UNIVERSIDAD DE GRANADA

Escuela Técnica Superior de Ingeniería Informática y Telecomunicaciones

Práctica 1: Análisis de eficiencia de algoritmos

Doble Grado Ingeniería Informática y Matemáticas

Autores:
Jose Alberto Hoces Castro
Javier Gómez López
Moya Martín Castaño

Marzo 2022

Índice

1. Introducción

A modo de notación, destacamos las siguientes equivalencias:

- *Ordenador de Javier* $\longrightarrow$ i7-6700 3.40 GHz.
- *Ordenador de Jose Alberto* $\longrightarrow$ i5-1035G1 1.00 GHz.
- *Ordenador de Manuel* $\longrightarrow$i5-7200 2.5GHz.

Esta primera práctica, **Práctica 1**, consiste en el análisis de eficiencia de algoritmos, consiste en tres partes distintas:

- **Análisis de la eficiencia teórica:** estudio de la complejidad teórica del algoritmos (Mejor caso, peor caso y caso promedio).
- **Análisis de la eficiencia empírica:** ejecución y medición de tiempos de ejecución de los algoritmos estudiados.
- **Análisis de la eficiencia híbrida:** obtención de las constantes ocultas

A continuación, se explican en más profundidad dichas partes.

1.1. Análisis de la eficiencia teórica

El análisis de la **eficiencia teórica** consiste en analizar el tiempo de ejecución de los algoritmos dados para encontrar el peor de los casos, es decir, en qué clase de funciones en notación $\mathcal{O}$ grande se encuentran. Para ello, hemos utilizado las técnicas de análisis de algoritmos vistas en clase y en la asignatura *Estructura de Computadores.*

1.2. Análisis de la eficiencia empírica

Para el análisis de la **eficiencia empírica**, hemos ejecutado los algoritmos en cada uno de nuestros equipos bajo las mismas normas y condiciones, hemos medido el tiempo de ejecución de dichos algoritmos con la biblioteca `<chrono>`, basándonos en la siguiente estructura del código:

```
#include <chrono>
...

high_resolution_clock::time_point tantes, tdespues;
duration <double> transcurrido;
..

tantes = high_resolution_clock::now();
//Sentencia o programa a medir
tdepues = high_resolution_clock::now();
transcurrido = duration_cast<duration<double>>(tdespues-tantes);
```

Además, para automatizar el proceso de ejecución de los algoritmos, hemos usado la siguiente estructura para generar nuestros scripts:

```
i = #valor de la primera iteracion

while [ $i -le #valor ultima iteracion ]
do
./programa_a_ejecutar $i >> salida.dat
i=$[i+#salto entre valores para conseguir 26 puntos]
done
```

Hemos ejecutado cada algoritmo 15 veces en cada uno de los tamaños que han sido probados, y hemos hecho la media de ellos para reducir perturbaciones que puedan ocurrir de manera aleatoria y que nos lleven al mejor o peor caso, obteniendo de esta forma el caso promedio.

Cabe destacar que para *seleccion* e *insercion* hemos además ejecutado dos programas adicionales para obtener el mejor y peor caso de estos, pero este hecho lo detallaremos más adelante.

1.3. Análisis de la eficiencia híbrida

Para el análisis de la eficiencia híbrida, hemos tomado los datos de cada uno de los alumnos del grupo y hemos hallado la K (constante oculta). Para ello, hemos usado gnuplot.

Lo primero que hacemos es definar la función a la que queremos ajustar los datos. Tenemos que tener en cuenta el análisis teórico que hemos realizado previamente para saber cuál va a ser la forma de esta función. Podemos definir esta función en gnuplot mediante el siguiente comando (ejemplo para $\mathcal{O}(n^2)$):

```
gnuplot> f(x) = a0*x*x+a1*x+a2
```

El siguiente paso es indicarle a gnuplot que haga la regresión usando el método de mínimos cuadrados:

```
gnuplot> fit f(x) 'salida.dat' via a0,a1,a2
```

donde `'salida.dat'` es nuestro dataset.

La parte que más nos interesa es la parte donde pone `Final set of parameters`, pues ahí están nuestros coeficientes junto con la bondad del ajuste realizado.

2. Desarrollo

A continuación, realizaremos el estudio individual de cada algortimo, como se ha descrito anteriormente.

2.1. Inserción

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

2.1.1. Eficiencia teórica

Tal y como se ha indicado en los comentarios del código, todas las operaciones de asignación son $\mathcal{O}(1)$. Estas, a su vez, se incluyen en un bucle `for` y un bucle `while`, que están anidados, y que por ser cada uno $\mathcal{O}(n)$, multiplicamos lor órdenes de ambos como se vio en teoría, obteniendo que la función `static void insercion_lims` es $\mathcal{O}(n^2)$, es decir,

$$T(n) \in \mathcal{O}(n^2)$$

donde $T(n)$ es la función que expresa el tiempo de ejecución del algoritmo.

2.1.2. Eficiencia empírica

Tras ejecutar el algortimo en un rango de 10000 a 200000 elementos, con saltos de 7600 unidades por ejecución, obtenemos los siguientes resultados:

En este caso, al igual que en el resto de algoritmos, percibimos un poco de diferencia entre los tiempos de ejecución, debido a las diferentes circunstancias de cada integrante del grupo, pues poseemos dispositivos con distinto potencial.

2.1.3. Eficiencia híbrida

El estudio de la eficiencia híbrida consiste en hallar la expresión de las funciones que representan el tiempo de ejecución a partir de un tamaño dado. Usando los datasets del anterior apartado, hemos usado gnuplot para graficar los 26 puntos obtenidos junto con su función de ajuste. A continuación mostramos la gráfica con los ajustes de cada uno de los integrantes:

Y las constantes ocultas son:

- Ordenador Javier $\rightarrow T_1(n) = 8{,}49924 \cdot 10^{-10}x^2 - 8{,}57879 \cdot 10^{-6}x + 0{,}546581$.
- Ordenador José Alberto $\rightarrow T_2(n) = 7{,}96341 \cdot 10^{-10}x^2 + 2{,}23563 \cdot 10^{-5}x - 0{,}592279$.
- Ordenador Manuel $\rightarrow T_3(n) = 1{,}04394 \cdot 10^{-9}x^2 + 1{,}58593 \cdot 10^{-6}x - 0{,}0969414$.

Ordenador Javier	
Elementos (n)	**Tiempo (s)**
17600	0.251124
25200	0.560574
32800	0.905768
40400	1.33038
48000	1.87672
55600	2.51991
63200	3.29735
70800	4.08019
78400	4.98866
86000	6.08448
93600	7.17045
101200	8.36352
108800	9.71516
116400	11.0357
124000	12.5611
131600	14.1345
139200	15.7984
146800	17.6155
154400	19.5025
162000	21.4432
169600	23.5908
177200	25.7055
184800	27.9704
192400	30.2777
200000	32.8911

Ordenador José Alberto	
Elementos (n)	**Tiempo (s)**
17600	0.35799
25200	0.609488
32800	0.998112
40400	1.52076
48000	2.12746
55600	2.89747
63200	3.74891
70800	4.70754
78400	6.08267
86000	6.88299
93600	8.15529
101200	9.6372
108800	10.9647
116400	12.6405
124000	14.1936
131600	16.3756
139200	18.3599
146800	20.0244
154400	22.1302
162000	24.3748
169600	26.8462
177200	30.5882
184800	30.0598
192400	32.0387
200000	34.7391

Ordenador Manuel	
Elementos (n)	**Tiempo (s)**
17600	0.321303
25200	0.661228
32800	1.12508
40400	1.705
48000	2.41886
55600	3.23931
63200	4.18747
70800	5.2435
78400	6.4519
86000	7.74454
93600	9.18276
101200	10.7333
108800	12.4379
116400	14.2603
124000	16.1453
131600	18.1743
139200	20.4184
146800	22.6048
154400	25.0412
162000	27.5086
169600	30.1526
177200	32.9759
184800	35.8989
192400	38.8935
200000	41.9351

