UNIVERSIDAD DE GRANADA
E.T.S. DE INGENIERÍAS INFORMÁTICA Y DE TELECOMUNICACIÓN

Asignatura: Algorítmica

Grupo nº: 4

Integrantes: Raúl Rodríguez Pérez, Francisco Javier Gallardo Molina, Inés Nieto Sánchez, Antonio Lorenzo Gavilán Chacón

Grupo: C1

Resolución Práctica 1: Análisis de Eficiencia de Algoritmos

Curso 2019-2020

Grado en Ingeniería Informática

Índice:

1. Introducción

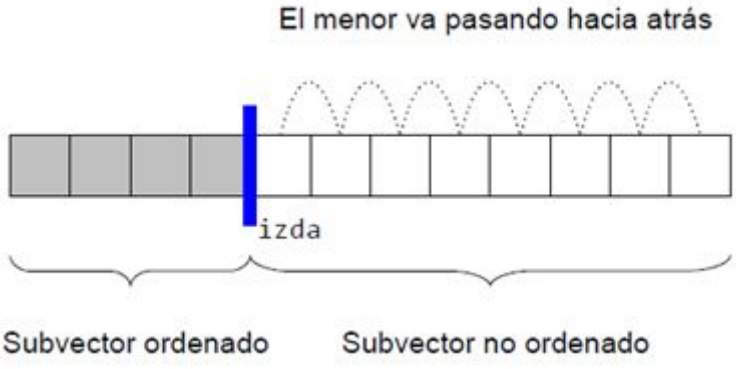
En este trabajo vamos a exponer las diferentes conclusiones que hemos sacado al realizar los análisis de las eficiencias de los distintos algoritmos propuestos en la práctica 1 de la asignatura de AlgorÍtmica. En cada uno de ellos procederemos a explicar brevemente la implementación y el uso que se la da a cada algoritmo, es decir, si se utiliza para resolver problemas concretos, para ordenación, búsqueda… Tras la breve presentación, reflejaremos a través de diversas gráficas, el estudio acerca de la eficiencia de los distintos algoritmos, donde mostraremos su eficiencia empírica (utilizando la librería 'ctime' para obtener el tiempo de ejecución, y el programa 'gnuplot' para plasmar los tiempos en gráficas), híbrida, diferentes ajustes(no solo los que equivalen a su eficiencia teórica) y además realizaremos el trabajo de comparar las diferencias presentes desde el punto de vista de la eficiencia empírica, según la opción de compilación. La diferentes opciones que vamos a emplear son:

- **-O0:** En este nivel se desconecta por completo la optimización y es el predeterminado si no se especifica ningún nivel de -O. El código no se optimizará.
- **-O1:** Es el nivel de optimización más básico, el compilador intentará producir un código rápido y pequeño sin tomar mucho tiempo de compilación.
- **-O2:** Un paso por delante del anterior nivel, es el nivel que se recomienda para optimizar a no ser que el sistema tenga necesidades especiales. Este activará algunas opciones añadidas a las que se activan en el nivel anterior, el compilador tratará de aumentar el rendimiento del código sin comprometer el tamaño y sin tomar mucho más tiempo de compilación.
- **-O3:** Este es el nivel más alto de optimización posible, activa optimizaciones que son caras en términos de tiempos de compilación y uso de memoria. El uso de este nivel no garantiza una forma de mejorar el rendimiento y, de hecho, en muchos casos puede ralentizar el sistema debido al uso de binarios de gran tamaño y mucho uso de la memoria.

2. Algoritmo de burbuja:

Introducción sobre el algoritmo

El algoritmo de burbuja o en inglés bubble sort, es un algoritmo de ordenación que se basa en dividir una lista en dos. Desde el final y hacia atrás, se van comparando elementos dos a dos y se deja a la izquierda el más pequeño (intercambiandolos) dejando en la sub-lista izquierda componentes ordenadas..



El cálculo de la eficiencia teórica está desarrollado en el guión de prácticas por lo que nos limitaremos a deducir las principales conclusiones. El algoritmo Burbuja tiene una complejidad cuadrática **O(n²)**. Esto es así porque forzosamente se ejecutan siempre los dos bucles for, mientras que las asignaciones del interior del segundo bucle no afectan al cálculo ya que son de orden constante. Según este código la eficiencia también es cuadrática en los casos promedio y mejor, por el motivo comentado anteriormente. Sin embargo este algoritmo es susceptible de realizar diversas mejoras. Entre ellas cabe destacar la posibilidad de introducir una condición de parada cuando se detecta que en un recorrido completo del vector no se realiza ningún intercambio, lo que significa que ya está ordenado. Al incorporar la condición la eficiencia de este algoritmo en el **peor de los casos** sigue siendo cuadrático, pero en el mejor caso, si está completamente ordenado la eficiencia viene dada por el número de pasos del bucle for interno:

Número de pasos = n-1 siendo n el número de elementos del vector.

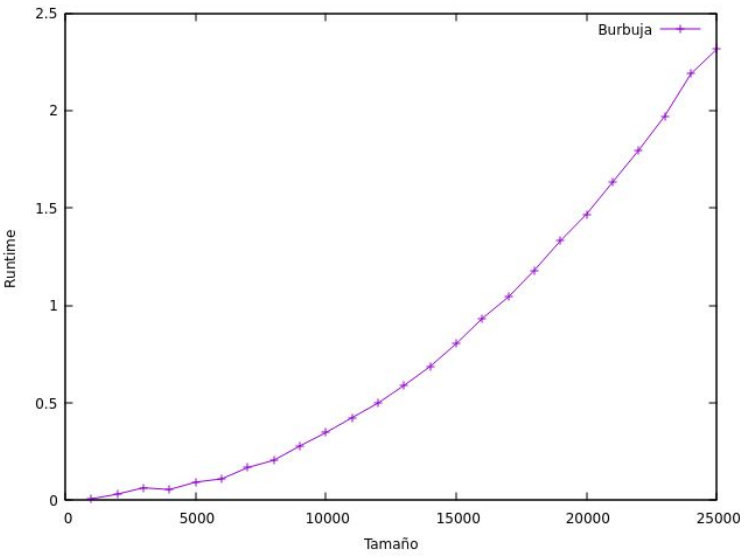
Por lo tanto la eficiencia en el mejor caso es lineal **O (n)**.

Es en conclusión sencillo de entender y de fácil implementación. Tiene un requerimiento mínimo de memoria ya que solo se necesita una variable adicional para realizar los intercambios (variable “intercambia”). Por otro lado, en comparación a otros algoritmos de ordenación es muy lento ya que realiza numerosas comparaciones e intercambios debido a que tiene que recorrer el vector cambiando de uno en uno los elementos que están contiguos.

Gráfica de la eficiencia empírica

Para el estudio de la eficiencia empírica del algoritmo de burbuja hemos usado el código propuesto por el profesor, ejecutando 25 iteraciones, partiendo de un tamaño del vector de 1000 elementos enteros hasta llegar a los 25000 elementos sumándose en cada una de las iteraciones 1000 elementos. Tras esto la gráfica resultante es la siguiente:

Siendo el eje Y el tiempo de ejecución (en segundos) podemos ver que éste oscila entre 0 y 2.3 segundos. También destacar que en el eje X se encuentra el número de elementos del vector. Observando la gráfica, también resaltamos que podemos deducir a partir de la forma convexa de la gráfica que se trata de una función cuadrática



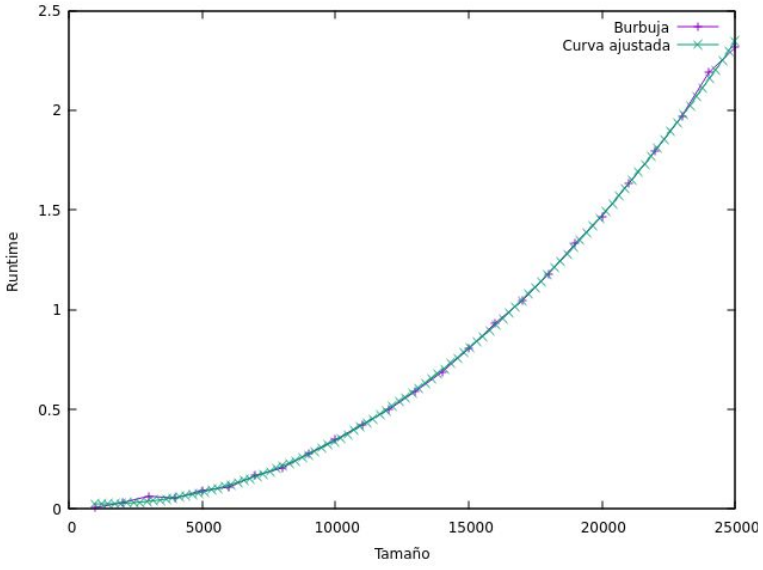
Gráficas y constantes eficiencia híbrida

Para describir completamente la expresión general dada por el cálculo teórico debemos averiguar el valor de las constantes que aparecen en dicha expresión. La forma de averiguar dichos valores es ajustar la función a un conjunto de puntos, siendo nuestra función $f(x) = a*x^2+b*x+c$ y el conjunto de puntos los resultados obtenidos del análisis empírico anterior.

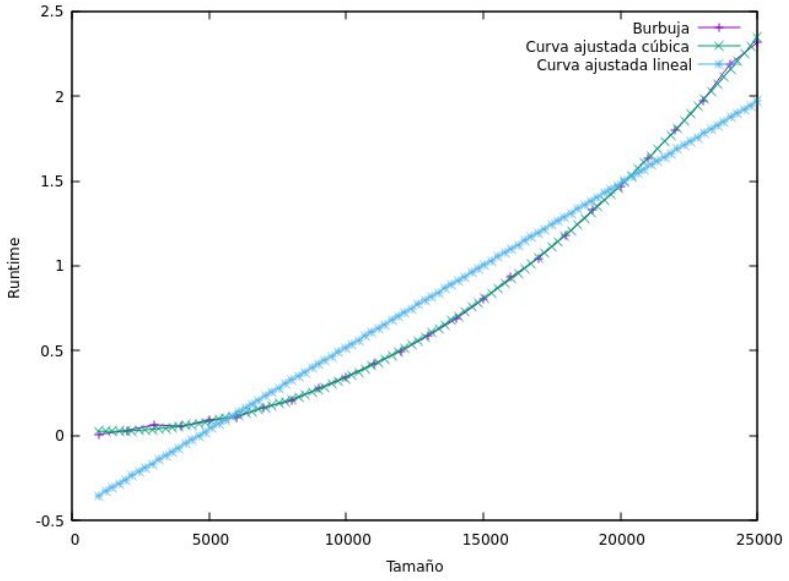
Tras ajustar la gráfica siguiendo los pasos que nos brinda el pdf de la práctica los valores de las constantes son:

- a = 4.09109e-09
- b = -9.74779e-06
- c = 0.0318564

Al obtener los valores ya podemos dibujar la gráfica para comparar el ajuste:

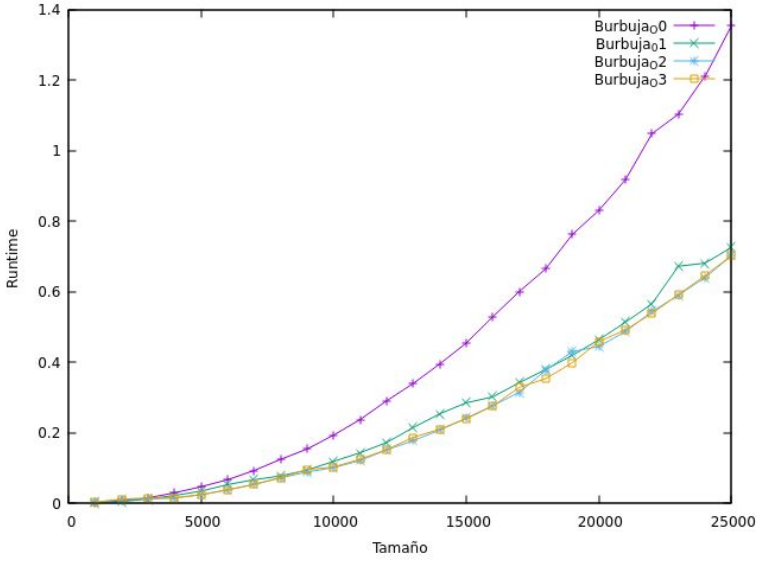


Se observa que el ajuste resultante es muy bueno y podemos concluir que la forma de nuestra gráfica era cuadrática. Para más investigación hemos realizado otro ajustes (lineal y cúbico) para observar cómo se ajustan el conjunto de resultados obtenidos mediante el análisis empírico a otra función. El resultado ha sido el siguiente:



Como era de esperar, si observamos el ajuste lineal, los puntos obtenidos mediante el análisis empírico no se ajustan correctamente con la función puesto que dicha función al ser lineal tiene una forma recta sin desviaciones, mientras que la gráfica resultante de nuestro estudio tenía una forma convexa. Por otro lado podemos observar como el ajuste cúbico sí acopla bastante bien a nuestra función debido a la tendencia convexa que tienen ambas funciones.

