Práctica 1: Análisis de Eficiencia de Algoritmos

Manuel Vicente Bolaños Quesada
Pablo Gálvez Ortigosa
Carlos García Jiménez

30/3/2022

- Algoritmo de inserción
- Algoritmo de selección
- Algoritmo Heapsort
- Algoritmo Quicksort

} Ordenación de vectores

- Algoritmo de las torres de Hanoi
- Algoritmo de Floyd $\longrightarrow$ Análisis de grafos

Eficiencia teórica

```
117
118  static void seleccion_lims(int T[], int inicial, int final)
119  {
120    int i, j, indice_menor;
121    int menor, aux;
122    for (i = inicial; i < final - 1; i++) {
123      indice_menor = i;
124      menor = T[i];
125      for (j = i; j < final; j++)
126        if (T[j] < menor) {
127    indice_menor = j;
128    menor = T[j];
129        }
130      aux = T[i];
131      T[i] = T[indice_menor];
132      T[indice_menor] = aux;
133    };
134  }
135
```

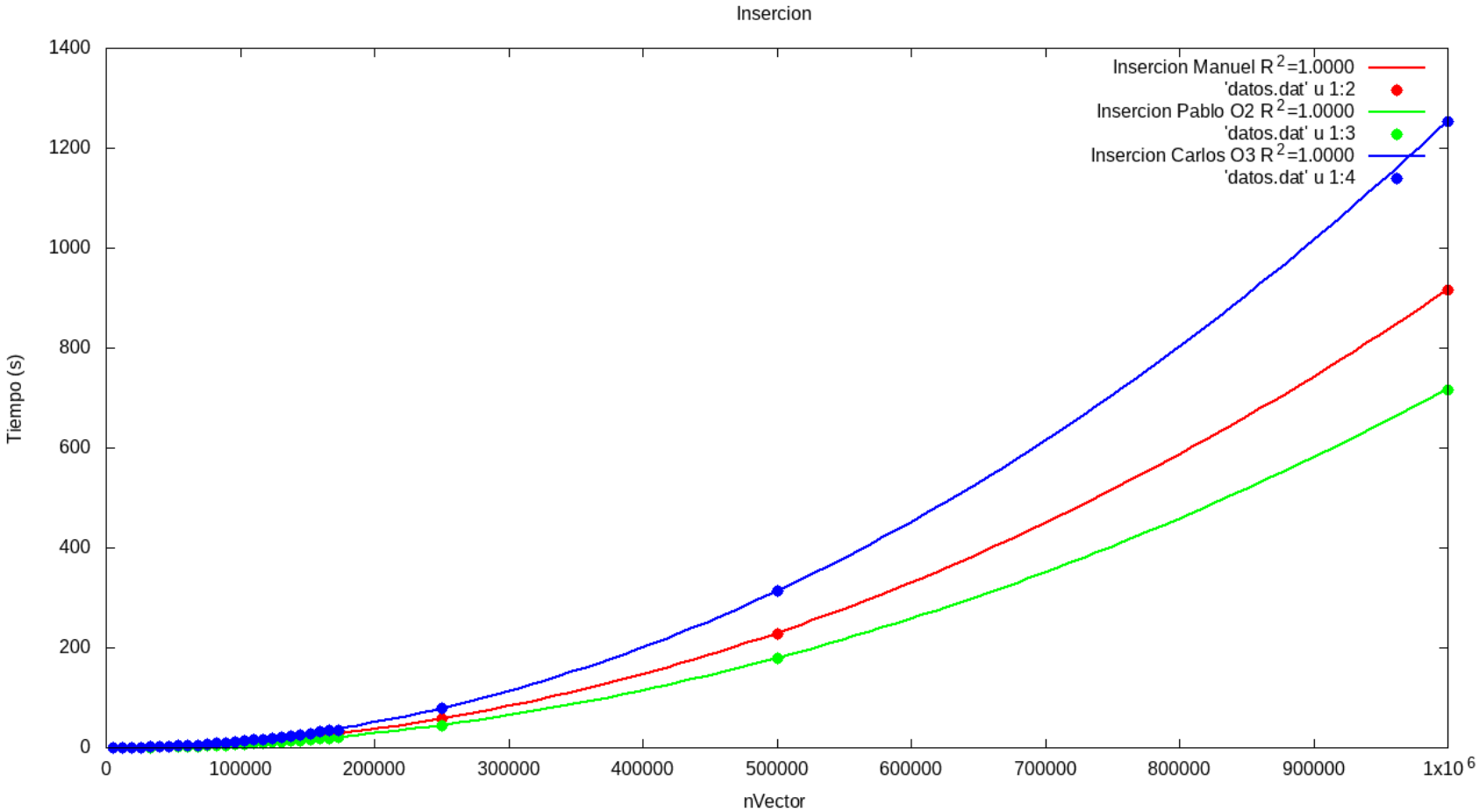
$$T(n) = a_0 n^2 + a_1 n + a_2$$

$$O(n^2)$$

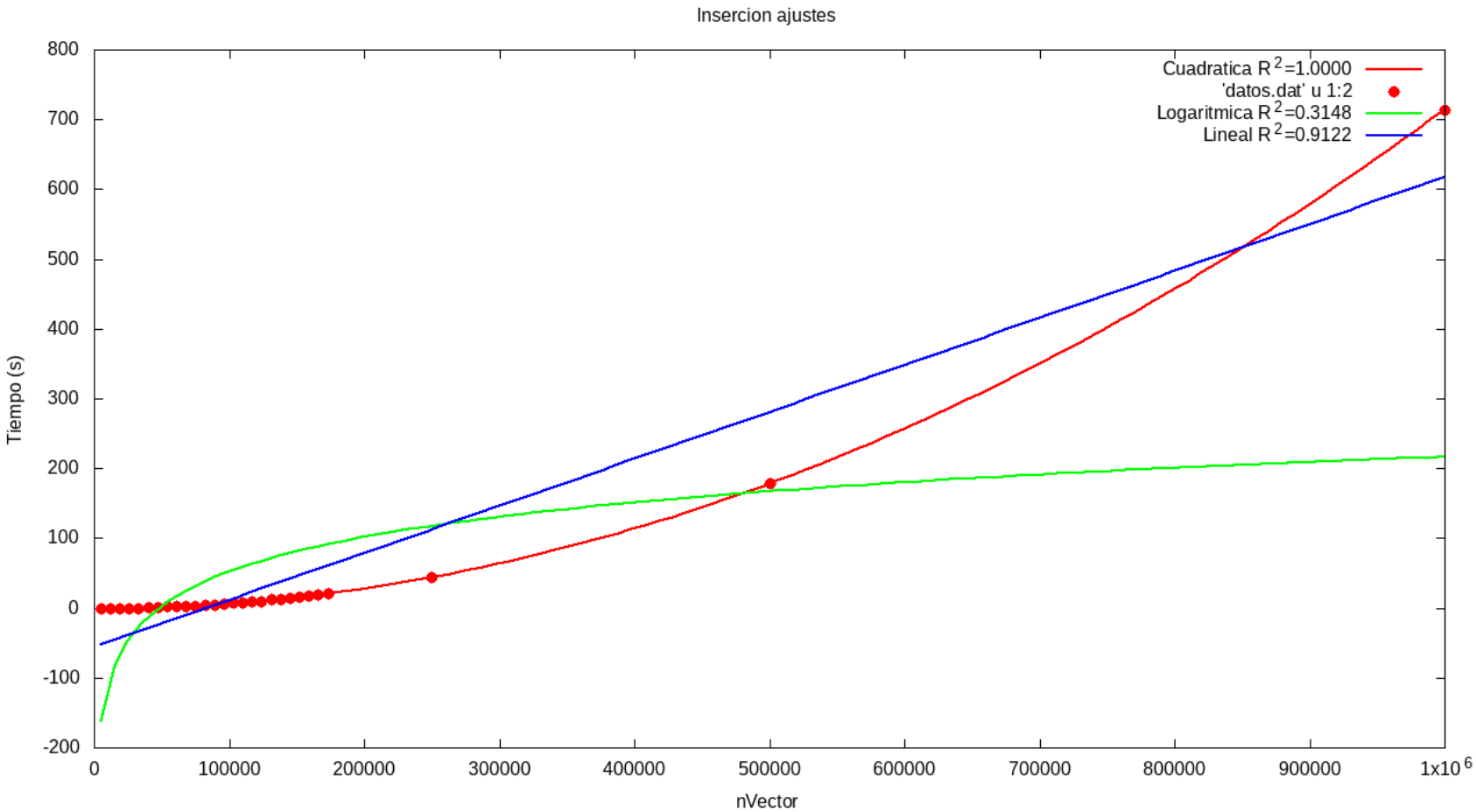
Eficiencia empírica

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

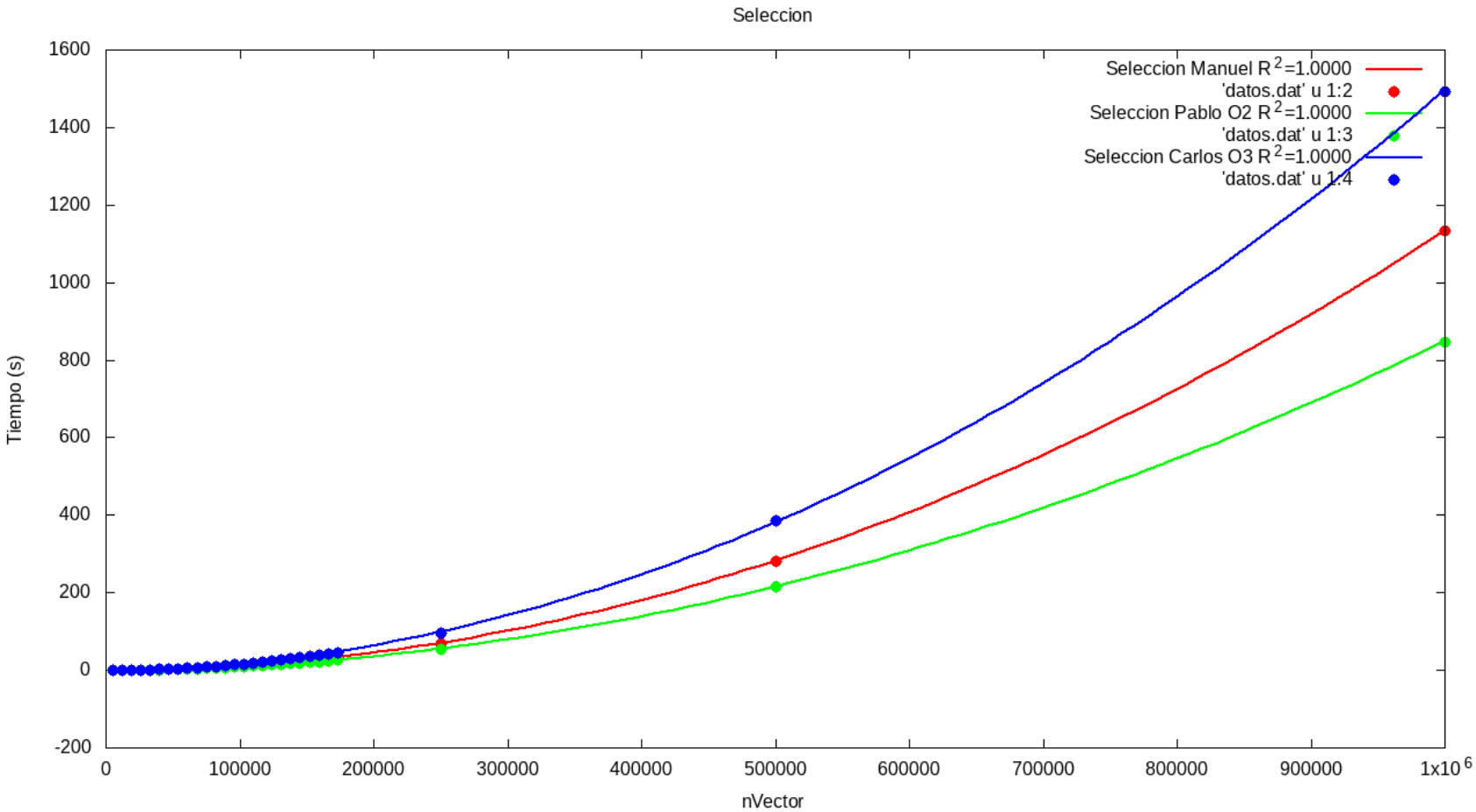
Inserción. Ajuste del grupo



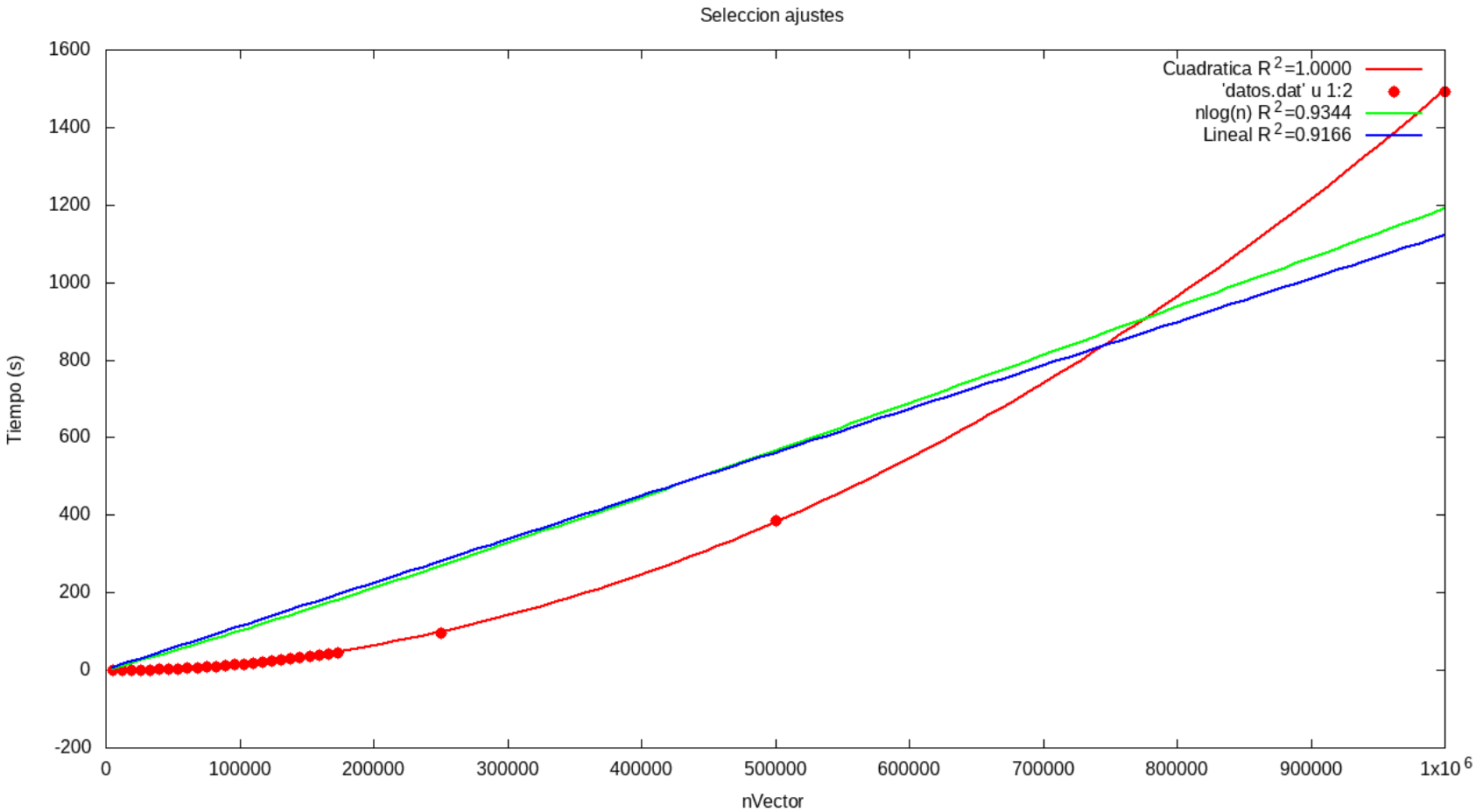
Inserción. Eficiencia híbrida



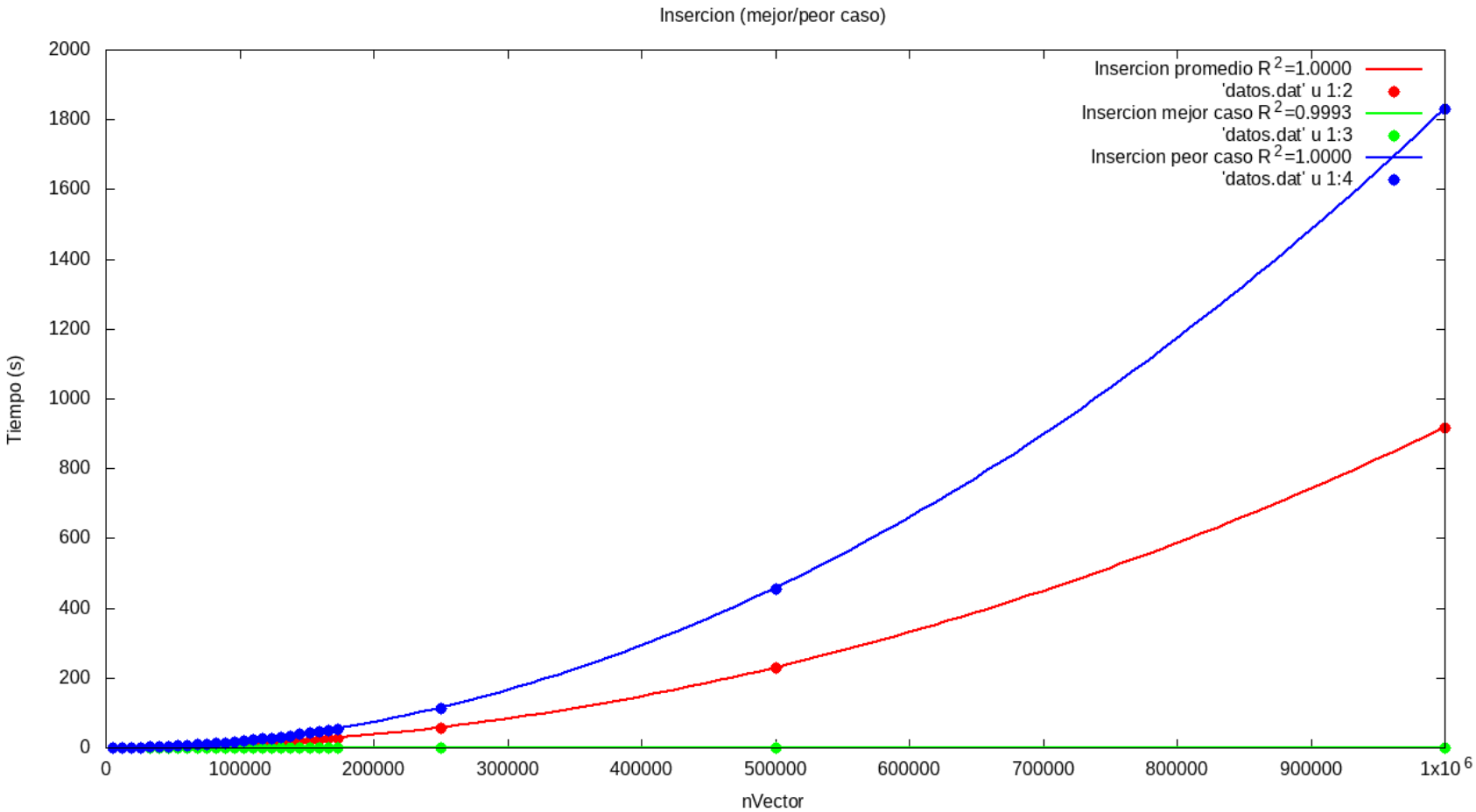
Selección. Ajuste del grupo



Selección. Eficiencia híbrida



Inserción

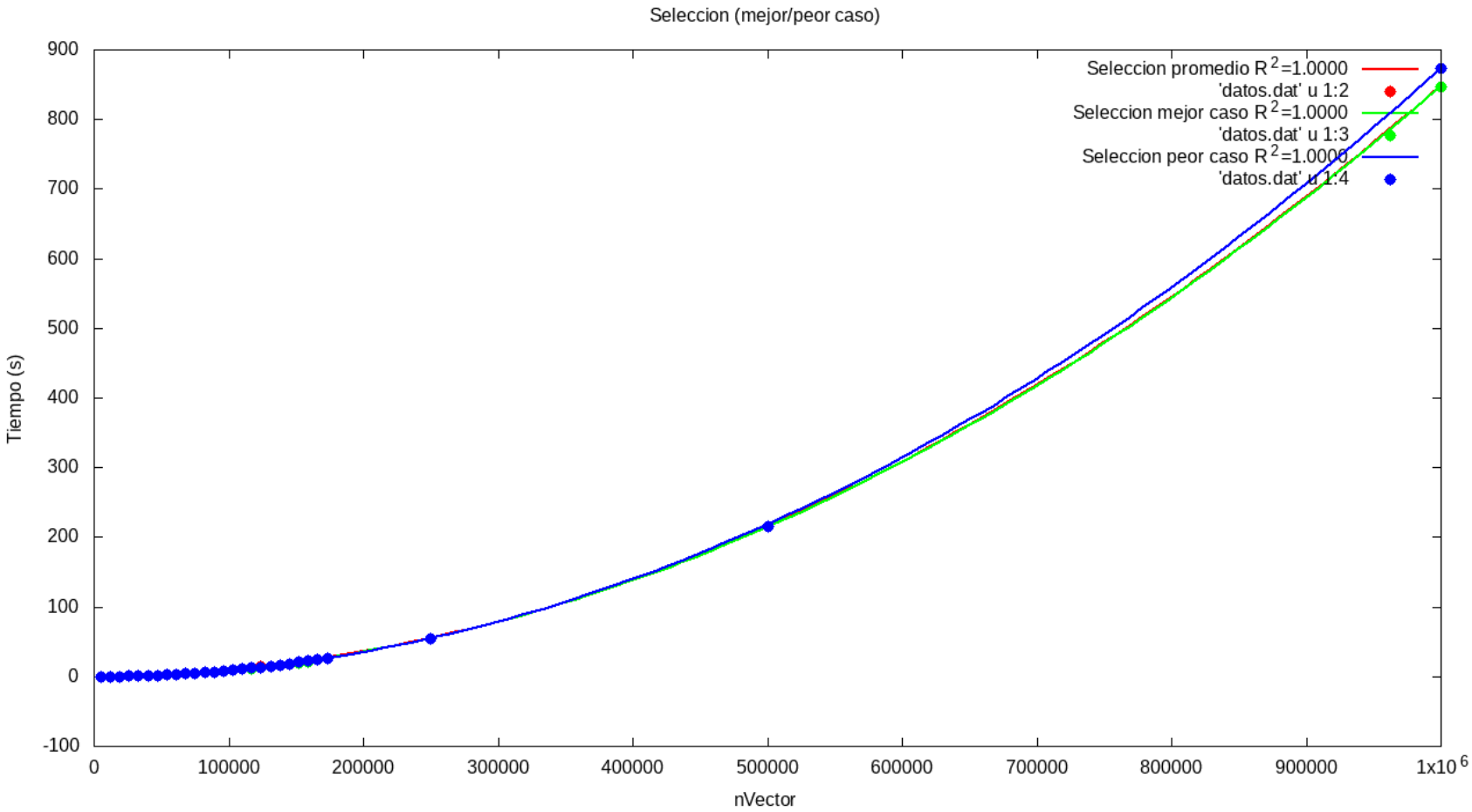

Insercion (mejor/peor caso)
Tiempo (s)
nVector
Insercion promedio R^2=1.0000
'datos.dat' u 1:2
Insercion mejor caso R^2=0.9993
'datos.dat' u 1:3
Insercion peor caso R^2=1.0000
'datos.dat' u 1:4
0
200
400
600
800
1000
1200
1400
1600
1800
2000
0
100000
200000
300000
400000
500000
600000
700000
800000
900000
