



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



Este trabajo se distribuye bajo una licencia CC BY-NC-SA 4.0.

Eres libre de distribuir y adaptar el material siempre que reconozcas a los autores originales del documento, no lo utilices para fines comerciales y lo distribuyas bajo la misma licencia.

creativecommons.org/licenses/by-nc-sa/4.0/

Índice

1. Introducción	3
1.1. Análisis de la eficiencia teórica	3
1.2. Análisis de la eficiencia empírica	3
1.3. Análisis de la eficiencia híbrida	3
2. Desarrollo	4
2.1. Inserción	4
2.1.1. Eficiencia teórica	4
2.1.2. Eficiencia empírica	4
2.1.3. Eficiencia híbrida	5
2.2. Selección	5
2.2.1. Eficiencia empírica	5
2.2.2. Eficiencia híbrida	6
2.3. Quicksort	7
2.4. Heapsort	7
2.5. Comparativa de los algoritmo de ordenación	7
2.6. Floyd	8
2.6.1. Eficiencia teórica	8
2.6.2. Eficiencia empírica	8
2.6.3. Eficiencia híbrida	9
2.7. Hanoi	9
2.7.1. Eficiencia teórica	9
2.7.2. Eficiencia empírica	10
2.7.3. Eficiencia híbrida	10
3. Casos especiales	10
3.1. Floyd optimizado	10
3.2. Otros posibles ajustes funcionales	11

1. Introducción

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
tdespues = 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 casos 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 definir 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:

```
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.

2. Desarrollo

A continuación, realizaremos el estudio individual de cada algoritmo, 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 $O(1)$. Estas, a su vez, se incluyen en un bucle **for** y un bucle **while**, que están anidados, y que por ser cada uno $O(n)$, multiplicamos los órdenes de ambos como se vio en teoría, obteniendo que la función **static void insercion_lims** es $O(n^2)$, es decir,

$$T(n) \in O(n^2)$$

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

2.1.2. Eficiencia empírica

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

Intel Core i7-6700 3.40 GHz		Ordenador Jota		Ordenador Moya	
Elementos (n)	Tiempo (s)	Elementos (n)	Tiempo (s)	Elementos (n)	Tiempo (s)
17600	0.251124	17600	0.35799	17600	0.321303
25200	0.560574	25200	0.609488	25200	0.661228
32800	0.905768	32800	0.998112	32800	1.12508
40400	1.33038	40400	1.52076	40400	1.705
48000	1.87672	48000	2.12746	48000	2.41886
55600	2.51991	55600	2.89747	55600	3.23931
63200	3.29735	63200	3.74891	63200	4.18747
70800	4.08019	70800	4.70754	70800	5.2435
78400	4.98866	78400	6.08267	78400	6.4519