Cuadro 1: Experiencia empírica de algoritmo de Inserción sin optimizar

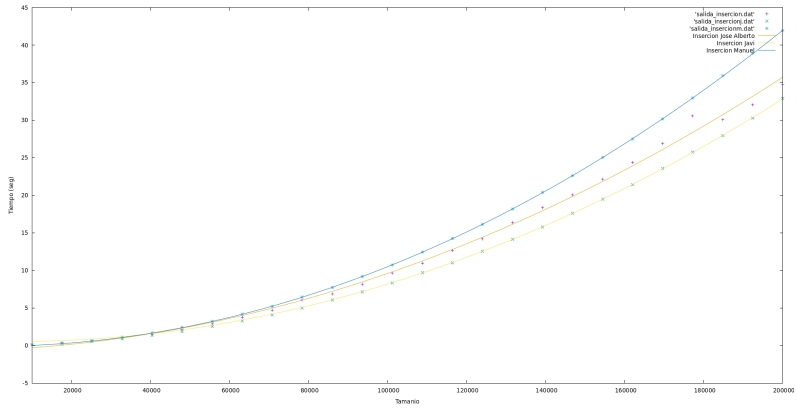


Figura 1: Gráfica con los tiempos de ejecución del algoritmo de Inserción

Para terminar nuestro análisis de este algoritmo, terminaremos de confirmar que el ajuste cuadrático es el óptimo viendo la varianza residual que nos ha proporcionado gnuplot:

- $T_1(n) \longrightarrow Var.res = 0{,}00162352$
- $T_2(n) \longrightarrow Var.res = 0{,}0050675$
- $T_3(n) \longrightarrow Var.res = 0{,}00161535$

Como todas son muy próximas a 0, podemos asegurar que el ajuste es muy bueno.

2.2. Selección

```
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++) { // O(n)
    indice_menor = i; // O(1)
    menor = T[i]; // O(1)
    for (j = i; j < final; j++) // O(n)
      if (T[j] < menor) {
        indice_menor = j; // O(1)
        menor = T[j]; // O(1)
      }
    aux = T[i]; // O(1)
    T[i] = T[indice_menor]; // O(1)
    T[indice_menor] = aux; // O(1)
```

```
    };
}
```

2.2.1. Eficiencia teórica

Tal y como se ha indicado en los comentarios del código, todas las operaciones de asignación son $\mathcal{O}(1)$. Estas, a su vez, se incluyen en dos bucles `for` que están anidados, que por ser cada uno $\mathcal{O}(n)$, multiplicamos lor órdenes de ambos como se vio en teoría, obteniendo que la función `static void seleccion_lims` es $\mathcal{O}(n^2)$, es decir,

$$T(n) \in \mathcal{O}(n^2)$$

donde $T(n)$ es la función que expresa el tiempo de ejecución del algoritmo.

2.2.2. Eficiencia empírica

Tras ejecutar el algortimo en un rango de 10000 a 200000 elementos, con saltos de 7600 unidades por ejecución, obtenemos los siguientes resultados:

Ordenador Javier	
Elementos (n)	**Tiempo (s)**
17600	0.260828
25200	0.504322
32800	0.835328
40400	1.25944
48000	1.7931
55600	2.54346
63200	3.42159
70800	4.46734
78400	5.61827
86000	6.80333
93600	8.16667
101200	9.86304
108800	11.3351
116400	12.9138
124000	14.7895
131600	16.9792
139200	18.6877
146800	20629
154400	23.2312
162000	25691
169600	28.1704
177200	30.78
184800	33.7999
192400	36.3688
200000	39.5352

Ordenador José Alberto	
Elementos (n)	**Tiempo (s)**
17600	0.391322
25200	0.76325
32800	1.27203
40400	1.92909
48000	2.71989
55600	3.65109
63200	4.71913
70800	5.91709
78400	7.25541
86000	8777
93600	10.3443
101200	12.0904
108800	14.0135
116400	16.0454
124000	18.19
131600	20.5113
139200	22.8553
146800	25.4735
154400	28141
162000	31.0438
169600	33.9582
177200	37.0641
184800	40.3583
192400	43.7206
200000	47.2209

Ordenador Manuel	
Elementos (n)	**Tiempo (s)**
17600	0.357489
25200	0.730079
32800	1.25708
40400	1.90694
48000	2.69298
55600	3.61969
63200	4.72056
70800	6.01156
78400	7.41941
86000	8.98684
93600	10.7273
101200	12.5778
108800	14.6332
116400	16.8798
124000	19.0523
131600	21.5316
139200	24.0439
146800	26.9219
154400	29.7736
162000	32.9393
169600	36.1122
177200	39.2833
184800	42.7955
192400	46.6683
200000	50.4019

Cuadro 2: Experiencia empírica de algoritmo de Selección sin optimizar

Observamos pequeñas diferencias en los tiempos de ejecución de cada uno de los ordenadores de los integrantes del grupo, y esto se debe a las condiciones específicas de cada uno de nuestros dispositivos y las prestaciones que estos tienen.

2.2.3. Eficiencia híbrida

Gracias al estudio de la eficiencia híbrida veremos que el ajuste teórico hecho hace dos subapartados es el correcto. Para ello, hemos tomado los datasets recién mostrados y hemos generado una gráfica en la que se representan los 26 tiempos obtenidos en función de los tamaños que hemos probado. Gnuplot nos ha facilitado esta gráfica junto con las constantes específicas asociadas a cada uno de nuestro, así como las varianzas residuales:

Y las constantes ocultas son:

- Ordenador Javier $\rightarrow T_1(n) = 1{,}0371 \cdot 10^{-9}x^2 + -9{,}86278 \cdot 10^{-6}x + 0{,}0216418$.
- Ordenador José Alberto $\rightarrow T_2(n) = 1{,}17905 \cdot 10^{-9}x^2 + 3{,}97249 \cdot 10^{-7}x - 0{,}00421685$.
- Ordenador Manuel $\rightarrow T_3(n) = 1{,}29484 \cdot 10^{-9}x^2 - 7{,}43377 \cdot 10^{-6}x + 0{,}0733569$.

Y para terminar de confirmar que nuestro ajuste es el correcto, podemos ver el valor de la varianza residual en cada caso:

- $T_1(n) \longrightarrow Var.res = 0{,}0164518$
- $T_2(n) \longrightarrow Var.res = 0{,}000537586$
- $T_3(n) \longrightarrow Var.res = 0{,}00387134$

Vemos que en todos los casos el ajuste cuadrático nos da varianzas muy próximas a 0, por lo que es un ajuste óptimo.

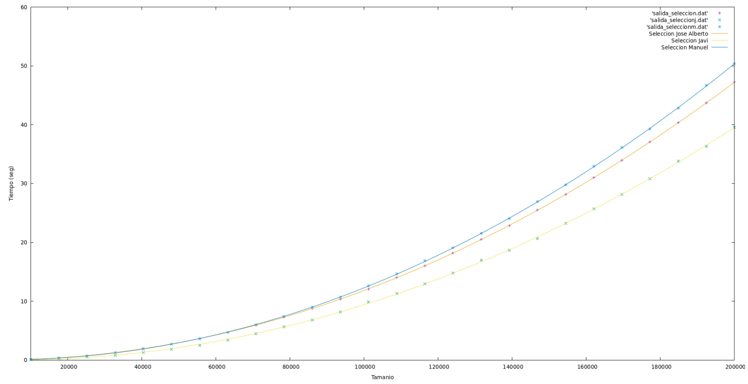


Figura 2: Gráfica con los tiempos de ejecución del algoritmo de Selección

2.3. Quicksort

```
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);          //O(n)
    quicksort_lims(T, inicial, k);
    quicksort_lims(T, k + 1, final);
  };
}


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));      //O(n)
  do {
    l--;
  } while (T[l] > pivote);
  while (k < l) {                           //O(n)
    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;
};
```

2.3.1. Eficiencia teórica

Tras un análisis rápido del código podemos ver como quicksort es una función recursiva que divide el vector a ordenar de tamaño n en dos vectores de tamaño aproximadamente $\frac{n}{2}$. Obviaremos la parte de la función en la que por debajo del umbral se ejecuta la función `insercion_lims` ya que no es de nuestro interés en el análisis. Luego es fácil ver que la ecuación recurrente asociada es la siguiente ya que la función `dividir_qs` es evidentemente $\mathcal{O}(n)$:
$T(n) = 2T(\frac{n}{2}) + n$
cuya solución sabemos que es por los resultados vistos en clase
$T(n) = c1 \cdot n + c2 \cdot n \log(n) donde c1, c2 \in \mathbb{R}^+$
Por tanto es claro que ocurre que:

$$T(n) \in \mathcal{O}(n \log(n))$$

donde $T(n)$ es la función que expresa el tiempo de ejecución del algoritmo.