1x10^6

Inserción: Explicación peor y mejor caso

```
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  }
```

1. Bucle for $\rightarrow n-1$ iteraciones siempre
2. Dependiendo del orden del vector, mayor o menor número de iteraciones del bucle while (vector ordenado o vector inversamente ordenado)
3. Mejor caso (ordenado): $O(n)$ (ninguna iteración del while)
4. Peor caso (inv. ordenado): $O(n^2)$ (todas las iteraciones posibles)

Selección


Seleccion (mejor/peor caso)
Seleccion promedio R^2=1.0000
'datos.dat' u 1:2
Seleccion mejor caso R^2=1.0000
'datos.dat' u 1:3
Seleccion peor caso R^2=1.0000
'datos.dat' u 1:4
Tiempo (s)
nVector
900
800
700
600
500
400
300
200
100
0
-100
0
100000
200000
300000
400000
500000
600000
700000
800000
900000
1x10^6

Selección: Explicación peor y mejor caso (I)

```
117
118  static void seleccion_lims(int T[], int inicial, int final)
119  {
120    int i, j, indice_menor;
121    int menor, aux;
122    for (i = inicial; i < final - 1; i++) {
123      indice_menor = i;
124      menor = T[i];
125      for (j = i; j < final; j++)
126        if (T[j] < menor) {
127    indice_menor = j;
128    menor = T[j];
129        }
130      aux = T[i];
131      T[i] = T[indice_menor];
132      T[indice_menor] = aux;
133    };
134  }
135
```

1. El algoritmo de selección siempre realiza el mismo número de iteraciones: primero $n-1$, después $n-2$, ... y así hasta uno, es decir:

$$\sum_{k=1}^{n-1} k = \frac{n(n-1)}{2} \text{comparaciones}$$

Selección: Explicación peor y mejor caso (II)

Por tanto, independientemente del orden del vector, siempre tardará lo mismo $\rightarrow$ Mejor caso = Caso promedio = Peor caso

