Análisis de eficiencia de algoritmos

ALGORÍTMICA

UNIVERSIDAD DE GRANADA

JOAQUÍN ARCILA PÉREZ
LAURA LÁZARO SORALUCE
CRISTÓBAL MERINO SÁEZ
ÁLVARO MOLINA ÁLVAREZ

ÍNDICE

INTRODUCCIÓN

Esta práctica consiste en la ejecución de varios algoritmos de distinto orden de eficiencia. En concreto:
- Algoritmo de inserción
- Algoritmo de selección
- Algoritmo de heapsort
- Algoritmo de quicksort
- Algoritmo de Floyd
- Algoritmo de Hanoi

Análisis teórico: Utilizando las técnicas de resolución de recurrencias vistas en clase, hemos analizado los códigos de los algoritmos para obtener las funciones de eficiencia.

Análisis empírico: Hemos modificado los códigos de los algoritmos y, utilizando la biblioteca ctime, medimos el tiempo que tarda en ejecutarse cada uno de ellos. Para evitar errores, hemos añadido un bucle que ejecuta el algoritmo 15 veces, para obtener una media del tiempo en lugar de un sólo valor. Hemos elegido distintos rangos de tamaños dependiendo de la eficiencia de cada algoritmo.
Por otro lado, hemos ejecutado cada algoritmo en ordenadores con distintas propiedades, sistemas operativos y optimizaciones.

Análisis híbrido: Hemos introducido en gnuplot las ecuaciones teóricas que nos dan el orden de eficiencia de los algoritmos, y ajustado dicha ecuación a los tiempos obtenidos para cada tamaño, para sacar las constantes ocultas.

Para terminar con el desarrollo, comparamos los componentes de los ordenadores utilizados en la práctica. Como se puede ver, los valores son bastante parecidos, por los que los resultados debrían ser bastante parecidos.

	Laura	Joaquín	Álvaro	Cristóbal
Arquitectura:	x86_64	x86_64	x86_64	x86_64
modo(s) de ope	32-bit, 64-bit	32-bit, 64-bit	32-bit, 64-bit	32-bit, 64-bit
Orden de los by	Little Endian	Little Endian	Little Endian	Little Endian
CPU(s):	8	12	8	12
Hilo(s) de proce	2	2	2	2
Núcleo(s) por «s	4	6	4	6
«Socket(s)»	1	1	1	1
Caché L1d:	128 KiB	192 KiB	192 KiB	192 KiB
Caché L1i:	128 KiB	192 KiB	128 KiB	192 KiB
Caché L2:	1 MiB	1,5 MiB	2 MiB	1,5 MiB
Caché L3:	8 MiB	12 MiB	8 MiB	12 MiB

DESARROLLO

1.Algoritmo de selección:

1.1 Eficiencia teórica

```
void seleccion(int T[], int num_elem)
{
  seleccion_lims(T, 0, num_elem);
}

static void seleccion_lims(int T[], int inicial, int final)
{
  int i, j, indice_menor;
  int menor, aux;
  for (i = inicial; i < final - 1; i++) {
    indice_menor = i;
    menor = T[i];
    for (j = i; j < final; j++)
      if (T[j] < menor) {
        indice_menor = j;
        menor = T[j];
      }
    aux = T[i];
    T[i] = T[indice_menor];
    T[indice_menor] = aux;
  };
}
```

El for externo podemos expresarlo como la suma desde i=inicial hasta final-2 del bucle interior. Tomando inicial=0, se nos queda suma desde i=0 hasta final-2 del bucle interior.

El for interno podemos expresarlo como la suma desde j=i hasta final-1 de a. Tomando final=n, tenemos la suma desde j=i hasta n de a.

El interior del bucle podemos agruparlo en una constante a, pues son de operaciones de asignación, de orden O(1).

Resolviendo los sumatorios anteriores, obtenemos la función polinómica $an^2/2 + an/2 - a$. Por lo tanto tenemos que $T(n) \in O(n^2)$.

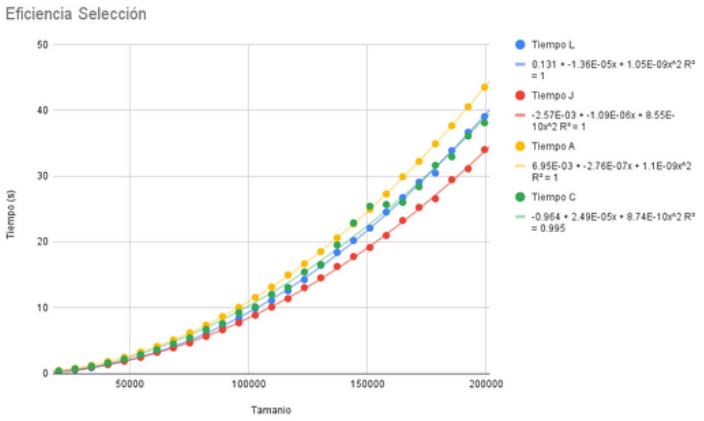
DESARROLLO

1.2 Eficiencia empírica

Este algoritmo lo hemos ejecutado para 27 tamaños en el rango de 20.000 a 200.000 y hemos obtenido tiempos desde 0'29 hasta 43'51 segundos.

Estos son los datos obtenidos para cada uno de los miembros del grupo:

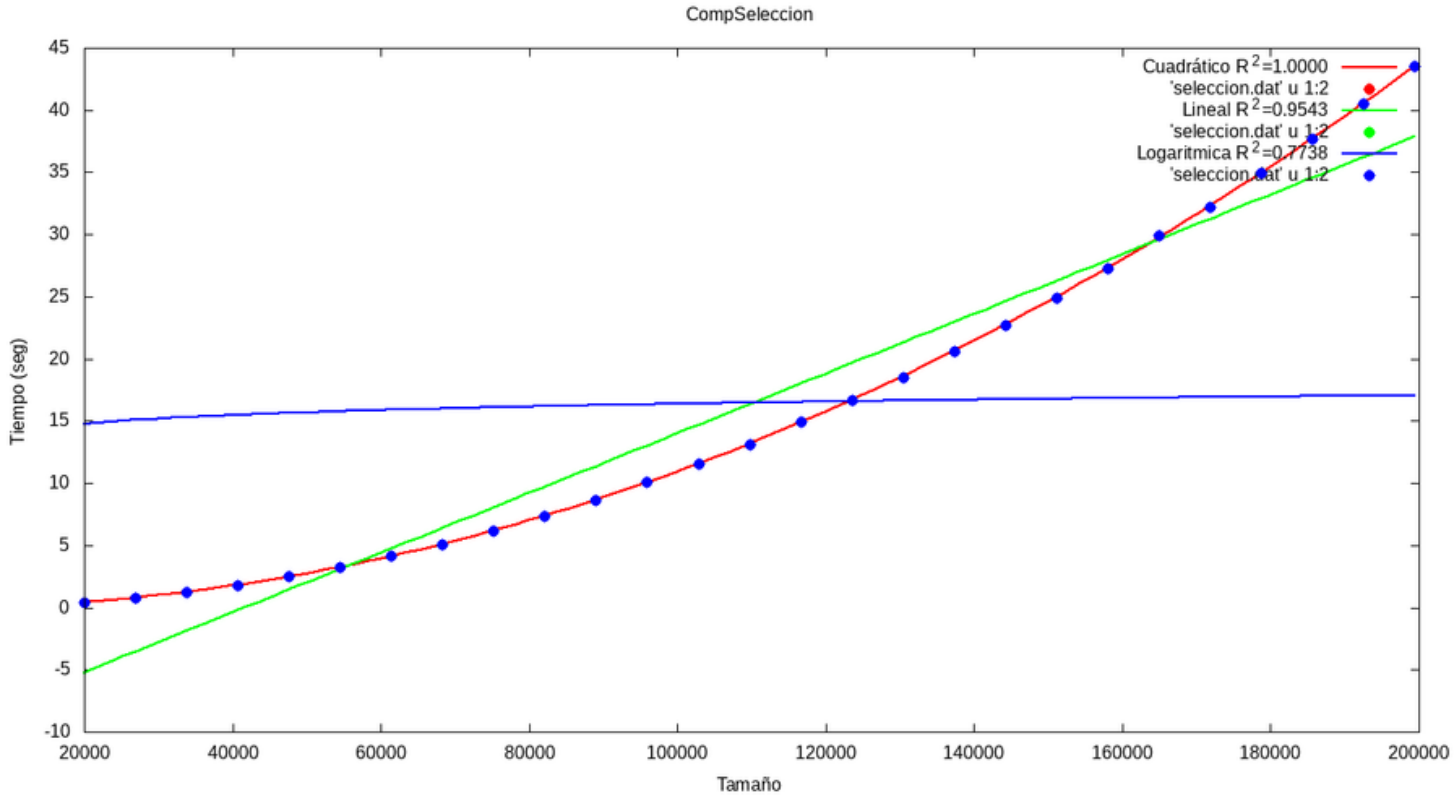
Tamanio	Tiempo L	Tiempo J	Tiempo A	Tiempo C
20000	0.290498	0.343629	0.439405	0.387053
26900	0.550093	0.598174	0.793031	0.693869
33800	0.850534	0.97914	1.24969	1.09476
40700	1.38592	1.36509	1.80915	1.58699
47600	1.94285	1.86493	2.47525	2.17016
54500	2.59248	2.44371	3.24633	2.85217
61400	3.23435	3.22685	4.11715	3.61086
68300	4.05006	3.896	5.09411	4.51218
75200	5.0023	4.65117	6.17491	5.41793
82100	6.06335	5.64024	7.35076	6.68123
89000	7.18832	6.63441	8.65639	7.58453
95900	8.41395	7.69666	10.0553	9.26192
102800	9.86142	8.86055	11.561	10.1168
109700	11.0719	10.0871	13.1492	12.0241
116600	12.5759	11.3772	14.9677	13.0673
123500	14.2642	13.0384	16.6705	15.4181
130400	16.4293	14.5355	18.5351	16.5698
137300	18.3911	16.2632	20.6087	19.5277
144200	20.2137	17.7741	22.7093	22.8955
151100	22.0993	19.1447	24.937	25.4309
158000	24.5306	20.9741	27.2732	25.6865
164900	26.7327	23.2599	29.8998	26.0058
171800	29.0702	25.2522	32.2329	28.3782
178700	30.4603	26.5561	34.9133	31.6436
185600	33.8792	29.4597	37.6527	32.9421
192500	36.6598	31.1243	40.553	36.0959
199400	39.0573	34.0273	43.507	38.1098



