MEMORIA

PRÁCTICA 1

ANÁLISIS DE LA EFICIENCIA DE ALGORITMOS

ASIGNATURA: Algorítmica

CURSO: 2023/2024

TITULACIÓN: Doble grado de ingeniería informática y matemáticas

GRUPO: J-Error404NotFound

INTEGRANTES:

- Laura Mandow Fuentes e.lauramandow@go.ugr.es
- Roberto González Lugo e.roberlks222@go.ugr.es
- Daniel Hidalgo Chica e.danielhc@go.ugr.es
- Chengcheng Liu e.cliu04@go.ugr.es
- Elías Monge Sánchez e.eliasmonge234@go.ugr.es

Índice

1. PARTICIPACIÓN

Participación general:

Elías: 100%
Laura: 100%
Roberto: 100%
Chengcheng (Olga): 100%
Daniel: 100%

Participación específica:

Aunque hayamos trabajado cada uno de forma global los contenidos de la práctica, a la hora de la redacción de la memoria, el trabajo se ha visto dividido en partes de carga de trabajo similar con el fin de aumentar la productividad.

En particular, para las ejecuciones de los algoritmos, salvo el caso concreto de comparación por hardware, se ha distribuido según las máquinas que se muestran en el apartado 4.

2. OBJETIVOS

El objetivo de esta práctica es analizar y comparar la eficiencia de los algoritmos de ordenación: inserción, quicksort, mergesort y bubble sort; además de los algoritmos de Floyd-Warshall, Hanoi y Fibonacci. Para este estudio se usará un enfoque híbrido: calcularemos la expresión teórica para la eficiencia de cada uno de los algoritmos y compararemos lo obtenido con datos calculados para los tiempos de ejecución de programas que implementan cada uno de ellos.

Este estudio se completará con:

- Gráficas para representar los tiempos obtenidos para la ejecución de cada programa en función de diferentes parámetros como el tamaño de la entrada y el tipo de dato que se procese
- Ajustes por mínimos cuadrados para precisar las fórmulas obtenidas en el cálculo teórico de la eficiencia
- Tablas y gráficas comparativas donde compararemos la actuación real de algoritmos con teóricamente iguales órdenes de eficiencia.
- Comparaciones de tiempos de ejecución de algunos algoritmos donde variaremos el entorno en el que se producen las pruebas: sistema operativo, compilador, máquina, etc.

Concluiremos esta memoria extrayendo conclusiones de los datos obtenidos en contraste con lo esperado teóricamente, justificando y corroborando diferentes conceptos estudiados en Teoría en la asignatura.

3. DISEÑO DEL ESTUDIO

Los tiempos de ejecución de los diferentes algoritmos propuestos han sido medidos en ordenadores distintos. Indicamos exactamente bajo qué entorno se ha estudiado la eficiencia de cada algoritmo; en particular, precisamos: el compilador de los programas que implementan los algoritmos, la máquina empleada especificando el procesador, el sistema operativo y / o la máquina virtual sobre la que se ejecuten los programas y el tamaño de entrada para los programas.

Algoritmos de ordenación Burbuja e Inserción

- Máquina: Acer Aspire A315-42
- Procesador: AMD Ryzen 5 3500U with Radeon Vega Mobile Gfx 2.10 GHz
- Sistema Operativo: Ubuntu 22.04 64bits
- Máquina virtual: Oracle VM VirtualBox, 2 cores
- Orden de compilación: g++ -std=gnu++0x -O2 algorithm.cpp -o algorithm
- Tamaño de entrada: [5.000, 125.000] con saltos de 5.000

Algoritmos de ordenación Mergesort y Quicksort

- Máquina: Asus TUF fx505dt
- Procesador: AMD Ryzen 7 3750h with Radeon Vega Mobile Gfx 2.3GHz
- Tarjeta gráfica: Nvidia Geforce GTX 1650
- Sistema Operativo: Ubuntu 22.04 64bits
- Orden de compilación: g++ algorithm.cpp -o algorithm
- Tamaño de entrada: [50.000, 1.250.000] con saltos de 50.000

Algoritmo para resolver el problema de Hanoi

- Máquina: Acer Aspire A515-45
- Procesador: AMD Ryzen 5 5500U with Radeon Graphics
- Sistema Operativo: Ubuntu 22.04 64bits
- Orden de compilación: g++ algorithm.cpp -o algorithm
- Tamaño de entrada: [7, 36] con saltos de 1

Algoritmo recursivo para calcular términos de la sucesión de Fibonacci

- Máquina: Surface Laptop 4
- Procesador: Intel Core i7
- Sistema Operativo: Ubuntu 22.04 64bits
- Orden de compilación: g++ algorithm.cpp -o algorithm
- Tamaño de entrada: [2-50] con saltos de 2

Algoritmo de Floyd-Warshall

- Máquina: HP Laptop 15s-eq1xxx
- Procesador: AMD Ryzen 5 4500U with Radeon Graphics
- Sistema Operativo: Ubuntu 22.04 64 bits
- Orden de compilación: g++ algorithm.cpp -o algorithm (cambiada en estudio comparativo para este algoritmo)
- Tamaño de entrada: [50,1250] con saltos de 50

4. ALGORITMOS CONSIDERADOS

4.1. Algoritmos cuadráticos: burbuja e inserción

Los primeros algoritmos que estudiaremos serán los de ordenación pudiendo ser o bien cuadráticos como el algoritmo burbuja y el algoritmo de inserción, o bien lineal logarítmicos como el algoritmo mergesort y el algoritmo quicksort.

El primero de ellos es el algoritmo bubble sort o algoritmo de ordenación burbuja. Se trata de un algoritmo de ordenación iterativo muy simple: consiste en ir comparando parejas de elementos del vector e intercambiarlas si están en orden equivocado. El código del algoritmo burbuja es el que sigue:

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

Le sigue el algoritmo insertion sort o algoritmo de inserción, otro algoritmo iterativo sencillo que consiste en insertar en cada iteración un valor elegido como clave en el lugar correspondiente tras compararlo con el resto. El código del algoritmo de inserción es el que sigue:

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

Análisis teórico

El tamaño de nuestro problema n será el número de datos del vector, es decir $n = final - inicial + 1$.

Burbuja

El tiempo de ejecución de las líneas 3 y 4 puede ser acotado por una constante a, por tanto será $O(1)$. En cuanto al bucle externo, este dependerá del número de veces que se ejecute el bucle interno, que a su vez dependerá del número de veces que se ejecute el cuerpo. El tiempo de ejecución del cuerpo, a su vez, se puede acotar por una constante b, y el número de veces que se ejecuta en el caso peor en función de n se calcula como:

$$nº\ de\ iteraciones = \sum_{i=inicial}^{final-2} \sum_{j=i+1}^{final-1} 1$$

Simplificando como $inicial = 1$ y $final = n$, nos queda:

$$nº\ de\ iteraciones = \sum_{i=1}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=1}^{n-2} (n - i - 1) = \sum_{i=1}^{n-2} n - \sum_{i=1}^{n-2} i - \sum_{i=1}^{n-2} 1 =$$

$$= n(n-2) - \frac{(n-1)(n-2)}{2} - (n-2) = \frac{n^2-3n+2}{2}$$

Por tanto podemos decir que el tiempo de ejecución de las líneas 5-12 está acotado en función del tamaño n del problema por $\frac{b}{2}(n^2 - 3n + 2)$, por lo que dicho código es de eficiencia $O(n^2)$ y aplicando la regla de la suma concluimos que la función es de eficiencia $O(n^2)$ o cuadrática.

Inserción

Por un razonamiento análogo al anterior, podemos calcular la eficiencia teórica calculando el número de veces que se ejecuta el cuerpo del bucle interno en función del tamaño del problema n.

El bucle externo se ejecuta desde $i = inicial + 1$ hasta $i = final - 1$, y el bucle interno se ejecuta desde $j = i$ hasta $j = inicial + 1$. Esto lo podemos escribir matemáticamente como:

$$nº\ de\ iteraciones = \sum_{i=inicial+1}^{final-1} \sum_{j=inicial+1}^{i} 1$$

Simplificando como $inicial = 0$, y $final = n - 1$, para que se mantenga el tamaño del problema, quedaría como:

$$nº\ de\ iteraciones = \sum_{i=1}^{n-2} \sum_{j=1}^{i} 1 = \sum_{i=1}^{n-2} i = \frac{(n-1)(n-2)}{2} = \frac{n^2}{2} - n + 1$$

Por tanto, como el cuerpo del bucle interno es $O(1)$ concluimos que la función es del orden de eficiencia cuadrático o $O(n^2)$ al igual que el algoritmo de la burbuja.

Análisis empírico

Ya tendríamos por tanto el análisis teórico de ambos algoritmos. Pasemos a continuación con el análisis de la eficiencia empírica.

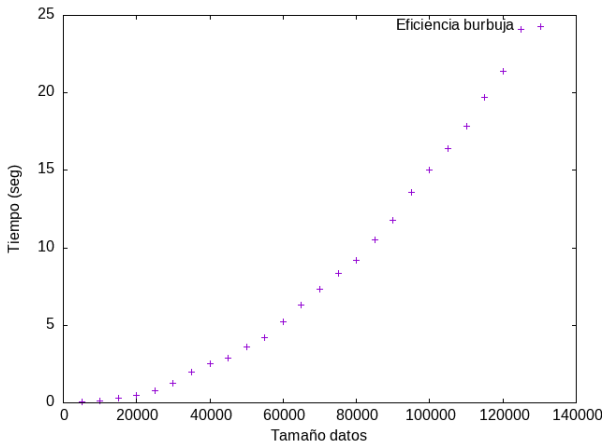
En dicho análisis, estudiamos los tiempos que toma ordenar por burbuja diferentes números de elementos de vectores de diferentes tipos. Posteriormente, compararemos ambos algoritmos, ya que ambos son de eficiencia cuadrática, y así observaremos las diferencias entre ellos.

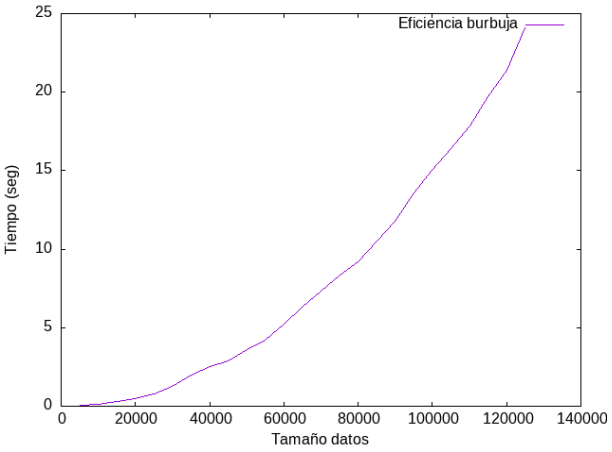
En concreto, tomaremos vectores de int, float, double y string, de tamaños de 5000 a 125000 con saltos de 5000, salvo la ordenación de string, para la cual hemos tomado

el libro del Quijote, que como tiene 202.308 palabras empezamos ordenando las 12.308 primeras hasta el total con saltos de 10.000 palabras. Los tiempos los observamos en las siguientes gráficas:

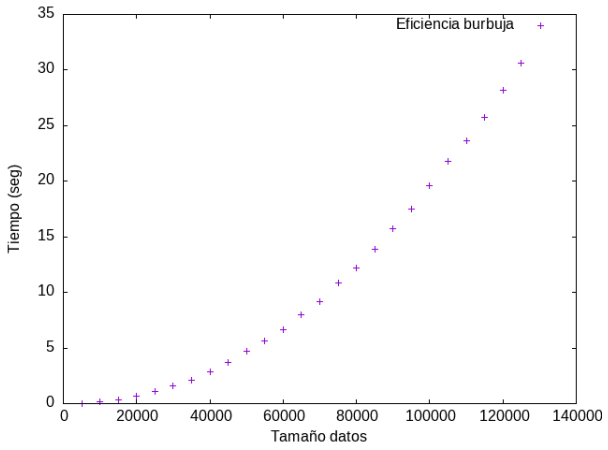
Burbuja

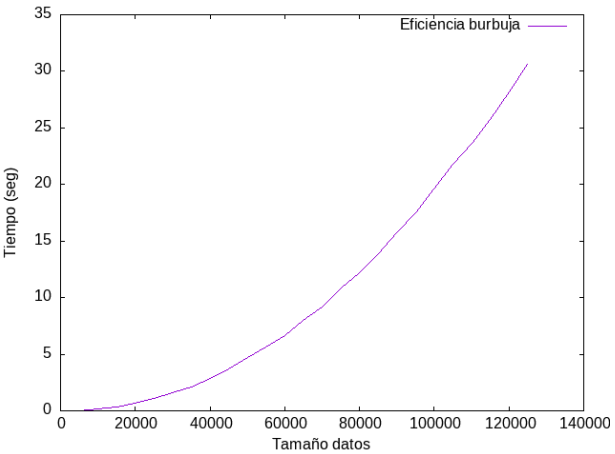
- Ordenación de int:



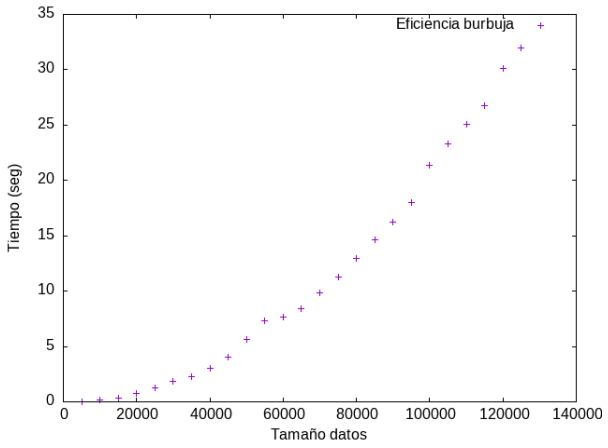


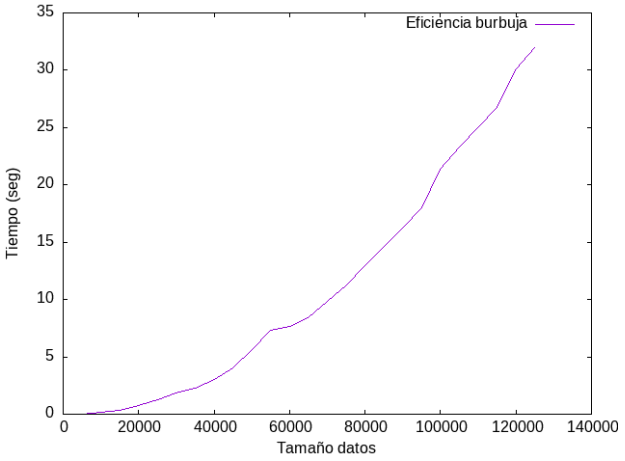
- Ordenación de float:



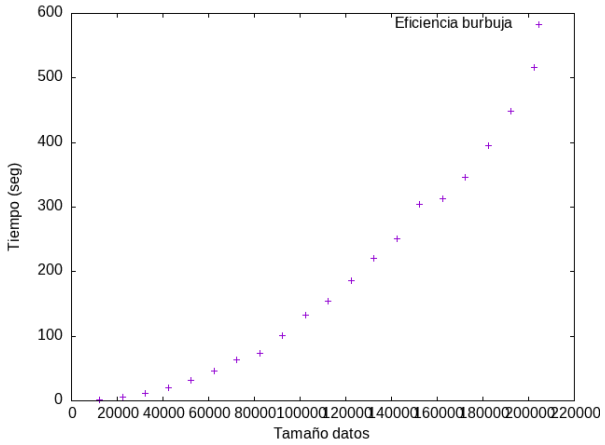


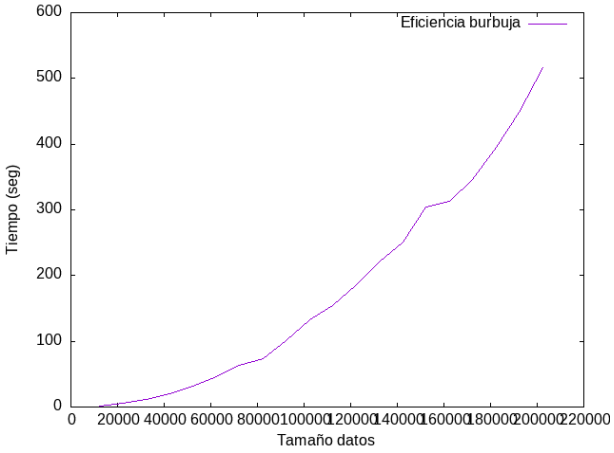
- Ordenación de double:



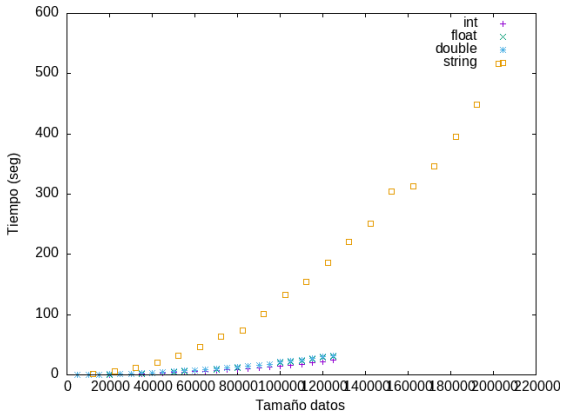


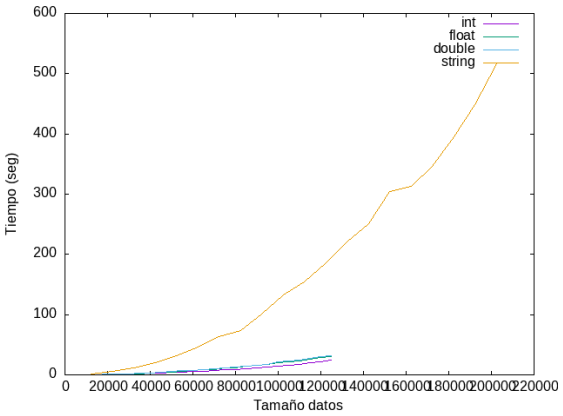
- Ordenación de string:





- Comparación de eficiencia para distintos tipos de datos:

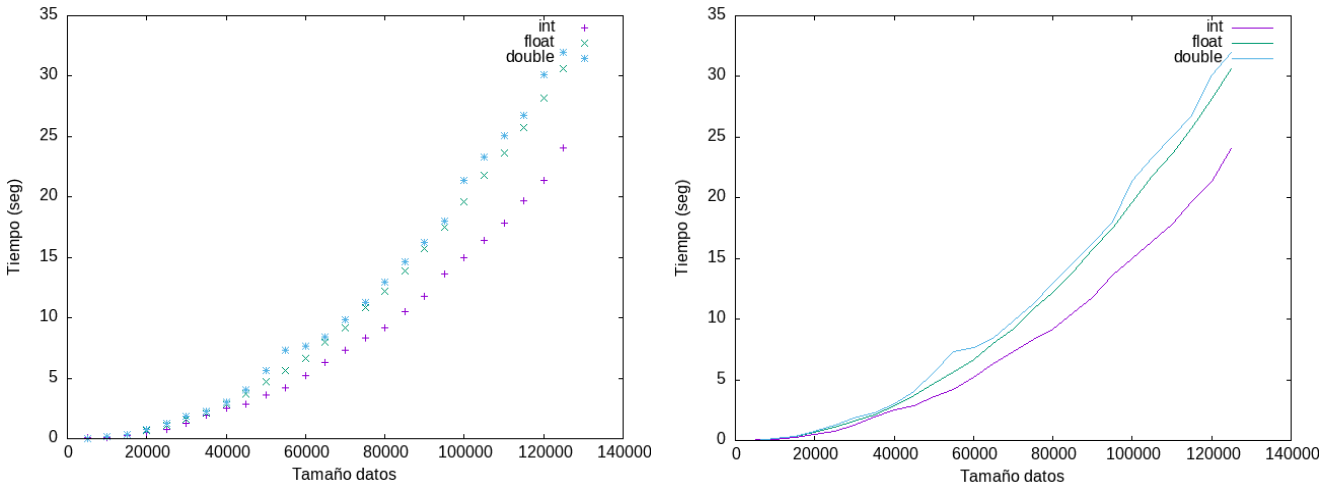




Como se observa los datos de tipo string tardan un tiempo significativamente mayor en ordenarse, al tratarse de tipos de datos más complejos que trabajan con memoria dinámica al reservar y liberar memoria. Por tanto, merece la pena comparar los otros 3

tipos de datos a parte, ya que en la gráfica anterior apenas se observan diferencias entre ellos.

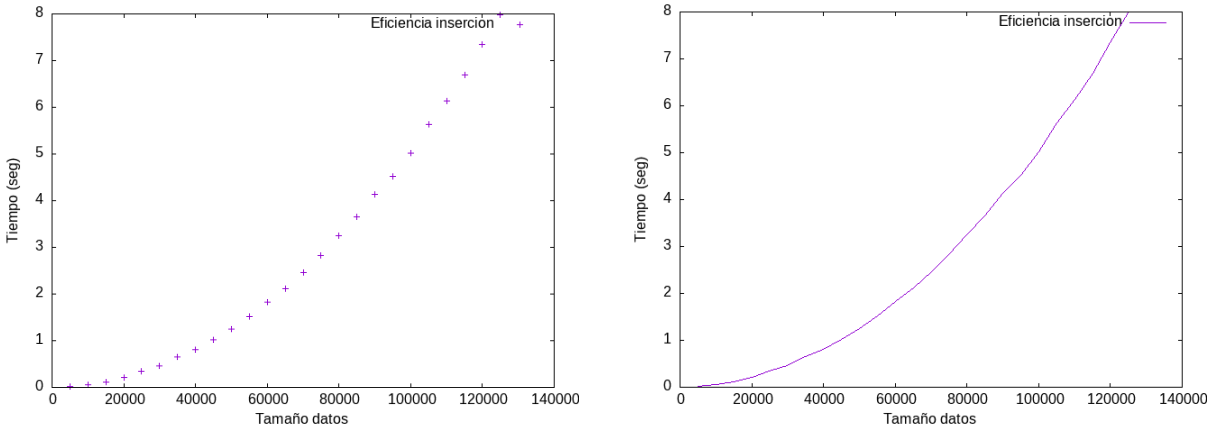
- Comparación de eficiencia para distintos tipos de datos (sin string):



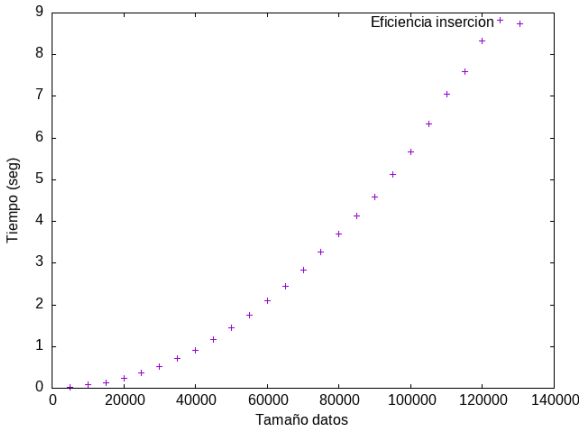
En las gráficas anteriores ya sí que se aprecian mucho mejor las diferencias entre la eficiencia de ordenación entre datos de tipo int, float y double, que no son muy significativas. Como mucho se puede destacar que los enteros tardan menos en ordenarse, al tratarse de un tipo de dato más sencillo. También se podría destacar que los datos de tipo double, a pesar de ocupar el doble de espacio que los float, no toman mucho más tiempo en ordenarse.