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  }
```

$$k_i = 2k_{i-1} + 1$$

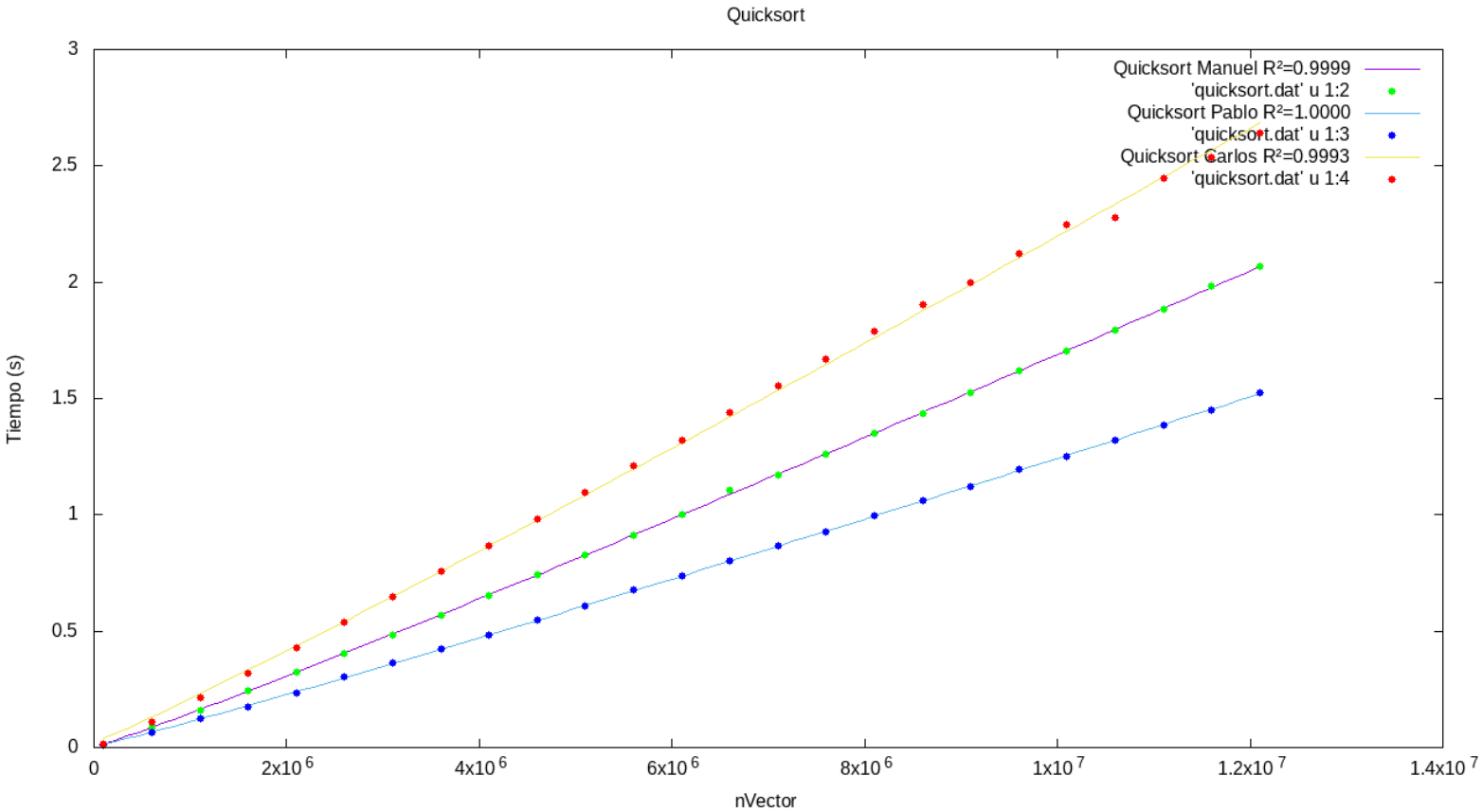
Máx. iteraciones de reajustar:
$log_2(n-2) - 1$

$$\mathbf{O(n \log n)}$$

Eficiencia empírica

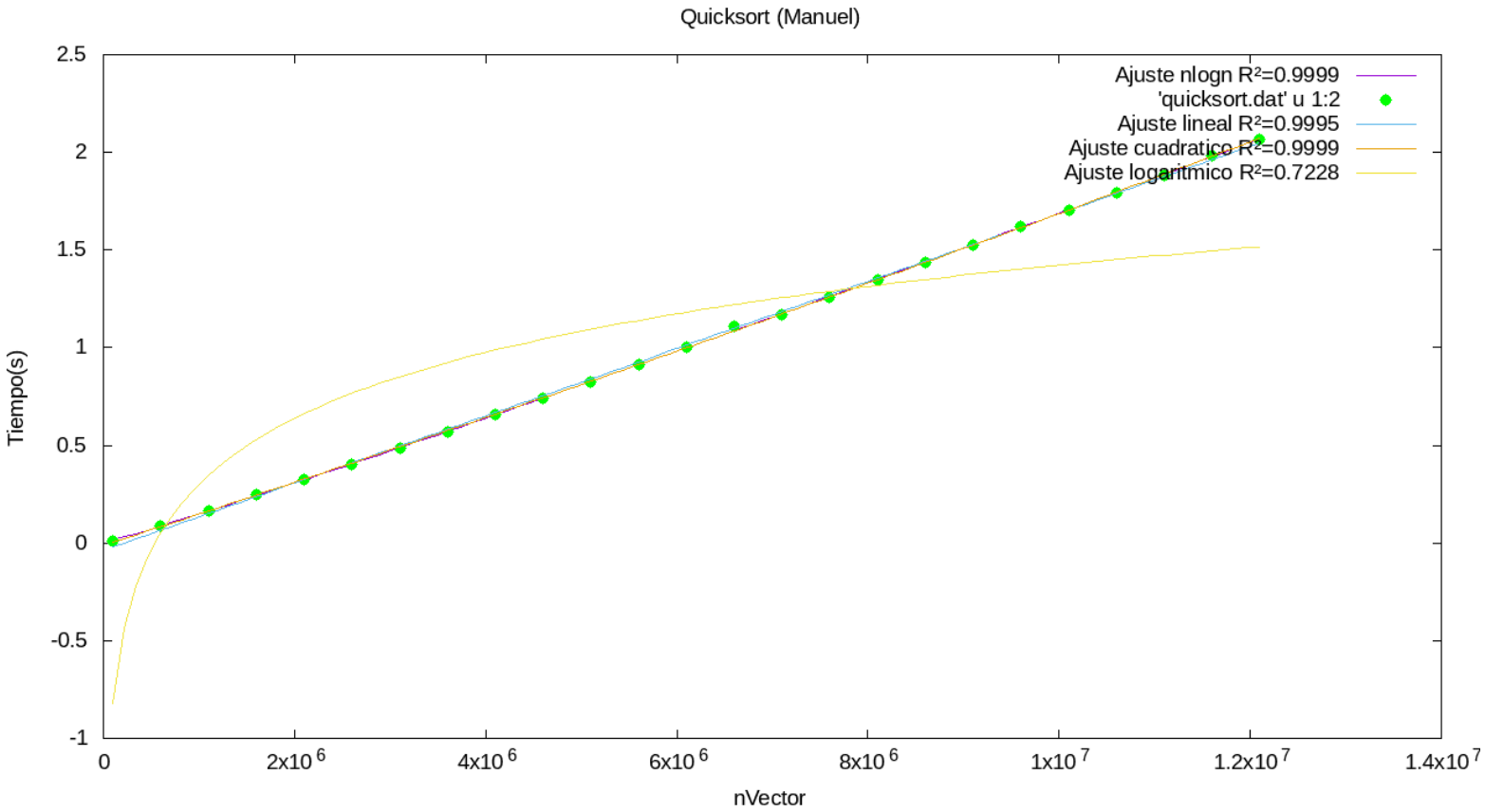
nVector	Quicksort Manuel	Quicksort Pablo	Quicksort Carlos	Heapsort Manuel	Heapsort Pablo	Heapsort Carlos
100000	0,012418	0,009424533333	0,01639626667	0,01664566667	0,014249	0,01751253333
600000	0,084653	0,06655053333	0,1119696	0,1216504667	0,111585	0,1690705333
1100000	0,161785	0,1226129333	0,214225	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

Quicksort Ajuste del Grupo



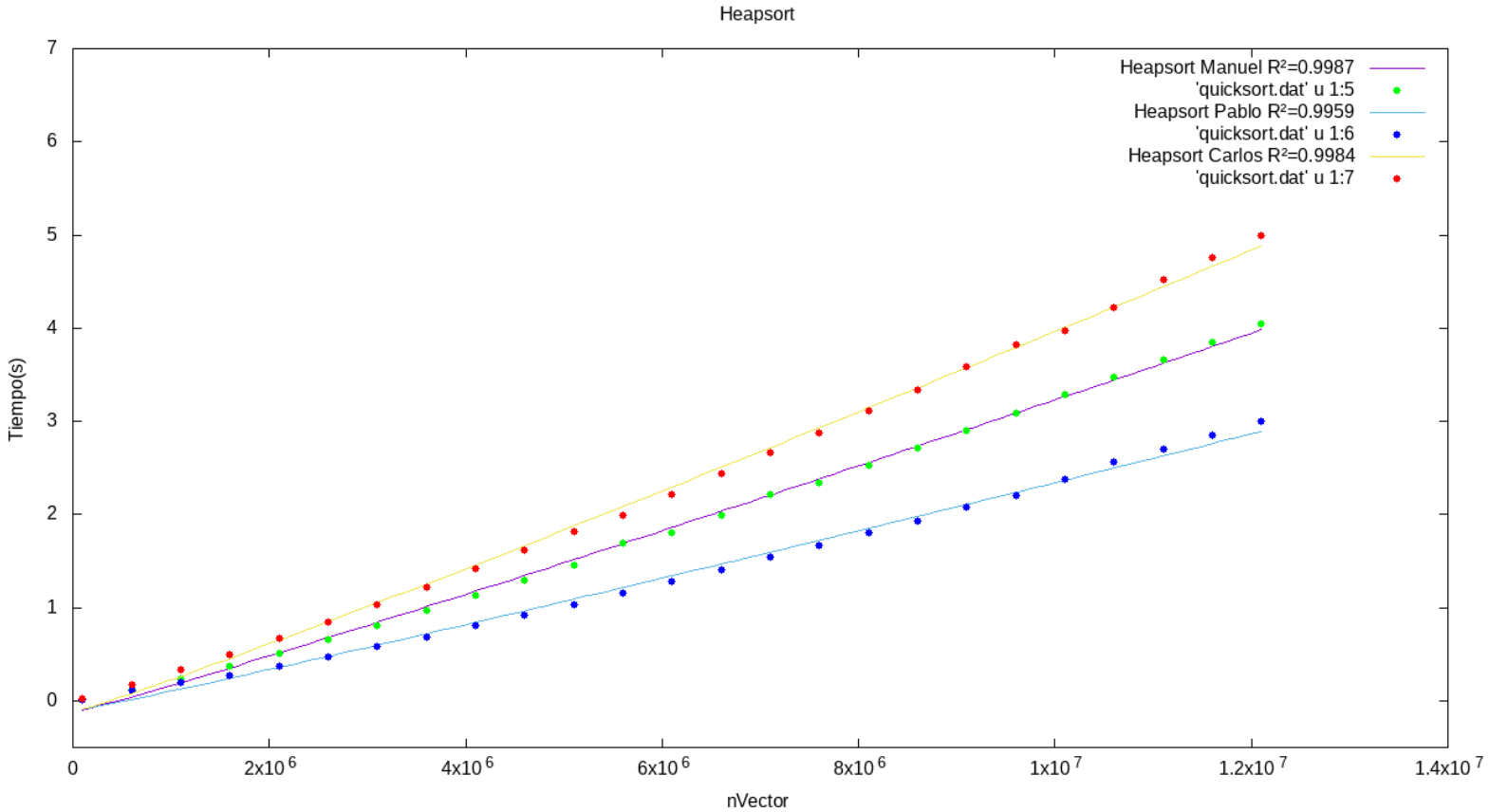
1. Gráficas con datos distintos, pero formas similares
2. $r \approx 1$

Quicksort Eficiencia Híbrida



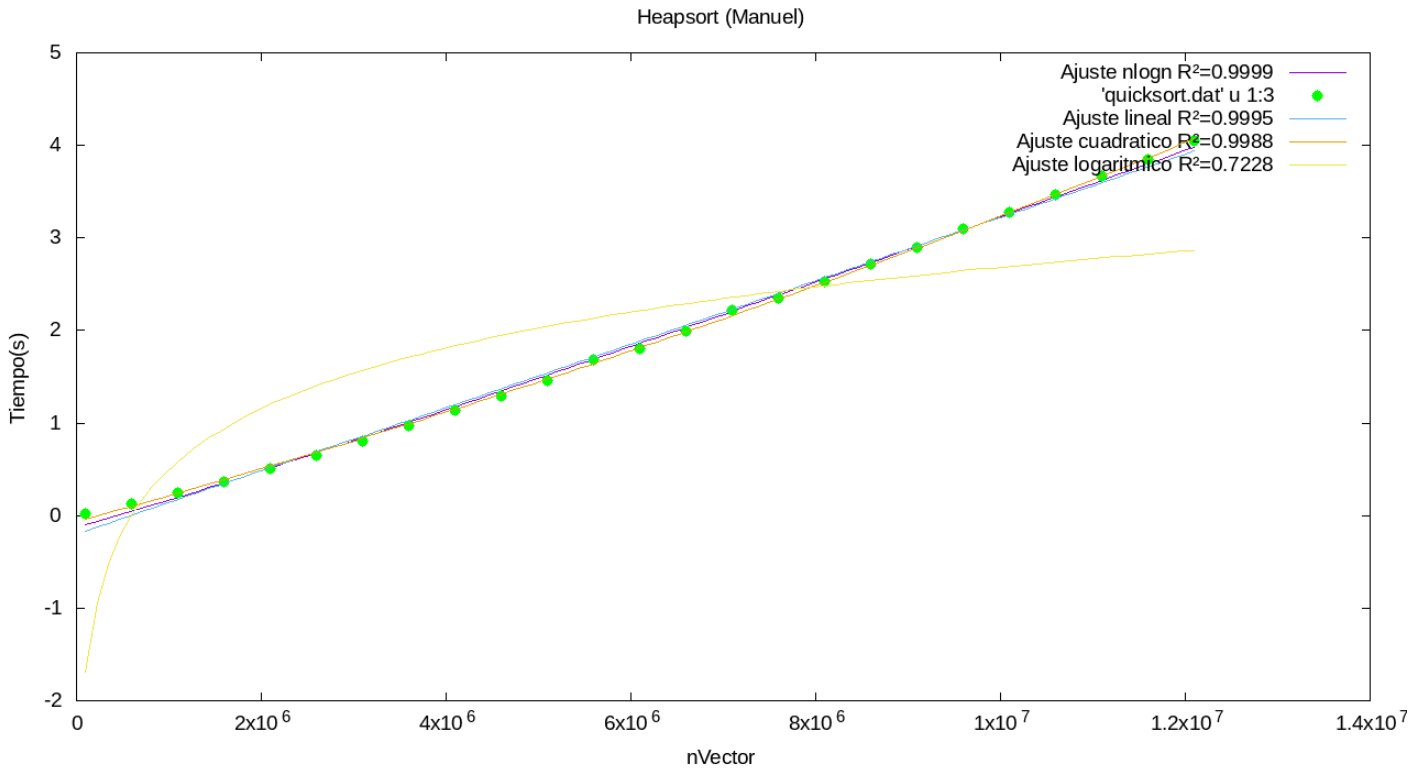
1. $T(n) = 1.0461 \cdot 10^{-8} \cdot n \log n + 0.00306$ para un ordenador
2. $R^2(n \log n) = 0.999948$, R^2 (lineal) $= 0.999901$, R^2 (exponencial) $= 0.6889$ y R^2 (logarítmica) $= 0.7225$

Heapsort Ajuste del Grupo



