Práctica 1
Algorítmica

Análisis de Eficiencia de Algoritmos

Realizado por grupo Orquídeas (subgrupo A1):

- Azorín Martí, Carmen
- Cribillés Pérez, María
- Ortega Sevilla, Clara
- Torres Fernández, Elena

Profesora: Lamata Jiménez, María Teresa

Curso 2021/2022 2º cuatrimestre

Índice

1. Introducción

En esta práctica hemos realizado el análisis de la eficiencia de **4 algoritmos: inserción, quicksort, floyd y hanoi**. En primer lugar, con el cálculo de la **eficiencia teórica** hemos obtenido sus órdenes de eficiencia: $O(n^2)$, $O(nlogn)$, $O(n^3)$ y $O(2^n)$ respectivamente.

En segundo lugar, hemos estudiado la **eficiencia empírica** ejecutando el mismo algoritmo para diferentes tamaños de entrada y obteniendo así tablas de al menos 25 datos que relacionan el tamaño con el tiempo en segundos. Con estas tablas hemos construido gráficas de nubes de puntos. Además, para resaltar la **variación de la eficiencia empírica** en función de parámetros externos, hemos realizado los análisis en ordenadores y sistemas operativos diferentes, y con distinta optimización.

En tercer lugar, hemos calculado la **eficiencia híbrida** usando la herramienta *gnuplot*. Así, hemos ajustado la función teórica con la nube de puntos del análisis empírico, obteniendo las constantes ocultas. De esta forma, hemos podido calcular los tiempos para tamaños muy grandes con una cota de error muy pequeña.

Por último, hemos escrito los aspectos más **relevantes** de esta práctica.

2. Propiedades de los equipos

Para poder apreciar bien la diferencia entre la eficiencia de ciertos algoritmos en diferentes entornos, lo que hemos hecho ha sido utilizar diferentes sistemas operativos y distintos ordenadores con características variadas. A continuación especificamos el ordenador utilizado por cada alumna con sus propiedades y el sistema operativo utilizado:

- Lenovo Ideapad 330-15IKB con Intel core i5 8th Gen 4GB RAM
  - Sistema operativo: Ubuntu 20.04.2 LTS
- ASUS ZenBook con AMD® Ryzen 7 3700U 16 GB RAM
  - Sistema operativo: Ubuntu 20.04.2 LTS
- HP Pavilion con Intel core i5 10th Gen 8 GB RAM
  - Sistema operativo: Windows 10 versión 21H2
- ASUS TUF Gaming FX505DT_FX505DT con AMD® Ryzen 7 3750H 16 GB RAM
  - Sistema operativo: Ubuntu 20.04.3 LTS

3. Eficiencia teórica

La eficiencia teórica o eficiencia a priori, es la que no depende del agente tecnológico utilizado, sino en cálculos matemáticos. Para ello, calculamos una función matemática que nos indica cómo va a evolucionar el tiempo de ejecución del algoritmo según varía el tamaño.

- **Inserción**

```
inline static void insercion(int T[], int num_elem){                    //O(n^2)
        insercion_lims(T, 0, num_elem);
}


static void insercion_lims(int T[], int inicial, int final){            //O(n^2)

        int i, j;                                                       //O(1)
        int aux;                                                        //O(1)
        for (i = inicial + 1; i < final; i++) {                         //O(n)
                j = i;                                                  //O(1)
                while ((T[j] < T[j-1]) && (j > 0)) {                    //O(n)
                        aux = T[j];                                     //O(1)
                        T[j] = T[j-1];                                  //O(1)
                        T[j-1] = aux;                                   //O(1)
                        j--;                                            //O(1)
                };
        };
}
```

Para analizar el código, debemos empezar desde lo más profundo e ir yendo hacia capas más superficiales. Por tanto, empezamos viendo las sentencias de dentro del while. Todas de ellas son O(1), mientras que el while se ejecuta en el peor de los casos un total de n veces. Es decir, el bucle while tiene una eficiencia de O(n). Este bucle while está dentro de un for que también se ejecuta en el peor de los casos n veces. Por tanto, como se trata de dos bucles anidados de cada uno O(n), la eficiencia será de O(n^2), ya que la declaración de variables son de O(1) y por la regla del máximo, la eficiencia de inserción es O(n^2).

- **Quicksort**

```
inline void quicksort(int T[], int num_elem){
        quicksort_lims(T, 0, num_elem);                    // O( n log n)
}

static void quicksort_lims(int T[], int inicial, int final){
        int k;                                             // O(1)
        if (final - inicial < UMBRAL_QS) {
                insercion_lims(T, inicial, final);         // O(n²)
        } else {
                dividir_qs(T, inicial, final, k);          // O(n)
                quicksort_lims(T, inicial, k);             // T(n/2)
                quicksort_lims(T, k + 1, final);           // T(n/2)
        };
}

static void dividir_qs(int T[], int inicial, int final, int & pp){   // O(n)
        int pivote, aux;                                   // O(1)
        int k, l;                                          // O(1)

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

En este algoritmo se distinguen dos casos según el tamaño del vector. Si este es menor que el de un determinado **umbral (UMBRAL_QS)** se aplica el método de **inserción** que, como hemos visto, tiene eficiencia $O(n^2)$. En caso contrario se ejecuta el else donde se aplica el algoritmo de **quicksort**, que es el que se ha usado en nuestras mediciones ya que nuestros tamaños de prueba comenzaban en 10000 superando el umbral prefijado (50).

El quicksort se divide en tres partes: **dividir**, que se encarga de separar el vector en dos mitades con el pivote en medio (con eficiencia O(n), de ahí que en la función recursiva sumemos un n) y **dos llamadas recursivas** a la propia función pasando como parámetros cada una de las mitades (hacen referencia a 2T(n/2). En cada caso irán cambiando los índices pero no el vector en sí.

Con todo esto se nos queda la siguiente función a trozos:

$$T(n) = \begin{cases} 2T(\frac{n}{2}) + n & \textit{si } n >= \textit{UMBRAL\_QS} \\ n^2 & \textit{si } n < \textit{UMBRAL\_QS} \end{cases}$$

Entonces, resolvemos la recurrencia de la siguiente forma:

1) Realizamos un cambio de variable: $n= 2^k$
2) La ecuación queda $T(2^k)= 2T(2^{k-1})+ 2^k$
3) Cambiamos $T(2^k)= t_k$ –> $t_k= 2t_{k-1}+ 2^k$
4) Resolviendo la ecuación no homogénea obtenemos:
   $t_k= 2t_{k-1}+ 2^k$ —> $t_k - 2t_{k-1} = 2^k$ —> $(x-2)(x-2)^{1+0} = 0$ —>
   $(x-2)^2 = 0$, teniendo una raíz doble —> $t_k = c_1 2^k + kc_2 2^k$
5) Rehaciendo los cambios de variable:
   $T(n) = c_1 n + c_2 n \log_2 n$
6) Entonces, el orden de eficiencia es O(nlogn).

- **Floyd**