Eficiencia con diferentes niveles de optimización



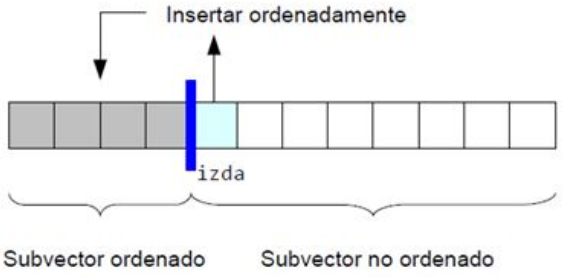
Podemos apreciar en la gráfica que existe una importante diferencia respecto a utilizar o no optimización en el código, siendo el caso en el que empleamos algún tipo de optimización mucho más eficiente que en el caso en el que no. Además, también destacar que a mayor grado de optimización mayor eficiencia tiene la ejecución del algoritmo.

3. Algoritmo de inserción:

Introducción del algoritmo

Al igual que el algoritmo de selección hay tener en cuenta que el vector se debe dividir en dos subvectores, tal que el de la izquierda contiene componentes ordenados y el de la derecha estan desordenados. En cada ciclo se coge el primer elemento del subvector desordenado (izda en la gráfica) y lo insertamos de forma ordenada en el subvector de la izquierda (el ordenado), aumentando así en uno el número de elementos del subvector ordenado. El algoritmo finaliza cuando no quedan elementos en subvector desordenado

Nota. La componente de la posición izquierda (primer elemento del sub-vector desordenado) será reemplazada por la anterior (después de desplazar). El procedimiento se asemeja a la forma en que muchas personas ordenan las cartas de una baraja que se le acaban de dar en una mano.




```
76    char a_desplazar;
77    int i;
78
79    for (int izda = 1; izda < total_utilizados; izda++){
80       a_desplazar = vector[izda];
81
82       for (i = izda; i > 0 && a_desplazar < vector[i-1]; i--)
83          vector[i] = vector[i-1];
84
85       vector[i] = a_desplazar;
86    }
```

La asignaciones de la líneas marcadas como 80 y 85 se ejecutarán un número fijo de veces, puesto que están dentro de un bucle for controlado por el valor de la talla, n. El número de veces que itera el bucle es exactamente n-1 y, por tanto, la líneas 80 y 85 contribuyen al coste del algoritmo (en cualquier caso) con **2(n-1)** asignaciones.

El número de veces que se ejecuta el bucle interior (marcado como línea 2) depende de la condición “a_desplazar < v[i-1]”, de la cual dependerán los posibles casos del algoritmo (mejor y peor). En el mejor de los casos, no se entrará ninguna vez en el bucle, con lo que la línea 82 sólo contribuirá al coste con una comparación y la línea 83 no se llegará a ejecutar.

Por tanto, el coste en el mejor caso del algoritmo sería **O (n)**:

Asignación = 2(n-1) * ta

Comparación = (n-1) * tc

En el peor de los casos, el bucle interno finaliza tras hacer todas las comparaciones y asignaciones posibles (sólo cuando el vector está inversamente ordenado). Si se analiza para cada valor de i el número máximo de veces que se puede entrar en el bucle de la línea 83 se obtiene que:

i= 1, entra 1 vez
i= 2, entra 2 veces
...
i= n-1, entra n-1 veces

Por tanto, el número total de veces que se entra el bucle para todos los valores de i será:

$$\sum_{j=1}^{n-1} j = (n-1) * \frac{(n-1)+1}{2} = \frac{(n-1)*n}{2}$$

Entonces el coste de asignaciones y comparaciones del algoritmo será:

$$Asig = \left[\frac{(n-1)*n}{2} + 2*(n-1)\right] * ta = \frac{n^2+3*n}{2} - 2$$

$$Comp = \left[\frac{(n-1)*n}{2} + (n-1)\right] * ta = \frac{n^2+n}{2} - 1$$

En este caso, el coste del algoritmo de inserción presenta una dependencia cuadrática con n, **O (n2)**. Se puede demostrar que el coste en el **caso medio** de este algoritmo es también cuadrático, es decir, se encuentra más cerca del peor caso que del mejor. El algoritmo se aprovecha del posible orden parcial que pueda existir entre los elementos. Es decir, si un elemento está "cerca" de su posición definitiva, el algoritmo hace menos intercambios. El caso extremo es que el vector esté totalmente ordenado... en cuyo caso, el algoritmo nunca entra en el bucle interior y por eso su complejidad en este caso el mejor es **O (n)**. En este caso, la cota inferior de la complejidad temporal baja hasta una función lineal.

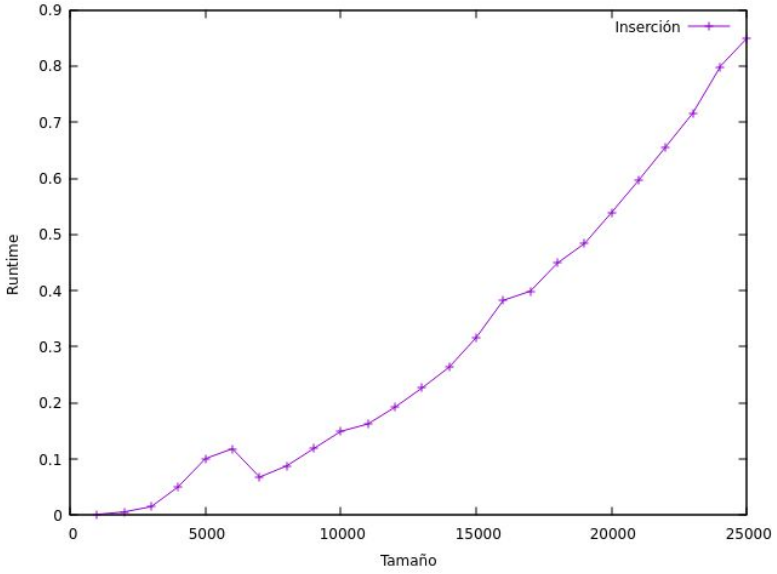
Como **conclusión** se ve en el código que es un algoritmo sencillo de entender y de codificar. Tiene requerimientos mínimos de memoria ya que solo utiliza una variable adicional para realizar los intercambios (variable “a_desplazar”). Además, Si el vector está "casi ordenado", el algoritmo se ejecuta mucho más rápidamente ya que en este caso la complejidad es de orden **O (n)**. Es algo lento si el vector está muy desordenado y realiza numerosas comparaciones, que se ejecutan en todos los ciclos, como por ejemplo con las instrucciones “a_desplazar < v[i-1]”. Como he comentado es un algoritmo lento, pero puede ser de utilidad para vectores que están ordenados o semiordenados, porque en ese caso realiza muy pocos desplazamientos.

Es por la propiedad anterior que este algoritmo, a pesar de no ser el más rápido para entradas grandes, suele usarse de la siguiente manera: Se semi ordena la entrada con algún otro algoritmo más rápido y más adecuado para entradas grandes. Luego, cuando

tenemos la entrada "casi ordenada" usamos este algoritmo. La velocidad de ejecución será muy buena por dos razones: su tiempo de ejecución tiende a **O(n)** con entradas "casi ordenadas" (lo cual es un tiempo excelente), y la simpleza de su implementación hará que se ejecute más rápido que otros algoritmos más complejos.

Gráfica de la eficiencia empírica

En este caso, para el estudio empírico del algoritmo de inserción hemos seguido el mismo procedimiento explicado anteriormente con el algoritmo de burbuja (con un tamaño inicial de 1000 hasta 25000 de 1000 en 1000). Obteniendo de esta forma la siguiente gráfica:



El eje Y se muestra el tiempo de ejecución representado en segundos que abarca los valores entre 0 y 0.8. Por otro lado el eje X representa el tamaño del vector. En la gráfica resultante se observa que tiene tendencia lineal a pesar de presentar pequeños “picos”

Gráficas y constantes eficiencia híbrida

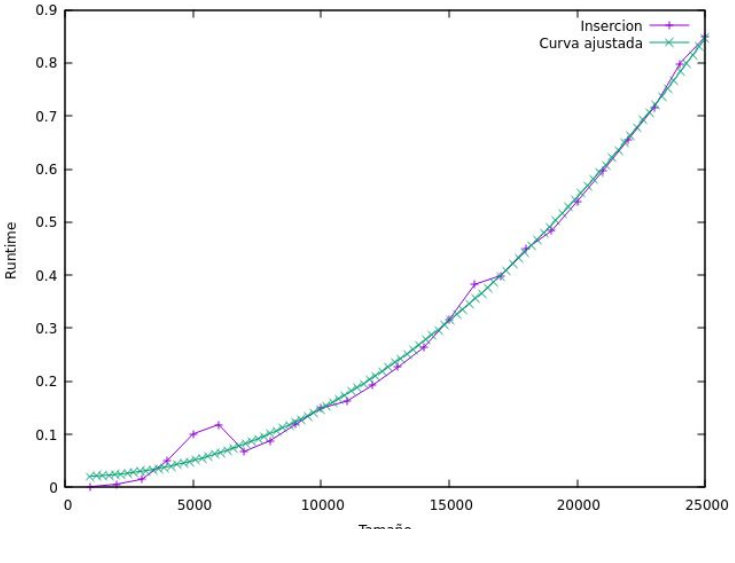
Para calcular la eficiencia híbrida en primer lugar vamos a realizar un ajuste para obtener las constantes de la función $f(x) = a*x^2+b*x+c$. Después de realizar el ajuste los valores de las constantes son:

·a = 1.35233e-09

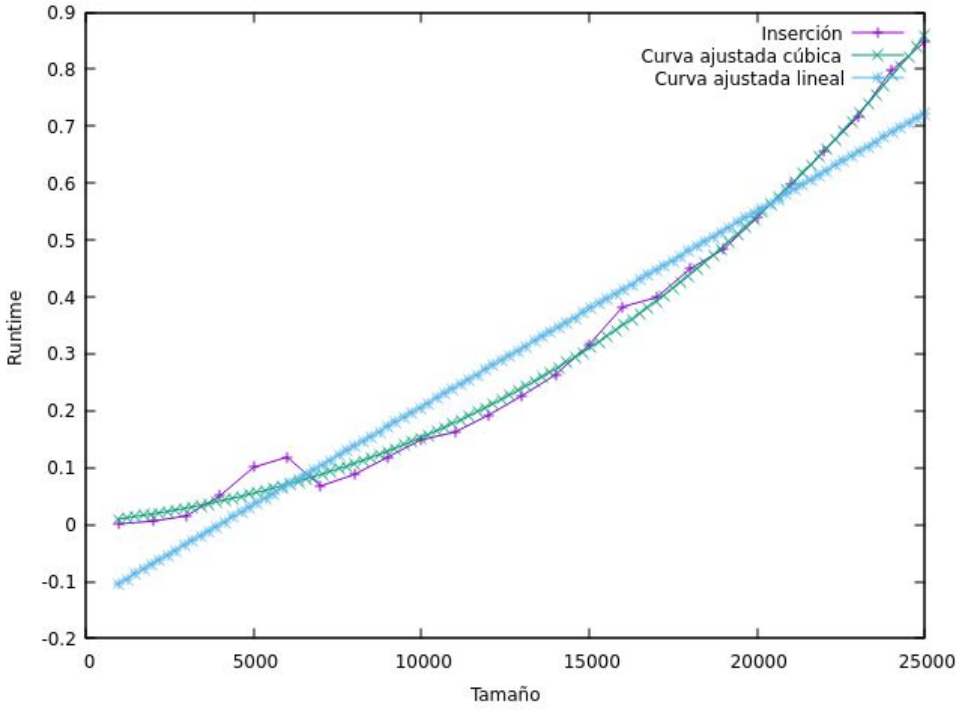
·b = -7.43938-07

·c = 0.0206693

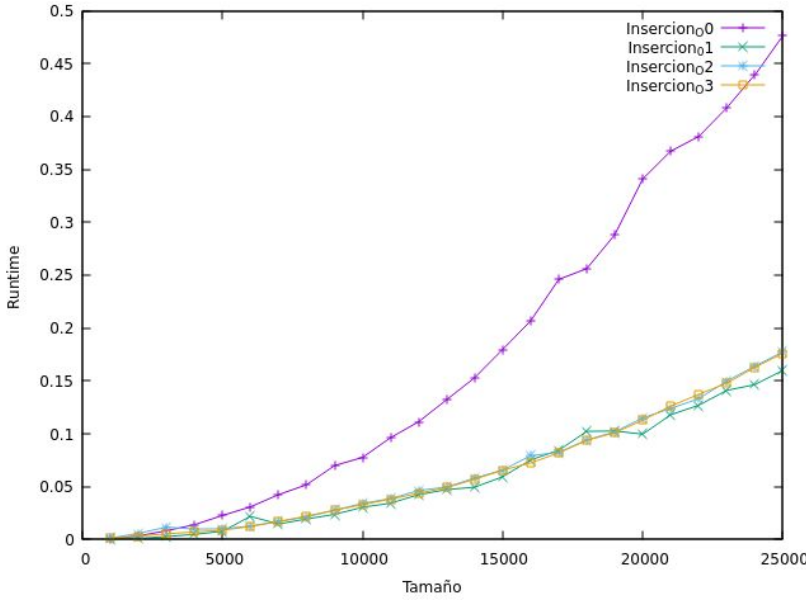
Procedemos a comparar el ajuste con los valores empírico. A pesar de que hasta el tamaño 7000 aproximadamente el ajuste no parece bueno debido a ese pequeño pico de la gráfica, pasado este umbral sí podemos afirmar que lo es.



