

**Nombre:** Antonio Carlos Perea Parras

**DNI:**45944310Q

**Grupo:** A2

Práctica 1

## EFICIENCIA

## Hardware usado



# macOS Mojave

Versión 10.14

MacBook Pro (Retina, 15 pulgadas, mediados de 2015)

Procesador 2,5 GHz Intel Core i7

Memoria 16 GB 1600 MHz DDR3

Disco de arranque Macintosh HD

Gráficos AMD Radeon R9 M370X 2048 MB  
Intel Iris Pro 1536 MB

Número de serie C02Q8121G8WP

### Ejercicio 1 : Ordenación de la burbuja

El siguiente código realiza la ordenación mediante el algoritmo de la burbuja:

```
void ordenar(int *v, int n) {  
    for (int i=0; i<n-1; i++)  
        for (int j=0; j<n-i-1; j++)  
            if (v[j]>v[j+1]) {  
                swap(v[j],v[j+1]);  
                //alternativa al swap  
                /*int aux = v[j]; //incluir algorithm  
                v[j] = v[j+1];  
                v[j+1] = aux;*/  
            }  
}
```

Línea 3. Hay 4 OE: una asignación, una resta, una comparación y un incremento.

Línea 4. Hay 5 OE: igual que la línea anterior, pero se realizan dos restas.

Línea 5. Hay 4 OE: dos accesos a un vector, una suma y una comparación.

Línea 6. Hay 2 OE: asignación y acceso al vector.

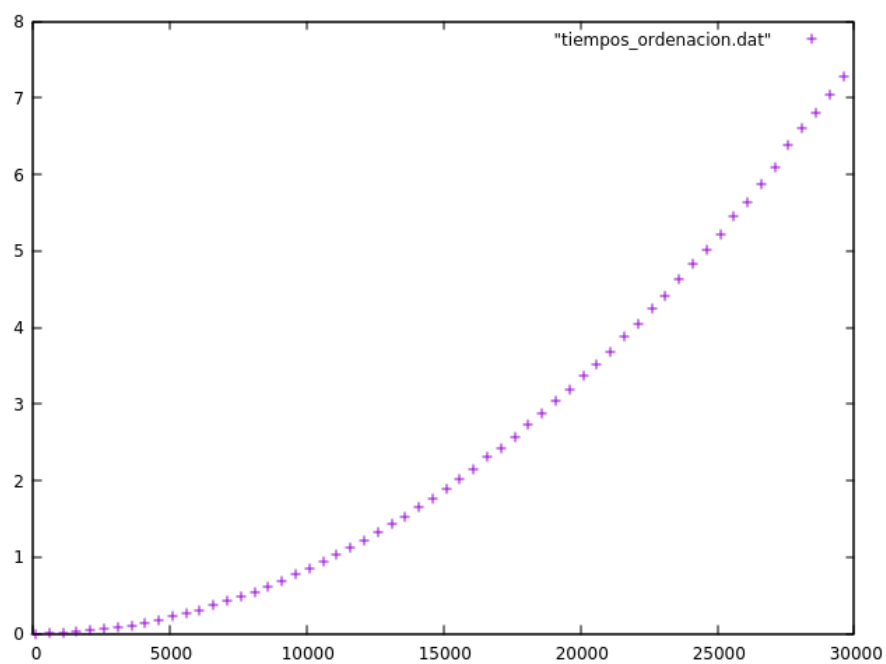
Línea 7. Hay 4 OE: dos accesos, suma y asignación.

Línea 8. Hay 3 OE: acceso, suma y asignación.

