ugr | Universidad de Granada

ALGORÍTMICA

Curso Académico 2021/2022

Práctica 1

PRÁCTICA 1: ANÁLISIS DE EFICIENCIA DE ALGORITMOS

Grupo: Buganvilla

Autores: Manuel Vicente Bolaños Quesada, Pablo Gálvez Ortigosa, Carlos García Jiménez

Índice

1. Introducción

Para realizar el análisis de la **eficiencia empírica del algoritmo**, hemos ejecutado el algoritmo en cada uno de nuestros ordenadores, obteniendo 25 datos distintos (para cada dato, se repite el algoritmo 15 veces y se hace la media). De esta forma, podremos estudiar el caso promedio del algoritmo y comprobaremos si la eficiencia empírica se corresponde con la eficiencia teórica obtenida en el paso anterior.

En lo que al análisis de la **eficiencia híbrida** se refiere, una vez estudiada la eficiencia empírica, disponemos los datos en gráficos, para poder ajustarle varios tipos de funciones, y poder ver cuál es la que más se ajusta a los datos. Esta función se podrá usar para predecir el tiempo que tardará el algoritmo en ejecutarse para una entrada de tamaño n.

Para profundizar en la variación de la eficiencia empírica en función de **parámetros externos**, realizaremos un estudio desde dos aspectos distintos: por una parte, comprobaremos la variación de tiempos dependiendo de las opciones de compilación utilizadas (con/sin optimización) y por otra parte probaremos los algoritmos utilizando distintos software (Linux, Windows y una máquina virtual que será VMBox).

2. Algoritmos cuadráticos

En esta sección estudiaremos dos algoritmos cuadráticos: el algoritmo de **inserción** y el de **selección**. Ambos algoritmos sirven para ordenar un vector de datos de tamaño n. A lo largo de esta sección estudiaremos su eficiencia teórica (para ver que, efectivamente son algoritmos cuadráticos), la empírica (para ver que los datos obtenidos se corresponden con el estudio teórico), y la híbrida (para obtener una serie de constantes con respecto a la función que aproxima los datos).

Por un lado, el algoritmo de **inserción** es una forma muy natural de ordenar, ya que va comparando los elementos del vector, y colocando en su posición directamente. Por otro lado, el algoritmo de **selección** se basa en encontrar el menor elemento que todavía no esté en su lugar, para después posicionarlo en el sitio correcto.

2.1. Eficiencia teórica

```
61   inline static void insercion(int T[], int num_elem)
62   {
63     insercion_lims(T, 0, num_elem);
64   }

139  static void insercion_lims(int T[], int inicial, int final)
140  {
141    int i, j;
142    int aux;
143    for (i = inicial + 1; i < final; i++) {
144      j = i;
145      while ((T[j] < T[j-1]) && (j > 0)) {
146        aux = T[j];
147        T[j] = T[j-1];
148        T[j-1] = aux;
149        j--;
150      };
151    };
152  }
```

Para comenzar, como la función *insercion* únicamente llama a la función *insercion_lims*, estudiaremos la eficiencia teórica de la segunda ya que será igual. Para ello veamos que en las líneas 141, 142, 144, 146-149 se encuentran instrucciones $O(1)$, por lo que quedaría observar los bucles. En primer lugar, el bucle *for* se ejecutará siempre $n-1$ veces; por otro lado, el bucle *while* se ejecuta en el peor caso $n-1$ veces, como podemos observar:

```
j = i
while ((T[j] < T[j-1]) && (j>0)) {...
```

En conclusión, este algoritmo es $\mathbf{O(n^2)}$. El análisis de eficiencia teórica del algoritmo de selección se realiza de forma análoga.

2.2. Eficiencia empírica e híbrida

Hemos ejecutado los programas por separado, con valores desde 5000 hasta 173000, en tramos de 7000. Además hemos añadido valores de 250.000, 500.000 y 1.000.000 de datos. El resultado ha sido el siguiente:

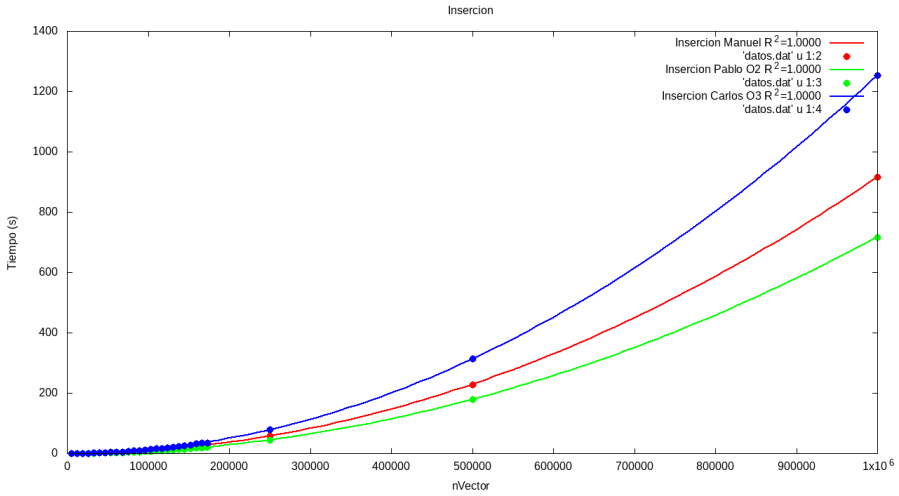
nVector	Selección Manuel	Selección Pablo	Selección Carlos	Inserción Manuel	Inserción Pablo	Inserción Carlos
5000	0,025617	0,02250833333	0,039695	0,0229905	0,01730393333	0,02660453333
12000	0,1276425	0,1285718	0,2252920667	0,130349	0,1044312667	0,1766918667
19000	0,3189785	0,3301279333	0,5635880667	0,3278045	0,2878446667	0,4440983333
26000	0,6063065	0,6152020667	1,054082	0,6095745	0,5045608667	0,8343289333
33000	0,9664215	1,021349333	1,697045333	0,9892	0,8054018	1,347536
40000	1,50094	1,507136667	2,491541333	1,447985	1,164980133	1,981326
47000	2,16725	2,049969333	3,439518667	2,007765	1,598563333	2,735817333
54000	2,8658	2,669086667	4,414734	2,653995	2,098915667	3,6238
61000	3,75968	3,388730667	5,650004	3,391155	2,6191594	4,617457333
68000	4,83031	4,226175333	7,09818	4,222825	3,255766467	5,751440667
75000	5,88069	5,059013333	8,652764	5,12039	3,9543452	6,990404
82000	7,060245	6,00543	10,35008667	6,131725	4,721145133	8,365864
89000	8,39528	7,046218	12,24201333	7,278495	5,563633267	9,867756667
96000	9,860225	8,189760667	14,25198	8,455175	6,452845467	11,42718
103000	11,49575	9,575940667	16,44670667	9,60801	7,4119136	13,15434667
110000	13,15545	10,92765333	18,69854667	11,03795	8,4765962	16,0165
117000	14,87815	12,12991333	21,16924667	12,544	9,5621954	17,02415333
124000	17,1876	13,84202667	23,80529333	14,06935	10,7699158	19,14423333
131000	18,9912	15,31369333	26,57835333	15,6564	12,0054758	21,38144
138000	20,7869	16,85308	29,50183333	17,3998	13,29033087	23,71659333
145000	22,9642	18,5653	32,58263333	19,1482	14,68728967	26,22794
152000	25,3143	20,78674	35,78056667	21,09035	16,14120447	28,75235333
159000	27,81525	22,53712667	39,19828	23,07515	17,87983527	31,4874
166000	30,3582	24,5298	42,64658667	25,06915	19,46714413	34,36643333
173000	32,87955	26,54596667	46,38744	27,37015	21,15399853	34,49705333
250000	69,8101	54,0286	96,82114667	57,0916	442.527	78,06967333
500000	281,4927	215,835	387,0198	228,8775	178,502	313,0037333
1000000	1132,493	839,015	1493,589733	915,406	716,258	1252,77375

2.2.1. Eficiencia híbrida de inserción

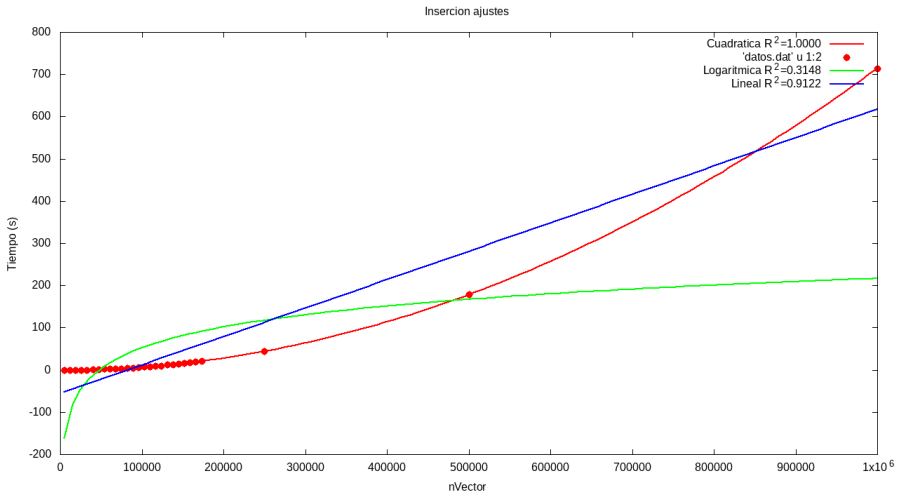
Estudiemos ahora la eficiencia híbrida del algoritmo de inserción. Para ello, representaremos en un gráfico los tiempos obtenidos al ejecutar el algoritmo (los del apartado anterior). Hallaremos también una función que se ajuste bien a los datos. Gracias al estudio teórico anterior, sabemos que dicha función será de la forma

$$T(n) = a_0 n^2 + a_1 n + a_2$$

donde a_0, a_1 y a_2 son constantes reales, que dependerán de la arquitectura del ordenador del que se analicen los datos.



Para hallar la función de la que se hablaba anteriormente, centrémonos en solo una de las series de datos de la gráfica anterior; por ejemplo, la de Pablo.



Como podemos ver en este último gráfico, a esa nube de puntos se le ha ajustado una función logarítmica (línea verde), otra polinomial de segundo grado (línea roja), y otra lineal (línea azul). Para ver cómo de bien aproximan esas funciones a nuestra nube de puntos, tenemos que fijarnos en el coeficiente de determinación (R^2). Mientras más próximo a 1 esté, más acertado será el ajuste.

En efecto, el coeficiente de correlación resultante de la función cuadrática es 1, tal y como el análisis teórico sugería. También observamos que las otras funciones no son buenos ajustes de los datos, al estar sus coeficientes de correlación alejados de 1.

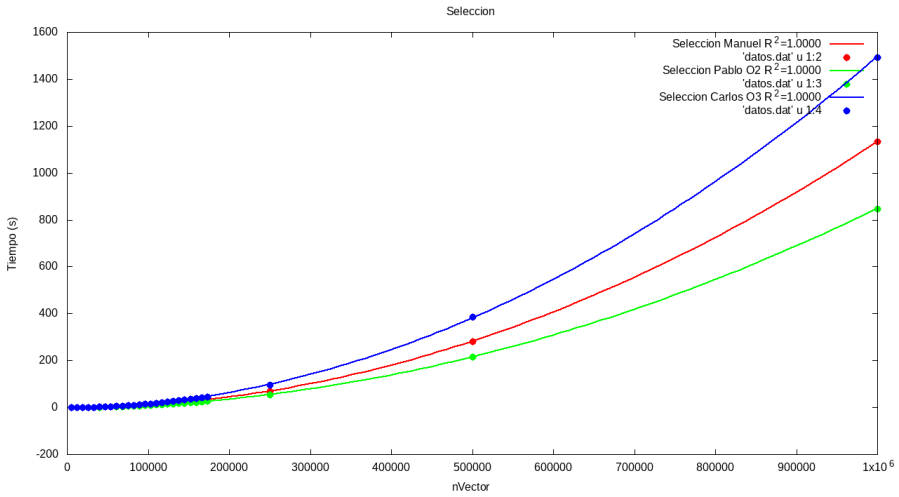
Finalmente, gracias a los resultados arrojados por *gnuplot* obtenemos que

$$a_0 = 7{,}16581 \cdot 10^{-10},\ a_1 = -2{,}13261 \cdot 10^{-6},\ a_2 = -0{,}377928$$

.