86000	6.08448	86000	6.88299	86000	7.74454
93600	7.17045	93600	8.15529	93600	9.18276
101200	8.36352	101200	9.6372	101200	10.7333
108800	9.71516	108800	10.9647	108800	12.4379
116400	11.0357	116400	12.6405	116400	14.2603
124000	12.5611	124000	14.1936	124000	16.1453
131600	14.1345	131600	16.3756	131600	18.1743
139200	15.7984	139200	18.3599	139200	20.4184
146800	17.6155	146800	20.0244	146800	22.6048
154400	19.5025	154400	22.1302	154400	25.0412
162000	21.4432	162000	24.3748	162000	27.5086
169600	23.5908	169600	26.8462	169600	30.1526
177200	25.7055	177200	30.5882	177200	32.9759
184800	27.9704	184800	30.0598	184800	35.8989
192400	30.2777	192400	32.0387	192400	38.8935
200000	32.8911	200000	34.7391	200000	41.9351

Cuadro 1: Experiencia empírica de algoritmo de Selección sin optimizar

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:

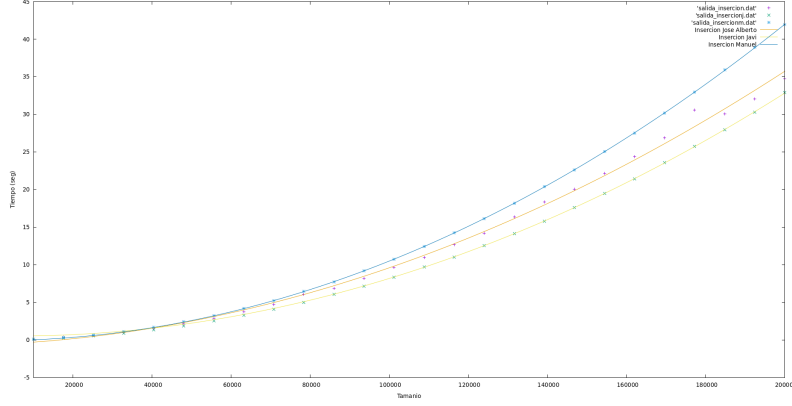


Figura 1: Gráfica con los tiempos de ejecución del algoritmo de Inserción

Y las constantes ocultas son:

- i7-6700 3.40Ghz $\rightarrow T_1(n) = 8,49924 \cdot 10^{-10}x^2 - 8,57879 \cdot 10^{-6}x + 0,546581$.
- Ordenador José Alberto $\rightarrow T_2(n) = 7,96341 \cdot 10^{-10}x^2 + 2,23563 \cdot 10^{-5}x - 0,592279$.
- Ordenador Manuel $\rightarrow T_3(n) = 1,04394 \cdot 10^{-9}x^2 + 1,58593 \cdot 10^{-6}x - 0,0969414$.

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:

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

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

2.2. Selección

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 los 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.

2.2.1. Eficiencia empírica

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

Intel Core i7-6700 3.40 GHz	
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	20.629
154400	23.2312
162000	25.691
169600	28.1704
177200	30.78
184800	33.7999
192400	36.3688
200000	39.5352

Ordenador Jota	
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	8.777
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	28.141
162000	31.0438
169600	33.9582
177200	37.0641
184800	40.3583
192400	43.7206
200000	47.2209

Ordenador Moya	
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.

2.2.2. 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:

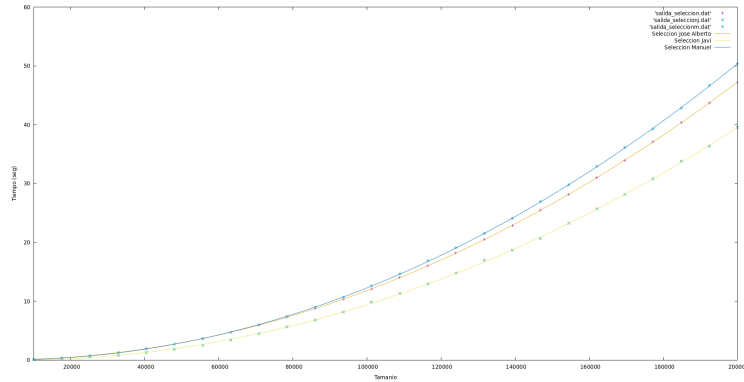


Figura 2: Gráfica con los tiempos de ejecución del algoritmo de Selección

Y las constantes ocultas son:

- i7-6700 3.40Ghz $\rightarrow T_1(n) = 1,0371 \cdot 10^{-9}x^2 + -9,86278 \cdot 10^{-6}x + 0,0216418$.