```
void Floyd(int **M, int dim) {
  for(int k = 0; k < dim; k++){                          // O(n³)
        for(int i = 0; i < dim; i++){                    // O(n²)
              for(int j = 0; j < dim; j++){              // O(n)
                    int sum = M[i][k] + M[k][j];         // O(1)
                    M[i][j] = (M[i][j] > sum) ? sum : M [i][j];   // O(1)
              }
        }
    }
}
```

Nos encontramos ante el algoritmo Floyd en su versión iterativa. La porción de código en el cuerpo del tercer bucle se puede acotar por una constante a. Por tanto, tendríamos una fórmula como la siguiente:

$$\sum_{k=1}^{dim}\sum_{i=1}^{dim}\sum_{j=1}^{dim} a$$

Renombrando en la ecuación anterior dim con n y realizando la sumatoria, obtenemos:

$$\sum_{k=1}^{n} an^2$$

Y finalmente tenemos: $an^3 \in O(n^3)$. Diremos por tanto que el algoritmo Floyd es de orden O(n³) o cúbico.

- **Hanoi**

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

El algoritmo de Hanoi recibe como parámetros el número de discos con los que trabajamos (M), el vástago inicial en el que están los discos (i) y el vástago donde queremos que estén tras el proceso (j).

Como vemos en su implementación, este algoritmo emplea recursividad, luego para hallar su orden de eficiencia tendremos que resolver una recurrencia. En este caso, observando el código vemos que la ecuación a resolver es $\mathbf{t_n = 2t_{n-1} + 1}$, pues se hacen dos llamadas recursivas a la función hanoi con un número de disco menos y se le suma 1 para mover el disco suelto.

Para resolverla, se trata de una recurrencia no homogénea que operamos como sigue: $\mathbf{t_n = 2t_{n-1} + 1}$ —> $\mathbf{t_n - 2t_{n-1} = 1}$ —> $\mathbf{(x-2)(x-1)^{1+0}= 0}$, obteniendo 2 raíces simples: 1 y 2. Por tanto, la solución general de la recurrencia es $\mathbf{t_n = c_1 1^n + c_2 2^n}$, siendo el **orden de eficiencia** $\mathbf{O(2^n)}$.

4. Eficiencia empírica

La eficiencia empírica consiste en medir los recursos empleados (tiempo) de un algoritmo para cada tamaño de entrada y poder estudiar así su comportamiento ante un determinado problema. En concreto, a cada algoritmo le hemos dado al menos **25 tamaños de entrada** y hemos obtenido el **tiempo medio en segundos** (con 15 mediciones por tamaño) que ha tardado en realizarse el algoritmo por cada tamaño. Estos datos los hemos recogido en distintas **tablas y gráficas de nubes de puntos**.

Esta medición sí está influida por los agentes tecnológicos empleados. Por ello, nosotras hemos realizado los cálculos en **distintos ordenadores, sistemas operativos y con distintas optimizaciones**, para poder comparar los resultados y ver las diferencias.

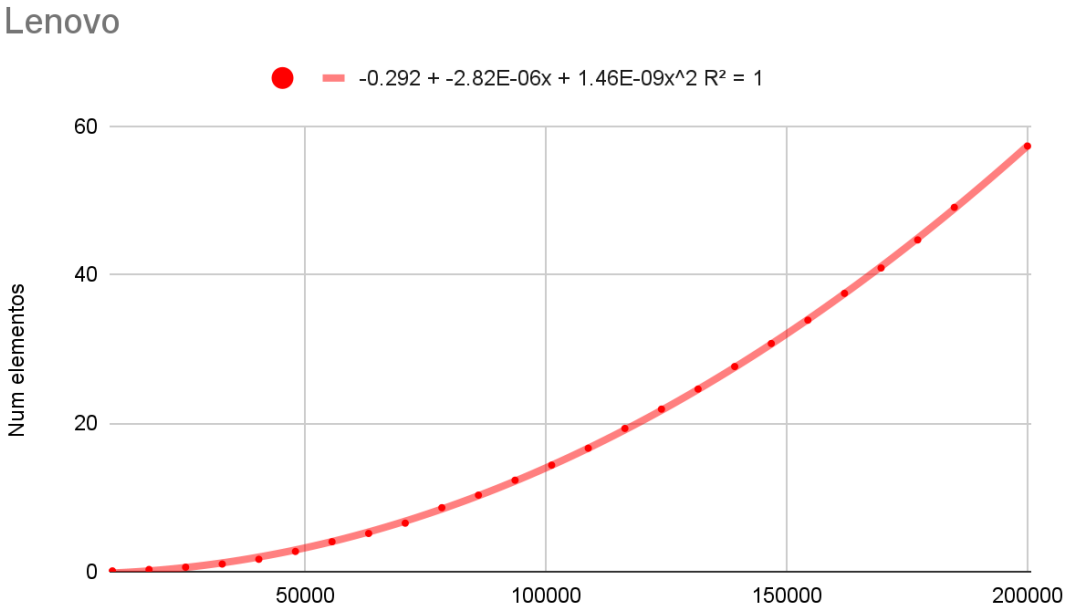
En las siguientes páginas mostramos los datos obtenidos para cada algoritmo:

➢ Eficiencia de Inserción

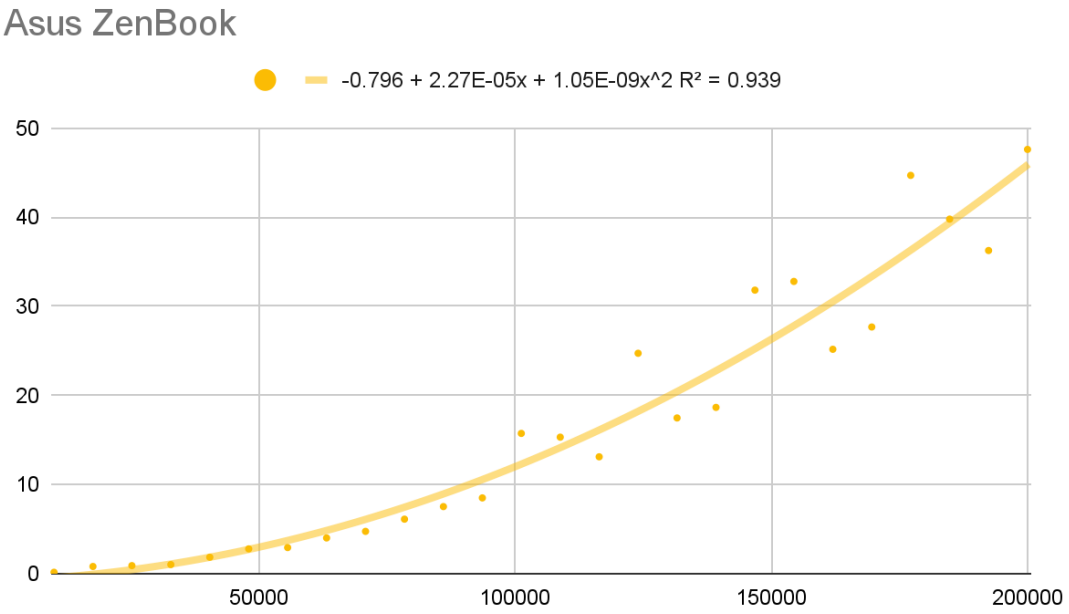
Tabla de resultados de cada equipo:

	Tiempo en segundos			
Tamaño	Lenovo Linux	Asus ZB Linux	HP Windows	Asus TUF Linux
10000	0.103739	0.154353	0.0895833	0.154754
17600	0.295425	0.806163	0.276042	0.477506
25200	0.60292	0.901037	0.567708	0.905295
32800	1.02349	1.01687	0.954167	1.34233
40400	1.66687	1.84485	1.46354	1.89785
48000	2.72089	2.78081	2.03854	2.55954
55600	4.02799	2.92721	2.74688	3.35165
63200	5.14102	4.00799	3.54896	4.23295
70800	6.52936	4.74449	4.45833	7.30038
78400	8.61307	6.11526	5.47813	6.16603
86000	10.3053	7.52466	6.58542	7.50179
93600	12.3138	8.5003	7.79688	8.91055
101200	14.3464	15.738	9.125	10.2724
108800	16.6199	15.3166	10.5552	11.8887
116400	19.2888	13.113	12.101	25.4573
124000	21.8875	24.7299	13.699	14.59
131600	24.5872	17.4731	15.5031	26.6347
139200	27.6224	18.658	17.2385	18.4113
146800	30.7228	31.8162	19.1792	20.4599
154400	33.8661	32.7927	21.375	22.5997
162000	37.4767	25.1787	23.375	50.7792
169600	40.892	27.68	25.599	45.3625
177200	44.6695	44.694	27.9792	42.7522
184800	49.0572	39.7848	30.4146	42.1534
192400	53.2235.	36.2612	32.9667	68.4989
200000	57.31	47.607	35.6542	37.9448

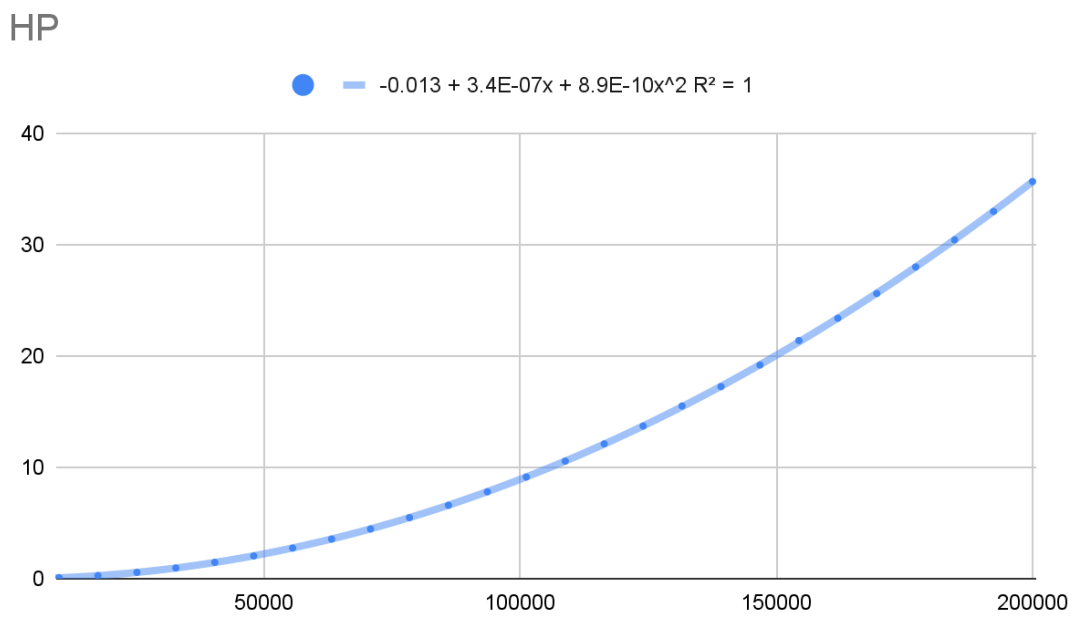
❖ Gráfica eficiencia Lenovo:



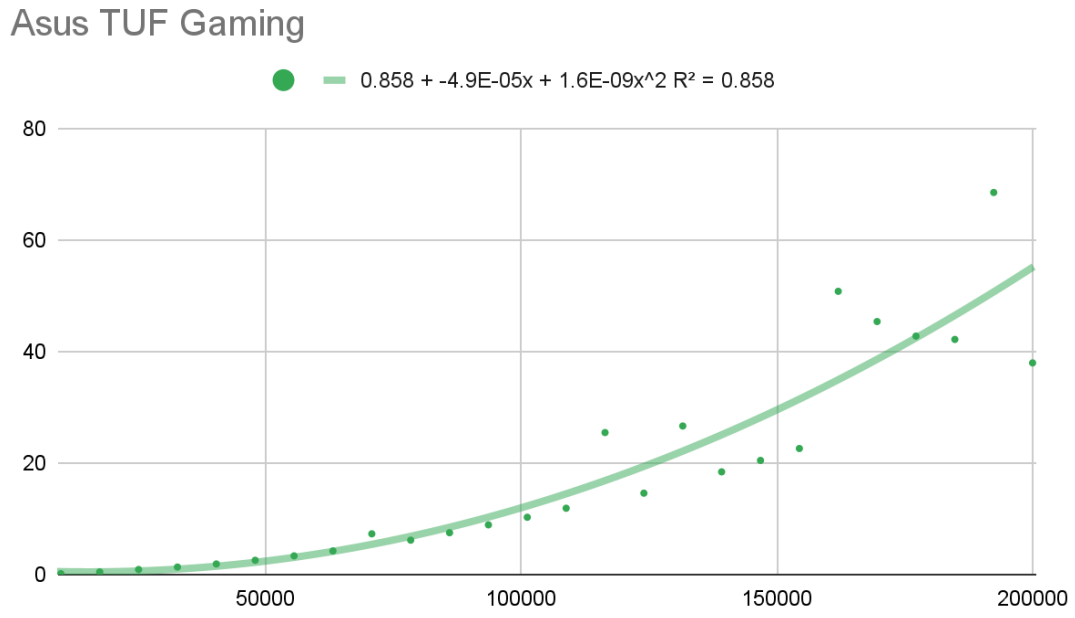
❖ Gráfica eficiencia Asus ZenBook:



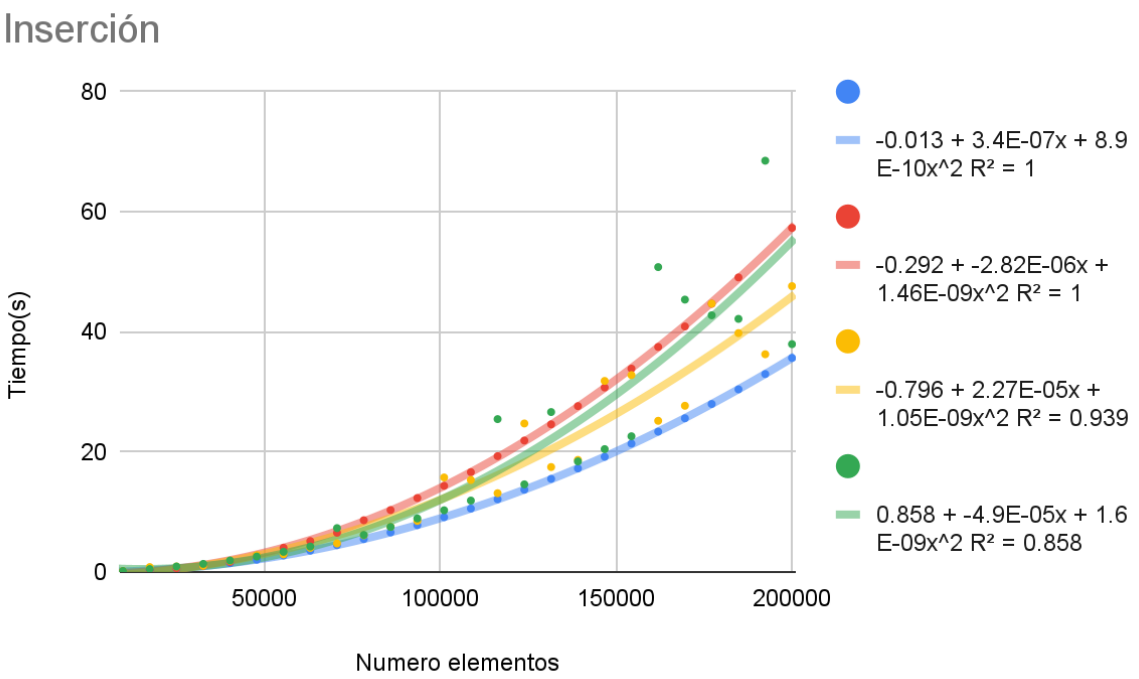
❖ Gráfica eficiencia HP:



❖ Gráfica eficiencia en Asus TUF Gaming:



❖ Gráfica con las cuatro eficiencias empíricas



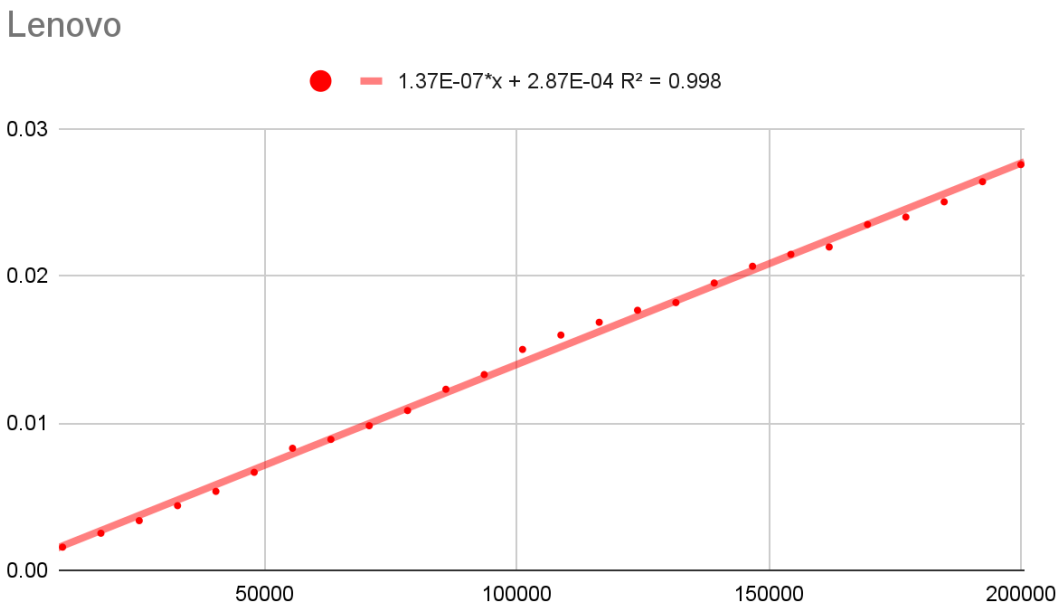
➢ Eficiencia de Quicksort

Tabla de resultados de cada equipo:

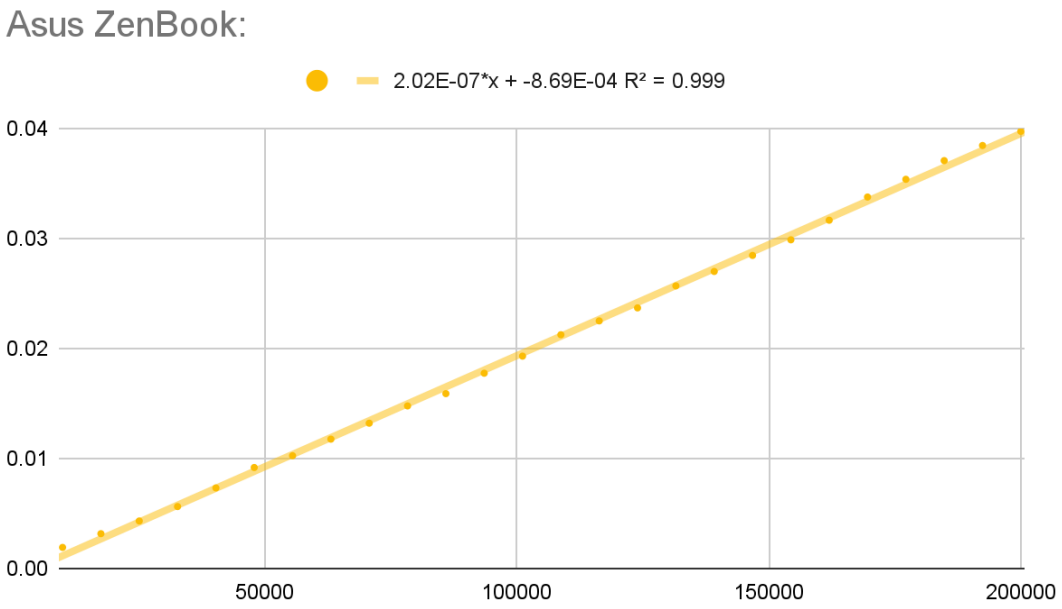
	Tiempo en segundos			
Tamaño	Lenovo Linux	Asus ZB Linux	HP Windows	Asus TUF Linux
