbuja.
- Mergesort.
- Quicksort.
- Heapsort.
- Mismo PC.
- Misma optimización (en este caso, O2).

---

### Representación Gráfica

---

**Problema:** los algoritmos  $O(n \cdot \log(n))$  son demasiado eficientes en comparación con los  $O(n^2)$ , por lo que no se ve nada claro un gráfico con estos dos algoritmos de diferente eficiencia:



---

## Comparación Algoritmos de ordenación

---

### Solución

---

Crear dos gráficas, una comparando los de  $O(n^2)$  y otro con los  $O(n \log n)$

### Representación Gráfica

---