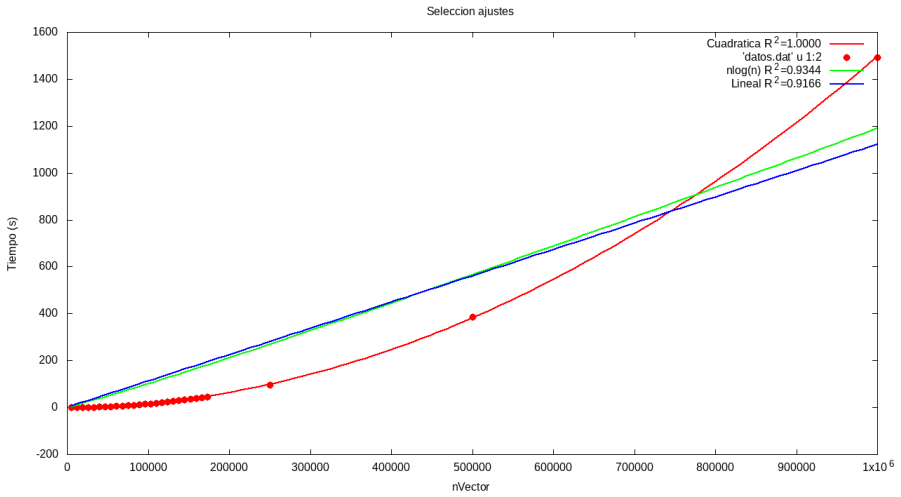
2.2.2. Eficiencia híbrida de selección

Para estudiar la eficiencia híbrida del algoritmo de selección actuaremos de igual manera que en el apartado anterior. Presentemos pues, el gráfico que contiene los tiempos de ejecución:



Analicemos ahora la eficiencia híbrida. Primeramente, centrémonos en una única nube de puntos. En este caso, usaremos la de Carlos. El estudio teórico anterior también es válido para este algoritmo, por lo que sabemos que la función que mejor se ajusta a estos datos será, de nuevo, un polinomio de segundo grado con coeficientes a_0, a_1, a_2, que, nuevamente, serán constantes a determinar por las características específicas de la arquitectura del ordenador utilizado.

Intentemos ajustar los datos con una parábola, una función de tipo $n \log n$, y una función lineal, para ver si es cierto que la parábola ajusta mejor los datos.



Como esperábamos, la parábola tiene por coeficiente de correlación un 1, mientras que la función de tipo $n \log n$ tiene un peor ajuste (0.9344), y la lineal es la que peor ajusta los datos, con un coeficiente de correlación de 0,9166.

Para terminar, podemos hallar los coeficientes a_0, a_1 y a_2, que aparecen en el gráfico anterior. Deducimos entonces que $a_0 = 1{,}46 \cdot 10^{-9},\ a_1 = 3{,}48 \cdot 10^{-5},\ a_2 = -2{,}06$

2.3. Mejor y peor caso

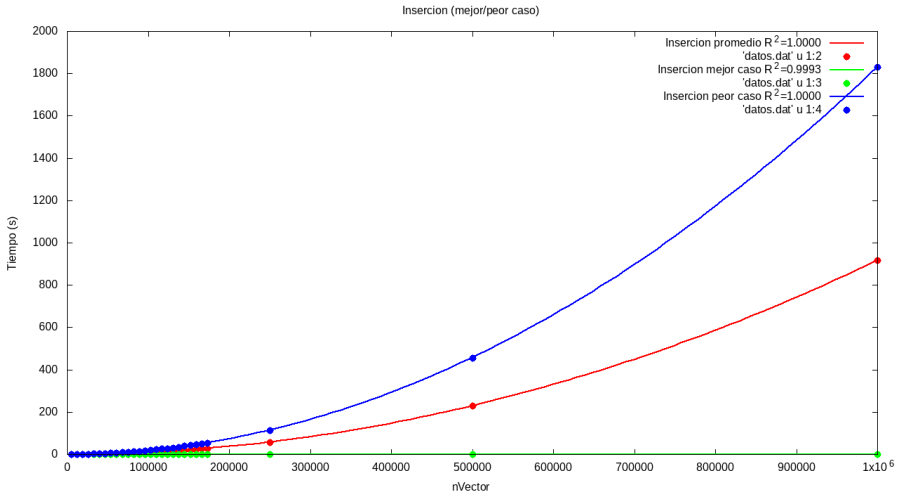
A continuación, vamos a exponer los resultados de los tiempos obtenidos al ejecutar los dos algoritmos sobre un vector totalmente ordenado (el mejor caso), y sobre un vector totalmente desordenado (el peor caso). Primero, veamos los datos obtenidos, para después, más detenidamente, ver los resultados arrojados por los algoritmo de inserción y de selección.

2.3.1. Inserción

	Manuel		Pablo		Carlos	
nVector	Inserción mejor caso	Inserción peor caso	Inserción mejor caso	Inserción peor caso	Inserción mejor caso	Inserción peor caso
5000	3.20E-05	0.047244	8.87E-06	0.038259	4.60E-05	0.036322
12000	5.40E-05	0.260256	2.06E-05	0.2125578	1.05E-04	0.225057
19000	4.20E-05	0.669545	3.26E-05	0.5177679	1.67E-04	0.506823
26000	5.70E-05	1.22646	4.47E-05	0.9792062	2.27E-04	0.944544
33000	7.30E-05	1.9792	6.05E-05	1.5409787	2.52E-04	1.52099
40000	8.90E-05	2.91406	7.72E-05	2.2697893	6.50E-05	2.2433

nVector	Mejor caso	Peor caso	Mejor caso	Peor caso	Mejor caso	Peor caso
47000	0.000104	4.03075	0.0000881	3.1403489	0.000122	3.10346
54000	0.000118	5.32543	0.000094	4.1312547	8.90E-05	4.09343
61000	0.000134	6.76549	0.0001104	5.3014948	0.0001	5.22391
68000	0.000149	8.42015	0.0001233	6.5495377	0.000112	6.51118
75000	0.000165	10.2911	0.0001879	7.8814421	0.000123	7.91901
82000	0.00018	12.2471	0.0001525	9.4715231	0.000168	9.48049
89000	0.000195	14.8521	0.0001959	11.1353334	0.000155	11.2078
96000	0.00021	16.8037	0.0001747	12.856603	0.000167	13.0252
103000	0.000225	19.3409	0.0001887	14.8214203	0.00018	15.0027
110000	0.000241	22.1514	0.0002004	16.8982107	0.000193	17.1038
117000	0.000294	25.0129	0.0002472	19.1107615	0.000204	19.3279
124000	0.000272	28.1884	0.0002184	21.4068457	0.000227	21.7165
131000	0.000287	31.3837	0.0002465	23.9160435	0.00023	24.2185
138000	0.000319	34.7945	0.0002917	26.4986945	0.000241	26.8347
145000	0.000317	38.4817	0.0003002	29.2240016	0.000253	29.699
152000	0.000333	42.2154	0.0003115	32.025619	0.000265	32.6862
159000	0.000359	46.1095	0.0002983	35.0100261	0.000278	35.6859
166000	0.000393	50.3181	0.0003211	38.1265397	0.000289	38.7998
173000	0.000379	54.7782	0.0003529	41.4334107	0.000304	42.2893
250,000	0.00055	114.23	0.00054569	77.1845611	0.000436	88.5615
500,000	0.001106	456.702	0.00101166	281.8651868	0.000877	353.297
1,000,000	0.002235	1830.510	0.0021769	946.16851	0.001774	1408.950

Podemos observar en la tabla anterior que en el mejor de los casos este algoritmo se ejecuta muy rápidamente. Esto es porque, a pesar de ser el algoritmo de orden $O(n^2)$, en dicho caso, se comporta como un algoritmo lineal, es decir, de orden $O(n)$. La explicación de esto último es porque, usando el código proporcionado por el profesora, el programa no entra en el bucle *while*. Con el fin de ver que esto es realmente así, y que en el peor de los casos el algoritmo es más lento (ya que entra en el bucle *while* todas las veces posibles), veamos el gráfico generado por los datos de Manuel, y comparemos estos nuevos resultados con el caso promedio, estudiado en 2.2.



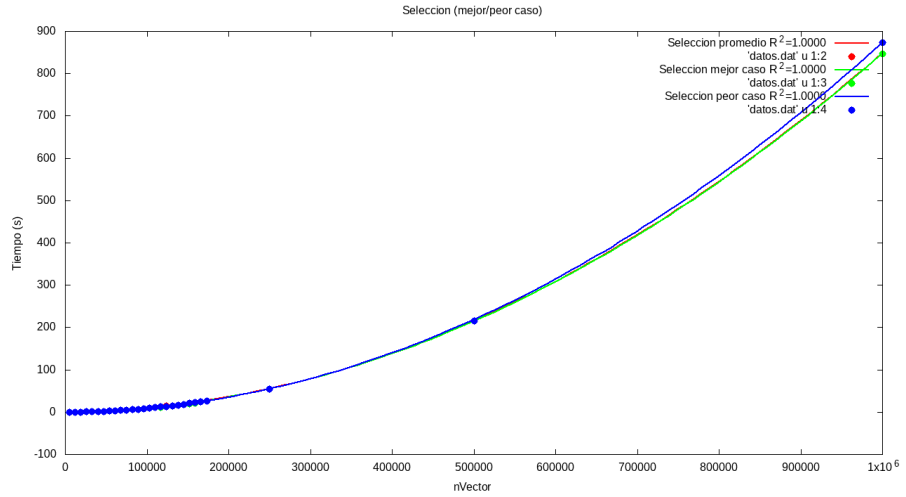
A los datos del mejor caso y del caso promedio le hemos ajustado un polinomio de segundo grado, con un coeficiente de correlación resultante de 1, en cada uno de esos casos. Esto quiere decir que estas funciones (cuyas expresiones están en el gráfico) ajustan muy bien a las nubes de puntos, por lo que será fiable utilizarlas para predecir otros resultados. También hemos ajustado una función lineal a los datos provenientes del mejor caso, y nos da un coeficiente de correlación muy cercano a 1, luego, efectivamente, el algoritmo se comporta linealmente cuando el vector está ordenado.

2.3.2. Selección

	Manuel		Pablo		Carlos	
nVector	Selección mejor caso	Selección peor caso	Selección mejor caso	Selección peor caso	Selección mejor caso	Selección peor caso
5000	2.83E-02	0.029147	2.35E-02	0.024544	3.95E-02	0.026289
12000	1.58E-01	0.17333	1.35E-01	0.141116	2.22E-01	0.131317
19000	3.39E-01	0.39891	3.72E-01	0.3521330	5.56E-01	0.329656
26000	6.63E-01	0.723183	6.50E-01	0.657584	1.04E+00	0.615733
33000	1.08E+00	1.18413	9.93E-01	1.1056700	1.68E+00	0.988235
40000	1.55E+00	1.75087	1.51E+00	1.6287500	2.46E+00	1.4526
47000	2.16589	2.3949	1.9911327	2.1178200	3.40094	2.00616
54000	3.07663	3.26149	2.538888667	2.6383000	4.20E+00	2.66043
61000	3.9456	4.26888	3.234396	3.36899	3.15124	3.38078
68000	4.96714	5.23472	4.0180107	4.1736500	3.94725	4.19531
75000	6.05579	6.40612	4.8828167	5.1027600	4.79398	5.1046
82000	7.36131	7.63694	5.8384393	6.0658800	5.72979	6.10005
89000	8.74945	8.96612	6.8822587	7.05343	6.74846	7.18884
96000	10.1561	10.623	8.0134093	8.20657	7.84964	8.37008
103000	11.7282	12.4265	9.2413880	9.4360500	9.03435	9.63981
110000	13.4053	14.3108	10.55116	10.7741000	10.3051	11.0022

nVector	Mejor caso	Peor caso	Mejor caso	Peor caso	Mejor caso	Peor caso
117000	15.3078	16.2196	11.94776	12.1915000	11.6562	12.4544
124000	17.1285	18.2624	13.41161333	13.6718000	13.104	13.9715
131000	19.1226	20.3806	14.9844533	15.2718000	14.6212	15.5885
138000	21.1846	22.5957	16.6218667	16.9186000	16.2215	17.281
145000	23.5023	24.7644	18.3503	18.6479	17.9083	19.0698
152000	25.888	27.344	20.1417067	20.505	19.674	20.956
159000	28.3811	29.9293	22.0159200	22.4236000	21.5284	22.9183
166000	30.8694	32.658	24.0913600	24.4280000	23.4592	24.9993
173000	33.6014	35.7603	26.0201467	26.4653000	25.4846	27.2608
250,000	70.4959	74.754	61.63561000	54.9764310	53.1694	56.7844
500,000	282.501	299.937	220.16512100	215.6534000	212.823	226.467
1,000,000	1133.89	1203.660	894.6515000	872.94613	851.257	905.937

Veamos también el gráfico generado por los datos obtenidos con el ordenador de Pablo, para ver más gráficamente el comportamiento de los tiempos del algoritmo.