También realizamos otros ajustes (lineal y cúbico), y como en el algoritmo de burbuja el resultado es muy similar a este. El ajuste lineal es un mal ajuste debido a su forma en línea recta, y el ajuste cúbico al acotarlo simplemente en valores positivos del eje x e y, y por su forma convexa, coincide bastante bien con nuestra función, siendo un buen ajuste para este.



Eficiencia con diferentes niveles de optimización



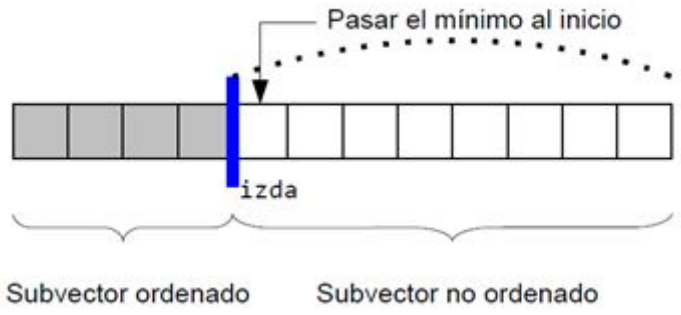
En cuanto al uso de optimización en la compilación del programa podemos apreciar en la gráfica obtenida de las ejecuciones que existe una gran diferencia

entre usarla o no. No obstante, la diferencia entre el uso de los parámetros -O1,-O2 y -O3 es bastante pequeña.

4. Algoritmo de selección:

Introducción del algoritmo

Para entender este algoritmo hay tener en cuenta que el vector se debe dividir en dos subvectores, tal que el de la izquierda contiene componentes ordenados y el de la derecha estan desordenados. Se van haciendo iteraciones en el subvector derecho desde el primer elemento al último. En cada iteración, se selecciona la componente más pequeña del subvector derecho y se coloca al final del subvector izquierdo (posición izda en la gráfica).




```
52    char minimo, intercambia;
53    int  posicion_minimo;
54
55    for (int izda = 0 ; izda < total_utilizados ; izda++){
56       minimo = vector[izda];
57       posicion_minimo = izda;
58
59       for (int i = izda + 1; i < total_utilizados ; i++){
60          if (vector[i] < minimo){
61             minimo = vector[i];
62             posicion_minimo = i;
63          }
64       }
65
66       intercambia = vector[izda];
67       vector[izda] = vector[posicion_minimo];
68       vector[posicion_minimo] = intercambia;
69    }
```

Para estudiar la eficiencia de este algoritmo me basaré en el código cuya instrucciones son **comparaciones** y **asignaciones**. Para la primera iteración (izda=0) se comparan n-1 datos en el bucle if() y en general para el elemento

i-ésimo se hacen n-i comparaciones. Como hay en total n iteraciones (de 0 a n-1) el total de comparaciones es:

$$\sum_{i=0}^{n-1}(n-i-1)=\frac{n^2-n}{2}$$

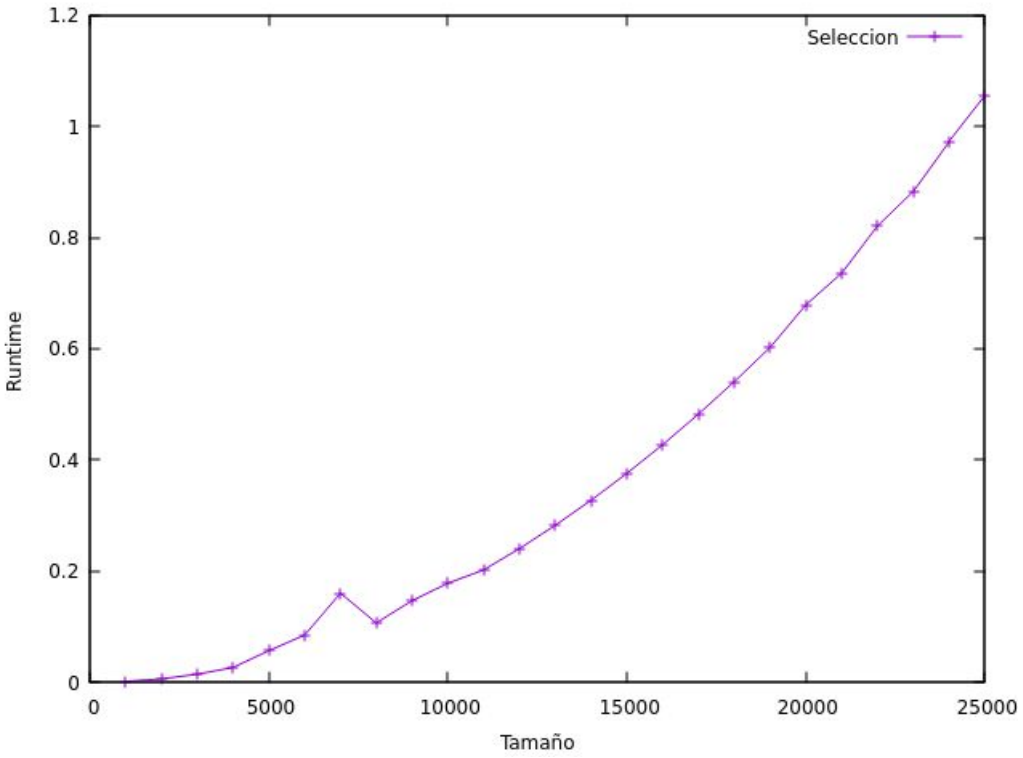
Por lo tanto el número de comparaciones no depende de la ordenación inicial del vector y depende siempre de forma cuadrática con el número de elementos, por lo que **este algoritmo presenta un comportamiento constante independiente del orden de los datos**. El número de **asignaciones** en el intercambio es en el mejor de los casos 0, que se produce cuando el vector está completamente ordenado y en el peor 3n, que ocurre cuando el vector está en orden inverso. Por lo tanto el orden de las asignaciones es lineal.

Así, al ser el mayor el orden de las comparaciones al de asignaciones y al estar ambas sumadas, aplicando la propiedad del máximo tenemos que la eficiencia del algoritmo de selección es de **O (n^2)** y además lo es también para el caso mejor y promedio.

Como **conclusión** puede observarse que el código es muy simple y aunque no tiene los mejores tiempos de ejecución, es apropiado utilizarlo para vectores de datos relativamente pequeños. Además no requiere memoria adicional ya que al igual que el ordenamiento burbuja , este algoritmo sólo necesita una variable adicional para realizar los intercambios (variable “intercambia”). Realiza pocos intercambios, dependiendo de lo desordenado que esté el vector. Por último, Tiene un rendimiento constante, pues existe poca diferencia entre el peor y el mejor caso. Es lento y poco eficiente cuando se usa en vectores grandes o medianos debido a su complejidad cuadrática y a que existen algoritmos muchos más eficientes como el mergesort. Además realiza numerosas comparaciones, como por ejemplo con el código “vector [i] < mínimo”, que se ejecuta obligatoriamente en todos los ciclos del programa.

Gráfica de la eficiencia empírica:

Para la obtención de las eficiencia empírica con el algoritmo de selección también hemos tomado un tamaño inicial de 1000 hasta uno final de 25000 de 1000 en 1000. Dibujando la gráfica con los resultados obtenidos nos queda:



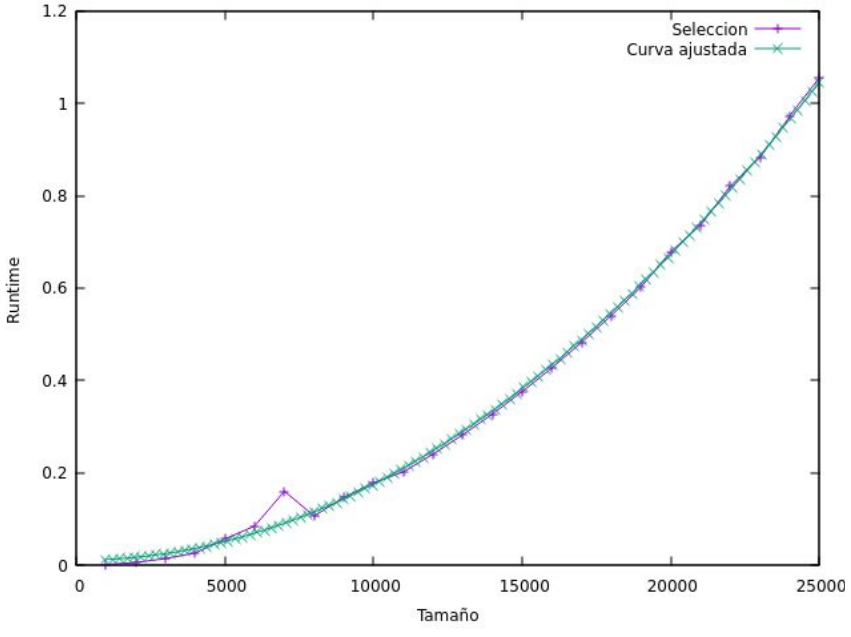
Al igual que con las anteriores hemos obtenido una gráfica con forma convexa que demuestra que es cuadrática.

Gráfica y constantes eficiencia híbrida:

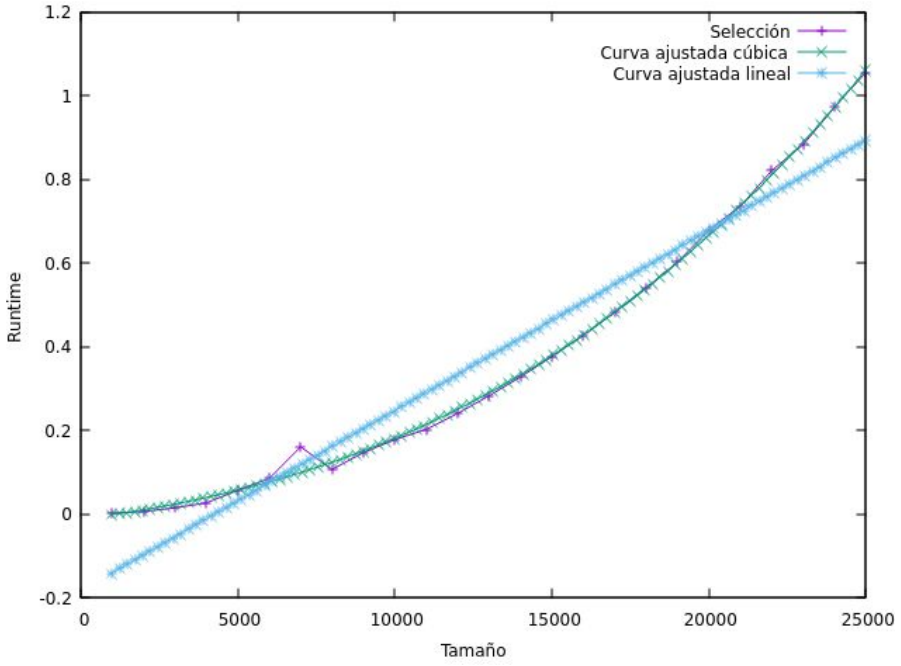
En cuanto al cálculo de la eficiencia híbrida en primer lugar hemos de obtener el valor de las constantes de la expresión general ($f(x) = a*x^2+b*x+c$) haciendo un ajuste. Una vez realizado los valores resultantes son:

- a = 1.66555e-09
- b = -2.52239e-07
- c = 0.0118689

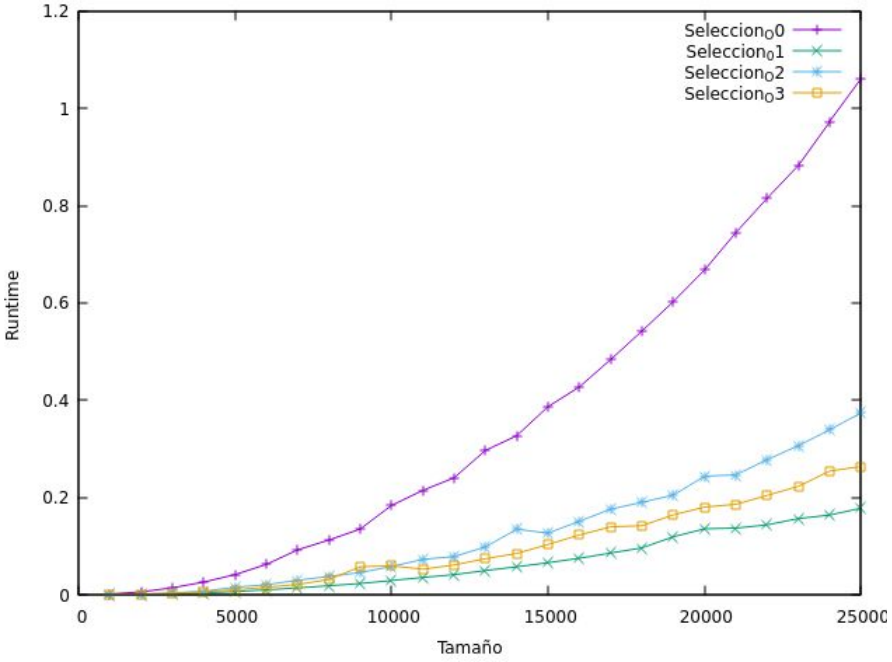
Si dibujamos la curva ajustada vemos que hemos realizado un buen ajuste:



Al realizar nuevamente los mismos ajustes que en los dos anteriores algoritmos, podemos concluir con bastante firmeza que el ajuste lineal para los algoritmos con eficiencia o(n^2) es un mal ajuste, mientras que un ajuste cúbico sí se acopla bastante bien y por lo tanto es un buen ajuste para dichos algoritmos.