10000	0.00157147	0.00191413	0.00104167	0.00202813
17600	0.00251467	0.00316233	0.00104167	0.00285733
25200	0.00337107	0.00431627	0.003125	0.00429553
32800	0.00438927	0.0056252	0.00416667	0.0056162
40400	0.00535993	0.0073186	0.00520833	0.00703013
48000	0.00665567	0.00917473	0.00520833	0.00870267
55600	0.0082884	0.0102435	0.00625	0.0100077
63200	0.00888773	0.0117489	0.00729167	0.0114201
70800	0.00981893	0.0131993	0.00833333	0.0129769
78400	0.0108499	0.0147686	0.009375	0.0148925
86000	0.0122959	0.0158852	0.0104167	0.0164071
93600	0.0132956	0.0177395	0.0114583	0.0181213
101200	0.0150033	0.0192915	0.0125	0.0190417
108800	0.0159787	0.0212374	0.0135417	0.0207113
116400	0.016847	0.0224931	0.015625	0.0223959
124000	0.017663	0.0236748	0.0177083	0.0239608
131600	0.018184	0.0256752	0.0166667	0.0251119
139200	0.0195171	0.0269841	0.01875	0.0268683
146800	0.0206531	0.0284414	0.0177083	0.0282424
154400	0.0214638	0.0298597	0.0197917	0.0308383
162000	0.0219619	0.0316391	0.0229167	0.0317703
169600	0.0234903	0.0337469	0.021875	0.0331234
177200	0.0239984	0.0353577	0.0229167	0.0348096
184800	0.0250309	0.0370546	0.0260417	0.0366994
192400	0.0263989	0.0384283	0.021875	0.0382026
200000	0.0275581	0.0397135	0.0260417	0.0401513

Se debe tener en cuenta que las gráficas son lineales cuando en realidad deberían tener la forma de $nlog(n)$. La razón es que en la aplicación usada para obtener las gráficas no existía el tipo que buscábamos y el tipo lineal era el que más se acercaba a lo pedido.

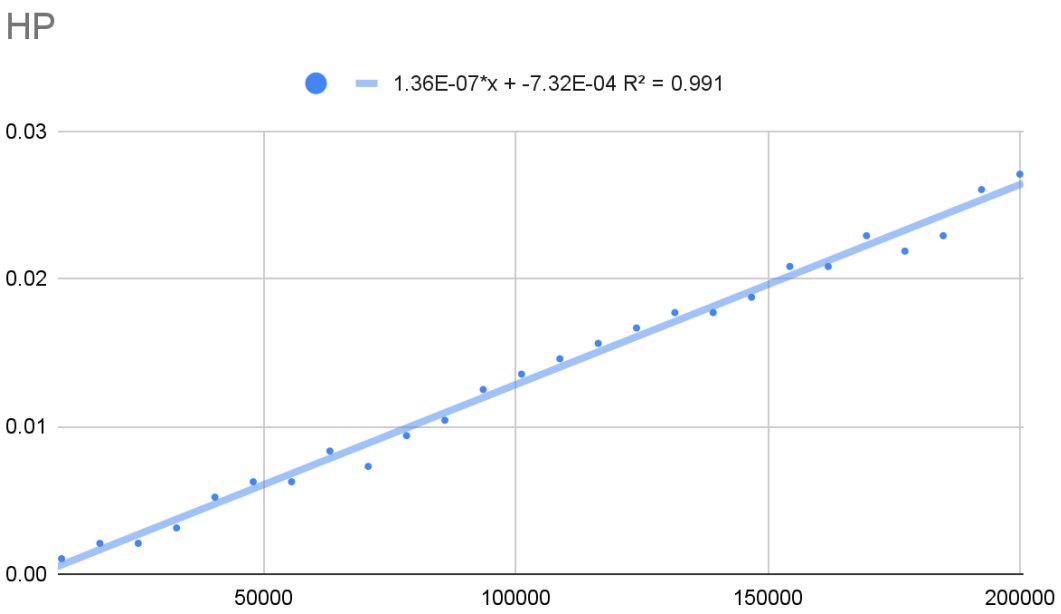
- Gráfica eficiencia Lenovo:



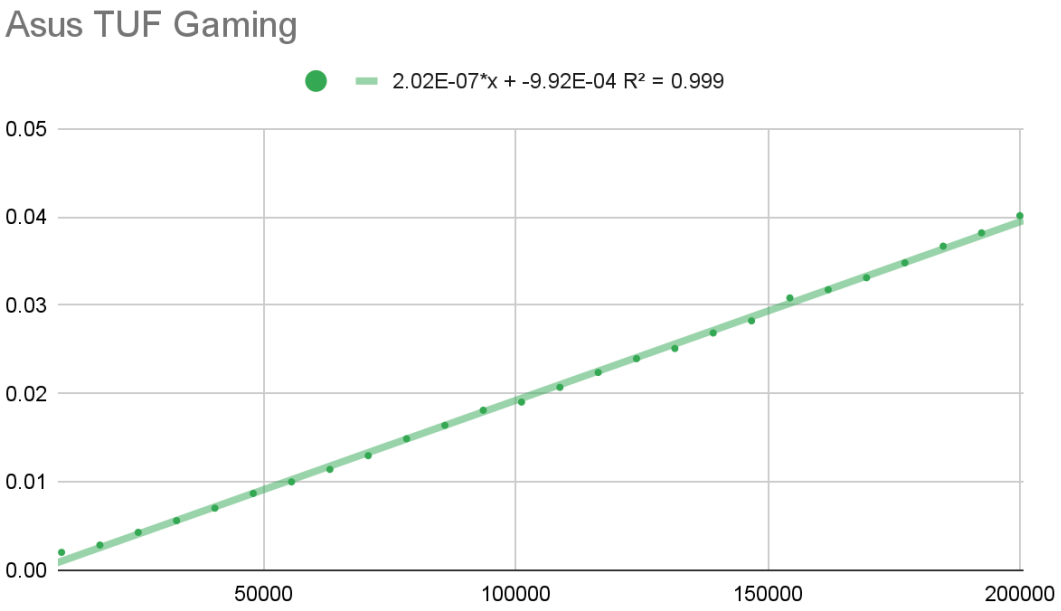
- Gráfica eficiencia Asus ZenBook:



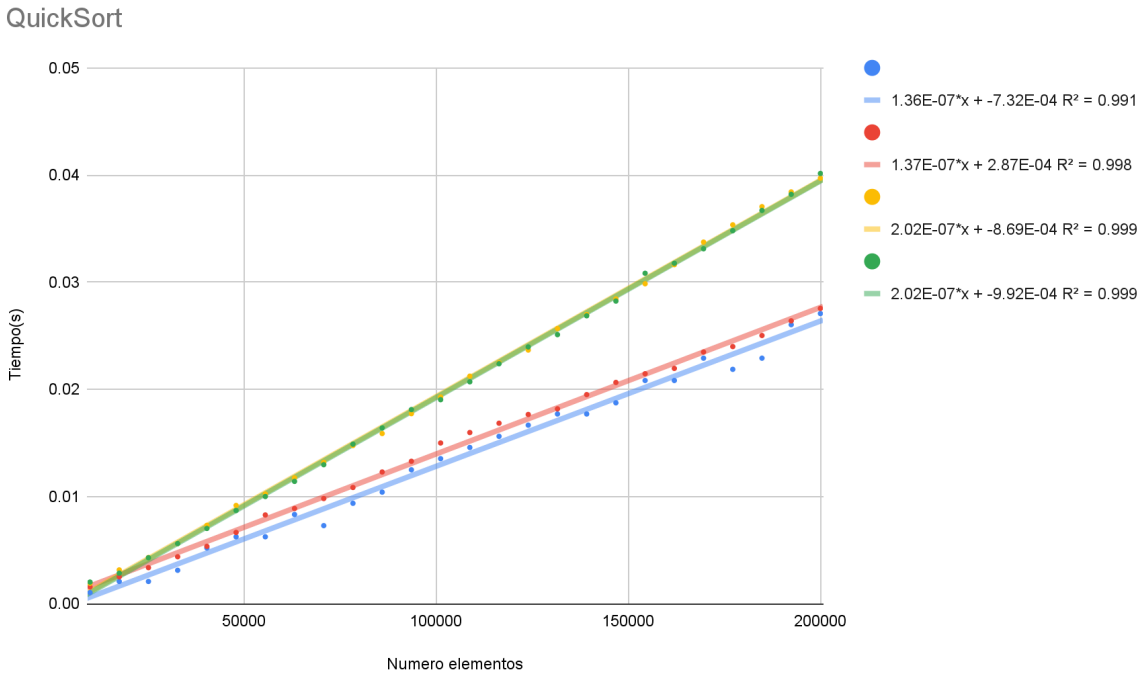
❖ Gráfica eficiencia HP:



❖ Gráfica eficiencia en Asus TUF Gaming:



❖ Gráfica con las cuatro eficiencias empíricas:



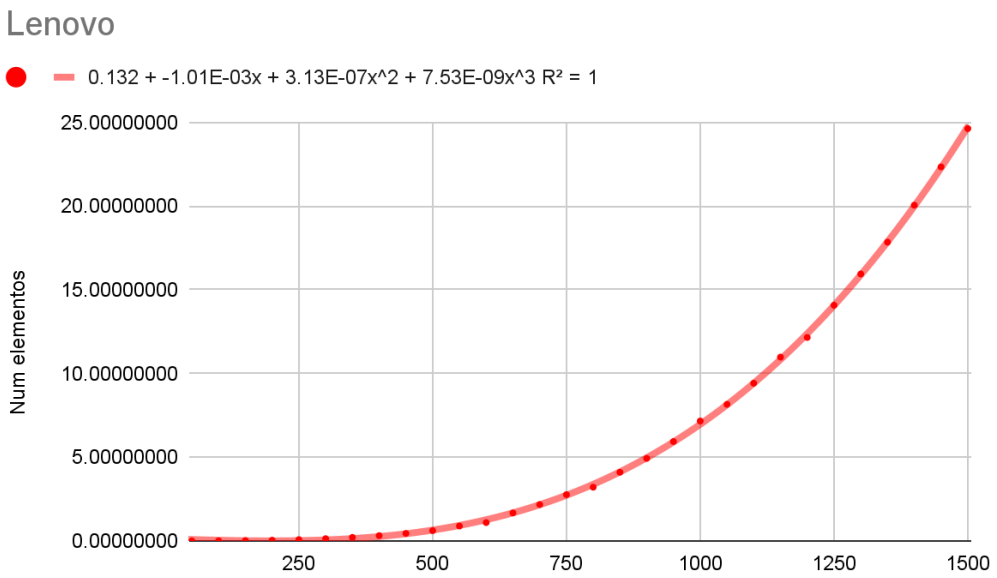
➢ Eficiencia de Floyd

Tabla de resultados de cada equipo:

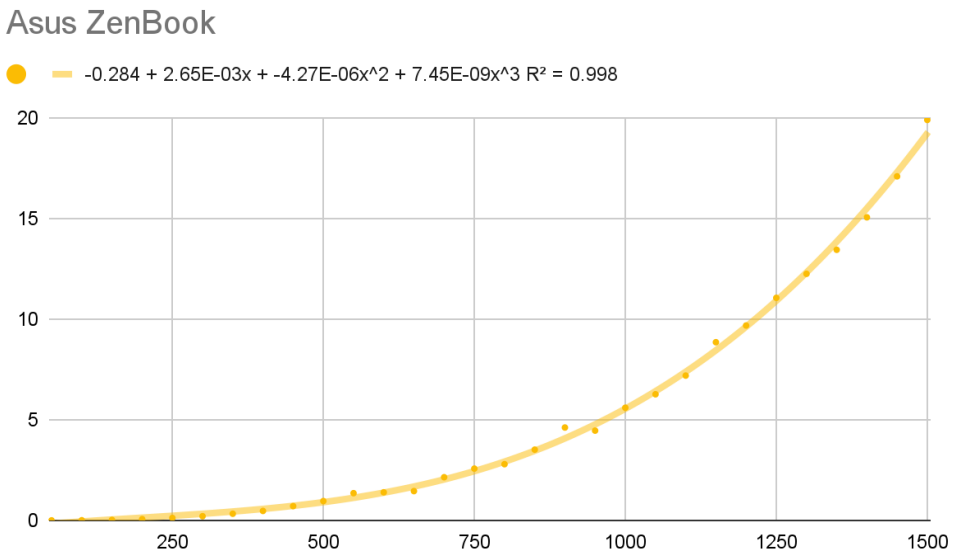
	Tiempo en segundos			
Tamaño	Lenovo Linux	Asus ZB Linux	HP Windows	Asus TUF Linux
50	0.000983	0.00145553	0.00104167	0.00106167
100	0.00620393	0.00784107	0.00416667	0.00471433
150	0.0201199	0.0267224	0.015625	0.0176391
200	0.0428373	0.0618765	0.040625	0.0376794
250	0.0784883	0.121865	0.0729167	0.0838178
300	0.13425	0.208489	0.127083	0.138684
350	0.211717	0.330932	0.202083	0.216805
400	0.314661	0.470496	0.421875	0.313732
450	0.447027	0.709409	0.722917	0.465873
500	0.612567	0.961088	1.08958	0.593325
550	0.889038	1.35316	0.761458	0.86355
600	1.09479	1.39438	0.969792	1.03555
650	1.67198	1.45497	1.21667	1.41451
700	2.17202	2.14428	1.52083	1.74663
750	2.76527	2.57215	1.88021	2.15129
800	3.20774	2.79175	2.27813	2.57763
850	4.09932	3.51353	2.75729	3.14067
900	4.92843	4.61217	3.41771	3.73609
950	5.93616	4.45798	3.89062	4.35527
1000	7.16106	5.59502	4.4625	5.2905
1050	8.15617	6.26307	5.17396	6.47196
1100	9.41535	7.19097	6.89479	7.19319
1150	10.9759	8.8481	7.76667	8.22589
1200	12.1508	9.67742	10.7677	9.13732
1250	14.0746	11.0512	12.3208	9.82922
1300	15.943	12.2448	12.2781	11.1545

1350	17.8421	13.4371	12.0969	12.5229
1400	20.0514	15.0495	13.4	13.9493
1450	22.3379	17.0835	13.8615	15.4988
1500	24.6255	19.8862	15.3052	17.105

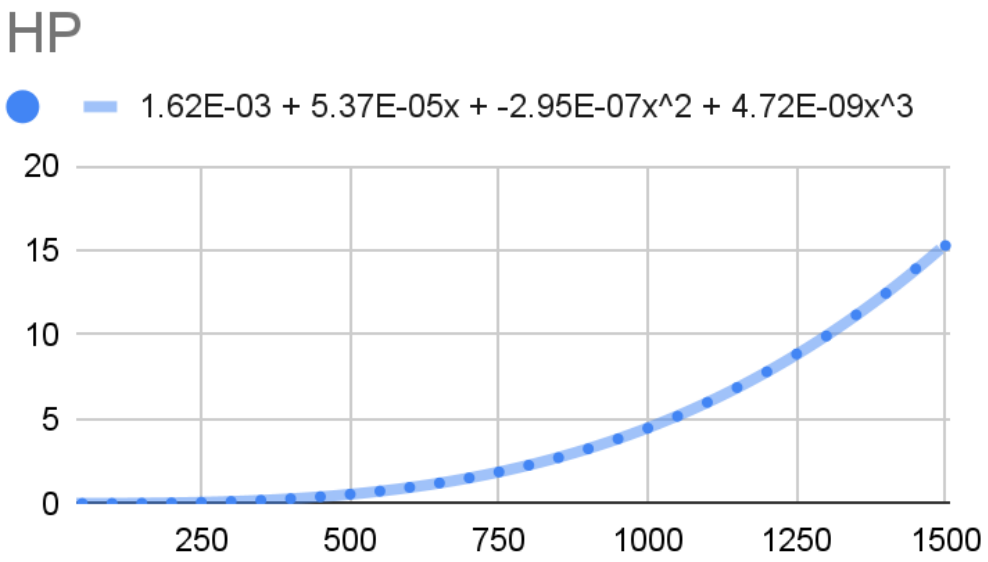
❖ Gráfica eficiencia Lenovo:



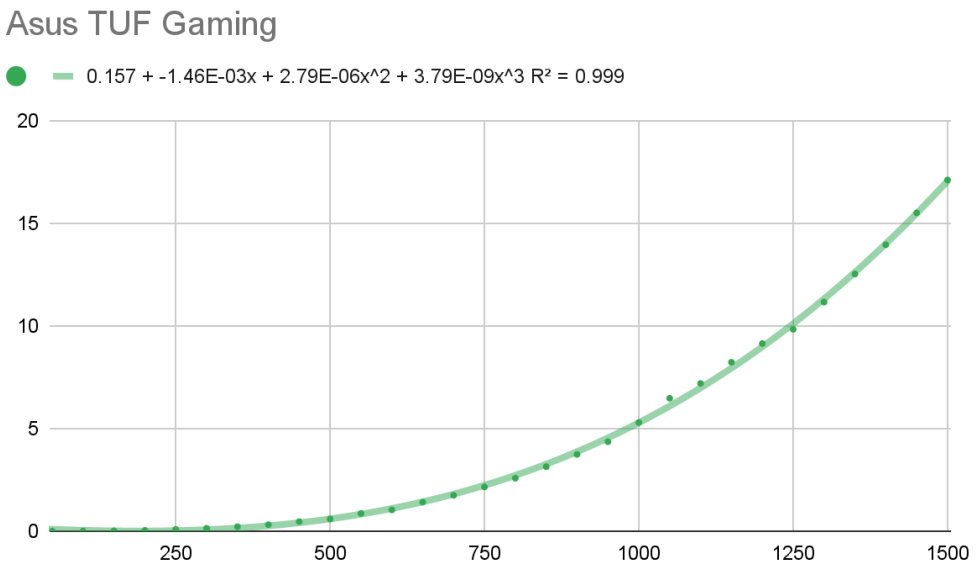
❖ Gráfica eficiencia Asus ZenBook:



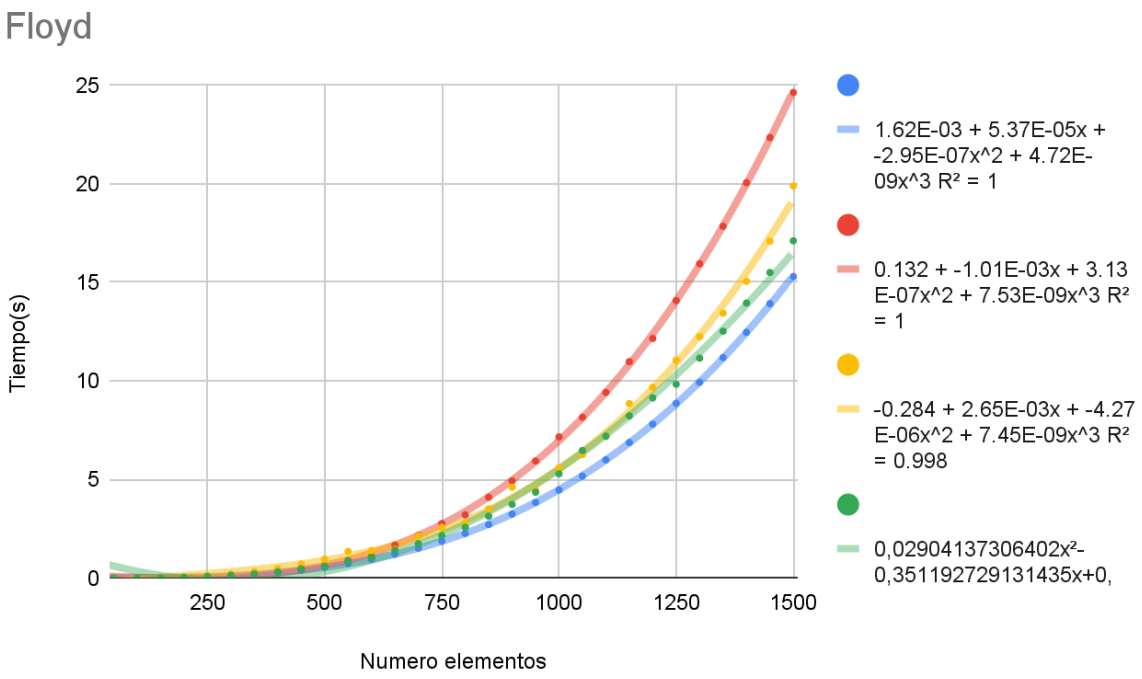
❖ Gráfica eficiencia HP:



❖ Gráfica eficiencia en Asus TUF Gaming:



❖ Gráfica con las cuatro eficiencias empíricas



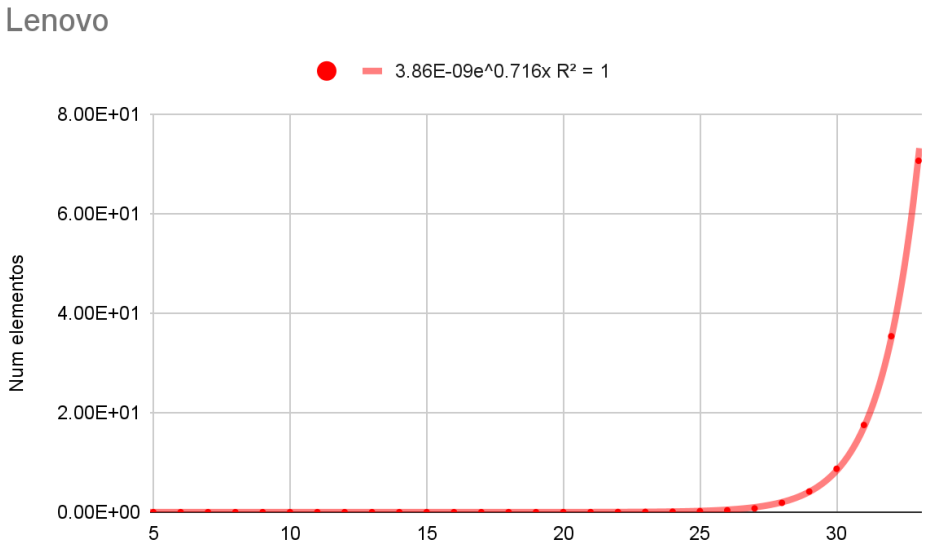
➢ Eficiencia de Hanoi

Tabla de resultados en cada equipo:

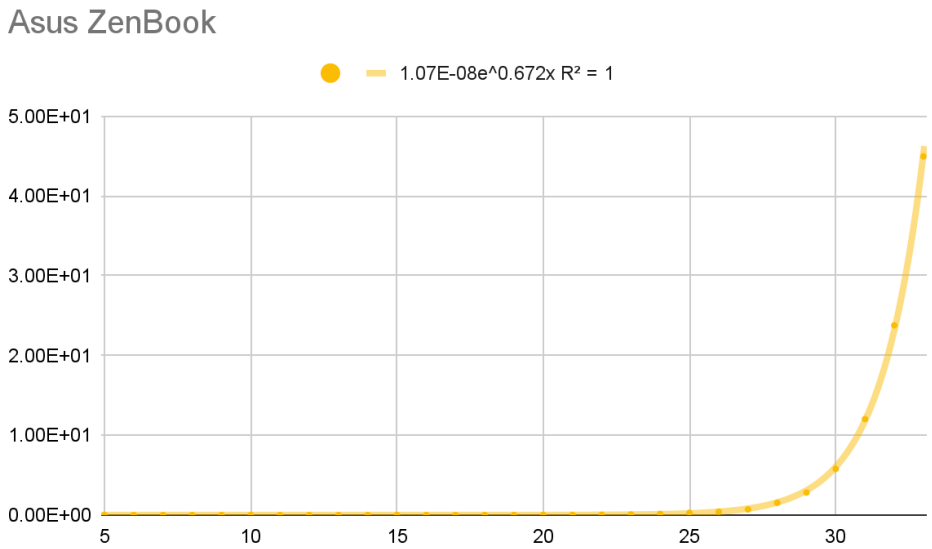
	Tiempo en segundos			
Tamaño	Lenovo Linux	Asus ZB Linux	HP Windows	Asus TUF Linux
5	1.13E-06	1.13333E-06	0	1.06667e-06
6	1.33E-06	1.93333E-06	0	1.53333e-06
7	2.07E-06	2.06667E-06	0	2.46667e-06
8	2.87E-06	3.86667E-06	0	4.13333e-06
9	5.07E-06	6.8E-06	0	6.66667e-06
10	9.20E-06	9.13333E-06	0	1.31333e-05
11	1.77E-05	2.71333E-05	0	2.6e-05
12	3.43E-05	5.45333E-05	0	4.94667e-05
13	6.91E-05	9.95333E-05	0.00104167	9.83333e-05
14	0.000136	0.000197667	0	0.000139067
15	0.000262867	0.000278	0.00104167	0.000426933
16	0.000522067	0.0005576	0	0.000593267
17	0.00102987	0.00110927	0	0.00111153
18	0.00185967	0.00247047	0.00104167	0.00251093
19	0.0034374	0.00444907	0.003125	0.00444873
20	0.00668567	0.00886113	0.00520833	0.0088814
21	0.0142907	0.0177522	0.009375	0.0177561
22	0.0270099	0.0354418	0.01875	0.0356909
23	0.0509185	0.0710407	0.0375	0.0710172
24	0.095712	0.141778	0.0739583	0.142126
25	0.202304	0.284003	0.147917	0.28437