Se observa que, en cualquiera de los casos, el algoritmo tarda prácticamente lo mismo. Esto se debe a que el algoritmo, independientemente del orden del vector, hace las mismas comprobaciones, por lo que van a salir unos tiempos similares. Por ello, podemos aproximar las tres nubes de puntos con tres funciones polinómicas de grado dos con una bondad bastante buena.

En este caso, es bastante fiable usar esas funciones para predecir resultados de la función $T(n)$, ya que ni siquiera depende del orden del vector.

2.4. Parámetros externos

2.4.1. Optimización

Como dijimos en la introducción, para ver el impacto de usar optimizaciones al compilar sobre el tiempo de ejecución, hemos compilado estos programas con los parámetros **-O2** y **-O3**, habiendo obtenido los siguientes resultados al ejecutarlos:

nVector	Inserción	Inserción O2	Inserción O3	Selección	Selección O2	Selección O3
5,000	0.0229905	0.0084175	0.008842	0.025617	0.012782	0.0123895
12000	1.30E-01	0.0480345	4.75E-02	0.127643	6.66E-02	0.061412
19000	3.28E-01	0.1199655	1.20E-01	0.318979	1.68E-01	0.1535665
26000	6.10E-01	0.22494	2.25E-01	0.606307	3.13E-01	0.28726
33000	9.89E-01	0.3619055	3.54E-01	0.966422	5.04E-01	0.4599655
40000	1.45E+00	0.5349805	5.29E-01	1.500940	7.37E-01	0.675109
47000	2.01E+00	0.734205	7.29E-01	2.167250	1.02E+00	0.937596
54000	2.653995	0.970557	0.9674275	2.865800	1.35262	1.23293
61000	3.391155	1.232415	1.24041	3.759680	1.73E+00	1.58123
68000	4.222825	1.536165	1.532775	4.830310	2.161895	1.975115
75000	5.12039	1.872135	1.8645000	5.880690	2.629445	2.383685
82000	6.131725	2.2376	2.2362700	7.060245	3.13317	2.87239
89000	7.278495	2.635865	2.6362150	8.395280	3.7021	3.37044
96000	8.455175	3.06937	3.0592400	9.860225	4.31184	3.904615
103000	9.60801	3.529285	3.5335500	11.495750	5.00225	4.495885
110000	11.03795	4.0309	4.0204000	13.155450	5.6742	5.11894
117000	12.544	4.53865	4.555185	14.878150	6.47027	5.79975
124000	14.06935	5.11199	5.11178	17.187600	7.504575	6.50486
131000	15.6564	5.7018	5.69168	18.991200	8.097065	7.262805
138000	17.3998	6.33898	6.3081050	20.786900	8.91841	8.06925
145000	19.1482	6.977455	6.9863100	22.964200	9.882675	8.90614
152000	21.09035	7.69376	7.66272	25.314300	10.914	9.79012
159000	23.07515	8.393875	8.4018050	27.815250	11.9539	10.8774
166000	25.06915	9.158495	9.1450300	30.358200	12.978	11.9118
173000	27.37015	9.92488	9.9346600	32.879550	14.0132	12.6728
250000	57.0916	20.73435	20.7596000	69.8101000	29.51765	26.4652
500,000	228.8775	83.08205	83.79390000	281.4927000	120.28465	105.824
1,000,000	915.406	332.2765	332.74900000	1132.4930000	501.8075	425.028

Representando en unos gráficos la información anterior:

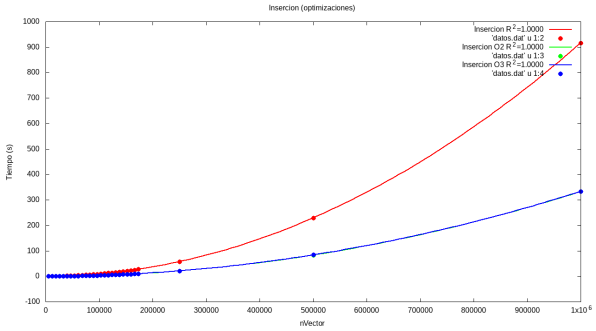


Figura 1: Inserción (-O2 y -O3)

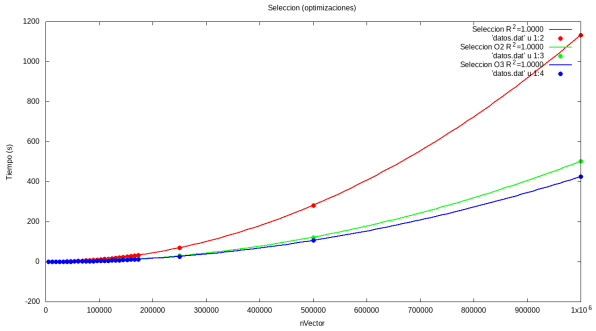


Figura 2: Selección (-O2 y -O3)

Analicemos lo ocurrido. Estudiemos primero el algoritmo de inserción. Hay una clara reducción del tiempo empleado usando optimizaciones. Aproximadamente, con optimizaciones tarda un tercio menos de lo que el programa tardaba sin estas. También se puede observar, tanto en el gráfico como en la tabla, que los valores con optimización O2 y con optimización O3 son prácticamente los mismos.

Por otra parte, compilar el código de selección con la opción -O2 ha ayudado a disminuir el tiempo de ejecución del programa más de la mitad. Con una optimización de O3 se consiguen tiempos todavía más pequeños, pero la diferencia entre O2 y O3 ya no es tan grande como la diferencia entre compilar sin optimización y con optimización.

2.4.2. Software

Realizando las pruebas detalladas en la introducción, se obtiene una tabla de tiempos como la siguiente:

nVector	Inserción (Linux)	Inserción (Windows)	Inserción (VMBox)	Selección (Linux)	Selección (Windows)	Selección (VMBox)
5000	0,01730393333	0,021	0,01981733333	0,022508	0,024	0,024349
12000	0,1044312667	0,1143333333	0,1138233333	0,128572	0,143	0,139861
19000	0,2878446667	0,2746666667	0,285051	0,330128	0,344	0,351064
26000	0,5045608667	0,5126666667	0,5350256667	0,615202	0,614	0,660775
33000	0,8054018	0,8236666667	0,8769793333	1,021349	0,999	1,06934
40000	1,164980133	1,167333333	1,262213333	1,507137	1,48	1,55407
47000	1,598563333	1,585666667	1,7631	2,049969	2,103	2,14632
54000	2,098915667	2,137	2,315196667	2,669087	2,678	2,85034
61000	2,6191594	2,729333333	2,96754	3,388731	3,333	3,61388
68000	3,255766467	3,401	3,70488	4,226175	4,135	4,55857
75000	3,9543452	4,135666667	4,519946667	5,059013	5,063	5,52099
82000	4,721145133	4,996333333	5,408283333	6,005430	6,01	6,58437
89000	5,563633267	5,890666667	6,394693333	7,046218	7,037	7,73963

96000	6,452845467	6,749	7,37191	8,189761	8,198	8,97789
103000	7,4119136	7,784	8,511333333	9,575941	9,428	10,3318
110000	8,4765962	8,671	9,717443333	10,927653	10,738	11,769
117000	9,5621954	9,905666667	10,9609	12,129913	12,151	13,3456
124000	10,7699158	11,024	12,3265	13,842027	13,643	14,9549
131000	12,0054758	12,386	13,76366667	15,313693	15,385	16,7154
138000	13,29033087	13,642	15,3135	16,853080	16,905	18,5604
145000	14,68728967	15,40133333	16,86666667	18,565300	19,305	20,4608
152000	16,14120447	16,57433333	18,5099	20,786740	20,115	22,4978
159000	17,87983527	18,231	20,32133333	22,537127	22,167	24,6421
166000	19,46714413	20,06166667	22,14406667	24,529800	24,197	26,8568
173000	21,15399853	21,502	24,10286667	26,545967	26,015	29,2616

, y si realizamos dos gráficas de lo anterior obtenemos:

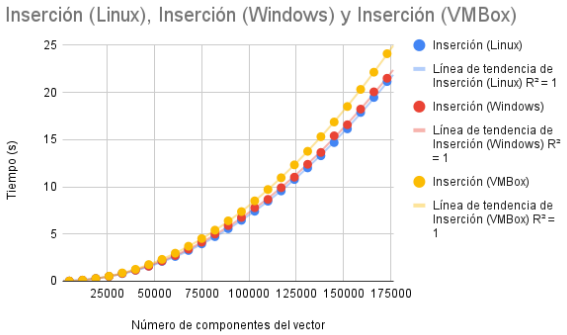


Figura 3: Inserción (Distintos Software)

Figura 4: Selección (Distintos Software)

Comentaremos ahora las gráficas para aclarar lo obtenido. Primero de todo, decir que las diferencias en la forma de las gráficas son inexistentes, en ambos casos tenemos una parábola que se ajusta perfectamente a las gráficas. En cuanto a los tiempos, se puede destacar que los tiempos obtenidos en Linux son ligeramente menores que los obtenidos en Windows y en ambos casos, bastante mejores que los que se tienen con una máquina virtual. Las razones son obvias, ya que en una máquina virtual, el porcentaje de CPU del ordenador que se puede utilizar es bastante menor que en los otros casos.

3. Algoritmos de orden $n \log n$

Analizaremos a continuación la eficiencia teórica, empírica e híbrida de dos de los más conocidos algoritmos de ordenación: **Quicksort** y **Heapsort**. Por una parte, el algoritmo de **Quicksort** se basa en la técnica de ”divide y vencerás”, el problema se dividirá en subproblemas de menor tamaño y se resuelven por separado para finalmente ser unidos de nuevo. Por su parte, **Heapsort** consiste en almacenar todos los elementos del vector a ordenar en un *heap*, y extraer el nodo que queda como nodo raíz en sucesivas iteraciones obteniendo el conjunto ordenado.

3.1. Eficiencia teórica

```
149  static void heapsort(int T[], int num_elem)
150  {
151    int i;
152    for (i = num_elem/2; i >= 0; i--)
153      reajustar(T, num_elem, i);
154    for (i = num_elem - 1; i >= 1; i--)
155      {
156        int aux = T[0];
157        T[0] = T[i];
158        T[i] = aux;
159        reajustar(T, i, 0);
160      }
161  }
162
163
164  static void reajustar(int T[], int num_elem, int k)
165  {
166    int j;
167    int v;
168    v = T[k];
169    bool esAPO = false;
170    while ((k < num_elem/2) && !esAPO)
171      {
172        j = k + k + 1;
173        if ((j < (num_elem - 1)) && (T[j] < T[j+1]))
174    j++;
175        if (v >= T[j])
176    esAPO = true;
177        T[k] = T[j];
178        k = j;
179      }
180    T[k] = v;
181  }
```

En este caso estudiaremos la función heapsort, de la cual las líneas 151, 156-158 son asig-

naciones de valores y declaración de variables, por lo que son $O(1)$. De esta función se pueden destacar los dos bucles for, por un lado tenemos el primero que se ejecutará $\frac{n}{2}$ veces; por otro lado, el segundo bucle for se ejecuta $n-1$ veces. Se puede observar entonces que el algoritmo es $O(n * P)$, siendo P la eficiencia de la función reajustar, que se encuentra dentro de los dos bucles for.