Eficiencia con diferentes niveles de optimización



Una vez más comprobamos que la diferencia entre usar o no optimización es importante, pero esta vez podemos observar un caso curioso en donde las opciones -O están en distinto orden al que debería ser el lógico. Pensamos que, como ya hemos definido al principio de este trabajo, este comportamiento inusual se puede dar debido a que por ejemplo a veces la opción -O3 puede comprometernos y no

mejorar el rendimiento del programa, y la opción -O1 al solo mejorar el código y no meterse en nada más, en algunos casos puede darse que sea la más eficiente.

Reflexión sobre los algoritmos vistos

A modo de pequeña reflexión, queremos destacar el hecho de las significativas diferencias en el tiempo de ejecución para los algoritmos de ordenación que hemos visto, siendo estos del mismo orden de eficiencia O(n^2). Investigando un poco hemos observado que a la hora de poner en práctica cualquier algoritmo, en este caso los de ordenación, es muy importante saber la naturaleza del problema a abordar. Es decir, no es lo mismo probar la eficiencia de estos 3 algoritmos que hemos investigado, sobre un vector ordenado, o casi ordenado, o desordenado, ya que los resultados varían significativamente. En nuestro caso podemos ver que el algoritmo de inserción es el más eficiente de todos aventajando en más de 1 segundo al de la burbuja. Esto se puede intentar explicar básicamente diciendo que el vector que se generaba aleatoriamente, hubo diversas ocasiones en el que el vector no estaba completamente desordenado, por lo que dejaría de ejecutarse el ciclo interior en aquellos casos en que el elemento se encuentra cerca de la posición definitiva en la que deba estar ordenado, por lo que se realizarían menos intercambios. Esto se ve en teoría porque en este caso la eficiencia es lineal, cosa que no ocurre con los algoritmos de selección y de la burbuja (a no ser que este último esté mejorado). En definitiva, como programadores debemos tener en cuenta la naturaleza de los problemas a resolver para elegir sabiamente qué algoritmo será el más óptimo.

5. Algoritmo mergesort:

Introducción del algoritmo

Es una aplicación clásica de la estrategia para resolución de algoritmos "divide y vencerás". Su funcionamiento se basa en estas ideas: si la longitud del vector es 0 ó 1, entonces ya está ordenado. En otro caso se divide el vector desordenado en dos subvectores de aproximadamente la mitad del tamaño, a las que se le aplica recursivamente el algoritmo de mergesort, hasta conseguir subvectores de tamaño unidad que se van fusionando dos a dos en un solo vector ordenado y así sucesivamente, con los subvectores que se van formando, hasta conseguir el vector ordenado. Si el número de elementos del vector que se está tratando en cada momento de la recursión es menor que una constante UMBRAL, entonces se ordenará mediante el algoritmo burbuja.

¿Cómo combinar eficientemente dos listas ordenadas? Usando un vector auxiliar se procede de la siguiente manera: se coloca un puntero en la posición del elemento más pequeño en cada mitad; se añade el más pequeño de los dos al vector auxiliar y se repite hasta que se hayan añadido todos los elementos.

La eficiencia teórica de este algoritmo está explicada con detalle en el guión de la práctica, por lo que me limitaré a comentarlo. Resolviendo la ecuación

recurrente del algoritmo tenemos que T(n) = **O(nlog n)**. Al ser el algoritmo Burbuja de orden **O (n2)** y cómo para valores menores del umbral se utiliza este algoritmo entonces la eficiencia se iguala al cuadrado de n. Este algoritmo **tiene la misma eficiencia para los casos peor, promedio y mejor** ya que el tiempo que consume el algoritmo no depende de la disposición inicial de los elementos del vector.

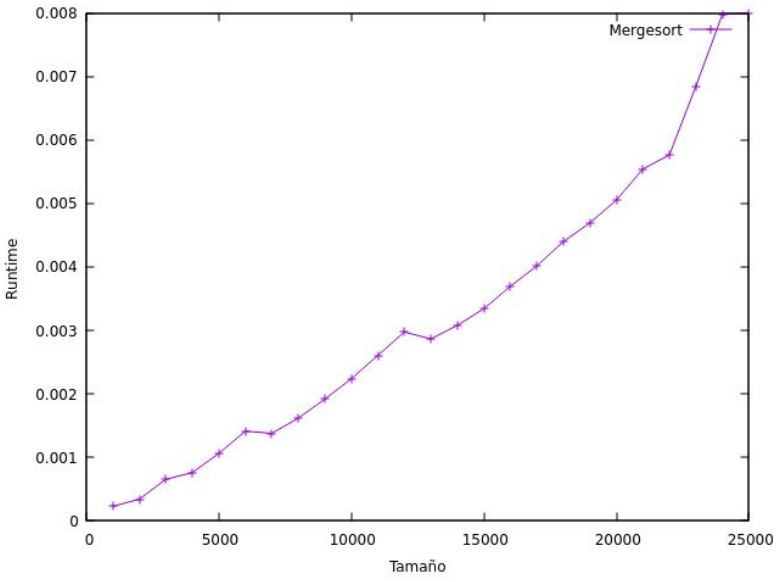
Si se aumenta el valor umbral del caso base de Mergesort, la eficiencia se ve empeorada. Esto es así ya que el valor umbral escogido en el algoritmo es el que se ajusta para que los resultados sean los más eficientes posibles y se decide después de realizar muchos intentos de aproximación. Esto quiere decir que si se modifica este umbral, probablemente el algoritmo sea más lento.

Como **conclusión** destacamos que es un algoritmo recursivo con un número de comparaciones mínimo, como se puede ver en el código. Y el tiempo de ejecución promedio es O(nlog(n)), que es el menor de todos los algoritmos estudiados de ordenación. Este algoritmo usa memoria extra de almacenamiento ya que trabaja sobre un vector auxiliar para hacer la fusión entre los dos vectores ordenados, por lo que se ve algo penalizado. Además realiza un trabajo extra consumido en las copias entre vectores (aunque es un trabajo de tiempo lineal). Sin embargo es uno de los algoritmos de ordenación más eficientes junto con el Quicksort, aunque a nivel práctico, este último se utiliza mucho más.

Este algoritmo es efectivo para ordenar un conjunto de datos que se pueden acceder secuencialmente, como vectores y listas ligadas o enlazadas. Esto es así ya que para leer el vector para determinar la mitad del mismo se necesita un acceso secuencial ya que es un método mucho más rápido.

Gráfica de la eficiencia empírica:

En este caso volvemos a tomar los mismo tamaños e incrementos (1000 hasta 25000 de 1000 en 1000) y a través del programa gnuplot dibujamos nuestros puntos y los unimos. Obteniéndose de esta forma un gráfica lineal:



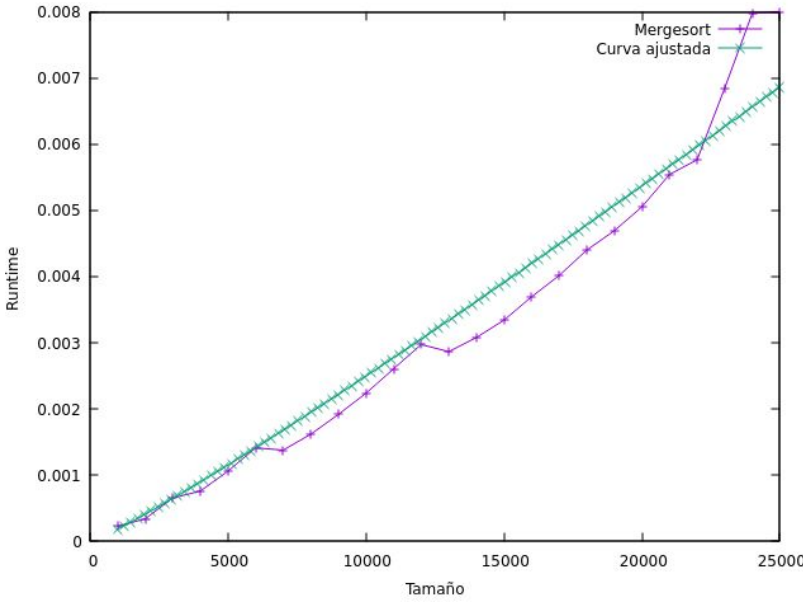
Siendo el eje Y el tiempo que tarde el programa en ejecutarse éste oscila entre 0 y 0.008 segundos. Los picos producidos en la gráfica se deben a que, la CPU no es precisa al calcular tiempos tan ínfimos, por lo que cualquier mínima interrupción en el sistema como la llegada de un nuevo proceso con ma prioridad, podría alterar la gráfica final. Y el eje X es el tamaño del vector que como hemos dicho se encuentra entre 0 y 25000.

Gráfica de la eficiencia híbrida:

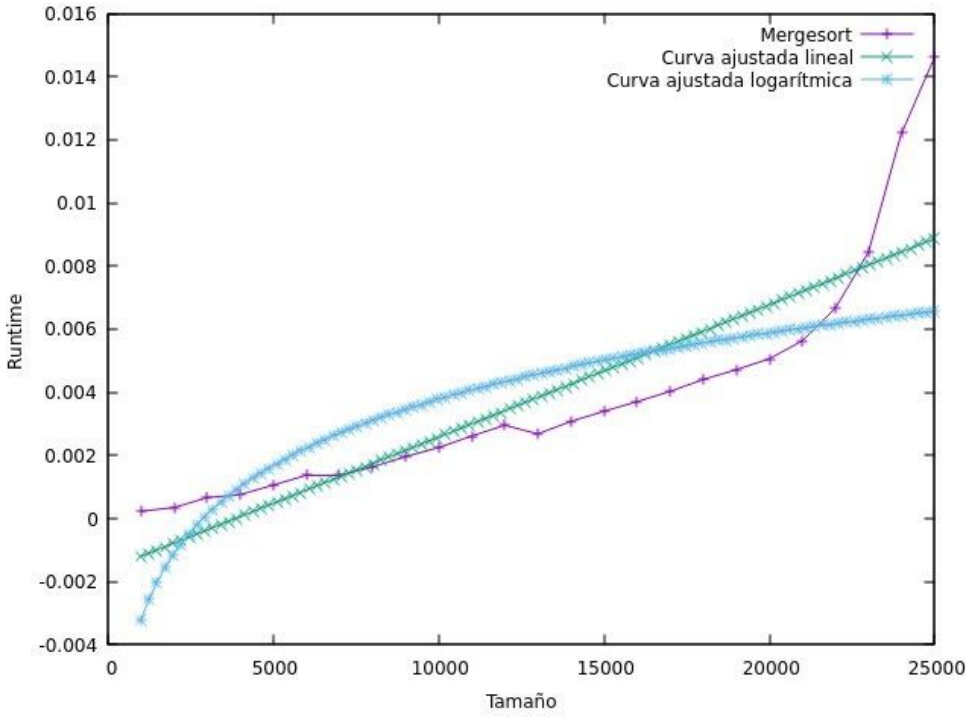
Tomamos como expresión general f(x) = a*x*log(x) para calcular la eficiencia híbrida y ajustamos para obtener la constante. Una vez se ha hecho esto tenemos:

·a = 2.83963e-08

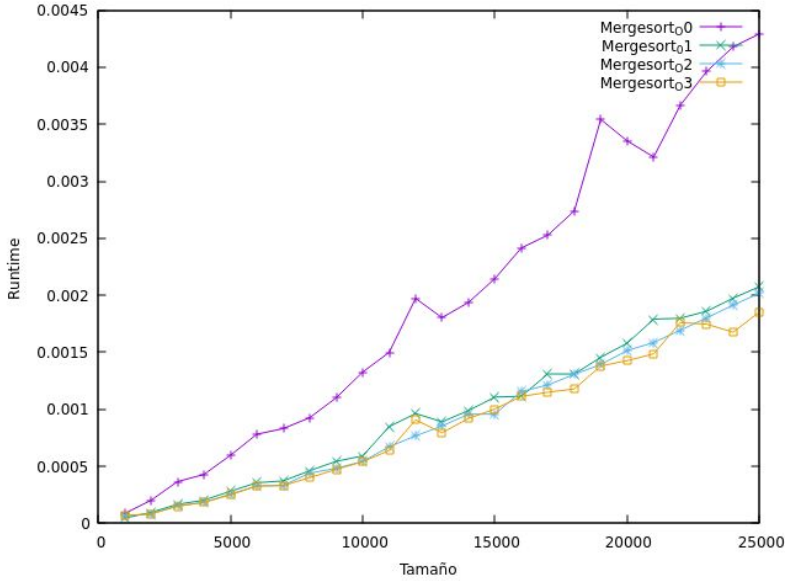
Si dibujamos el ajuste con los datos obtenidos empíricamente tenemos un buen ajuste, aunque la función ajustada difiere un poco de nuestros datos:



Procedemos ahora a realizar diferentes ajustes a nuestra función calculada a partir de los datos empíricos. Para esta ocasión vamos a comparar dicha función con un ajuste lineal y con un ajuste logarítmico para observar cómo se acoplan dichos ajustes. Observando la gráfica nos damos cuenta de que el ajuste lineal, aunque sea el que por razonamiento debería tener un mejor ajuste con nuestra función, no se ajusta tan bien como era de esperar. Y por otro lado, el ajuste logarítmico se aprecia claramente que el ajuste no es bueno.