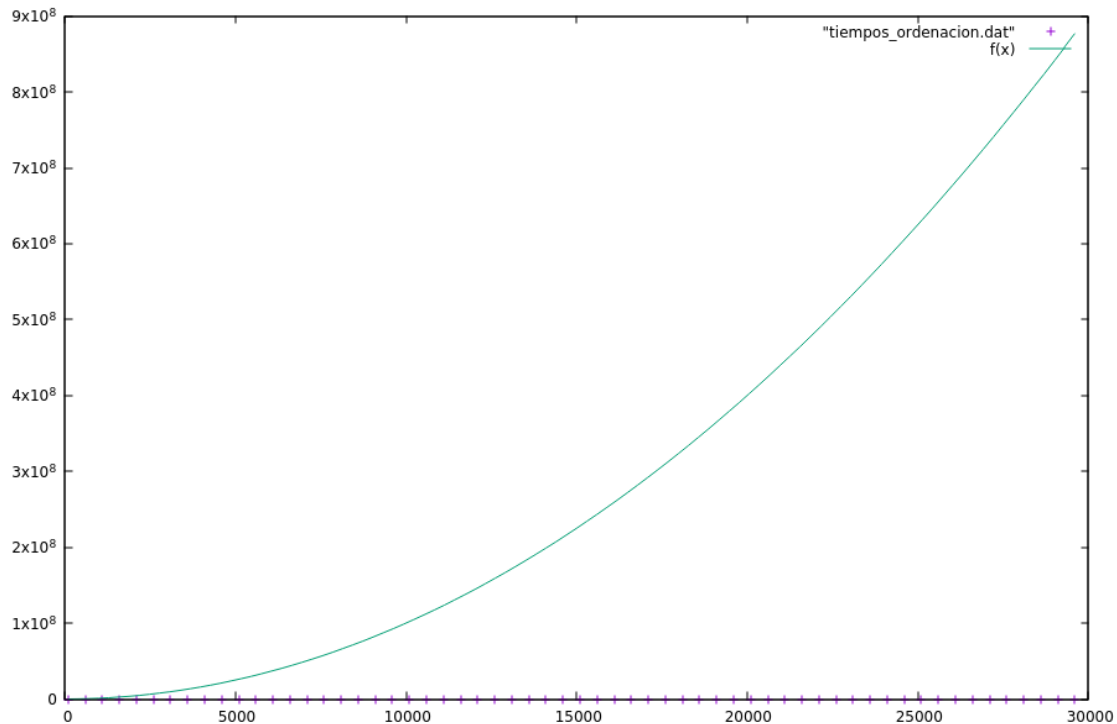
$$\begin{aligned} T(n) &= 1 + \sum_{i=0}^{n-2} \left( 4 + 1 + \sum_{j=0}^{n-i-2} 18 \right) = 1 + \sum_{i=0}^{n-2} (5 + 18(n-i-1)) = 1 + \sum_{i=0}^{n-2} 18n - \sum_{i=0}^{n-2} 18i - \sum_{i=0}^{n-2} 13 = \\ &= 1 + 18n(n-1) - 18 \left( \frac{0 + (n-2)}{2} \cdot (n-1) \right) - 13(n-1) = 9n^2 - \frac{111}{2}n - \frac{21}{2}. \end{aligned}$$

Por tanto, afirmamos que  $T(n) \in O(n^2)$ , y el algoritmo de ordenación burbuja es de orden de eficiencia  $O(n^2)$ .

Al analizar la eficiencia empírica del algoritmo, obtenemos la siguiente gráfica:



Al representar superpuestas la función de la eficiencia teórica y la empírica, obtenemos lo siguiente:



Podemos observar que la gráfica de “*tiempos\_ordenacion.dat*” es una constante en 0 debido a que comparada con la gráfica de la eficiencia teórica, los valores son tan pequeños que parece que no cambia. Solucionaremos esto en el ejercicio 2.