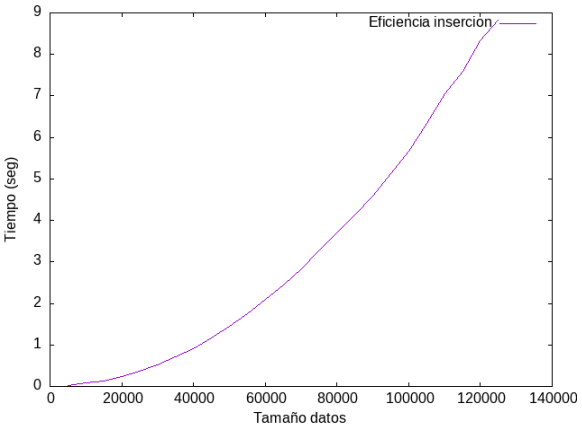
Inserción

- Ordenación de int:

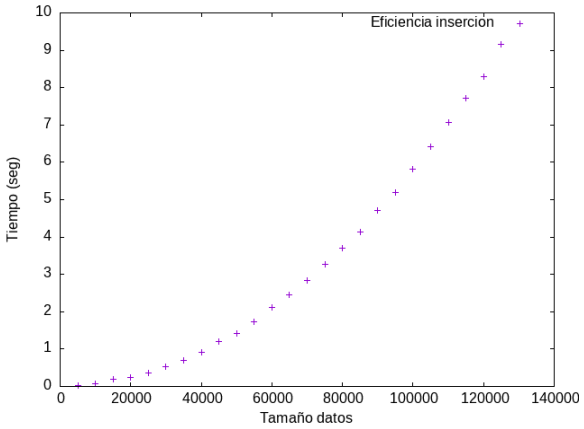


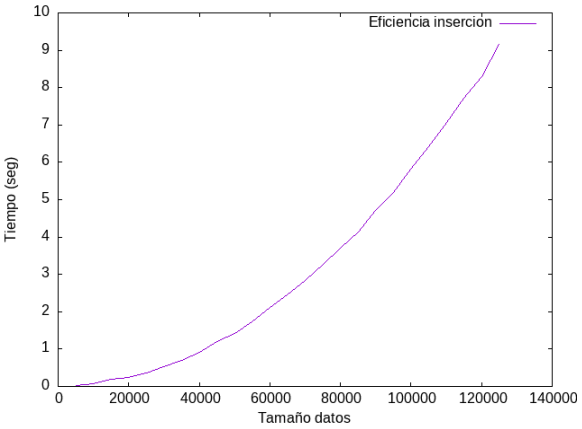
- Ordenación de float:



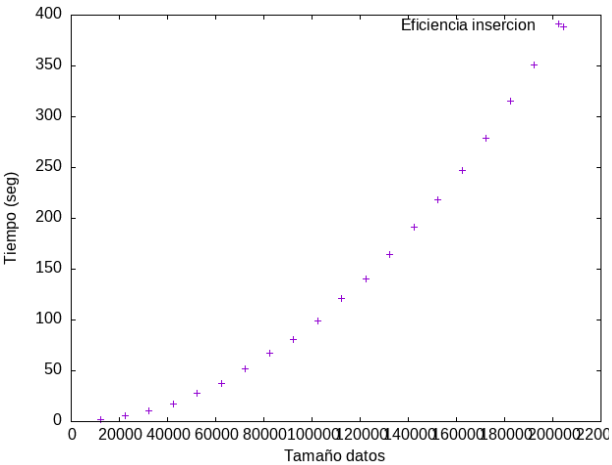


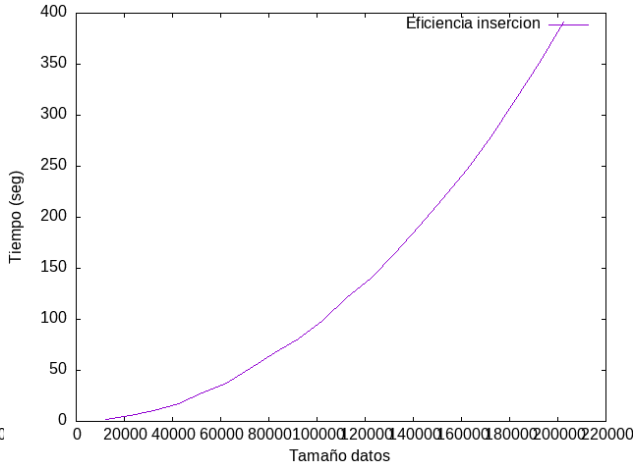
- Ordenación de double:



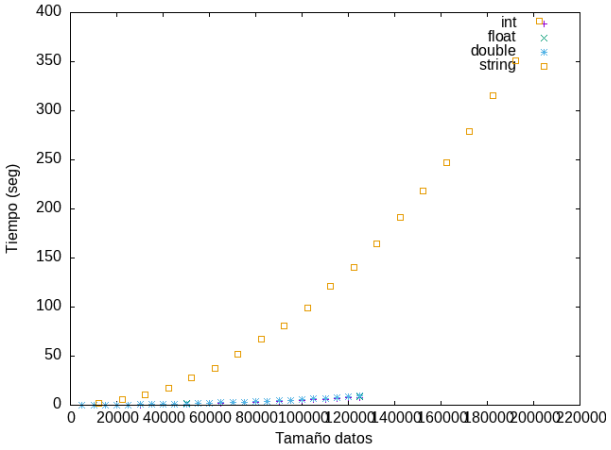


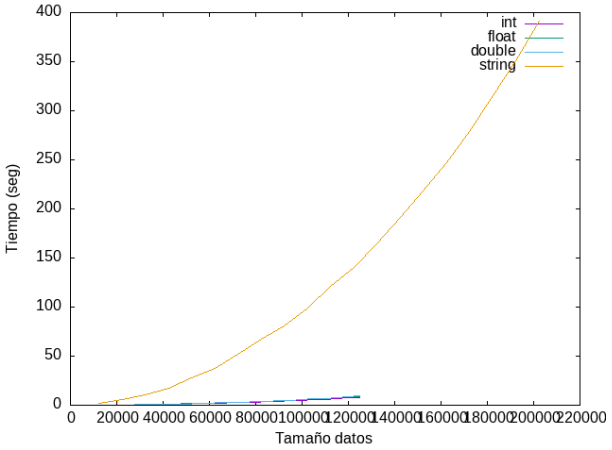
- Ordenación de string:





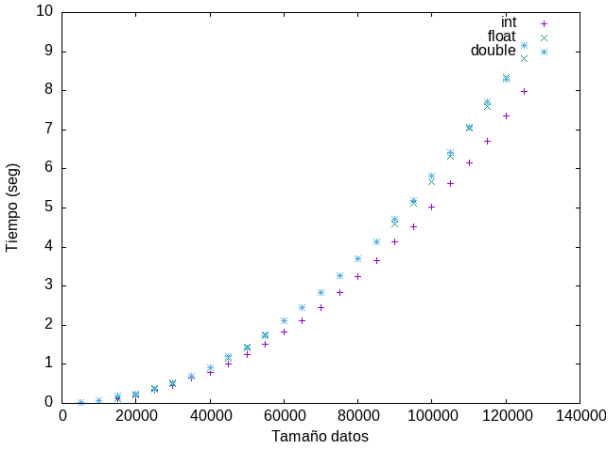
- Comparación de eficiencia para distintos tipos de datos:

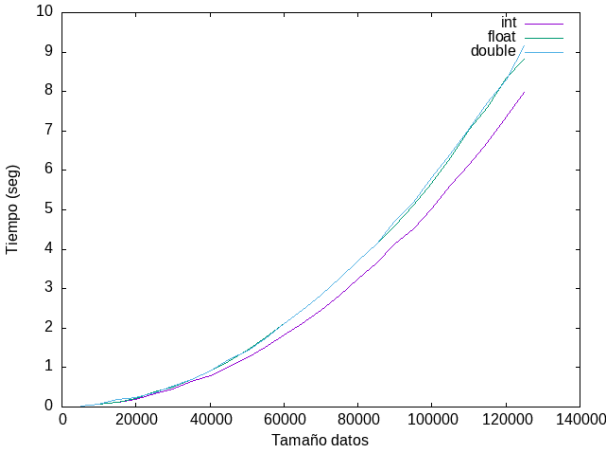




Al igual que ocurría con el algoritmo de burbuja, no se distinguen casi las diferencias entre los tiempos de ordenación de los datos int, float y double, puesto que la ordenación de string ocupa un tiempo considerablemente mayor. Es por esto que conviene graficarlos aparte:

- Comparación de eficiencia para distintos tipos de datos (sin string):





Análisis de la eficiencia híbrida

Como hemos observado en las gráficas anteriores, en especial en los gráficos de líneas, la forma de la gráfica se corresponde a la de una función cuadrática, cosa que encaja con el análisis teórico previamente realizado. Nuestro siguiente objetivo será averiguar la expresión concreta de la función cuadrática, es decir, realizar una regresión con la función

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

Aunque por simplificar también podemos escoger la función

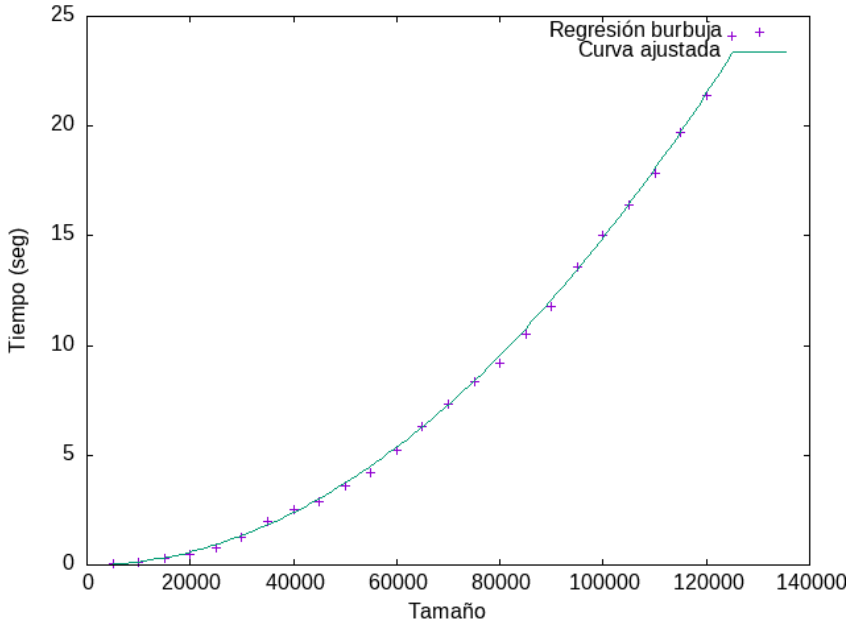
$$f(x) = ax^2$$

Procedemos por tanto a realizar la regresión con cada uno de los tipos de datos y con cada uno de los algoritmos. En las gráficas además viene etiquetada la función de regresión con la constante oculta.

Burbuja

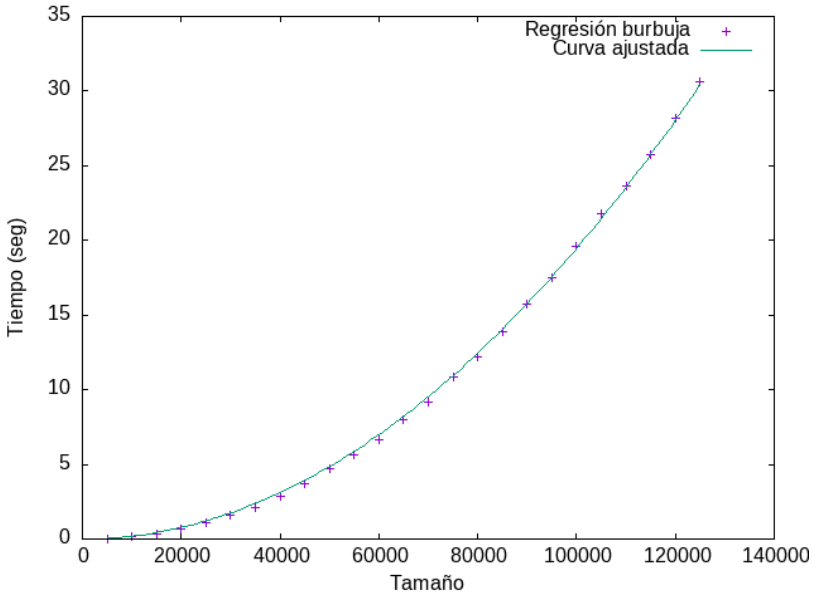
- Regresión de int:

$$f(x) = 1.49261 \cdot 10^{-9} \cdot x^2$$



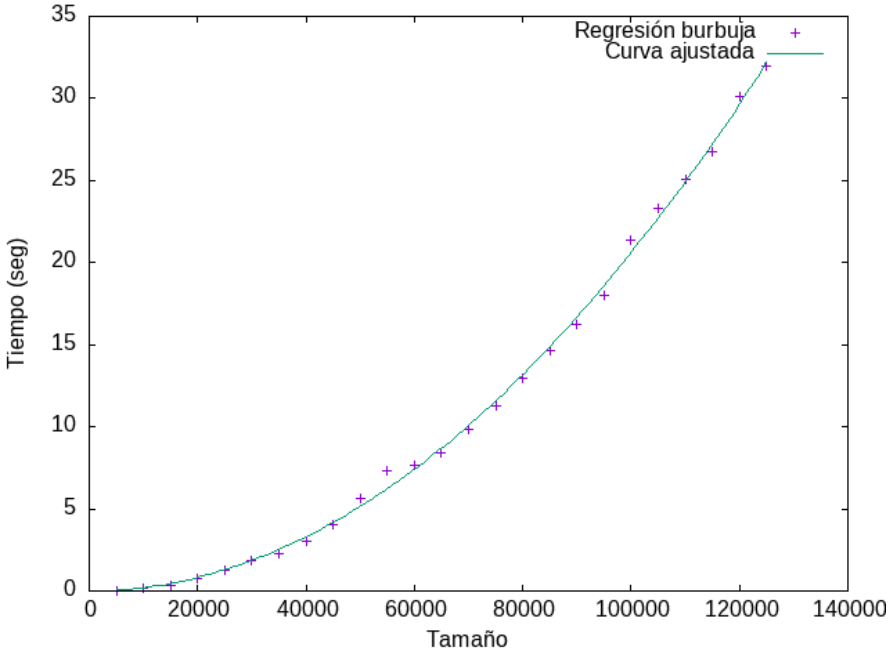
- Regresión de float:

$$f(x) = 1.94678 \cdot 10^{-9} \cdot x^2$$



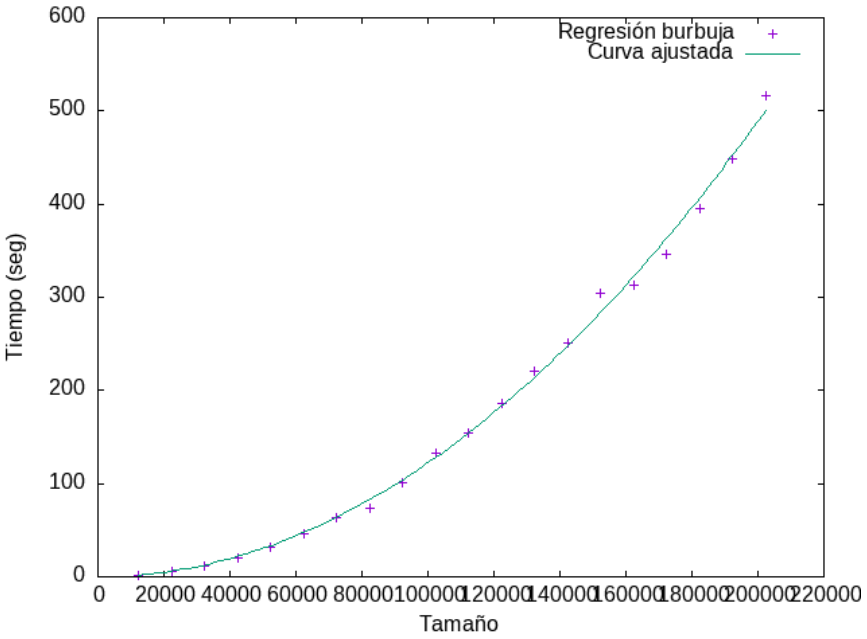
- Regresión de double:

$$f(x) = 2.05976 \cdot 10^{-9} \cdot x^2$$



- Regresión de string:

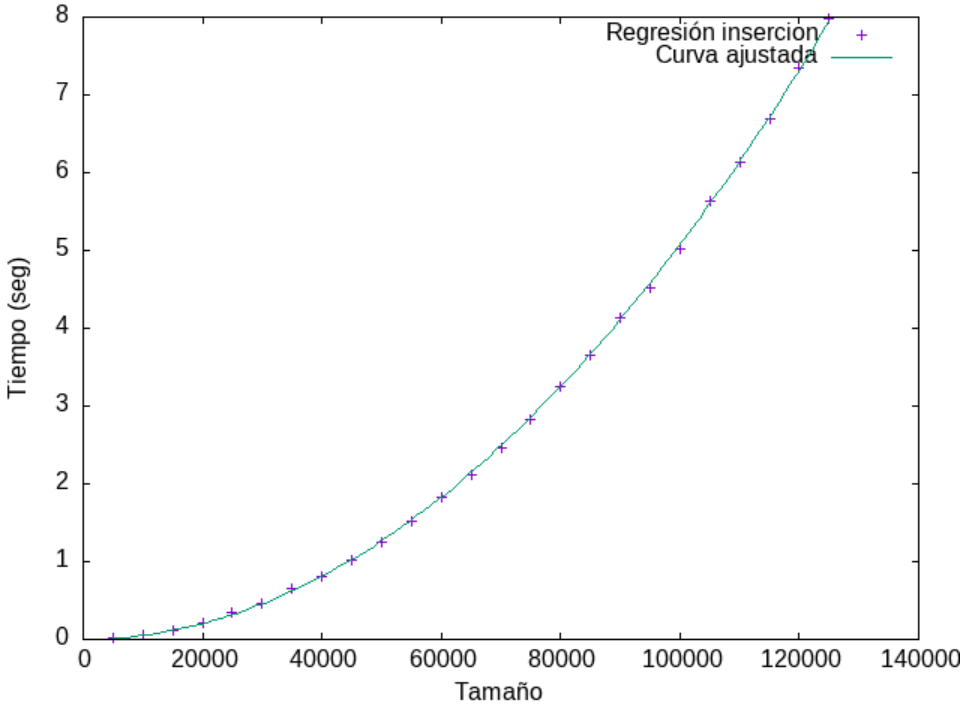
$$f(x) = 1.22298 \cdot 10^{-8} \cdot x^2$$



Inserción

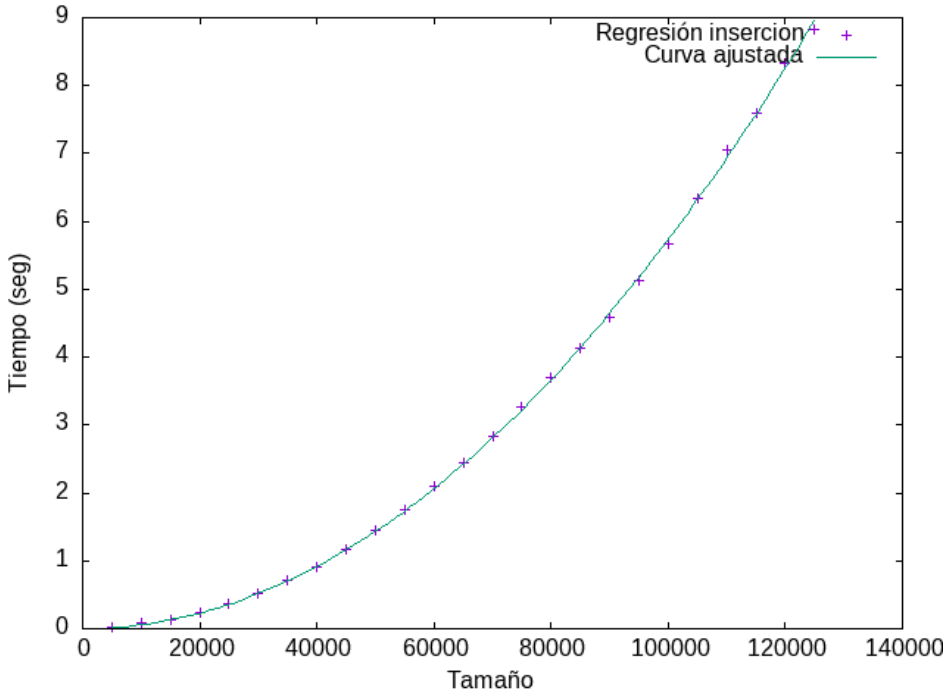
- Regresión de int:

$$f(x) = 5.07875 \cdot 10^{-10} \cdot x^2$$



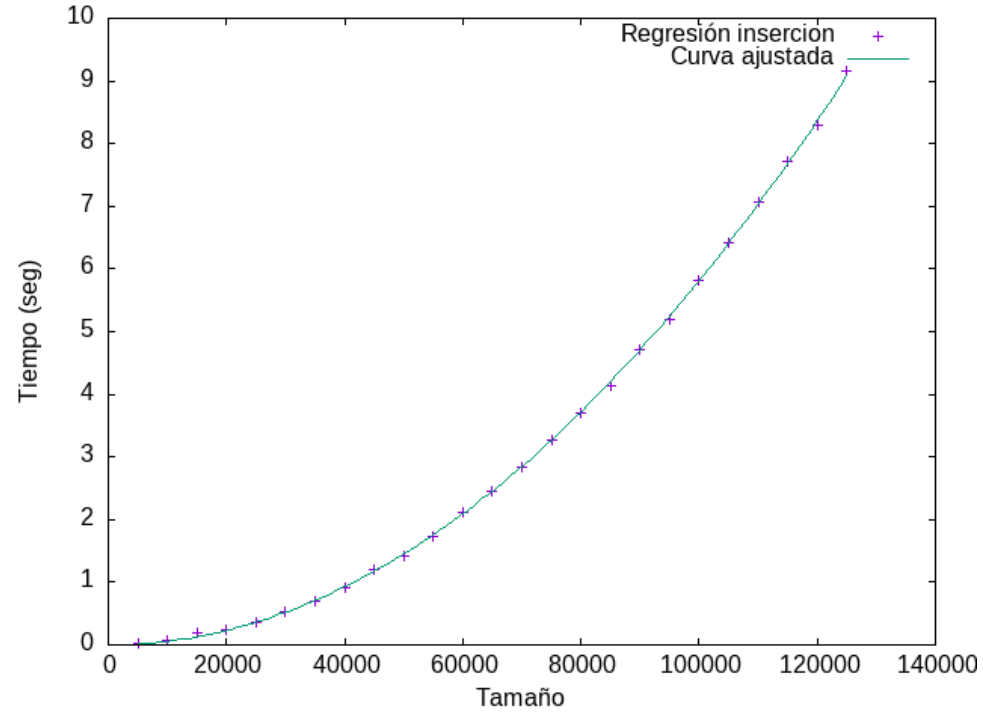
- Regresión de float:

$$f(x) = 5.73344 \cdot 10^{-10} \cdot x^2$$



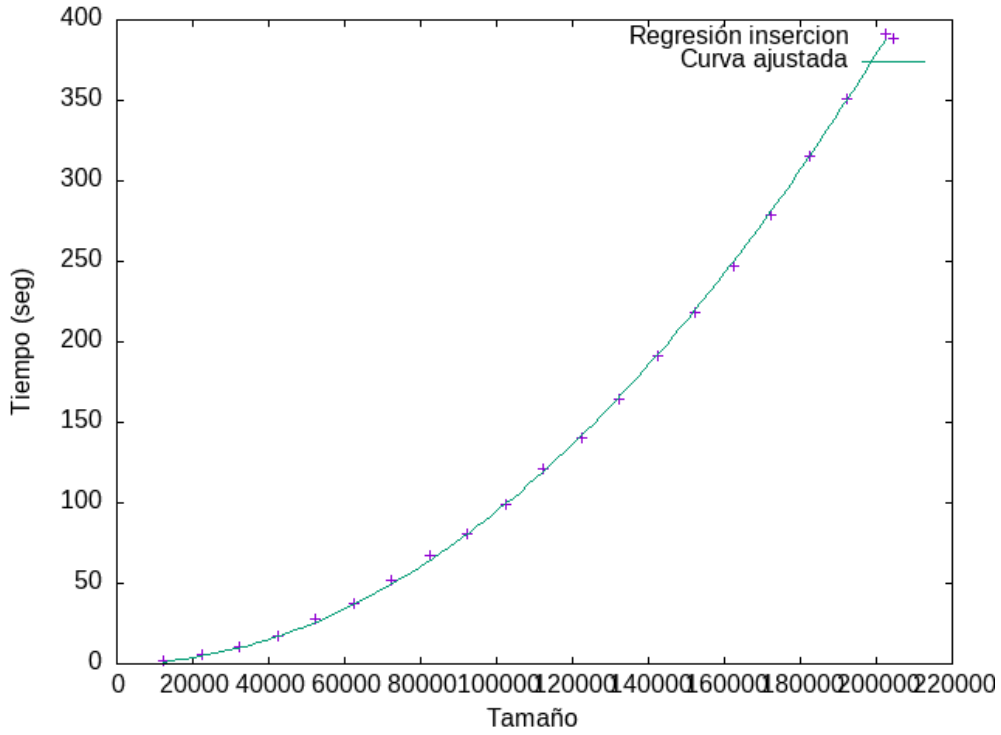
- Regresión de double:

$$f(x) = 5.81273 \cdot 10^{-10} \cdot x^2$$



- Regresión de string:

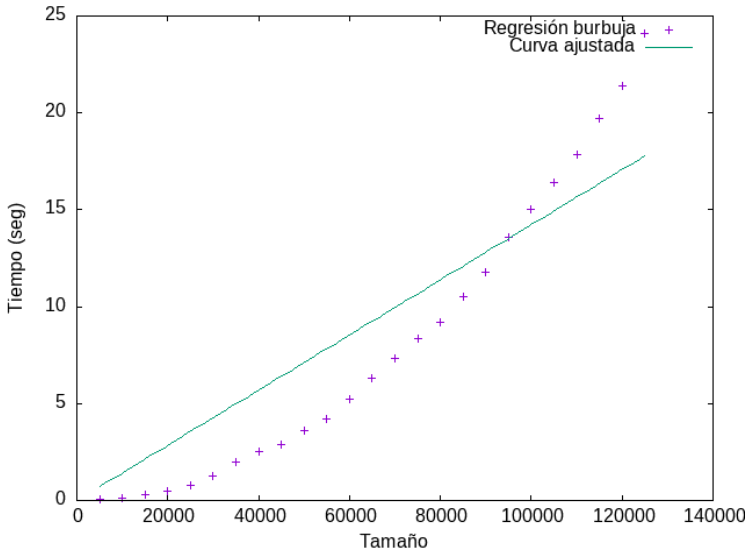
$$f(x) = 9.47818 \cdot 10^{-9} \cdot x^2$$



Se puede observar que todas las regresiones son buenas, pues las curvas pasan muy cerca de los puntos graficados por gnuplot, lo que reafirma nuestro resultado del análisis teórico.
Sin embargo, podríamos preguntarnos cómo quedaría una regresión con otro tipo de función no cuadrática, por ejemplo una lineal o una exponencial o una logarítmica. Probaremos estas tres regresiones para la gráfica de la ordenación de enteros.

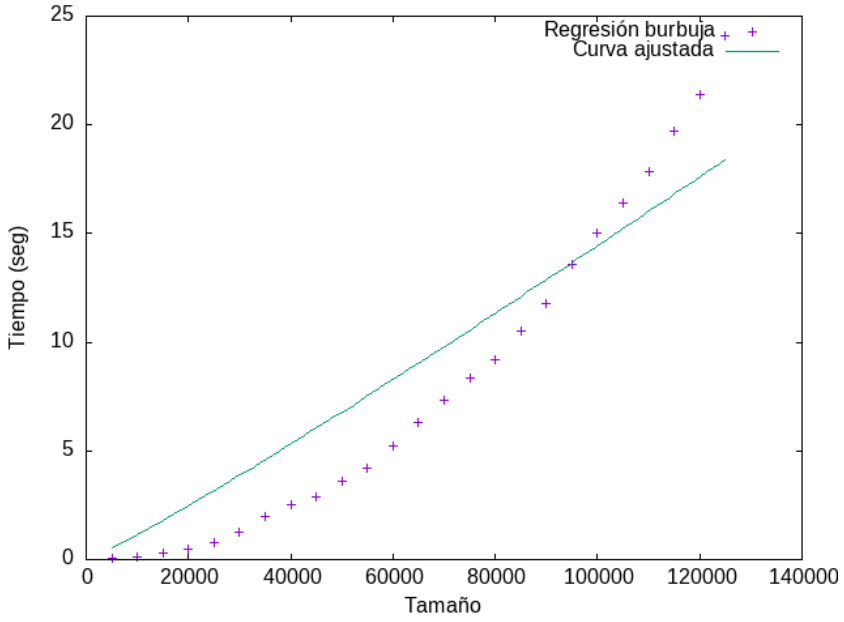
- Regresión lineal:

$$f(x) = 0.000142324 \cdot x$$



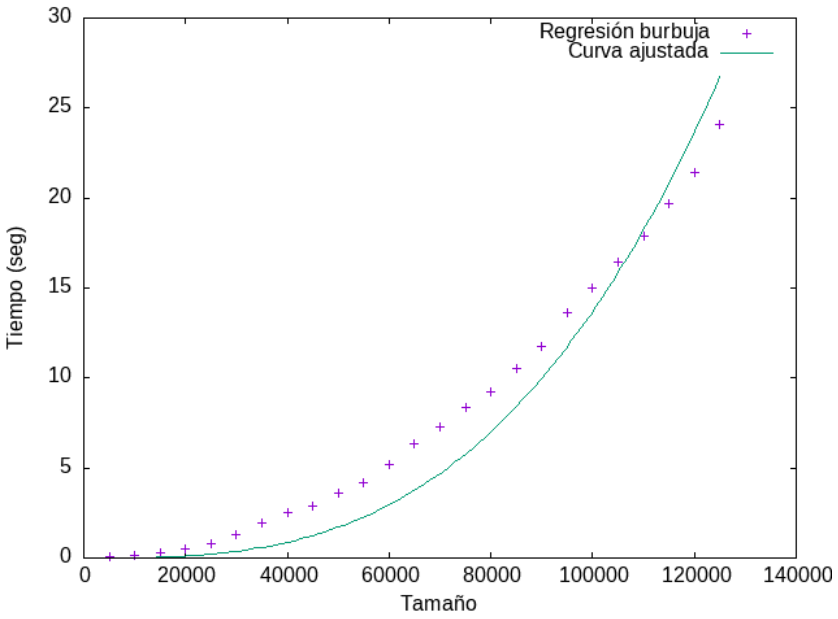
- Regresión lineal-logarítmica:

$$f(x) = 1.26425 \cdot 10^{-5} \cdot x \cdot log(0.914885 \cdot x)$$



- Regresión cúbica:

$$f(x) = 1.36951 \cdot 10^{-14} \cdot x^3$$



Observamos que los ajustes lineal y lineal-logarítmico son bastante imprecisos, con gráficas que pasan bastante más lejos de los puntos que la curva cuadrática, lo que indica que estos no son los ajustes correctos. Podríamos destacar el caso de la cúbica, que sí que mantiene una cierta correlación con la gráfica, ya que al fin y al cabo, sigue siendo un ajuste polinómico.

4.2. Algoritmos lineal logarítmicos: mergesort y quicksort

Continuando con los algoritmos de ordenación, nos disponemos a estudiar dos ejemplos de algoritmos que son del orden $O(n \cdot \log(n))$. Aunque cabe resaltar que el quicksort en el caso peor, no lo es realmente, como veremos a continuación. Aún así generalmente se comporta como si tuviera este orden, y por tanto así se suele tratar.

En primer lugar nos centraremos en el algoritmo de ordenación por mezcla, conocido como mergesort. Se trata de un algoritmo considerablemente más complejo que los dos anteriores. Su procedimiento se basa en el principio de "divide y vencerás". Esto significa que el algoritmo divide el conjunto de elementos a ordenar en dos mitades, hasta que se llega a conjuntos que contienen un solo elemento o ninguno, en nuestro caso hasta conjuntos con tamaño inferior a UMBRAL_MS. A continuación, estos conjuntos se van combinando (o mezclando) de manera ordenada, formando conjuntos mayores, hasta reconstruir el conjunto inicial pero ya ordenado.

El código del algoritmo de mezcla es el siguiente:

```
static void mergesort_lims(TYPE T[], int inicial, int final)
{
    if (final - inicial < UMBRAL_MS)
    {
        insercion_lims(T, inicial, final);
    }
    else
    {

        int k = (final - inicial) / 2;

        TYPE *U = new TYPE[k - inicial + 1];
        assert(U);
        int l, l2;


        for (l = 0, l2 = inicial; l < (k-inicial); l++, l2++)
            U[l] = T[l2];



        TYPE *V = new TYPE[final - k + 1];
        assert(V);
        for (l = 0, l2 = k; l < final - k; l++, l2++)
            V[l] = T[l2];



        mergesort_lims(U, 0, k-inicial);
        mergesort_lims(V, 0, final - k);


        fusion(T, inicial, final, U, V);

        delete[] U;
        delete[] V;
    };
```

Además, el código de la función fusión (mezcla) es el siguiente:

```
static void fusion(TYPE T[], int inicial, int final, TYPE U[], TYPE V[])
{
    int j = 0;
    int k = 0;
    int tam = (final - inicial) / 2;
    for (int i = inicial; i < final; i++)
    {


        if ((k==tam)  || ((j<tam) && (U[j] < V[k])) )
        {
            T[i] = U[j];
            j++;
        }
        else
        {
            T[i] = V[k];
            k++;
        };
    };
```

Continuando, el algoritmo de ordenación rápida, o quicksort, es otro ejemplo clásico del enfoque "divide y vencerás" aplicado a los algoritmos de ordenación. A diferencia del mergesort, que divide el conjunto de datos en mitades cada vez más pequeñas de manera uniforme, el quicksort selecciona un elemento del conjunto de datos como pivote y luego organiza los demás elementos en dos subconjuntos: los que son menores que el pivote y los que son mayores. Este proceso se denomina particionado. Después del particionado, el algoritmo aplica recursivamente el mismo proceso a los dos subconjuntos hasta que el conjunto completo está ordenado.

El código del algoritmo quicksort es el siguiente:

```
static void quicksort_lims(TYPE 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 código de la función dividir_qs (que es la que se encarga de escoger dicho pivote) es el siguiente:

```
static void dividir_qs(TYPE T[], int inicial, int final, int &pp)
{
    TYPE pivote, aux;
    int k, l;