1. Las constantes ocultas cambian con el hardware, el tipo de función no lo hace
2. Coeficiente de correlación prácticamente 1

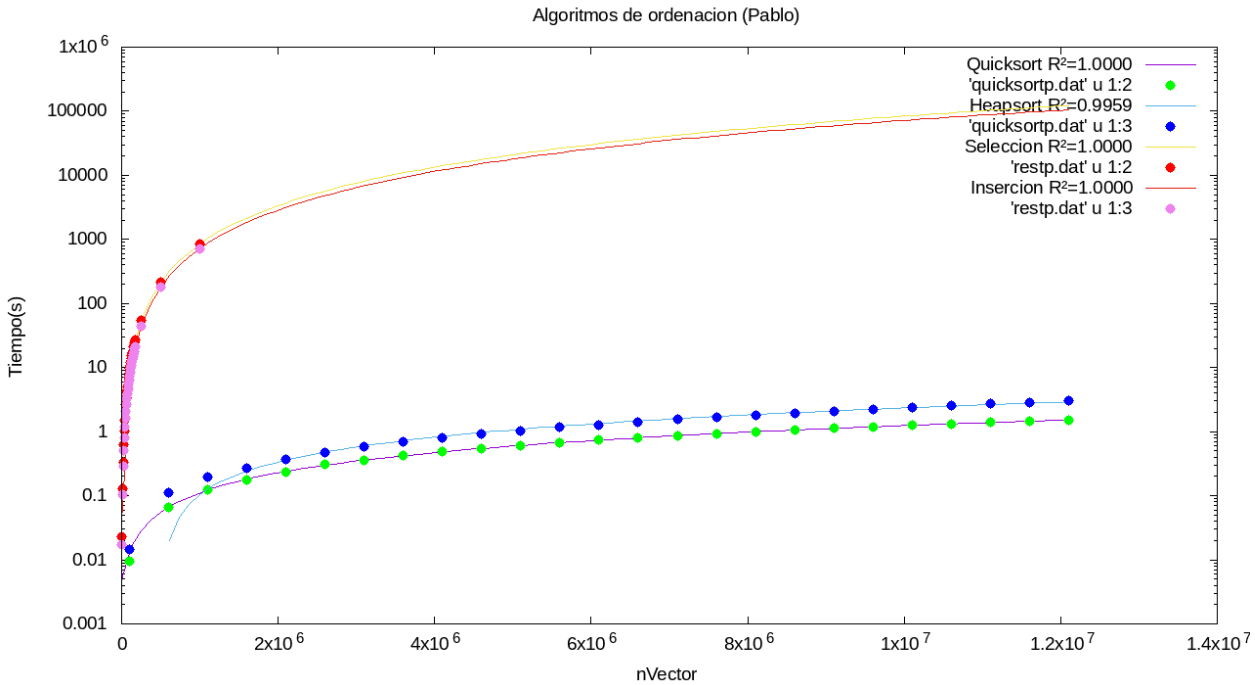
Heapsort Eficiencia Híbrida



1. $T(n) = 2.0794 \cdot 10^{-8} \cdot n \log n - 0.121342$ para un ordenador
2. $R^2(n \log n) = 0.9999$, R^2 (lineal) $= 0.9995$, R^2 (cuadrático) $= 0.9988$ y R^2 (logarítmica) $= 0.7228$
3. Mejor ajuste $\rightarrow$ $n \log n$

Comparación de Quicksort, Heapsort, Inserción y Selección (I)

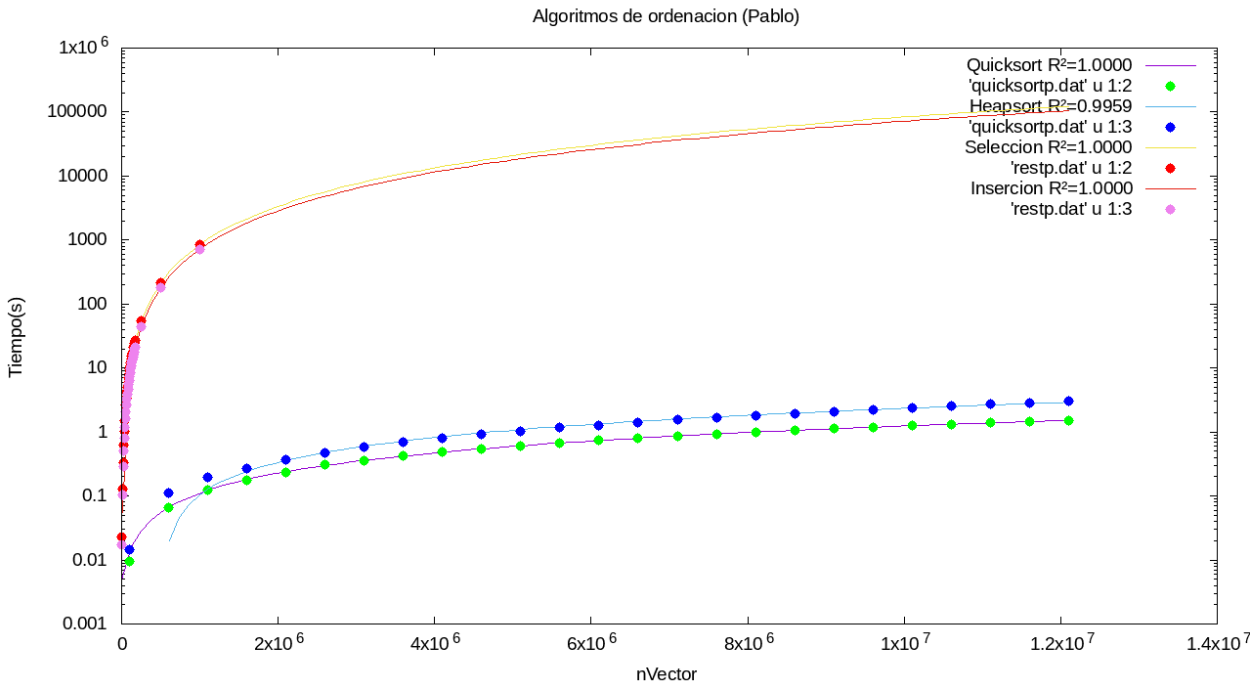
Eje Y en escala logarítmica!



1. Diferentes tiempos $\rightarrow$ Distintas constantes
2. Para todos, Quicksort y Heapsort son $n \log n$ y Inserción y Selección son n^2 en caso promedio

Comparación de Quicksort, Heapsort, Inserción y Selección (II)

Eje Y en escala logarítmica!



1. Quicksort el algoritmo más rápido a pesar de ser muy lento en peor caso ($O(n^2)$)
2. Para valores de n pequeños, los algoritmos se comportan de forma parecida

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  }
```