2.3.2. Eficiencia empírica

Tras ejecutar el algortimo en un rango de 100000 a 2000000 elementos, con saltos de 76000 unidades por ejecución, obtenemos los siguientes resultados:

Ordenador Javier	
Elementos (n)	**Tiempo (s)**
176000	0.0204758
252000	0.0328
328000	0.0423033
404000	0.0499678
480000	0.0581441
556000	0.0679442
632000	0.0781102
708000	0.0877897
784000	0.0981701
860000	0.107906
936000	0.118434
1012000	0.129027
1088000	0.139197
1164000	0.14944
1240000	0.160327
1316000	0.169802
1392000	0.180587
1468000	0.191436
1544000	0.201973
1620000	0.212855
1696000	0.223251
1772000	0.234445
1848000	0.2444
1924000	0.25494
2000000	0.265787

Ordenador José Alberto	
Elementos (n)	**Tiempo (s)**
176000	0.0256406
252000	0.0370882
328000	0.0482084
404000	0.0611705
480000	0.0719635
556000	0.0839136
632000	0.0969097
708000	0.107333
784000	0.122233
860000	0.131432
936000	0.144035
1012000	0.158176
1088000	0.168443
1164000	0.181351
1240000	0.193264
1316000	0.206478
1392000	0.219195
1468000	0.231497
1544000	0.243313
1620000	0.257193
1696000	0.271076
1772000	0.285365
1848000	0.299989
1924000	0.308371
2000000	0.320497

Ordenador Manuel	
Elementos (n)	**Tiempo (s)**
176000	0.0262329
252000	0.038546
328000	0.0511573
404000	0.0639862
480000	0.0766981
556000	0.0896447
632000	0.1026
708000	0.115985
784000	0.129564
860000	0.142968
936000	0.156471
1012000	0.170292
1088000	0.184214
1164000	0.197666
1240000	0.211558
1316000	0.22524
1392000	0.239413
1468000	0.252893
1544000	0.267751
1620000	0.28121
1696000	0.295194
1772000	0.309444
1848000	0.323865
1924000	0.337683
2000000	0.351026

Cuadro 3: Experiencia empírica de algoritmo de Quicksort

Al igual que ocurría en anteriores algoritmos las diferencias son pequeñas entre los distintos tiempos de ejecución y se deben a que se han utillizado distintos ordenadores en la ejecución de los programas.

2.3.3. Eficiencia híbrida

Comprobamos que la función de ajuste obtenida en el apartado teórico es correcta con la eficiencia híbrida. De esta forma utilizando los datasets de los integrantes del grupo lo comprobaremos.

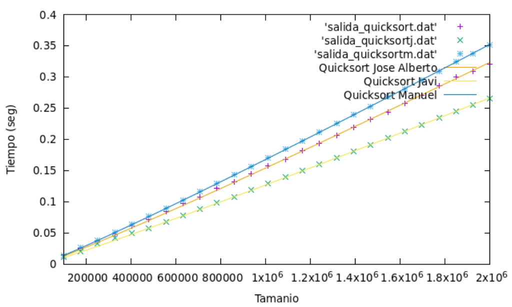


Figura 3: Gráfica con los tiempos de ejecución del algoritmo de Quicksort

Tras representar en la gráfica los 26 puntos obtenidos con la ejecución del algoritmo de Quicksort, con gnuplot vemos que las constantes ocultas son:

- Ordenador Javier $\rightarrow T_1(n) = 6{,}30121 \cdot 10^{-9} x \log(x) + 0{,}00188054$.
- Ordenador José Alberto $\rightarrow T_2(n) = 7{,}62891 \cdot 10^{-9} x \log(x) + 0{,}00283755$.
- Ordenador Manuel $\rightarrow T_3(n) = 8{,}39387 \cdot 10^{-9} x \log(x) + 0{,}000666717$.

Podemos observar que nuestro análisis teórico es correcto. Además, podemos observarlo con el coeficiente de regresión para cada una de nuestra funciones de ajuste:

- $T_1(n) \longrightarrow Var.res = 0{,}000000796551$
- $T_2(n) \longrightarrow Var.res = 0{,}00000158349$

- $T_3(n) \longrightarrow Var.res = 0{,}0000000913545$

Como los valores son muy próximos a 0 vemos realmente que el ajuste es bueno.

2.4. Heapsort

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


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)     //O(og(n))
    {
      j = k + k + 1;
      if ((j < (num_elem - 1)) && (T[j] < T[j+1]))     //O(1)
                j++;
      if (v >= T[j])                                    //O(1)
                esAPO = true;
      T[k] = T[j];
      k = j;
    }
  T[k] = v;

  //Total = O(log(n))
}
```

2.4.1. Eficiencia teórica

Observando el código podemos ver como el algoritmo se divide en dos bucles los cuales al no estar anidados cogeremos el tiempo del máximo de los dos. Por tanto es fácil ver como ambos bucles tienen la misma complejidad donde en el peor de los casos se ejecuta n veces y por tanto, al ser la función `reajustar` de complejidad $\mathcal{O}(\log(n))$, tenemos que la complejidad del algoritmo es $\mathcal{O}(n\log(n))$. La función `reajustar` es $\mathcal{O}(\log(n))$ porque en cada iteración se ejecuta como máximo la profundidad del árbol que se usa en el algoritmo, siendo dicha profundidad $\log(n)$. Por tanto tenemos que

$$T(n) \in \mathcal{O}(n\log(n))$$

donde $T(n)$ es la función que expresa el tiempo de ejecución del algoritmo.

2.4.2. Eficiencia empírica

Tras ejecutar el algortimo en un rango de 100000 a 2000000 elementos, con saltos de 76000 unidades por ejecución, obtenemos los siguientes resultados:
Se pueden observar pequeñas diferencias entre los distintos tiempos de ejecución pero no muy notables, debidas como hemos comentado anteriormente a las distintas características entre los ordenadores en los que se han ejecutado los programas.

2.4.3. Eficiencia híbrida

A continuación comprobaremos como la función de ajuste teórico obtenida es la correcta con la eficiencia híbrida. Tomaremos los datasets de los integrantes del grupo para visualizarlo.

Tras representar en la gráfica los 26 puntos obtenidos con la ejecución del algoritmo de Heapsort, es fácil calcular con gnuplot que las constantes ocultas son:

Ordenador Javier	
Elementos (n)	**Tiempo (s)**
176000	0.0298662
252000	0.0424575
328000	0.0557584
404000	0.0703343
480000	0.0849947
556000	0.100102
632000	0.11496
708000	0.130215
784000	0.145564
860000	0.161019
936000	0.176979
1012000	0.193231
1088000	0.208841
1164000	0.225306
1240000	0.242024
1316000	0.257695
1392000	0.273646
1468000	0.290959
1544000	0.306955
1620000	0.323347
1696000	0.341016
1772000	0.357579
1848000	0.37525
1924000	0.393665
2000000	0.410519

Ordenador José Alberto	
Elementos (n)	**Tiempo (s)**
176000	0.0351443
252000	0.061422
328000	0.0826658
404000	0.0982664
480000	0.11934
556000	0.109141
632000	0.125644
708000	0.142213
784000	0.159653
860000	0.177089
936000	0.192978
1012000	0.211015
1088000	0.229603
1164000	0.247574
1240000	0.26633
1316000	0.279554
1392000	0.300906
1468000	0.314309
1544000	0.335106
1620000	0.357587
1696000	0.376747
1772000	0.393121
1848000	0.412652
1924000	0.432016
2000000	0.527477

Ordenador Manuel	
Elementos (n)	**Tiempo (s)**
176000	0.0363807
252000	0.0540769
328000	0.0722292
404000	0.0911349
480000	0.1131
556000	0.141662
632000	0.161156
708000	0.173526
784000	0.193796
860000	0.213735
936000	0.235137
1012000	0.26813
1088000	0.292796
1164000	0.31178
1240000	0.327787
1316000	0.355995
1392000	0.392703
1468000	0.403927
1544000	0.427843
1620000	0.453352
1696000	0.478573
1772000	0.514379
1848000	0.543194
1924000	0.556125
2000000	0.582786

Cuadro 4: Experiencia empírica de algoritmo de Heapsort

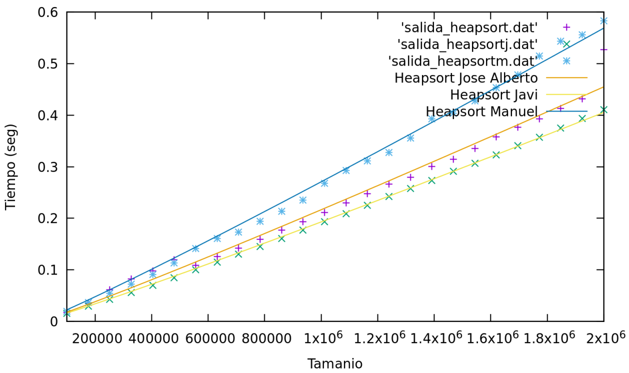


Figura 4: Gráfica con los tiempos de ejecución del algoritmo de Heapsort

- Ordenador Javier $\rightarrow T_1(n) = 9{,}81428 \cdot 10^{-9} x log(x) - 0{,}00367709$.
- Ordenador José Alberto $\rightarrow T_2(n) = 1{,}08739 \cdot 10^{-8} x log(x) - 0{,}000156439$.
- Ordenador Manuel $\rightarrow T_3(n) = 1{,}41147 \cdot 10^{-8} x log(x) - 0{,}0148078$.

Podemos observar que nuestro análisis teórico es correcto. Además, podemos observarlo con el coeficiente de regresión para cada una de nuestra funciones de ajuste:

- $T_1(n) \longrightarrow Var.res = 0{,}00265428$
- $T_2(n) \longrightarrow Var.res = 0{,}00031452$
- $T_3(n) \longrightarrow Var.res = 0{,}000056906$

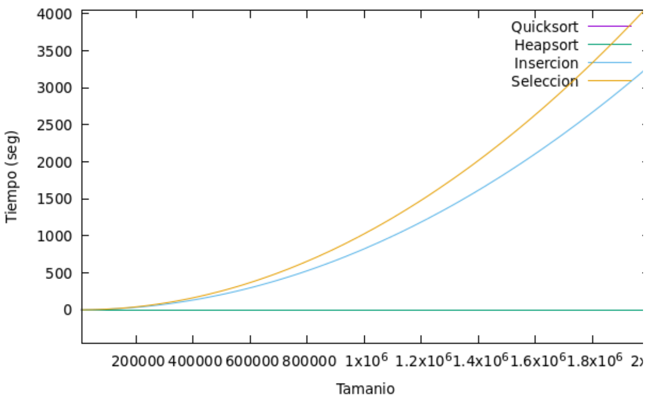
El ajusto es bueno pues los valores resultantes son próximos a 0.

2.5. Comparativa de los algoritmos de ordenación

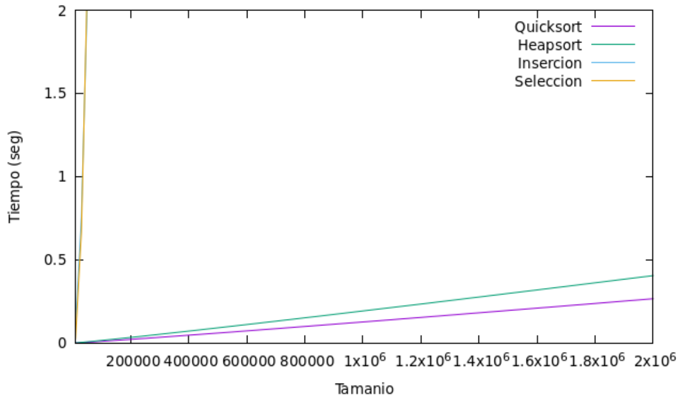
En este apartado vamos a ver claramente cuál es la diferencia entre los 4 algoritmos de ordenación que acabamos de analizar. Para ello, hemos generado una gráfica conjunta con las funciones de ajuste ya halladas antes, que eran:

- Inserción Javier $\rightarrow T_1(n) = 8{,}49924 \cdot 10^{-10} x^2 - 8{,}57879 \cdot 10^{-6} x + 0{,}546581$.
- Selección Javier $\rightarrow T_1(n) = 1{,}0371 \cdot 10^{-9} x^2 + -9{,}86278 \cdot 10^{-6} x + 0{,}0216418$.
- Quicksort Javier $\rightarrow T_1(n) = 9{,}18701 \cdot 10^{-9} \cdot x \cdot log(x)$.

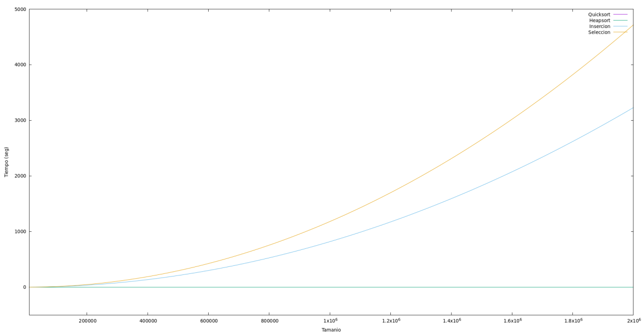
- Heapsort Javier $\rightarrow T_1(n) = 1{,}39707 \cdot 10^{-8} \cdot x \cdot log(x)$.
- Inserción José Alberto $\rightarrow T_2(n) = 7{,}96341 \cdot 10^{-10}x^2 + 2{,}23563 \cdot 10^{-5}x - 0{,}592279$.
- Selección José Alberto $\rightarrow T_2(n) = 1{,}17905 \cdot 10^{-9}x^2 + 3{,}97249 \cdot 10^{-7}x - 0{,}00421685$.
- Quicksort José Alberto $\rightarrow T_2(n) = 1{,}11515 \cdot 10^{-8} \cdot x \cdot log(x)$.
- Heapsort José Alberto $\rightarrow T_2(n) = 1{,}56798 \cdot 10^{-8} \cdot x \cdot log(x)$.
- Inserción Manuel $\rightarrow T_3(n) = 1{,}04394 \cdot 10^{-9}x^2 + 1{,}58593 \cdot 10^{-6}x - 0{,}0969414$.
- Selección Manuel $\rightarrow T_3(n) = 1{,}29484 \cdot 10^{-9}x^2 - 7{,}43377 \cdot 10^{-6}x + 0{,}0733569$.
- Quicksort Manuel $\rightarrow T_3(n) = 1{,}21439 \cdot 10^{-8} \cdot x \cdot log(x)$.
- Heapsort Manuel $\rightarrow T_3(n) = 1{,}96051 \cdot 10^{-8} \cdot x \cdot log(x)$.



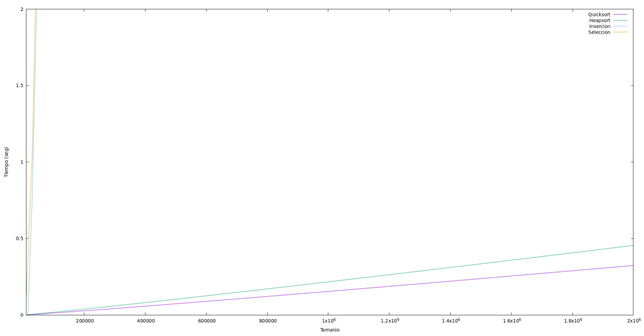
(a) Ordenador Javier



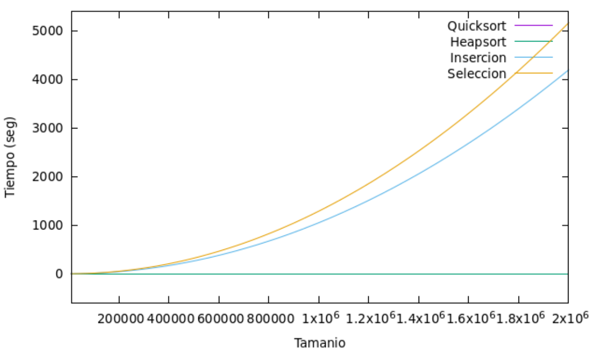
(b) Ordenador Javier



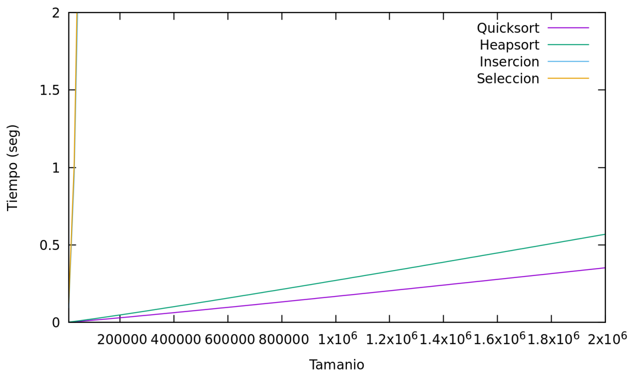
(a) Ordenador José Alberto



(b) Ordenador José Alberto



(a) Ordenador Manuel



(b) Ordenador Manuel

Hemos hecho 2 gráficas porque al realizar la primera de cada par correspondiente a cada integrante, vimos que quicksort y heapsort se veían como una línea paralela al eje X en la que no se diferencian entre sí. Esto se debe a la gran diferencia entre los tiempos de estos dos algoritmos con respecto a los que tienen eficiencia cuadrática, que son inserción y selección, los cuales llegan a más de 5000 segundos por ejecución, mientras que los de eficiencia logarítmica no llegan a sobrepasar el segundo por ejecución. Así, en la segunda gráfica restringimos el rango del eje Y de 0 a 2 para que se pudiese ver que realmente quicksort y heapsort sí se diferencian entre ellos y se ve que tienen cierta pendiente, insignificante respecto a la pendiente de inserción y selección. Por lo tanto, una vez más vemos que nuestro análisis tiene sentido y se corresponde con los datasets obtenidos.

2.6. Floyd