## Ejercicio 2: Ajuste en la ordenación de la burbuja

A continuación ajustaremos la eficiencia teórica usando la función

$$f(x) = ax^2 + bx + c$$

Para ello introducimos los siguientes comandos en la terminal

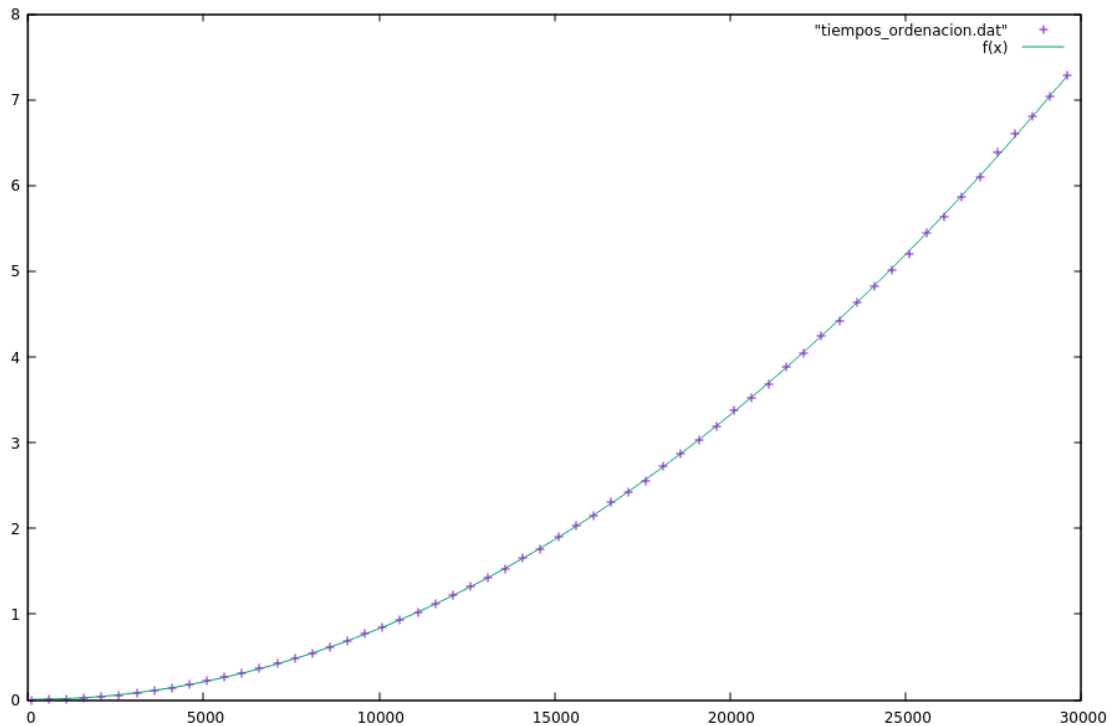
$$f(x) = a*(x*x) + b*x + c$$

fit f(x) “*tiempos\_ordenacion.dat*” via a, b, c

y como resultado obtendremos los siguiente valores para a, b, c:

Final set of parameters		Asymptotic Standard Error	
=====		=====	
a	= 8.34235e-09	+/- 2.379e-11	(0.2852%)
b	= -8.90801e-07	+/- 7.302e-07	(81.97%)
c	= 0.00560431	+/- 0.004692	(83.72%)

Una vez ajustada, volvemos a pintar la gráfica y observamos que ambas se superponen



### Ejercicio 3: Problemas de precisión

a) El algoritmo del fichero “*ejercicio\_desc.cpp*” se trata del algoritmo de búsqueda binaria, el cual realiza la búsqueda de un elemento dentro de un vector ordenado.

Para ello, se pasan como parámetros el inicio (inf) y el final (sup) de dicho vector. La función devuelve la posición donde se ha encontrado el elemento, o -1 si no estaba en el vector. Para el correcto funcionamiento del programa el vector debe estar ordenado.

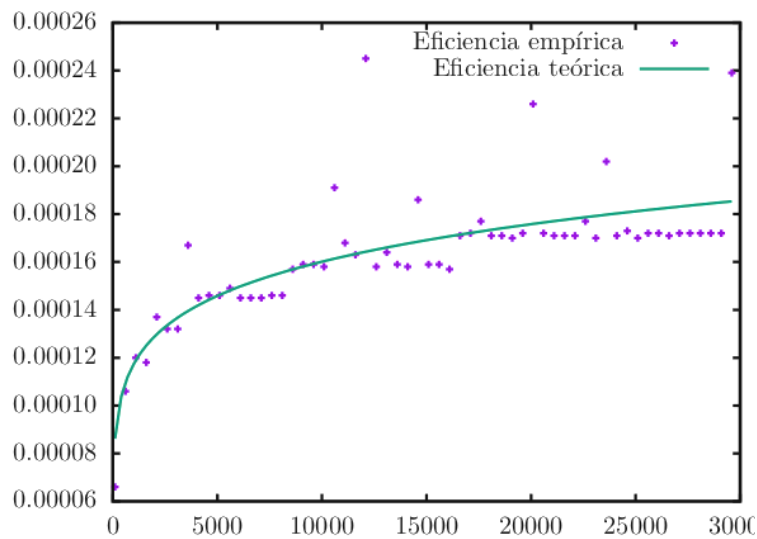
En resumen, el procedimiento se basa en dividir el vector original por la mitad, y si no está ahí el elemento buscado, nos quedamos únicamente con el sub-vector donde se puede encontrar dicho elemento (recordemos que el vector está ordenado). Repitiendo el proceso, acabaremos encontrando el elemento, o descubriendo que no está en el vector, cuando ya no se puedan hacer más divisiones.