- 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:

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

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

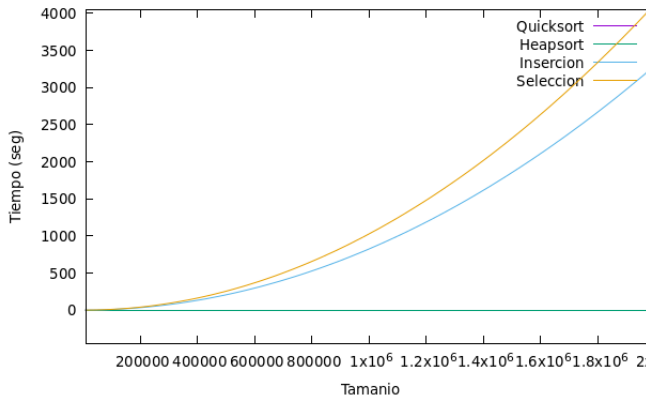
2.3. Quicksort

2.4. Heapsort

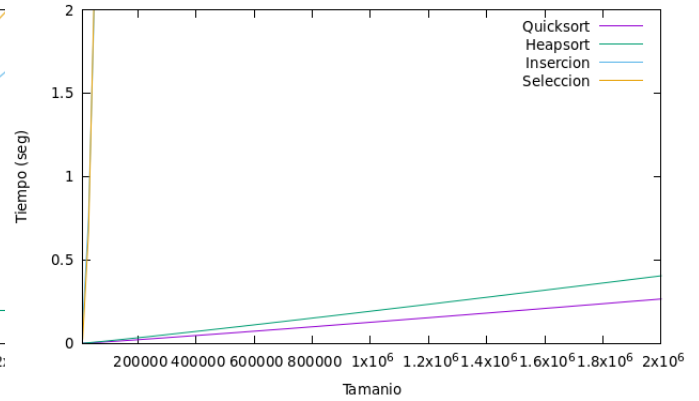
2.5. Comparativa de los algoritmo de ordenación

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:

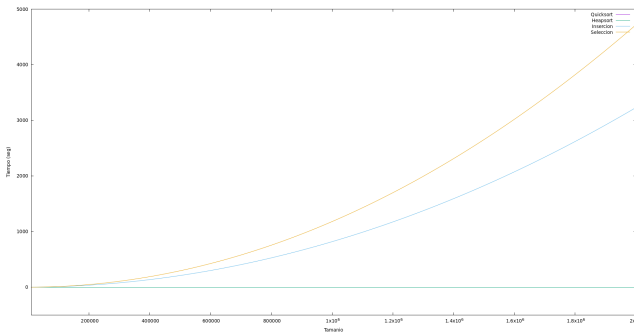
- Inserción Javier $\rightarrow T_1(n) = 8,49924 \cdot 10^{-10}x^2 - 8,57879 \cdot 10^{-6}x + 0,546581$.
- Selección Javier $\rightarrow T_1(n) = 1,0371 \cdot 10^{-9}x^2 + -9,86278 \cdot 10^{-6}x + 0,0216418$.
- Quicksort Javier $\rightarrow T_1(n) = 9,18701 \cdot 10^{-9} \cdot x \cdot \log(x)$.
- Heapsort 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 $\rightarrow T_3(n) = 1,96051 \cdot 10^{-8} \cdot x \cdot \log(x)$.