    pivote = T[inicial];
    k = inicial;
    l = final;
    do
    {
        k++;
    } while ((T[k] <= pivote) && (k < final - 1));
    do
    {
        l--;
    } while (T[l] > pivote);
    while (k < l)
    {
        aux = T[k]; // swap(k,l)
        T[k] = T[l];
        T[l] = aux;
        do
            k++;
        while (T[k] <= pivote);
        do
            l--;
        while (T[l] > pivote);
    };
    aux = T[inicial]; // swap(inicial,l)
    T[inicial] = T[l];
    T[l] = aux;
    pp = l;
};
```

Análisis teórico

El tamaño de nuestro problema n será el número de datos del vector, es decir $n = final - inicial + 1$.

Mergesort

En primer lugar, del código es fácil ver que la función fusión es de orden O(n), ya que consiste en sentencias elementales y un bucle for de orden O(n) que itera n veces y cuyo cuerpo está formado por una sentencia condicional de orden O(1) ; tanto la evaluación de la condición del if, como los cuerpos del if-else son de orden O(1), así aplicando las reglas de la suma y del máximo podemos concluir que dicha función es del orden O(n).

Ahora veamos la eficiencia de la función mergesort_lims. En el código proporcionado podemos ver que empieza por una sentencia condicional donde el bloque if es O(1), ya que llama a la función de inserción cuando el número de datos es menor que el umbral establecido, es decir, n < UMBRAL_MS, y por tanto, el tiempo de ejecución de esta parte es acotable por una constante.

En el bloque del else, a parte de sentencias simples, tenemos:

- Dos bucles for que copian cada uno medio vector original en un vector auxiliar. Para ello cada uno itera n/2 veces siendo n el tamaño del vector original. Así, los bucles son de orden O(n) (hay que tener en cuenta que el cuerpo de los bucles es O(1) por ser operaciones elementales).
- Dos llamadas recursivas al propio *mergesort_lims* donde el tamaño del vector con el que se llama es n/2.
- Llamada a la función *fusion* que es de orden O(n), como comentamos anteriormente.

Por tanto, aplicando otra vez las reglas de la suma y del máximo obtenemos la siguiente expresión:

$$T(n) = 2 * T(n/2) + n \quad si \ n > UMBRALMS$$
$$T(n) = 1 \quad si \ n <= UMBRALMS$$

Para resolver esta recurrencia, realizamos un cambio de variable $n=2^m$

$T(2^m) = 2 * T(2^{m-1}) + 2^m = 2 * (2 * T(2^{m-2}) + 2^{m-1}) + 2^m = 2^2 * T(2^{m-2}) + 2 * 2^m$

Desarrollando k veces, con k = m - $\log_2$ UMBRAL_MS (aproximando al entero superior):

$T(2^m) = 2^k * T(2^{m-k}) + k * 2^m =$

$= 2^{m-\log2\,(UMBRAL\_MS)} * T(2^{\log2\,(UMBRAL\_MS)}) + (m - \log_2 UMBRAL\_MS) * 2^m =$

$= (2^m / UMBRAL\_MS) * T(UMBRAL\_MS) + (m - \log_2 UMBRAL\_MS) * 2^m =$

$= (2^m / UMBRAL\_MS) + (m - \log_2 UMBRAL\_MS) * 2^m$

Deshaciendo el cambio com m = $\log_2$ n, finalmente obtenemos:

$$T(2^m) = T(n) = \frac{n}{UMBRAL\,MS} + n\log_2 n - n\log_2(UMBRALMS) \in O(n \log n)$$

Lo cual indica que la eficiencia de Mergesort es del orden O(n log(n)). (NOTA: UMBRAL_MS es una constante).

Quicksort

El análisis teórico del quicksort se realiza de la siguiente forma:
En primer lugar, vemos la función quicksort_lims, en la cual, si n es menor que un umbral dado (llamémosle γ), podemos considerarlo constante, de todas formas, como analizamos el caso peor (notación O grande), podemos asumir que entramos directamente al else: suponiendo entonces $n \geq \gamma$, entramos en el else, que lo primero que hace es llamar a la función dividir.
Analizando dicha función, en el caso peor (que es que el vector original esté ordenado, o esté en orden inverso), observamos que (obviando las instrucciones que son O(1)):

1. El primer 'do while' solo se ejecutaría 1 vez, y entonces $k_{dividir} = inicial + 1$.
2. El segundo 'do while' movería l hasta inicial, es decir, se ejecutaría n veces (O(n)).
3. Por último, el tercer do while no hace nada relevante.

En conclusión, esta función, la *primera* vez que se llama *(en el caso peor)* es O(n). Volviendo a *quicksort_lims*, la segunda instrucción del else llama a *quicksort_lims* desde *inicial* hasta $k_{quicksort'}$, osea, desde *inicial* hasta *inicial+1*, por tanto, sería un O(2) = O(1). La tercera instrucción sin embargo, lo llama desde *inicial +2* hasta final, osea, lo llamaría con *n-1* datos. Por tanto, tendríamos la ecuación de recurrencia:

$T(n) = n + T(n-1) \Rightarrow T(n) = n + (n-1) + T(n-2) = \cdots = \sum_{i=\gamma}^{n} i$

En el caso peor, podemos suponer $\gamma = 0$ (ya que este es el umbral constante del que hablábamos antes). Por tanto, nos queda $\sum_{i=0}^{n} i = 0 + 1 + 2 + \cdots + n$. Es decir, la suma de los n primeros números. Aplicando la fórmula de Gauss:

$= \frac{n(n+1)}{2} \Rightarrow O(\frac{n(n+1)}{2}) = O(\frac{n^2+n}{2}) = O(\frac{1}{2} \cdot n^2 + \frac{1}{2} \cdot n) = O(\frac{1}{2} \cdot n^2) = O(n^2)$

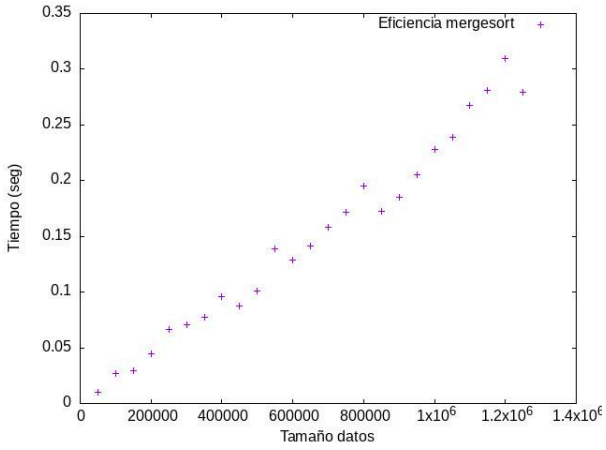
Análisis empírico

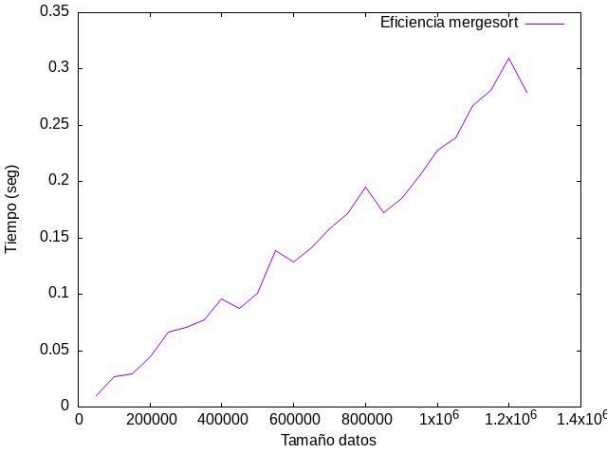
Ya tenemos pues el análisis teórico de la eficiencia de estos algoritmos. Con esta información es hora de pasar al análisis empírico, es decir, ver realmente a nivel pragmático como se comportan.

Al igual que con los algoritmos anteriores, probaremos con distintos tipos de datos. En concreto, tomaremos vectores de int, float, double y string, de tamaños de 50000 a 1250000 con saltos de 50000, salvo la ordenación de string, para la cual hemos tomado el libro del Quijote, que como tiene 202.308 palabras empezamos ordenando las 12.308 primeras hasta el total con saltos de 10.000 palabras. Los tiempos los observamos en las siguientes gráficas:

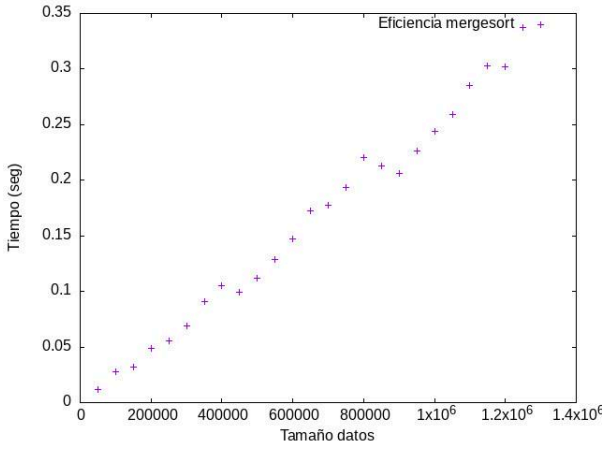
Mergesort

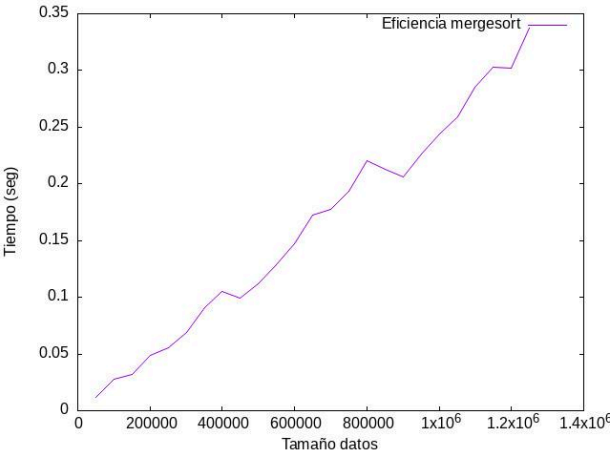
- Ordenación de int:



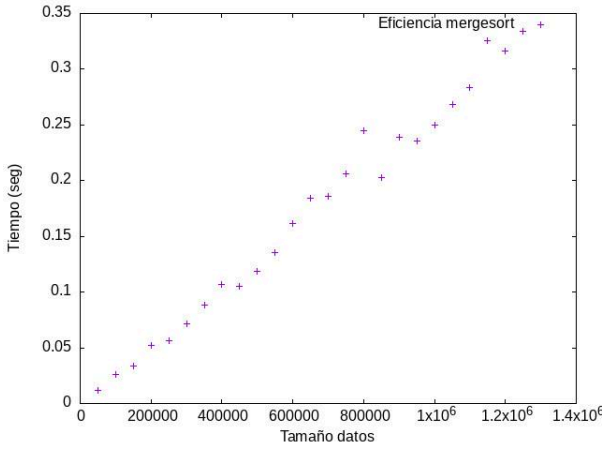


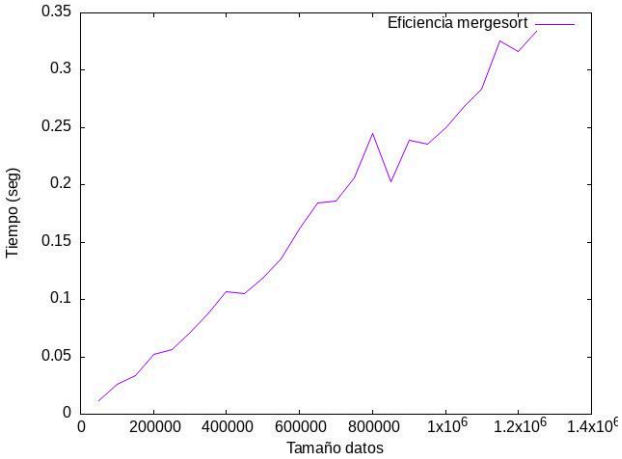
- Ordenación de float:



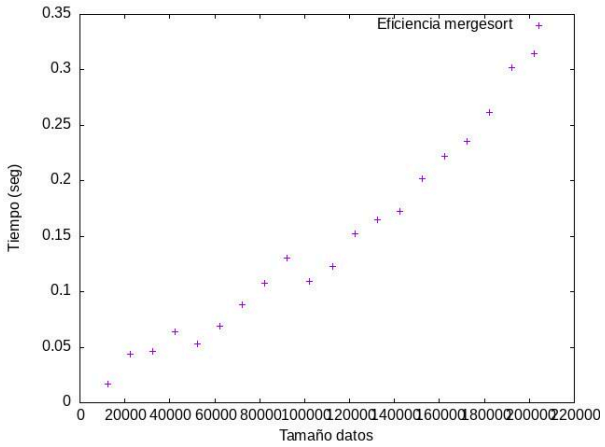


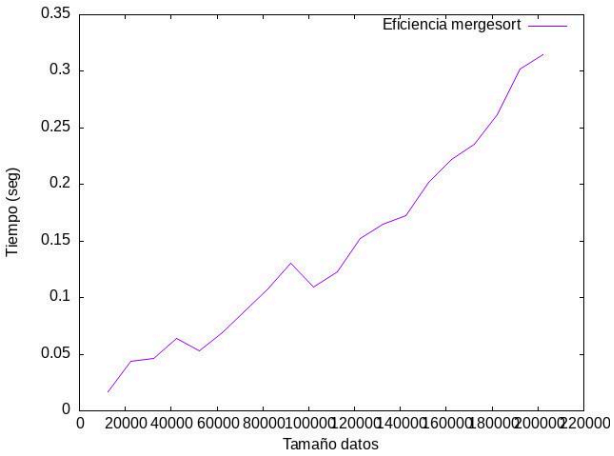
- Ordenación de double:



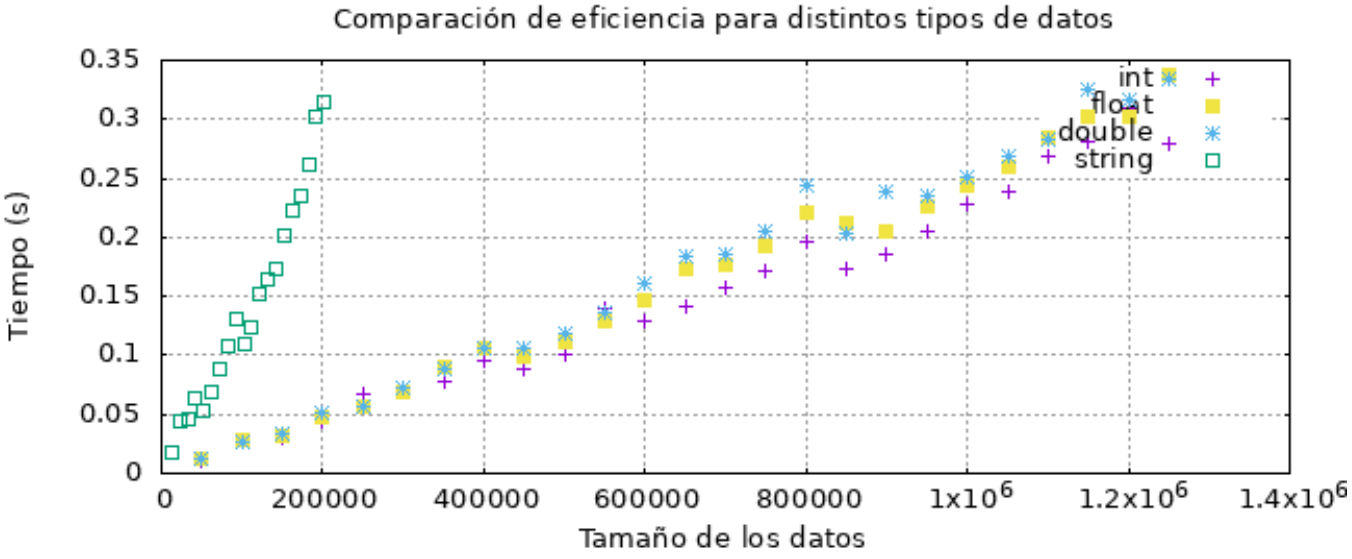


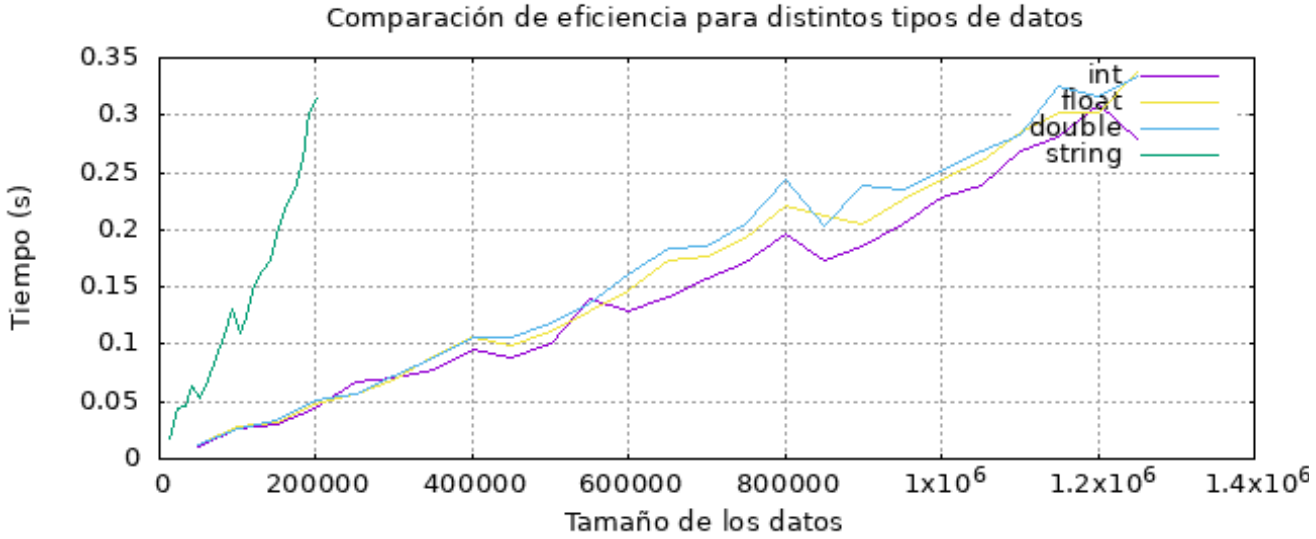
- Ordenación de string:





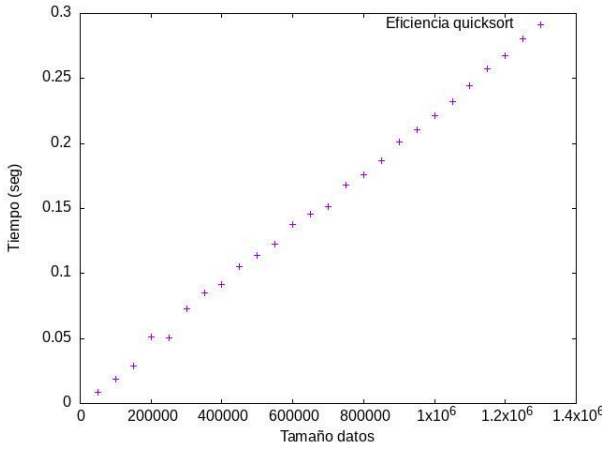
- Comparación de eficiencia para distintos tipos de datos

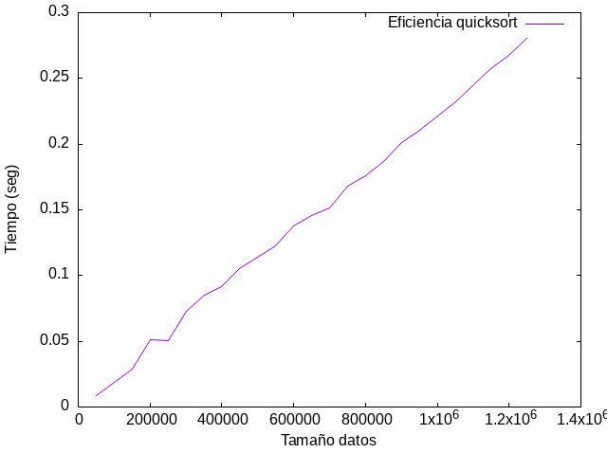




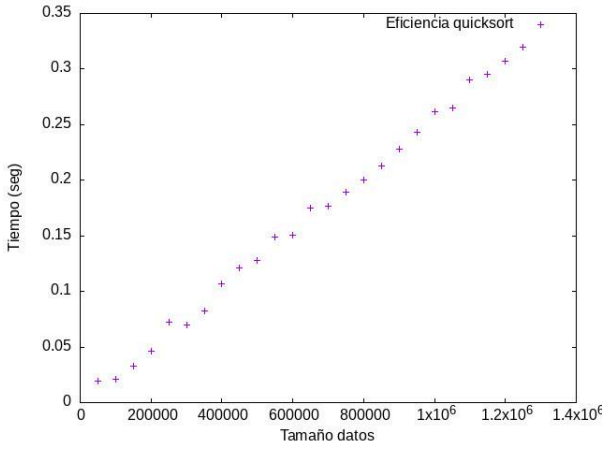
Quicksort

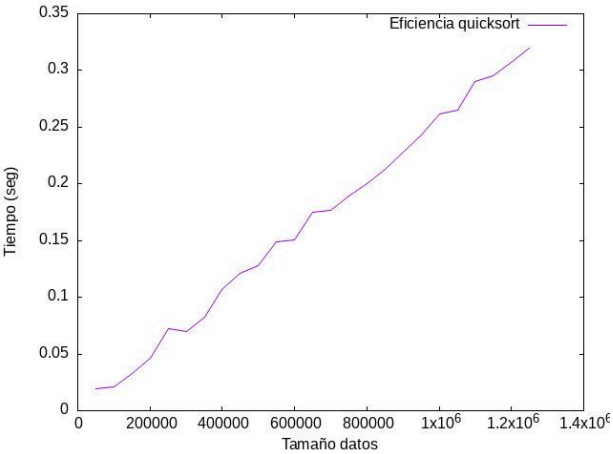
- Ordenación de int:



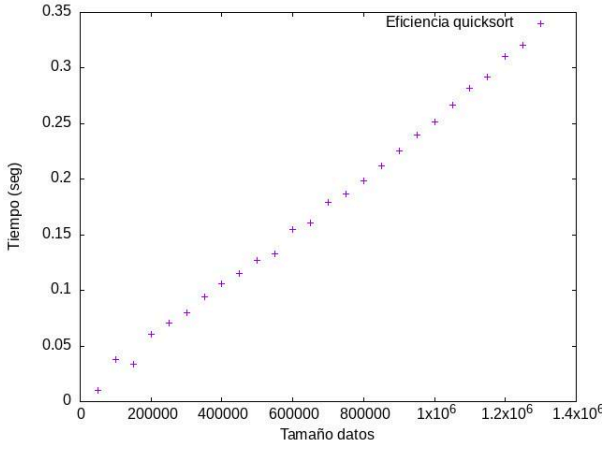


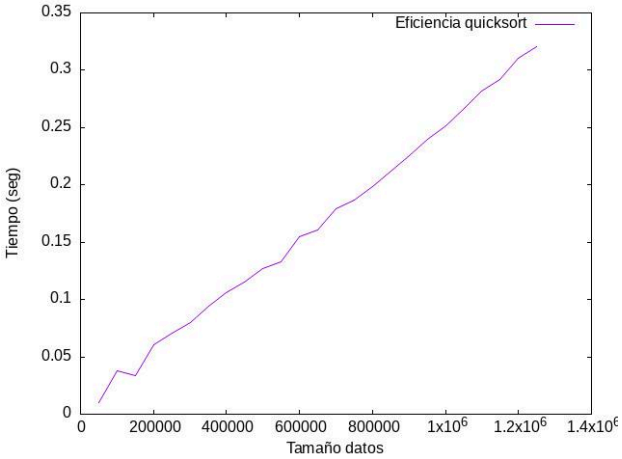
- Ordenación de float:



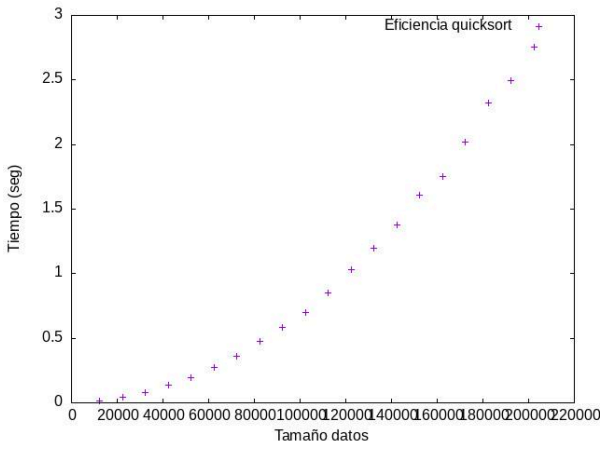


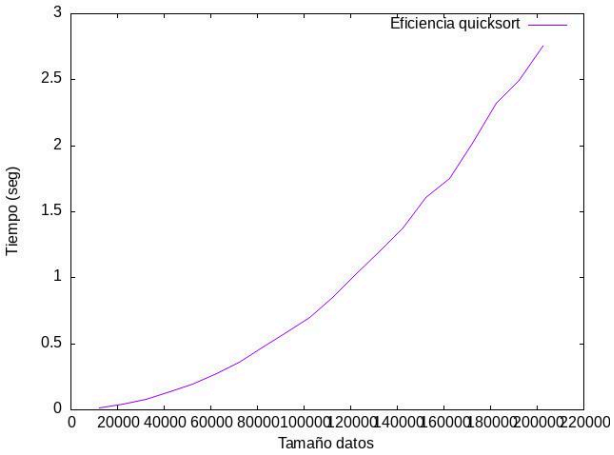
- Ordenación de double:



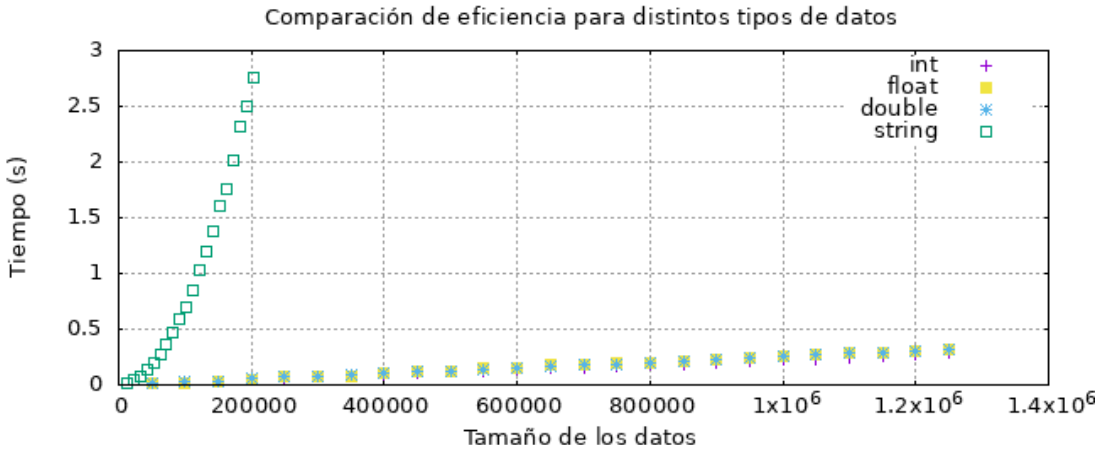


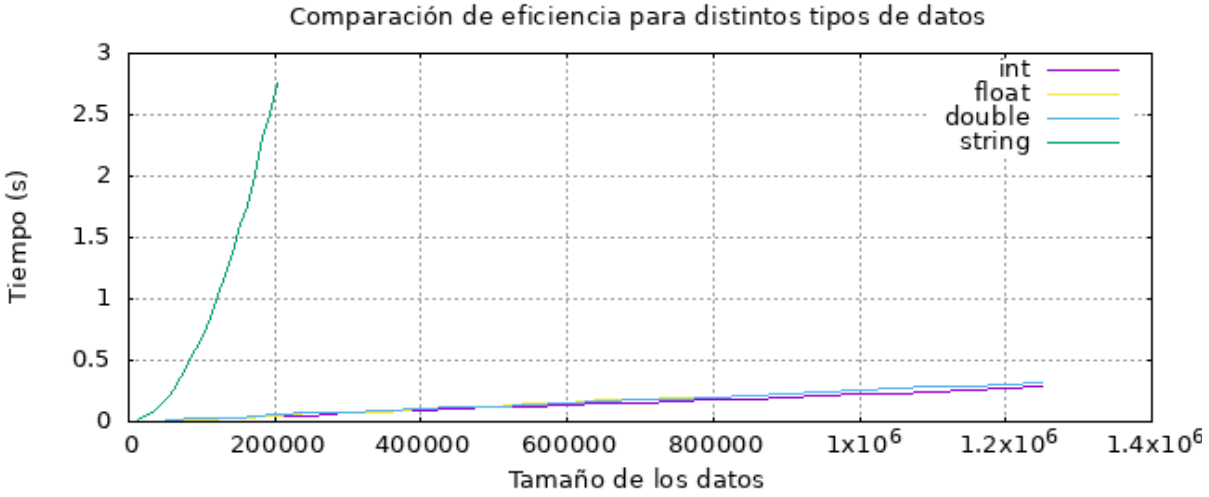
- Ordenación de string:



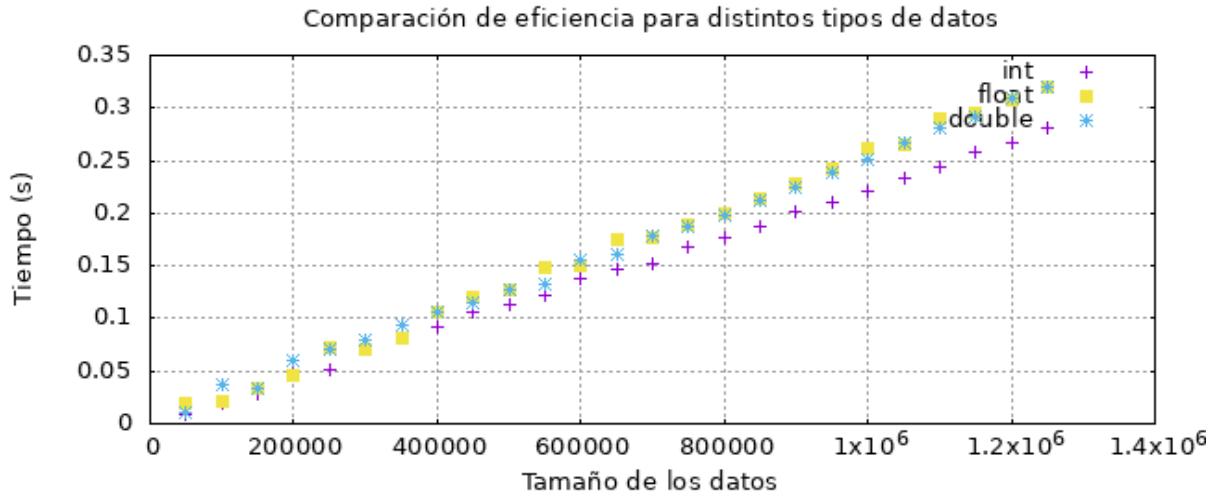


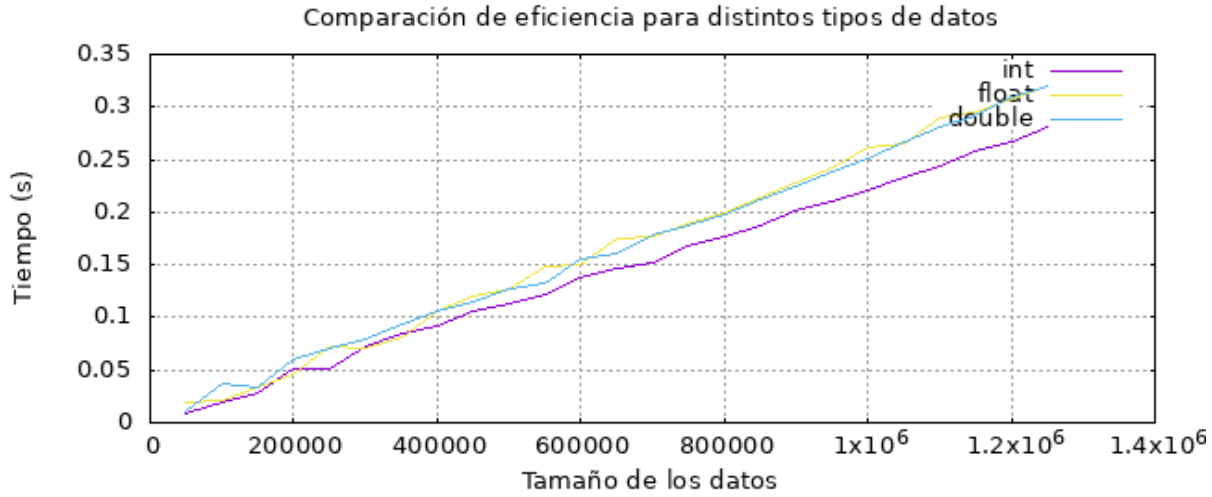
- Comparación de eficiencia para distintos tipos de datos





Ya que los datos de tipo string escalan de forma muchísimo más rápida (cuadrática), veamos la gráfica sin dichos datos:





Antes de pasar a comparar los algoritmos entre sí, conviene dar una explicación a esos picos que aparecen en las gráficas. Mucho más visibles en las de mergesort. Esto es debido a que son algoritmos muy sensibles al tamaño de cada dato de entrada, al fin y al cabo, si comprendemos su funcionamiento nos damos cuenta de que pueden variar mucho sus tiempos de ejecución en función de los propios datos del vector con el que estén trabajando. Como los datos son generados aleatoriamente para rellenar el vector, ocurren estas situaciones límite que nos demuestran una vez más, la importancia de realizar análisis más allá de lo teórico.

Análisis de la eficiencia híbrida

Como hemos observado en las gráficas anteriores, en especial en los gráficos de líneas, la forma de la gráfica se corresponde a la de una función lineal logarítmica (excepto el caso peor de quicksort), cosa que encaja con el análisis teórico

previamente realizado. Nuestro siguiente objetivo será averiguar la expresión concreta de la función lineal-logarítmica, es decir, realizar una regresión con la función

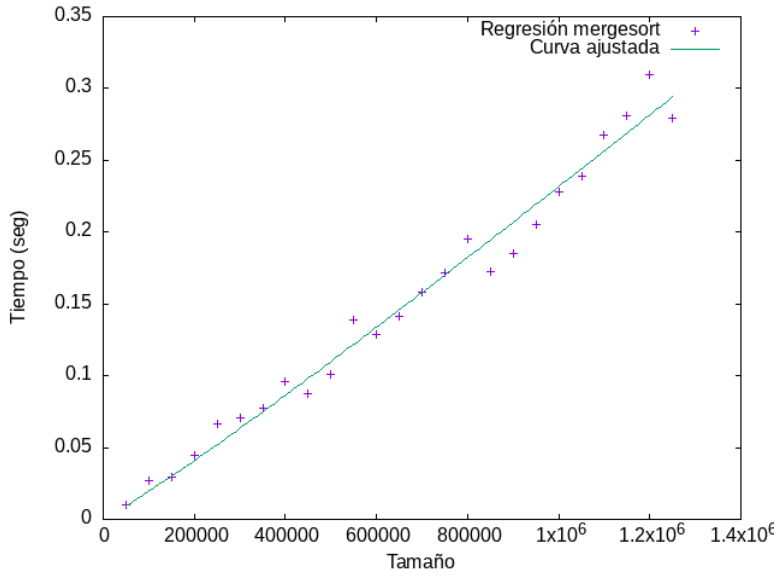
$$f(x) = ax \cdot \log(bx)$$

Procedemos por tanto a realizar la regresión con cada uno de los tipos de datos y con cada uno de los algoritmos. En las gráficas además viene etiquetada la función de regresión con la constante oculta.

Mergesort

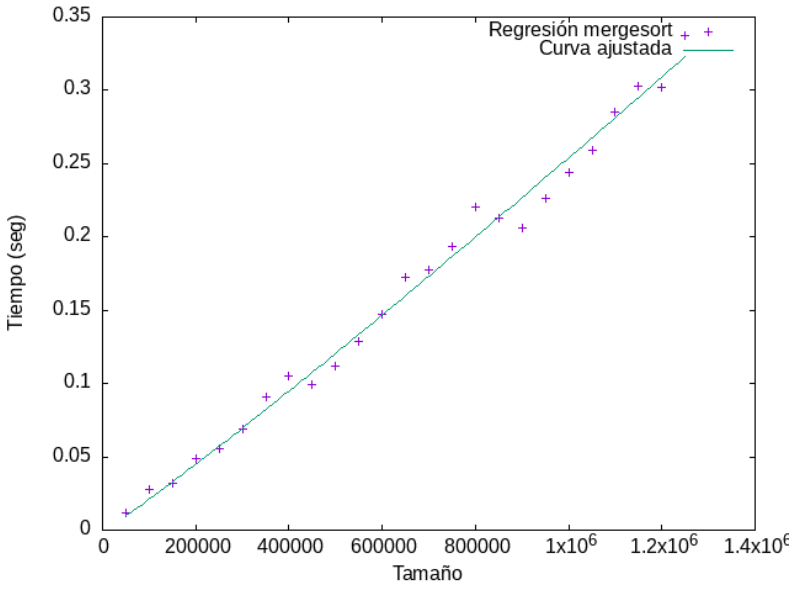
- Regresión de int:

$$f(x) = 1.6853 \cdot 10^{-8} \cdot x \log(0.928984 \cdot x)$$



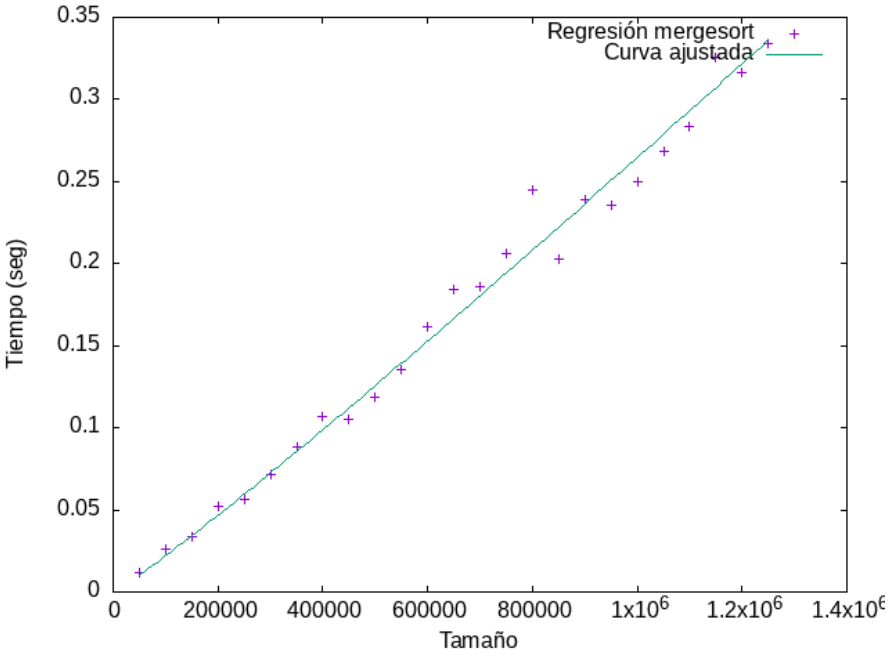
- Regresión de float:

$$f(x) = 1.84883 \cdot 10^{-8} \cdot x \log(0.928984 \cdot x)$$



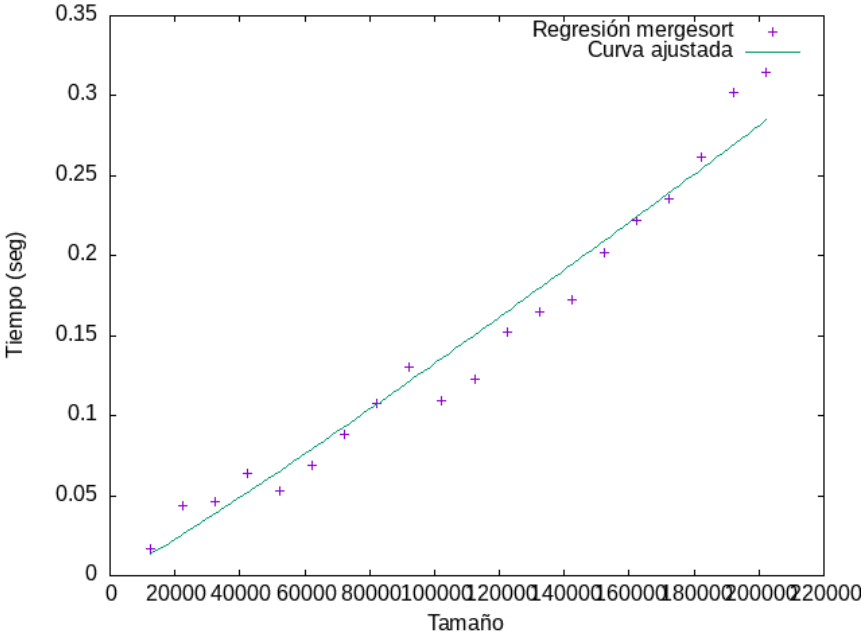
- Regresión de double:

$$f(x) = 1.92603 \cdot 10^{-8} \cdot x \log(0.928984 \cdot x)$$



- Regresión de string:

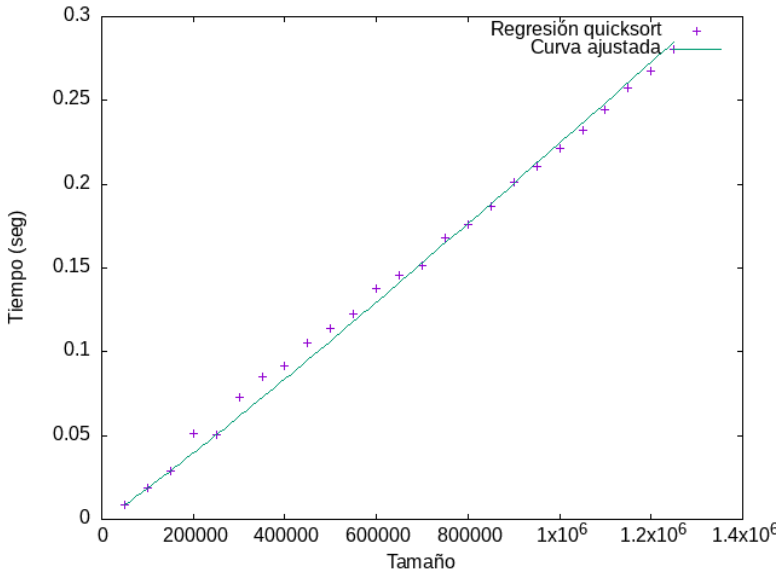
$$f(x) = 1.16046 \cdot 10^{-7} \cdot x \log(0.918677 \cdot x)$$



Quicksort

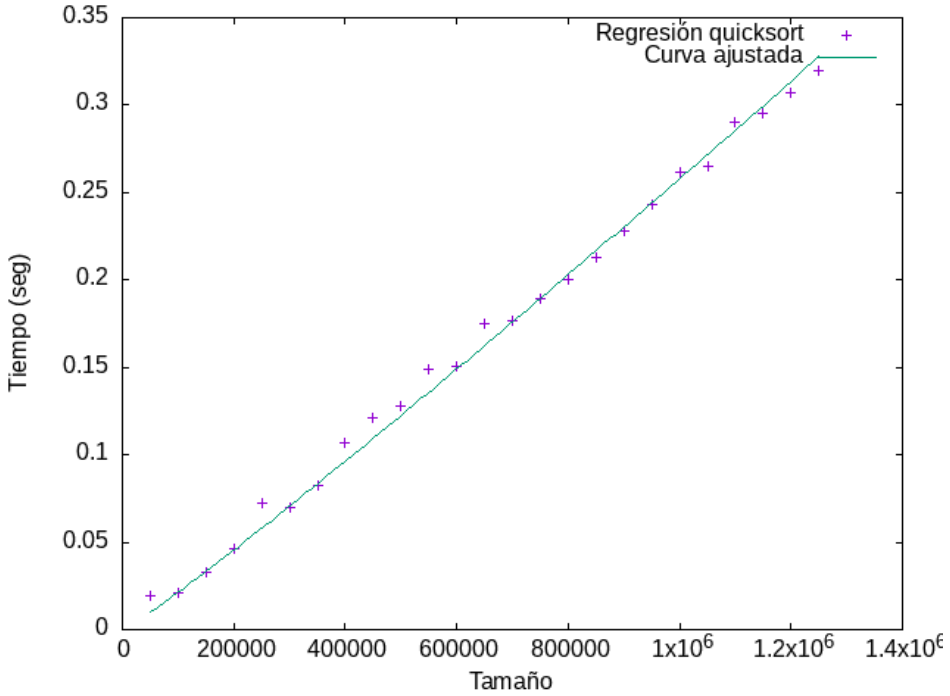
- Regresión de int:

$$f(x) = 1.63346 \cdot 10^{-8} \cdot x \log(0.928984 \cdot x)$$



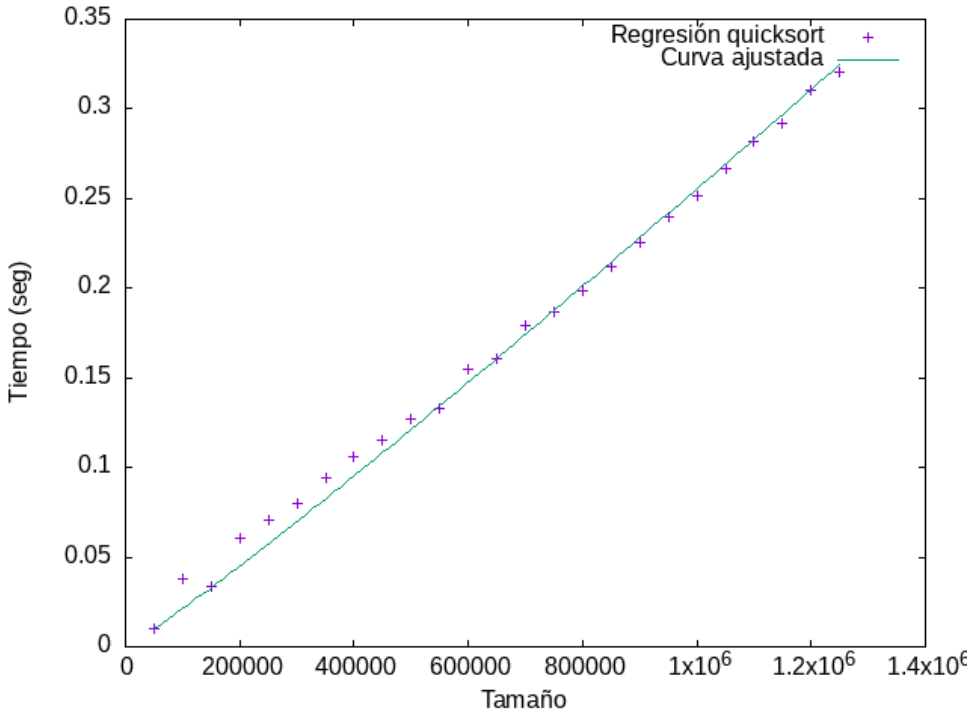
- Regresión de float:

$$f(x) = 1.87788 \cdot 10^{-8} \cdot x \log(0.928984 \cdot x)$$



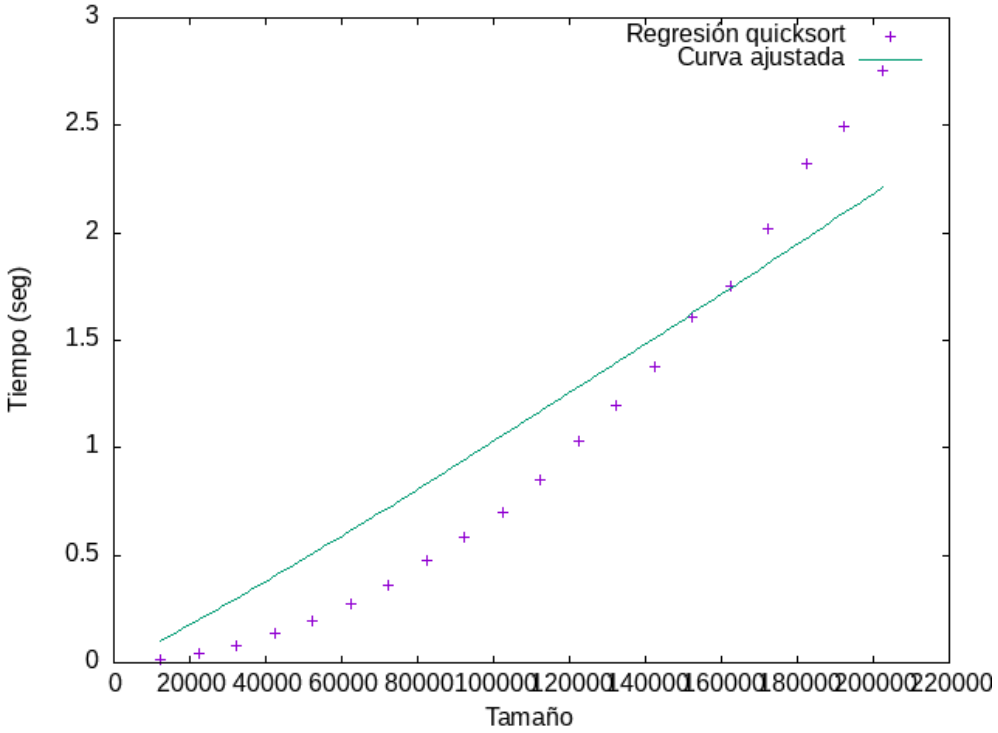
- Regresión de double:

$$f(x) = 1.85954 \cdot 10^{-8} \cdot x \log(0.928984 \cdot x)$$

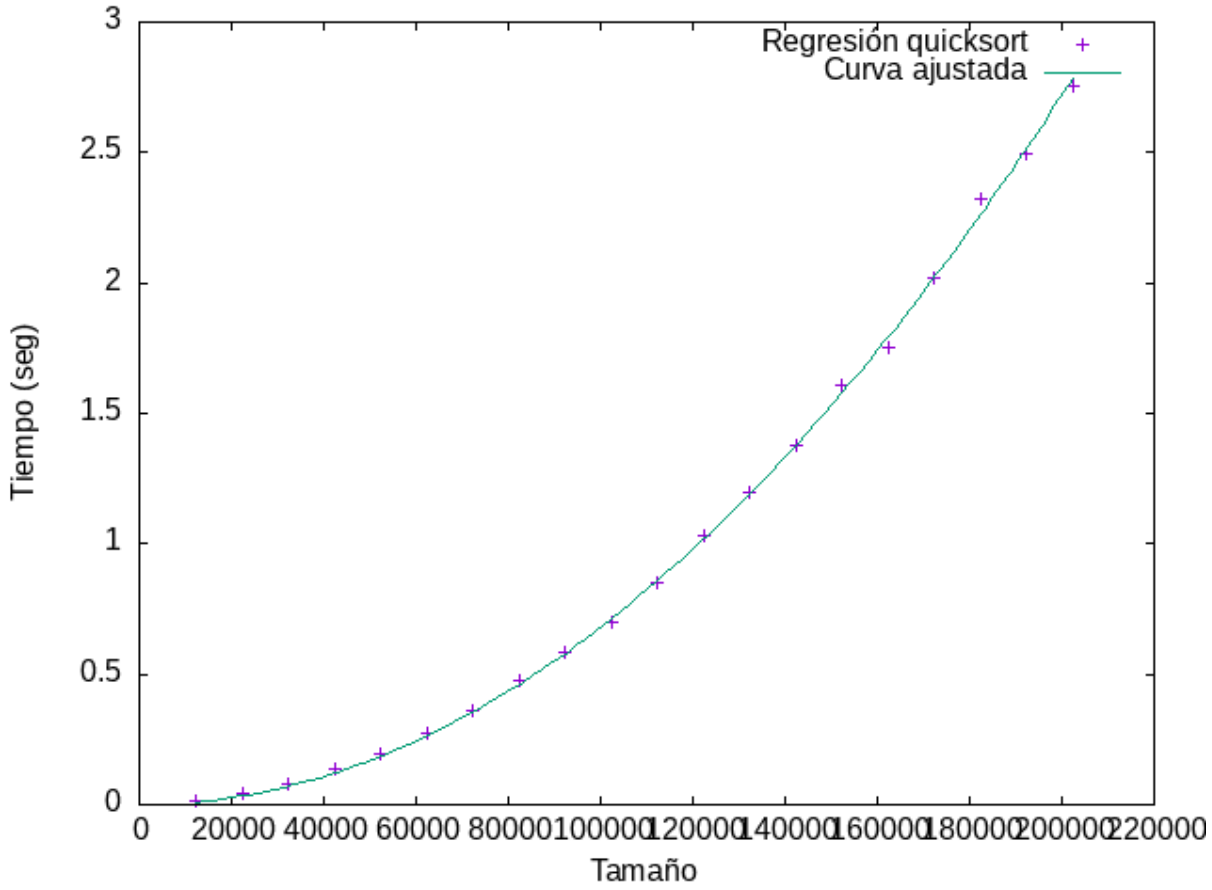


- Regresión de string:

$$f(x) = 9.0051 \cdot 10^{-7} \cdot x \cdot \log(0.918677 \cdot x)$$



Se puede observar que todas las regresiones son buenas, pues las curvas pasan muy cerca de los puntos graficados por gnuplot, lo que reafirma nuestro resultado del análisis teórico. Exceptuando por el string para quicksort, lo cual recordemos, se debe a que quicksort, aunque suela comportarse como nlog(n), realmente es n^2, Esto lo vemos pues ajustando por una cuadrática en el caso de los string (que demuestra la teoría), nos queda así:



$$f(x) = 6.80208 \cdot 10^{-11} \cdot x^2$$

4.3. Algoritmos cúbicos

Floyd

A continuación se desarrolla un estudio con enfoque híbrido de la eficiencia del algoritmo de Floyd-Warshall para el cálculo de los caminos mínimos en grafos dirigidos y ponderados. A diferencia del conocido algoritmo de Dijkstra para el cálculo de todos los caminos mínimos de todos los nodos a cierto nodo fuente, este algoritmo calcula la totalidad de los caminos mínimos para cada uno de los pares de vértices en el grafo en una única ejecución. Téngase en cuenta que la variación del algoritmo que va a explicarse y estudiarse tan solo calcula el coste del camino mínimo: el propio camino mínimo es recuperable sin más que añadir un par de líneas de código que no afectan a la eficiencia ni a la complejidad del código.
La idea del algoritmo es considerablemente simple:

Supónganse numerados los nodos del 0 al $n - 1$, siendo n el número de nodos en el grafo. Procedemos de la siguiente manera:

- Partimos del caso básico conocido del camino mínimo entre cada 2 nodos sin usar ningún nodo intermedio, de tal forma que si dos nodos están conectados por una arista entonces el coste del camino mínimo es el peso de esa arista, y si no lo están entonces ponemos el coste del camino mínimo como infinito (o algún límite estándar en el lenguaje de programación que utilicemos para valores positivos).
- En cada iteración del algoritmo, se añade flexibilidad para el coste mínimo en tanto que se permite usar un nodo más: en la k-ésima iteración, se permiten los k primeros nodos como nodos intermedios en el cálculo de los caminos mínimos entre cada dos nodos, de tal forma que en la $(n - 1)$-ésima y última iteración hay total flexibilidad y obtenemos el verdadero camino mínimo entre cada dos nodos.

La corrección del algoritmo reside en una sencilla inducción: es claro que en el caso de $k = 0$, tenemos el coste mínimo usando k nodos intermedios para cada par de nodos i, j. Suponiendo que en la k-ésima iteración tenemos el coste mínimo entre ciertos dos nodos i y j, nos aseguramos de tenerlo también en la $(k + 1)$-ésima escogiendo el coste menor entre éste y el del camino que siga la ruta:

$$i \rightarrow (k + 1) \rightarrow j$$

Donde, recuérdese, i y j son los nodos inicial y final, y $(k + 1)$ es el nodo intermedio que se permite en la iteración $(k + 1)$-ésima. Esto es claro, pues de haber

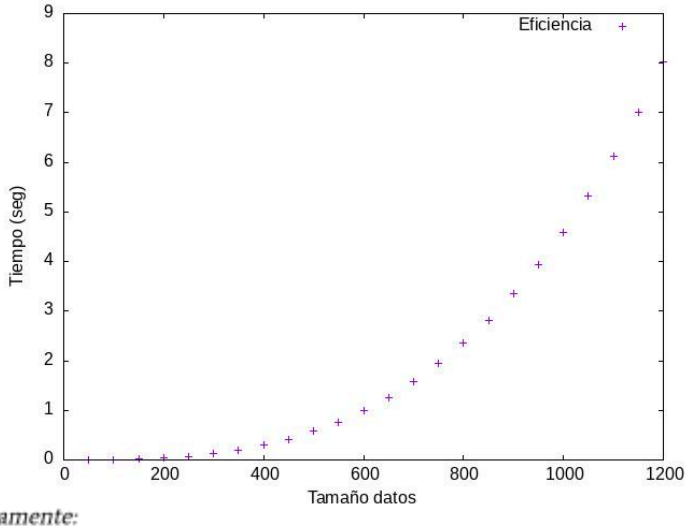
un camino de coste menor (al que era posible en la iteración k-ésima) en la iteración $(k+1)$-ésima, debe ser precisamente el que parta del nodo inicial, pase por el nodo $(k+1)$-ésimo y llegue al nodo final, haciendo en todos estos recorridos caminos mínimos que ya han sido calculados en anteriores iteraciones. Queda más clara la idea viendo una implementación del algoritmo:

```
void Floyd(int **M, int dim)
{
    for (int k = 0; k < dim; k++)
      for (int i = 0; i < dim;i++)
     for (int j = 0; j < dim;j++)
           {
       int sum = M[i][k] + M[k][j];
            M[i][j] = (M[i][j] > sum) ? sum : M[i][j];
           }
}
```

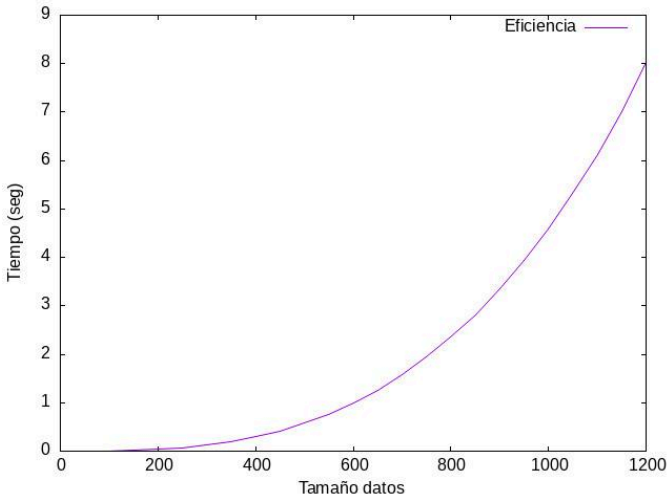
Donde i, j, k son lo esperable, la matriz dinámica M guarda el coste del camino mínimo hasta ese momento calculado entre cada dos nodos del grafo, y dim es el número de nodos en el grafo.

De esta manera, es totalmente evidente que el la eficiencia teórica en peor caso de este algoritmo es $O(n^3)$, dado el triple anidamiento de los bucles y el hecho de que todas las operaciones hechas en el interior del bucle más interno son de eficiencia teórica constante.

En efecto, y comenzando la parte "empírica" o práctica de nuestro estudio de la eficiencia de este algoritmo, los resultados de las pruebas de eficiencia hechas siguiendo los procedimientos descritos en la sección 4 son los siguientes:



Datos obtenidos empíricamente:



Usando las instrucciones provistas por el profesor, calculamos por mínimos cuadrados una función polinómica de grado 3 (la prevista por el cálculo de la eficiencia teórica) que se ajuste a nuestros datos proporcionados, y obtenemos, asumiendo la siguiente forma para la función: $f(x) = a_0x^3 + a_1x^2 + a_2x + a_3$ los siguientes resultados:

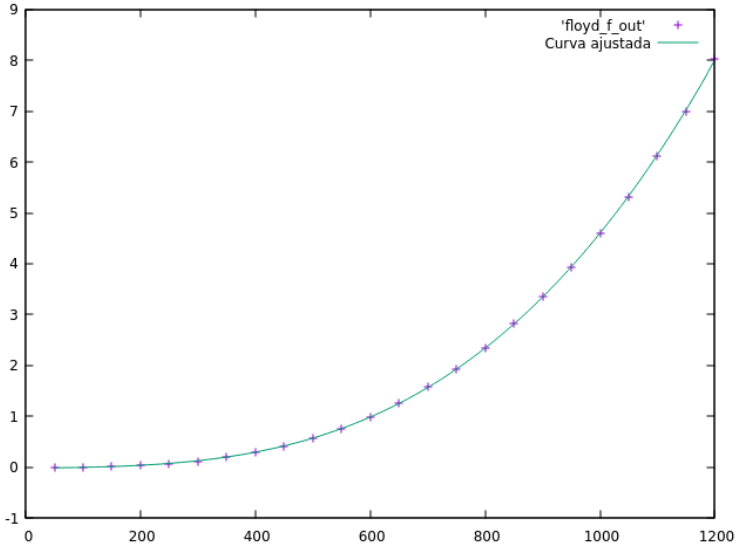
$a_0 = 4.89091e-09$

$a_1 = -4.75075e-07$

$a_2 = 0.000201522$

$a_3 = -0.0184917$

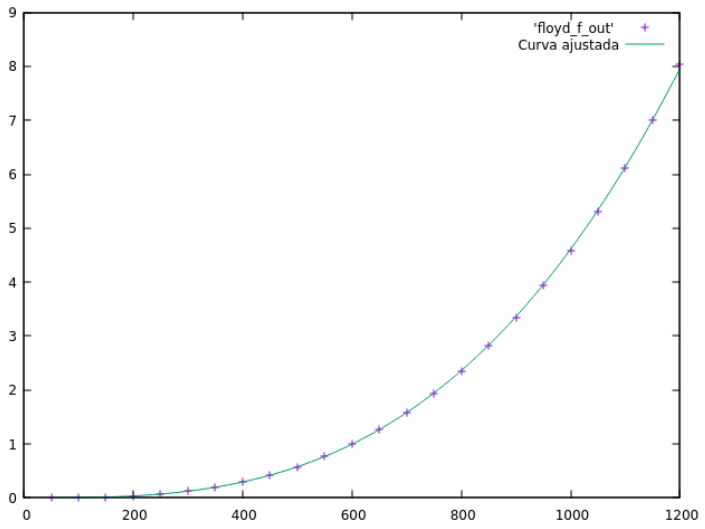
El ajuste obtenido puede verse en la siguiente gráfica:



Ajuste obtenido usando un polinomio de grado tres con todos sus coeficientes

Probando a ajustar con una función más fiel a lo que nos sugería la eficiencia teórica, tomamos: $f(x) = a_0 x^3$, y obtenemos:

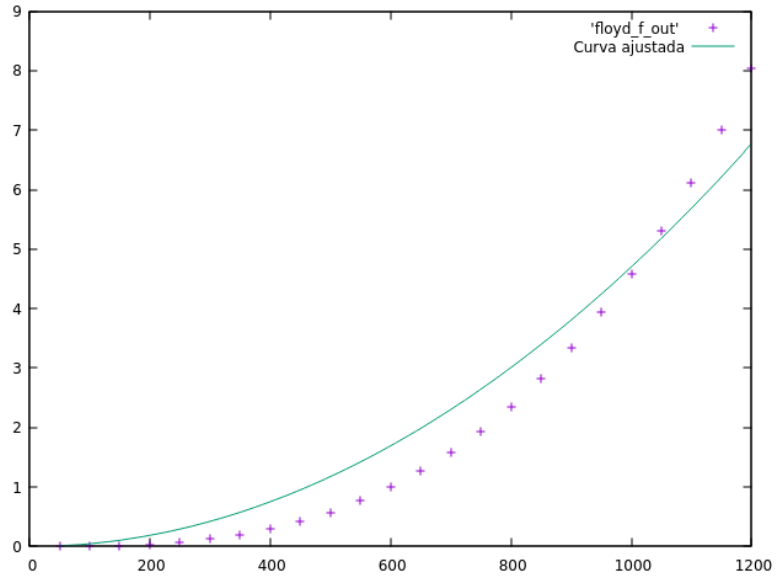
$a_0 = 4.6087e-09$



Ajuste obtenido usando un polinomio mónico de grado tres

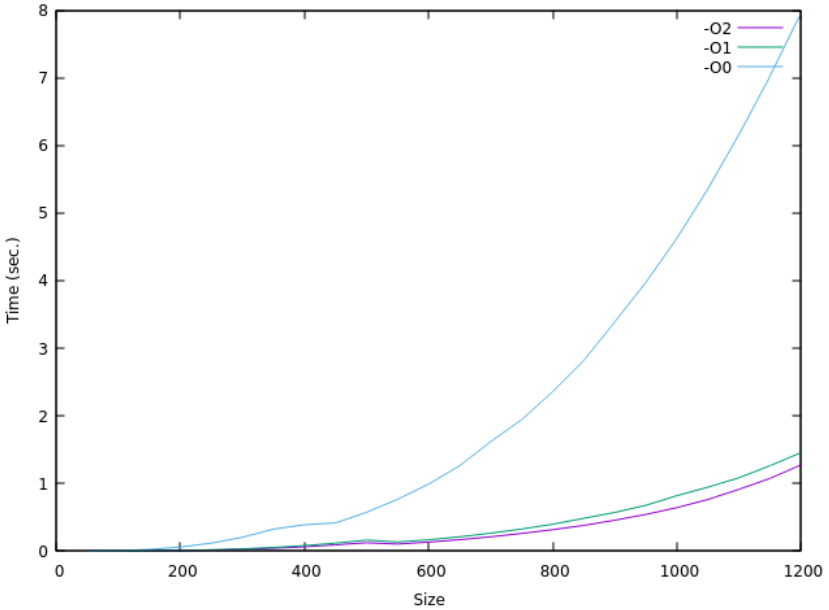
Un ajuste de, también, bastante calidad.

Para afianzar más aún nuestra convicción sobre la corrección en la práctica de nuestra predicción teórica sobre la eficiencia del algoritmo, también hemos probado a intentar ajustar los datos obtenidos en las sucesivas ejecuciones del programa con una curva cuadrática, y hemos obtenido lo esperado: incorrección



Ajuste obtenido usando un polinomio de grado dos

Como extra, se representan juntas ahora las eficiencias del algoritmo de Floyd usando diferentes órdenes de compilación, para ver hasta qué punto es algo que debamos tener en cuenta si buscamos rapidez en nuestras ejecuciones:



Comparación en función de la eficiencia impuesta al compilador: "g++ -O[0,1,2] floyd.cpp -o floyd"

Observamos diferencias más que relevantes.

4.4. Algoritmos exponenciales

Fibonacci

Algoritmo

El algoritmo de fibonacci es un algoritmo recursivo que calcula la sucesión de números de fibonacci en la cual los dos primeros términos son 1 y el resto son iguales a la suma de los dos anteriores:

$$\begin{aligned} fibo(n) &= 1 && si \quad n < 2 \\ fibo(n) &= fibo(n-1) + fibo(n-2) && si \quad n \geq 2 \end{aligned}$$

De aquí nos surge la famosa sucesión: 1, 1, 2, 3, 5, 8, 13, 21, ...

Implementación

La implementación de este algoritmo ha sido llevada a cabo mediante una función recursiva que calcula el n-ésimo término de la sucesión en base al algoritmo recurrente previamente mostrado:

```
int fibo(int n)
{
  if (n < 2)
    return 1;
  else
    return fibo(n - 1) + fibo(n - 2);
}
```

Eficiencia teórica

Calculemos la eficiencia teórica del algoritmo. Analizando el código observamos que la función consiste en una condición ($if/else$). Estudiemos la eficiencia de cada rama por separado:

- $if\ (n < 2)$:
  En el caso de que la condición sea cierta, o lo que es lo mismo, estemos en uno de los casos bases del algoritmo, vemos que la eficiencia teórica es constante al tratarse meramente de devolver un literal y por tanto es O(1).

- $else\ (n \geq 2)$:
  En el caso de que la condición sea falsa, es decir, no se trata de un caso base, la ecuación de la eficiencia teórica es la siguiente:
  $$T(n) = T(n-1) + T(n-2) + 1$$
  Puesto que se trata de dos llamadas a la misma función para $n-1$ y $n-2$ y la suma de estas (la cual es constante).
  Para resolver la ecuación teórica, dado que se trata de una ecuación recurrente, aplicaremos el **método de la ecuación característica**. Para ello distinguimos dos tipos:
  - Ecuaciones Lineales Homogéneas de coeficientes constantes (ELH):
    $$T(n) = a_1 T(n-1) + a_2 T(n-2) + \ldots + a_k T(n-k)$$
  - Ecuaciones Lineales No Homogéneas de coeficientes constantes (ELNH):
    $$a_0 T(n) + a_1 T(n-1) + \ldots + a_{n-k} T(n-k) =$$
    $$= b_1{}^n q_1(x) + b_2{}^n q_2(x) + \ldots + b_z{}^n q_z(x)$$

  Podemos observar que en nuestro caso se trata de una ecuación lineal no homogénea de coeficientes constantes (ELNH), dado que tomando $b = 1$ y $q(x) = 1$ (polinomio de grado 0), tenemos:
  $$1 = 1^n * 1 = b_1{}^n q_1(x)$$
  Procedamos a obtener la ecuación característica (ELNH):
  1. Pasamos los términos recurrentes a un lado:
     $$T(n) - T(n-1) - T(n-2) = 1$$
  2. Resolvemos la parte recurrente como si fuera una ecuación homogénea, es decir, resolvemos:
     $$T(n) - T(n-1) - T(n-2) = 0$$
     2.1. Reescribimos $T(n-i)$ como x^i:
     $$x^n - x^{n-1} - x^{n-2} = 0$$
     2.2. Sacamos x^{n-k} como factor común:
     $$x^{n-2}(x^2 - x - 1) = 0$$
     Como $x^{n-k} \neq 0$ (los tiempos no pueden ser 0) lo podemos sacar de la ecuación al estar esta igualada a 0, y nos queda la ecuación característica de la ELN a la cual denominados polinomio característico:
     $$p_H(x) = x^2 - x - 1$$
     2.3. Por el **Teorema Fundamental del Álgebra**, sabemos que, notando como r_i a las raíces del polinomio $p_H(x)$:

$$p_H(x) = (x - r_1)(x - r_2)\dots(x - r_k)$$

Por tanto calculando las raíces del polinomio:

$p_H(x) = 0 \Rightarrow x^2 - x - 1 = 0 \Rightarrow x = \frac{1 \pm \sqrt{1-4*1*(-1)}}{2*1} = \frac{1 \pm \sqrt{5}}{2}$

Y nos queda:

$$p_H(x) = (x - \frac{1+\sqrt{5}}{2})(x - \frac{1-\sqrt{5}}{2})$$

3. Una vez obtenido el polinomio característico de la parte homogénea $p_H(x)$, la fórmula del polinomio característico de la ELNH es:

$$P(x) = p_H(x)(x - b_1)^{d_1+1}(x - b_2)^{d_2+1}\dots(x - b_z)^{d_z+1}$$

siendo d_i el grado del polinomio $q_i(x)$. Por tanto dicho polinomio nos queda:

$$P(x) = (x - \frac{1+\sqrt{5}}{2})(x - \frac{1-\sqrt{5}}{2})(x - 1)^1$$

4. Por último, sacamos el tiempo de ejecución de la siguiente expresión:

$$t_n = \sum_{i=1}^{r} \sum_{j=0}^{M_i - 1} c_{ij} r_i^n n^j$$

donde:

- t_n es el tiempo de ejecución para n
- c_{ij} coeficientes constantes
- r es el número de raíces distintas del polinomio característico
- r_i las raíces del polinomio característico
- M_i la multiplicidad de las raíces del polinomio característico

Por tanto el tiempo de ejecución para n nos queda:

$$t_n = c_{10}(\frac{1+\sqrt{5}}{2})^n + c_{20}(\frac{1-\sqrt{5}}{2})^n + c_{30} * 1^n \in O((\frac{1+\sqrt{5}}{2})^n)$$

En conclusión, en caso de que $n > 2$ la eficiencia del algoritmo es $O((\frac{1+\sqrt{5}}{2})^n)$.

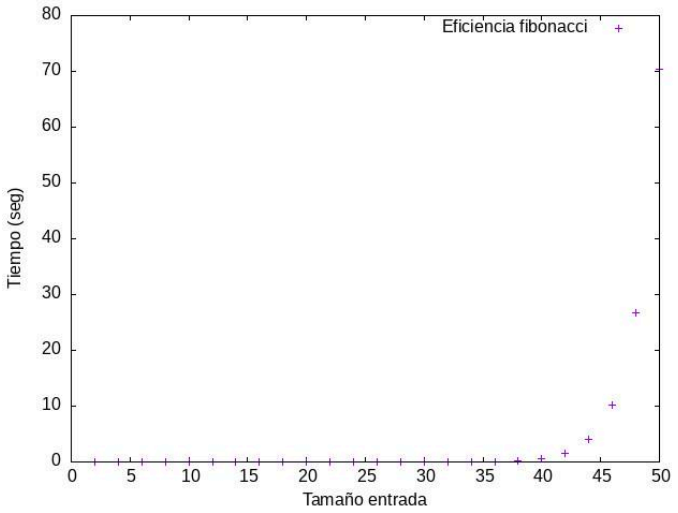
Al tratarse de una condición, la eficiencia del algoritmo es $O(máx(O(if), O(else)) = O(máx(O(1), O((\frac{1+\sqrt{5}}{2})^n) = O((\frac{1+\sqrt{5}}{2})^n)$.

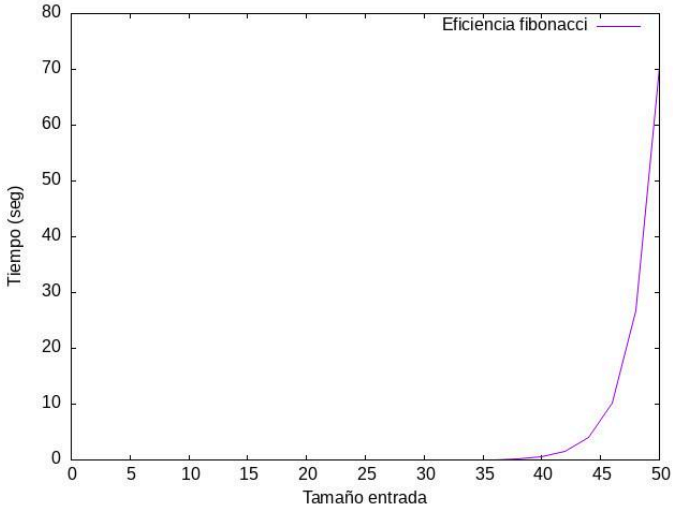
Eficiencia empírica

Para realizar el cálculo de la eficiencia empírica se han tomado los tiempos de lo que tardaba en ejecutarse el algoritmo de Fibonacci variando el tamaño de la entrada (que

en este caso es el parámetro n que indica el número de la sucesión que desea calcularse). En particular para los casos más pequeños (n más pequeña) hemos tomado la media del tiempo que tardaban 30 ejecuciones.
Los valores de entrada testados han sido desde $n = 2$ hasta $n = 50$ con saltos de 2 en 2, y hemos considerado n pequeña hasta $n = 41$.
Los datos y las gráficas obtenidas han sido:

Tamaño entrada(n)	Tiempo (seg)
2	3.33E-08
4	3.33E-08
6	6.67E-08
8	1.67E-07
10	3.33E-07
12	8.33E-07
14	2.3E-06
16	5.93E-06
18	1.42E-05
20	3.72E-05
22	9.73E-05
24	0.000255767
26	0.000678733
28	0.00175757
30	0.00462313
32	0.0121136
34	0.0323394
36	0.085879
38	0.223684
40	0.584576
42	1.5658
44	3.98274
46	10.1841
48	26.7216
50	70.3743





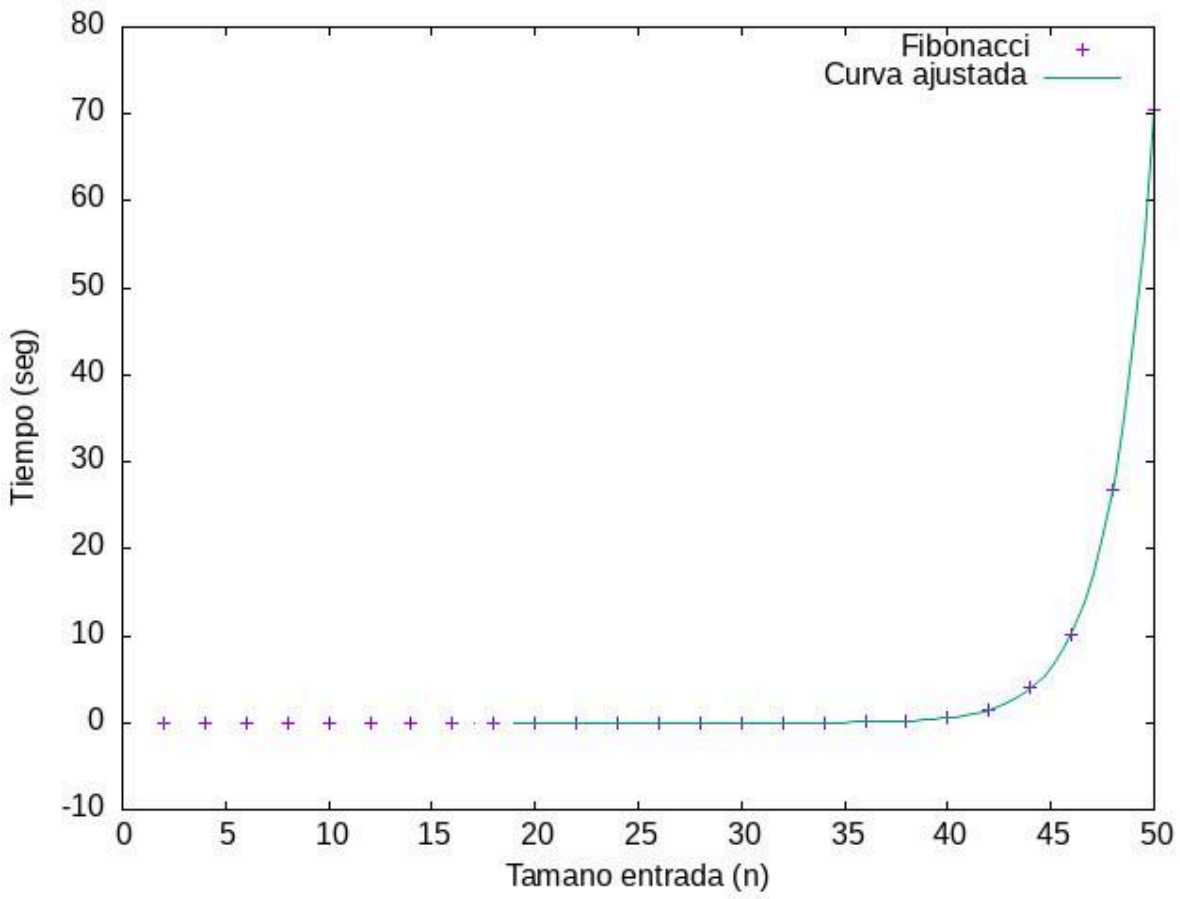
Observando las gráficas constatamos que el tiempo de ejecución crece exponencialmente respecto al tamaño de la entrada, confirmando la predicción del estudio teórico de la eficiencia de que fibonacci tiene una eficiencia del orden **exponencial**.

Eficiencia híbrida

Previamente hemos calculado que la eficiencia de fibonacci es $O((\frac{1+\sqrt{5}}{2})^n)$. En particular en el cálculo teórico obtuvimos que la ecuación que nos daba el tiempo de ejecución del algoritmo era $t_n = c_{10}(\frac{1+\sqrt{5}}{2})^n + c_{20}(\frac{1-\sqrt{5}}{2})^n + c_{30}$. Para describir completamente esta ecuación, necesitamos conocer las constantes ocultas $c_{10}, c_{20}\ y\ c_{30}$. Para ello, tomaremos la ecuación calculada teóricamente y los datos obtenidos empíricamente y procederemos a realizar un ajuste por mínimos cuadrados.