b) Para el cálculo de la eficiencia teórica de la búsqueda binaria, nos basamos en el hecho de que se divide sucesivamente en dos un vector de tamaño  $n$ . En el caso peor, el proceso continuará hasta que no se puedan hacer más divisiones del vector.

Como el resto de operaciones son elementales ( $O(1)$ ), concluimos que la eficiencia del algoritmo es logarítmica, es decir,  $T(n) \in O(\log(n))$ .

c) El algoritmo es tan rápido que es imposible apreciar cambios en el tiempo de ejecución para vectores de tamaño mayor, lo que nos lleva a obtener una gráfica prácticamente horizontal. Por tanto, para analizar empíricamente la eficiencia de este algoritmo debemos ejecutarlo más de una vez por cada tamaño del vector.

En nuestro caso lo hemos ejecutado 1000 veces por cada tamaño, obteniendo la siguiente gráfica:



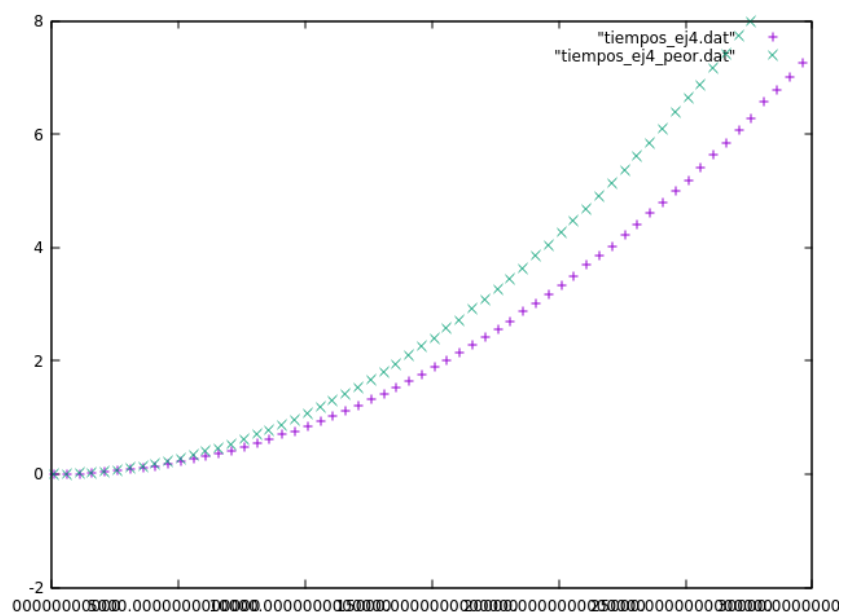
#### Ejercicio 4: Dependencia de la implementación

Eficiencia teórica se corresponde a la calculada en el ejercicio 1.