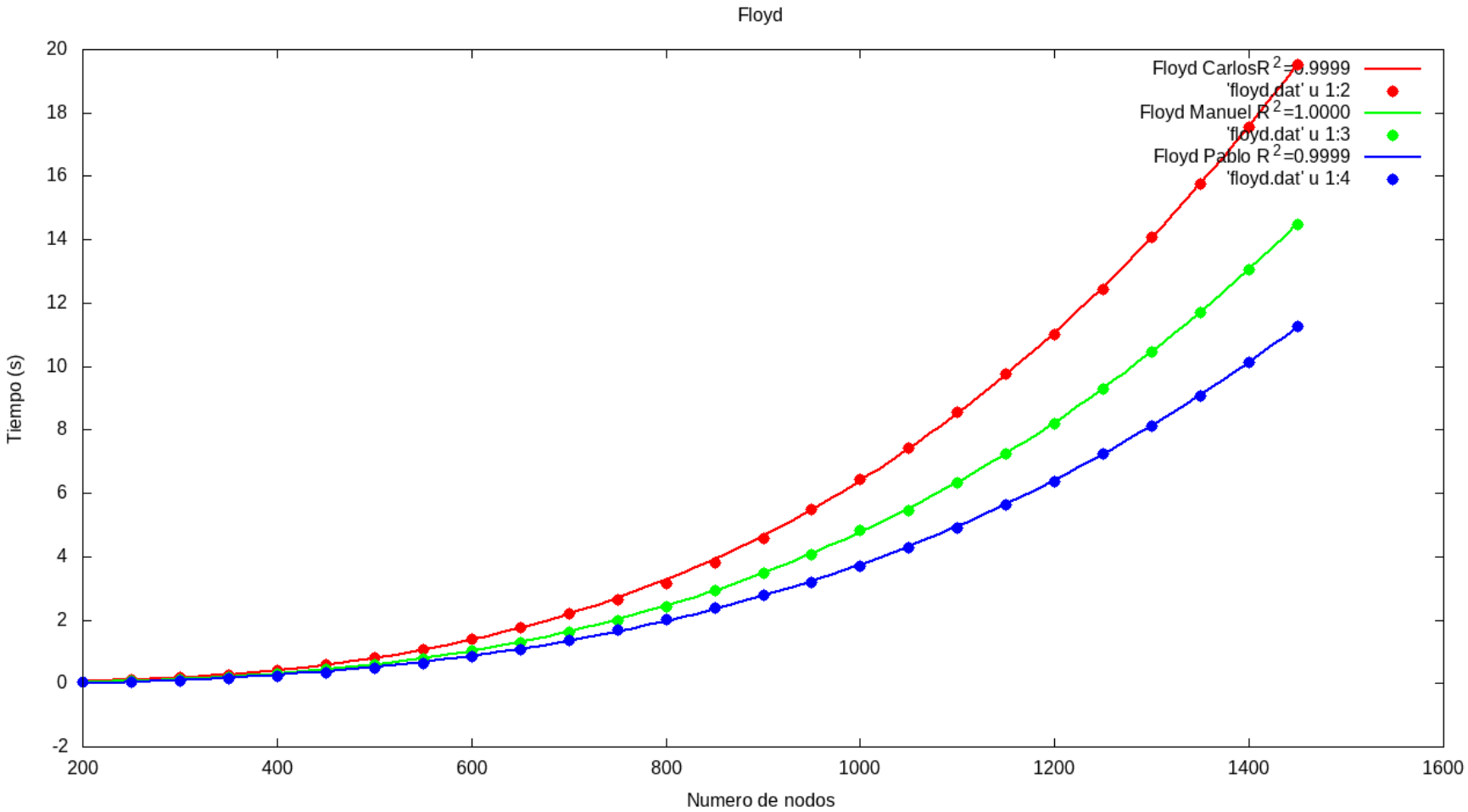
$$T(n) = a_0 \cdot n^3 + a_1 \cdot n^2 + a_2 \cdot n + a_3$$