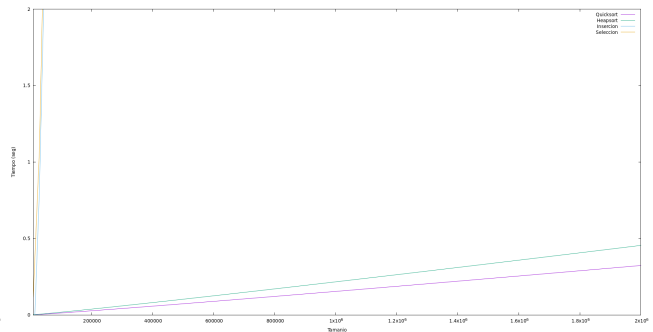
(a) Ordenador Javier



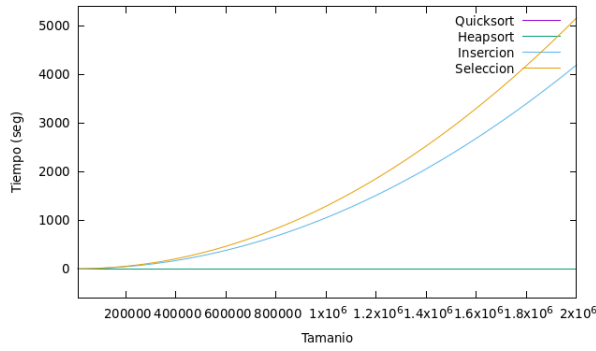
(b) Ordenador Javier



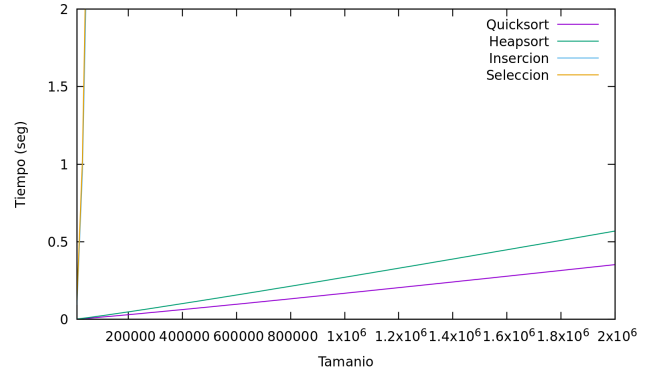
(a) Ordenador José Alberto



(b) Ordenador José Alberto



(a) Ordenador Manuel



(b) Ordenador Manuel

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:

Intel Core i7-6700 3.40 GHz	
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

i5-1095G1 1.00 GHz	
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 Moya	
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 3: 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.

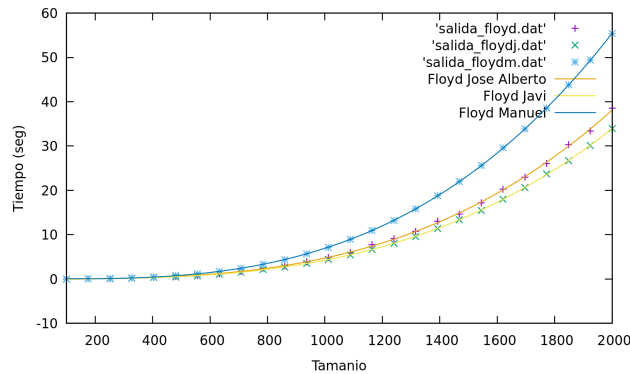


Figura 6: Gráfica con los tiempos de ejecución del algoritmo de Floyd

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:

- i7-6700 3.40GHz $\rightarrow T_1(n) = 4,38237 \cdot 10^{-9}x^3 - 4,33753 \cdot 10^{-7}x^2 + 0,000337001x - 0,0504332$.
- i5-1095G1 1.00 GHz $\rightarrow T_2(n) = 5,12922 \cdot 10^{-9}x^3 - 1,11315 \cdot 10^{-6}x^2 + 0,00083571x - 0,134397$.
- Ordenador Moya $\rightarrow T_3(n) = 6,77297 \cdot 10^{-9}x^3 + 5,13099 \cdot 10^{-7}x^2 - 0,000427834x + 0,0714028$.