Por tanto, analicemos la eficiencia de la función reajustar. En primer lugar se observa que las líneas 166-169, 180 son O(1), por tanto pasemos ahora a estudiar el bucle while. Éste, en el peor de los casos, se ejecuta $\frac{n}{2}$ veces como se puede ver:

```
while ((k < num_elem/2) ...
```

Para ello, k debe crecer lo más lento posible. Para analizarlo debemos tener en cuenta que en la línea 178 tenemos:

```
k = j;
```

Es decir, si la j crece lo más despacio posible, también lo hace la k. Por tanto suponemos que no entra en el primer bucle if ($O(1)$) ya que éste incrementa la j. Además, poniéndonos en el peor de los casos, tampoco entraría en el segundo if ($O(1)$) porque asignaría true a esAPO, lo que provocaría salir del bucle while. Por tanto se nos queda:

```
j = k + k + 1;
k = j
```

Es decir, $k_i = 2k_{i-1}+1$, con i el número de iteración. Por tanto debemos resolver la recurrencia:

$$k_i = 2^i \cdot c$$

La cual tiene como solución particular:

$$x = 2x + 1$$

$$x = -1$$

Obtenemos entonces la solución de la recurrencia:

$$k_i = 2^i \cdot c - 1$$

Como hemos comentado al principio, queremos ver el peor caso del algoritmo, es decir, cuando k esté más distanciado de $\frac{n}{2}$ por lo que suponemos el valor inicial de k en 0, $k_0 = 0$:

$$k_0 = 0$$

$$k_0 = 2^0 \cdot c - 1$$

$$0 = 1 \cdot c - 1$$

$$c = 1$$

Y así tenemos la solución de la recurrencia:

$$k_i = 2^i - 1$$

Ahora veamos en qué iteración $k = \frac{n}{2}$:

$$2^i - 1 = \frac{n}{2}$$

$$2^i = \frac{n-2}{2}$$

$$i = log_2(n-2) - log_2 2$$

$$i = log_2(n-2) - 1$$

Entonces podemos observar que el cuerpo del while, y por tanto la función reajustar tiene una eficiencia $O(log_2(n))$. Como dijimos al principio la eficiencia del algoritmo será $O(n * P)$, siendo P la eficiencia de la función reajustar, por tanto, el algoritmo es $\mathbf{O(nlogn)}$. El análisis de eficiencia teórica del algoritmo quicksort se realiza de forma análoga.

3.2. Eficiencia empírica e híbrida

Utilizando el procedimiento descrito en la introducción, se obtienen los siguientes datos, que recogemos en la siguiente tabla:

nVector	Quicksort Manuel	Quicksort Pablo	Quicksort Carlos	Heapsort Manuel	Heapsort Pablo	Heapsort Carlos
100000	0,012418	0,009424533333	0,01639626667	0,01664566667	0,014249	0,01751253333
600000	0,084653	0,06655053333	0,1119696	0,1216504667	0,111585	0,1690705333
1100000	0,161785	0,1226129333	0,214225	0,2410622667	0,194853	0,329057
1600000	0,246597	0,1768092667	0,3199090667	0,3677804667	0,272669	0,4954477333
2100000	0,323029	0,2361205333	0,4266232	0,5075307333	0,372752	0,6687072
2600000	0,405287	0,3053693333	0,5359213333	0,6544696667	0,473122	0,8464478667
3100000	0,483884	0,3619072667	0,6465873333	0,8097462667	0,578324	1,029725333
3600000	0,570496	0,4216494667	0,7572128667	0,9730548667	0,6882	1,220544667
4100000	0,655023	0,4841651333	0,8678809333	1,13182	0,80286	1,417642
4600000	0,743493	0,547418	0,9820567333	1,294256	0,920331	1,618022
5100000	0,826197	0,60977	1,095056667	1,459831333	1,038131	1,823295333

5600000	0,912782	0,6763012	1,208618	1,687880667	1,160697	1,996006667
6100000	1,001989	0,7373888667	1,322552667	1,808884667	1,284137	2,215902667
6600000	1,107589	0,8025293333	1,441727333	1,988263333	1,411327	2,435152667
7100000	1,170603	0,8665004	1,554965333	2,21497	1,537646	2,668646
7600000	1,260971	0,9285397333	1,670010667	2,346161333	1,667612	2,88229
8100000	1,348918	0,9973234	1,787754667	2,529062667	1,80147	3,115236
8600000	1,437366	1,060276	1,902576	2,71282	1,933794	3,337984667
9100000	1,524943	1,122298667	2,000810667	2,895624	2,073293	3,589226667
9600000	1,618905	1,196016	2,121182	3,092762	2,206901	3,821922
10100000	1,706428	1,251622	2,245252667	3,282677333	2,375785	3,968524
10600000	1,791969	1,320443333	2,279494667	3,469946667	2,5596	4,220862667
11100000	1,88164	1,385769333	2,445923333	3,666698	2,702133	4,516011333
11600000	1,982073	1,451252667	2,534514	3,842466	2,848075	4,752447333
12100000	2,069075	1,522953333	2,640994667	4,046161333	2,994913	4,993111333

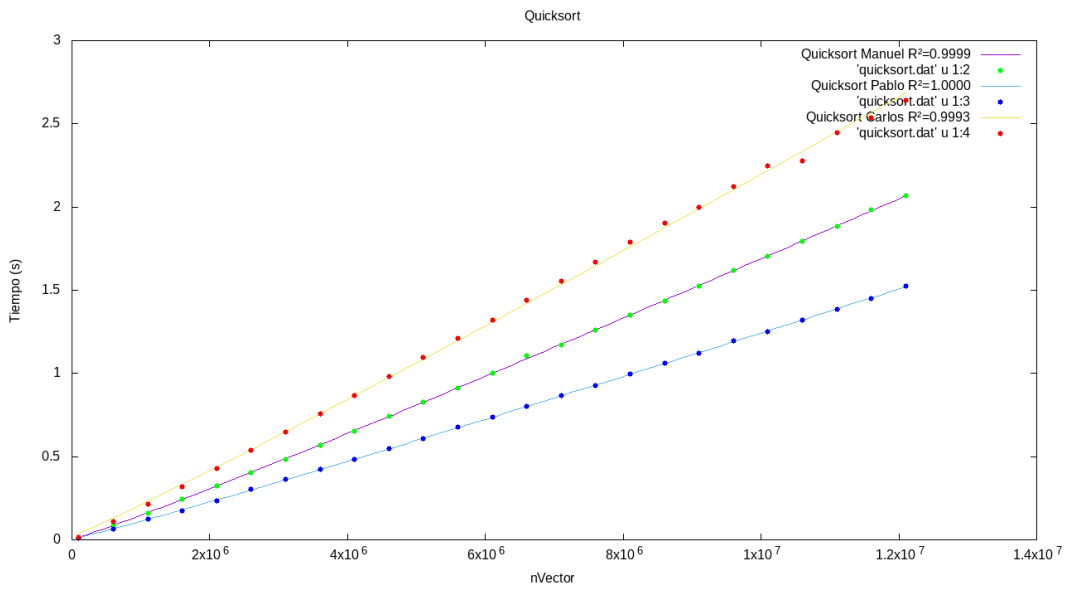
3.2.1. Eficiencia híbrida de Quicksort

Para hallar la eficiencia híbrida del algoritmo de Quicksort, representaremos en una gráfica los tiempos obtenidos en el apartado anterior, e intentaremos hallar la función que mejor se ajusta a dichos datos. Como ya se comentó en el apartado de eficiencia teórica, dicha función será de la forma:

$$T(n) = a_0 \cdot n \log n + a_1$$

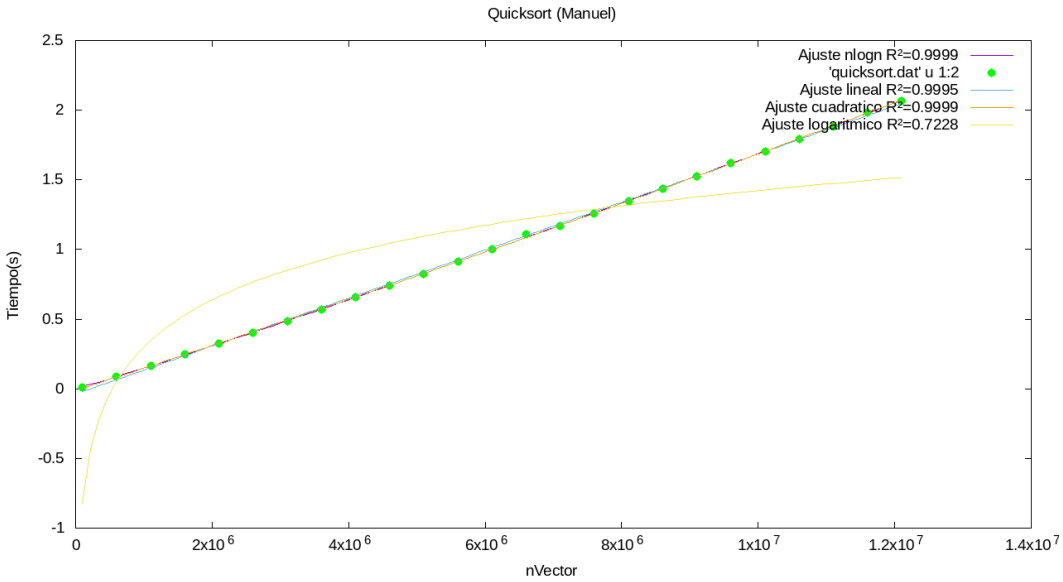
, donde a_0 y a_1 serán constantes reales que dependerán de la arquitectura concreta del ordenador en el que se hayan realizado las pruebas.

Si representamos en una gráfica los tiempos obtenidos en cada ordenador se obtiene:



Aunque se observan diferencias entre los tiempos obtenidos por los tres ordenadores (que depende del hardware utilizado), en los tres casos se observan funciones parecidas, que podrían ajustarse al ajuste deseado. Para comprobar si realmente el ajuste es realmente bueno, nos fijaremos en una de las tres gráficas y determinaremos las constantes ocultas y el coeficiente de correlación.

Tomando una de las gráficas, ajustaremos por mínimos cuadrados una función, intentando encontrar la aplicación que mejor se ajusta a dichos datos:



El ajuste es prácticamente perfecto con la función $T(n)$ con $a_0 = 1{,}0461 \cdot 10^{-8}$ y $a_1 = 0{,}00306$. De hecho, el coeficiente de correlación es $r = 0{,}999969$, prácticamente 1, lo que quiere decir que la función $T(n)$ es, de hecho, muy fiable para realizar predicciones de cuánto tiempo tardará nuestro algoritmo para un cierto número de componentes de un vector.

También podemos comprobar que se trata, de hecho, del mejor ajuste posible, ya que con un ajuste lineal se obtiene $r = 0{,}997$, con uno logarítmico $r = 0{,}850$ y con uno exponencial $r = 0{,}83$.

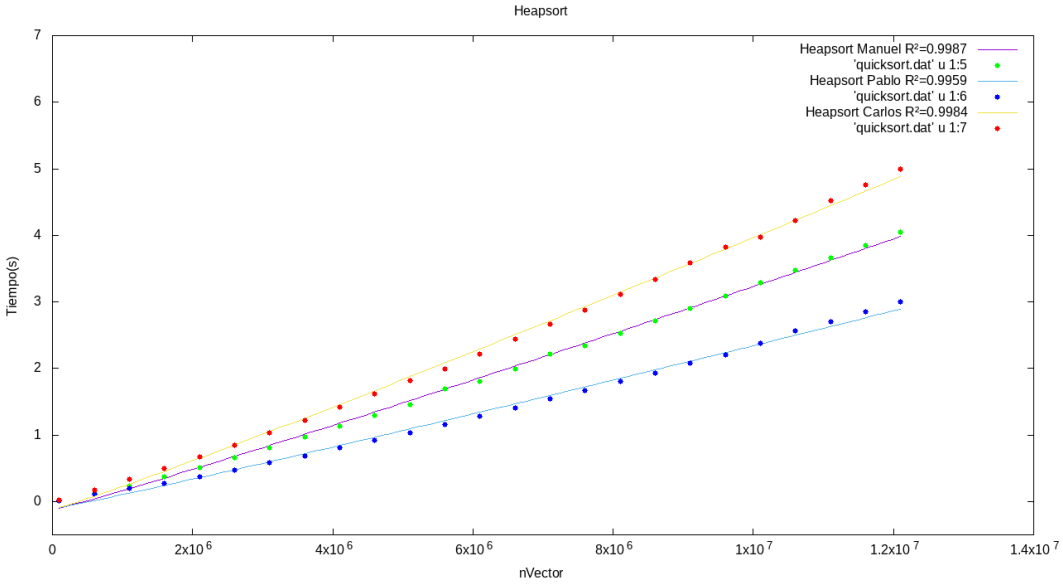
3.2.2. Eficiencia híbrida de Heapsort

De forma análoga a Quicksort, representaremos todos los tiempos obtenidos por los integrantes del grupo, intentando encontrar la función que, por mínimos cuadrados, mejor se ajusta a lo obtenido. De nuevo, dicha función será de la forma:

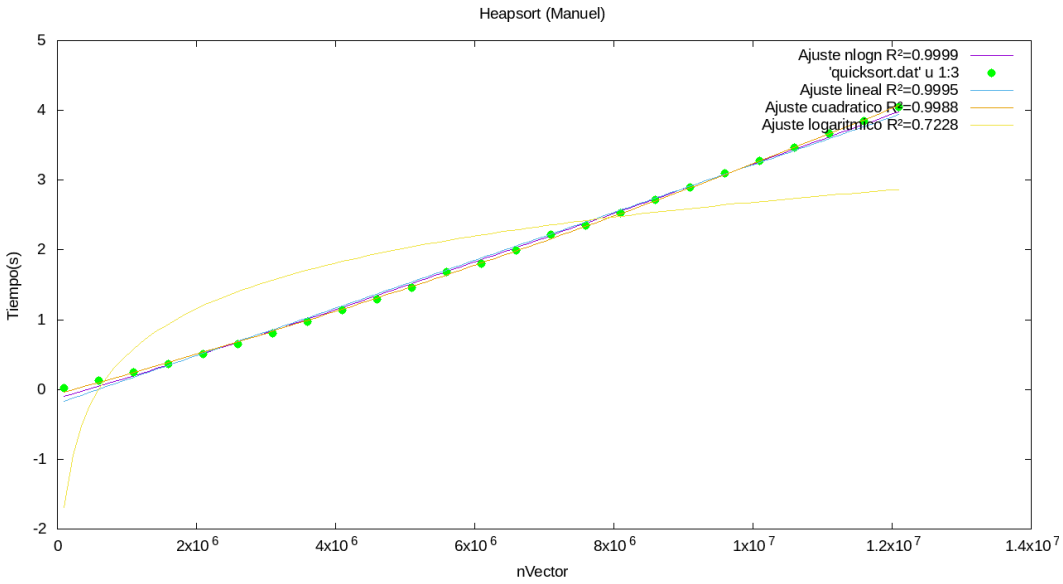
$$T(n) = a_0 \cdot n \log n + a_1$$

,con a_0 y a_1 las constantes ocultas a determinar.

Representando en una gráfica los tiempos que se han obtenido aparece una gráfica como la siguiente:



De nuevo, obtenemos diferencias en los tiempos concretos de cada ordenador, pero no en la forma de la gráfica, a la que como comprobaremos enseguida, se ajusta perfectamente una función de orden $n \log n$. De hecho, concretando y tomando por ejemplo los datos pintados de color verde en la gráfica, se obtiene que con la función $T(n)$ con $a_0 = 2{,}0794 \cdot 10^{-8}$ y $a_1 = -0{,}121342$ se obtiene un coeficiente de correlación $r = 0{,}9998$. Es decir, la variación de los datos en verde está explicada casi por completo por la función $T(n)$.



Podemos también comprobar que se trata del mejor ajuste que se puede obtener, ya que en el caso lineal se tendría $r = 0{,}996$, en el exponencial $r = 0{,}944$, logarítmico $r = 0{,}743$ y con una parábola $r = 0{,}956$.

3.3. Parámetros externos

3.3.1. Optimización

Análogamente a la sección 2.4.1, veamos primero una tabla de datos con los valores, y luego compararemos los resultados obtenidos.

nVector	Quicksort	Quicksort O2	Quicksort O3	Heapsort	Heapsort O2	Heapsort O3
600000	0.0124176	0.0069316	0.0067776	0.016646	9.02E-03	0.0087534
1100000	8.47E-02	0.04573326667	4.55E-02	0.121650	6.47E-02	0.064642
1600000	1.62E-01	0.08646846667	8.65E-02	0.241062	1.29E-01	0.1351321333
2100000	2.47E-01	0.1292997333	1.29E-01	0.367780	1.98E-01	0.2212292
2600000	3.23E-01	0.1734407333	1.73E-01	0.507531	2.75E-01	0.2978595333
3100000	4.05E-01	0.216925	2.17E-01	0.654470	3.57E-01	0.3589892
3600000	4.84E-01	0.2614607333	2.61E-01	0.809746	0.44423	0.4465891333
4100000	0.5704959333	0.3069345333	0.3055174	0.973055	5.37E-01	0.5385930667
4600000	0.6550230667	0.3526368	0.3515678	1.131820	0.6291287333	0.6343986
5100000	0.7434926667	0.3981735333	0.3957756	1.294256	0.7275104667	0.7314488
5600000	0.8261965333	0.4442105333	0.4455494	1.459831	0.8278727333	0.8313005333
6100000	0.9127822	0.4919548667	0.4891088	1.687881	0.9295035333	0.9324008667
6600000	1.0019886	0.5368153333	0.5402283	1.808885	1.033704	1.036194667
7100000	1.107589333	0.585413	0.5842445	1.988263	1.13975	1.14225
7600000	1.170602667	0.6302939333	0.6323895	2.214970	1.246320667	1.248612
8100000	1.260970667	0.6813816	0.6809895	2.346161	1.394364667	1.358221333
8600000	1.348918	0.7264036	0.7286623333	2.529063	1.461824667	1.467294
9100000	1.437366	0.7738471333	0.7759434	2.712820	1.568036	1.568013333
9600000	1.524943333	0.8226232	0.8259160667	2.895624	1.673893333	1.678941333
10100000	1.618905333	0.8697450667	0.8705420	3.092762	1.787406667	1.789988
10600000	1.706428	0.920033	0.9198887	3.282677	1.907242667	1.90301
11100000	1.791969333	0.9676832	0.9628114667	3.469947	2.024172	2.027756667
11600000	1.88164	1.01463	1.0160040	3.666698	2.134653333	2.131654667
12100000	1.982072667	1.062574667	1.0646660	3.842466	2.244662667	2.241628
12600000	2.069074667	1.109184667	1.1148107	4.0461613	2.363446	2.365234
13100000	2.157006	1.161658	1.1604573	4.2811627	2.489776	2.478453333

, datos que representados en un gráfico para ver más visualmente el comportamiento, quedarían como sigue:

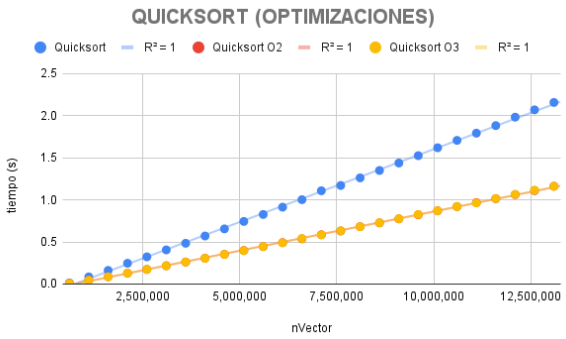


Figura 5: Quicksort (O2, O3)

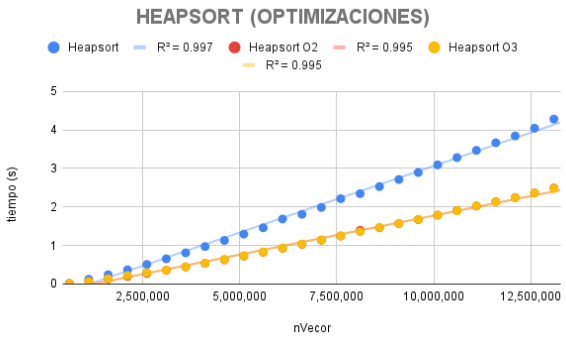


Figura 6: Heapsort (O2, O3)

Con estos dos algoritmos ocurre algo similar al algoritmo de inserción: al compilar con optimizaciones, los tiempos de ejecución bajan considerablemente. No obstante, apenas hay diferencias entre compilar con -O2, y con -O3. Destacamos también que, en cualquiera de los casos, se puede ajustar con gran fiabilidad una función del tipo $n \log n$.

3.3.2. Software

Probando estos dos algoritmos en distintos software se obtiene:

nVector	Heapsort (Linux)	Heapsort (Windows)	Heapsort (VMBox)	Quicksort (Linux)	Quicksort (Windows)	Quicksort (VMBox)
100000	0,014249	0,01566666667	0,01722366667	0,009424533333	0,01566666667	0,0106
600000	0,111585	0,1126666667	0,1088693333	0,06655053333	0,1116666667	0,07165633333
1100000	0,194853	0,2103333333	0,2166733333	0,1226129333	0,2016666667	0,1395916667
1600000	0,272669	0,3323333333	0,327617	0,1768092667	0,301	0,2033903333
2100000	0,372752	0,428	0,4495646667	0,2361205333	0,377	0,2710103333
2600000	0,473122	0,5436666667	0,5790733333	0,3053693333	0,487	0,34023
3100000	0,578324	0,588	0,718614	0,3619072667	0,596	0,412628
3600000	0,688200	0,7063333333	0,858786	0,4216494667	0,704	0,479584
4100000	0,802860	0,8256666667	1,002259333	0,4841651333	0,819	0,5533033333
4600000	0,920331	0,9383333333	1,156473333	0,547418	0,9256666667	0,621675
5100000	1,038131	1,043	1,35	0,60977	1,069	0,6928673333
5600000	1,160697	1,170666667	1,520896667	0,6763012	1,289333333	0,7700713333
6100000	1,284137	1,305	1,627826667	0,7373888667	1,293	0,844388
6600000	1,411327	1,446333333	1,793783333	0,8025293333	1,428666667	0,9145636667
7100000	1,537646	1,605333333	1,960926667	0,8665004	1,586666667	0,9845306667
7600000	1,667612	1,698	2,12846	0,9285397333	2,078333333	1,063133333
8100000	1,801470	1,802666667	2,294196667	0,9973234	1,986	1,130483333
8600000	1,933794	1,917333333	2,453366667	1,060276	2,075	1,203556667
9100000	2,073293	2,067	2,61537	1,122298667	2,215333333	1,27672
9600000	2,206901	2,2	2,78889	1,196016	2,195666667	1,3579
10100000	2,375785	2,344333333	2,962306667	1,251622	2,323333333	1,43374

10600000	2,559600	2,481666667	3,13094	1,320443333	2,455	1,509723333
11100000	2,702133	2,615	3,312746667	1,385769333	2,597666667	1,579236667
11600000	2,848075	2,72	3,506056667	1,451252667	2,729333333	1,666
12100000	2,994913	2,850333333	3,67775	1,522953333	2,923	1,738903333

, obteniendo una representación gráfica como la siguiente:

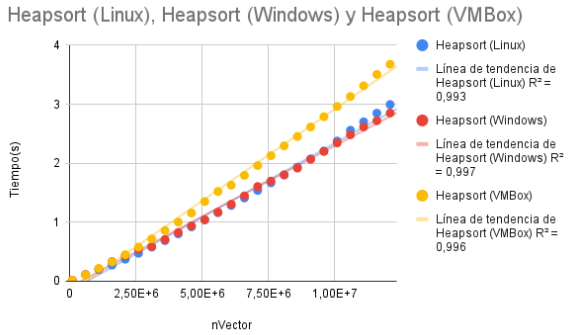


Figura 7: Heapsort (Distintos Software)

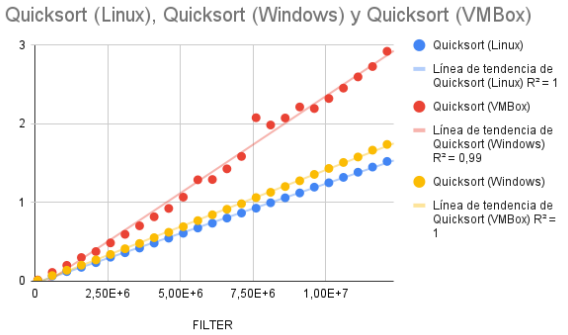


Figura 8: Quicksort (Distintos Software)

Las conclusiones de las gráficas son muy similares a las obtenidas en el punto anterior, con gráficas a las que, en todos los casos, se les puede ajustar prácticamente de forma perfecta una función del tipo $n \log n$, y con ligeras diferencias en los tiempos, con tiempos casi iguales en Windows y Linux, y algo más elevados en la máquina virtual.