26	0.375161	0.424577	0.295833	0.432404
27	0.742514	0.703119	0.588542	0.712361
28	1.86142	1.53708	1.18542	1.42787
29	4.14526	2.81984	2.36979	3.16698
30	8.71095	5.78281	4.73229	6.01432

31	17.5069	12.0255	9.43958	11.6163
32	35.3444	23.7847	18.8656	23.5292
33	70.5962	44.9584	37.7885	46.6829

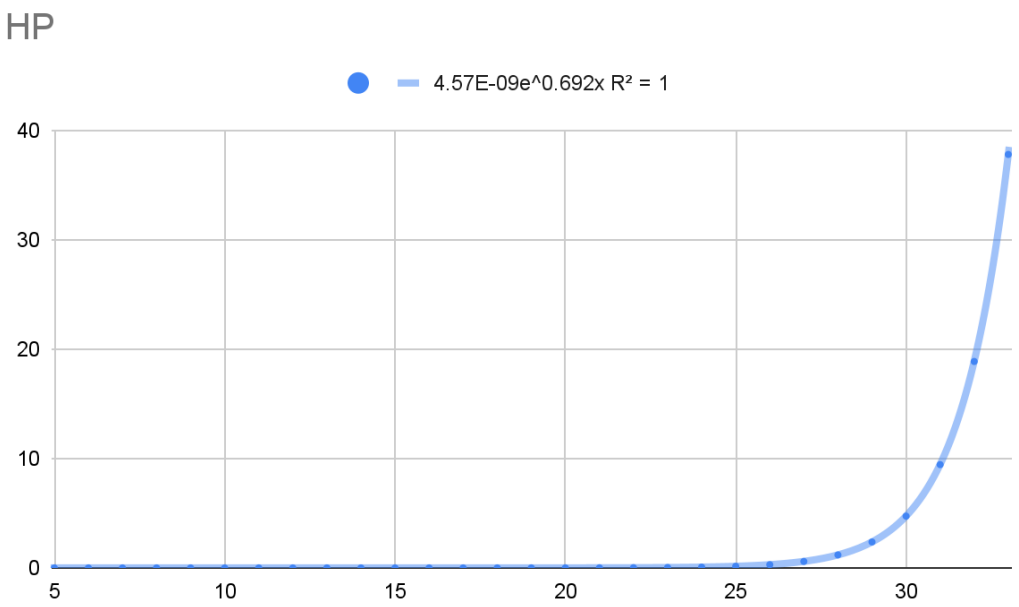
❖ Gráfica eficiencia Lenovo:



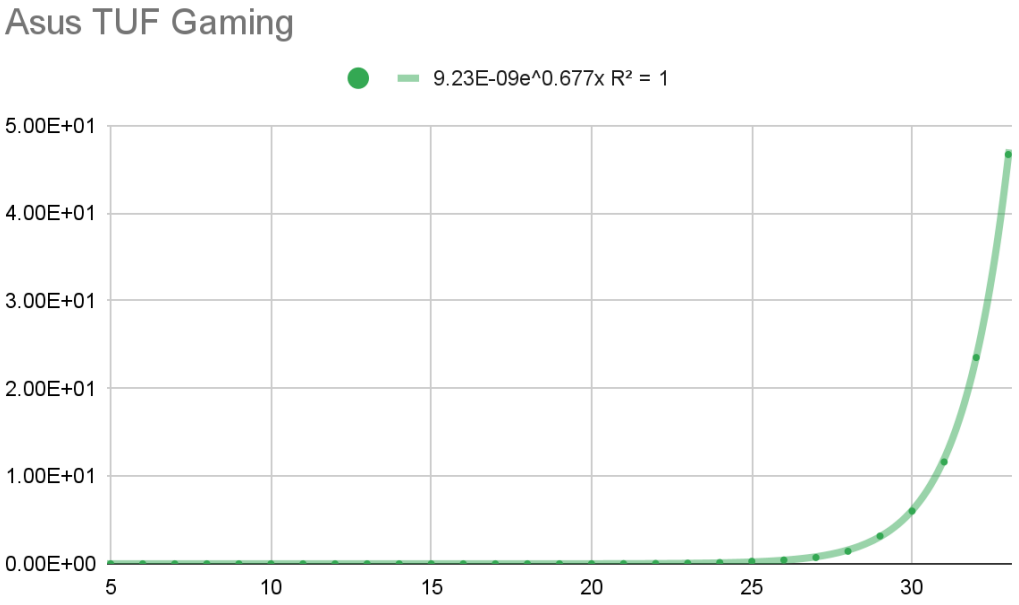
❖ Gráfica eficiencia Asus ZenBook:



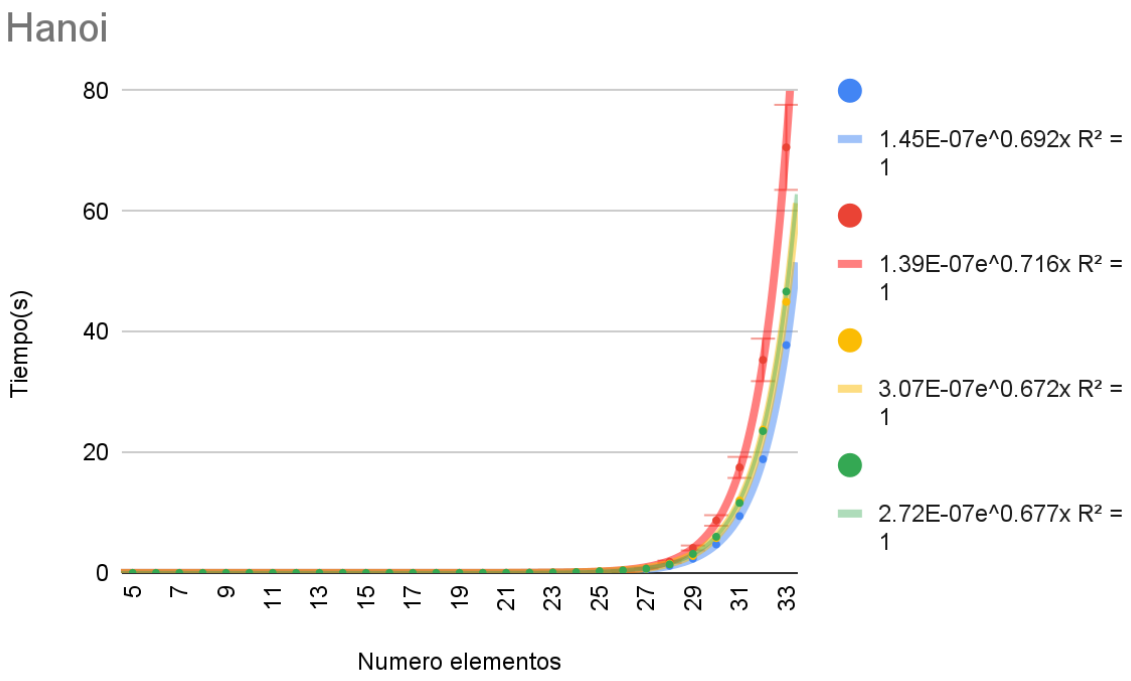
❖ Gráfica eficiencia HP:



❖ Gráfica eficiencia Asus TUF Gaming:



❖ Gráfica con las cuatro eficiencias empíricas:



➔ Estudio de variación

Para ver las diferencias del tiempo de ejecución, hemos cambiado una serie de condiciones. Hemos probado con diferentes optimizaciones y con diferentes sistemas operativos.

Optimizaciones

Para ver un ejemplo de las diferencias entre la ejecución con optimización y sin optimización, hemos ejecutado el código de Hanoi compilado de las dos siguientes formas:

- Sin optimización: g++ hanoi.cpp -o hanoi -std=gnu++0x
- Con optimización: g++ -Og hanoi.cpp -o hanoi -std=gnu++0x

Tamaño	Con optimización	Sin optimización
5	1.00E-06	1.13E-06
6	1.07E-06	1.33E-06
7	1.93E-06	2.07E-06
8	2.40E-06	2.87E-06
9	4.20E-06	5.07E-06
10	7.07E-06	9.20E-06
11	1.73E-05	1.77E-05
12	3.01E-05	3.43E-05
13	5.85E-05	6.91E-05
14	0.000116933	0.000136
15	0.0002308	0.000262867
16	0.0004634	0.000522067
17	0.000620533	0.00102987
18	0.000953867	0.00185967
19	0.00178527	0.0034374
20	0.00348447	0.00668567
21	0.00686473	0.0142907
22	0.0136527	0.0270099
23	0.0269057	0.0509185
24	0.0524056	0.095712
25	0.209438	0.202304
26	0.305327	0.375161
27	0.624765	0.742514
28	1.4049	1.86142
29	3.04428	4.14526

30	6.40463	8.71095
31	12.7555	17.5069
32	25.3154	35.3444
33	52.4056	70.5962

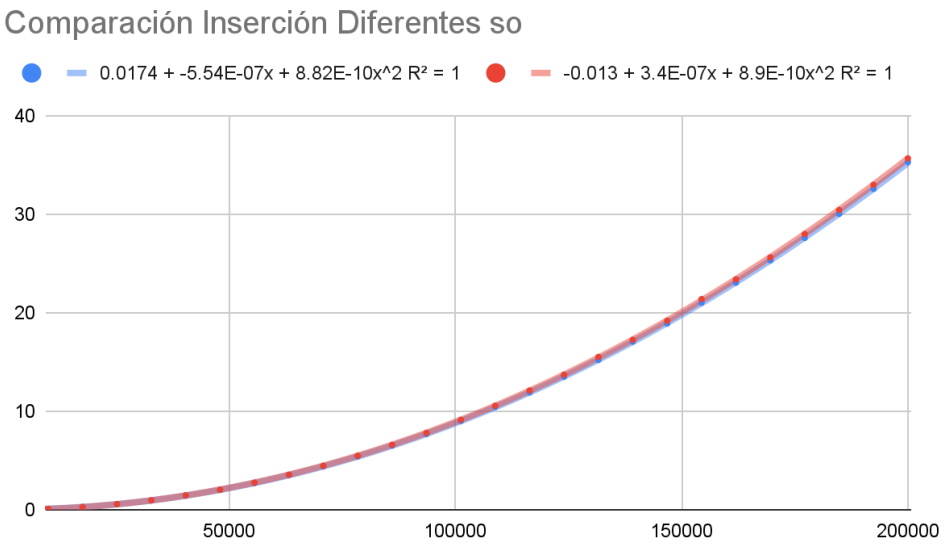
Como bien se puede observar en la gráfica, el tiempo de ejecución se reduce si compilamos con optimización. Por tanto, la forma de compilar es algo que se debe tener en cuenta a la hora de ejecutar un programa.

➔ Comparativa en distintos sistemas operativos

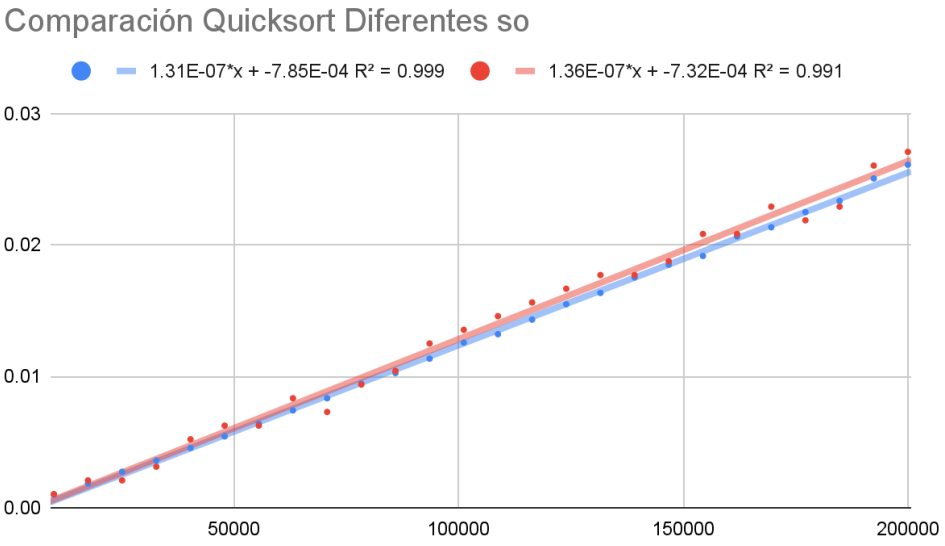
Hemos probado también con diferentes sistemas operativos.En particular, hemos ejecutado los cuatro algoritmos en **Linux** y en **Windows**.

Con el portátil HP Pavilion hemos ejecutado primero los algoritmos de Inserción, QuickSort, Hanoi y FLoyd en Windows 10. Para ello, hemos instalado la terminal Ubuntu para Windows y hemos ejecutado los códigos bajo las mismas condiciones en ambos sistemas operativos. Es decir, con todas las aplicaciones cerradas, el brillo al mínimo y el ordenador cargando. Los resultados han sido los siguientes (la función roja representa Ubuntu y la azul, Windows):

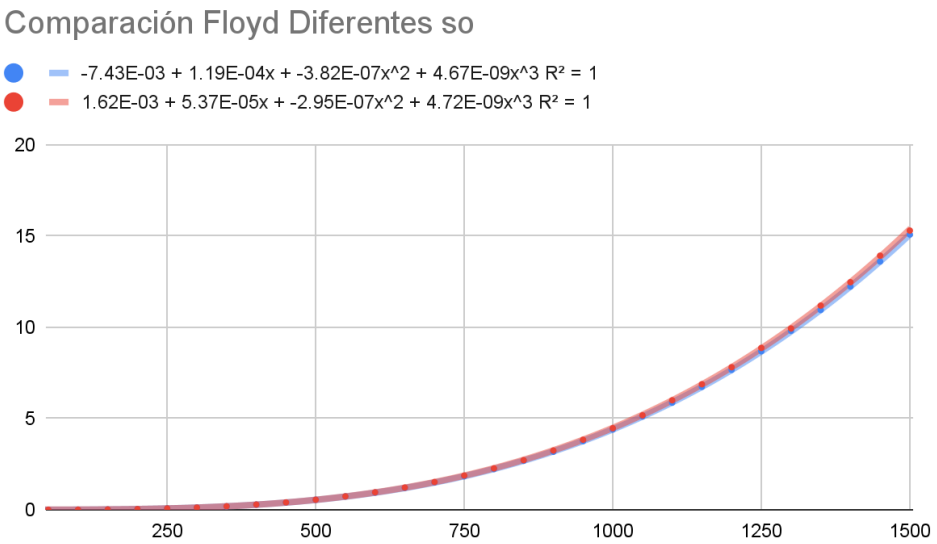
- Algoritmo Inserción:



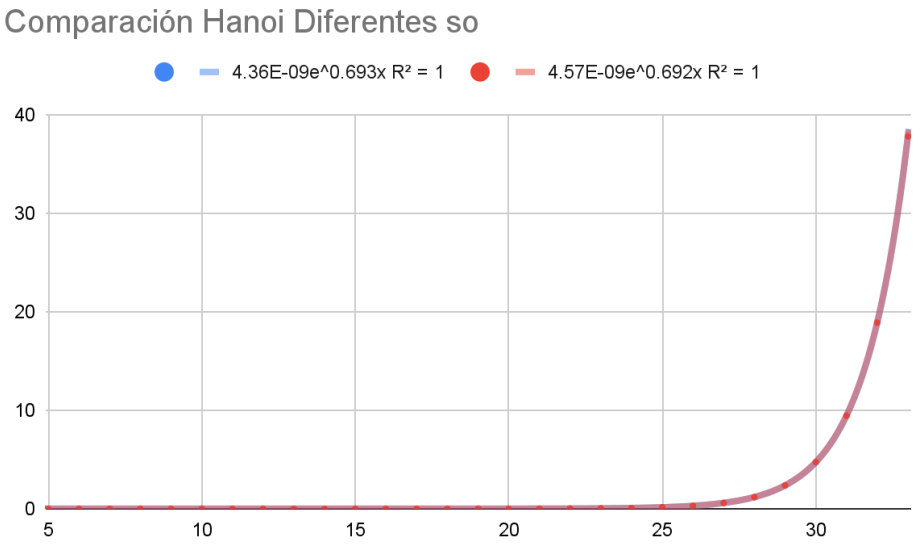
- Algoritmo Quicksort:



- Algoritmo Floyd:



- Algoritmo Hanoi:



Como podemos observar en los gráficos, apenas se ha obtenido diferencia al ejecutar los programas en los distintos sistemas operativos. Podemos asegurar entonces, que el tiempo de ejecución de los programas no depende del sistema operativo en el que se ejecute, si no de las propiedades del ordenador.

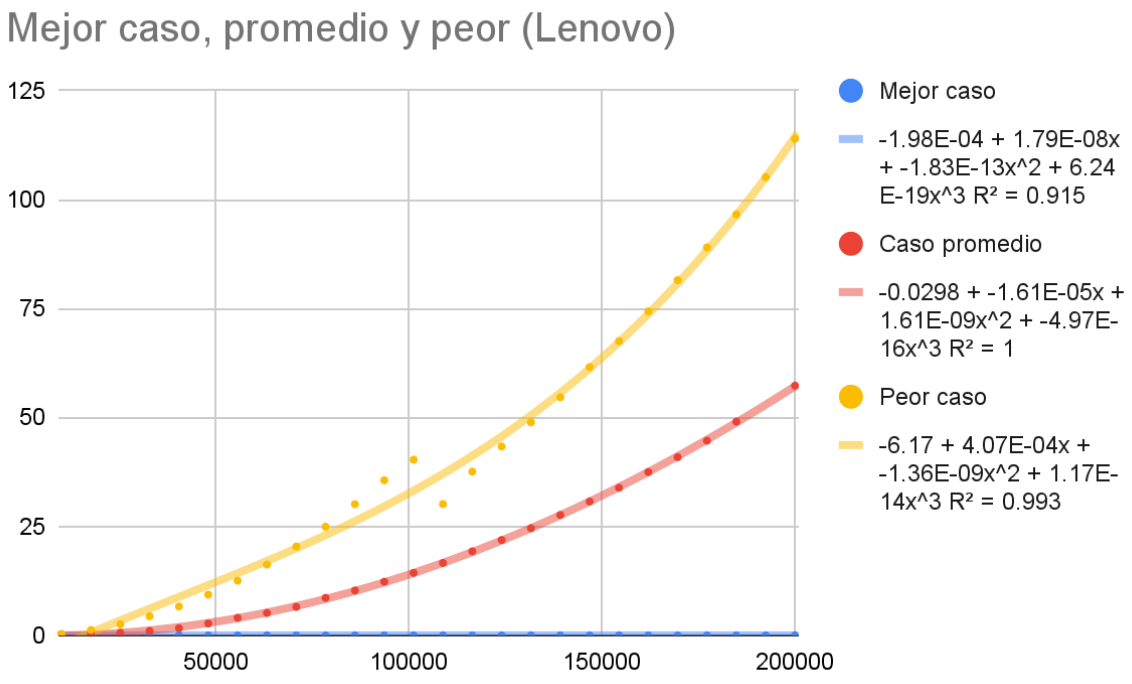
➔ Estudio de casos para Inserción y Selección

En este apartado analizamos el peor caso, el caso promedio y el mejor de los algoritmos de inserción y selección. El **caso mejor** se refiere a la situación de los datos que genera una ejecución del algoritmo con la menor complejidad computacional y, por tanto, mejor tiempo. Mientras que el **caso peor** hace referencia a los datos iniciales que producen una ejecución del algoritmo con la mayor complejidad computacional, es decir, peor tiempo. Por otro lado, el **caso promedio** es aquella situación inicial que no sigue ningún patrón preestablecido que aporte ventajas o desventajas, por lo que se puede considerar la situación típica de ejecución del algoritmo.

En concreto, para los **algoritmos de inserción y selección** que tratan de ordenar una lista de números, el peor caso se da cuando el vector está ordenado en el orden inverso; el mejor caso, cuando está ordenado correctamente; y el caso promedio, en el resto de posibilidades de colocación de los números.

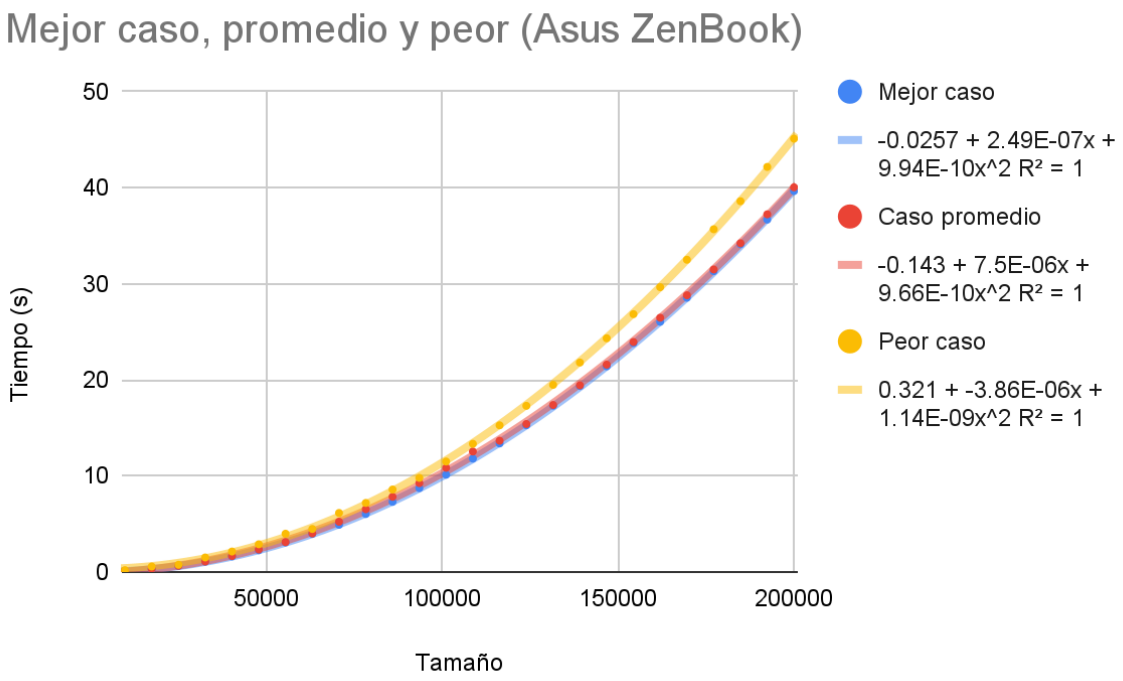
A continuación mostramos los resultados obtenidos en las cada una de las ejecuciones:

➔ Inserción:



Siendo la función roja el caso promedio, podemos observar cómo el peor caso (amarillo) y el mejor caso (azul) nos dan una cota del caso promedio (rojo), es decir, siempre podemos asegurar que el promedio esté entre los dos valores del peor y mejor caso. Podemos considerar que se obtienen tiempos considerablemente diferentes según los valores iniciales del vector.

➔ Selección:



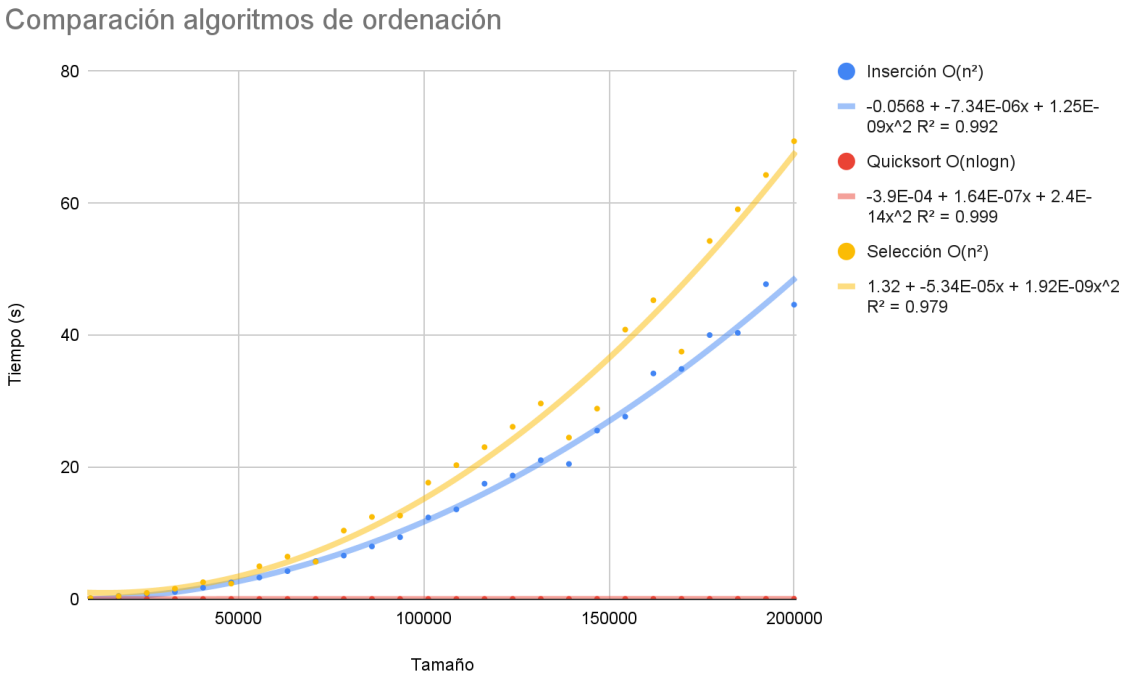
De la misma forma vemos cómo en el algoritmo de selección el peor caso (amarillo) y el mejor caso (azul) también nos dan una cota del caso promedio (rojo). No obstante, ahora la diferencia entre los tres casos considerados es mucho menor, es decir, el algoritmo de selección no varía tanto en función de los datos de entrada, como sí lo hacía el de inserción.

➔ Comparativa algoritmos de ordenación

A lo largo de esta memoria se han estudiado varios algoritmos de ordenación: **inserción O(n²), selección O(n²) y quicksort O(nlogn)**, por lo que tiene sentido compararlos todos ellos en una misma gráfica para ver visualmente cuáles son mejores según los valores de entrada. Como hemos cogido los mismos tamaños para todos ellos, la comparativa es coherente.

	Tiempo en segundos		
Tamaño	Inserción	Selección	Quicksort
10000	0.125607325	0.144394	0.00163885
17600	0.463784	0.4486035	0.002654415
25200	0.74424	0.9260465	0.00351655
32800	1.08421425	1.571175	0.0046889175
40400	1.7182775	2.55704	0.0062292475
48000	2.524945	2.34953	0.0076957675
55600	3.2634325	4.97394	0.0086974
63200	4.23273	6.43622	0.010097515
70800	5.75814	5.66458	0.0108217
78400	6.5931225	10.378495	0.0124715
86000	7.9792925	12.450385	0.013751225
93600	9.3803825	12.65034	0.0154141
101200	12.37045	17.6501	0.01671955
108800	13.5951	20.3027	0.018127675
116400	17.490025	23.03035	0.01934025
124000	18.7266	26.11885	0.020491325
131600	21.049525	29.6522	0.02166985
139200	20.48255	24.484	0.02276945
146800	25.544525	28.8686	0.024021725
154400	27.658375	40.8477	0.025748775
162000	34.2024	45.3006	0.02655115
169600	34.883375	37.51455	0.028319325

177200	40.023725	54.28605	0.029010175
184800	40.3525	59.08575	0.0304254
192400	47.737575	64.29145	0.032267875
200000	44.629	69.4135	0.03362655



Por un lado, en cuanto a los **órdenes de eficiencia**, vemos cómo los algoritmos de inserción y selección se corresponden con una función cuadrática y el de quicksort, con una logarítmica, de ahí que esta última parezca prácticamente constante en comparación con las otras dos. Es decir, se ve cómo claramente los órdenes logarítmicos son mejores que los cuadráticos.

Por otro lado, observamos cómo el algoritmo de selección es ligeramente peor que el de inserción, excepto algunos datos puntuales de menor tamaño, aunque tienen el mismo orden de eficiencia. Aquí la diferencia la marcan las **constantes ocultas**, las cuales se pueden ver en la aproximación por mínimos cuadrados del excel.

En resumen, de menor a mayor tiempo de ejecución tenemos: **quicksort < inserción < selección.**

5. Eficiencia híbrida

Mientras que el cálculo de la eficiencia teórica nos da una expresión general de la eficiencia, hay veces que necesitamos una función más exacta, con las **constantes ocultas**. Para hallarlas, hemos utilizado los datos de la salida con **gnuplot**. Primero definimos la función del orden correspondiente a cada algoritmo y hacemos la regresión con gnuplot.

➢ Eficiencia de Inserción

Ejecutamos gnuplot y ejecutamos los siguientes comandos:

- **gnuplot> f(x)=a0*x*x+a1*x+a2**
- **gnuplot> fit f(x) 'salida.dat' via a0,a1,a2**
- **gnuplot> plot 'salida.dat', f(x) title 'Curva ajustada'**

Con esto podemos ver cuál sería el tiempo de ejecución para un n elevado y que sería poco rentable hacerlo con la eficiencia empírica ya que en nuestros ordenadores tardaría bastante. Algunos puntos notables más elevados:

- f(500000):
  - Lenovo: f(500000)= 370 segundos = 6 minutos y 10 segundos
  - Asus ZenBook: f(500000)= 371 segundos = 6 minutos y 11 segundos
  - HP: f(500000)= 223 segundos = 3 minutos y 43 segundos
  - Asus TUF Gaming: f(500000) = 378 segundos = 6 minutos y 18 segundos
- f(1000000):
  - Lenovo: f(1000000)= 1500 segundos=25 minutos
  - Asus ZenBook: f(1000000)= 1496 s = 24 minutos y 56 segundos
  - HP: f(1000000)= 890 segundos = 14 minutos y 50 segundos
  - Asus TUF Gaming: f(1000000)= 1561 s = 26 min y 1 s

Vemos como para el valor de un millón ya se iría a 25 minutos. Pero como nosotros hacemos la media de 15 ejecuciones para cada n, sería ejecutarlo quince veces para un millón de elementos. Por tanto, para solo ese punto tardaría aproximadamente (ya que sería una media)

25 mins x 15 veces = 6 horas y 15 minutos

- Lenovo:

```
iter      chisq       delta/lim   lambda     a0             a1             a2
   0 1.8960615094e+22  0.00e+00  1.08e+10   1.000000e+00   1.000000e+00   1.000000e+00
   1 7.1363680845e+17 -2.66e+09  1.08e+09   6.128852e-03   9.999939e-01   1.000000e+00
   2 5.0537458030e+10 -1.41e+12  1.08e+08  -5.776170e-06   9.999938e-01   1.000000e+00
   3 4.7818165222e+10 -5.69e+03  1.08e+07  -6.154825e-06   9.999897e-01   1.000000e+00
   4 4.7779116868e+10 -8.17e+01  1.08e+06  -6.152311e-06   9.995813e-01   1.000000e+00
   5 4.4101930281e+10 -8.34e+03  1.08e+05  -5.910762e-06   9.603452e-01   9.999992e-01
   6 1.7052543867e+09 -2.49e+06  1.08e+04  -1.161012e-06   1.888176e-01   9.999841e-01
   7 1.0183089691e+04 -1.67e+10  1.08e+03  -1.264782e-09   4.336183e-04   9.999804e-01
   8 1.6531904414e+01 -6.15e+07  1.08e+02   1.573834e-09  -2.747343e-05   9.999771e-01
   9 1.6529398748e+01 -1.52e+01  1.08e+01   1.573877e-09  -2.747831e-05   9.996503e-01
  10 1.6293575473e+01 -1.45e+03  1.08e+00   1.571353e-09  -2.686191e-05   9.682193e-01
  11 1.3489914415e+01 -2.08e+04  1.08e-01   1.520422e-09  -1.442497e-05   3.340379e-01
  12 1.3370932088e+01 -8.90e+02  1.08e-02   1.507581e-09  -1.128921e-05   1.741398e-01
  13 1.3370931332e+01 -5.66e-03  1.08e-03   1.507548e-09  -1.128128e-05   1.737356e-01
iter      chisq       delta/lim   lambda     a0             a1             a2

