



# UNIVERSIDAD DE GRANADA

DOBLE GRADO EN INGENIERÍA INFORMÁTICA Y MATEMÁTICAS

ALGORÍTMICA

---

## Práctica 1: Análisis de Eficiencia de Algoritmos

---

Shao Jie Hu Chen  
Mario Megías Mateo  
Jesús Samuel García Carballo

*Grupo Rojo*

1 de abril de 2022

## Resumen

Esto es una prueba

# Índice

<b>1. Introducción</b>	<b>1</b>
1.1. Estructura de la memoria . . . . .	1
1.2. Objetivos . . . . .	1
1.3. Equipo empleado . . . . .	1
<b>2. Eficiencia teórica</b>	<b>2</b>
2.1. Algoritmo de inserción . . . . .	2
2.2. Algoritmo de selección . . . . .	3
2.3. Algoritmo HeapSort . . . . .	3
2.4. Algoritmo QuickSort . . . . .	4
2.5. Algoritmo de Floyd . . . . .	6
2.6. Algoritmo Hanoi . . . . .	7
<b>3. Eficiencia empírica</b>	<b>7</b>
3.1. Algoritmo de inserción . . . . .	7
3.2. Algoritmo de selección . . . . .	7
3.3. Algoritmo HeapSort . . . . .	7
3.4. Algoritmo QuickSort . . . . .	7
3.5. Algoritmo de Floyd . . . . .	8
<b>4. Eficiencia híbrida</b>	<b>8</b>
4.1. Algoritmo de inserción . . . . .	9
4.2. Algoritmo de selección . . . . .	9
4.3. Algoritmo HeapSort . . . . .	9
4.4. Algoritmo QuickSort . . . . .	9
4.5. Algoritmo de Floyd . . . . .	9

# 1. Introducción

En esta primera práctica de la asignatura Algorítmica vamos a realizar un análisis teórico y empírico de diversos algoritmos vistos en la asignatura. En particular, dicho análisis se hará de los siguientes:

- **Algoritmos de ordenación:** Inserción, selección, quicksort y mergesort.
- **Algoritmo de análisis de caminos mínimos sobre grafos:** Algoritmo de Floyd.
- **Hanoi.** Resolución del juego de las torres de Hanoi.

## 1.1. Estructura de la memoria

Esta memoria presenta la estructura típica de un trabajo académico. En primer lugar, presentamos brevemente el contexto en el que nos manejamos en esta práctica. Posteriormente, analizaremos teóricamente cada uno de los algoritmos propuestos para, seguidamente, comprobar mediante un análisis empírico las soluciones obtenidas. Asimismo, se ha realizado un análisis híbrido contrastando los resultados teóricos con los experimentales. Este análisis se ha realizado para cada algoritmo, extrayéndose información relevante asociada a ellas.

Además, para cada uno de los algoritmos considerados, se ha elaborado conjuntamente entre los miembros del grupo tablas asociadas a cada una de ellas, que se irán presentando a lo largo de esta memoria. Finalmente, se escribirán unas líneas asociadas a las conclusiones obtenidas tras el desarrollo de esta práctica, así como los resultados más relevantes obtenidos.

## 1.2. Objetivos

Entre los objetivos que vamos a realizar asociadas a esta práctica, caben destacar los siguientes:

- **Estudio teórico, empírico e híbrido y comparación** de los algoritmos de ordenación más empleados, verificando los resultados teóricos.
- **Estudio teórico, empírico e híbrido** de algoritmos de alta complejidad, poniendo especial énfasis en su **viabilidad** en diferentes equipos.
- Estudio del **aumento de eficiencia** de un mismo algoritmo para diferentes **tipos de optimización** del compilador.
- Determinación del algoritmo **más adecuado** para cada situación en función del estado de los datos.