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

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

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 algoritmo 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 algoritmos. 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:

Intel Core i7-6700 3.40 GHz		i5-1095G1 1.00 GHz		Ordenador Moya	
Elementos (n)	Tiempo (s)	Elementos (n)	Tiempo (s)	Elementos (n)	Tiempo (s)
8	0.00000136207	8	0.0000037376	8	0.0000017978
9	0.00000267907	9	0.00000737613	9	0.00000348253
10	0.00000528653	10	0.0000145867	10	0.00000737093
11	0.0000112702	11	0.0000283526	11	0.0000137999
12	0.0000234959	12	0.0000460821	12	0.0000274451
13	0.0000457819	13	0.0000722887	13	0.0000548052
14	0.0000904406	14	0.000106264	14	0.000110116
15	0.000198225	15	0.000213395	15	0.000198426
16	0.000439214	16	0.000353459	16	0.000427075
17	0.00088158	17	0.000717674	17	0.000796963
18	0.00145113	18	0.00142487	18	0.00159355
19	0.00253865	19	0.00278949	19	0.00321857
20	0.00499491	20	0.00534407	20	0.00633508
21	0.0100156	21	0.0101673	21	0.012697
22	0.0209075	22	0.0238254	22	0.0253476
23	0.0402523	23	0.0555082	23	0.0506946
24	0.0878626	24	0.112827	24	0.101314
25	0.171153	25	0.207041	25	0.202542
26	0.339115	26	0.344851	26	0.405264
27	0.633015	27	0.761311	27	0.809707
28	1.28649	28	1.41561	28	1.6195
29	2.60592	29	2.68719	29	3.23942
30	5.05092	30	5.41493	30	6.47798
31	10.1126	31	9.82069	31	12.9623
32	20.301	32	20.2358		

Cuadro 4: 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.

INSERTAR GRÁFICA

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:

- i7-6700 3.40GHz $\rightarrow T_1(n) =$.
- i5-1095G1 1.00 GHz $\rightarrow T_2(n) =$.
- Ordenador Moya $\rightarrow T_3(n) =$.

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

3. Casos especiales

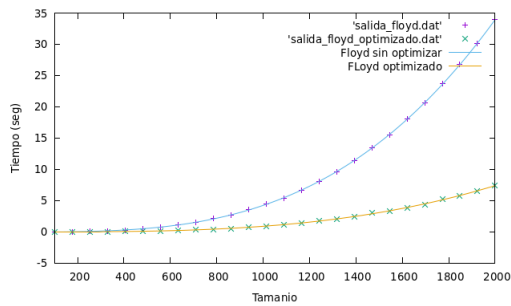
Además del análisis mostrado de los seis algoritmos anteriores, también se ha realizado un análisis de algunos de ellos bajo condiciones distintas, para mostrar así además una experiencia más amplia y diversa y conseguir un mejor entendimiento de los algoritmos trabajados.

3.1. Floyd optimizado

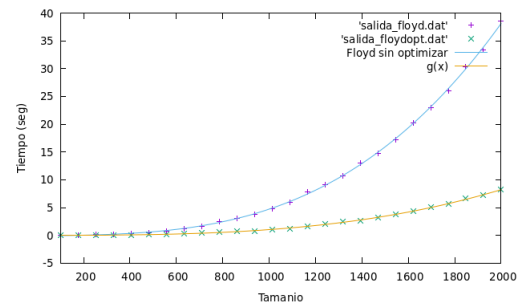
En este caso, queríamos mostrar la diferencia que obtenemos cuando realizamos la compilación de nuestro código bajo ciertas condiciones que pueden modificar "la pureza del mismo".

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.

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.