After 13 iterations the fit converged.
final sum of squares of residuals : 13.3709
rel. change during last iteration : -5.65696e-08

degrees of freedom    (FIT_NDF)                       : 51
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 0.51203
variance of residuals (reduced chisquare) = WSSR/ndf   : 0.262175

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a0              = 1.50755e-09        +/- 2.422e-11    (1.607%)
a1              = -1.12813e-05       +/- 5.221e-06    (46.28%)
a2              = 0.173736           +/- 0.238        (137%)

correlation matrix of the fit parameters:
               a0      a1      a2
a0             1.000
a1            -0.972  1.000  ,
a2             0.789 -0.894  1.000
```

- Asus ZenBook:

```
iter      chisq       delta/lim  lambda   a0             a1             a2
   0 9.2414297479e+21   0.00e+00  1.09e+10    1.000000e+00   1.000000e+00   1.000000e+00
   1 1.4807610787e+18  -6.24e+08  1.09e+09    1.265217e-02   9.999939e-01   1.000000e+00
   2 4.7481628687e+10  -3.12e+12  1.09e+08   -4.512498e-06   9.999938e-01   1.000000e+00
   3 2.3149167359e+10  -1.05e+05  1.09e+07   -6.135135e-06   9.999919e-01   1.000000e+00
   4 2.3140123985e+10  -3.91e+01  1.09e+06   -6.133939e-06   9.997965e-01   1.000000e+00
   5 2.2261702595e+10  -3.95e+03  1.09e+05   -6.016364e-06   9.806361e-01   9.999996e-01
   6 2.5514085850e+09  -7.73e+05  1.09e+04   -2.035992e-06   3.319769e-01   9.999869e-01
   7 6.6507124394e+04  -3.84e+09  1.09e+03   -9.176727e-09   1.678073e-03   9.999804e-01
   8 3.5237842558e+02  -1.88e+07  1.09e+02    1.196158e-09  -1.233875e-05   9.999770e-01
   9 3.5237550967e+02  -8.28e-01  1.09e+01    1.196662e-09  -1.241861e-05   9.996368e-01
iter      chisq       delta/lim  lambda   a0             a1             a2

After 9 iterations the fit converged.
final sum of squares of residuals : 352.376
rel. change during last iteration : -8.27501e-06

degrees of freedom    (FIT_NDF)                        : 23
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 3.91416
variance of residuals (reduced chisquare) = WSSR/ndf   : 15.3207