4. Comparación de algoritmos de ordenación

Si se recogen los datos obtenidos anteriormente en Quicksort, Heapsort, Inserción y Selección se obtiene una gráfica como la siguiente (con eje Y logarítmico):

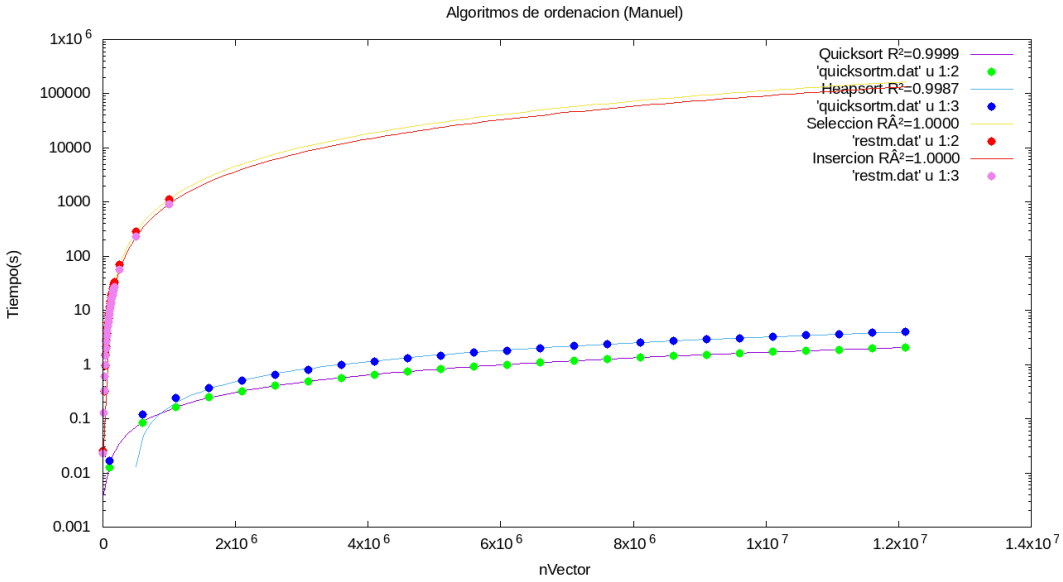


Figura 9: Comparación de algoritmos de ordenación (i7 7700HQ 2.8GHz)

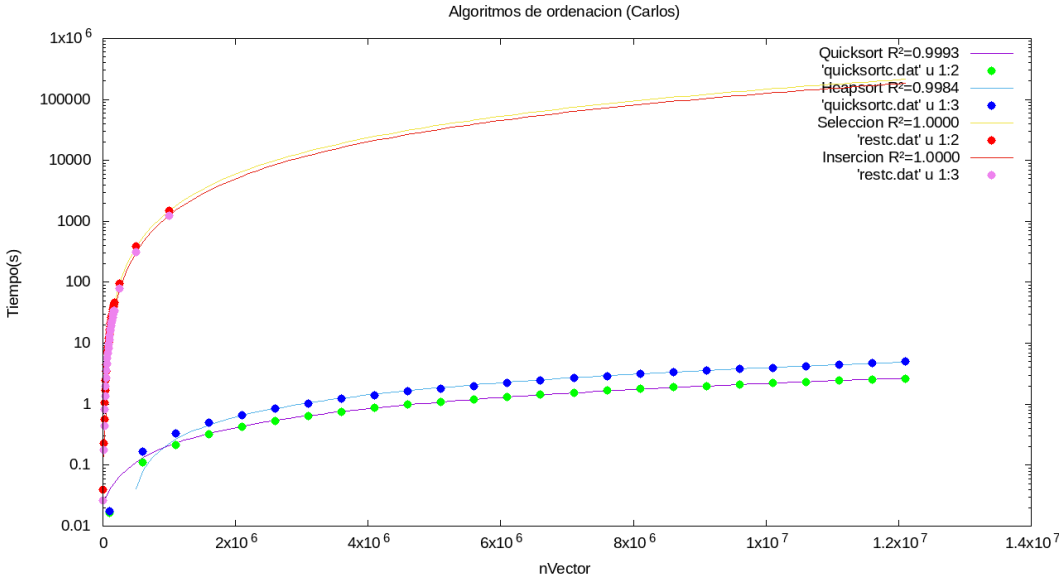


Figura 11: Comparación de algoritmos de ordenación (i7 10750H 1.8GHz)

A pesar de que los tres integrantes tengamos distintas características de hardware al tener distintos computadores, tanto en Quicksort como en Heapsort se observa un crecimiento mucho

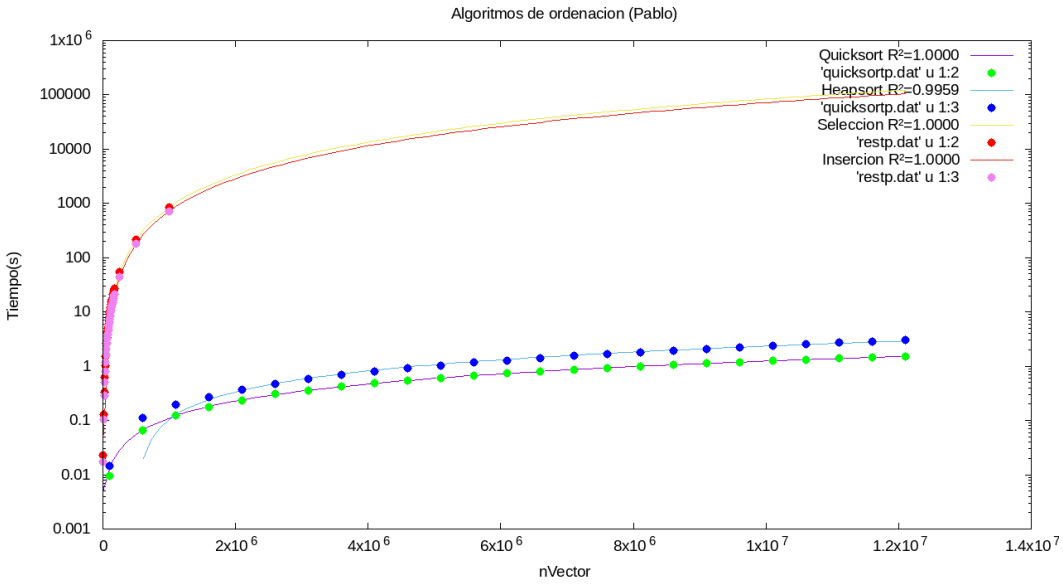


Figura 10: Comparación de algoritmos de ordenación (i7 10750H 2.6GHz)

menor (Caso promedio $n \log n$)) que en el caso de inserción y selección ($O(n^2)$). En ambos casos, los ajustes por mínimos cuadrados son prácticamente perfectos. También observamos que, a pesar de que, en el peor caso (vector ordenado) Quicksort sea uno de los peores algoritmos ($O(n^2)$), en el caso promedio es el que mejores tiempos aporta. De hecho, supera a Heapsort cuyo peor caso tiene la misma eficiencia que el promedio.

Se trata de una gráfica interesante, ya que debido a la enorme bondad del ajuste podemos prácticamente predecir el número máximo de componentes que tendrá el vector que vamos a poder ordenar con nuestro ordenador para un determinado tiempo. Evidentemente, con la gráfica vemos que con uno de los algoritmos recursivos el n al que podremos llegar será bastante mayor.

Sin embargo, otra consecuencia importante que se deriva de la gráfica es el hecho de que para valores de n pequeños importará muy poco qué algoritmo elijamos para ordenar el vector, ya que los tiempos son muy similares. De hecho, es esta la razón por la que en el algoritmo de Quicksort elegimos inserción o selección para ordenar el vector cuando n está por debajo de un umbral.

5. Algoritmo de Floyd

5.1. Eficiencia teórica

```
124  void Floyd(int **M, int dim)
125  {
126    for (int k = 0; k < dim; k++)
127      for (int i = 0; i < dim;i++)
128        for (int j = 0; j < dim;j++)
129          {
130      int sum = M[i][k] + M[k][j];
131          M[i][j] = (M[i][j] > sum) ? sum : M[i][j];
132          }
133  }
```

En este algoritmo vemos por un lado tres bucles *for* anidados que se ejecutan n veces cada uno, siendo n la dimensión. Y dentro del último *for* se realizan operaciones $O(1)$. Por tanto, podemos concluir que el algoritmo es $\mathbf{O(n^3)}$

5.2. Eficiencia empírica e híbrida

Hemos ejecutado el programa con valores desde 200 hasta 1450 en tramos de 50, reproduciendo 15 veces cada valor y haciendo la media de éstos. Dando esta tabla como resultado:

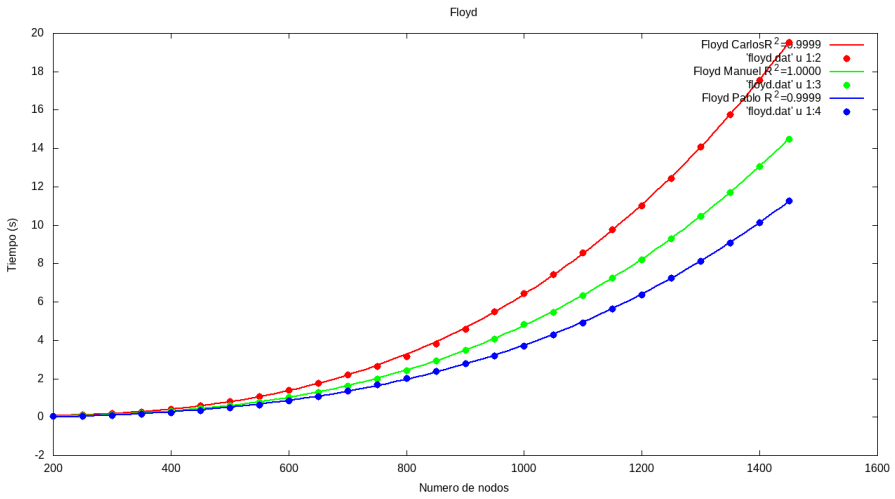
Nvector	Floyd Carlos	Floyd Manuel	Floyd Pablo
200	0.052687	0.038531	0.02977026667
250	0.1048616	0.0743125	0.05757333333
300	0.1767574667	0.1283135	0.1004068
350	0.2799698	0.203885	0.1539906
400	0.4170722667	0.30409	0.2307229333
450	0.5926227333	0.43097	0.3445431333
500	0.8116933333	0.590117	0.4676940667
550	1.078774667	0.7844395	0.6285770667
600	1.396923333	1.018325	0.8352748
650	1.773050667	1.292065	1.080143333
700	2.201822667	1.61738	1.360819333
750	2.63458	1.97988	1.692018667
800	3.154136	2.43431	2.016167333
850	3.806810667	2.93247	2.396283333
900	4.579523333	3.47463	2.781681333
950	5.505131333	4.05779	3.171832
1000	6.436344667	4.847055	3.697248
1050	7.43951	5.46877	4.272787333
1100	8.561508	6.31509	4.917258667
1150	9.779517333	7.253315	5.632534667
1200	11.02625333	8.18928	6.375234667
1250	12.44532	9.28862	7.233436667

Nvector	Floyd Carlos	Floyd Manuel	Floyd Pablo
1300	14.07688	10.45405	8.113726
1350	15.74758667	11.7167	9.090595333
1400	17.56455333	13.07075	10.12635333
1450	19.52052	14.49155	11.26074667

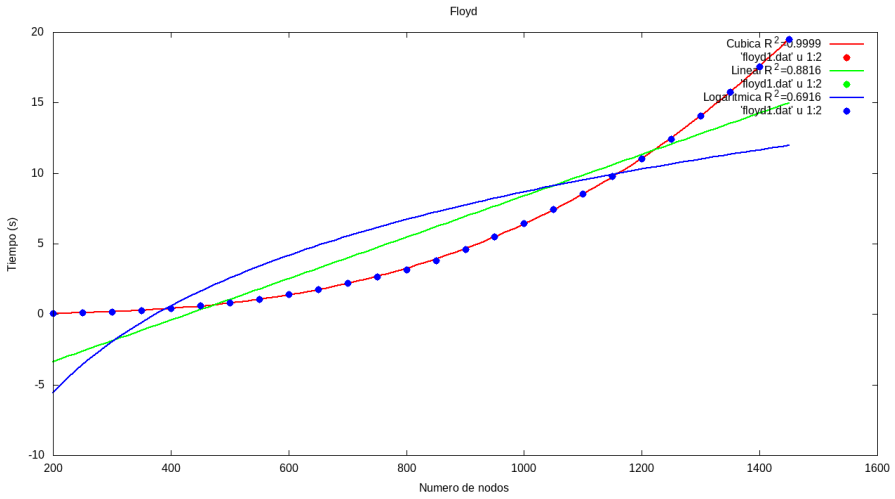
En este caso, para hallar la eficiencia híbrida del algoritmo de Floyd, usaremos una gráfica con los tiempos obtenidos en la tabla anterior y hallaremos la función que más se ajusta a los datos. De la misma forma que se ha explicado en el apartado de eficiencia teórica, la función será de la forma:

$$T(n) = a_0 \cdot n^3 + a_1 \cdot n^2 + a_2 \cdot n + a3$$

donde a_0, a_1, a_2 y a_3 son constantes reales que van a depender de las características de cada ordenador en el cuál se ha ejecutado el programa. Representado en una gráfica se quedaría:



Podemos comparar el ajuste de la cúbica con otras curvas en esta gráfica:



Como podemos observar, la cúbica es la que mejor se ajusta a los datos, a diferencia de la logarítmica y la lineal, con un coeficiente de determinación $R^2 = 0{,}9999$. Especificando más, con los datos de gnuplot vemos que las constantes son: $a_0 = 6{,}38989 \cdot 10^9$, $a_1 = 1{,}2563 \cdot 10^7$, $a_2 = -0{,}000186463$ y $a_3 = 0{,}0526725$.