$$\mathbf{O(n^3)}$$

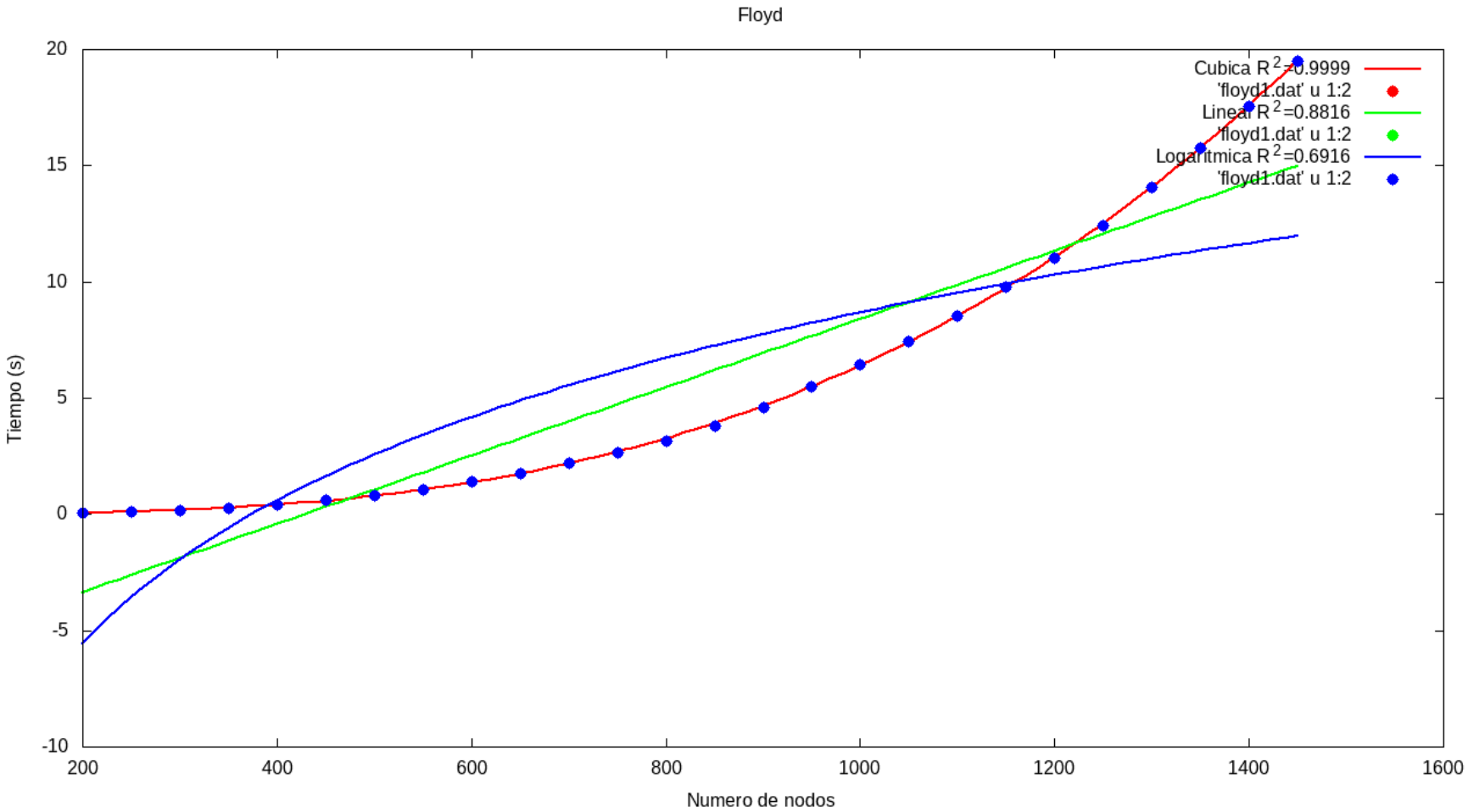
Eficiencia empírica e híbrida

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
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