```
void Floyd(int **M, int dim)
{
        for (int k = 0; k < dim; k++) //O(n)
          for (int i = 0; i < dim;i++) //O(n)
            for (int j = 0; j < dim;j++) //O(n)
              {
                int sum = M[i][k] + M[k][j];
               M[i][j] = (M[i][j] > sum) ? sum : M[i][j]; //O(1)
              }
}               //Total O(n^3)
```

2.6.1. Eficiencia teórica

Como podemos observar en los comentarios del código que hemos hecho en la función `void Floyd`, estamos ante una función que pertenece a $\mathcal{O}(n^3)$. Son tres bucles `for` que están anidados, cada uno $\mathcal{O}(n)$, por tanto, multiplicando los órdenes obtenemos que la función es $\mathcal{O}(n^3)$, es decir,

$$T(n) \in \mathcal{O}(n^3)$$

donde $T(n)$ es la función que expresa el tiempo de ejecución del algoritmo.

2.6.2. Eficiencia empírica

Tras ejecutar el algoritmo en un rango de 176 a 2000 elementos, con saltos de 76 unidades por ejecución, obtenemos los siguientes resultados:

Ordenador Javier	
Elementos (n)	**Tiempo (s)**
176	0.0244106
252	0.0721776
328	0.155828
404	0.288165
480	0.465947
556	0.724968
632	1.09236
708	1.54374
784	2.13392
860	2.67022
936	3.52897
1012	4.4074
1088	5.42559
1164	6.6698
1240	8.06967
1316	9.55022
1392	11.4197
1468	13.3942
1544	15.5
1620	18.0399
1696	20.5893
1772	23.6714
1848	26.7337
1924	30.1601
2000	33.9673

Ordenador José Alberto	
Elementos (n)	**Tiempo (s)**
176	0.0274773
252	0.0995705
328	0.20657
404	0.307902
480	0.51806
556	0.799187
632	1.16729
708	1.65895
784	2.42549
860	3.00331
936	3.84788
1012	4.84029
1088	5.97643
1164	7.78043
1240	9.08228
1316	10.7251
1392	12.9933
1468	14.6689
1544	17.2185
1620	20.2626
1696	22.9733
1772	26.0557
1848	30.2843
1924	33.4252
2000	38.5217

Ordenador Manuel	
Elementos (n)	**Tiempo (s)**
176	0.038495
252	0.111472
328	0.244523
404	0.45528
480	0.761621
556	1.17395
632	1.73408
708	2.4355
784	3.29426
860	4.35444
936	5.64407
1012	7.16827
1088	8.91362
1164	10.9311
1240	13.2386
1316	15.8513
1392	18.7744
1468	21.9844
1544	25.5768
1620	29.5543
1696	33.8275
1772	38.5849
1848	43.8038
1924	49.4368
2000	55.3965

Cuadro 5: Experiencia empírica de algoritmo de Floyd sin optimizar

Observamos pequeñas diferencias, pero en general nada fuera de lo común. Estas diferencias son debidas a los distintos agentes tecnológicos usados para la realización del análisis de la eficiencia empírica en esta práctica.

2.6.3. Eficiencia híbrida

A través de la eficiencia híbrida, comprobaremos que el ajuste teórico realizado es correcto. Para realizar este análisis, tomamos los datasets de todos los integrantes del grupo.

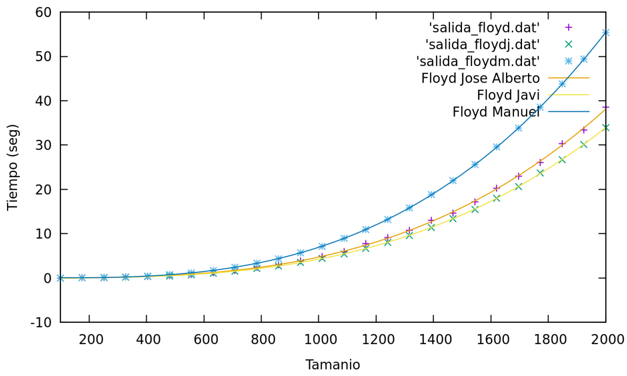


Figura 8: Gráfica con los tiempos de ejecución del algoritmo de Floyd

En esta gráfica están representados los 26 puntos obtenidos tras la ejecución del algoritmo de Floyd en los distintos equipos de los integrantes del grupo. Tras una serie de cálculos con gnuplot, observamos que las constantes ocultas son:

- Ordenador Javier $\to T_1(n) = 4{,}38237 \cdot 10^{-9}x^3 - 4{,}33753 \cdot 10^{-7}x^2 + 0{,}000337001x - 0{,}0504332$.
- Ordenador José Alberto $\to T_2(n) = 5{,}12922 \cdot 10^{-9}x^3 - 1{,}11315 \cdot 10^{-6}x^2 + 0{,}00083571x - 0{,}134397$.
- Ordenador Manuel $\to T_3(n) = 6{,}77297 \cdot 10^{-9}x^3 + 5{,}13099 \cdot 10^{-7}x^2 - 0{,}000427834x + 0{,}0714028$.

Podemos observar que nuestro análisis teórico es correcto. Además, podemos observarlo con el coeficiente de regresión para cada una de nuestra funciones de ajuste:

- $T_1(n) \longrightarrow Var.res = 0{,}00204522$
- $T_2(n) \longrightarrow Var.res = 0{,}044778$
- $T_3(n) \longrightarrow Var.res = 0{,}000855184$

Estos valores son muy cercanos a 0, y por tanto indican que el ajuste es muy bueno.

2.7. Hanoi