#### Peor Caso

```
// Generación del vector |
int *v=new int[tam];      // Reserva de memoria
v[0] = vmax;
for (int i=0; i<tam; i++)  // Recorrer vector
    v[i] = v[i-1] - 1;
```

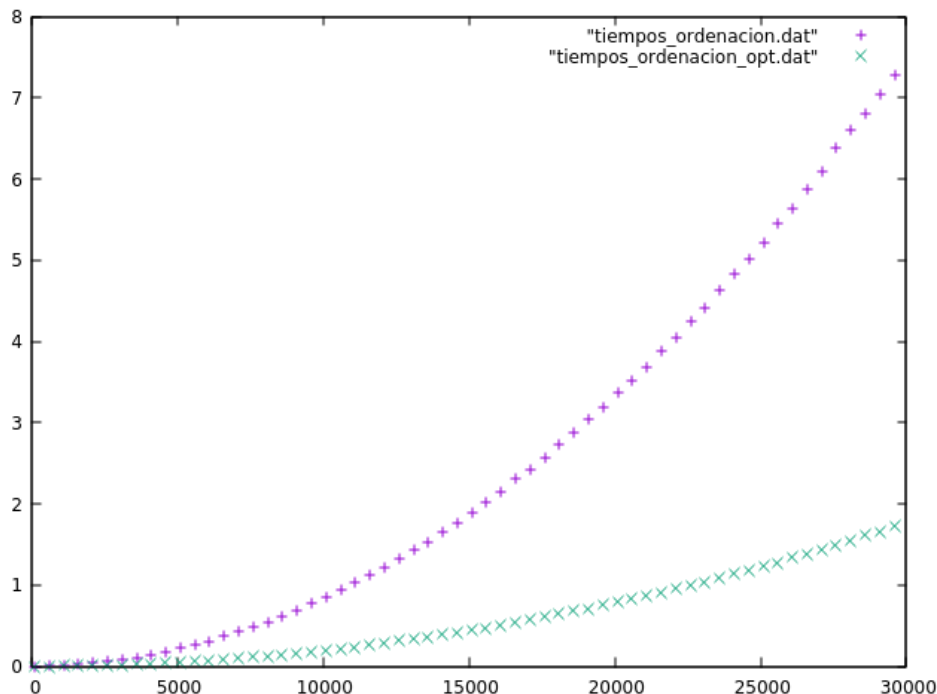
#### Comparación







Como vemos en la siguiente gráfica, el tiempo de ordenación en el programa compilado con la línea proporcionada, es menor que el tiempo de ordenación con el programa compilado de forma convencional.



### Ejercicio 7: Multiplicación matricial

El programa creado tiene el siguiente algoritmo para el cálculo de la multiplicación de matrices:

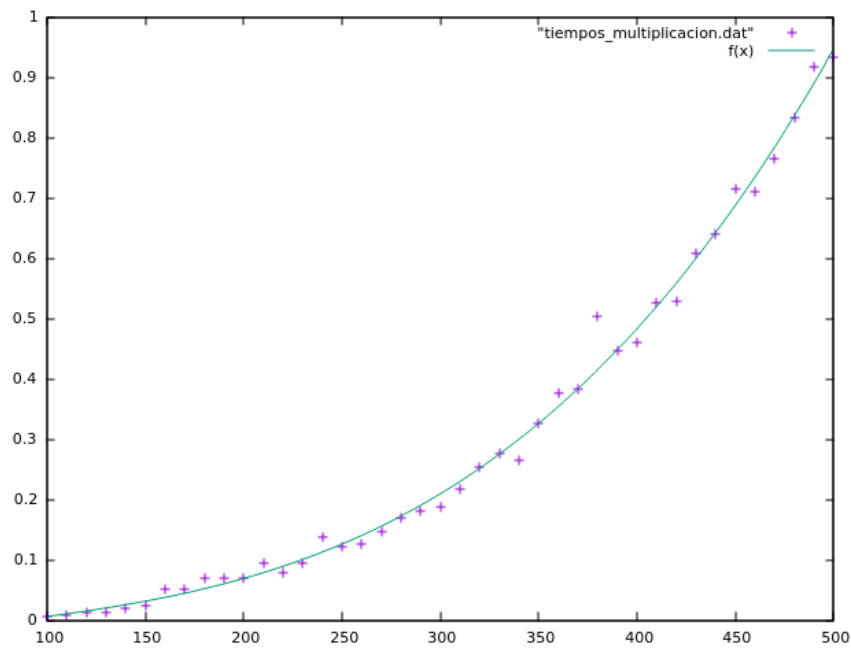
```
void multiplicacion(int **v_m, int **v1, int **v2, int n) {
    for (int i=0; i<n; i++){
        for (int j=0; j<n; j++){
            for (int k=0; k<n; k++){
                v_m[i][j] += v1[i][k]*v2[k][j];
            }
        }
    }
}
```

La eficiencia teórica de este algoritmo, teniendo en cuenta el número de multiplicaciones realizadas en el peor de los casos, es de orden  **$O(n) = n^3$** . Ahora vamos a pasar a comprobar el algoritmo de forma empírica.