Eficiencia con distintos niveles de optimización



Tras realizar la optimización del código con los distintos parámetros de compilación, volvemos a ver que la mejor optimización es la realizada con el parámetro -O3 pero, sin mucha diferencias entre las demás opciones. Además al no realizar ninguna opción podemos apreciar que el tiempo de ejecución del programa es mayor que si la realizamos.

6. Algoritmo quicksort:

Introducción del algoritmo

QuickSort es un algoritmo basado en la técnica de divide y vencerás, que permite, en promedio, ordenar n elementos de un vector en un tiempo proporcional a n*logn.

La técnica se basa en elegir un elemento del vector al cual llamaremos pivote, a continuación, debemos colocar los números restantes del vector de manera que a la izquierda queden los elementos menores que el pivote y a la derecha los mayores. Así el pivote ya estará bien colocado. Tras esto nos quedaremos con las 2 sublistas (izquierda del pivote y a su derecha), en donde repetiremos el proceso recursivamente mientras contengan más de un elemento. Una vez terminado este proceso todos los elementos estarán ordenados.

La eficiencia de este algoritmo dependerá de la posición en la que acabe el pivote elegido. En el mejor caso, el pivote termina en el centro de la lista, dividiéndola en dos sublistas de igual tamaño, por lo que el orden de complejidad del algoritmo es O(n*logn). En el caso promedio, su eficiencia sigue siendo O(n*logn), aquí el pivote se inclina más por un lado y las 2 sublistas tienen diferente tamaño. En el peor caso, el pivote termina en un extremo de la lista, dando lugar a un orden de complejidad del algoritmo $O(n^2)$.

MEJOR CASO **PEOR CASO** **CASO PROMEDIO**

Al estudiar tanto este algoritmo como Mergesort, nos dimos cuenta de que existe una pequeña rivalidad por ver cuál es el mejor algoritmo de ordenación puesto que, tanto quicksort como mergesort se dice que tienen el mismo orden de eficiencia O(n*logn). La discusión de este problema se soluciona fácilmente al investigar un poco sobre el asunto, ambos algoritmos son recursivos puesto que su eficiencia podemos calcularla a través de recurrencias:

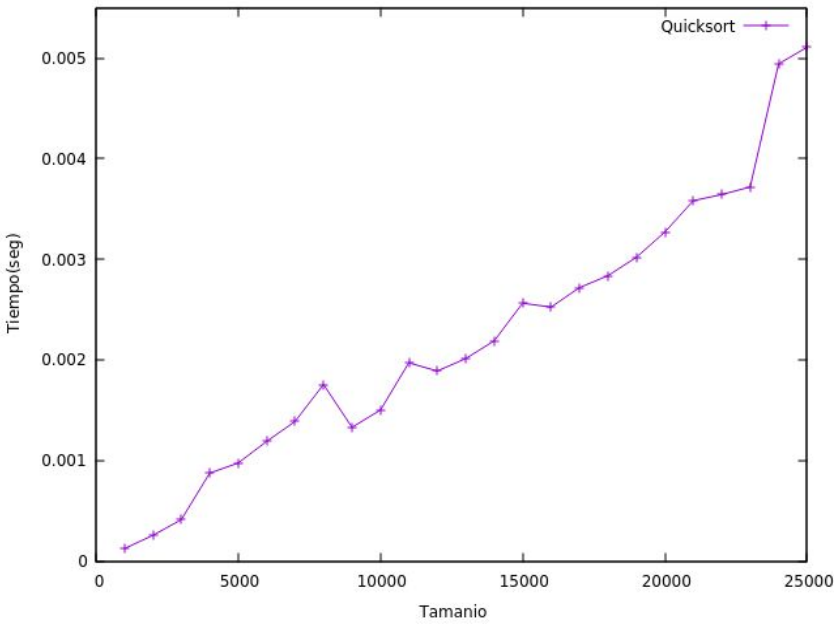
Mergesort -> $T(n) = 2T(n/2) + cn$ -> $O(n\log n)$

Quicksort -> $T(n) = T(1) + T(n-1) + an$ -> $O(n^2)$

Como podemos apreciar, teóricamente el algoritmo de mergesort es más eficiente que el de quicksort, pero entonces, ¿Por qué se le atribuye a Quicksort una eficiencia de O(nlogn) cuando acabamos de comprobar que es cuadrática?. La respuesta simplemente se basa en que, el caso promedio del algoritmo quicksort si tiene una eficiencia de O(nlogn), teniendo una ecuación recurrente muy semejante a la de mergesort. Y debido a los valores que toman las diversas constantes de ambas recurrencias, podremos observar que para el algoritmo de quicksort en la práctica, se comportan mucho mejor, dando mejores tiempos que mergesort.

Gráfica de la eficiencia empírica:

Para el estudio de la eficiencia empírica del algoritmo Quicksort hemos usado el código propuesto por el profesor, ejecutando 25 iteraciones, partiendo de un tamaño del vector de 1000 elementos enteros hasta llegar a los 25000 elementos sumándose en cada una de las iteraciones 1000 elementos. Tras esto la gráfica resultante es la siguiente:



Siendo el eje Y el tiempo de ejecución (en segundos) podemos ver que éste oscila entre 0 y 0.005 segundos, siendo tiempos extremadamente pequeños para la ejecución de algoritmos. Debido a estos tiempos podemos apreciar pequeños picos al representar la gráfica, esto se debe a que la CPU no da tiempo tan preciso por lo que cualquier pequeño imprevisto (petición al sistema, otro proceso en ejecución...) pueden alterar la gráfica. Destacar también que como los demás algoritmos de ordenación, en el eje X se encuentra el número de elementos del vector. A primera vista por la forma de la gráfica no podemos deducir su tendencia, es decir, si la forma de la función se asemeja más a una eficiencia lineal, cuadrática...

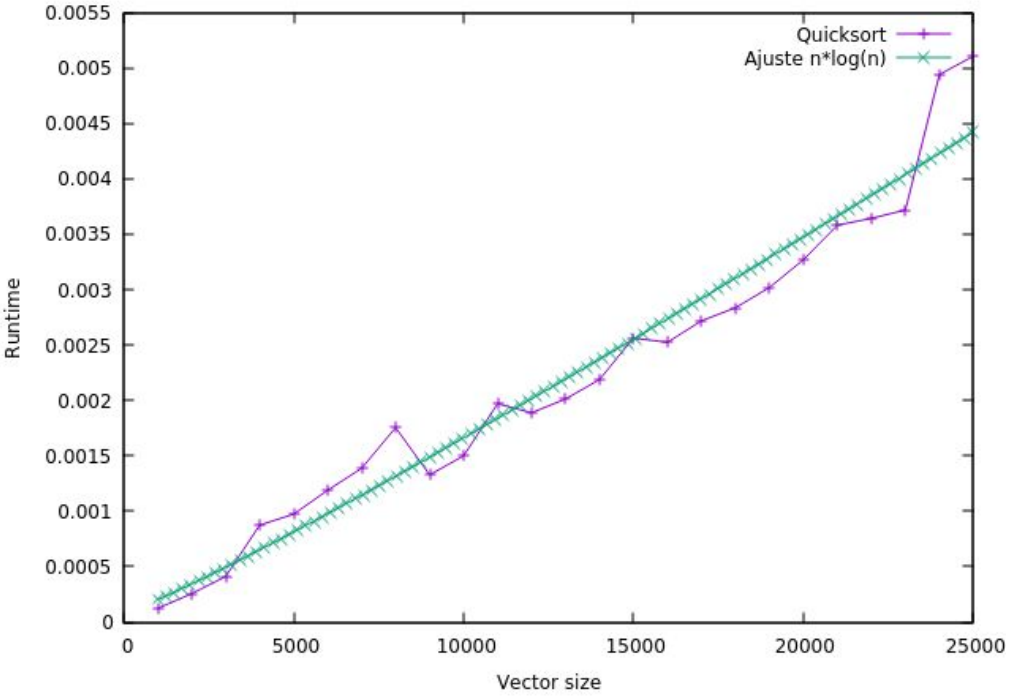
Gráficas y constantes eficiencia híbrida

Para describir completamente la expresión general dada por el cálculo teórico debemos averiguar el valor de las constantes que aparecen en dicha expresión. En esta ocasión nuestra función será f(x) = a*x*log(x)+b y el conjunto de puntos los resultados obtenidos del análisis empírico anterior.

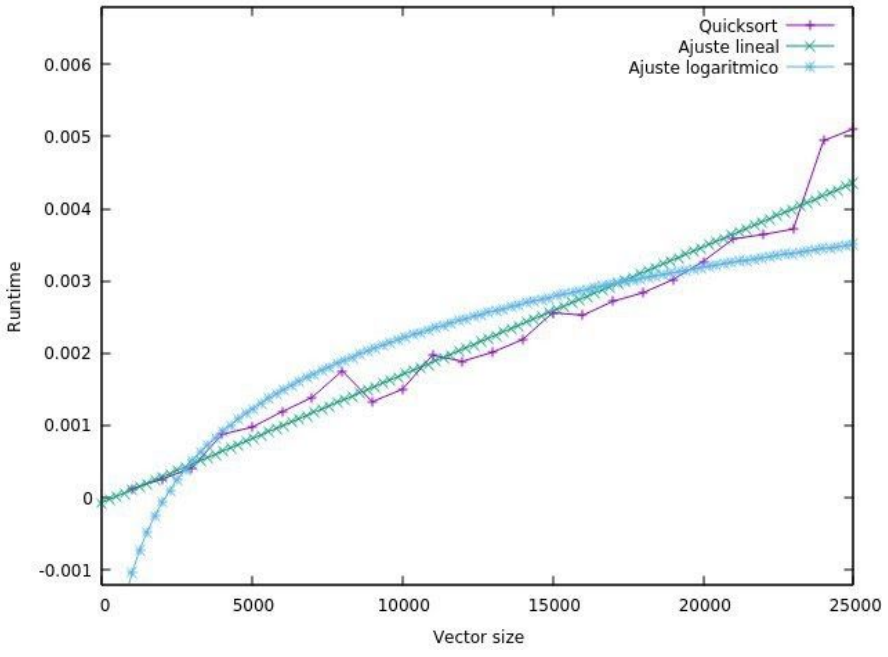
Tras ajustar la gráfica siguiendo los pasos que nos brinda el pdf de la práctica los valores de las constantes son:

- a = 1.7128e-08
- b = 8.27823e-05

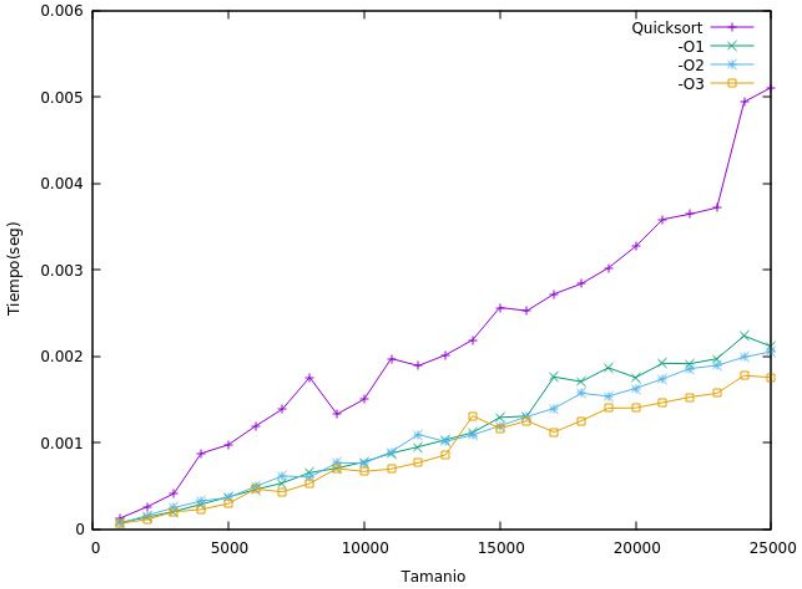
Al obtener los valores ya podemos dibujar la gráfica para comparar el ajuste:



A pesar de los picos, los cuales ya explicamos su origen, el ajuste de dicha función no es muy bueno pero sí se asemeja bastante bien. Para más investigación hemos realizado otros ajustes (lineal y logarítmico) para observar cómo se ajustan el conjunto de resultados obtenidos mediante el análisis empírico a otra función. Podemos apreciar en la gráfica que el ajuste lineal se acopla decentemente en algunos tramos de la función pero en general no es un muy buen ajuste. Por otro lado sí que observamos que el ajuste logarítmico es un mal ajuste para nuestra función.