Eficiencia empírica e híbrida



Comparación con distintos ajustes



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  }
```

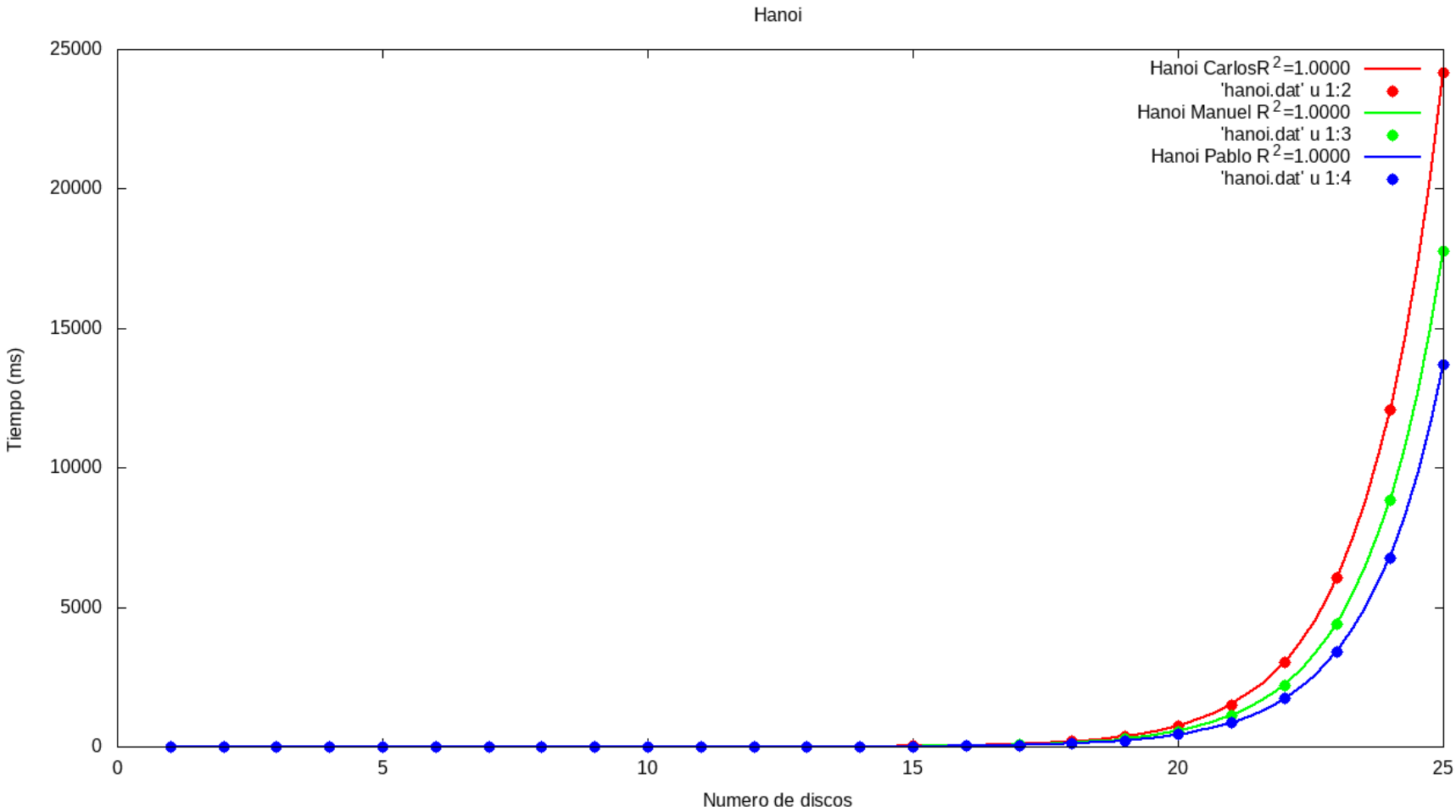
$$T(n) = a_0 \cdot 2^n + a_1$$

$$\mathbf{O(2^n)}$$

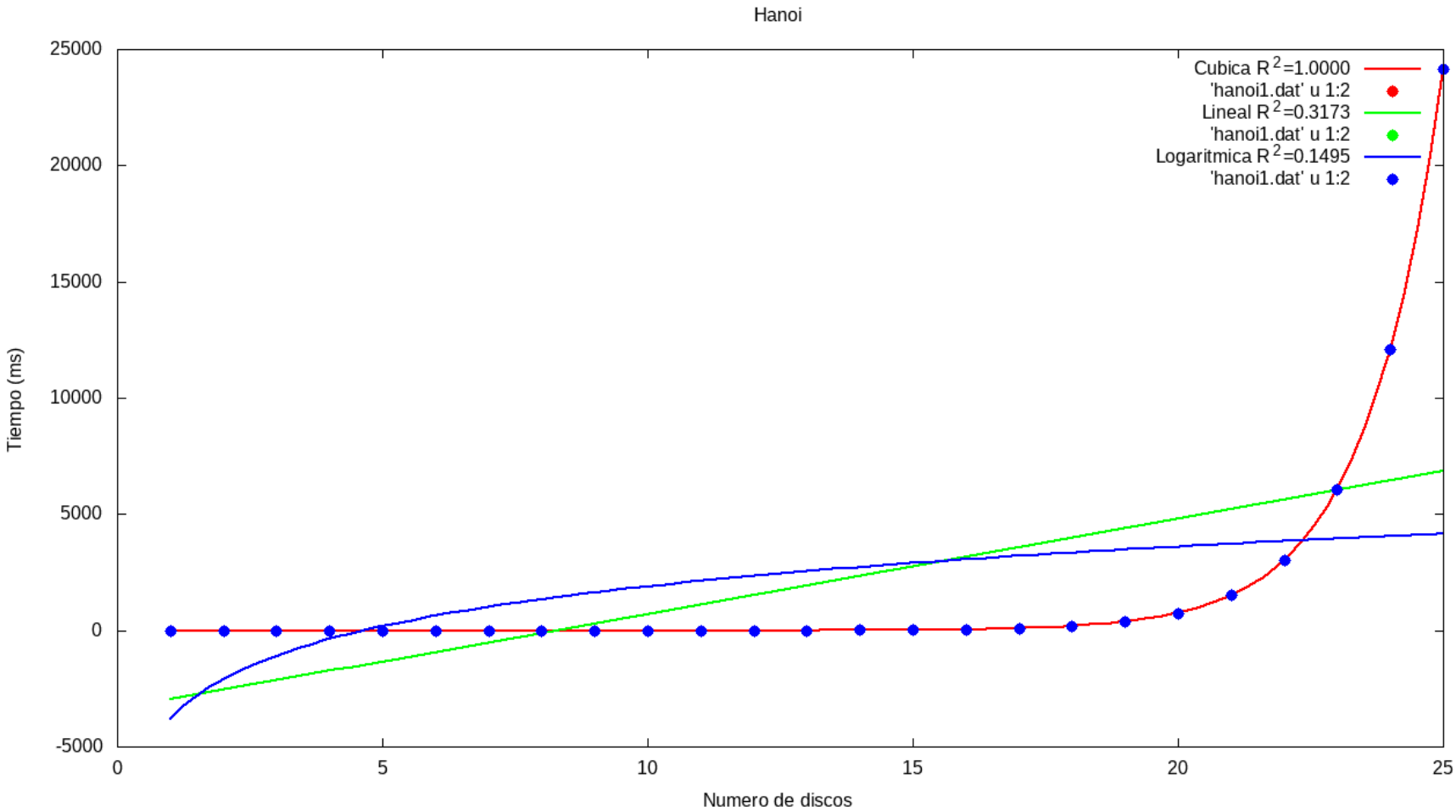
Eficiencia empírica e híbrida

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
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

Eficiencia empírica e híbrida

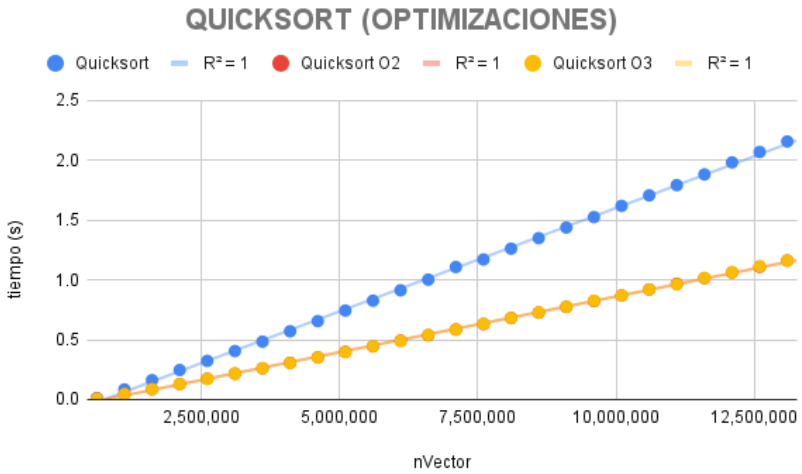


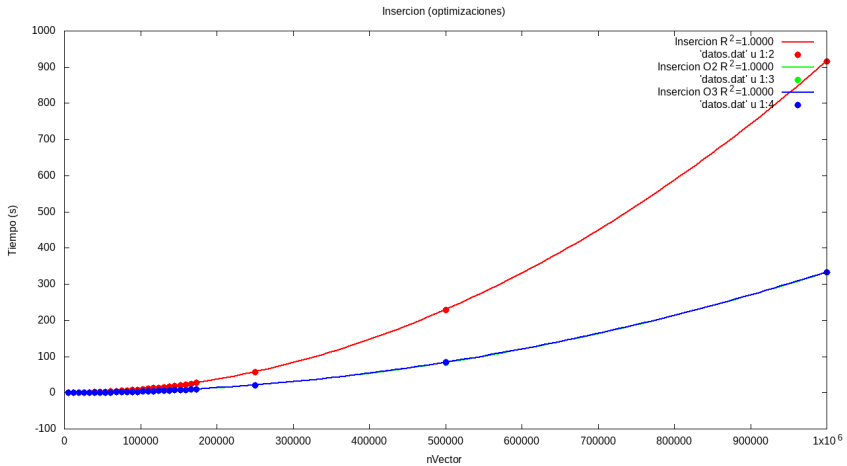
Comparación con distintos ajustes



Pruebas con optimizaciones (I)

Compilado sin optimización, con -O2 y con -O3

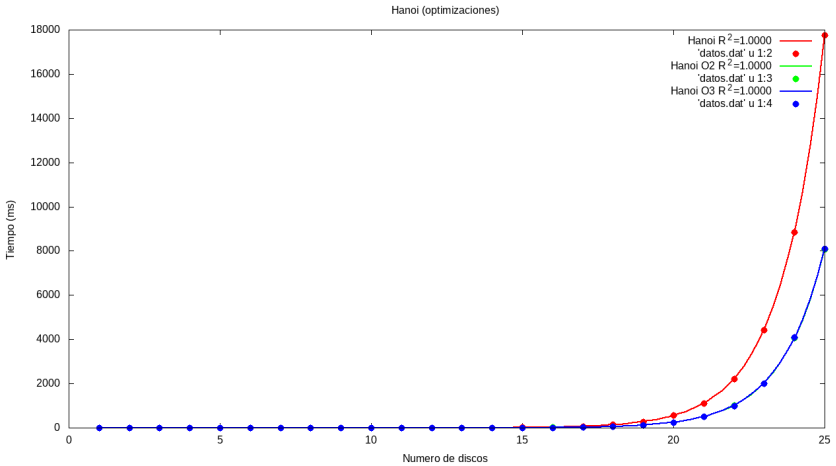


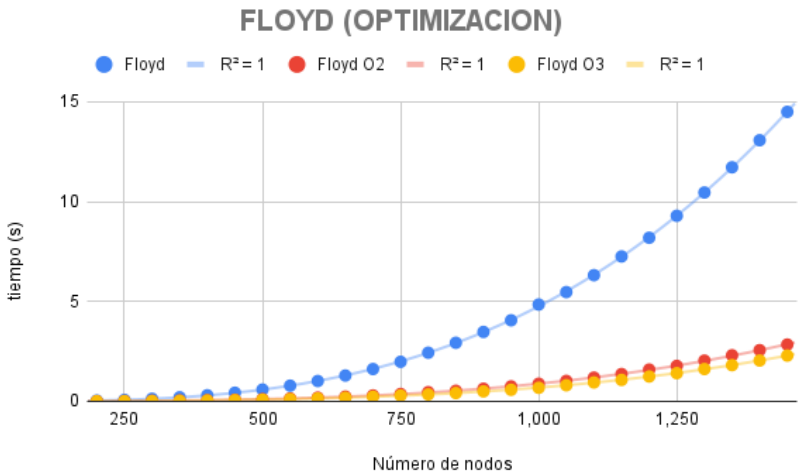


1. No hay mucha diferencia entre -O2 y -O3
2. Mejoran bastante los tiempos obtenidos
3. Se mantienen los tipos de funciones ajustadas en los otros apartados.

Pruebas con optimizaciones (II)

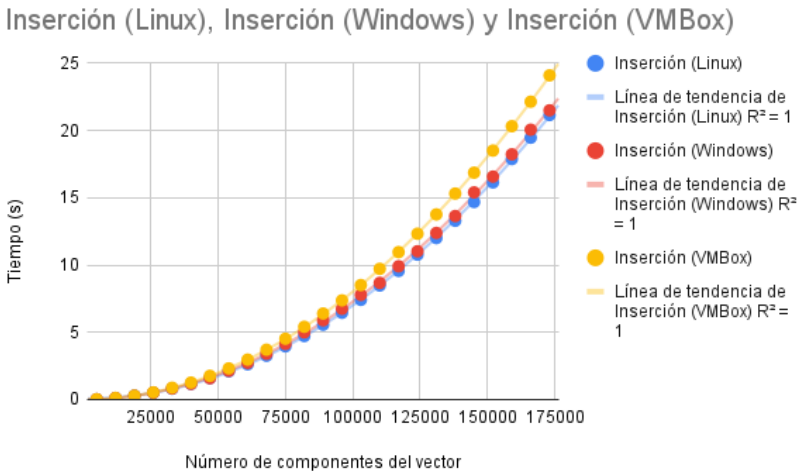
Compilado sin optimización, con -O2 y con -O3

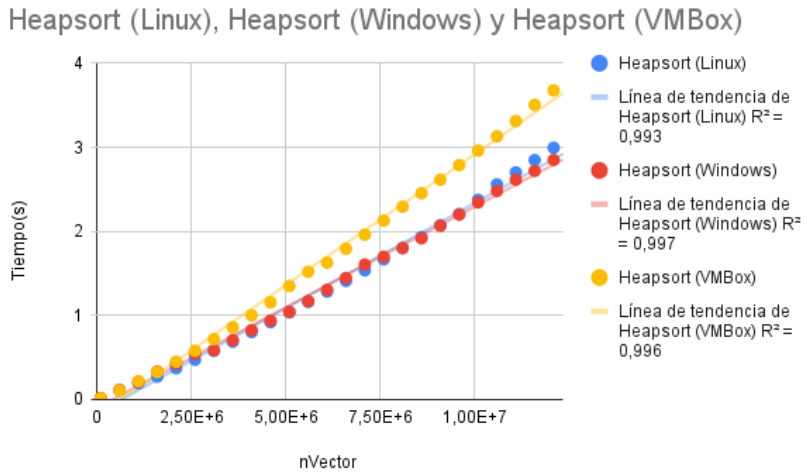




Pruebas con distintos software (I)

Pruebas con Linux, Windows y VMBox

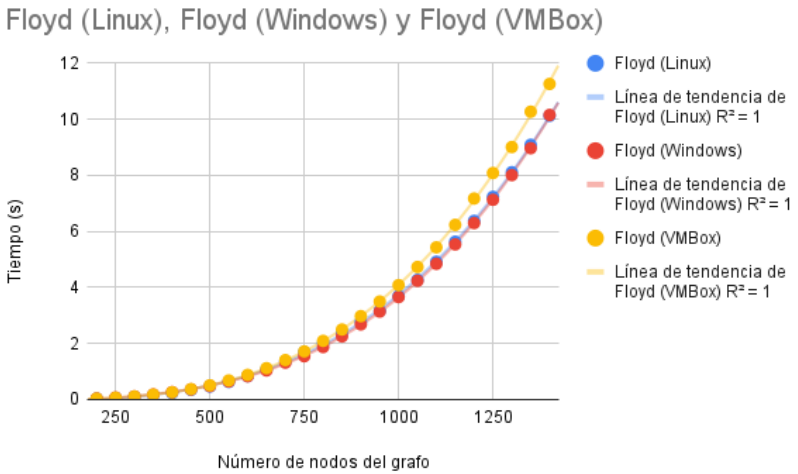


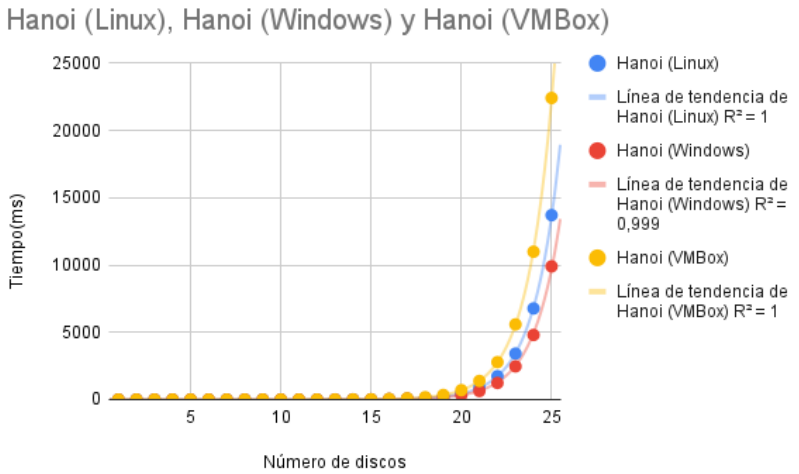


1. Generalmente se tiene Linux $>$ Windows $>$ VMBox
2. Siempre se tiene que la máquina virtual es ligeramente más lenta $\rightarrow$ menos CPU y RAM.