```
Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a                = 2.49847e-09     +/- 1.414e-12    (0.05659%)
b                = -8.45989e-05    +/- 0.1042       (1.231e+05%)
c                = 2.34089e-05     +/- 0.009024     (3.855e+04%)
```

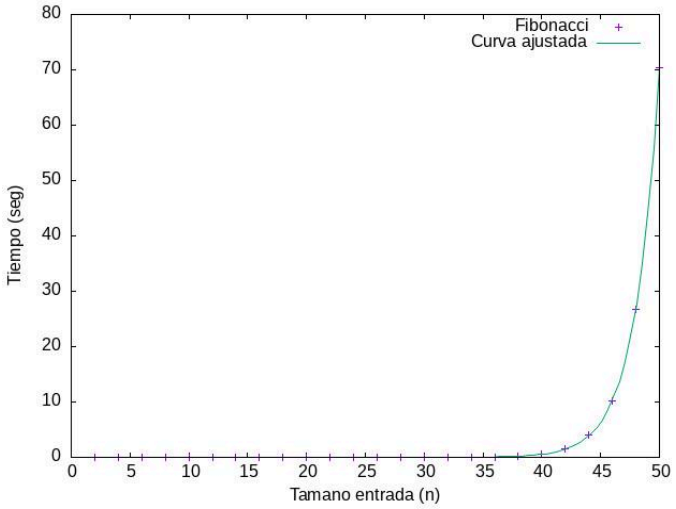
Por tanto la ecuación de tiempo real de fibonacci es $f(x) = 2.49847 * 10^{-9}((\frac{1+\sqrt{5}}{2})^x - 8.45989 * 10^{-5}((\frac{1-\sqrt{5}}{2})^x + 2.34089 * 10^{-5}$



Como podemos apreciar, el ajuste a los puntos es perfecto.

Realmente podríamos usar directamente $t_n = a(\frac{1+\sqrt{5}}{2})^n$. Definiendo de esta forma la curva de regresión obtenemos:

```
iter      chisq       delta/lim  lambda   a
   0 9.2737268756e+20   0.00e+00  6.09e+09    1.000000e+00
   1 1.3718530881e+18  -6.75e+07  6.09e+08    3.846154e-02
   2 2.1932100198e+11  -6.26e+11  6.09e+07    1.538096e-05
   3 3.5458874497e+00  -6.19e+15  6.09e+06    2.559982e-09
   4 3.6779713518e-02  -9.54e+06  6.09e+05    2.498468e-09
   5 3.6779713518e-02  -1.39e-08  6.09e+04    2.498468e-09
iter      chisq       delta/lim  lambda   a

After 5 iterations the fit converged.
final sum of squares of residuals : 0.0367797
rel. change during last iteration : -1.39043e-13

degrees of freedom    (FIT_NDF)                        : 24
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 0.039147
variance of residuals (reduced chisquare) = WSSR/ndf   : 0.00153249

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a               = 2.49847e-09      +/- 1.285e-12    (0.05145%)
```

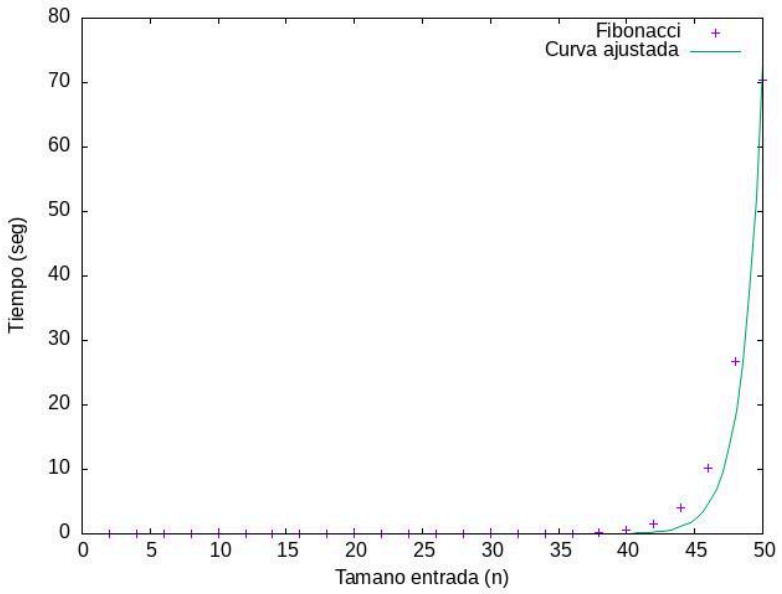


Observamos que la expresión de la curva ajustada es $f(x) = 2.49847 * 10^{-9}((\frac{1+\sqrt{5}}{2})^x$ y se ajusta perfectamente a todos los puntos. (Notamos que nos da el mismo valor para el coeficiente de a, de forma que el peso del resto de términos de la ecuación es insignificante).

Otros ajustes

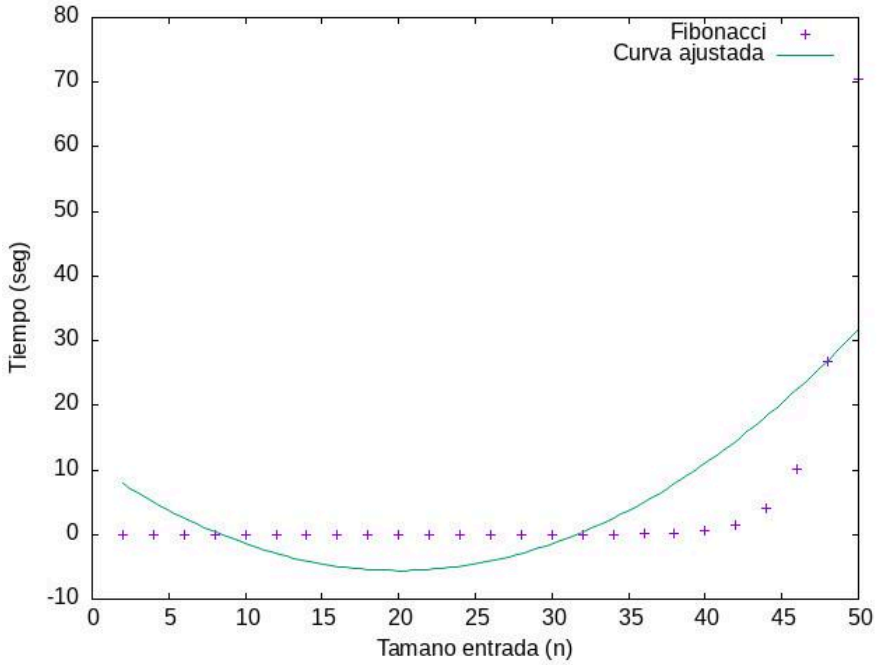
Probemos cómo serían los ajustes con otras ecuaciones distintas a la calculada teóricamente:

- $f(x) = a * 2^n$ (la expresión de la eficiencia de Hanoi, como veremos más adelante).



Podemos observar que el ajuste no es perfecto, pero tampoco es del todo erróneo al tratarse también de una expresión exponencial con una base relativamente próxima a la real.

- $f(x) = ax^3 + bx^2 + cx + d$



Podemos observar que este ajuste si que es claramente erróneo, no pudiendo ajustar una exponencial mediante un polinomio de grado 3.

Hanoi

El algoritmo de Hanoi es un algoritmo recursivo utilizado para resolver el problema de las torres de Hanoi. Este consiste en tener inicialmente tres torres, de las cuales la primera tiene apiladas N fichas de mayor a menor tamaño. El problema es pasar las N fichas de la primera a la tercera torre teniendo en cuenta que solo se puede mover una ficha en un sólo paso y una ficha de mayor tamaño no puede apilarse sobre una más pequeña.

Para resolverlo vamos a dividir el problema. Para mover las N fichas de la primera torre a la tercera, tenemos que mover en particular la ficha más grande y, de hecho, esta tiene que estar en el fondo de la tercera torre. Por tanto, si conseguimos mover primero las N-1 primeras fichas a la segunda torre, que funcionará como torre auxiliar, podremos mover la ficha más grande a la torre destino y, finalmente, solo nos faltará mover las N-1 fichas de la segunda torre a la tercera torre. Es decir, si tenemos el problema resuelto para el caso N-1, lo tendremos para N. Luego, solo tenemos que reducir el problema a casos más pequeños y, sabemos que para el caso N = 1 solo tenemos que mover una ficha. Así, el problema está resuelto.

Este algoritmo implementado en C++ tiene el siguiente código:

```
void hanoi (int M, int i, int j)
{
    if (M > 0)
    {
        hanoi(M-1, i, 6-i-j);
        cout << i << " -> " << j << endl;
        hanoi (M-1, 6-i-j, j);
    }
}
```

ESTUDIO EFICIENCIA TEÓRICA:

Es fácil ver que si n es el número de datos de entrada y T es la función que calcula el tiempo de ejecución del algoritmo de Hanoi en función del tamaño de entrada, T tendría la siguiente expresión:

$$T(n) = 2\,T(n-1) + 1 \quad si\ n\ != 1\ con \quad T(1) = 1$$

Desarrollándose en serie sería:

$$T(n) = 2\,T(n-1) + 1 = 2\,(2\,T(n-2) + 1) + 1 = 2^2\,T(n-2) + 2 =$$

$$= 2^k\,T(n-k) + k = 2^{n-1}\,T(1) + n - 1 = 2^{n-1} + n - 1$$

Lo cual quiere decir que T(n) es de orden $O(2^n)$.

ESTUDIO EFICIENCIA EMPÍRICA:

Para realizar el cálculo de la eficiencia empírica hemos ejecutado código del algoritmo variando los tamaños de entrada (que en este caso es el valor de la variable M que representa el número de fichas), siempre controlando el tiempo que ha tardado el algoritmo en resolver el problema de Hanoi.

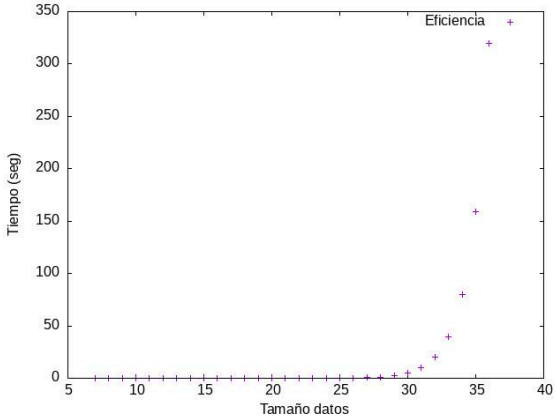
Hemos variado los valores de entrada de 7 a 34 fichas con saltos de 1 y hemos obtenido los siguientes datos:

Tamaño	Tiempo(seg)
7	1.57E-06
8	1.38E-06
9	7.39E-06
10	4.92E-06
11	9.65E-06
12	1.96E-05
13	3.84E-05
14	7.63E-05
15	0.000152415
16	0.000305597
17	0.00063374
18	0.00122143
19	0.00243705
20	0.00487234
21	0.0100298

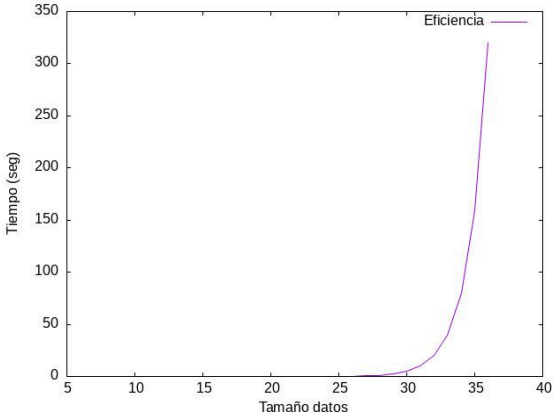
Tamaño	Tiempo(seg)
22	0.0194775
23	0.0411721
24	0.077917
25	0.15752
26	0.316574
27	0.623867
28	1.24699
29	2.49381
30	4.99046
31	9.95991
32	19.9635
33	39.8527
34	79.7783
35	160.9685
36	336.1598

Graficando los datos obtenemos:

- Con puntos



- Con líneas

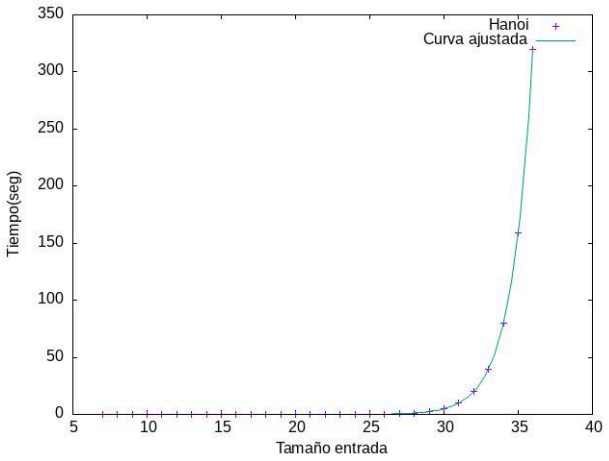


Podemos observar en las gráficas que el tiempo tardado en resolver el problema crece exponencialmente a medida que crece el tamaño de entrada (que es el número de fichas). Luego, como predecía el estudio teórico, el algoritmo de Hanoi tiene orden exponencial.

EFICIENCIA HÍBRIDA:

Dado que teóricamente el algoritmo es del orden $O(2^n)$, definiremos una curva de regresión de la forma $a0*2^n$ que se ajuste a nuestros datos. Veamos los resultados:

```
FIT:    data read from dato_h
        format = z
        x range restricted to [0.00000 : 56.0000]
        #datapoints = 30
        residuals are weighted equally (unit weight)

function used for fitting: f(x)
        f(x)=a0* (2**x)
fitted parameters initialized with current variable values

iter      chisq       delta/lim  lambda   a0
   0 6.2964885853e+21   0.00e+00  1.45e+10    1.000000e+00
   5 1.6352370674e-01  -5.50e-09  1.45e+05    4.646446e-09

After 5 iterations the fit converged.
final sum of squares of residuals : 0.163524
rel. change during last iteration : -5.49939e-14

degrees of freedom    (FIT_NDF)                       : 29
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 0.0750916
variance of residuals (reduced chisquare) = WSSR/ndf   : 0.00563875

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a0              = 4.64645e-09      +/- 9.463e-13    (0.02037%)
```



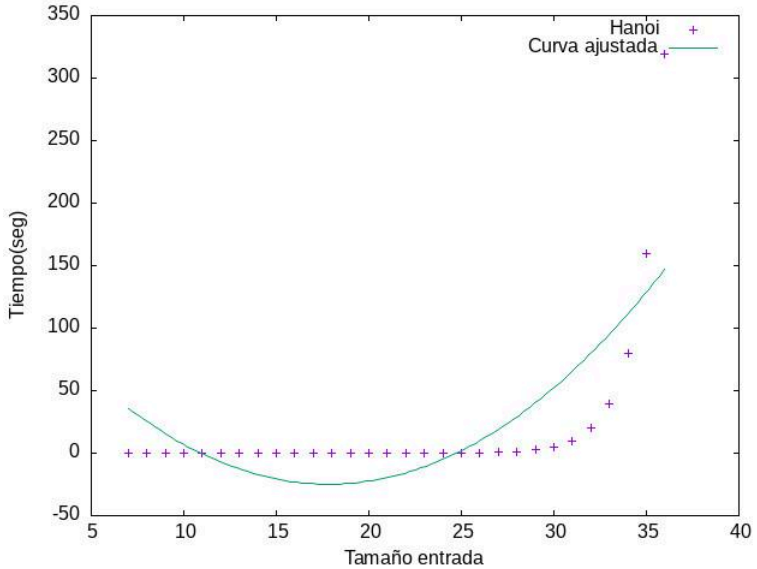
Obtenemos que la expresión de la curva obtenida es f(x) = 4.64E-9 (2^n) y podemos observar que se ajusta casi a la perfección a nuestros datos tanto gráficamente como teóricamente, ya que el valor de la varianza residual y el error asintótico estándar son prácticamente nulos.

OTROS AJUSTES:

Veamos qué pasa si realizamos un ajuste cuadrático o cúbico (es obvio que no merece la pena probar el caso lineal o n-log n):

- $f(x) = a0\ x^2 + a1\ x + a2$

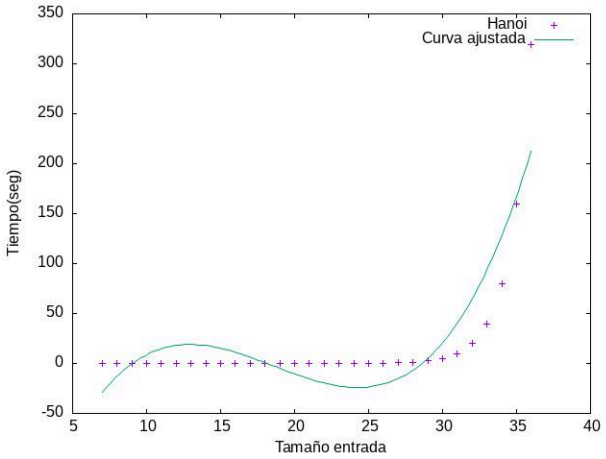
```
*******************************************************************************
Fri Mar  8 08:42:19 2024


FIT:    data read from dato_h
        format = z
        #datapoints = 30
        residuals are weighted equally (unit weight)

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

iter      chisq       delta/lim  lambda   a0            a1
   0 1.2405657448e+07   0.00e+00  3.80e+02    1.000000e+00   1.000006
   6 5.3018173581e+04  -2.59e-03  3.80e-04    5.199349e-01  -1.852276

After 6 iterations the fit converged.
final sum of squares of residuals : 53018.2
rel. change during last iteration : -2.58758e-08

degrees of freedom    (FIT_NDF)                       : 27
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 44.3129
variance of residuals (reduced chisquare) = WSSR/ndf   : 1963.64

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a0              = 0.519935         +/- 0.1209       (23.26%)
a1              = -18.5228         +/- 5.284        (28.53%)
a2              = 140.226          +/- 51.61        (36.81%)
```



Con $f(x) = 0.52x^2 - 18.52x + 140.23$ podemos ver que se ajusta bastante mal a nuestros datos tanto gráficamente como teóricamente teniendo una varianza residual con valor 1964.

- $f(x) = a0x^3 + a1x^2 + a2x + a3$

```
After 8 iterations the fit converged.
final sum of squares of residuals : 25710.9
rel. change during last iteration : -1.2323e-09

degrees of freedom    (FIT_NDF)                        : 26
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 31.4465
variance of residuals (reduced chisquare) = WSSR/ndf   : 988.88

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a0              = 0.0595916        +/- 0.01134      (19.03%)
a1              = -3.32372         +/- 0.7365       (22.16%)
a2              = 56.0919          +/- 14.69        (26.18%)
a3              = -279.502         +/- 87.87        (31.44%)
```



Podemos apreciar que con $f(x) = 0.059x^3 - 3.32x^2 + 56.09x - 259.502$, un polinomio cúbico, tampoco se ajusta a nuestros datos, teniendo una varianza residual de valor 988.88.

Luego el mejor ajuste es indudablemente el exponencial.

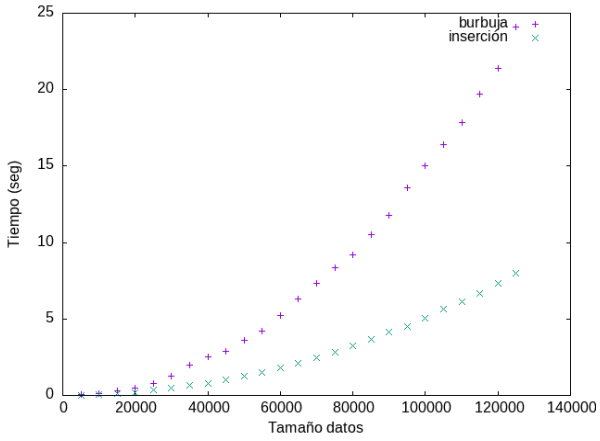
5. ESTUDIO COMPARATIVO EFICIENCIA

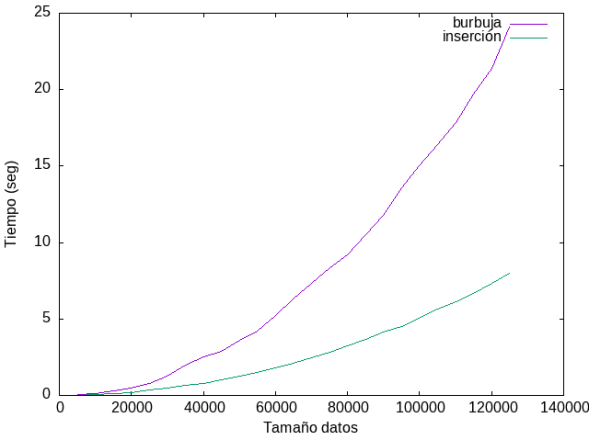
A continuación procedemos al estudio comparativo de la eficiencia entre los algoritmos del mismo orden de eficiencia, a excepción del algoritmo de Floyd, que es el único cúbico. En este estudio observaremos cómo distintos algoritmos a pesar de tener el mismo orden de eficiencia o similares pueden tener tiempos de ejecución significativamente diferentes.

5.1. Comparación eficiencia algoritmos cuadráticos

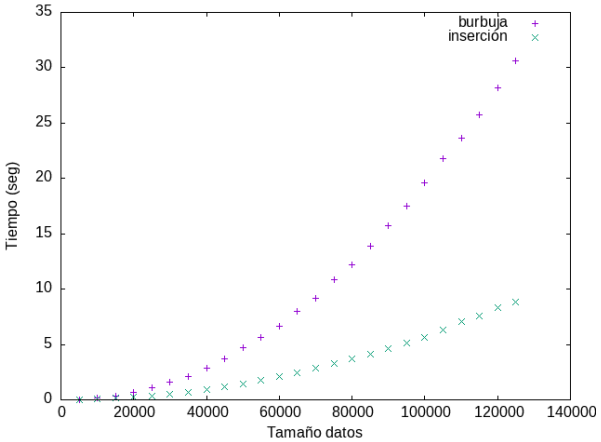
En primer lugar realizaremos la comparación entre los algoritmos cuadráticos, burbuja e inserción, para cada uno de los distintos tipos de datos:

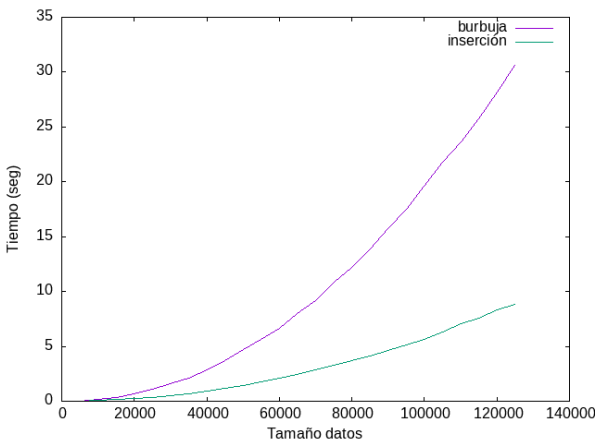
- Ordenación de int:



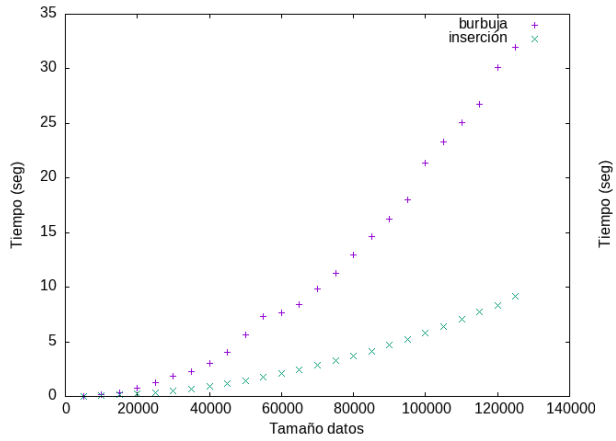


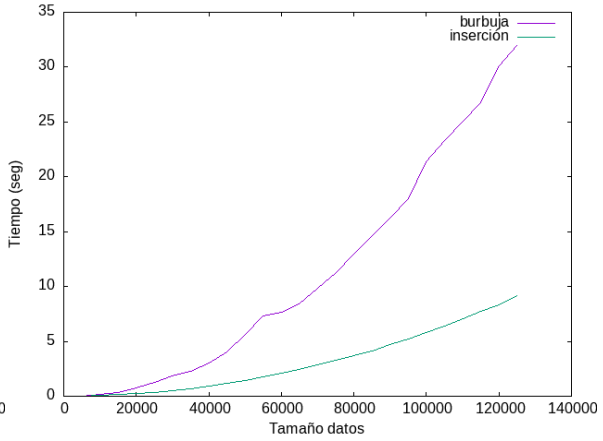
- Ordenación de float:



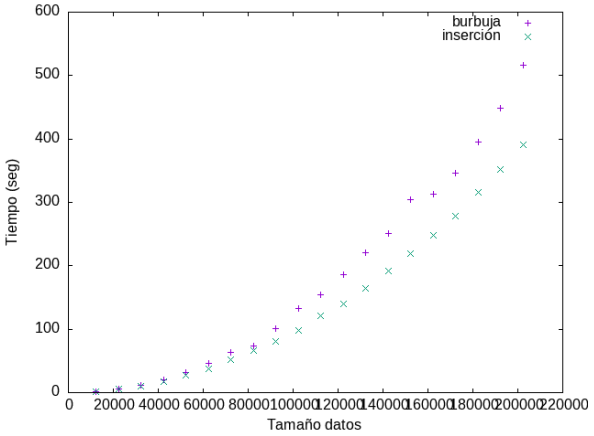


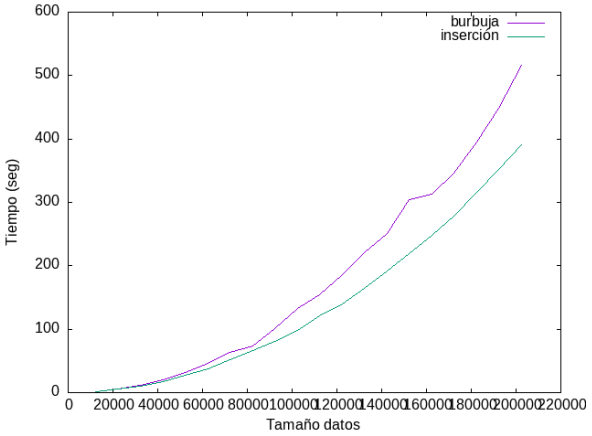
- Ordenación de double:





- Ordenación de string:



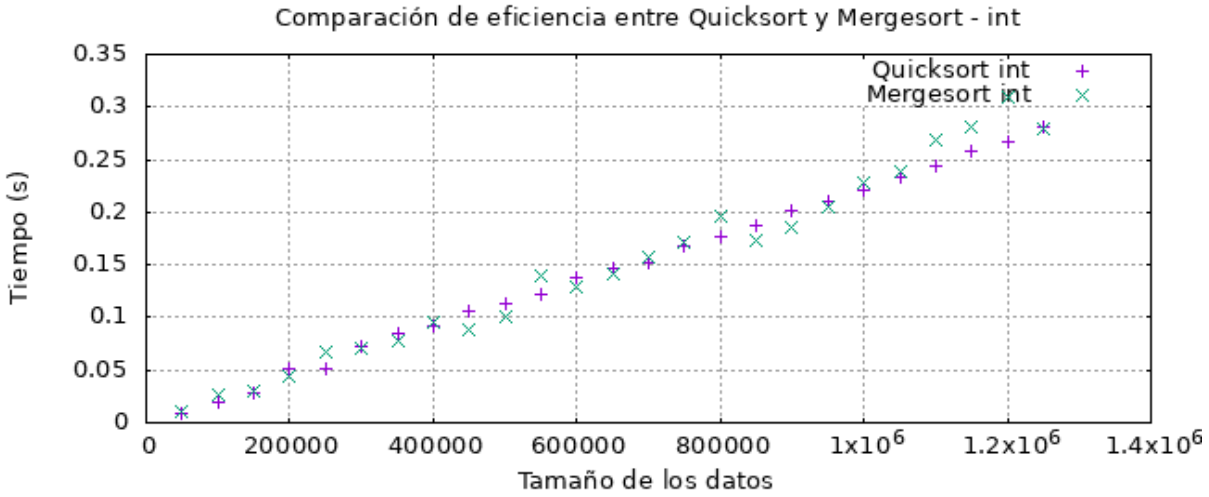


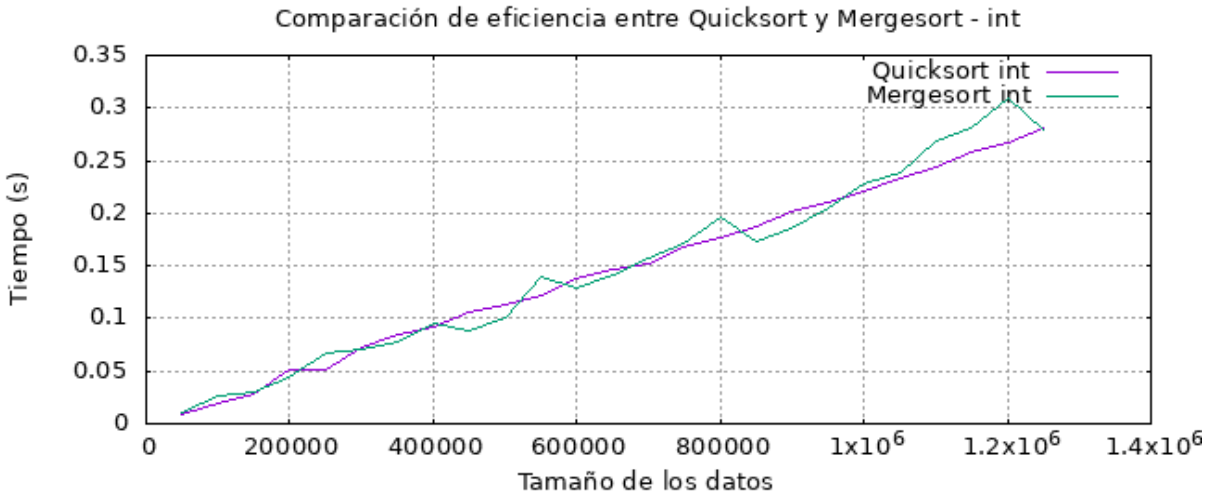
Observamos en las gráficas anteriores que el método de ordenación por burbuja es más lento que la ordenación por inserción, a pesar de que ambos tienen el mismo orden de eficiencia. Esto ya lo podíamos ver mejor en el cálculo previo de las constantes ocultas al realizar las regresiones cuadráticas.

5.2. Comparación eficiencia algoritmos lineal-logarítmicos

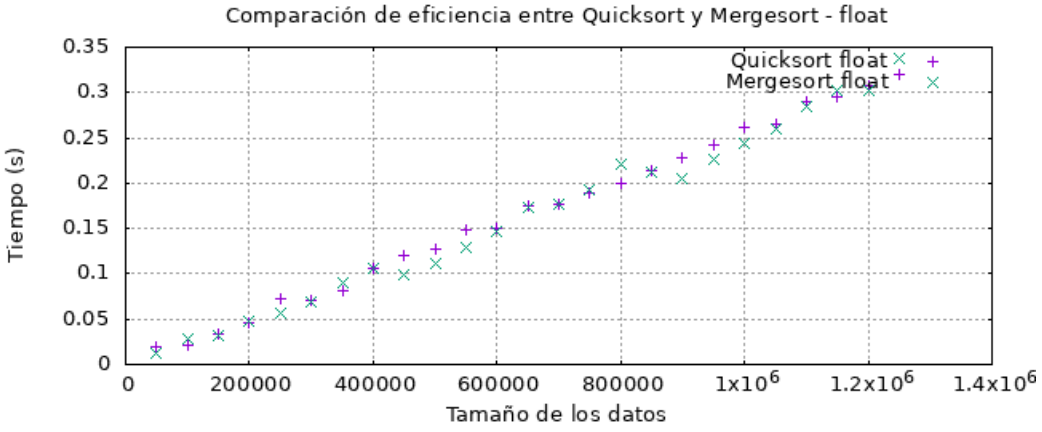
Veamos ahora las diferencias entre los tiempos de ejecución del *quicksort* y el *mergesort*:

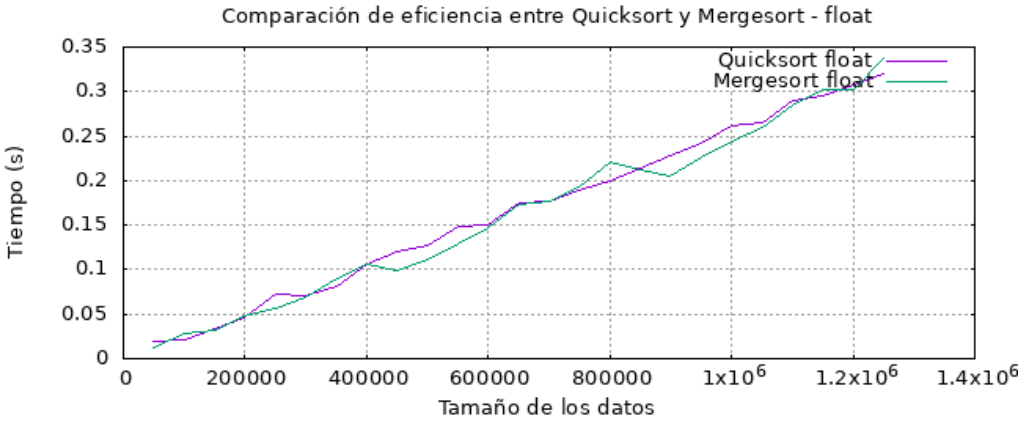
- Ordenación de int:



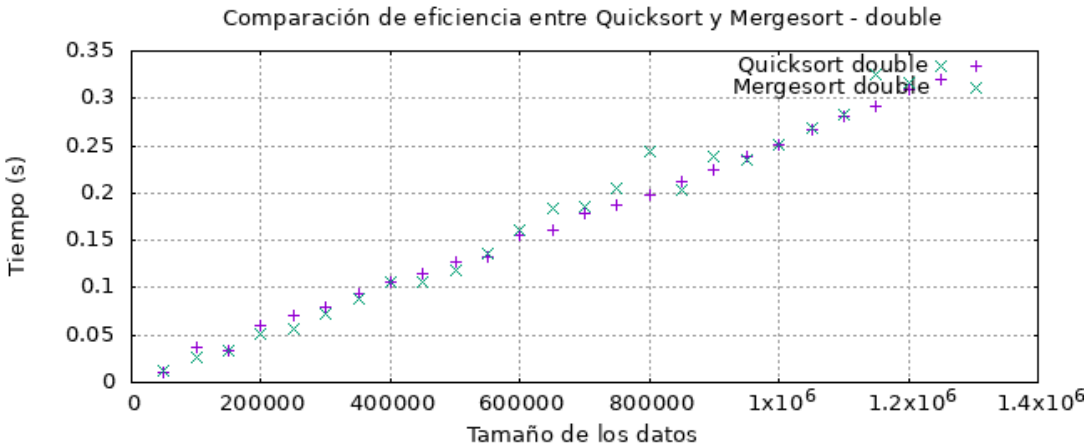


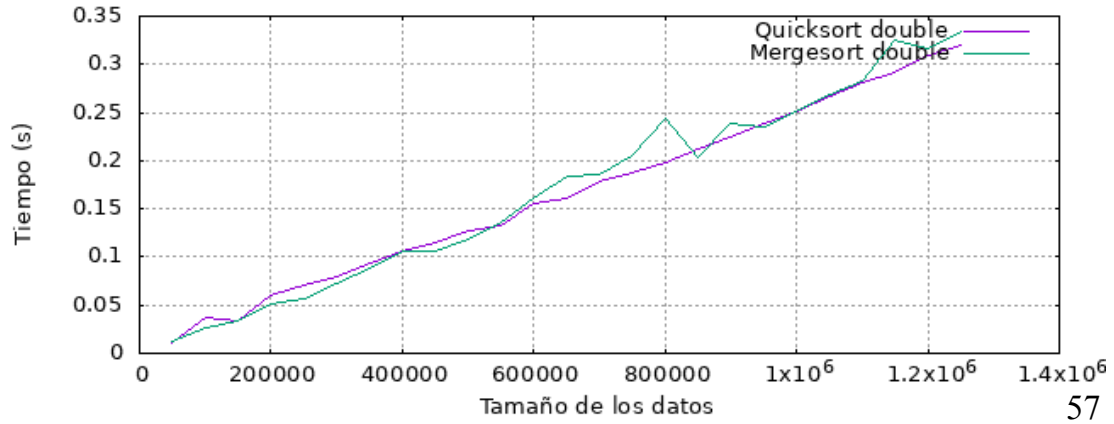
- Ordenación de float:



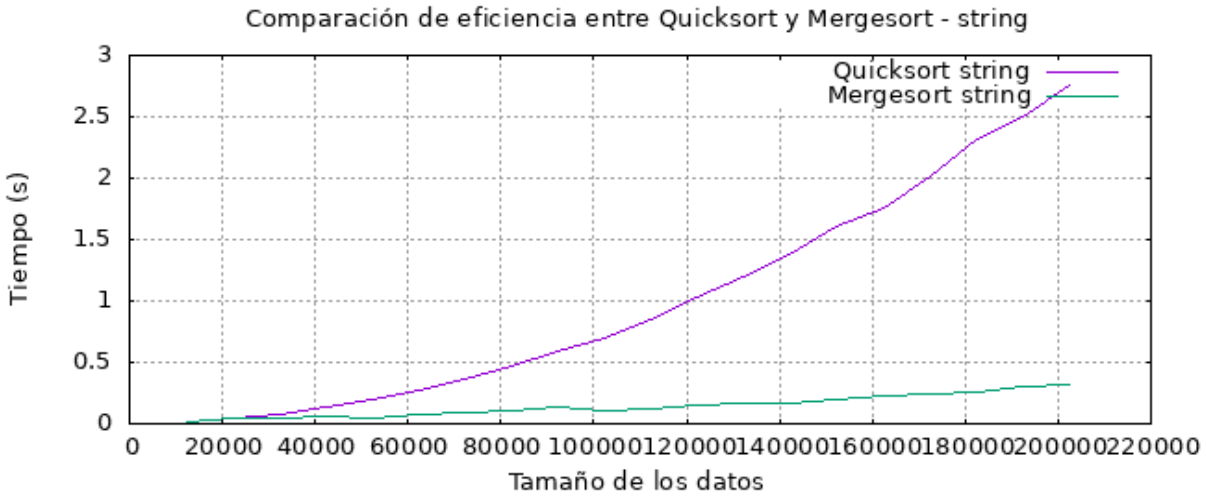


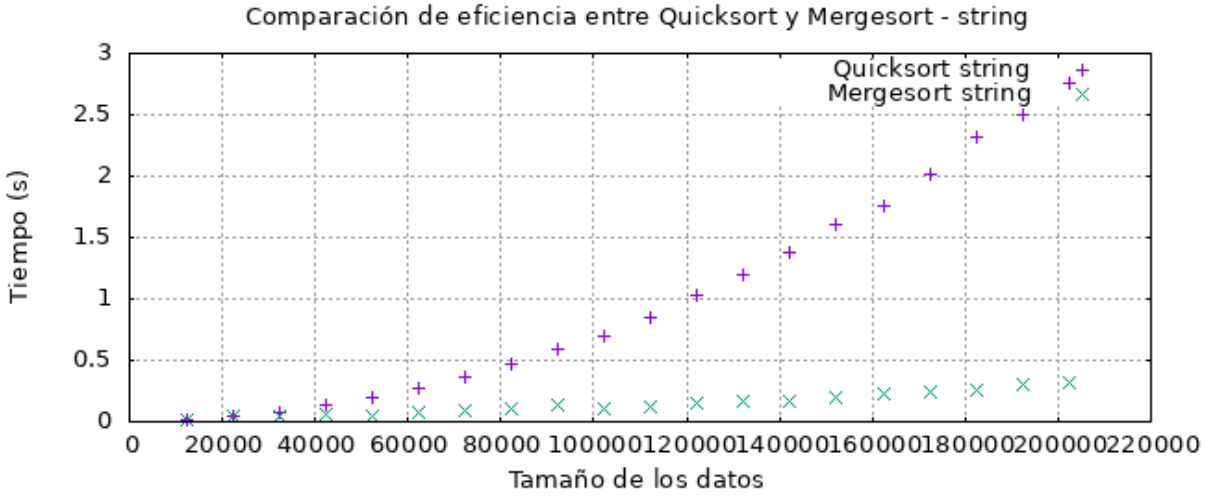
- Ordenación de double:





- Ordenación de string:



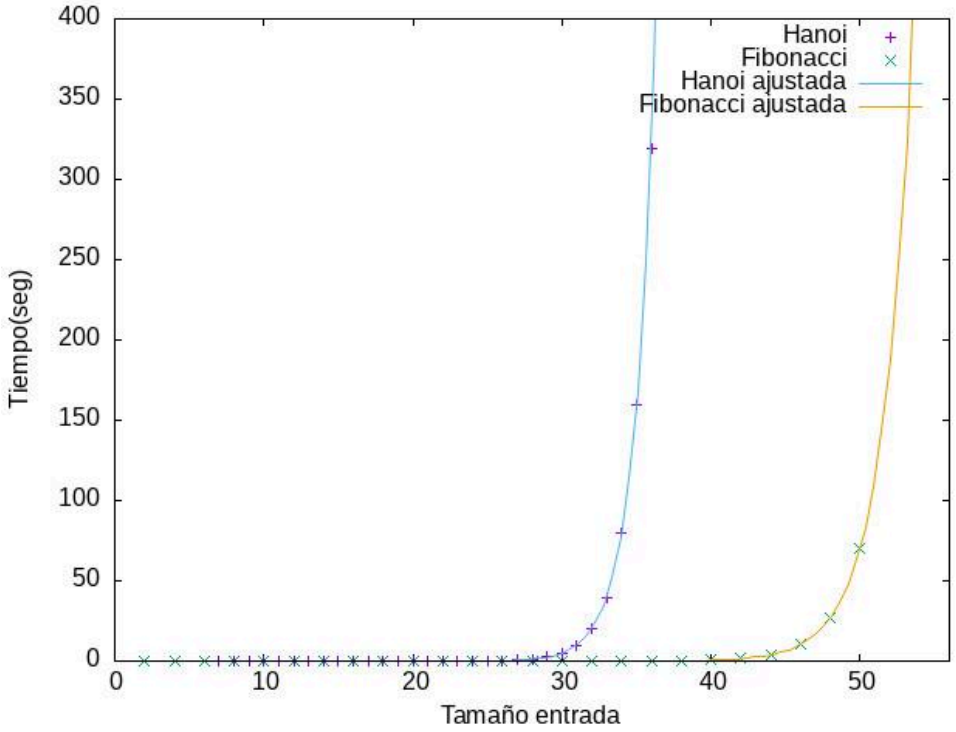


Como podemos observar, en prácticamente todos los tipos de datos, quicksort y mergesort son extremadamente similares. Esto se debe a que se comportan siguiendo el mismo orden de eficiencia, y, como veremos en el siguiente apartado, con una constante muy similar.
Sin embargo, el caso de los strings hace saltar las alarmas, al menos hasta que recordamos la verdadera eficiencia en el caso peor del algoritmo quicksort, que como ya se mencionó es cuadrática (del orden de $(O(n^2))$.

5.3. Comparación eficiencia algoritmos exponenciales

Vamos ahora a comparar la eficiencia de los algoritmos de Hanoi y Fibonacci, ya que ambos son exponenciales. Hay que tener en cuenta que aunque ambos sean del mismo tipo, al tener distinta base (Fibonacci con base $\frac{1+\sqrt{5}}{2}$ y Hanoi con base 2) son de órdenes distintas en eficiencia; la base en las funciones exponenciales importan.

Mediante los datos y las curvas de regresión anteriores obtenemos :



Si bien al principio con tamaños de entrada pequeños, ambos algoritmos tardan casi lo mismo, pero a medida que aumentamos la entrada el algoritmo de Hanoi tarda considerablemente más. Esto es, como antes hemos comentado, porque el algoritmo de Hanoi es una exponencial con base mayor que la del algoritmo de Fibonacci. Así, a mayor n, T_Hanoi(n) es mayor que T_Fibonacci(n), donde n es el tamaño datos, y T_algoritmo es la función que nos da el tiempo de ejecución según la entrada.

Conclusión: entre algoritmos con eficiencia exponencial, siempre es mejor el de menor base, siendo la diferencia entre ambos órdenes considerable a partir de n relativamente pequeño.

5.4. Comparación hardware Hanoi

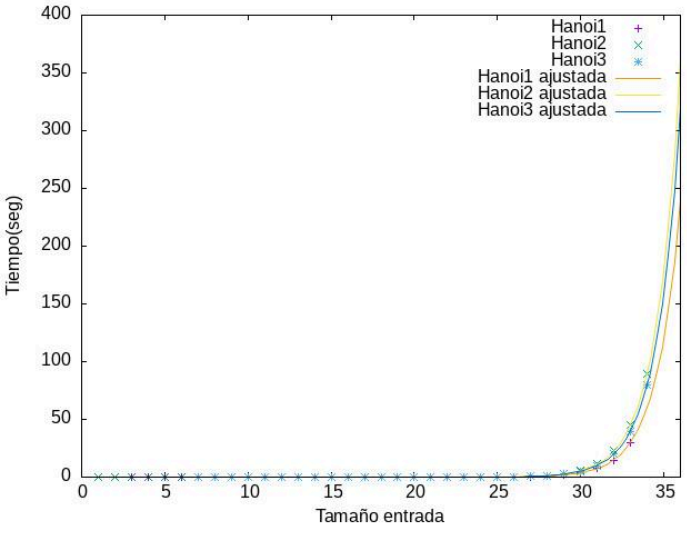
Ahora vamos a ejecutar el mismo programa del algoritmo de Hanoi en tres computadores distintos para hacer un estudio de cómo influye el HW en el tiempo de ejecución.

Los computadores a comparar son :

- Portátil 1 :
  - Modelo: Surface Laptop 4
  - CPU: i7
  - RAM: 16 GB
  - SO: Linux (Ubuntu 22.04 64bits)
- Portátil 2:
  - Modelo: Asus TUF fx505dt
  - Cpu: AMD RYZEN 7 3750h
  - RAM: 16GB
  - SO: Linux (Ubuntu 22.04 64bits)
- Portátil 3:
  - Modelo: Acer-Aspire-A515-45
  - CPU: AMD Ryzen 5 5500U
  - RAM: 16GB
  - SO: Linux (Ubuntu 22.04 64bits)

Recogiendo los datos en cada portátil y graficando obtenemos:

```
FIT:    data read from Hanoi1
        format = z
        x range restricted to [0.00000 : 36.0000]
        #datapoints = 31
        residuals are weighted equally (unit weight)

function used for fitting: f(x)
        f(x)=a0* (2**x)

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a0              = 3.48891e-09      +/- 5.206e-12    (0.1492%)


*******************************************************************
FIT:    data read from Hanoi2
        format = z
        x range restricted to [0.00000 : 36.0000]
        #datapoints = 34
        residuals are weighted equally (unit weight)

function used for fitting: g(x)
        g(x)=a1* (2**x)

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a1              = 5.23654e-09      +/- 9.034e-12    (0.1725%)


*******************************************************************
FIT:    data read from Hanoi3
        format = z
        x range restricted to [0.00000 : 36.0000]
        #datapoints = 28
        residuals are weighted equally (unit weight)

function used for fitting: h(x)
        h(x)=a2* (2**x)

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a2              = 4.64307e-09      +/- 4.035e-13    (0.00869%)
```



Podemos ver que aunque salgan curvas levemente diferentes, las tres siguen el mismo comportamiento exponencial y lo único que cambia son las constantes ocultas (la variable a_1 que multiplica a 2^n).

Conclusión: En distintos HW los algoritmos mantienen el mismo comportamiento y exclusivamente varían las constantes multiplicativas.

6. CONCLUSIONES

Durante el estudio hemos visto corroboradas todas y cada una de las enseñanzas presentadas en clase teórica respecto al análisis de la eficiencia de algoritmos, mas incluso hemos asimilado de manera más profunda las mismas por haber experimentado las decisivas diferencias que en cuanto a tiempo de ejecución de programas suponen. También nos hemos encontrado con resultados inesperados, como la variabilidad de ciertos algoritmos de ordenación con respecto al tipo de dato de entrada y las irregularidades en los tiempos de algunas ejecuciones por la aleatoriedad de los datos, que sabemos deberemos tener en cuenta en el futuro. En definitiva, hemos comprendido que la eficiencia de un algoritmo es uno de los aspectos que con más seriedad hay que tener en cuenta para juzgar su calidad.

En particular, algunos de los conceptos teóricos que se nos han visto manifestados muy claramente en el desarrollo de este estudio han sido:

- La relevancia del estudio del "caso peor" (notación "O grande") a nivel teórico por su similitud con la realidad.
- La corrección de las reglas de cálculo de eficiencia teórica de un algoritmo iterativo: a saber, la regla del máximo, la regla de la suma, y la regla del producto (además de las específicas para el cálculo de la eficiencia teórica de porciones de código con instrucciones de control como bucles, condicionales, y demás). Particularmente, comprobando que los ajustes polinómicos obtenidos en eficiencia híbrida, al considerar tan sólo el término de mayor orden, no perdían apenas corrección (regla del máximo).
- La importancia de la elección de una implementación coherente de un algoritmo teniendo en cuenta las implicaciones teóricas de ciertos diseños, como por ejemplo el riesgo de obtener órdenes de eficiencia exponenciales usando sucesivas llamadas recursivas (como en el algoritmo para calcular términos de la sucesión de Fibonacci).
- Cómo realmente la diferencia de tiempo de ejecución de un mismo algoritmo al variar la plataforma (Hardware o Software) sobre la que se ejecuta difiere tan solo por una constante de proporcionalidad.

En síntesis, consideramos exitosa la investigación práctica que hemos hecho sobre la eficiencia de los algoritmos propuestos, por habernos dotado de perspectivas y herramientas relevantes para el abordaje de problemas reales que se nos presenten en nuestro futuro académico, laboral o recreativo.