```
void hanoi (int M, int i, int j)
{
  if (M > 0)
    {
      hanoi(M-1, i, 6-i-j);
      hanoi (M-1, 6-i-j, j);
  }
}
```

2.7.1. Eficiencia teórica

En este caso no podemos realizar un análisis de la misma manera que en el algoritmo anterior, pues estamos ante un algoritmo recursivo. De esta manera, trataremos de buscar la relación de recurrencia.

Suponiendo que estamos en la n-ésima iteración, el algoritmo comprobará lo indicado en el `if`, que es de $\mathcal{O}(1)$, y volverá a llamarse a sí misma otras dos veces. Por tanto, la ecuación de recurrencia es la siguiente:

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

Si resolvemos la ecuación de recurrencia obtenemos que:

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

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

Y concluimos que $T(n) \in \mathcal{O}(2^n)$, donde $T(n)$ es la función que expresa el tiempo de ejecución de nuestro algortimo para n elementos.

2.7.2. Eficiencia empírica

Debido al orden del algoritmo, el número de elementos que tenemos que tomar es mucho menor que los que usamos en el resto de algortimos. En este caso, tras ejecutar el algoritmo en un rango de 7 a 32 elementos, con saltos de 1 elemento por iteración, obtenemos los siguientes resultados:

Ordenador Javier	
Elementos (n)	**Tiempo (s)**
8	0.00000136207
9	0.00000267907
10	0.00000528653
11	0.0000112702
12	0.0000234959
13	0.0000457819
14	0.0000904406
15	0.000198225
16	0.000439214
17	0.00088158
18	0.00145113
19	0.00253865
20	0.00499491
21	0.0100156
22	0.0209075
23	0.0402523
24	0.0878626
25	0.171153
26	0.339115
27	0.633015
28	1.28649
29	2.60592
30	5.05092
31	10.1126
32	20.301

Ordenador José Alberto	
Elementos (n)	**Tiempo (s)**
8	0.0000037376
9	0.00000737613
10	0.0000145867
11	0.0000283526
12	0.0000460821
13	0.0000722887
14	0.000106264
15	0.000213395
16	0.000353459
17	0.000717674
18	0.00142487
19	0.00278949
20	0.00534407
21	0.0101673
22	0.0238254
23	0.0555082
24	0.112827
25	0.207041
26	0.344851
27	0.761311
28	1.41561
29	2.68719
30	5.41493
31	9.82069
32	20.2358

Ordenador Manuel	
Elementos (n)	**Tiempo (s)**
8	0.0000017978
9	0.00000348253
10	0.00000737093
11	0.0000137999
12	0.0000274451
13	0.0000548052
14	0.000110116
15	0.000198426
16	0.000427075
17	0.000796963
18	0.00159355
19	0.00321857
20	0.00633508
21	0.012697
22	0.0253476
23	0.0506946
24	0.101314
25	0.202542
26	0.405264
27	0.809707
28	1.6195
29	3.23942
30	6.47798
31	12.9623
32	25.9203

Cuadro 6: Experiencia empírica de algoritmo de Hanoi

Aquí si que observamos grandes diferencias entre los dos primeros equipos y el tercero. Esto puede ser debido a el procesador de estos, o el hecho de que el tercer equipo es un portátil y la ejecución del programa se realizó sin cargar el equipo. Esto en ocasiones puede provocar bajada de rendimiento.

2.7.3. Eficiencia híbrida

Este análisis confirmará nuestro análisis teórico. Para realizar este análisis, tomamos los datasets de todos los integrantes del grupo.

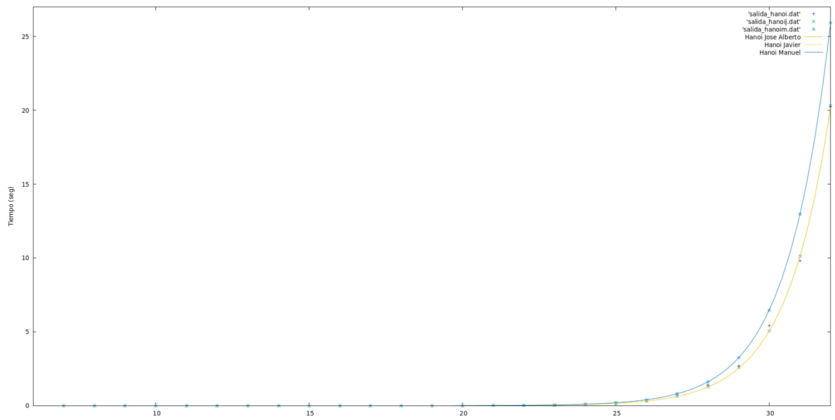


Figura 9: Gráfica con los tiempos de ejecución del algoritmo de las torres de Hanoi

En esta gráfica están representados los 26 puntos obtenidos tras la ejecución del algoritmo de Hanoi en los distintos equipos de los integrantes del grupo. Tras una serie de cálculos con gnuplot, observamos que las constantes ocultas son:

- Ordenador Javier $\rightarrow T_1(n) = 4{,}72408 \cdot 10^{-9} \cdot 2^x$.
- Ordenador José Alberto $\rightarrow T_2(n) = 4{,}70707 \cdot 10^{-9} \cdot 2^x$.
- Ordenador Manuel $\rightarrow T_3(n) = 6{,}03512 \cdot 10^{-9} \cdot 2^x$.

Podemos observar que nuestro análisis teórico es correcto. Además, podemos observarlo con el coeficiente de regresión para cada una de nuestras funciones de ajuste:

- $T_1(n) \longrightarrow Var.res = 0{,}000302074$
- $T_2(n) \longrightarrow Var.res = 0{,}0113386$
- $T_3(n) \longrightarrow Var.res = 3{,}87795 \cdot 10^{-7}$

Estos valores son muy cercanos a 0, y por tanto indican que el ajuste es muy bueno.

3. Casos especiales

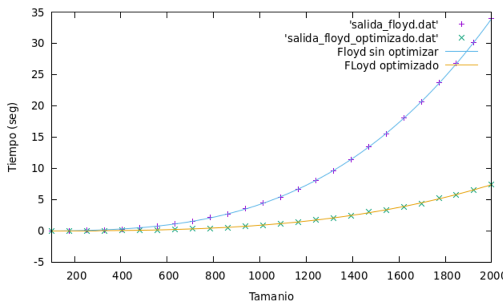
Además del análisis mostrado de los seis algoritmos anteriores, también se ha realizado un análisis de algunos de ellos bajo condiciones distintas, para mostrar así además una experiencia más amplia y diversa y conseguir un mejor entendimiento de los algortimos trabajados.

3.1. Floyd optimizado

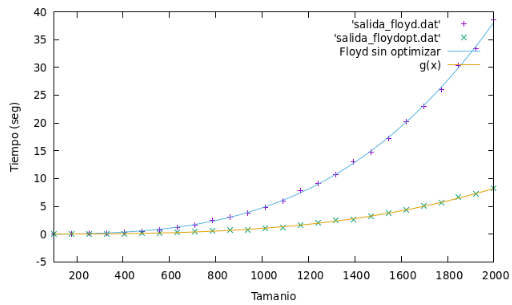
En este caso, queríamos mostrar la diferencia que obtenemos cuando realizamos la compilación de nuestro código bajo ciertas condiciones que pueden modificar "la pureza del mismo".

Con una compilación normal, el compilador tratará de convertir nuestros `.cpp` a código máquina de la manera más fiel posible. Sin embargo, si la introducimos la orden `-Og` estamos indicando a este que reduzca en lo máximo la ineficiencia de nuestro código, optimizándolo.

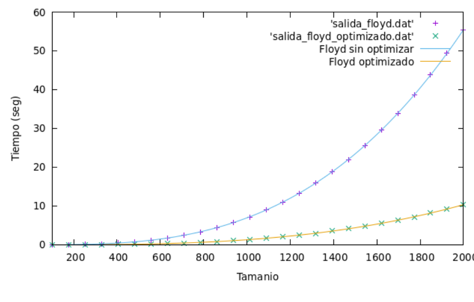
A continuación, se muestra una comparación de tiempos de ejecución:



(a) Ordenador Javier



(b) Ordenador José Alberto



(c) Ordenador Manuel

Ordenador Javier	
T. Sin optimizar (s)	**T. Optimizado (s)**
0.0244106	0.00554839
0.0721776	0.0154236
0.155828	0.0333405
0.288165	0.0614654
0.465947	0.10192
0.724968	0.156739
1.09236	0.228405
1.54374	0.32042
2.13392	0.41954
2.67022	0.537747
3.52897	0.718656
4.4074	0.923106
5.42559	1.13561
6.6698	1.39419
8.06967	1.8317
9.55022	2.00464
11.4197	2.45868
13.3942	3.08659
15.5	3.26148
18.0399	3.84594
20.5893	4.38846
23.6714	5.19954
26.7337	5.77858
30.1601	6.54927
33.9673	7.4387

Ordenador José Alberto	
T. Sin optimizar (s)	**T. Optimizado (s)**
0.0274773	0.00755066
0.0995705	0.0167159
0.20657	0.0389106
0.307902	0.0851063
0.51806	0.143528
0.799187	0.182412
1.16729	0.350968
1.65895	0.419597
2.42549	0.537551
3.00331	0.677508
3.84788	0.78477
4.84029	0.991477
5.97643	1.22178
7.78043	1.65727
9.08228	1.98969
10.7251	2.42308
12.9933	2.67575
14.6689	3.24925
17.2185	3.78885
20.2626	4.32095
22.9733	5.026
26.0557	5.6235
30.2843	6.62451
33.4252	7.24857
38.5217	8.19595

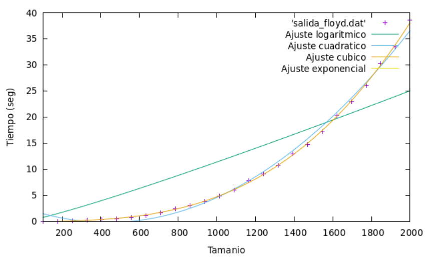
Ordenador Manuel	
T. Sin optimizar (s)	**T. Optimizado (s)**
0.038495	0.00753551
0.111472	0.0195449
0.244523	0.0442298
0.45528	0.0795414
0.761621	0.136642
1.17395	0.211945
1.73408	0.302502
2.4355	0.431404
3.29426	0.590716
4.35444	0.794588
5.64407	1.03657
7.16827	1.32908
8.91362	1.65594
10.9311	2.02603
13.2386	2.4668
15.8513	2.92335
18.7744	3.5334
21.9844	4.12724
25.5768	4.77742
29.5543	5.52418
33.8275	6.32056
38.5849	7.16984
43.8038	8.22908
49.4368	9.15268
55.3965	10.3229

Cuadro 7: Comparación optimización Floyd

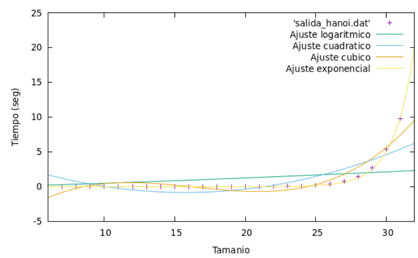
Podemos observar que el uso de la instrucción `-Og`, y todas sus variantes de su optimización, reducen considerablemente el tiempo de ejecución de nuestro código. Por tanto, su uso debe estar presente a la hora de compilar ciertos programas.

3.2. Otros posibles ajustes funcionales

A continuación, se muestran dos gráficas donde se observan otras posibilidades de ajuste para los algoritmos de Floyd y Hanoi, y se observa que los ajustes utilizados en los análisis previos son los mejores, confirmando nuestro análisis teórico.



(a) Algoritmo de Floyd



(b) Algoritmo de Hanoi

Vemos que los órdenes que habíamos obtenido en nuestro análisis teórico son los que mejor se ajustan a nuestra nube de puntos, siendo esto una confirmación de la bondad de nuestro análisis.

4. Casos en la ejecución de inserción y selección: mejor, peor y promedio

Otra de las tareas a realizar en esta práctica ha sido medir los tiempos para el mejor caso y peor caso de los algoritmos de inserción y selección y compararlos con el promedio, el cual ya hemos analizado anteriormente. El peor caso es el de un vector ordenado a la inversa, para lo cual hemos introducido en los códigos de inserción y selección el siguiente bucle:

```
for (int i = 0; i < n; i++)
    {
      T[i] = n - i;
    };
```

Y para el mejor caso hemos introducido un bucle que crea un vector ya ordenado:

```
for (int i = 0; i < n; i++)
    {
      T[i] = i;
    };
```

A continuación comenzamos mostrando los datasets del peor y mejor caso de selección y peor y mejor caso de inserción, junto a su representación gráfica:

Ordenador Javier	
Elementos (n)	**Tiempo (s)**
17600	0.294915
25200	0.643749
32800	1.03357
40400	1.59778
48000	2.23197
55600	2.99818
63200	3.8157
70800	5.11308
78400	6.28133
86000	7.52288
93600	8.77745
101200	10.6892
108800	12.3541
116400	13.8101
124000	16.0165
131600	18.2028
139200	20.2703
146800	22.6626
154400	25.1417
162000	27.8535
169600	31.1783
177200	34.1867
184800	36.2439
192400	39.3318
200000	42.4879

Ordenador José Alberto	
Elementos (n)	**Tiempo (s)**
17600	0.43298
25200	1.05541
32800	2.06101
40400	2.43286
48000	3.83567
55600	4.94909
63200	5.53895
70800	7.26986
78400	9.73586
86000	10.9612
93600	13.2779
101200	16.2492
108800	18.5379
124000	21.05
131600	24.3881
139200	27.383
146800	31.2269
154400	32.5782
162000	36.6571
169600	39.8171
177200	43.9252
184800	47.8018
192400	52.3513
200000	55.4702

Ordenador Moya	
Elementos (n)	**Tiempo (s)**
17600	0.426382
25200	0.874735
32800	1.4813
40400	2.24773
48000	3.1727
55600	4.25578
63200	5.48874
70800	6.9053
78400	8.45013
86000	10.1796
93600	12.0609
101200	14.0973
108800	16.2901
116400	18.6421
124000	21.1758
131600	23.8494
139200	26.6751
146800	29.6576
154400	32.8149
162000	36.1294
169600	39.7108
177200	43.2185
184800	47.0736
192400	50.9572
200000	55.0463

Cuadro 8: Datasets de la ejecución del peor caso para Selección

Ordenador Javier	
Elementos (n)	**Tiempo (s)**
17600	0.260234
25200	0.575504
32800	0.890206
40400	1.47874
48000	2.11678
55600	2.89931
63200	3.86399
70800	4.81428
78400	6.04563
86000	7.16048
93600	8.57613
101200	10.1238
108800	11.7447
116400	13.3527
124000	15.3081
131600	17.2786
139200	19.8345
146800	22.1474
154400	24.4118
162000	26.3101
169600	28.861
177200	31.5309
184800	34.3333
192400	37.2464
200000	40.3089

Ordenador José Alberto	
Elementos (n)	**Tiempo (s)**
17600	0.280803
25200	0.583418
32800	1.09678
40400	1.68938
48000	2.4139
55600	3.26631
63200	4.24695
70800	5.477
78400	6.74688
86000	8.11066
93600	9.54485
101200	11.4866
108800	15.3544
116400	15.8972
124000	17.35
131600	20.0171
139200	21.5481
146800	24.2838
154400	27.0934
162000	31.5408
169600	32.8615
177200	35.3771
184800	40.99
192400	51.2152
200000	53.4133

Ordenador Moya	
Elementos (n)	**Tiempo (s)**
17600	0.404172
25200	0.829167
32800	1.40418
40400	2.13082
48000	3.00788
55600	4.03586
63200	5.21357
70800	6.54202
78400	8.02234
86000	9.65307
93600	11.4335
101200	13.3659
108800	15.448
116400	17.6802
124000	20.0639
131600	22.5975
139200	25.2821
146800	28.116
154400	31.1008
162000	34.2373
169600	37.5229
177200	41.0938
184800	44.5503
192400	48.2842
200000	52.1786

Cuadro 9: Datasets de la ejecución del mejor caso para Selección

Ordenador Javier	
Elementos (n)	**Tiempo (s)**
17600 0.541558	52.1786
25200 1.042	52.1786
32800 1.7487	52.1786
40400 2.66625	52.1786
48000 3.76577	52.1786
55600 5.02335	52.1786
63200 6.5175	52.1786
70800 8.22182	52.1786
78400 9.99542	52.1786
86000 12.0556	52.1786
93600 14.307	52.1786
101200 16.6827	52.1786
108800 19.296	52.1786
116400 22.0829	52.1786
124000 25.091	52.1786
131600 28.6517	52.1786
139200 32.558	52.1786
146800 35.3069	52.1786
154400 39.0694	52.1786
162000 42.8941	52.1786
169600 46.9762	52.1786
177200 51.1649	52.1786
184800 56.4017	52.1786
192400 61.3711	52.1786
200000 65.3477	52.1786

Ordenador José Alberto	
Elementos (n)	**Tiempo (s)**
17600	0.604135
25200	1.23844
32800	2.193
40400	3.83771
48000	4.92643
55600	6.65173
63200	9.69109
70800	10.9557
78400	11.5942
86000	14.0269
93600	16.1625
101200	19.6924
108800	22.6823
116400	25.7428
124000	28.4764
131600	32.0522
139200	37.1527
146800	40.1051
154400	45.1864
162000	49.014
169600	56.3554
177200	58.3219
184800	69.8676
192400	75.6931
200000	81.4641

Ordenador Moya	
Elementos (n)	**Tiempo (s)**
17600	0.651326
25200	1.32968
32800	2.26908
40400	3.42426
48000	4.82964
55600	6.48277
63200	8.39325
70800	10.5201
78400	12.8629
86000	15.5291
93600	18.3665
101200	21.5378
108800	24.8369
116400	28.4794
124000	32.2615
131600	36.367
139200	40.736
146800	45.3961
154400	50.2232
162000	55.0915
169600	60.3752
177200	66.0735
184800	72.1095
192400	77.8989
200000	84.1458

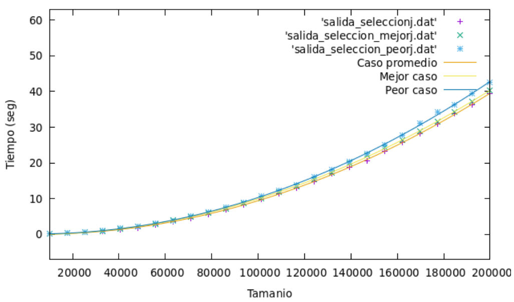
Cuadro 10: Datasets de la ejecución del peor caso para Inserción

Ordenador Javier	
Elementos (n)	**Tiempo (s)**
17600 3.7468e-05	84.1458