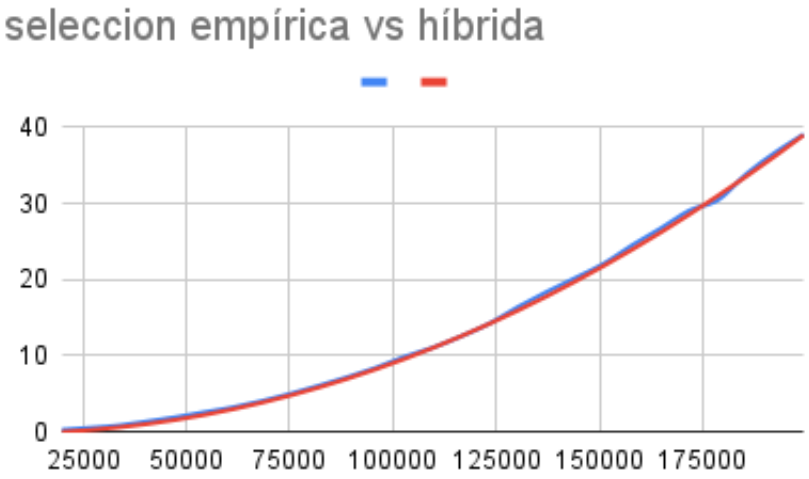
Se puede apreciar en la gráfica que las cuatro gráficas están muy cerca de ser iguales, corroborando la eficiencia de este algoritmo, mostrando además que tenemos unos ordenadores con componentes bastante parecidos.

1.3 Eficiencia híbrida

Para la eficiencia híbrida nos hemos ayudado de gnuplot. Primero hemos introducido los datos conseguidos del algoritmo y, ajustando con la ecuación cuadrática, conseguimos las variables ocultas. Hemos utilizado los valores de la gráfica A, y así conseguimos que las variables ocultas de la ecuación cuadrática (a +bx +cx^2) son a=6.95E-03, b=2.76E-07 y c=1.1E-09. A continuación se presenta la gráfica comparativa de este ajuste con otros, siendo estos el logarítmico y el lineal.



En la gráfica podemos ver que el ajuste que hemos escogido (el cuádratico) es el idóneo, y otros erróneos que hemos escogido para ilustrarlo, claramente no concuerda con las expectaticas, por lo que no son correctos.



Además, hemos comparado la eficiencia empírica con la híbrida y, como podemos observar, son bastante similares, demostrando que la eficiencia del programa se ajusta a la que debería ser.

DESARROLLO

2.Algoritmo de inserción:

2.1 Eficiencia teórica

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

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

El for podemos expresarlo como la suma desde i=inicial+1 hasta final-1 del bucle interior. Tomando inicial=0, se nos queda suma desde i=1 hasta final-1 del bucle interior.

El while podemos expresarlo como la suma desde j=1 hasta i de a, ya que este sería el peor caso.

El interior del bucle podemos agruparlo en una constante a, pues son de operaciones de asignación, de orden O(1).

Resolviendo los sumatorios anteriores, obtenemos la función polinómica $an^2/2 - an/2$. Por lo tanto tenemos que $T(n) \in O(n^2)$.

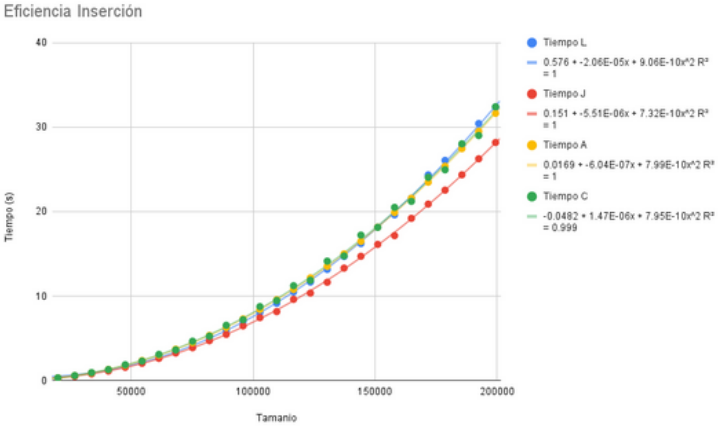
DESARROLLO

2.2 Eficiencia empírica

Este algoritmo lo hemos ejecutado para 27 tamaños en el rango de 20.000 a 200.000 y hemos obtenido tiempos desde 0'28 hasta 32'42 segundos.

Estos son los datos obtenidos para cada uno de los miembros del grupo:

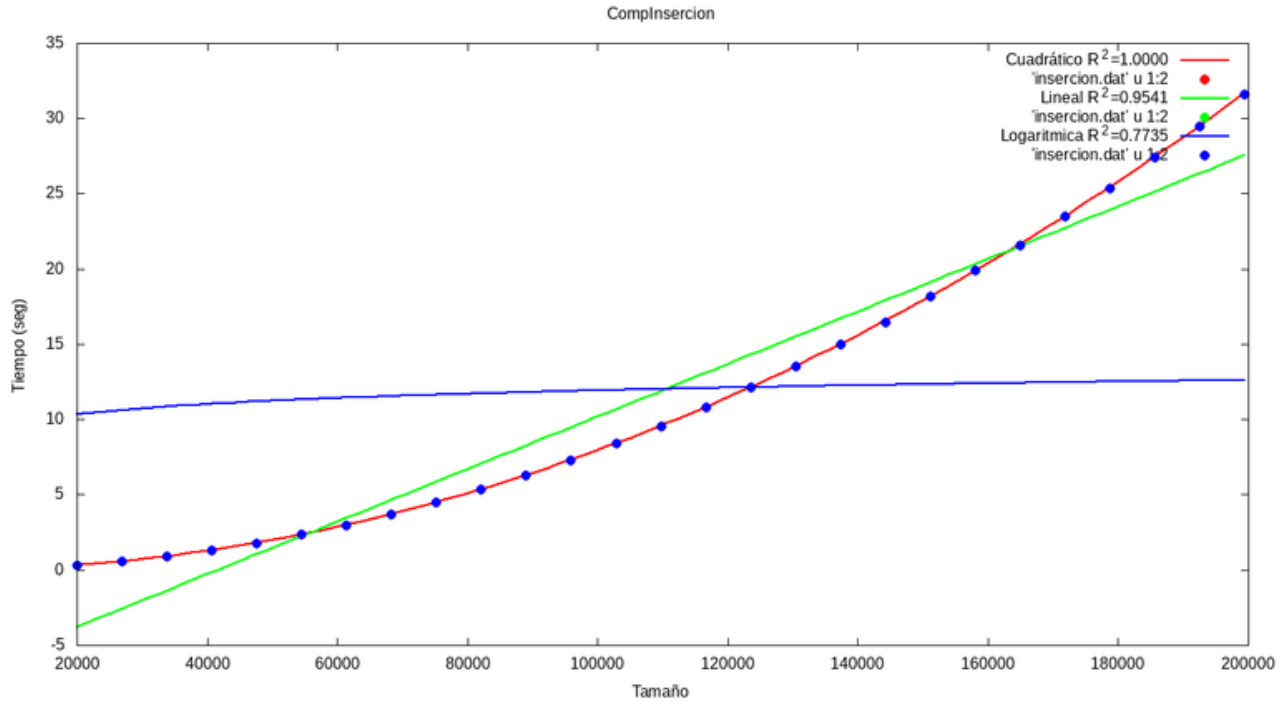
Tamanio	Tiempo L	Tiempo J	Tiempo A	Tiempo C
20000	0.301599	0.278072	0.318581	0.320694
26900	0.548447	0.481997	0.574314	0.584418
33800	0.860261	0.785912	0.909523	0.936840
40700	1.24357	1.1216	1.31544	1.299140
47600	1.71462	1.56355	1.79921	1.864620
54500	2.29191	2.03808	2.35629	2.300820
61400	2.91343	2.62213	3.00266	3.094580
68300	3.5285	3.26257	3.71053	3.624810
75200	4.33791	3.88334	4.48068	4.658170
82100	5.13381	4.71848	5.33961	5.248800
89000	5.99073	5.46298	6.2856	6.538950
95900	7.00642	6.47303	7.29162	7.201120
102800	8.05786	7.44979	8.41463	8.755050
109700	9.14771	8.16025	9.5637	9.473700
116600	10.4899	9.61018	10.7936	11.227100
123500	11.6591	10.355	12.1549	11.867600
130400	13.1571	11.6386	13.5178	14.136400
137300	14.6865	13.3131	14.996	14.757200
144200	16.217	14.7131	16.4757	17.218600
151100	18.1614	16.1187	18.163	18.135500
158000	19.618	17.1587	19.8945	20.502900
164900	21.5898	19.2066	21.5848	21.214000
171800	24.3502	20.8897	23.4863	24.058900
178700	26.0664	22.5417	25.3759	24.952200
185600	27.9036	24.3555	27.4577	28.035700
192500	30.432	26.2573	29.5027	29.006600
199400	32.3792	28.183	31.6392	32.419500



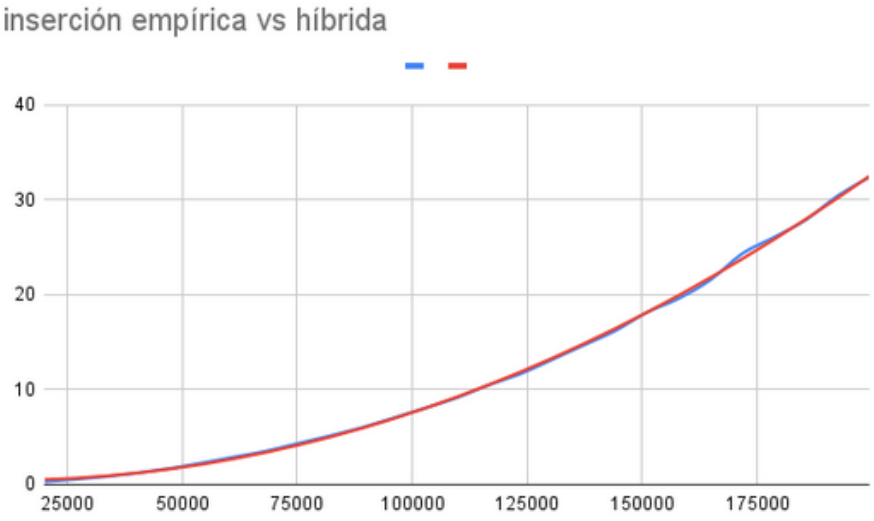
DESARROLLO

2.3 Eficiencia híbrida

Para la eficiencia híbrida nos hemos ayudado de gnuplot. Primero hemos introducido los datos conseguidos del algoritmo, y ajustando con la ecuación cuadrática conseguimos las variables ocultas. Hemos utilizado los valores de la gráfica A, y así conseguimos que las variables ocultas de la ecuación cuadrática (a +bx +cx^2) son a=0.169, b=-6.04E-07 y c=7.99E-02. A continuación se presenta la gráfica comparativa de este ajuste con otros, siendo estos el logarítmico y el lineal.