Lo primero que haremos va a ser modificar el script porque si no el tiempo que tarda es muy grande (en mi caso voy a poner los tamaños de la matriz 100, 110, 120, ... , 500). Una vez modificado lo ejecutamos y obtenemos los tiempos. Ahora realizaremos la regresión para ajustar la curva teórica a la empírica utilizando la fórmula  $f(x) = a*x^{**3} + b*x^{**2} + c*x + d$ . El resultado es:

```
a = 9.34251e-09
b = -1.73615e-06
c = 0.000497637
d = -0.0358442
```

La gráfica resultante para estos valores sería:

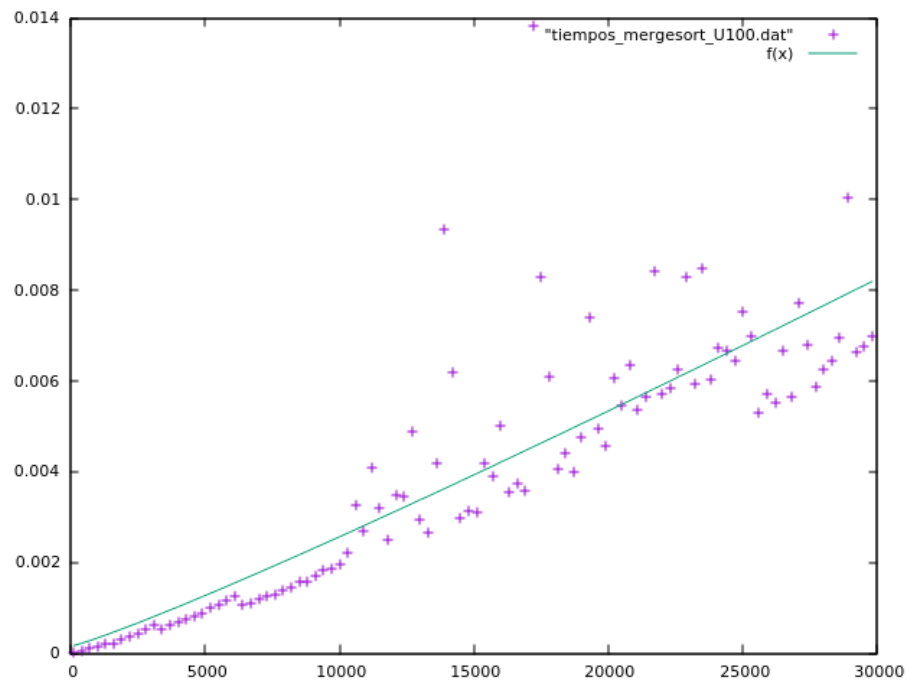


### Ejercicio 8: Ordenación por Mezcla

Una vez realizado el análisis de la eficiencia empírica y el ajuste obtenemos los siguientes resultados para la función  $f(x) = a \cdot (n \cdot \log(n)) + b$ :

a 2.61826e-08  
b 0.000154399

La gráfica con el ajuste de regresión es:



Para el estudio de cómo afecta el parámetro UMBRAL\_MS a la eficiencia del algoritmo probamos con distintos valores.

En mi caso lo he realizado con los valores 50, 100 y 200. Gracias a estos datos podemos ver que la eficiencia del algoritmo va disminuyendo a medida que aumentamos el valor del UMBRAL\_MS, es decir, que el tiempo que tarda en ejecutarse es mayor.

Esto es debido al “filtro” que hace el programa de la siguiente forma:

```
if (final - inicial < UMBRAL_MS){  
    insercion_lims(T, inicial, final);  
}
```

A continuación una gráfica con la eficiencia empírica del algoritmo con estos valores:

