

**Grupo C1-G2**

Pedro Bedmar López

Alejandro Escalona García

Joaquín Alejandro España Sánchez

Joaquín García Venegas

**Eficiencia de los algoritmos**

**Índice:**

1. **Planteamiento del problema**
2. **Tablas con eficiencia empírica de algunos algoritmos**
3. **Gráficas resultantes**
4. **Eficiencia híbrida de algunos algoritmos**
5. **Eficiencia empírica y parámetros externos**
6. **Conclusión**
7. **Planteamiento del problema**

En esta práctica se pretende abordar la comparación de eficiencia entre algoritmos. Ya que en la parte teórica de la asignatura se está estudiando eficiencia teórica, en esta parte de prácticas nos centraremos en mayor medida en la empírica.

Para ello, vamos a comparar la eficiencia de una serie de algoritmos implementados en C++ utilizando entradas de diferentes tamaños. Calcularemos los tiempos de ejecución de cada uno de los algoritmos para cada tamaño y también seguiremos los demás pasos descritos en el índice, comprobando que la eficiencia empírica se ajusta a la teórica.

1. **Tablas con eficiencia empírica de algunos algoritmos**

Utilizando la implementación de los algoritmos proporcionada por los profesores, calcularemos el tiempo que tardan en ejecutarse gracias al reloj del sistema. Este reloj es capaz de capturar en una variable el momento temporal actual, y si realizamos una medición al inicio y otra al final de la ejecución del algoritmo y restamos la final menos la inicial, conseguimos la duración. Para transformarla a segundos la dividimos entre la constante CLOCKS\_PER\_SEC.

Hemos elegido los algoritmos ordenación por burbuja, mergesort, Floyd y el de las torres de Hanoi para calcular sus tiempos de ejecución. Para facilitar el trabajo, hemos compilado todos los ficheros .cpp donde se implementan los algoritmos y hemos utilizado la macro de terminal facilitada por los profesores para generar todos los datos de cada tabla con una sola sentencia.

En algunos casos, vamos a realizar varios cálculos de tiempo, para así obtener un tiempo promedio y eliminar posibles valores atípicos en las gráficas. Estos valores pueden producirse porque en la máquina se ejecutan múltiples procesos al mismo tiempo, y quizás en un momento dado se ha colado algún proceso con más prioridad y esto ha provocado una bajada en el rendimiento del algoritmo. En otros casos puede deberse a factores relacionados con la aleatoriedad de los números: hay algoritmos que tardan más o menos según la entropía de los datos, y eso puede afectar al tiempo de ejecución. Promediando la solución reducimos el impacto de estos factores.

**Burbuja -> O(n2)**

Es un sencillo algoritmo de ordenamiento. Funciona revisando cada elemento de la lista que va a ser ordenada con el siguiente, intercambiándolos de posición si están en el orden equivocado. Es necesario revisar varias veces toda la lista hasta que no se necesiten más intercambios, lo cual significa que la lista está ordenada. Este algoritmo obtiene su nombre de la forma con la que suben por la lista los elementos durante los intercambios, como si fueran pequeñas burbujas. También es conocido como el método del intercambio directo. Dado que solo usa comparaciones para operar elementos, se lo considera un algoritmo de comparación, siendo el más sencillo de implementar.

*static* *void* burbuja\_lims(*int* *T*[], *int* *inicial*, *int* *final*)

{

*int* i, j;

*int* aux;

  for (i = inicial; i < final - 1; i++)

    for (j = final - 1; j > i; j--)

      if (T[j] < T[j-1])

  {

    aux = T[j];

    T[j] = T[j-1];

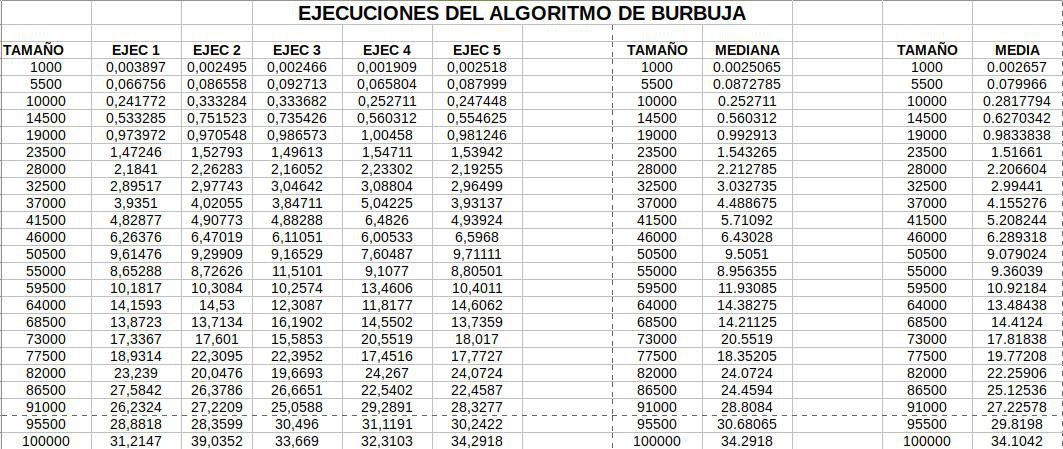
    T[j-1] = aux;

  }

}

Como se observa, al existir dos bucles anidados para llevar a cabo la ejecución del algoritmo, se deduce fácilmente que la eficiencia es **O(n2)**.

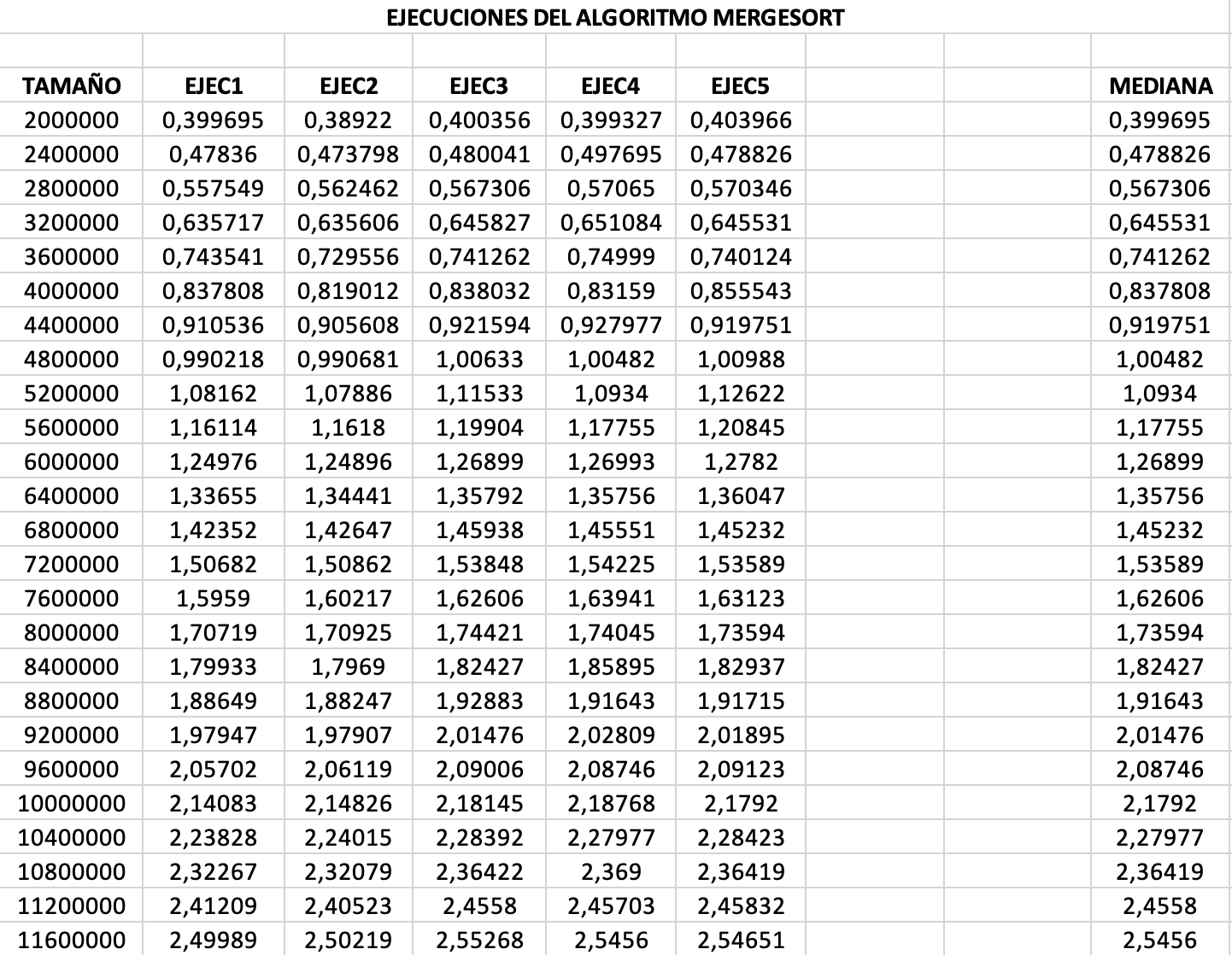
En este ejercicio hemos aplicado la técnica mencionada en el apartado 1 para promediar el resultado. Este promedio lo hemos llevado a cabo mediante la varianza y la media, para analizar cual se ajusta mejor al resultado que queremos obtener.

****

**Mergesort -> O(n\*log(n))**

En este algoritmo de ordenación que aplica la técnica divide y vencerás para mejorar la eficiencia respecto a otros algoritmos de ordenación convencionales. Es un algoritmo recursivo donde el vector se divide en dos partes del mismo tamaño (generalmente), hasta llegar al caso base, que es aquel donde tenemos vectores de tamaño 0 o 1. Dentro del proceso de recursividad, estos subvectores se van recomponiendo de forma ordenada.

Teóricamente es el algoritmo con mejor eficiencia, pero en la práctica quicksort lo supera en muchas ocasiones, ya que las constantes ocultas que multiplican la función son mayores en mergesort. Aquí también aplicamos la mediana a las cinco ejecuciones para conseguir un resultado fiable.

****

|  |  |
| --- | --- |
| **Rendimiento del algoritmo de Floyd** | |
| **Número de nodos** | **Runtime** |
| 100 | 0.009918 |
| 200 | 0.048741 |
| 300 | 0.160568 |
| 400 | 0.325063 |
| 500 | 0.616948 |
| 600 | 1.04604 |
| 700 | 1.6858 |
| 800 | 2.50732 |
| 900 | 3.51525 |
| 1000 | 4.87015 |
| 1100 | 6.58778 |
| 1200 | 8.53515 |
| 1300 | 10.947 |
| 1400 | 13.4565 |
| 1500 | 16.5788 |
| 1600 | 20.1431 |
| 1700 | 24.2614 |
| 1800 | 28.7446 |
| 1900 | 33.7018 |
| 2000 | 39.4295 |
| 2100 | 45.5004 |
| 2200 | 52.4182 |
| 2300 | 60.2006 |
| 2400 | 68.5271 |
| 2500 | 77.4754 |

**Floyd -> O(n3)**

Se trata de un algoritmo de análisis de grafos ponderados para encontrar el camino mínimo entre dos vértices cualesquiera.

El algoritmo representa al grafo con una matriz cuadrada de orden n (número de nodos del grafo). El valor de un elemento, por ejemplo, M[i][j] representa el coste de ir desde el nodo i hasta el j, si no están relacionados el valor será infinito. La matriz tendrá una diagonal con ceros, que sería la distancia de un nodo a él mismo.

*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];

        }

}

Como se observa claramente en el código del algoritmo de Floyd, su eficiencia teórica es **O(n3)** debido a que hay tres bucles anidados.

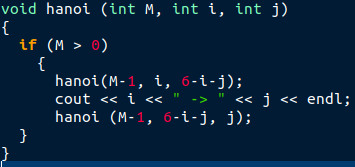
**Torres de Hanoi -> O(2n)**

En este algoritmo el tamaño del problema viene representado por el número de discos con los que se juega. El juego consiste en desplazar todos los discos desde una torre a otra siguiendo las siguientes reglas:

* En cada movimiento solo puede desplazarse un disco y el resto deben estar colocados en las torres.
* Nunca puede haber un disco de mayor tamaño sobre uno de menor tamaño.
* Solo será posible mover el disco que esté arriba en cada torre.

Este algoritmo tiene una eficiencia 0(2^n) porque para mover un disco es necesario realizar n-1 desplazamientos. Por eso, por ejemplo, para resolver un problema con 3 discos sería necesario realizar 2^n-1, cuya eficiencia es O(2^n).

# Este algoritmo tiene una eficiencia 0(2^n) porque para mover un disco es necesario realizar n-1 desplazamientos. Por eso, por ejemplo, para resolver un problema con 3 discos sería necesario realizar 2^n-1, cuya eficiencia es O(2^n).

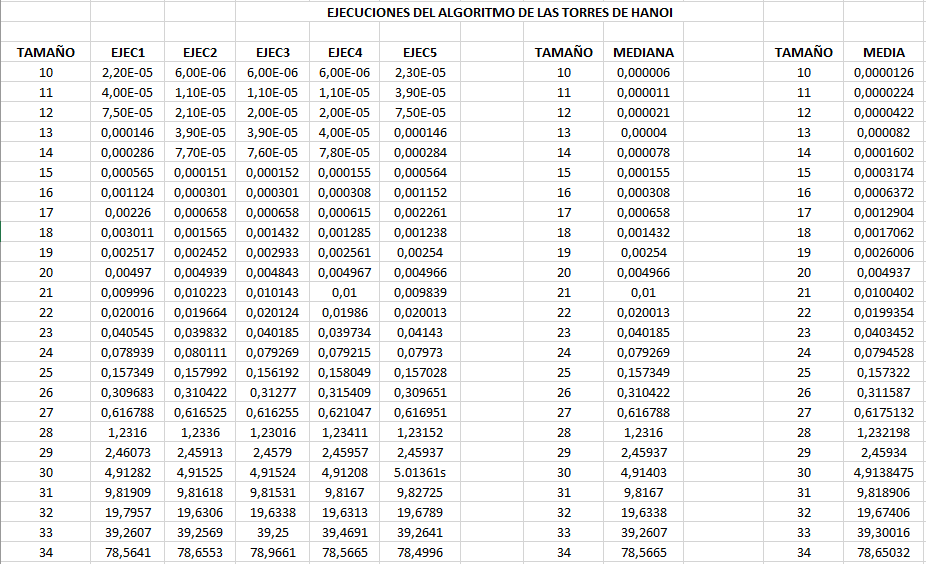
En este algoritmo vemos que la implementación es bastante sencilla, tenemos M que es el número de discos, i es el número de columna en el que están los discos y j es el número de columna al que se van a mover los discos. Tanto i como j pueden tener un valor del conjunto {1, 2, 3}.

Podemos hacer un breve análisis para averiguar qué ecuación de recurrencia está relacionada con este algoritmo, si lo hacemos en relación al número de movimientos (t) en términos del número de discos(n), tenemos:

1. Considerando que con un disco se produce un movimiento tendríamos nuestra primera condición inicial, T(1)=1.
2. Si hacemos el juego con 2 discos, tendríamos un movimiento para pasar el disco menor a una torre, otro para pasar el disco mayor a otra torre, y por último un tercer movimiento para colocar el menor encima del mayor, por lo que T(2)=3. Si lo hacemos para 3 discos tendríamos un total de 8 movimientos, por lo que T(3)=7.
3. Teniendo estas tres condiciones, podemos deducir que T(n) podría tener como solución no recurrente a 2^n - 1.
4. Sabiendo que se producen n-1 movimientos para poder mover un disco inferior a otra torre, la ecuación de recurrencia asociada tiene que venir en términos de T(n-1), teniendo de condición inicial T(1)=1 y que T(2)=3 y T(3)=7, se deduce que T(n)= 2T(n-1)+1.



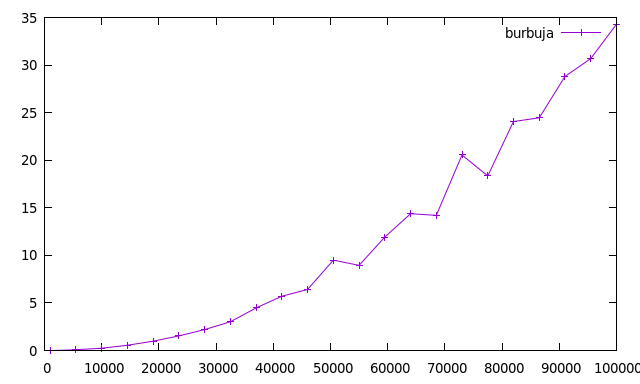
A la hora de generar los datos, para calcular la eficiencia empírica lógicamente hay que eliminar la salida que genera dicha función ya que solo nos interesa generar un fichero con dos columnas, una para la el tamaño del problema y otra para el tiempo de ejecución. Si no eliminamos esa salida el sistema de archivos no respondería al crearse un fichero de salida con tantos MB, ya que se genera un gran número de llamadas recursivas cuando el tamaño del problema aumenta.

 Para hacer una representación lo más exacta posible hemos optado por hacer cinco ejecuciones del mismo algoritmo, de los datos obtenidos de estas cinco ejecuciones obtenemos la media y la mediana para después en el apartado siguiente decidir cuál de las dos es más representativa de los datos, en este caso, como veremos, escogeremos la mediana.

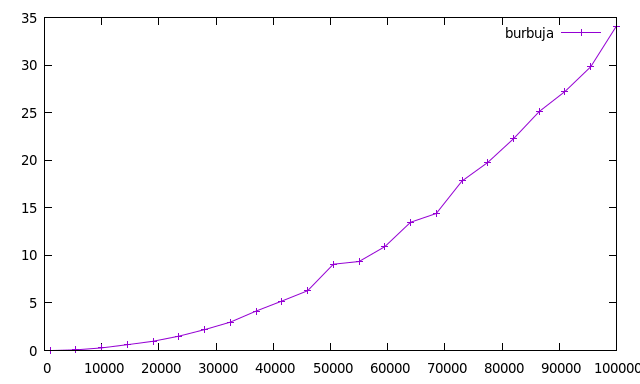
1. **Gráficas resultantes**

**Algoritmo de Burbuja**

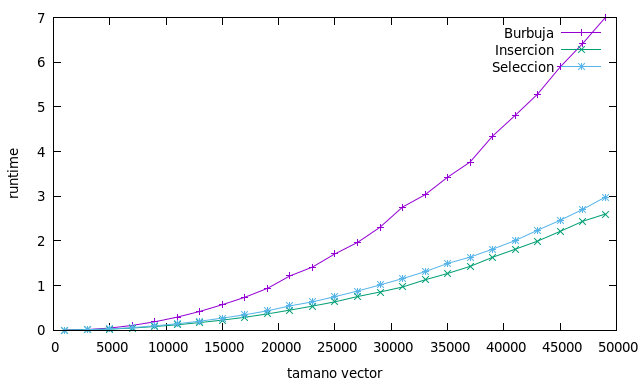
Primeramente, representamos los datos obtenidos con la mediana de las cinco ejecuciones. Comprobamos como aparentan crecer de forma cuadrática, pero observamos que aparecen valores atípicos en exceso.

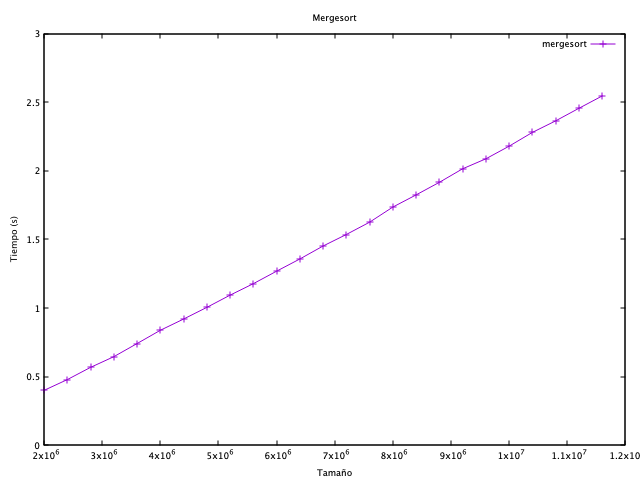


Por tanto, decidimos calcular la media en lugar de la mediana, y obtenemos un resultado mucho más preciso y continuo.



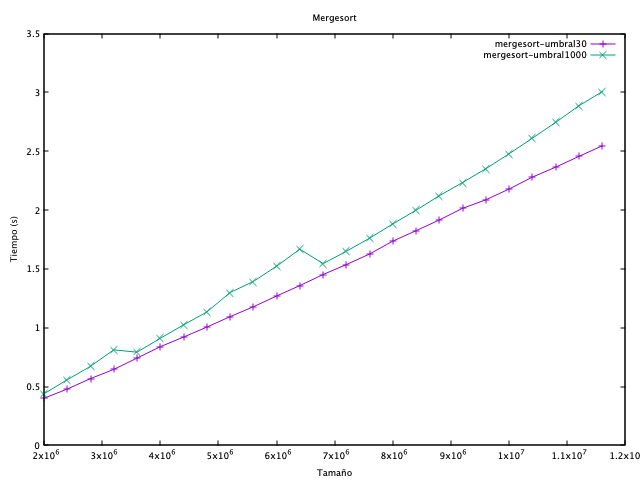
En la siguiente gráfica se puede apreciar la comparación de todos los algoritmos de orden cuadrático, en el que salta a la vista la gran diferencia entre burbuja y los otros dos algoritmos.

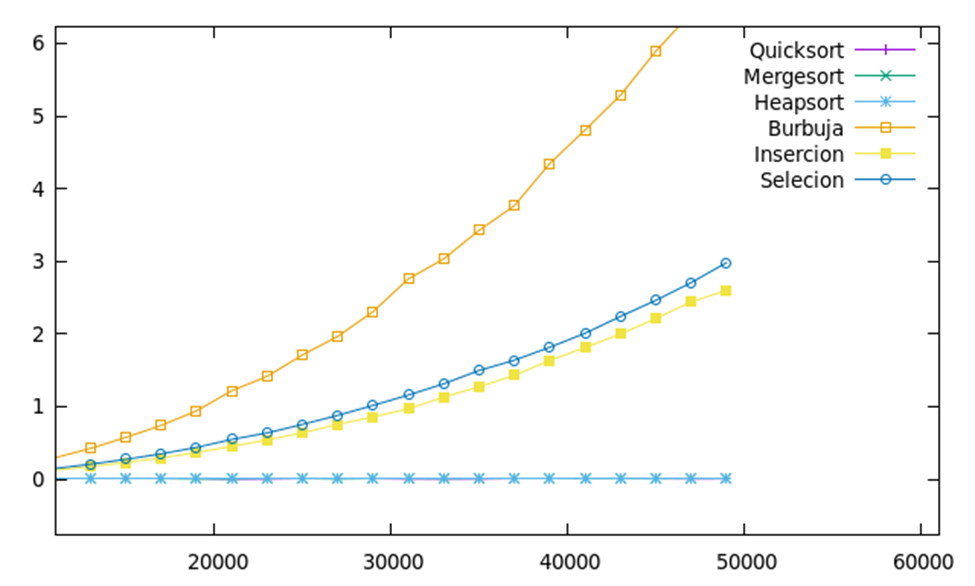
******

**Algoritmo de Mergesort**

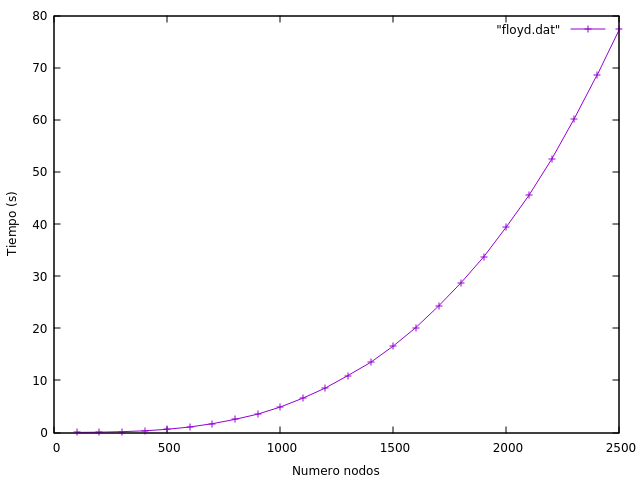
Como podemos observar mergesort es muy rápido, siendo capaz de ordenar vectores de tamaño muy grande en muy poco tiempo.

También es importante comentar que esta gráfica no es la que hemos obtenido directamente. Después de realizar la mediana de los datos, observamos cómo se seguían produciendo saltos en la gráfica, y debido a la serie de precauciones que estábamos tomando, sospechamos que debía haber algún factor que estuviese afectando a los cálculos del que no nos habíamos percatado. Descartamos la opción de que la aleatoriedad de los números estaba afectando, ya que no produce impacto en este algoritmo. Investigando descubrimos en el código de mergesort una constante de tipo entero (const int UMBRAL\_MS=1000), que determina en que momento mergesort deja de ejecutarse recursivamente y pasa a ordenar el subvector por inserción. Y es reduciendo esta constante a un número más bajo (30 en nuestro ejemplo) como eliminamos el problema de los saltos, ya que inserción se activa en tamaños más pequeños y no perjudica el rendimiento de mergesort, que funciona mejor en tamaños grandes. Por tanto, realmente no estamos ejecutando un mergesort puro, sino un mergesort-inserción:

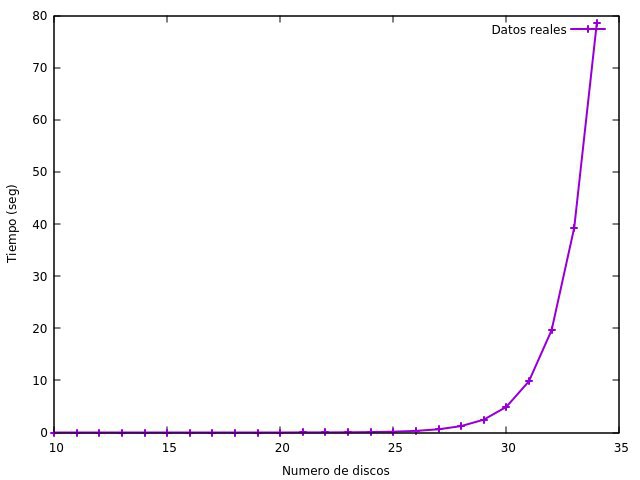


Una vez obtenidas todas las gráficas de los algoritmos de ordenación, vamos a realizar una gráfica comparativa de todos ellos:

Como podemos ver, el algoritmo de burbuja sería el que peor rendimiento ha dado, después estarían inserción y selección. Y por último los algoritmos de Orden n\*log(n).

**Algoritmo de Floyd:**

**Algoritmo de las Torres de Hanoi:**



En este caso para la representación de los datos hemos elegido la mediana ya que es realmente la más representativa de los datos. Por ejemplo, en el siguiente conjunto de números: {1 1 1 1 10} la media nos dice que el valor medio es 2,8, sin embargo, si hacemos la mediana obtenemos que el valor que deja al 50% de los datos a sus lados es el 1, el cual representa mejor al conjunto de datos obtenido, ya que el 10 ha sido un caso aislado.

Tras la representación del conjunto de datos que tenemos en el archivo generado por mimacro.sh llamado hanoi.dat, podemos observar como la gráfica describe una curva exponencial que se corresponde efectivamente con su eficiencia teórica O(2^n).

1. **Eficiencia híbrida de algunos algoritmos**

En este ejercicio lo que nos piden es calcular la eficiencia híbrida de cada algoritmo. Para ello, debemos seguir los siguientes pasos:

1. Definir la función que queremos ajustar a nuestros datos. Según la eficiencia teórica de    nuestro algoritmo definimos la ecuación general de la función que queremos ajustar.
2. Indicarle a gnuplot que haga regresión, es decir, que estime la relación entre nuestros datos y la función que hemos definido.
3. Dibujar nuestros datos y la función definida para poder ver cómo se ajusta esta función a los datos.

**-> Ajustes que se corresponden con su eficiencia teórica:**

• ***Burbuja ajustada a f(x)=a\*x2+b\*x+c:***

◦ gnuplot> f(x) = **a**\*x\*x+**b**\*x+**c**

◦ gnuplot> fit f(x) 'salida\_burbuja.dat' via a,b,c. Salida:

**Final set of parameters Asymptotic Standard Error**

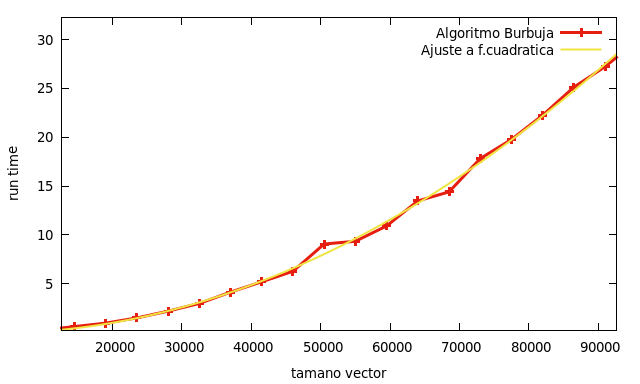
**======================= ==========================**

**a** = 3.51929e-09 +/- 2.041e-10 (5.8%)

**b** = -1.78002e-05 +/- 2.195e-05 (123.3%)

**c** = -0.0333484 +/- 0.5182 (1554%)

◦ gnuplot> plot f(x) 'salida\_burbuja.dat' title 'Curva ajustada'



* ***Mergesort ajustado a f(x)=a\*x\*log2(x)+b***

◦ gnuplot> f(x) = **a**\*x\*(log(x)/log(2))+**b**

◦ gnuplot> fit f(x) 'salida\_mergesort.dat' via a,b. Salida:

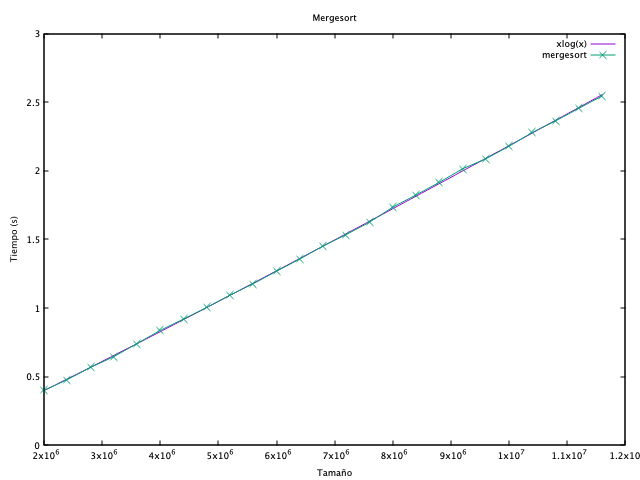
**Final set of parameters       Asymptotic Standard Error**

**======================= ==========================**

**a** = 9.35851e-09 +/- 1.973e-11 (0.2108%)

**b** = 0.0072172 +/- 0.003356 (46.5%)

◦ gnuplot> plot f(x) title “xlog(x)”, 'salida\_mergesort.dat' title “mergesort”



• ***Floyd ajustada a f(x)=ax3+bx2+cx+d***

◦ gnuplot> f(x) = **a**\*x\*x\*x+**b**\*x\*x+**c**\*x+**d**

◦ gnuplot> fit f(x) 'floyd.dat' via a,b,c,d

**Final set of parameters Asymptotic Standard Error**

**======================= ==========================**

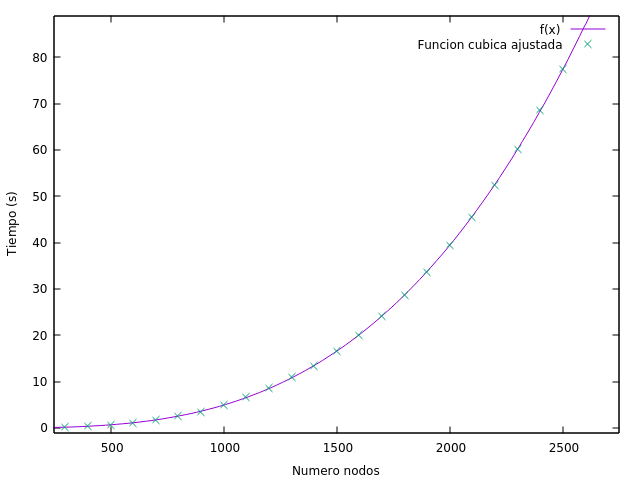
**a** = 5.14211e-09 +/- 6.163e-11 (1.199%)

**b** = -6.73147e-07 +/- 2.435e-07 (36.17%)

**c** = 0.000544829 +/- 0.0002753 (50.52%)

**d** = -0.0943319 +/- 0.08428 (89.35%)

◦ gnuplot> plot f(x), 'floyd.dat' title 'Funcion cubica ajustada'



* ***Hanoi ajustada a f(x)=a\*b^x***

◦ gnuplot> f(x)=a\*b\*\*x

◦ gnuplot> fit f(x) 'hanoi.dat' via a,b

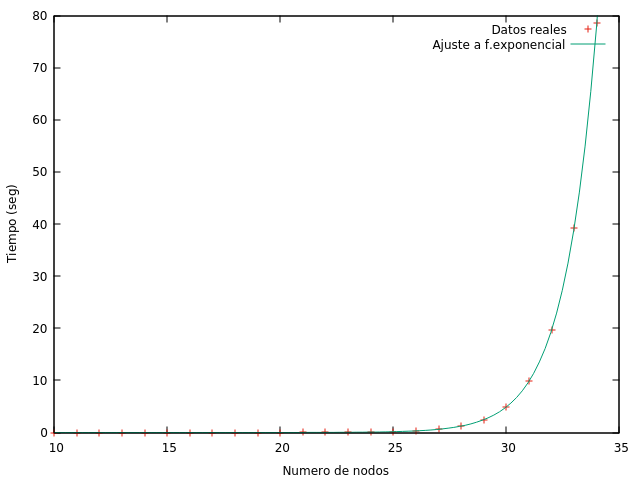
**Final set of parameters Asymptotic Standard Error**

**================== ======================**

a = 6.3623e-09 +/- 3.945e-10 (6.2%)

b = 1.98379 +/- 0.003489 (0.1759%)

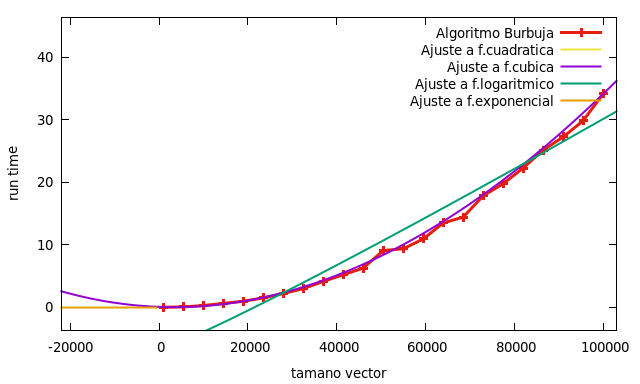
gnuplot> plot f(x), 'hanoi.dat' via a, b



El ajuste realizado se corresponde con los datos generados.

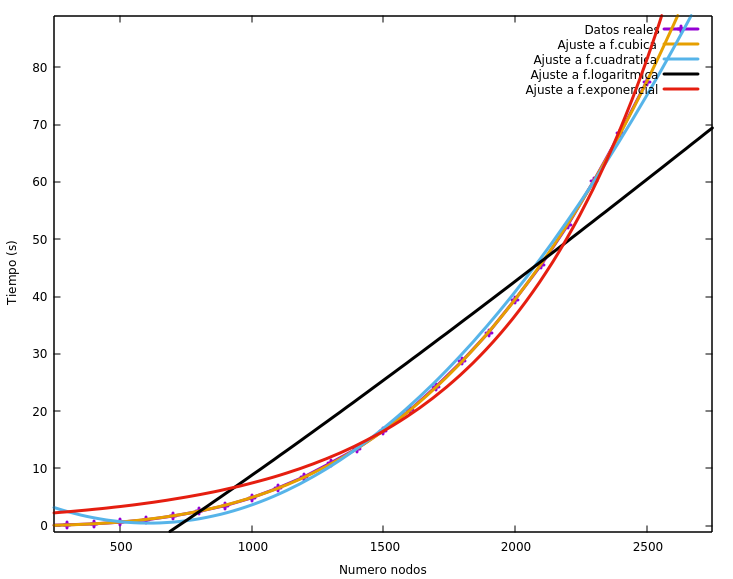
**-> Ahora vamos a realizar una comparación de algunos algoritmos con diferentes ajustes para compararlos con el original:**

• ***Burbuja ajustada al resto de funciones:***



Observamos que la que mejor se ajusta es la función cuadrática, produciendo las demás un peor ajuste.

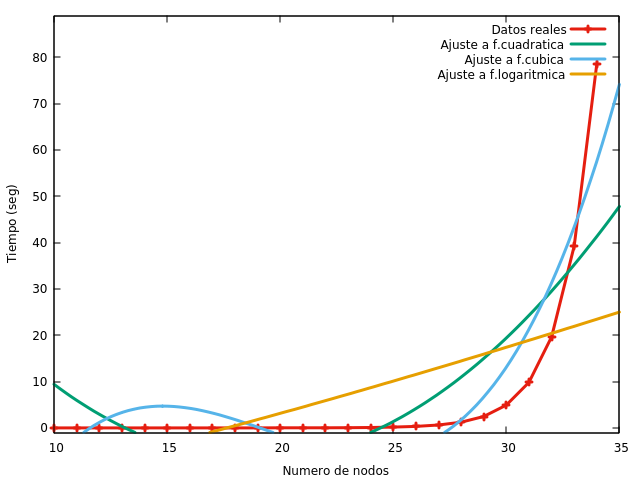
* ***Floyd ajustada al resto de funciones:***



Como se observa en la imagen, la función que mejor ajusta a los datos reales es la función cúbica, ya que es la que coincide perfectamente con los datos reales. Las demás funciones no ajustan muy bien, la función exponencial y la cuadrática, al final se acercan un poco a los datos reales, pero al inicio no coinciden.

**NOTA:** La gráfica de los datos reales no se aprecia bien al estar solapada por la función cúbica.

# ***Hanoi ajustada al resto de funciones:***



Como se observa en la gráfica, realmente no hay ninguna función que se ajuste bien a los datos reales de nuestro algoritmo, excepto la función exponencial (la hemos omitido porque arriba ya hemos hecho su ajuste).

1. **Eficiencia empírica y parámetros externos**

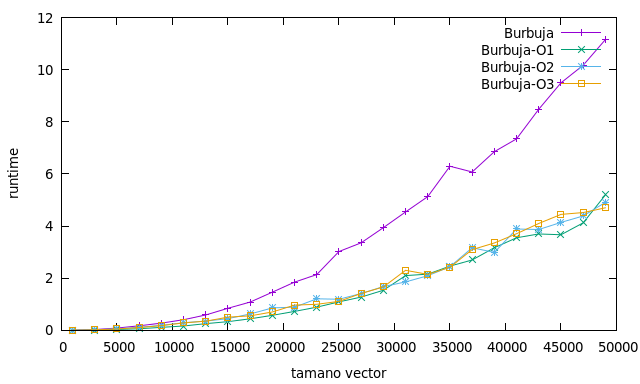
En este ejercicio se pide que hagamos un estudio de la eficiencia en función de otros parámetros externos, ya que la eficiencia no sólo depende de cómo sea el algoritmo, sino que hay otros factores que pueden influenciar en la ejecución de este tales como las opciones de compilación usadas para crear el ejecutable, el sistema operativo sobre el que se ejecuta, el ordenador sobre el que se ejecuta, etc. En este caso, se va a realizar un estudio mediante las diferentes opciones de optimización en la compilación. Los flags son los siguientes, y cada uno de ellos activa y desactiva unos subflags de más bajo nivel:

1. -O0: El código no se optimizará.
2. -O1: El compilador trata de reducir el tamaño del código y el tiempo de ejecución sin efectuar optimizaciones que conlleven una gran cantidad de tiempo de compilación.
3. -O2: Aplica casi todas las optimizaciones posibles que no implican un sacrificio entre tiempo de ejecución y memoria. Comparado con -O1 aumenta el tiempo de compilación y el rendimiento.
4. -O3: Este nivel de optimización es el más alto. En algunas ocasiones puede provocar que la ejecución sea más lenta, debido a un flag al que llama (-ftree-vectorize) que intenta vectorizar los bucles dando lugar a este enlentecimiento. Suele acarrear un código más largo.

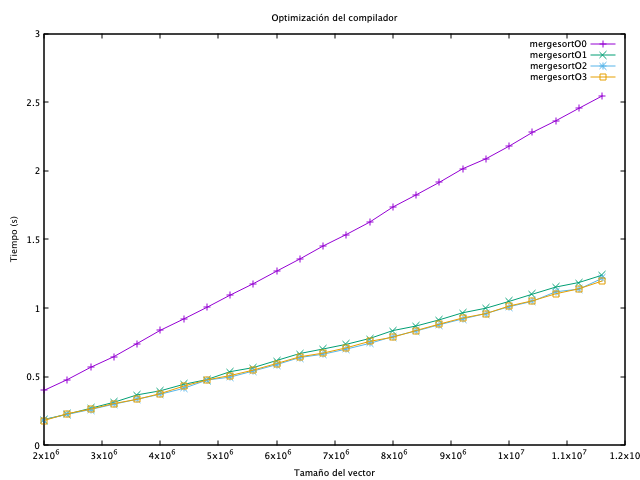
Ahora procedemos a mostrar una gráfica para cada algoritmo, para ello he tenido que compilarlos previamente con g++ y la opción de compilación deseada, muestro este ejemplo, para el resto de algoritmos se procede de igual forma:

1. g++ -O0 -o burbuja burbuja.cpp
2. g++ -O1 -o burbuja burbuja.cpp
3. g++ -O2 -o burbuja burbuja.cpp
4. g++ -O3 -o burbuja burbuja.cpp

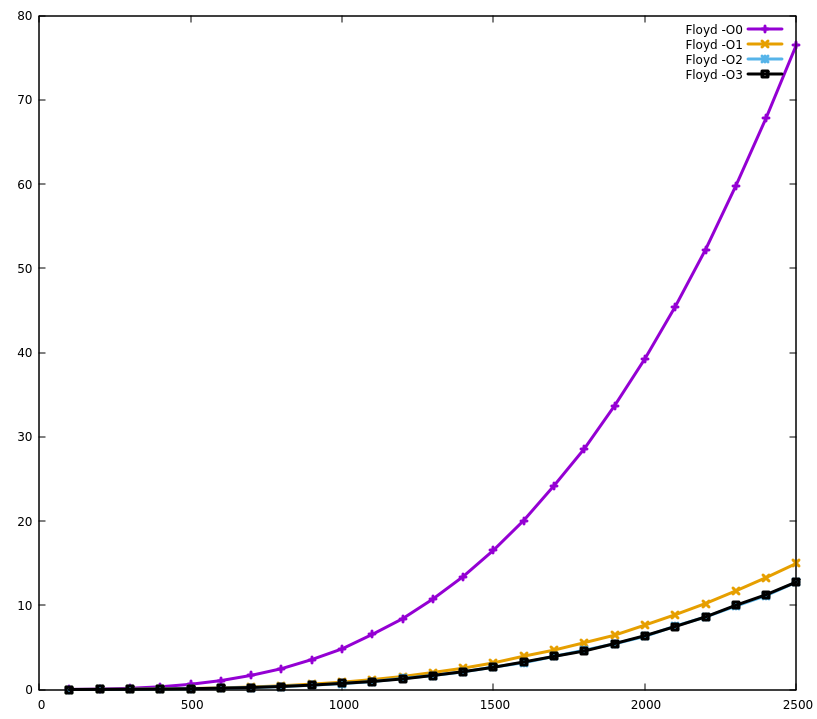
* ***Algoritmo de ordenación por Burbuja:***



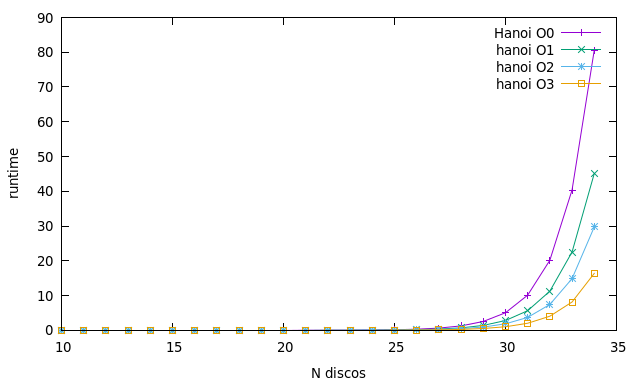
* ***Algoritmo mergesort:***



* ***Algoritmo de Floyd:***



**NOTA:** La optimización con –O2 no aprecia ve apenas porque está superpuesta con la optimización –O3, es decir, ambas optimizaciones son prácticamente idénticas para este algoritmo.

* ***Algoritmo de las Torres de Hanoi:***

En esta gráfica vemos que el tiempo que se ha tardado en la ejecución va reduciéndose conforme se va aumentando el nivel de optimización, en este algoritmo la opción de optimización -O3 produce mejor rendimiento que la opción -O2.

1. **Conclusión:**

En la práctica hemos demostrado como la eficiencia empírica se ajusta en gran medida a la eficiencia teórica. También hemos observado como factores del propio sistema donde se ejecutan los algoritmos afectan a esta eficiencia empírica, en lo que llamamos las constantes ocultas, y que aplicando optimizaciones a la hora de compilar se pueden reducir ampliamente los tiempos de ejecución.