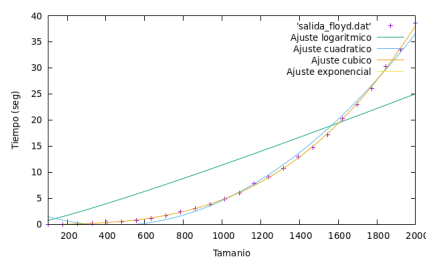
(a) Intel i7-6700 3.40GHz



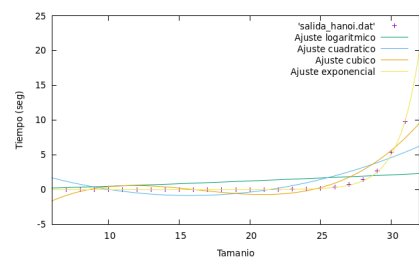
(b) i5-1095G1 1.00 GHz

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) i7-6700 3.40GHz



(b) i5-1095G1 1.00 GHz

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

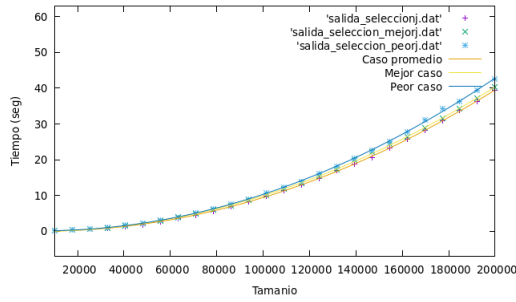
Otra de las tareas a realizar en esta práctica ha sido medir los tiempos para el mejor caso y peor caso de los algoritmos de inserción y selección y compararlos con el promedio, el cual ya hemos analizado anteriormente. El peor caso es el de un vector ordenado a la inversa, para lo cual hemos introducido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

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

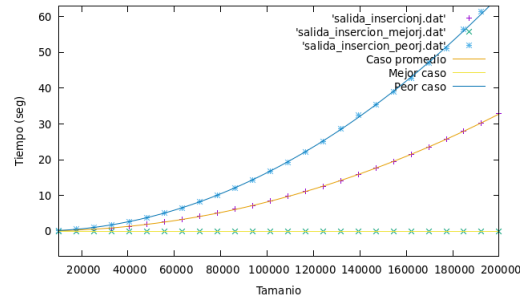
Y para el mejor caso hemos introducido un bucle que crea un vector ya ordenado:

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

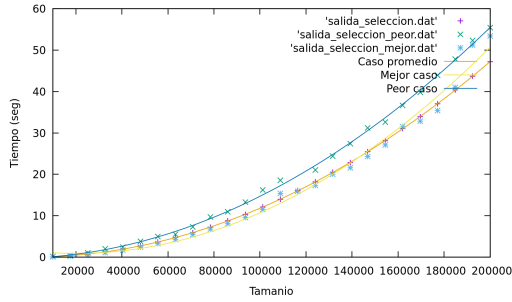
Y hemos generado las siguientes gráficas comparativas para cada integrante, las cuales son muy parecidas:



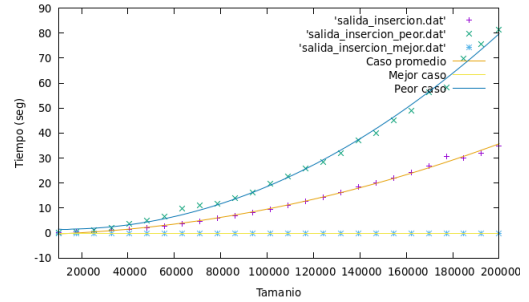
(a) Selección Javier



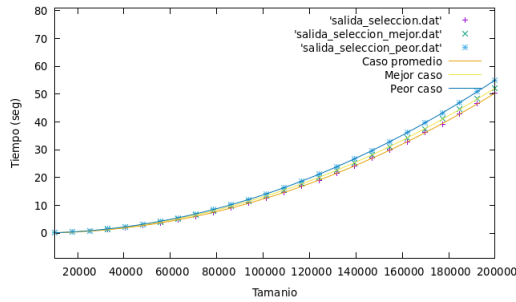
(b) Inserción Javier



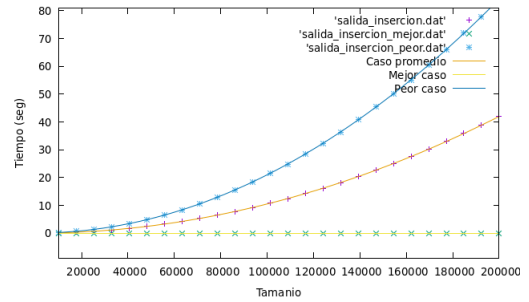
(a) Selección José Alberto



(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.