3. Se mantienen los tipos de funciones ajustadas en apartados anteriores para ambos algoritmos $\rightarrow$ Se mejoran las constantes, no la eficiencia del algoritmo!

Pruebas con distintos software (II)

Pruebas con Linux, Windows y VMBox





1. De nuevo se verifica que VMBox es más lenta
2. Sin embargo, se mantienen siempre los tipos de funciones ajustadas.
3. Cúbica para Floyd y Exponencial para Hanoi $\rightarrow r \approx 1$

Conclusión

1. Importancia análisis teórico y empírico de la eficiencia
2. Tiempos muy distintos dependiendo del hardware y software $\rightarrow$ mismos tipos para todos.
3. Parábola en algoritmos cuadráticos, cúbica en Floyd, $n \log n$ en Quicksort y Heapsort y una exponencial en Hanoi.
4. Podemos saber hasta donde puede llegar nuestro ordenador en un algoritmo $\rightarrow$ Enorme importancia en Floyd y Hanoi.
5. Peor y mejor caso de inserción y selección $\rightarrow$ Importante conocer cómo funcionan los algoritmos $\rightarrow$ No siempre existen enormes diferencias entre los casos mejor, promedio y peor!
6. Hemos podido aprender algoritmos que no conocíamos, como Floyd y Hanoi.
7. Las constantes ocultas pueden cambiar dependiendo del ordenador que tengamos $\rightarrow$ La eficiencia de los algoritmos se mantiene siempre, los ejecutemos donde los ejecutemos!