## 1.3. Equipo empleado

Para el desarrollo de esta práctica, se han empleado diferentes equipos, cuyas respectivas especificaciones técnicas se especifican a continuación:

- **ASUS**
  - **Modelo:** ZenBook 15 UX534F
  - **Procesador:** Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz5
  - **Memoria Ram:** 16 GB DDR4 @ 2.133 MHz.
  - **Sistema Operativo** Ubuntu 20.04.2 LTS

## ■ LENOVO

- **Modelo:** YOGA 530-14IKB
- **Procesador:** Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz
- **Memoria RAM:** 8 GB DDR4
- **Sistema Operativo:** Ubuntu 20.04.4 LTS

## ■ HP

- **Modelo:** HP Pavilion Gaming Laptop 15-dk0xxx
- **Procesador:** Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz
- **Memoria RAM:** 32 GB DDR4
- **Sistema Operativo:** Ubuntu 20.04.4 LTS

Debido a las limitaciones técnicas de estos equipos, para el cálculo de la eficiencia empírica se ha optado por medir tiempos de ejecución en tiempos y **tamaños razonables** para obtener datos relevantes para el comportamiento tanto a pequeñas escalas como a nivel asintótico de ellas.

## 2. Eficiencia teórica

Para el análisis de los siguientes algoritmos se ha empleado la notación  $O(\cdot)$ .

### 2.1. Algoritmo de inserción

El código de este algoritmo lo podemos encontrar a continuación.

```
1 static void insercion_lims(int T[], int inicial, int final)
2 {
3     int i, j;
4     int aux;
5     for (i = inicial + 1; i < final; i++) {
6         j = i;
7         while ((T[j] < T[j-1]) && (j > 0)) {
8             aux = T[j];
9             T[j] = T[j-1];
10            T[j-1] = aux;
11            j--;
12        };
13    };
14 }
```

Listing 1: Implementación en C++ del algoritmo de inserción.

El análisis de eficiencia se ha de hacer sobre el número de elementos del vector  $n$ .

Para el análisis asintótico, calculamos el número de operaciones elementales asociadas al código. Las tres primeras líneas tienen complejidad  $O(1)$ . Para el ciclo for, démonos cuenta de que itera sobre el número de componentes del vector. Por su parte, dentro del for existe un while que realiza un conjunto de operaciones de orden  $O(1)$  tantas veces como se indica en el iterador. Por tanto, llamando  $T(n)$  al números de operaciones asociadas al algoritmo, tenemos que:

$$T(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{i-1} k = \sum_{i=0}^{n-1} ki = k \frac{n(n-1)}{2}$$

Por tanto, deducimos que  $T(n) \in O(n^2)$ .

## 2.2. Algoritmo de selección

```

1 static void seleccion_lims(int T[], int inicial, int final)
2 {
3     int i, j, indice_menor;
4     int menor, aux;
5     for (i = inicial; i < final - 1; i++) {
6         indice_menor = i;
7         menor = T[i];
8         for (j = i; j < final; j++) {
9             if (T[j] < menor) {
10                indice_menor = j;
11                menor = T[j];
12            }
13            aux = T[i];
14            T[i] = T[indice_menor];
15            T[indice_menor] = aux;
16        };
17    }

```

Listing 2: Implementación en C++ del algoritmo de selección.

La estructura principal del código anterior consta de dos bucles for anidados. Las primeras sentencias de declaración antes del inicio del primer bucle for no repercuten en el análisis. Procedamos a analizar la estructura repetitiva: el primer for recorre el vector desde su inicio hasta la penúltima componente del mismo incluida. El segundo for comienza en la posición dada por el índice del primero hasta la última componente del vector incluida. Dentro del bucle interno tenemos una operación elemental correspondiente a la comparación realizada en el if, que es  $O(1)$ . Dentro del bucle externo excluyendo el código del bucle interno nos encontramos con dos sentencias de asignación que no se tendrán en cuenta para el análisis. De igual modo, fuera del bucle externo al final del código nos encontramos con otras sentencias de asignación que también serán despreciadas. También tendremos en cuenta las operaciones elementales aritméticas correspondientes a la gestión de los bucles, que son  $O(1)$ . En conclusion, acotando las expresiones  $O(1)$  por  $k$  tenemos que:

$$\begin{aligned}
 T(n) &= \sum_{i=0}^{n-2} \sum_{j=i}^{n-1} k = \sum_{i=0}^{n-2} k(n-1-i+1) = \sum_{i=0}^{n-2} k(n-i) = kn \sum_{i=0}^{n-2} 1 - k \sum_{i=0}^{n-2} i \\
 &= kn(n-2+1) - k \frac{(n-2)(n-1)}{2} = kn^2 + kn - \frac{kn^2 - 3kn + 2k}{2}
 \end{aligned}$$

Deducimos, por tanto, que  $T(n) \in O(n^2)$ .