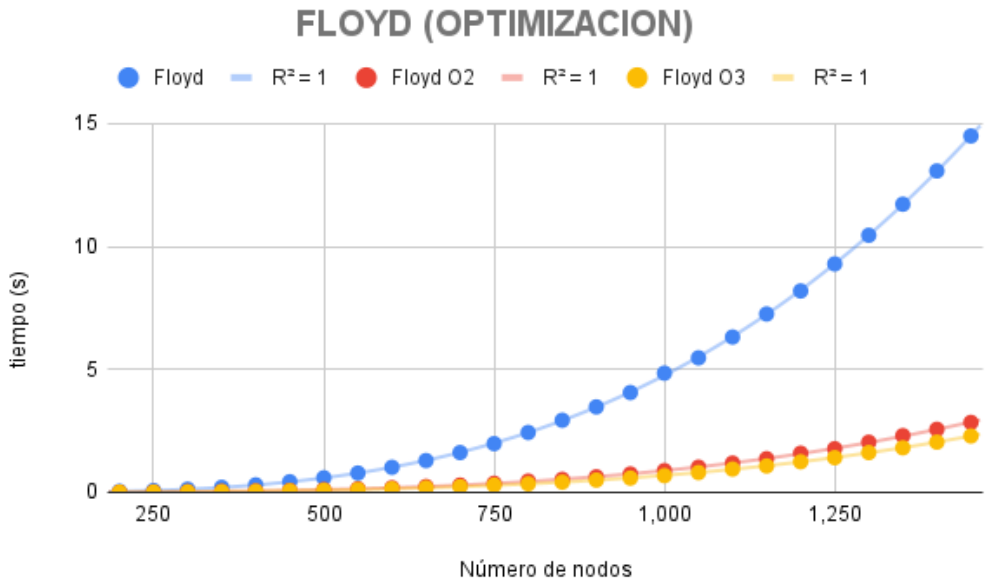
5.3. Parámetros externos

5.3.1. Optimización

Presentemos ahora los valores obtenidos al compilar este código con optimizaciones de O2 y de O3. Similarmente a apartados anteriores, primero mostraremos una tabla con los resultados explícitos, para después hacer una gráfica y así poder compararlos más visualmente.

nVector	Floyd	Floyd O2	Floyd O3
200	0.038531	0.007487	0.0058385
250	7.43E-02	0.014144	1.13E-02
300	1.28E-01	0.0237375	1.83E-02
350	2.04E-01	0.0374525	2.93E-02
400	3.04E-01	0.0557935	4.35E-02
450	4.31E-01	0.0791645	6.16E-02
500	5.90E-01	0.1086775	8.44E-02
550	0.7844395	0.1443705	0.1118315
600	1.018325	0.186796	0.1449935
650	1.292065	0.2371785	0.1834595
700	1.61738	0.2952755	0.2289420
750	1.97988	0.367927	0.2935060
800	2.43431	0.459574	0.3453640
850	2.93247	0.529246	0.4144620
900	3.47463	0.631737	0.4977030
950	4.05779	0.7478075	0.5777850
1000	4.847055	0.8820075	0.678323
1050	5.46877	1.02332	0.805058
1100	6.31509	1.188975	0.9456555
1150	7.253315	1.36653	1.0732000
1200	8.18928	1.59141	1.2442450
1250	9.28862	1.773485	1.410195
1300	10.45405	2.03878	1.6176750
1350	11.7167	2.304795	1.8116750
1400	13.07075	2.563345	2.0432350
1450	14.49155	2.843285	2.2912250

las gráficas quedarían de la siguiente manera:



En este caso podemos ver cómo al compilar con optimizaciones, el tiempo de ejecución disminuye drásticamente, conservando en cualquier caso la forma de una función polinomial de grado 3.

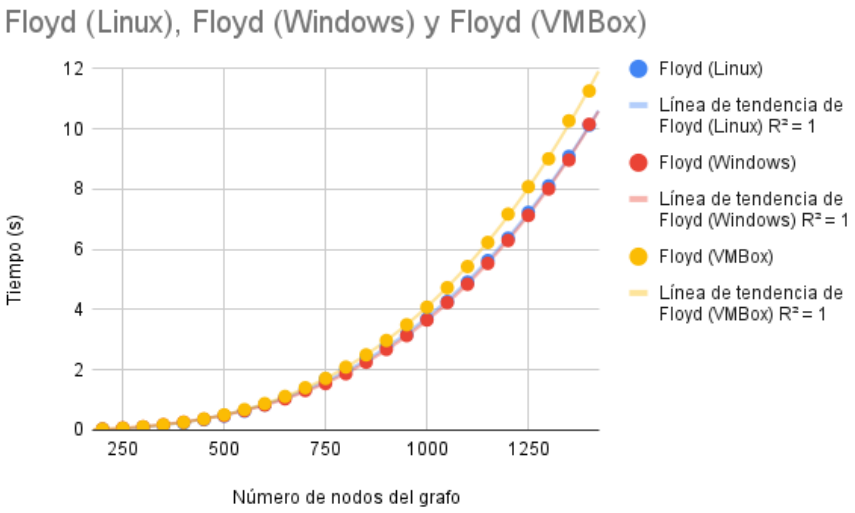
Al igual que con otros algoritmos ya vistos, los resultados con la optimización con -O3 no difieren mucho de los de la optimización con -O2, aunque claramente la mejora.

5.3.2. Software

Realizando las mismas pruebas con distinto software se obtienen los siguientes datos:

nNodes	Floyd (Linux)	Floyd (Windows)	Floyd (VMBox)
200	0,02977026667	0,035	0,03372333333
250	0,05757333333	0,07066666667	0,065018
300	0,1004068	0,111	0,110608
350	0,1539906	0,1766666667	0,1776446667
400	0,2307229333	0,2633333333	0,260372
450	0,3445431333	0,363	0,3708483333
500	0,4676940667	0,48	0,5083066667
550	0,6285770667	0,645	0,6806953333
600	0,8352748	0,8323333333	0,8794413333
650	1,080143333	1,038333333	1,118943333
700	1,360819333	1,315666667	1,408173333
750	1,692018667	1,545666667	1,717546667
800	2,016167333	1,873333333	2,09302
850	2,396283333	2,254	2,49901
900	2,781681333	2,681333333	2,977303333

950	3,171832	3,139	3,49794
1000	3,697248	3,656333333	4,082276667
1050	4,272787333	4,236	4,731023333
1100	4,917258667	4,843	5,4316
1150	5,632534667	5,536	6,233856667
1200	6,375234667	6,296	7,173463333
1250	7,233436667	7,128	8,083303333
1300	8,113726	8,009	9,01133
1350	9,090595333	8,969	10,27286667
1400	10,12635333	10,15566667	11,26216667



Para los tres casos, existe una función cúbica que se ajusta perfectamente a los datos, aunque se observa que los datos obtenidos en Linux y Windows son mejores que en una máquina virtual.

6. Algoritmo de Hanoi

6.1. Eficiencia teórica

```
30  void hanoi (int M, int i, int j)
31  {
32    if (M > 0)
33      {
34        hanoi(M-1, i, 6-i-j);
35        hanoi (M-1, 6-i-j, j);
36    }
37  }
```

Al ser un algoritmo recursivo, debemos buscar la recurrencia. Como tras entrar en el if ($O(1)$), se llama a sí misma dos veces, la ecuación de recurrencia del algoritmo es:

$$T_n = 2T_{n-1} + 1$$

Despejando y operando, resolvemos la recurrencia:

$$(x-2)(x-1) = 0$$

$$t_n = c_1 \cdot 2^n + c_2 * 1^n = c_1 \cdot 2^n + c_2$$

Y podemos concluir entonces que nuestro algoritmo tiene eficiencia $\mathbf{O(2^n)}$.

6.2. Eficiencia empírica e híbrida

Hemos ejecutado el programa con valores desde 1 hasta 25 en tramos de 1, reproduciendo 15 veces cada valor y haciendo la media de éstos. Dando esta tabla como resultado:

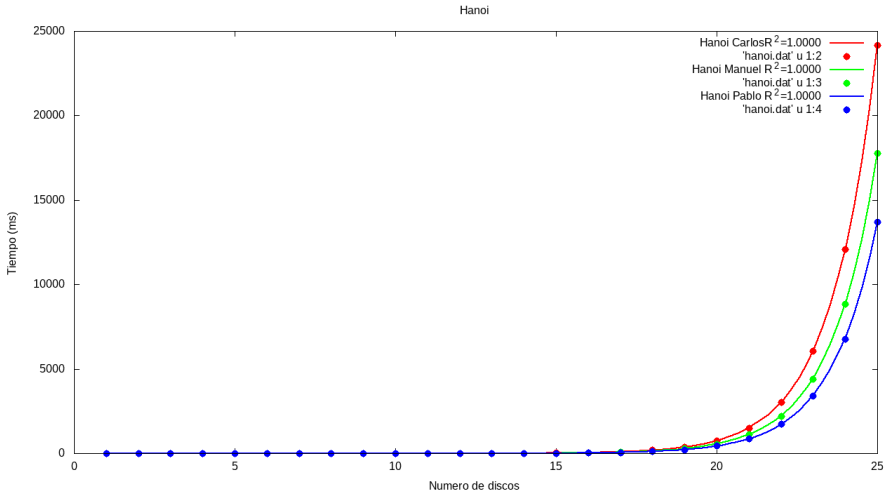
Ndisk	Hanoi Carlos	Hanoi Manuel	Hanoi Pablo
1	0.0009868666667	0.000733	0.0005750666667
2	0.002253133333	0.001778	0.001369933333
3	0.005174133333	0.003895	0.003072866667
4	0.01010526667	0.008128	0.006279333333
5	0.02245033333	0.016380	0.0142532
6	0.04564066667	0.033264	0.03095166667
7	0.09162593333	0.067158	0.061898
8	0.1838962	0.134278	0.1193006667
9	0.3682004667	0.268360	0.2206479333
10	0.7369194667	0.535656	0.4922432667
11	1.473742667	1.072800	0.9050988667
12	2.894882	2.155610	1.775977333
13	5.812209333	4.305080	3.378328

Ndisk	Hanoi Carlos	Hanoi Manuel	Hanoi Pablo
14	11.72312	8.630080	6.650476
15	23.46904	17.246400	13.19578667
16	47.26666667	37.650000	31.74
17	94.45	68.300000	56.52
18	188.8333333	136.600000	110.9933333
19	377.6133333	276.600000	221.06
20	755.2642857	553.400000	442.3066667
21	1510.613333	1104.250000	880.0533333
22	3021.493333	2207.200000	1735.566667
23	6044.206667	4415.600000	3416.953333
24	12081.36	8837.900000	6771.126667
25	24172.91333	17757.850000	13708.72667