25200 5.3663e-05	84.1458
32800 0.000106905	84.1458
40400 9.3457e-05	84.1458
48000 0.000110966	84.1458
55600 0.000128851	84.1458
63200 0.000164639	84.1458
70800 0.000252943	84.1458
78400 0.000198703	84.1458
86000 0.000178361	84.1458
93600 0.000193726	84.1458
101200 0.000210377	84.1458
108800 0.000213392	84.1458
116400 0.000228798	84.1458
124000 0.000243479	84.1458
131600 0.000312568	84.1458
139200 0.000280611	84.1458
146800 0.000295903	84.1458
154400 0.000311014	84.1458
162000 0.000325946	84.1458
169600 0.000340919	84.1458
177200 0.000388042	84.1458
184800 0.000372042	84.1458
192400 0.000387548	84.1458
200000 0.000461947	84.1458

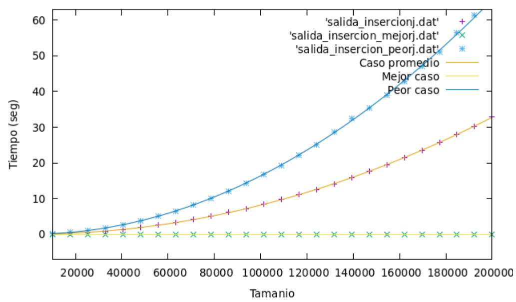
Ordenador José Alberto	
Elementos (n)	**Tiempo (s)**
25200	0.000154839
32800	0.000254579
40400	0.000268849
48000	0.000315367
55600	0.000221064
63200	0.000253476
70800	0.000218935
78400	0.000203775
86000	0.000202846
93600	0.000349709
101200	0.000310166
108800	0.000319687
116400	0.000467919
124000	0.000452129
131600	0.000604762
139200	0.000651269
146800	0.00058346
154400	0.000549033
162000	0.000648826
169600	0.000551999
177200	0.000769622
184800	0.00049894
192400	0.000406281
200000	0.000481777

Ordenador Moya	
Elementos (n)	**Tiempo (s)**
17600	0.000067881
25200	0.000090394
32800	0.000135741
40400	0.000167224
48000	0.000284697
55600	0.000230947
63200	0.000261484
70800	0.000292646
78400	0.000325176
86000	0.000362248
93600	0.000320247
101200	0.000498554
108800	0.000320164
116400	0.000339361
124000	0.000316208
131600	0.00033449
139200	0.000370265
146800	0.000403541
154400	0.000392099
162000	0.000424873
169600	0.000434162
177200	0.000453234
184800	0.000509574
192400	0.000492694
200000	0.000515208

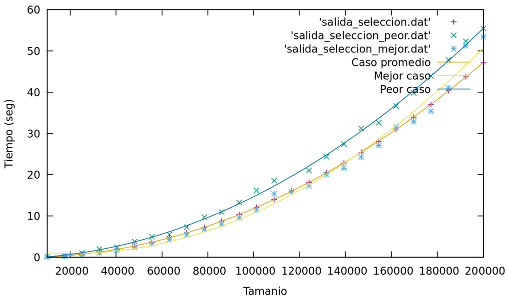
Cuadro 11: Datasets de la ejecución del mejor caso para Inserción



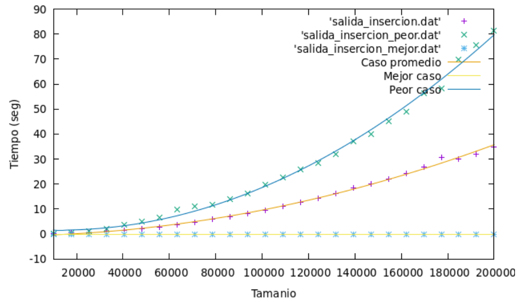
(a) Selección Javier



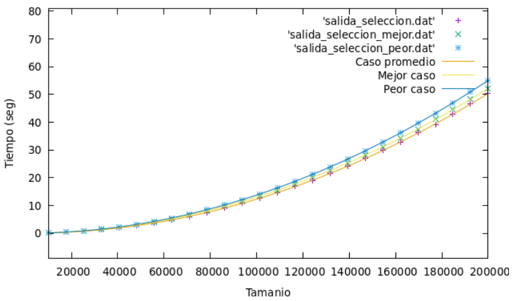
(b) Inserción Javier



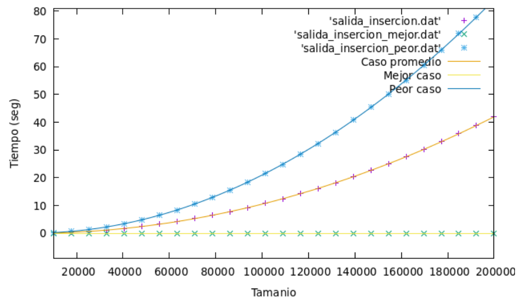
(a) Selección José Alberto



(b) Inserción José Alberto



(a) Selección Manuel



(b) Inserción Manuel

Como hemos apreciado en las gráficas, en inserción los casos se diferencian perfectamente, tardando casi 0 segundos para el mejor caso y tardando mucho más para el peor caso. Sin embargo, en el algoritmo de selección vemos que las gráficas oscilan en torno a los mismos valores. Esto se debe a cómo están hechos los códigos. En el algoritmo de selección, se comienza hallando el mínimo de los n elementos y se coloca en la primera posición. Tras esto, se calcula el mínimo de los n-1 elementos restantes que no están ordenados y se coloca en segunda posición, y así sucesivamente hasta llegar al final. Por ello, independientemente de que el vector esté ordenado o no, siempre va a tener que recorrer el vector de la forma descrita para hallar todos los mínimos, de ahí que los tiempos en los 3 casos no se diferencien mucho.

Por otra parte, en inserción sí se diferencian, y esto se debe a que se ordena de una forma concreta. Este algoritmo ordena "subconjuntos"del vector empezando con los dos primeros elementos. Una vez ordena los dos primeros, inserta el tercero en la posición correcta de entre estos dos. En la siguiente iteración, añade el cuarto elemento a los tres ya ordenados y así sucesivamente hasta acabar. La razón principal de por qué tarda tanto cuando está ordenado es porque cada vez que se va a añadir un elemento a los ya ordenados, se comienza comparando con el último de los ya ordenados, es decir, el mayor de todos (esta comprobación se realiza en el bucle `while` que se encuentra dentro de un `for`):

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

De esta forma, como el vector ya está ordenado, la condición del bucle `while` nunca se da y por lo tanto en cada iteración del `for` se ahorra la ejecución del cuerpo del bucle `while` y solo se realizan comparaciones entre pares de números consecutivos.

5. Conclusiones

Como conclusiones de esta práctica, hemos observado los siguientes hechos a destacar:

- El orden de eficiencia de un algoritmo $\mathcal{O}$ es un factor clave cuando el número de iteraciones a realizar es grande.
- Usar optimización a la hora de compilar nuestros programas debe de ser un factor a tener en cuenta, pues puede suponernos una gran ventaja en lo relativo al tiempo de ejecución.
- El análisis teórico es importante, pues puede darnos información sobre si merece la pena implementar un algoritmo o no,
- El análisis híbrido y la obtención de las constantes ocultas son una manera de confirmar nuestros análisis teórico.