## 2.3. Algoritmo HeapSort

```

1 static void heapsort(int T[], int num_elem)
2 {

```

```

3  int i;
4  for (i = num_elem/2; i >= 0; i--)
5      reajustar(T, num_elem, i);
6  for (i = num_elem - 1; i >= 1; i--)
7      {
8          int aux = T[0];
9          T[0] = T[i];
10         T[i] = aux;
11         reajustar(T, i, 0);
12     }
13 }

```

Listing 3: Implementación en C++ del algoritmo heapsort.

El heapsort es un algoritmo de ordenación por montículos que consta de dos bucles for independientes, que llaman a una función llamada reajustar. En la primera línea de código podemos encontrar una declaración de variable cuya eficiencia es  $O(1)$  y luego los dos ciclos mencionados, que interan sobre el número de elementos del vector que denotaremos por  $n$ , luego realizaremos la eficiencia en torno a este dato. Ambos ciclos son independientes, así que vamos a analizar cada uno por separado y aplicando la regla del máximo nos quedaremos con el mayor.

La función reajustar trata de imitar el uso de un árbol APO mediante vectores y recorrerlo insertando elementos, luego como en un árbol APO la inserción se realiza con eficiencia  $O(\log_2(n))$ . Así que respecto a ambos ciclos tenemos lo siguiente:

#### ■ Primer ciclo

$$T(n) = \sum_{i=0}^{n/2+1} \log_2(k) = ((n/2 + 1) * (n/2 + 2))/2 * \log_2(k) \in O(n * \log(n))$$

#### ■ Segundo ciclo

$$T(n) = \sum_{i=1}^{n-1} \log_2(k) = (((n - 1) * n)/2) * \log_2(n). \in O(n * \log(n))$$

## 2.4. Algoritmo QuickSort

```

1  inline void quicksort(int T[], int num_elem)
2  {
3      quicksort_lims(T, 0, num_elem);
4  }
5
6  static void quicksort_lims(int T[], int inicial, int final)
7  {
8      int k;
9      if (final - inicial < UMBRAL_QS) {
10         insercion_lims(T, inicial, final);
11     } else {
12         dividir_qs(T, inicial, final, k);
13         quicksort_lims(T, inicial, k);
14         quicksort_lims(T, k + 1, final);
15     };
16 }
17
18

```

```

19 static void dividir_qs(int T[], int inicial, int final, int & pp)
20 {
21     int pivote, aux;
22     int k, l;
23
24     pivote = T[inicial];
25     k = inicial;
26     l = final;
27     do {
28         k++;
29     } while ((T[k] <= pivote) && (k < final-1));
30     do {
31         l--;
32     } while (T[l] > pivote);
33     while (k < l) {
34         aux = T[k];
35         T[k] = T[l];
36         T[l] = aux;
37         do k++; while (T[k] <= pivote);
38         do l--; while (T[l] > pivote);
39     };
40     aux = T[inicial];
41     T[inicial] = T[l];
42     T[l] = aux;
43     pp = l;
44 };

```

Listing 4: Implementación en C++ del algoritmo QuickSort.

En este apartado, consideramos  $n$  como el número de componentes del vector.