Eficiencia con distintos grados de optimización



De nuevo vemos que la optimización mejora en gran medida los tiempos del algoritmo de ordenación. Se puede ver que el grado de optimización -O3 mejora ligeramente los otros dos grados. Aún así, no sale demasiado rentable utilizar la mayor opción de compilación, puesto que podemos ver algún pico que iguala o supera en ese instante el tiempo de ejecución del algoritmo, por lo que lo más rentable para este algoritmo sería la opción de compilación -O1. De esta forma, la computadora no consume demasiados recursos y no afecta demasiado en los tiempos de ejecución.

7. Algoritmo heapsort:

Introducción del algoritmo

El algoritmo Heapsort es un método de ordenamiento basado en comparación, usando un Montículo o Heap como estructura de datos. Este método es más lento que otros métodos, pero es más eficaz en escenarios más rigurosos como con datos desordenados. Se define como un método No Recursivo, No Estable y con eficiencia O(log n) para todos sus casos.

Este algoritmo consiste en almacenar todos los elementos de una lista a ordenar en un montículo (heap), y luego extraer el nodo raíz, o la cima, en sucesivas iteraciones obteniendo el conjunto ordenado. Basa su funcionamiento en una propiedad de los heap, por la cual, la raíz contiene siempre el menor elemento (o el mayor, según se haya definido el heap) de todos los almacenados en él. El algoritmo, después de cada extracción, recoloca en el nodo raíz o cima, la última hoja por la derecha del último nivel. Este suceso destruye la propiedad heap del árbol, pero tras esto, se realiza un proceso descendente del número insertado de

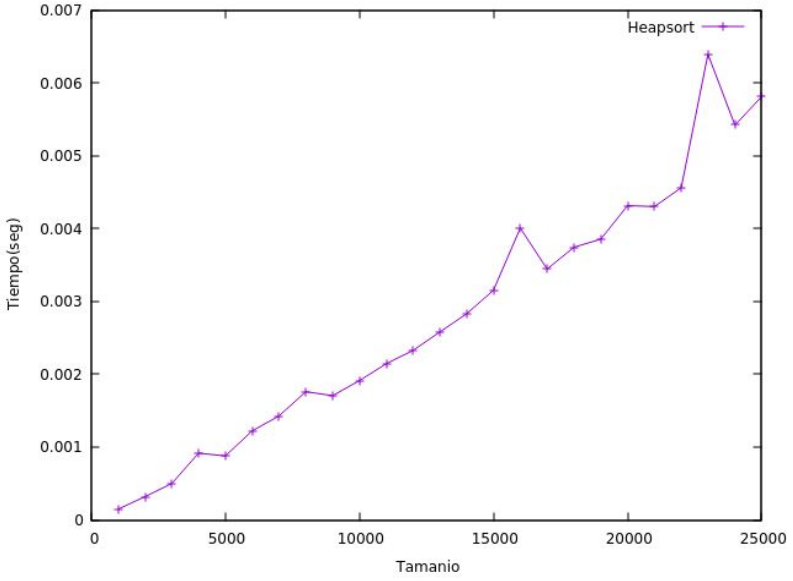
forma que a cada movimiento se selecciona el mayor de sus dos hijos con el cual se intercambia.

Heapsort garantiza eficiencia O(nlogn) en todos sus casos, lo que es mucho mejor que el peor de los casos en Quicksort. Además no necesita más memoria para que otro vector coloque los datos ordenados como la que se necesita en mergesort. Pero entonces, ¿Por qué no se menciona Heapsort como un mejor algoritmo que estos dos?. Tras probar los diferentes algoritmos te das cuenta de que Quicksort tiene algo realmente especial, ya que se ejecuta rápido, mucho más rápido que los algoritmos Heap y Merge. Analizando los tres algoritmos nos damos cuenta de que el misterio se puede tratar en que el algoritmo Quicksort casi no hace intercambios de elementos innecesarios.

El intercambio lleva mucho tiempo, y por ejemplo en Heapsort, incluso si todos sus datos ya están ordenados, va a intercambiar el 100% de los elementos para ordenar el vector. Y en el caso de Mergesort, es aún peor, ya que va a escribir el 100% de los elementos en otro vector y volver a escribirlo en el original, incluso si los datos ya están ordenados. Mientras que con Quicksort, hace el mismo número de comparaciones que los demás, pero a diferencia del resto no intercambia casi nada. Eso es lo que le da a Quicksort el mejor momento; menos intercambio, más velocidad.

Gráfica de la eficiencia empírica

Para el estudio de la eficiencia empírica del algoritmo Heapsort hemos usado el código propuesto por el profesor, ejecutando 25 iteraciones, partiendo de un tamaño del vector de 1000 elementos enteros hasta llegar a los 25000 elementos sumándose en cada una de las iteraciones 1000 elementos. Tras esto la gráfica resultante es la siguiente:



Siendo el eje Y el tiempo de ejecución (en segundos) podemos ver que éste oscila entre 0 y 0.007 segundos, siendo tiempos extremadamente pequeños para la ejecución de algoritmos. Debido a estos tiempos podemos apreciar pequeños picos

al representar la gráfica, esto se debe a que la CPU no da tiempo tan preciso por lo que cualquier pequeño imprevisto (petición al sistema, otro proceso en ejecución...) pueden alterar la gráfica. Destacar también que como los demás algoritmos de ordenación, en el eje X se encuentra el número de elementos del vector. A primera vista por la forma de la gráfica no podemos deducir su tendencia, es decir, si la forma de la función se asemeja más a una eficiencia lineal, cuadrática...

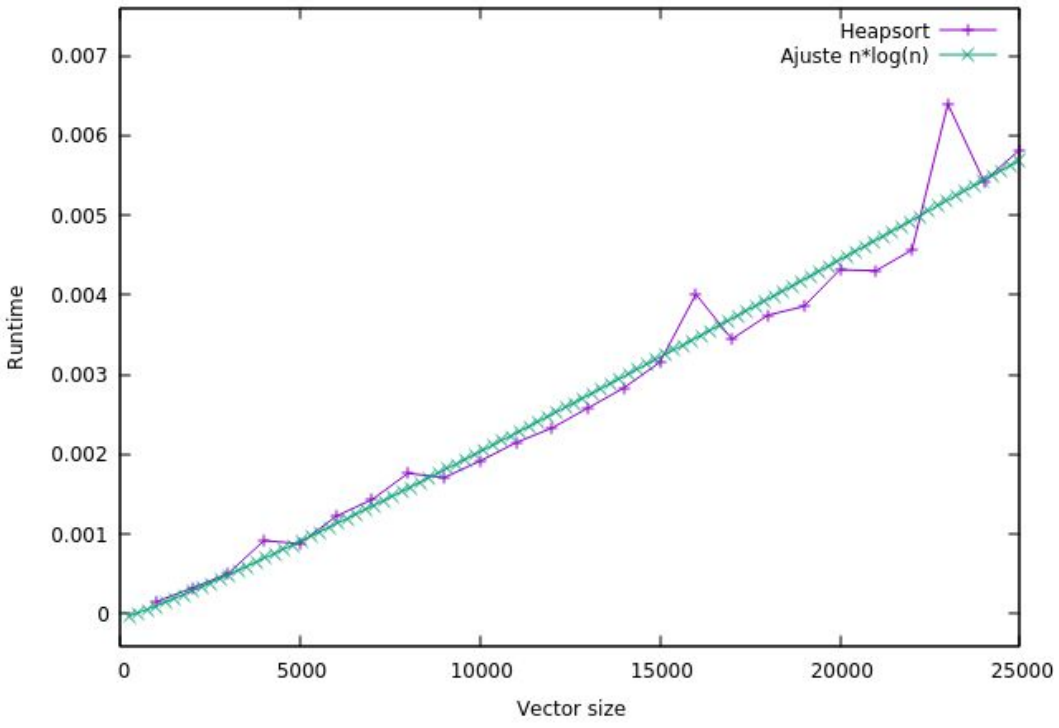
Gráficas y constantes eficiencia híbrida

Para describir completamente la expresión general dada por el cálculo teórico debemos averiguar el valor de las constantes que aparecen en dicha expresión. En esta ocasión nuestra función será f(x) = a*x*log(x)+b y el conjunto de puntos los resultados obtenidos del análisis empírico anterior.

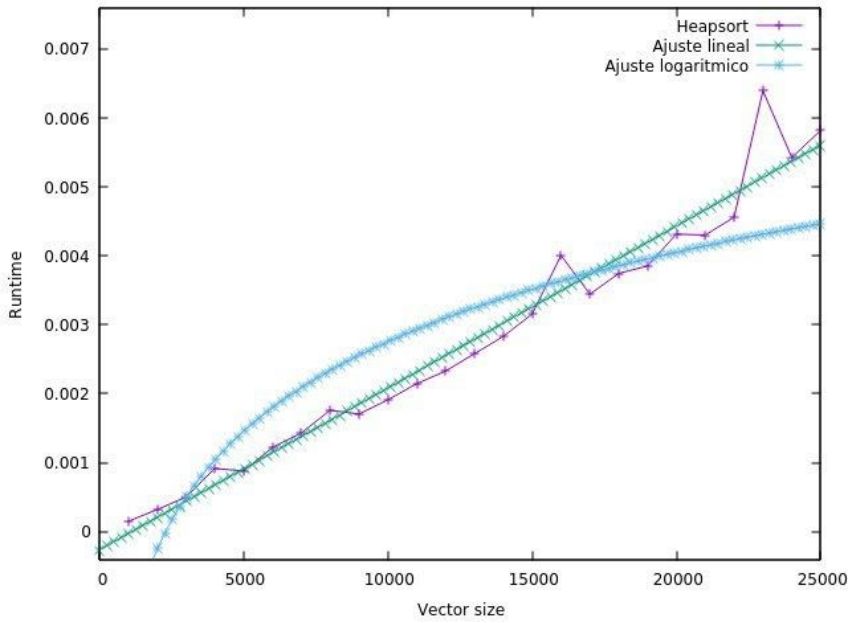
Tras ajustar la gráfica siguiendo los pasos que nos brinda el pdf de la práctica los valores de las constantes son:

- a = 2.27127e-08
- b = -6.10804e-05

Al obtener los valores ya podemos dibujar la gráfica para comparar el ajuste:

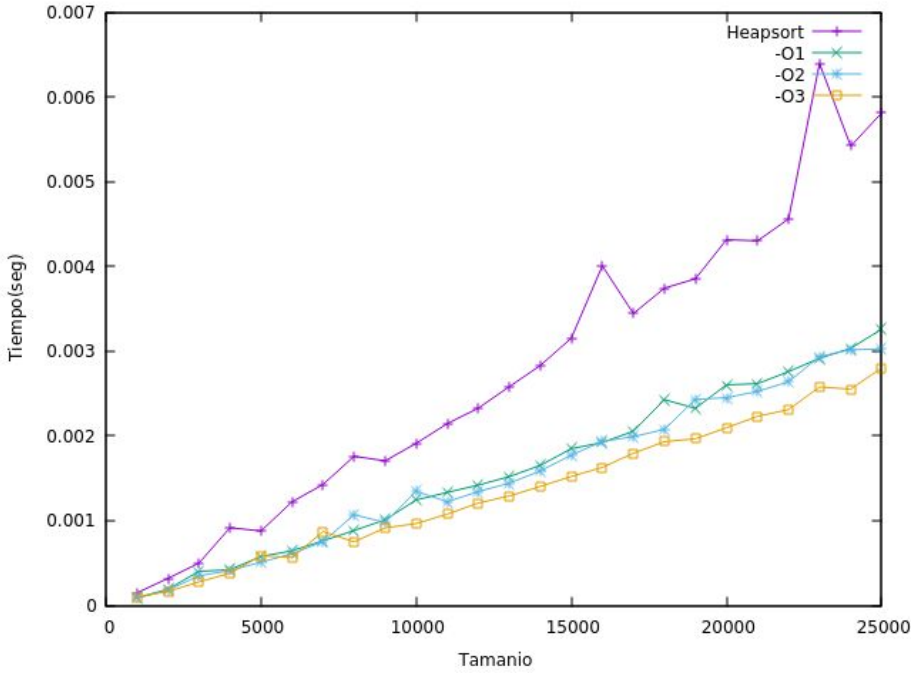


A pesar de los picos, los cuales ya explicamos su origen, el ajuste es bastante bueno. Para más investigación hemos realizado otro ajuste (lineal y logarítmico) para observar cómo se ajustan el conjunto de resultados obtenidos mediante el análisis empírico a otra función. El resultado ha sido el siguiente:



En conclusión junto con los otros algoritmos de ordenación de orden O(nlogn), no podemos afirmar con certeza que el ajuste lineal sea un buen ajuste para dichos algoritmos puesto que, dependiendo del mismo, se ajusta en mejor o peor medida (destacando que el heapsort es el algoritmo en donde mejor se acopla).Pero, lo que sí podemos concluir es que el ajuste logarítmico es un pésimo ajuste para todos los algoritmos de dicha ordenación.

Eficiencia para diferentes niveles de optimización



Como podemos ver en la gráfica, las opciones de compilación tardan menos que el algoritmo sin optimizar. Podemos ver que ocurre el mismo caso que en el algoritmo de Quicksort, puesto que la opción -O3 mejora ligeramente los tiempos, pero no lo suficiente, por los picos generados. Por tanto, al igual que en Quicksort,

no sale rentable usar la opción -O3. Y como las otras dos opciones son prácticamente iguales, la mejor opción es -O1.

8. Algoritmo de Floyd:

Introducción del algoritmo

El algoritmo de Floyd-Warshall, es un algoritmo de análisis sobre grafos para indicar el camino más corto entre cualquier par de nodos en grafos dirigidos ponderados. El algoritmo encuentra el camino entre todos los pares de vértices en una única ejecución con una eficiencia de $O(n^3)$.

El algoritmo de Floyd representa una red de 'n' nodos como una matriz cuadrada de orden 'n', que llamaremos la matriz C. De esta forma, el valor 'Cij' representa el coste de ir desde el nodo i al nodo j. Los valores que representamos en la matriz puede ser cualquier número entero o infinito. Cabe destacar cuando una posición valdrá 0 o infinito; cuando tenga el valor de infinito significará que no existe unión entre los nodos. Y las diagonal representa el coste para ir de un nodo a sí mismo, por lo que no sirven para nada, y se inicializan a 0. Los pasos a dar para realizar el algoritmo son:

- Formar las matrices iniciales C y D, donde C es la matriz de adyacencia del grafo, y D es una matriz del mismo tamaño vacía que se irá rellenando con la longitud del camino mínimo que una cada par de nodos.
- Se toma k=1, se selecciona la fila y la columna k de la matriz C y entonces, para i y j, con $i \neq k$, $j \neq k$ e $i \neq j$, hacemos:

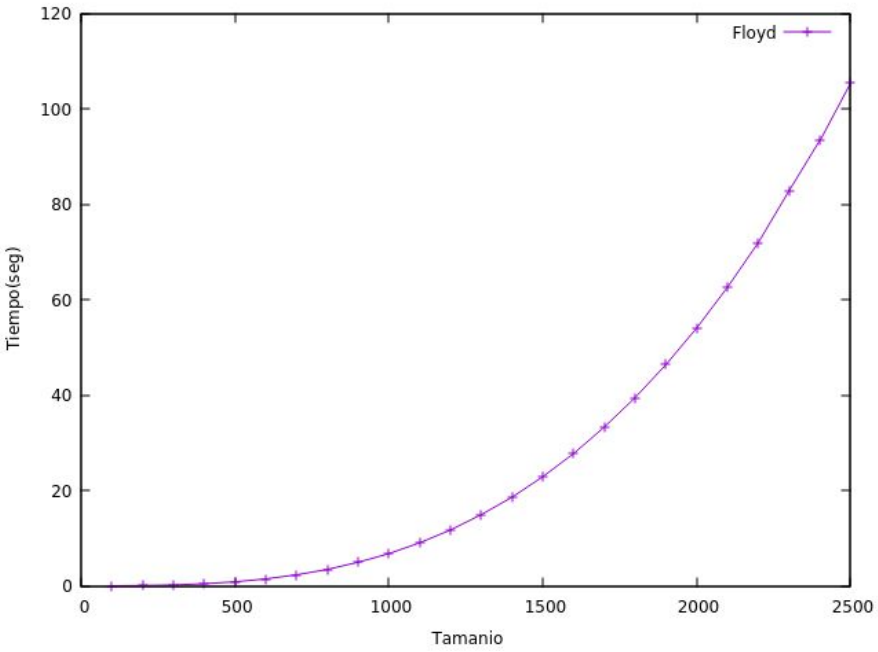
  $$\text{Si } (Cik + Ckj) < Cij \rightarrow Dij = Dkj \text{ y } Cij = Cik + Ckj$$

  En caso contrario, dejamos las matrices como están.
- Si $k \leq n$, aumentamos k en una unidad y repetimos el paso anterior, en caso contrario páramos las interacciones.

En definitiva el algoritmo de Floyd funciona ejecutando y encontrando primero el camino mínimo con k=1 para todos los pares (i,j), usándolos para después hallar los caminos mínimos con k=2 para los pares (i,j), y así recursivamente hasta que k=n. Y como podemos observar en todo lo que llevamos explicado, su eficiencia viene dado por el triple bucle en donde se recorre para cada k<=n, todos los pares (i,j).

Gráfica de la eficiencia empírica

Para el estudio de la eficiencia empírica del algoritmo de Floyd hemos usado el código propuesto por el profesor, ejecutando 25 iteraciones, partiendo de un tamaño del vector de 100 elementos enteros hasta llegar a los 2500 elementos sumándose en cada una de las iteraciones 100 elementos. Tras esto la gráfica resultante es la siguiente:



Siendo el eje Y el tiempo de ejecución (en segundos) podemos ver que éste oscila entre 0 y 120 segundos. Destacar también que en el eje X de este algoritmo se encuentra el número de nodos en un grafo ponderado. A primera vista por la forma de la gráfica podemos deducir que su tendencia es cuadrática o cúbica.

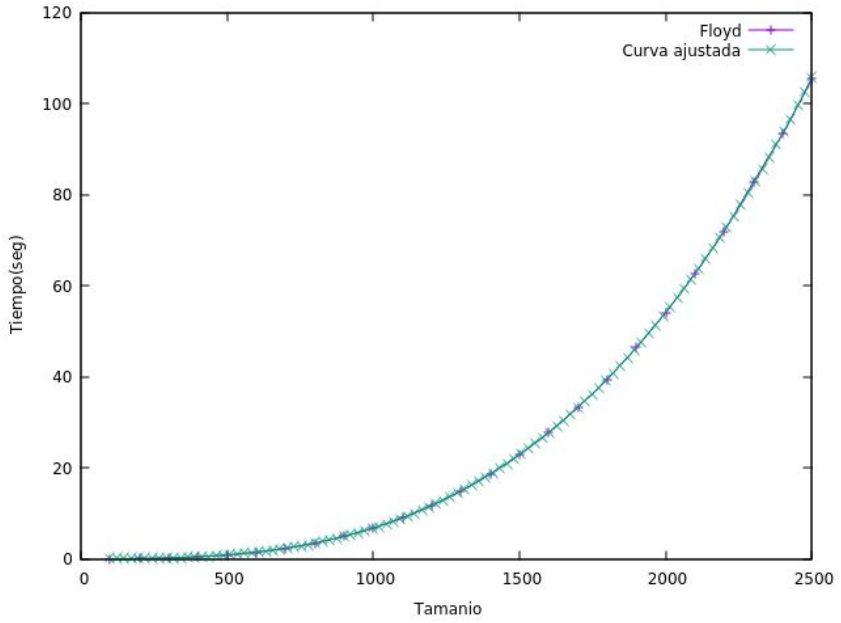
Gráficas y constantes eficiencia híbrida

Para describir completamente la expresión general dada por el cálculo teórico debemos averiguar el valor de las constantes que aparecen en dicha expresión. En esta ocasión nuestra función será f(x) = a*x**3+b*x**2+c*x+d y el conjunto de puntos los resultados obtenidos del análisis empírico anterior.

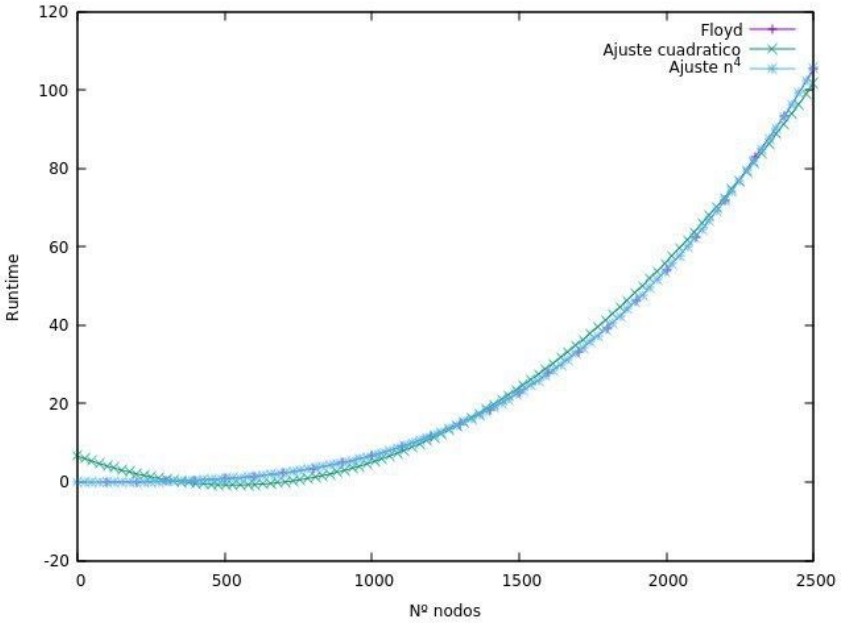
Tras ajustar la gráfica siguiendo los pasos que nos brinda el pdf de la práctica los valores de las constantes son:

- a = 6.72177e-09
- b = 1.60832e-07
- c = -0.000158545
- d = 0.0301171

Al obtener los valores ya podemos dibujar la gráfica para comparar el ajuste:

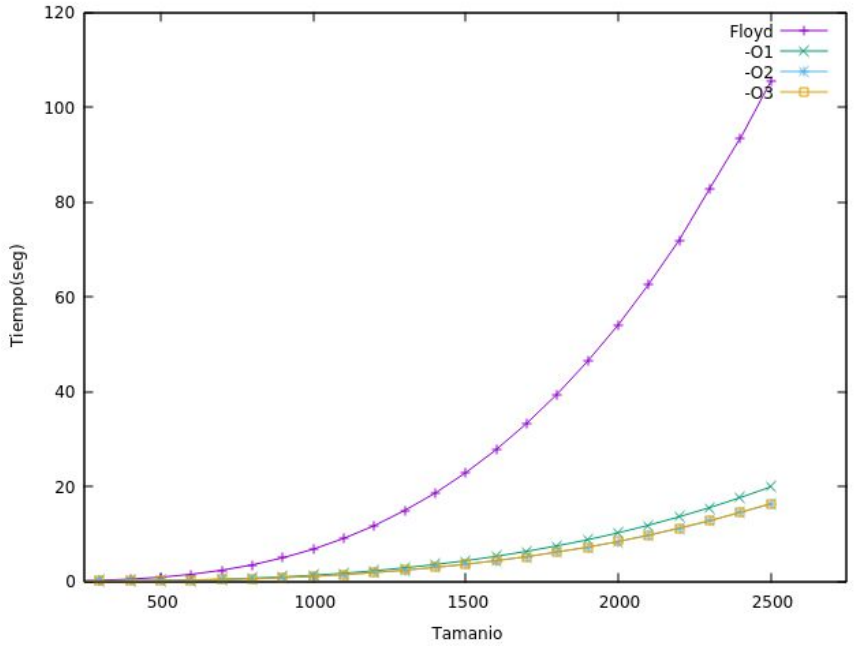


Al observar la gráfica vemos que el ajuste es muy bueno. Para una investigación más profunda hemos realizado otro ajuste (cuadrático y n^4) para observar cómo se ajustan el conjunto de resultados obtenidos mediante el análisis empírico a otra función. El resultado ha sido el siguiente:



Podemos apreciar en la gráfica que, en primer lugar, el ajuste cuadrático no se acopla muy bien en los valores más pequeños, pero, según va creciendo la función, mejor es el ajuste entre ambos. Por otro lado vemos que el ajuste de n^4, es un muy buen ajuste ya que coincide en gran medida con nuestra función.

Eficiencia con diferentes niveles de optimización



Se observa que hay una notable diferencia en tiempos de ejecución al utilizar grados de optimización con respecto a la ejecución sin optimizar. Entre ellos no se aprecia igual puesto que los grados -O2 y -O3 tienen prácticamente los mismos tiempos en ejecución por lo que es lo mismo qué grado de optimización uses a partir de -O2. Por tanto, no sale rentable para el sistema usar la opción -O3 puesto que consume más memoria de la computadora que las demás opciones.

9. Algoritmo de las torres de Hanoi:

Introducción del algoritmo

Las Torres de Hanoi es un juego inventado por el matemático francés Edouard Lucas en 1883. Dicho matemático se inspiró en una leyenda acerca de un templo hindú donde al principio de los tiempos, se les propuso a unos sacerdotes transferir 64 discos de oro (cada uno más pequeño que el inferior) colocados todos en un poste, a uno de los dos postes restantes. Además se les impuso dos limitaciones importantes.

- Sólo podían mover un disco a la vez
- Nunca podían colocar un disco más grande encima de uno más pequeño

La leyenda dice que cuando terminaran su trabajo,el templo se desmenuzaría en polvo y el mundo se desvanecería. Y a día de hoy sabemos que, no estaba tan mal encaminada la leyenda puesto que sabemos que el número de movimientos necesarios para mover correctamente una torre de 64 discos es 18,446,744,073,709,551,615. En donde si hacemos un movimiento cada segundo tardaríamos la friolera de 584,942,417,355 años.

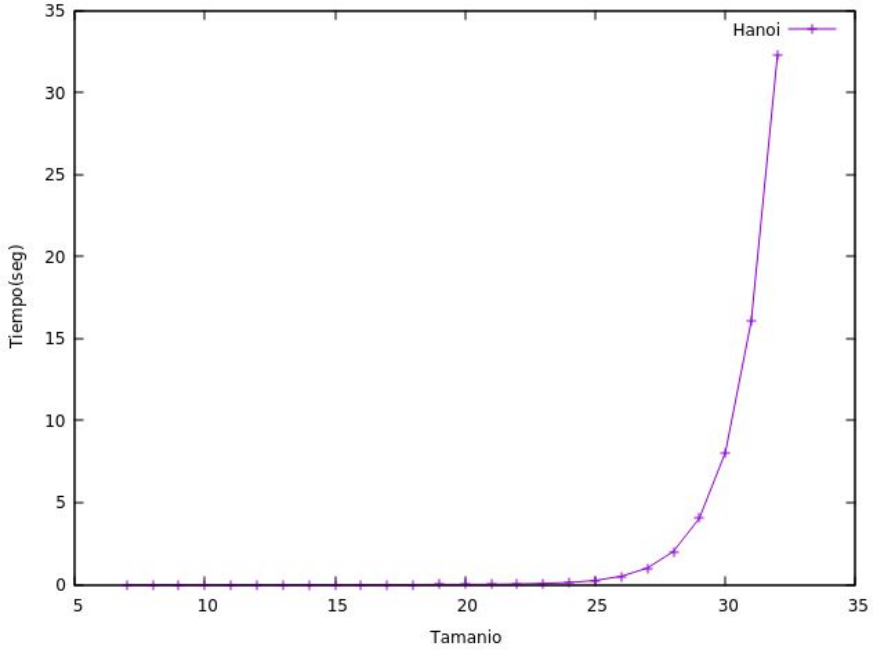
Existen diversas formas de resolver este juego, pero hemos elegido la forma recursiva para analizarla. Los pasos a seguir para resolverlo de esta manera serían:

- Crear 3 pilas de números, con nombres como origen, auxiliar y destino
- Si origen == 1
    - Primero movemos el disco 1 de la pila origen a la pila destino
    - terminamos
- sí no
    - movemos todas las fichas menos la más grande al poste auxiliar
- Movemos la ficha grande a la varilla final
- Movemos todas los discos restantes, 1 .. n-1, encima de la ficha grande
- terminamos

De este modo, el número de movimientos mínimo a realizar para resolver el problema de este modo es de $2^n - 1$, siendo 'n' el número de discos.

Gráfica de la eficiencia empírica

Para el estudio de la eficiencia empírica del algoritmo de las torres de Hanoi hemos usado el código propuesto por el profesor, ejecutando 25 iteraciones, partiendo de un tamaño del vector de 7 elementos enteros hasta llegar a los 32 elementos sumándose en cada una de las iteraciones 1 elemento. Tras esto la gráfica resultante es la siguiente:



Siendo el eje Y el tiempo de ejecución (en segundos) podemos ver que éste oscila entre 0 y 35 segundos. Destacar también que en el eje X de este algoritmo se encuentra el número de discos que hay que mover. A primera vista por la forma de la gráfica podemos deducir que su tendencia es exponencial.

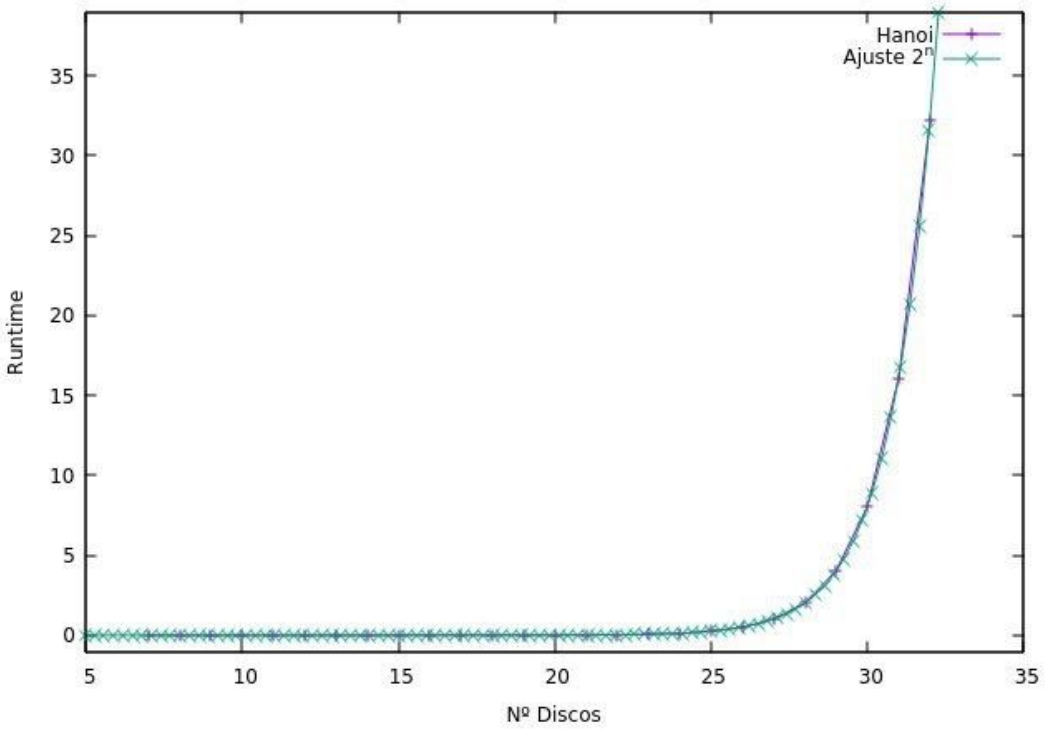
Gráficas y constantes eficiencia híbrida

Para describir completamente la expresión general dada por el cálculo teórico debemos averiguar el valor de las constantes que aparecen en dicha expresión. En esta ocasión nuestra función será f(x) = a*2**x+b y el conjunto de puntos los resultados obtenidos del análisis empírico anterior.

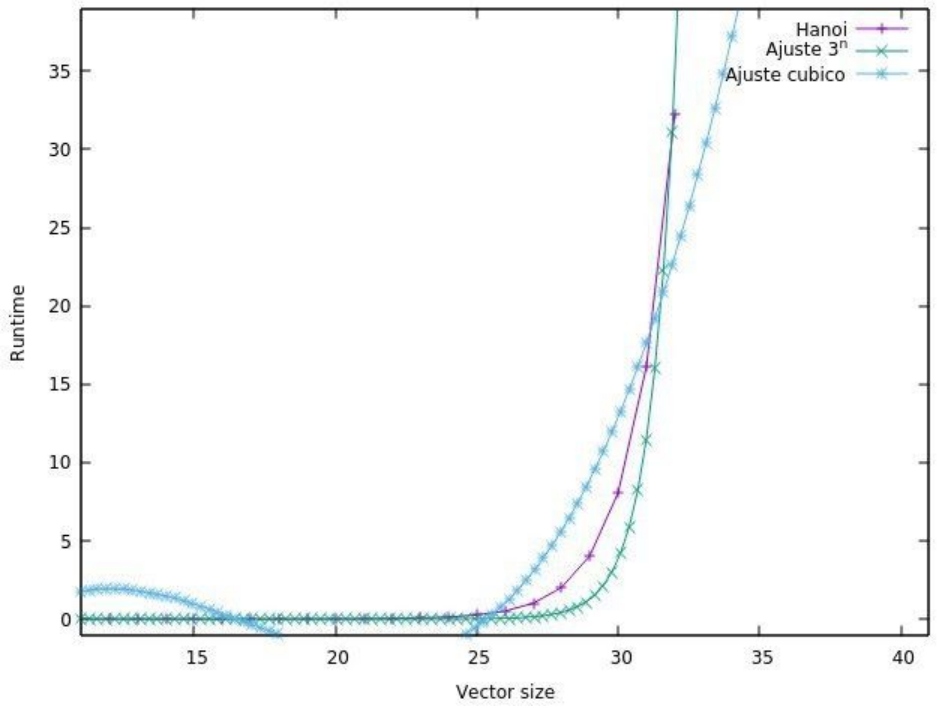
Tras ajustar la gráfica siguiendo los pasos que nos brinda el pdf de la práctica los valores de las constantes son:

- a = 7.51274e-09
- b = -0.0144888

Al obtener los valores ya podemos dibujar la gráfica para comparar el ajuste:

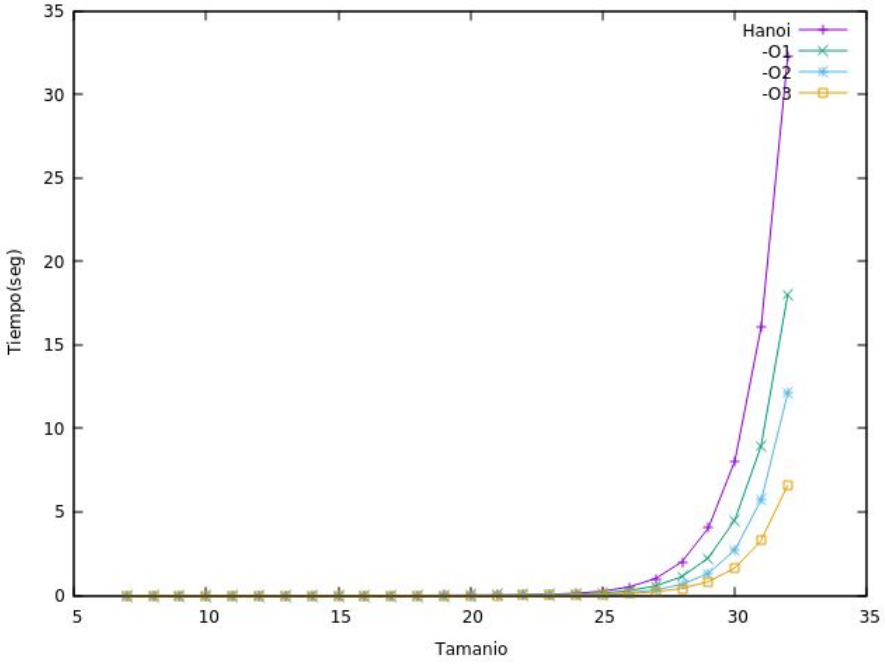


Al observar la gráfica vemos que el ajuste es muy bueno. Para una mayor investigación hemos realizado otro ajuste (cúbico y 3^n) para observar cómo se ajustan el conjunto de resultados obtenidos mediante el análisis empírico a otra función. El resultado ha sido el siguiente:



Al observar la gráfica, en primer lugar, nos damos cuenta de que el ajuste 3^n, como era de esperar, empieza a crecer después que nuestra función. Aún así, cabe destacar como tanto al principio el ajuste es muy bueno para los valores más pequeños, y también al final, podemos apreciar un tramo de la función que se acopla bastante bien. En definitiva podríamos decir que el ajuste no es pésimo pero tampoco sería el óptimo. Por otro lado, el ajuste cúbico, se ve perfectamente como no se acopla en ningún momento a nuestra función, siendo por ende, un pésimo ajuste.

Eficiencia con diferentes niveles de optimización



Como podemos ver, por cada grado de optimización, se mejora en gran medida los tiempos de ejecución llegando incluso tardar menos de la mitad de lo que tarda el algoritmo sin optimizar en el caso de -O3. Además, por cada grado de

optimización, se mejora levemente lo que tarda el algoritmo, por lo que saldría muy rentable utilizar la mayor opción de optimización para el algoritmo de las torres de Hanoi.

Conclusión

A modo de conclusión, nos gustaría reflejar brevemente las ideas más representativas que hemos desarrollado e investigado a lo largo de este trabajo. En primer lugar destacar las significativas diferencias en el tiempo de ejecución para algoritmos de ordenación con el mismo orden de eficiencia, en los cuales, será tarea del programador elegir cual utilizar dependiendo de la naturaleza de su problema. En segundo lugar, las diferencias entre los algoritmo de ordenación quick-heap-merge/sort, en donde destacamos al mergesort como el mejor algoritmo de ordenación teóricamente, y el quicksort el mejor en la práctica gracias a no emplear intercambios en su código. Finalmente destacar las diferencias abismales que hemos apreciado entre los algoritmos descritos anteriormente con los que analizamos a su posterior en la práctica, es decir, Floyd y hanoi, donde sus órdenes de eficiencia eran peores que la de los algoritmos de ordenación. Además resaltar la importancia de los algoritmos recursivos, ya que se encuentran en un montón de problemas ya no solo para la ordenación de vectores, sino en cosa más cotidianas como para la resolución de un juego o para indagar más en la programación con grafos.