En la gráfica podemos ver que el ajuste que hemos escogido (el cuádratico) es el idóneo, y otros erróneos que hemos escogido para ilustrarlo, claramente no concuerda con las expectaticas, por lo que no son correctos.



Hemos comparado la eficiencia empírica con la híbrida y, como podemos observar, son bastante similares, demostrando que la eficiencia del programa se ajusta a la que debería ser.

3.Algoritmo de heapsort:

3.1 Eficiencia teórica

```
static void heapsort(int T[], int num_elem)
{
 int i;
 for (i = num_elem/2; i >= 0; i--)
  reajustar(T, num_elem, i);
 for (i = num_elem - 1; i >= 1; i--)
  {
   int aux = T[0];
   T[0] = T[i];
   T[i] = aux;
   reajustar(T, i, 0);
  }
}
```

Este for es la suma desde i=0 hasta num_elem/2 de reajustar. Tomando num_elem como n, tenemos la suma desde i=0 hasta n/2, que nos da (n+2)/2 * log(n).

Este for es la suma desde i=1 hasta n-1 de a*log(n). Esto nos da (n-1)*log(n).

Las tres instrucciones anteriores a reajustar podemos tomarlas como una constante a1.

```
static void reajustar(int T[], int num_elem, int k)
{
 int j;
 int v;
 v = T[k];
 bool esAPO = false;
 while ((k < num_elem/2) && !esAPO)
  {
   j = k + k + 1;
   if ((j < (num_elem - 1)) && (T[j] < T[j+1]))
j++;
   if (v >= T[j])
esAPO = true;
   T[k] = T[j];
   k = j;
  }
 T[k] = v;
}
```

Este bucle while es de orden O(log(n)), pues en cada vuelta, k pasa a ser 2k+1, por lo tanto recorre num_elem/2 elementos pero no de 1 en 1, sino de forma logarítmica.

Resolviendo los sumatorios anteriores, obtenemos la función 3n/2 * log(n). Por lo tanto tenemos que $T(n) \in O(n\log(n))$.

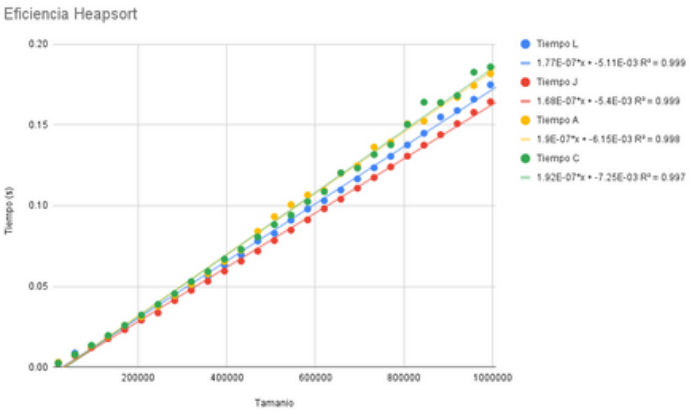
DESARROLLO

3.2 Eficiencia empírica

Este algoritmo lo hemos ejecutado para 27 tamaños en el rango de 20.000 a 1.000.000 y hemos obtenido tiempos desde 0'0024 hasta 0'20 segundos.

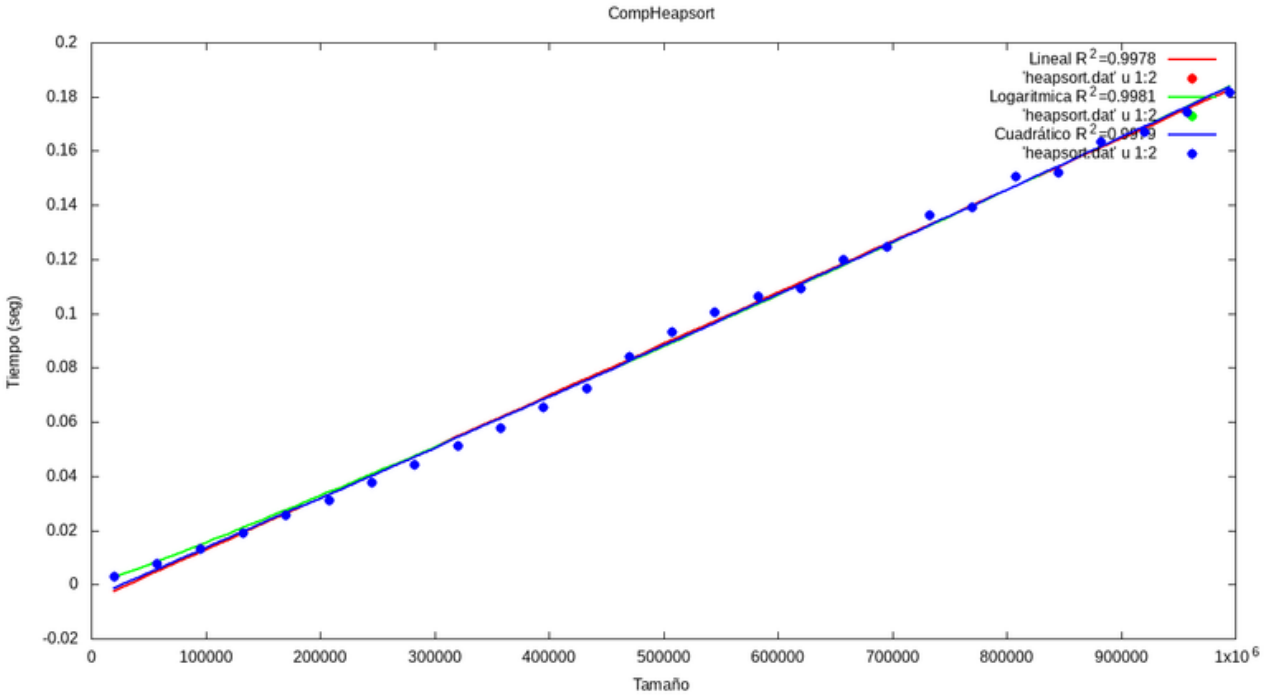
Estos son los datos obtenidos para cada uno de los miembros del grupo:

Tamanio	Tiempo L	Tiempo J	Tiempo A	Tiempo C
20000	0.00301493	0.0023532	0.0029864	0.00240867
57500	0.0087846	0.00721673	0.0077228	0.00774693
95000	0.0132101	0.0121781	0.0131655	0.0135409
132500	0.0188517	0.0177228	0.019074	0.0196246
170000	0.0246943	0.0232321	0.0257967	0.0259009
207500	0.0308806	0.0291134	0.0311007	0.0322879
245000	0.0375922	0.0336276	0.0376542	0.0389052
282500	0.043334	0.0412245	0.044358	0.0455075
320000	0.0498258	0.0475629	0.0511535	0.0529631
357500	0.0564683	0.0531944	0.0578393	0.0591946
395000	0.0632343	0.0594537	0.0654277	0.0669481
432500	0.0695085	0.0656189	0.0724561	0.0729267
470000	0.0780311	0.0718825	0.0840601	0.0806055
507500	0.0828549	0.0783795	0.0931705	0.0882319
545000	0.0909889	0.0848305	0.100574	0.0941709
582500	0.097854	0.0911869	0.106586	0.102462
620000	0.103022	0.0980629	0.10933	0.108623
657500	0.109643	0.103986	0.120026	0.120299
695000	0.116515	0.110765	0.124693	0.123205
732500	0.123405	0.117344	0.136234	0.13161
770000	0.130494	0.123953	0.139266	0.137629
807500	0.13752	0.130639	0.150585	0.150222
845000	0.144771	0.137424	0.152288	0.164017
882500	0.154829	0.143888	0.163297	0.16377
920000	0.158828	0.150732	0.166954	0.168127
957500	0.1658	0.157752	0.174289	0.18258
995000	0.174768	0.16427	0.181656	0.185831

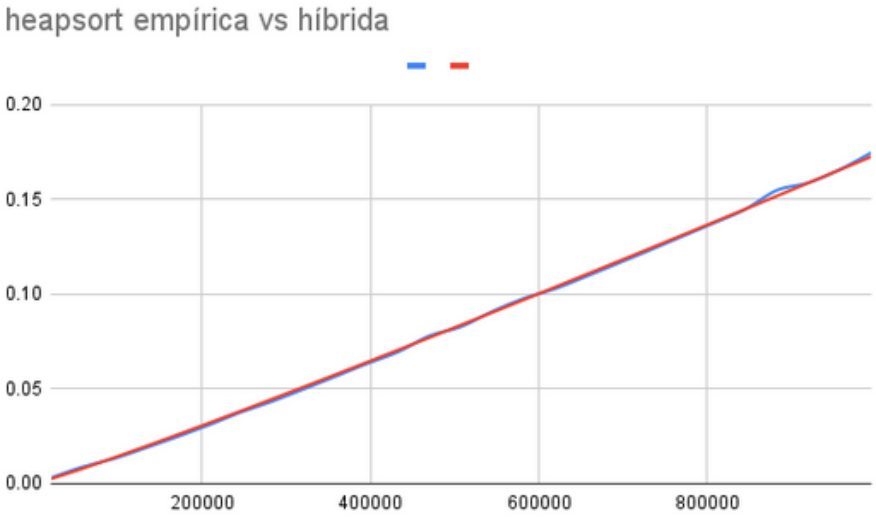


3.3 Eficiencia híbrida

Para la eficiencia híbrida nos hemos ayudado de gnuplot. Primero hemos introducido la fórmula teórica que nos da el orden de eficiencia de este algoritmo. luego hemos introducido los datos conseguidos del algoritmo, y ajustando con la ecuación n_logarítmica conseguimos las variables ocultas. Hemos utilizado los valores de la gráfica A, y así conseguimos que las variables ocultas de la ecuación n_logarítmica (axlog(x) +b) son a=1.33752e-08 y c=3.12312e-5. A continuación se presenta la gráfica comparativa de este ajuste con otros, siendo estos el cuádratico y el lineal.



En la gráfica podemos ver que el ajuste que hemos escogido (el n_logarítmico) es el idóneo, en este caso no se aprecia mucha diferencia con los erróneos ya que al ser tan pequeñs los valores, todas las gráficas acaban solapando.



Además, hemos comparado la eficiencia empírica con la híbrida y, como podemos observar, son bastante similares, demostrando que la eficiencia del programa se ajusta a la que debería ser.

4.Algoritmo de quicksort:

4.1 Eficiencia teórica

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

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

En el peor caso, el vector está ordenador de mayor a menor, por tanto este bucle va desde k=inicial hasta k=final-1. Tomando inicial como 0 y final como n, tenemos que es de orden O(n).

En dicho peor caso, este bucle va desde l=final hasta l=inicial, por lo que tenemos que es de orden O(n).

En el peor caso, no se entraría en este bucle, pues l quedaría en la posición inicial y k en la final-1.

Estas operaciones son de orden O(1), por lo que el método en sí, sería de orden O(n) en el peor caso.

DESARROLLO

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

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

En este else, tenemos un algoritmo de orden O(n) y dos llamadas recurrentes al propio método, que ordenan n/2 elementos.

O(n)

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

Este método es el mismo que se utiliza en el algoritmo de inserción, que como ya hemos visto, es de orden $O(n^2)$.

En el caso en que n>=UMBRAL_QS, utilizamos la fórmula de recurrencia, obteniendo: T(n) = n+2(T(n/2)).

Haciendo el cambio de variable: $n= 2^m$, tenemos $T(2^m)=2^m + 2T(2^{m-1})$, si m>=log2(UMBRAL_QS).

Operando llegamos a tener $T_m - 2T(_m-1) = 2^m$, donde la constante b=2 y p(n)=1. Sacamos la ecuación característica de nuestra recurrencia no homogénea, que es: (x-2)(x-2)=0, por lo que el 2 es raíz doble.

Tenemos $T_m = c1*2^m + c2*m*2^m$, y deshaciendo el cambio de variable: T(n)=c1*n+c2*n*log2(n).

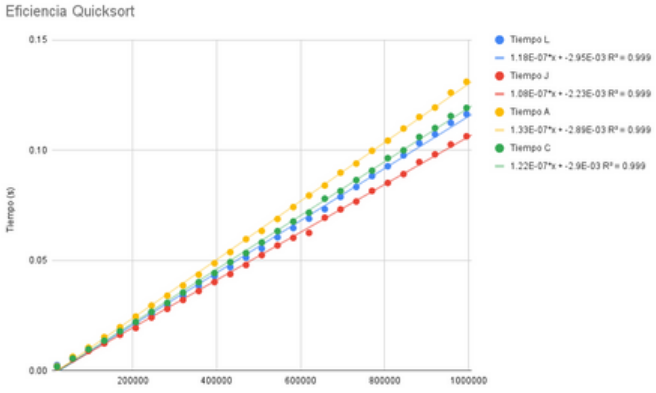
DESARROLLO

4.2 Eficiencia empírica

Este algoritmo lo hemos ejecutado para 27 tamaños en el rango de 20.000 a 1.000.000 y hemos obtenido tiempos desde 0'0018 hasta 0'13 segundos.

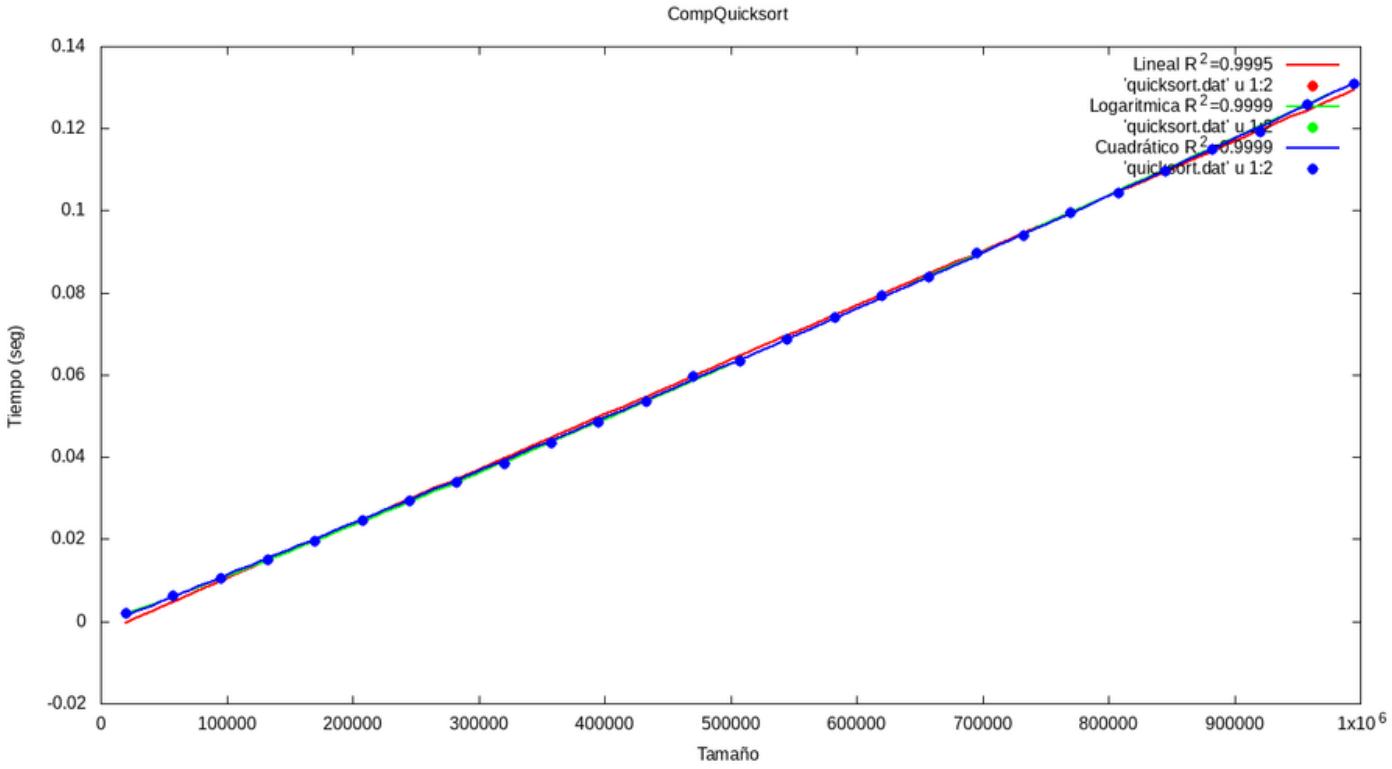
Estos son los datos obtenidos para cada uno de los miembros del grupo:

Tamanio	Tiempo L	Tiempo J	Tiempo A	Tiempo C
20000	0.0025096	0.0017786	0.002042	0.00177667
57500	0.00576307	0.00519227	0.00627387	0.00552653
95000	0.00942633	0.00893793	0.0104625	0.00966033
132500	0.0131791	0.0123411	0.0152	0.0135806
170000	0.0173298	0.0162187	0.019706	0.0178679
207500	0.0216701	0.0193439	0.0245244	0.0220659
245000	0.0258111	0.0240567	0.0294941	0.0266742
282500	0.0299367	0.0279946	0.0339822	0.0307427
320000	0.034001	0.0320465	0.0385838	0.0353873
357500	0.0385256	0.0360812	0.0435092	0.0401142
395000	0.0428881	0.0401525	0.0486824	0.0442753
432500	0.0469161	0.043763	0.0537028	0.049206
470000	0.0512508	0.047897	0.0596074	0.0533071
507500	0.0553668	0.0523335	0.0633293	0.0581102
545000	0.0604378	0.0567571	0.0687465	0.0632529
582500	0.0646449	0.0601965	0.0741323	0.0676179
620000	0.0688848	0.0624616	0.0794037	0.0716318
657500	0.0732006	0.0693719	0.0839329	0.077982
695000	0.0787838	0.0731093	0.0897564	0.0814543
732500	0.0832715	0.0766734	0.0939017	0.0863909
770000	0.0882021	0.0815045	0.0996819	0.0906719
807500	0.0926757	0.0850987	0.104273	0.0963509
845000	0.0975901	0.0890966	0.109765	0.0998927
882500	0.103136	0.0945858	0.115012	0.105933
920000	0.107145	0.0980822	0.119299	0.10998
957500	0.112381	0.102547	0.126046	0.115443
995000	0.116295	0.106273	0.130986	0.119146

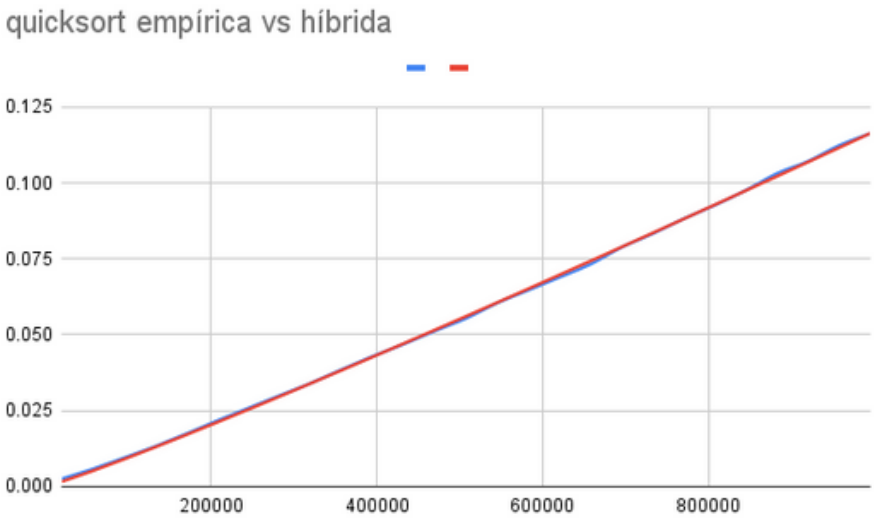


4.3 Eficiencia híbrida

Para la eficiencia híbrida nos hemos ayudado de gnuplot. Primero hemos introducido la fórmula teórica que nos da el orden de eficiencia de este algoritmo. luego hemos introducido los datos conseguidos del algoritmo, y ajustando con la ecuación n_logarítmica conseguimos las variables ocultas. Hemos utilizado los valores de la gráfica A, y así conseguimos que las variables ocultas de la ecuación n_logarítmica (axlog(x) +b) son a=9.53554e-09 y c=4.12312e-06. A continuación se presenta la gráfica comparativa de este ajuste con otros, siendo estos el cuádratico y el lineal.



En la gráfica podemos ver que el ajuste que hemos escogido (el n_logarítmico) es el idóneo, en este caso no se aprecia mucha diferencia con los erróneos ya que al ser tan pequeñs los valores, todas las gráficas acaban solapando.



Además, hemos comparado la eficiencia empírica con la híbrida y, como podemos observar, son bastante similares, demostrando que la eficiencia del programa se ajusta a la que debería ser.

DESARROLLO

5.Algoritmo de Floyd:

5.1 Eficiencia teórica

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

Tenemos 3 bucles anidados, los cuales se pueden expresar como la suma desde i=0 hasta dim-1. Luego cada uno de ellos es de orden O(n).

El interior del bucle podemos agruparlo en una constante a, pues son de operaciones de asignación, de orden O(1).

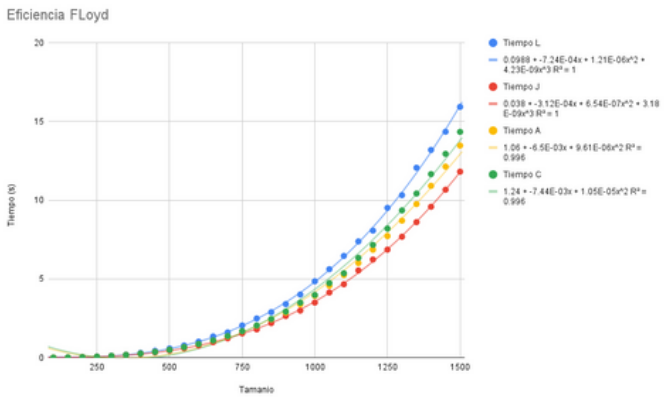
Resolviendo los sumatorios anteriores, obtenemos la función polinómica an^3. Por lo tanto tenemos que $T(n) \in O(n^3)$.

5.2 Eficiencia empírica

Este algoritmo lo hemos ejecutado para 29 tamaños en el rango de 100 a 1.500 y hemos obtenido tiempos desde 0'004 hasta 15'93 segundos.

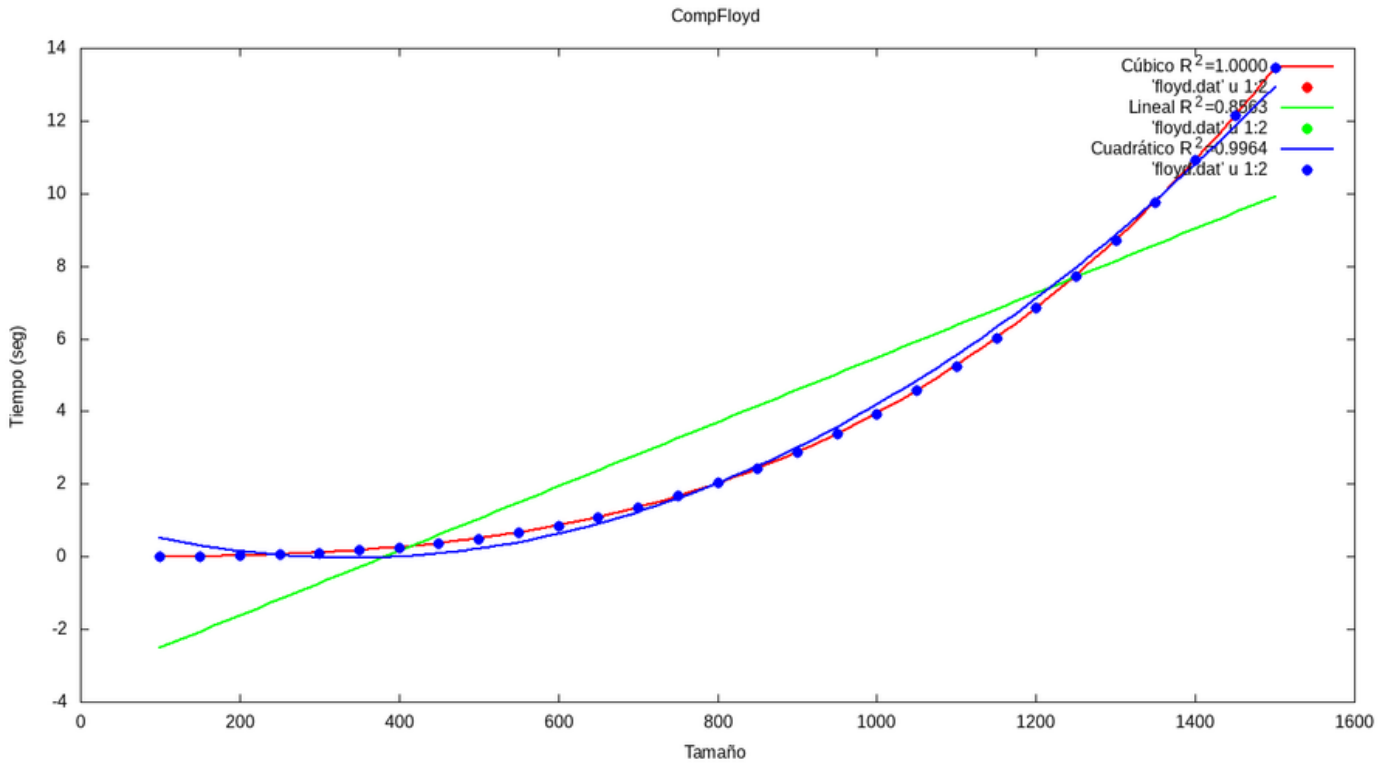
Estos son los datos obtenidos para cada uno de los miembros del grupo:

Tamanio	Tiempo L	Tiempo J	Tiempo A	Tiempo C
100	0.00591527	0.00412613	0.0042422	4.95E-03
150	0.0159097	0.0131253	0.0135397	1.35E-02
200	0.0361424	0.0282329	0.0316015	3.25E-02
250	0.0707361	0.0549193	0.0631014	6.36E-02
300	0.122718	0.0944375	0.109288	1.08E-01
350	0.193656	0.155449	0.171967	1.71E-01
400	0.304055	0.228613	0.253529	0.255101
450	0.434619	0.328178	0.364059	0.360566
500	0.584113	0.448897	0.492907	0.492583
550	0.773383	0.597088	0.661187	0.662751
600	1.01452	0.77342	0.858421	0.86317
650	1.34711	0.98263	1.0898	1.09551
700	1.60119	1.22735	1.34993	1.37133
750	2.04279	1.54127	1.67723	1.68606
800	2.48851	1.79436	2.02666	2.04314
850	2.88539	2.19434	2.42931	2.45308
900	3.40224	2.64458	2.8622	2.92508
950	4.01064	2.98971	3.38604	3.49565
1000	4.85129	3.49904	3.92764	3.98226
1050	5.62023	4.13973	4.57013	4.73894
1100	6.46328	4.66231	5.25261	5.36227
1150	7.38432	5.53939	6.01307	6.33794
1200	8.07171	6.23215	6.85778	7.16257
1250	9.51634	6.86325	7.72299	8.19994
1300	10.3231	7.68149	8.70321	9.35491
1350	12.0649	8.60936	9.75737	10.427
1400	13.1955	9.58565	10.9137	11.6603
1450	14.3529	10.6687	12.1364	12.9463
1500	15.9312	11.8135	13.4753	14.344

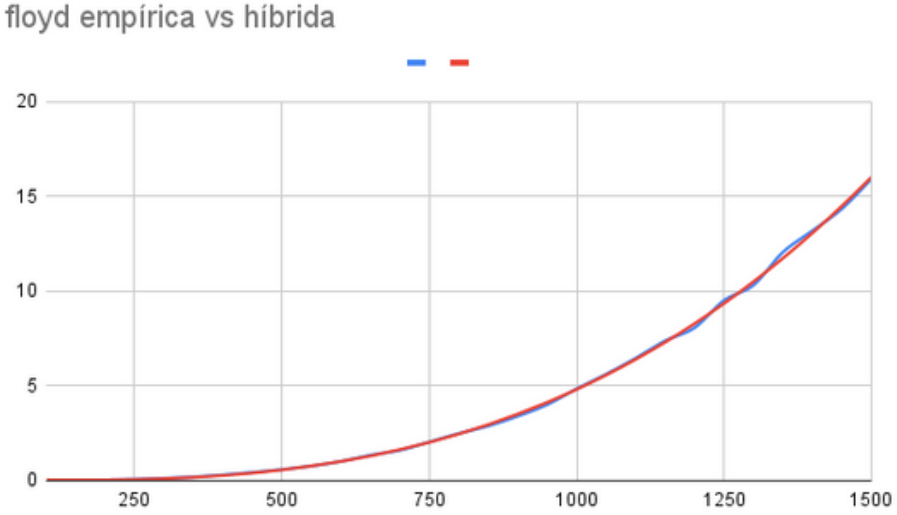


5.3 Eficiencia híbrida

Para la eficiencia híbrida nos hemos ayudado de gnuplot. Primero hemos introducido los datos conseguidos del algoritmo, y ajustando con la ecuación cuadrática conseguimos las variables ocultas. Hemos utilizado los valores de la gráfica A, y así conseguimos que las variables ocultas de la ecuación cuadrática (a +bx +cx^2) son a=1.06, b=-6.5E-03x y c=9.61E-06. A continuación se presenta la gráfica comparativa de este ajuste con otros, siendo estos el logarítmico y el lineal.



En la gráfica podemos ver que el ajuste que hemos escogido (el cuádratico) es el idóneo, y otros erróneos que hemos escogido para ilustrarlo, claramente no concuerda con las expectaticas, por lo que no son correctos.



Además hemos comparado la eficiencia empírica con la híbrida y, como podemos observar, son bastante similares, demostrando que la eficiencia del programa se ajusta a la que debería ser.

6.Algoritmo de Hanoi:

6.1 Eficiencia teórica

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

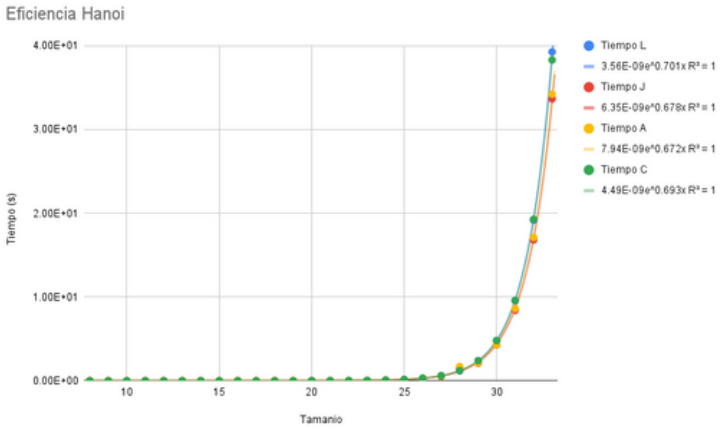
Para este método, utilizamos la fórmula para recurrencias. Vemos que se llama 2 veces a un método de orden n. Tenemos T(n)-2T(n)=0. Obtenemos la ecuación característica: $x^2-2x=0$, con lo que el 2 y el 0 son raíces, y se nos queda: $c1*2^n$. Por lo tanto tenemos que $T(n) \in O(2^n)$.

6.2 Eficiencia empírica

Este algoritmo lo hemos ejecutado para 26 tamaños en el rango de 8 a 33 y hemos obtenido tiempos desde 0'0000013 hasta 39'27 segundos.

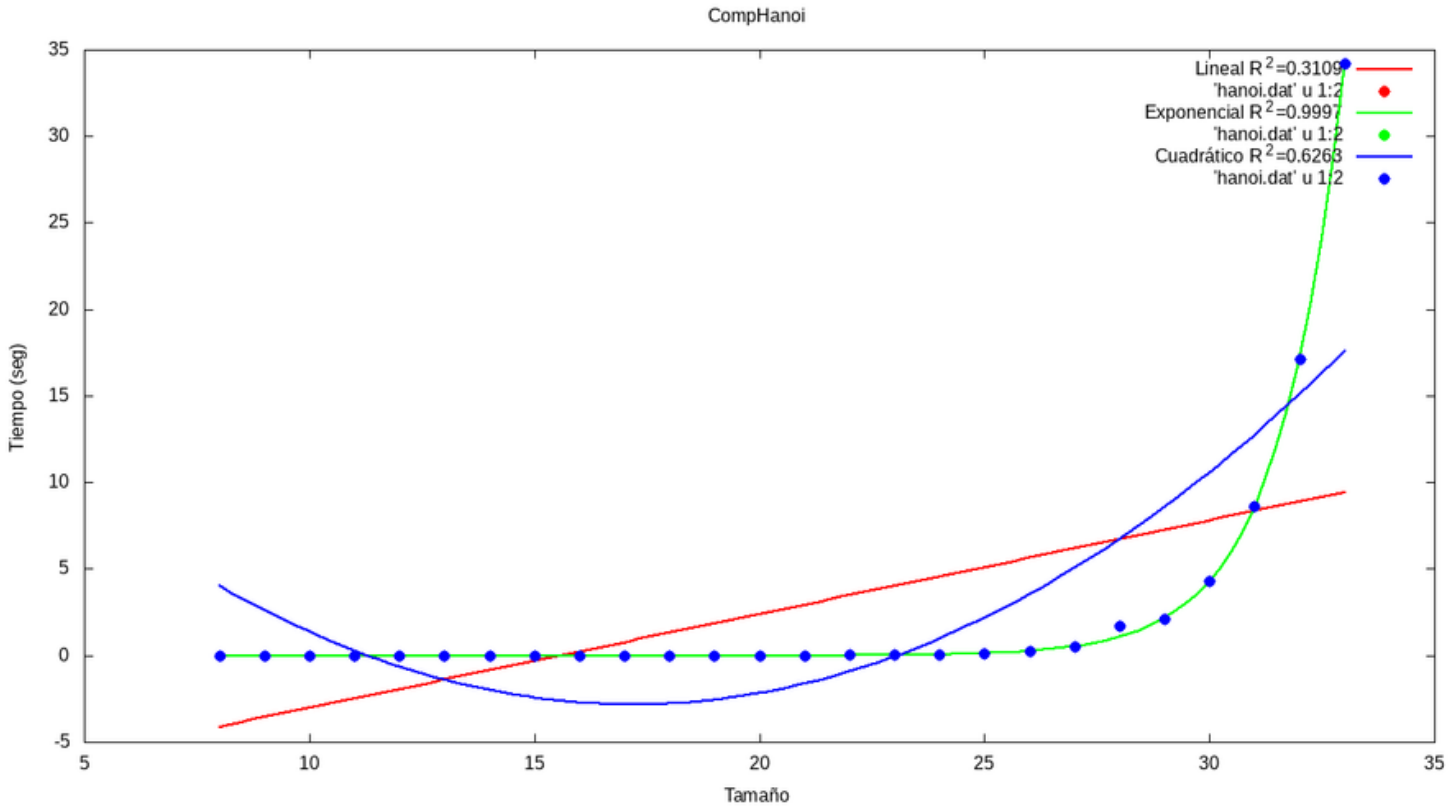
Estos son los datos obtenidos para cada uno de los miembros del grupo:

Tamanio	Tiempo L	Tiempo J	Tiempo A	Tiempo C
8	8.13E-06	6.87E-06	1.40E-06	1.33E-06
9	4.47E-06	2.20E-06	2.13E-06	2.53E-06
10	9.67E-06	4.13E-06	4.33E-06	4.93E-06
11	1.89E-05	8.27E-06	9.27E-06	9.27E-06
12	4.01E-05	1.62E-05	1.97E-05	1.85E-05
13	8.62E-05	3.42E-05	3.35E-05	3.67E-05
14	0.0001726	7.84E-05	6.75E-05	7.32E-05
15	0.000335933	0.0001594	0.000163867	0.0001466
16	0.0003758	0.000283867	0.000290333	0.000292467
17	0.000751133	0.000525933	0.0006296	0.000584933
18	0.00141293	0.00106393	0.00113267	0.00117013
19	0.0023494	0.00216953	0.002146	0.00233833
20	0.00470253	0.004419	0.00429733	0.00467433
21	0.0091522	0.00882973	0.0083758	0.00934853
22	0.0182772	0.0161823	0.0168065	0.0186939
23	0.0365451	0.0335385	0.0336234	0.0374174
24	0.0731529	0.0646779	0.0673519	0.0747874
25	0.146489	0.134183	0.133931	0.14981
26	0.302123	0.258518	0.267701	0.299274
27	0.587849	0.519062	0.534666	0.598559
28	1.1841	1.4867	1.6887	1.20378
29	2.36388	2.08613	2.13805	2.39315
30	4.80955	4.2826	4.27761	4.78664
31	9.56683	8.40774	8.60882	9.57596
32	19.1624	16.8145	17.1047	19.2778
33	39.2685	33.6576	34.2296	38.3052

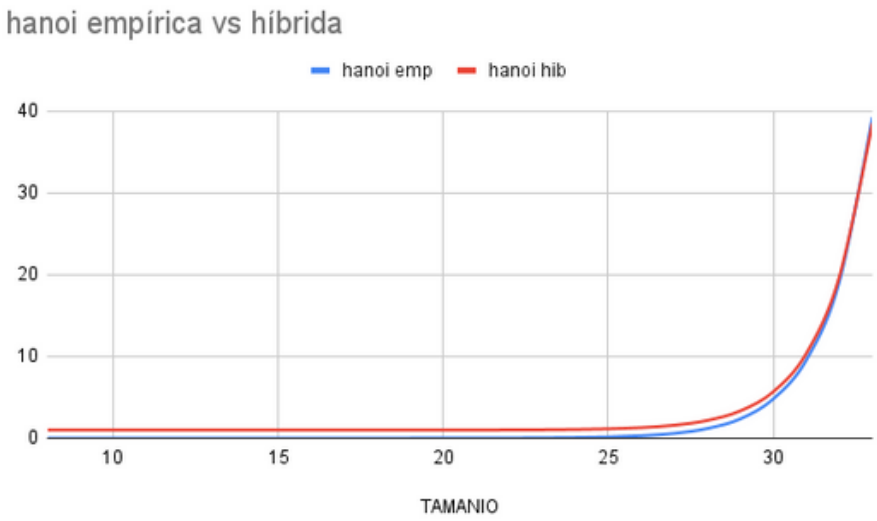


6.3 Eficiencia híbrida

Para la eficiencia híbrida nos hemos ayudado de gnuplot. Primero hemos introducido los datos conseguidos del algoritmo, y ajustando con la ecuación cuadrática conseguimos las variables ocultas. Hemos utilizado los valores de la gráfica A, y así conseguimos que las variables ocultas de la ecuación exponecial (a*2^x +b) son a=3.81258e-09 y b=1. A continuación se presenta la gráfica comparativa de este ajuste con otros, siendo estos el cuadrático y el lineal.



En la gráfica podemos ver que el ajuste que hemos escogido (el exponecial) es el idóneo, y otros erróneos que hemos escogido para ilustrarlo, claramente no concuerda con las expectaticas, por lo que no son correctos.



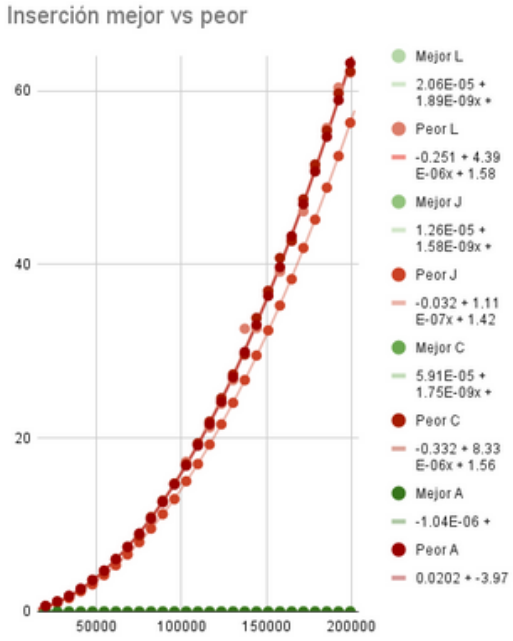
Además, hemos comparado la eficiencia empírica con la híbrida y, como podemos observar, son bastante similares, demostrando que la eficiencia del programa se ajusta a la que debería ser.

COMPARACIONES

1.Mejor vs Peor Caso

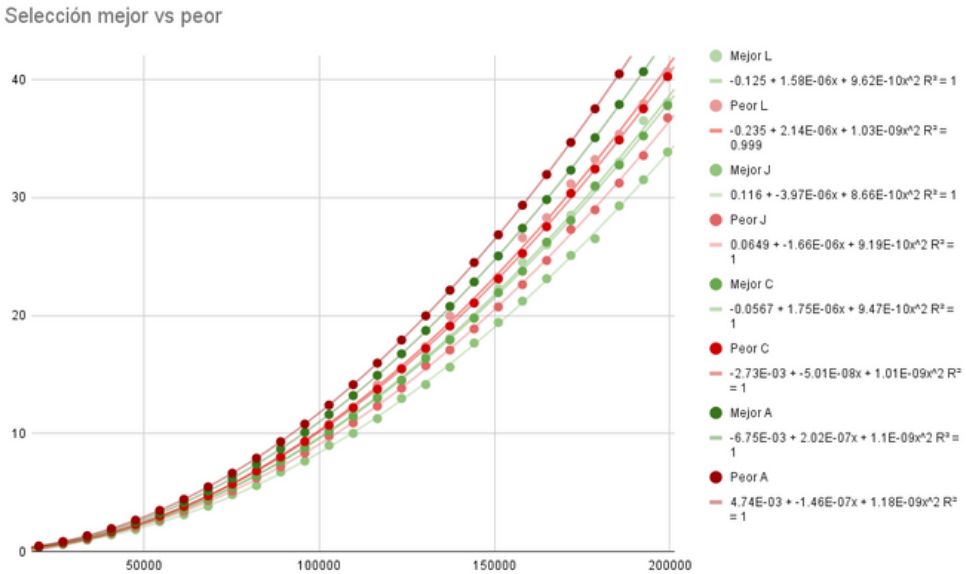
1.1 Algoritmo de Inserción

En este caso, podemos observar que la diferencia entre el mejor y el peor caso es abismal, siendo el caso mejor únicamente el tiempo necesario para recorrer completamente el vector.



1.2 Algoritmo de Selección

Por otro lado, en este caso podemos ver que la diferencia es escasa, ya que en cualquiera de los casos acaba recorriendo el vector el mismo número de veces.

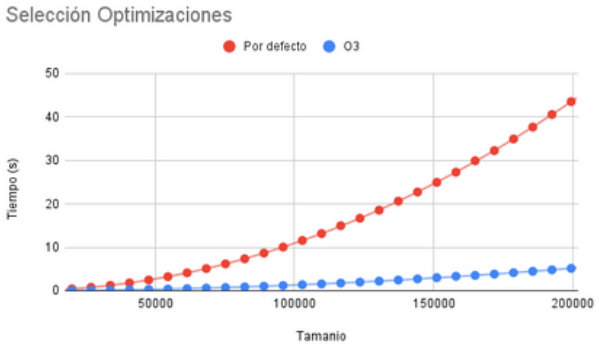


COMPARACIONES

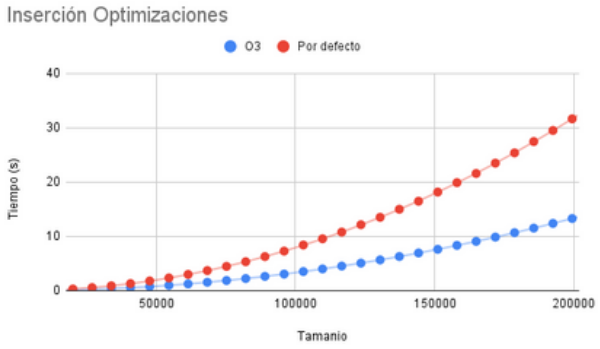
2.Optimización

En esta comparación se puede apreciar la gran utilidad que aporta el compilador con las herramientas para aumentar la eficiencia del código, ya que como se puede apreciar los tiempos de ejecución, aun siguiendo los mismos órdenes de eficiencia, reducen su tiempo en gran manera, convirtiéndose en un gran aliado a la hora de ejecutar grandes cantidades de código, pero todo sin sobreponerse al propio orden de eficiencia.

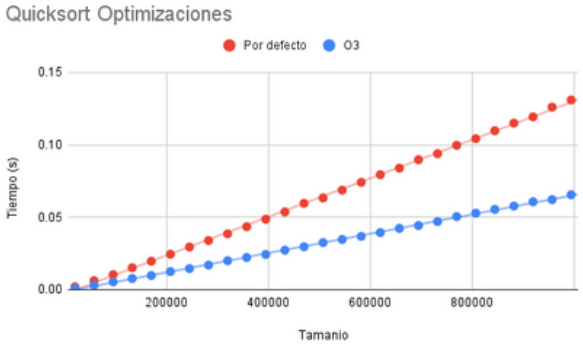
- Algoritmo de selección:



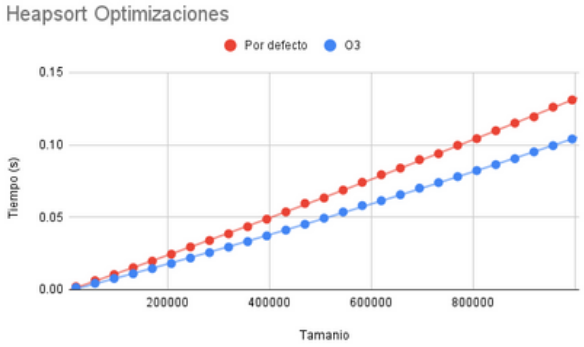
- Algoritmo de inserción:



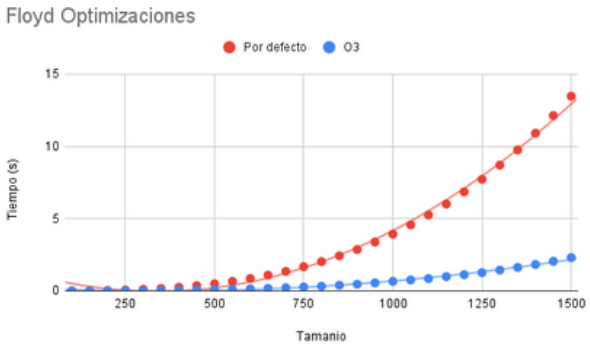
- Algoritmo de quicksort:



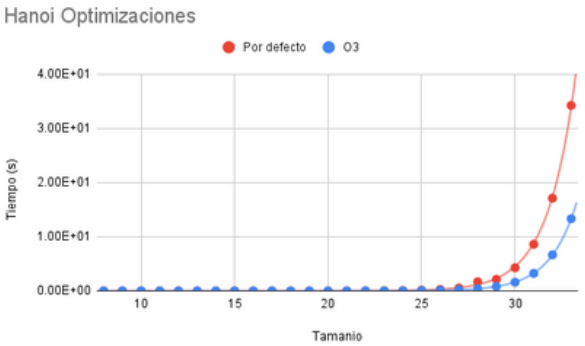
- Algoritmo de heapsort:



- Algoritmo de Floyd:



- Algoritmo de Hanoi:

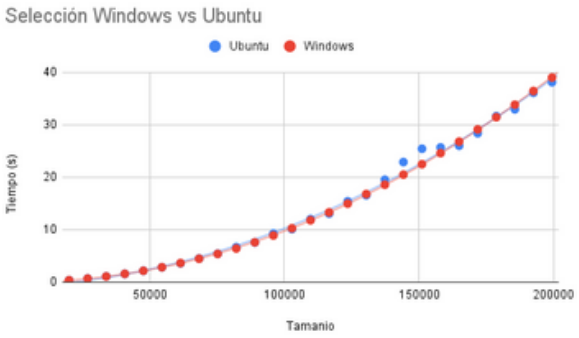


COMPARACIONES

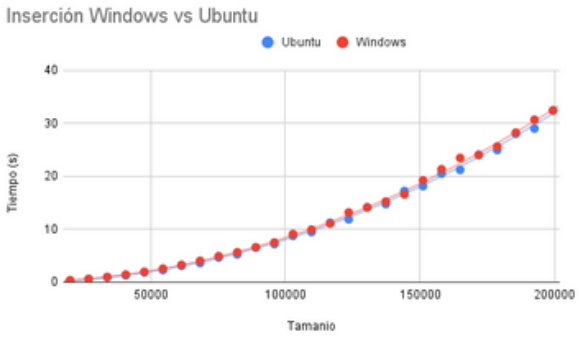
3.Windows vs Ubuntu

Tras ejecutar los algoritmos en distintos sistemas operativos, Windows y Ubuntu en este caso, podemos observar que no hay practicamente diferencia entre los tiempos resultantes.

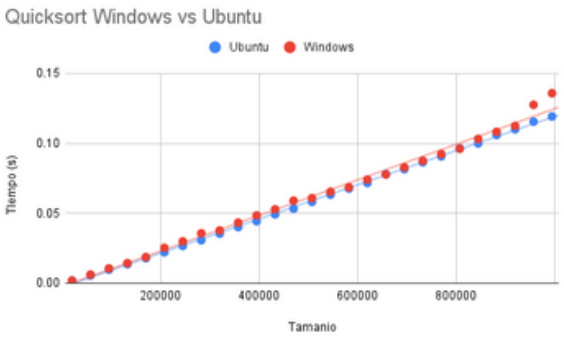
- Algoritmo de selección:



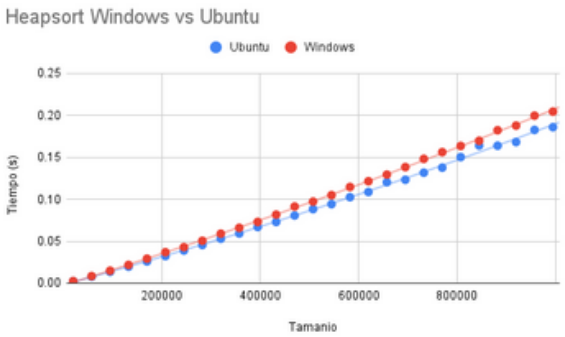
- Algoritmo de inserción:



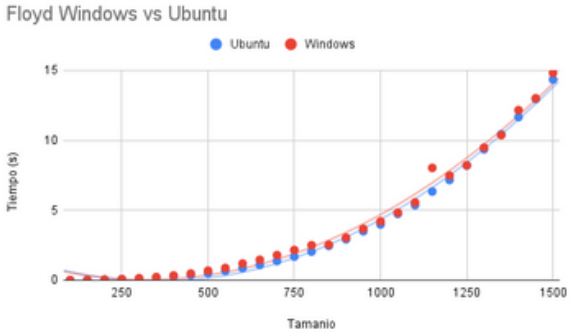
- Algoritmo de quicksort:



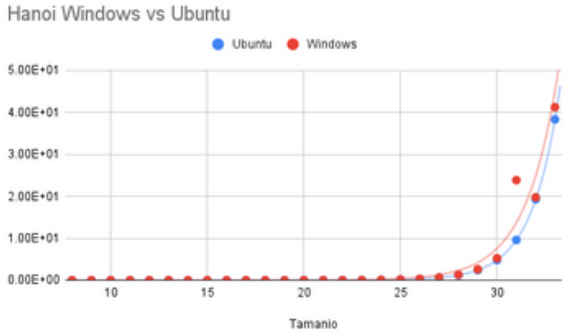
- Algoritmo de heapsort:



- Algoritmo de Floyd:



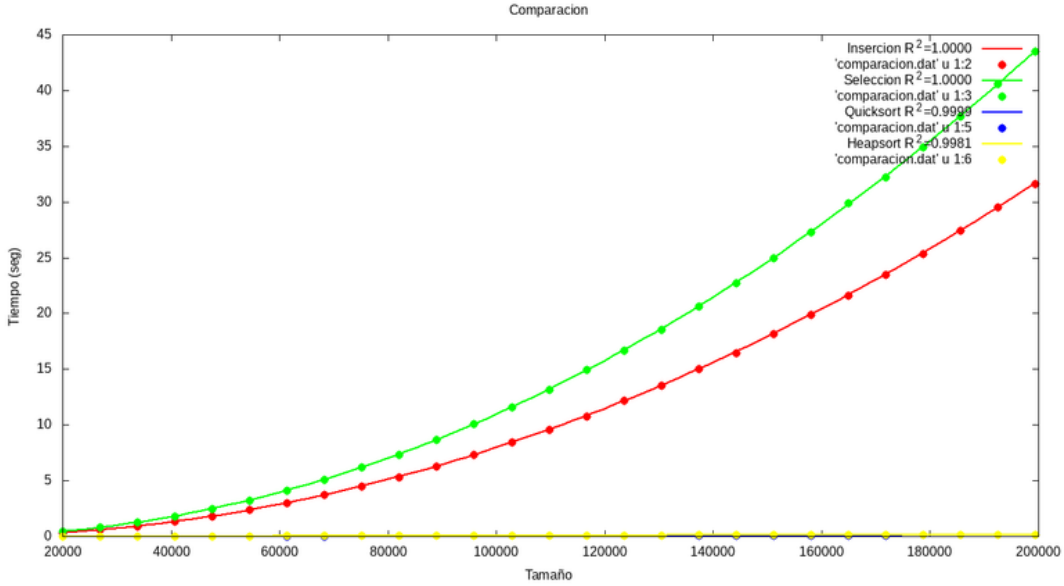
- Algoritmo de Hanoi:



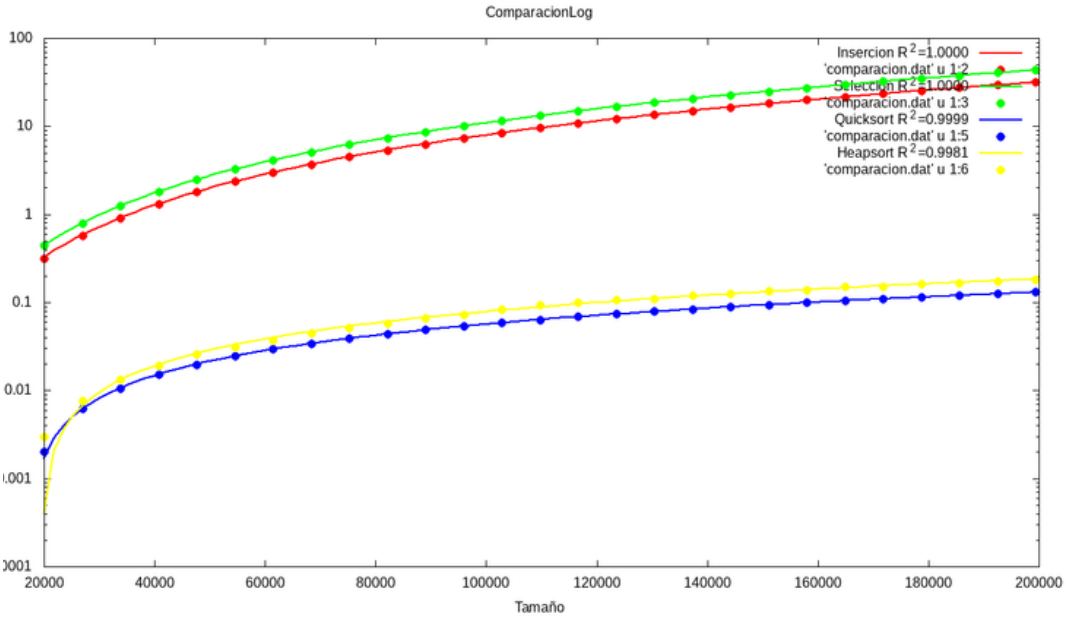
COMPARACIONES

4.Algoritmos

Como se puede observar en la gráfica, hay una diferencia abismal entre los dos algoritmos cuadráticos y los dos algoritmos n-logarítmicos, ya que, comparados con los tiempos de los primeros, los tiempos de los segundos son poco más que cero, y no se aprecia siquiera su forma.



Para observar mejor las diferencias entre los algoritmos hemos optado también por usar una gráfica en escala logarítmica en el eje y, que representa estas de una forma más diferenciada



CONCLUSIÓN

Para terminar con el trabajo, vamos a describir una breve conclusión.

Cabe entonces destacar la similitud entre los componentes de los ordenadores utilizados para este proyecto. Como esperábamos los datos obtenidos son bastante parecidos entre sí.

Además, comparando el análisis teórico elaborado con el análisis empírico obtenido a través de los diversos ordenadores, vemos que, además de coincidir entre ellos, también comparten la misma eficiencia de los algoritmos.

Haber estudiado la eficiencia desde el mismo ordenador, pero distinto sistema operativo nos presenta una gran similitud entre los tiempos obtenidos, además de presentar la misma eficacia, tal y como cabía esperar.

Nos hemos dado cuenta también que en los algorítmicos n_logarítmicos, la recopilación de datos mayores nos hubiera dado una mayor visibilidad a la hora de diferenciar con los diferentes ajustes, como el cuadrático y el lineal, y además la grafica tendría más forma de logaritmo.

El estudio elaborado sobre los mejores y peores casos en los algoritmos de búsqueda cuadráticos (insercion y selección) nos aporta información bastante interesante:

- Selección: en este caso podemos observar la minúscula diferencia entre el peor caso y el mejor, debido a que se recorre el vector las mismas veces en ambos casos.
- Inserción: a diferencia del algoritmo de selección, la diferencia entre el peor y el mejor caso es abismal, ya que el mejor caso es lineal y casi instantánea, mientras que el peor caso toma forma polinómica.

La práctica además nos ha ayudado a comprender como funcionan los algoritmos empleados, y el por qué de la eficacia asociada, comprendiendo que a veces la diferencia entre el peor y el mayor de los casos puede puede ser bastante significativa para determinar la eficacia de ellos.

Por último, cabe destacar la importancia de la eficiencia constante que se presenta en los distintos ordenadores, aunque tengan propiedades del hardware diferentes, distintos sistemas operativos, o incluso si se han compilado con diferentes optimizaciones. Esto nos lleva a la reflexionar sobre la importancia de la eficiencia de los algoritmos .