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a0                 = 1.19666e-09   +/- 2.647e-10   (22.12%)
a1                 = -1.24186e-05  +/- 5.72e-05    (460.6%)
a2                 = 0.999637      +/- 2.613       (261.4%)

correlation matrix of the fit parameters:
               a0      a1      a2
a0              1.000
a1             -0.972   1.000
a2              0.788  -0.893   1.000
```

- HP

```
iter      chisq       delta/lim  lambda   a0            a1            a2
   0 9.2414297529e+21   0.00e+00  1.09e+10    1.000000e+00   1.000000e+00   1.000000e+00
   1 1.4807610795e+18  -6.24e+08  1.09e+09    1.265217e-02   9.999939e-01   1.000000e+00
   2 4.7481954423e+10  -3.12e+12  1.09e+08   -4.512767e-06   9.999938e-01   1.000000e+00
   3 2.3149493097e+10  -1.05e+05  1.09e+07   -6.135405e-06   9.999919e-01   1.000000e+00
   4 2.3140449595e+10  -3.91e+01  1.09e+06   -6.134208e-06   9.997965e-01   1.000000e+00
   5 2.2262015831e+10  -3.95e+03  1.09e+05   -6.016633e-06   9.806360e-01   9.999996e-01
   6 2.5514441751e+09  -7.73e+05  1.09e+04   -2.036233e-06   3.319722e-01   9.999869e-01
   7 6.6158007284e+04  -3.86e+09  1.09e+03   -9.402947e-09   1.671041e-03   9.999805e-01
   8 2.3294527670e+00  -2.84e+09  1.09e+02    9.700120e-10  -1.938198e-05   9.999785e-01
   9 2.3284073562e+00  -4.49e+01  1.09e+01    9.705276e-10  -1.946475e-05   9.997867e-01
  10 2.2436187181e+00  -3.78e+03  1.09e+00    9.690245e-10  -1.909659e-05   9.809594e-01
  11 2.9116896788e-01  -6.71e+05  1.09e-01    9.170907e-10  -6.375870e-06   3.304365e-01
  12 2.6482028737e-02  -9.99e+05  1.09e-02    8.898171e-10   3.045690e-07  -1.119344e-02
  13 2.6474728599e-02  -2.76e+01  1.09e-03    8.896731e-10   3.398355e-07  -1.299693e-02
  14 2.6474728599e-02  -7.85e-08  1.09e-04    8.896731e-10   3.398373e-07  -1.299703e-02
iter      chisq       delta/lim  lambda   a0            a1            a2

After 14 iterations the fit converged.
final sum of squares of residuals : 0.0264747
rel. change during last iteration : -7.84712e-13

degrees of freedom    (FIT_NDF)                       : 23
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 0.0339275
variance of residuals (reduced chisquare) = WSSR/ndf   : 0.00115108

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a0              = 8.89673e-10      +/- 2.295e-12    (0.2579%)
a1              = 3.39837e-07      +/- 4.958e-07    (145.9%)
a2              = -0.012997        +/- 0.02265      (174.3%)

correlation matrix of the fit parameters:
               a0     a1     a2
a0              1.000
a1             -0.972  1.000
a2              0.788 -0.893  1.000
```

- ## Asus TUF Gaming:

```
iter      chisq     delta/lim  lambda   a0             a1         a2
   0 9.2414297447e+21   0.00e+00  1.09e+10    1.000000e+00   1.000000e+00   1.000000e+00
   9 1.2447988935e+03  -1.53e-02  1.09e+01    1.611879e-09  -5.175122e-05   9.999532e-01

After 9 iterations the fit converged.
final sum of squares of residuals : 1244.8
rel. change during last iteration : -1.53046e-07

degrees of freedom    (FIT_NDF)                       : 23
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf): 7.35674
variance of residuals (reduced chisquare) = WSSR/ndf   : 54.1217

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a0                 = 1.61188e-09   +/- 4.976e-10   (30.87%)
a1                 = -5.17512e-05  +/- 0.0001075   (207.8%)
a2                 = 0.999953      +/- 4.911       (491.1%)

correlation matrix of the fit parameters:
               a0      a1      a2
a0             1.000
a1            -0.972   1.000
a2             0.788  -0.893   1.000
```

➢ Eficiencia de Quicksort

Ejecutamos gnuplot y ejecutamos los siguientes comandos:

- **gnuplot> f(x)=a0*x*log(a1*x)**
- **gnuplot> fit f(x) 'salida.dat' via a0,a1**
- **gnuplot> plot 'salida.dat', f(x) title 'Curva ajustada'**

Con esto podemos ver cuál sería el tiempo de ejecución para un n elevado y que sería poco rentable hacerlo con la eficiencia empírica ya que en nuestros ordenadores tardaría bastante. Algunos puntos notables más elevados:

- f(1000000):
  - Lenovo: f(1000000)= 0.3 s
  - Asus ZenBook: f(1000000)= 0.0987 s
  - HP: f(1000000)=0.1495 s
  - Asus TUF Gaming: f(1000000)= 0.0984 s
- f(400000000):
  - Lenovo: f(400000000)= 180 segundos= 3 minutos
  - Asus ZenBook: f(400000000)= 56 s
  - HP: f(400000000)= 1 min y 25 s
  - Asus TUF Gaming: f(400000000)= 57 s

❖ Lenovo:

```
Final set of parameters
=======================
a0              = 2.27898e-08
a1              = 0.937319
```

❖ HP:

```
Final set of parameters
=======================
a0              = 1.08913e-08
a1              = 0.918137
```

❖ Asus ZenBook:

```
Final set of parameters
=======================
a0              = 1.65598e-08
a1              = 0.918137
```

❖ Asus TUF Gaming:

```
Final set of parameters
=======================
a0              = 1.65087e-08
a1              = 0.918137
```

➢ Eficiencia de Floyd

Ejecutamos gnuplot y ejecutamos los siguientes comandos:

- **gnuplot> f(x)=a0*x*x*x+a1*x*x+a2*x+a3**
- **gnuplot> fit f(x) 'salida.dat' via a0,a1,a2,a3**
- **gnuplot> plot 'salida.dat', f(x) title 'Curva ajustada'**

Con esto podemos ver cuál sería el tiempo de ejecución para un n elevado y que sería poco rentable hacerlo con la eficiencia empírica ya que en nuestros ordenadores tardaría bastante. Algunos puntos notables más elevados:

- f(5000):
  - Lenovo: f(5000) = 945 segundos = 15 minutos y 45 segundos
  - Asus ZenBook: f(5000) = 837 segundos = 13 minutos y 57 segundos
  - HP: f(5000) = 291 segundos = 4 minutos y 51 segundos
  - Asus TUF Gaming: f(5000) = 537 s = 8 min y 57 s
- f(10000):
  - Lenovo: f(10000)= 7550 segundos= 2 horas, 5 minutos y 50 segundos
  - Asus ZenBook: f(10000)= 7049 segundos = 1 hora, 57 minutos y 29 s
  - HP: f(10000)= 1521 segundos = 25 min y 21 s
  - Asus TUF Gaming: f(10000) = 4060 s = 1 hora, 7 min y 40 s

❖ Lenovo:

```
Final set of parameters
======================
a0              = 7.52852e-09
a1              = 3.12788e-07
a2              = -0.00100926
a3              = 0.131915
```

❖ HP:

```
Final set of parameters
======================
a0              = 6.05129e-10
a1              = 9.71707e-06
a2              = -0.00566951
a3              = 0.716411
```

❖ Asus ZenBook:

```
Final set of parameters
======================
a0              = 7.45016e-09
a1              = -4.27305e-06
a2              = 0.00265262
a3              = -0.283731
```

❖ Asus TUF Gaming:

```
Final set of parameters
======================
a0              = 3.79458e-09
a1              = 2.79415e-06
a2              = -0.00146385
a3              = 0.15668
```

➢ Eficiencia de Hanoi

Ejecutamos gnuplot y ejecutamos los siguientes comandos:

- **gnuplot> f(x)=a0*2**(a1*x)**
- **gnuplot> fit f(x) 'salida.dat' via a0,a1**
- **gnuplot> plot 'salida.dat', f(x) title 'Curva ajustada'**

Con esto podemos ver cuál sería el tiempo de ejecución para un n elevado y que sería poco rentable hacerlo con la eficiencia empírica ya que en nuestros ordenadores tardaría bastante. Algunos puntos notables más elevados:

- f(40): (esto lo podeis mirar en la gráfica que da en el último comando ejecutado de gnuplot)
  - Lenovo: f(40)= 9750 segundos = 2 horas, 42 minutos y 30 segundos
  - Asus ZenBook: f(40)= 4690 segundos= 1 hora, 18 min y 10 segundos
  - HP: f(40)= 4783 s = 1 hora, 19 min y 43 s
  - Asus TUF Gaming: f(40)= 5741 s = 1 hora, 35 min y 41 s
- f(41):
  - Lenovo: f(41)= 17750 segundos= 4 horas, 55 minutos y 50 segundos
  - Asus ZenBook: f(41)= 9106 segundos = 2 horas, 31 min y 46 seg
  - HP: f(41)= 9567 s = 2 horas, 39 min y 27 s
  - Asus TUF Gaming: f(41)= 11418 s = 3 horas, 10 min y 18 s
- f(42):
  - Lenovo: f(42)= 41440 segundos= 11 horas, 30 minutos y 40 segundos
  - Asus ZenBook: f(42)= 17676 seg = 4 horas, 54 minutos y 36 segundos
  - HP: f(42)= 19134 s = 5 horas, 18 min y 54 s
  - Asus TUF Gaming: f(42)= 22705 s = 6 horas, 18 minutos y 25 s

Podemos observar cómo para un solo valor de diferencia en el tamaño, para un n alrededor de 40, el algoritmo ya tarda bastantes horas en ejecutarse y cómo aumenta exponencialmente de un valor a otro.

❖ Lenovo:

```
Final set of parameters
======================
a0              = 6.83254e-09
a1              = 1.00813
```

❖ HP:

```
Final set of parameters
======================
a0              = 4.35051e-09
a1              = 1.00048
```

❖ Asus ZenBook:

```
Final set of parameters
======================
a0              = 1.4065e-08
a1              = 0.956973
```

❖ Asus TUF Gaming:

```
Final set of parameters
======================
a0              = 6.5604e-09
a1              = 0.991769
```

6. Conclusiones

Tras realizar los tres análisis teórico, empírico e híbrido hemos llegado a importantes conclusiones.

Por un lado, podemos ver que **los tres concuerdan**: en primer lugar, hemos analizado los algoritmos sacando el orden de **eficiencia de forma teórica** y ha resultado que el algoritmo de inserción es $O(n^2)$; el de Quicksort es $O(nlogn)$; el algoritmo Floyd es $O(n^3)$; y Hanoi es $O(2^n)$. Al observar los tiempos que tardan en ejecutarse los algoritmos con diferentes elementos, nos hemos dado cuenta de que, al meterlos en una gráfica, efectivamente los **resultados empíricos** cuadran con los teóricos. Finalmente, para obtener con mayor precisión cuál es la ecuación de cada algoritmo se ha hecho un **análisis híbrido** con el programa gnuplot y, ciertamente, coinciden con las gráficas.

Por otro lado, una de las conclusiones más importantes que hemos visto es que **es fundamental prestar atención al orden de la eficiencia del algoritmo que se está realizando**. Durante los años de carrera que hemos estudiado, nunca nos hemos parado a pensar en ésto porque el número de elementos que introducíamos en las funciones eran relativamente pequeños. Sin embargo, en esta práctica nos hemos percatado de que cuando el número de elementos es muy elevado, el orden del algoritmo tiene suma importancia. Por ejemplo, el algoritmo Quicksort es el que menos orden tiene y, al pasarle como parámetro un vector de 200000 elementos, tarda entre 0.02 y 0.04 segundos en dar resultados. Sin embargo, el algoritmo Hanoi, que es el que mayor orden tiene, al pasarle un vector de 33 elementos, tarda entre 50 y 80 segundos. Es una diferencia muy grande, ya que si metiésemos un vector de 1500 elementos en el algoritmo Hanoi, podría llegar a tardar siglos en ejecutarse.

De lo reflexionado en el párrafo anterior, deducimos que en nuestra vida laboral **deberemos buscar algoritmos más cercanos a Quicksort**, es decir, con orden bajo. Siempre podemos encontrar múltiples algoritmos que resuelvan correctamente el mismo problema, pero con eficiencias diferentes. Por ello, medir la eficiencia algorítmica es realmente importante, para ayudarnos a escoger el algoritmo más eficiente.