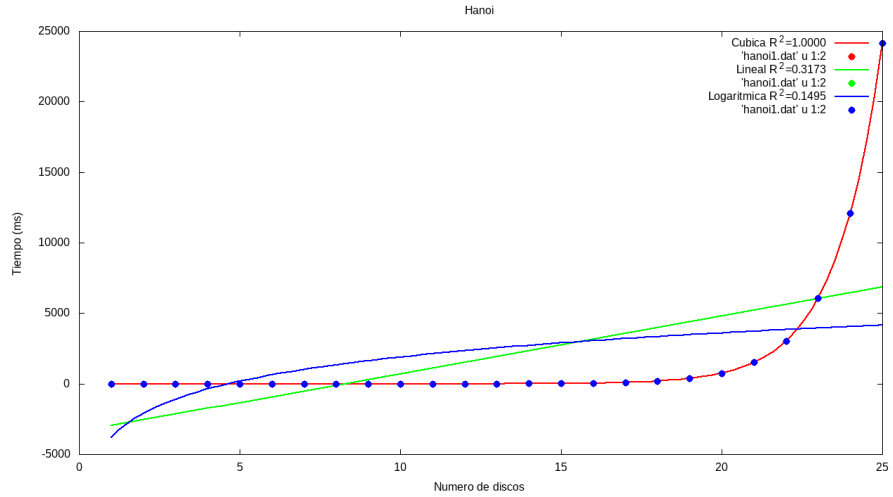
Ahora vamos a hallar la eficiencia híbrida del algoritmo de Hanoi, crearemos una gráfica con los tiempos obtenidos en la tabla anterior y encontraremos la función que más se ajusta a los datos. De la misma forma que se ha explicado en el apartado de eficiencia teórica, la función será de la forma:

$$T(n) = a_0 \cdot 2^2 + a_1$$

donde a_0 y a_1 son constantes reales que van a depender de las características de cada ordenador en el cuál se ha ejecutado el programa. Representado en una gráfica se quedaría:



Podemos comparar el ajuste de la exponencial con otras curvas en esta gráfica:



Como podemos observar, la exponencial es la que mejor se ajusta a los datos, a diferencia de la logarítmica y la lineal, con un coeficiente de correlación $R^2 = 1$. Especificando más, con los datos de gnuplot vemos que las constantes son: $a_0 = 0{,}000720312$ y $a_1 = 0{,}999999$.

6.3. Parámetros externos

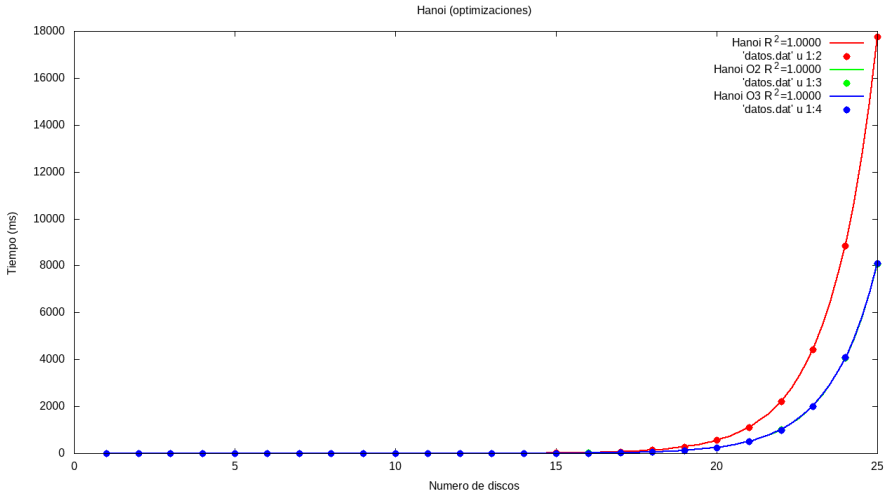
6.3.1. Optimización

Veamos los datos que hemos obtenido al compilar con optimizaciones O2 y O3:

nDisks	Hanoi	Hanoi O2	Hanoi O3
1	0.000733	0.000241	0.000077
2	0.001778	0.000633	0.000302
3	0.003895	0.001515	0.001040
4	0.008128	0.003152	0.002613
5	0.016380	0.006490	0.005834
6	0.033264	0.013439	0.012896
7	0.067158	0.028569	0.027783
8	0.134278	0.056598	0.056946
9	0.268360	0.117059	0.116474
10	0.535656	0.233789	0.235396
11	1.072800	0.474831	0.469234
12	2.155610	0.949974	0.942533
13	4.305080	1.923660	1.903145
14	8.630080	3.924175	3.825470
15	17.246400	7.790175	7.677440
16	37.650000	19.000000	14.900000
17	68.300000	29.900000	31.600000
18	136.600000	62.350000	59.650000
19	276.600000	119.500000	119.350000

nDisks	Hanoi	Hanoi O2	Hanoi O3
20	553.400000	244.450000	240.900000
21	1104.250000	508.800000	500.600000
22	2207.200000	1028.500000	991.200000
23	4415.600000	1990.550000	2009.000000
24	8837.900000	4077.250000	4090.600000
25	17757.850000	8075.850000	8095.550000

Una vez con los datos recogidos, veamos las gráficas de las tres nubes de puntos, y comparémoslas.



Una vez más, vemos que, como esperábamos, los códigos optimizados tardan bastante menos que el que está sin optimizar. Además, vemos que los puntos de los tiempos de ejecución con O2 y con O3 se solapan; esto es, tardan prácticamente lo mismo, no hay apenas diferencia entre O2 y O3.

También destacamos que al haber optimizado, la función que mejor ajusta los datos sigue siendo una función del tipo 2^n.

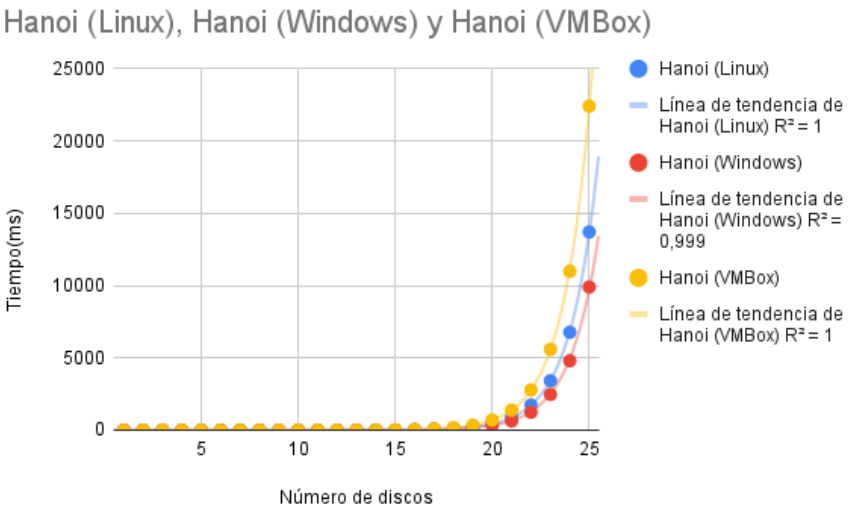
6.3.2. Software

Tras ejecutar los programas en los tres software considerados, se obtienen los siguientes datos (medidos en ms):

Ndisks	Hanoi (Linux)	Hanoi (Windows)	Hanoi (VMBox)
1	0,0005750666667	0,0006666666667	0,001264666667
2	0,001369933333	0,001666666667	0,002659666667
3	0,003072866667	0,002666666667	0,005569333333

4	0,006279333333	0,005333333333	0,01142766667
5	0,0142532	0,01033333333	0,02309733333
6	0,03095166667	0,02066666667	0,04451466667
7	0,061898	0,042	0,087731
8	0,1193006667	0,08266666667	0,1761683333
9	0,2206479333	0,1593333333	0,3501703333
10	0,4922432667	0,3156666667	0,7009396667
11	0,9050988667	0,6393333333	1,39975
12	1,775977333	1,258	2,796416667
13	3,378328	2,439	5,595183333
14	6,650476	4,975	10,98013333
15	13,19578667	9,649333333	21,87803333
16	31,74	33,33333333	46,36666667
17	56,52	66,66666667	89,93333333
18	110,9933333	100	174
19	221,06	166,6666667	339,5333333
20	442,3066667	333,3333333	696,1
21	880,0533333	633,3333333	1381
22	1735,566667	1233,333333	2780,833333
23	3416,953333	2466,666667	5586,566667
24	6771,126667	4800	10989,73333
25	13708,72667	9900	22419,26667

y tras realizar las gráficas:



se obtienen las mismas conclusiones que en casos anteriores.

7. Conclusión

Finalmente, escribiremos una pequeña conclusión sobre lo aprendido en la práctica.

Las conclusiones del análisis se basan, por supuesto, en la importancia que, como hemos aprendido, tienen el análisis teórico y empírico de la eficiencia de los algoritmos. A pesar de que los tres integrantes del grupo hemos obtenido tiempos muy distintos en todos los algoritmos, siempre se ha mantenido una constante en todo el proceso: la eficiencia de los algoritmos.

Cuando hemos analizado algoritmos cuadráticos, todos hemos obtenido una parábola como mejor aproximación a la nube de puntos, lo mismo ha ocurrido con algoritmos de orden $n \log n$ y así con el de Floyd y el de Hanoi.

El trabajo realizado con los parámetros externos, como la compilación o el uso de distintos tipos de software, no ha hecho más que reforzar la idea de que, a pesar de que dichos parámetros puedan cambiar ligeramente los tiempos obtenidos (en el caso de la optimización en gran medida) ni mucho menos tienen un efecto en el tipo de función obtenida en el apartado de eficiencia híbrida.

En todo el trabajo, la importancia del estudio de la eficiencia ha probado ser enorme, ya no solo para saber hasta donde podíamos llegar en un determinado algoritmo, que en algoritmos como Floyd o Hanoi ha sido vital; sino también al estudiar el peor y mejor caso de inserción y selección. Lo que parecía que iba a ser en ambos algoritmos una enorme diferencia entre caso peor, promedio y mejor ha demostrado que dependiendo de cómo funciona el algoritmo no tiene por qué ser así.

La práctica también ha permitido familiarizarnos con algoritmos, que nunca habíamos visto como Floyd y Hanoi, lo que ha hecho que podamos aprender algoritmos con eficiencias que nunca habíamos visto.

Como conclusión final, podríamos decir que, a pesar de que si repitiéramos la práctica en unos años los tiempos que obtendríamos serían distintos a los que hoy tenemos, ya que seguramente tendríamos un nuevo ordenador; la eficiencia de los algoritmos seguiría siendo completamente la misma, será la constante que se mantendrá siempre, pase el tiempo que pase, ejecutemos el algoritmo donde lo ejecutemos.

8. Herramientas y ordenadores utilizados

8.1. Herramientas

- Google Spreadsheets
- gnuplot 5.2
- Visual Studio Code y Oracle VM VirtualBox
- Códigos suministrados por el profesorado de Algorítmica
- Overleaf (LaTeX)

8.2. Ordenadores de los integrantes del grupo

	MANUEL	PABLO	CARLOS
Arquitectura:	x86_64	x86_64	x86_64
Orden bytes:	Little Endian	Little Endian	Little Endian
Address sizes:	39 bits physical, 48 bits virtual	39 bits physical, 48 bits virtual	39 bits physical, 48 bits virtual
CPU(s):	8	12	12
Hilo(s) por núcleo:	2	2	2
Núcleo(s) por «socket»:	4	6	6
«Socket(s)»	1	1	1
ID de fabricante:	GenuineIntel	GenuineIntel	GenuineIntel
Modelo:	158	165	165
Nombre del modelo:	Intel i7-7700HQ @ 2.80GHz	Intel i7-10750H @ 2.60GHz	Intel i7-10750H @ 1.8GHz
Revisión:	9	2	2
CPU MHz:	2.800.000	2.600	1.800
CPU MHz máx.:	3800	5000	2600
CPU MHz mín.:	800	800	800.110
BogoMIPS:	5599.85	5199.98	5199.98
Caché L1d:	128 KiB	192 KiB	192 KiB
Caché L1i:	128 KiB	192 KiB	192 KiB
Caché L2:	1 MiB	1.5 MiB	1.5 MiB
Caché L3:	6 MiB	12 MiB	12 MiB
Memoria RAM:	16384 MB	16384 MB	8192 MB

Se verifica lo que se ha ido viendo en la memoria: los tiempos obtenidos por el ordenador de Pablo son mejores que los de Manuel y que, finalmente los de Carlos. Claramente se debe a una mejor CPU, mayor cantidad de RAM y/o mayor potencia.