El código del algoritmo comienza en la función `quicksort.lims` en la línea 6. Tomaremos inicial como ñaa posición de comienzo del vector ( $inicial = 0$ ), y final como el siguiente al último elemento, que coincide con el número de componentes del vector ( $final = n$ ). Tenemos una estructura condicional que determina el comportamiento del algoritmo según si el tamaño del vector supera o no un cierto umbral, por lo que el tiempo de ejecución será una función definida a trozos. Si es menor que el umbral, ejecutamos la ordenación con el algoritmo de inserción previamente analizado, que es  $O(n^2)$ . De lo contrario, realizamos la ordenación según el propio algoritmo quicksort. Dicho algoritmo se basa en la técnica de Divide y Vencerás, implementándose de forma recursiva. En primer lugar, se realiza una llamada a la función `dividir` `dividir_qs`, cuya eficiencia procedemos a analizar. En las líneas 24-26 y 40-43 tenemos sentencias de asignación que serán despreciadas. Siguiendo código de forma secuencial, en las líneas 27 y 30 tenemos dos bucles `do-while`, si consideramos el peor caso para el primero (el pivote es mayor que el resto de elemento) tiene eficiencia  $O(n)$ , y el caso contrario al anterior sería el más desfavorable para el segundo (el pivote es menor que el resto de los elementos), luego tendríamos  $O(n)$ . Como el caso más favorable para uno (eficiencia  $O(1)$ ) es el más desfavorable para el otro, aplicando la regla del Máximo tenemos que la eficiencia de la parte del código sería  $O(n)$ .

Para la función `dividir_qs`, tenemos que su eficiencia es  $O(n)$ . En efecto, para cada bucle de la función tenemos que se procesa 1 (o una cantidad finita de veces) un elemento del vector.

Ahora bien, para la eficiencia del algoritmo, sea  $T(n)$  el número de operaciones elementales que realiza el algoritmo, y  $k_n$  el número de componentes del vector desde el inicio hasta la posición



para el pivote calculada por `dividir_qs`. Tenemos que:

$$T(n) = \begin{cases} T(k_n) + T(n - k_n) + n & \text{si } n \geq \text{UMBRAL} \\ n^2 & \text{si } n < \text{UMBRAL} \end{cases} \quad (1)$$

Supongamos que  $k_n = \frac{n}{2}$ . Resolvamos la recurrencia de (1):

$$T(n) = T(k_n) + T(n - k_n) + n \text{ si } n \geq \text{UMBRAL} \quad (2)$$

Realizamos el cambio de variable  $n = 2^m$  en la ecuación (2) obtenemos:

$$T(2^m) = 2T(2^{m-1}) + 2^m \text{ si } n \leq \log_2(\text{UMBRAL}) \quad (3)$$

$$T(2^m) - 2T(2^{m-1}) = 2^m \quad (4)$$

Renombramos la expresión anterior como  $T(2^m) = t_m$  obtenemos:

$$t_m - 2t_{m-1} = 2^m \quad (5)$$

Obtenemos la siguiente ecuación de recurrencia de la ecuación (4):

$$(x - 2)^2 = 0 \quad (6)$$

Luego la solución general de la recurrencia es:

$$t_m = c_1 2^m + c_2 m 2^m \quad (7)$$

Deshacemos el cambio de variable:

$$T(n) = c_1 n + c_2 n \log_2(n) \quad (8)$$

Por tanto  $T(n) \in O(n \log_2(n))$

## 2.5. Algoritmo de Floyd

A continuación se presenta el código de este algoritmo:

```

1 void Floyd(int **M, int dim)
2 {
3     for (int k = 0; k < dim; k++)
4         for (int i = 0; i < dim; i++)
5             for (int j = 0; j < dim; j++)
6                 {
7                     int sum = M[i][k] + M[k][j];
8                     M[i][j] = (M[i][j] > sum) ? sum : M[i][j];
9                 }
10 }
```

Listing 5: Implementación en C++ del algoritmo de Floyd.

En este caso, consideramos como  $n$  el número de filas/columnas de la matriz. Como los tres bucles iteran cada uno sobre  $n$ , siendo el número de iteraciones independientes entre sí y se realizan  $k$  operaciones en cada una de ellas, tenemos que llamando  $T(n)$  al número de operaciones para un tamaño  $n$ , se verifica:

$$T(n) = \sum_{i=1}^n \sum_{j=1}^n \sum_{k=1}^n k = kn^3$$

Deducimos fácilmente que  $T(n) \in O(n^3)$ .

## 2.6. Algoritmo Hanoi

Listing 6: Implementación en C++ del algoritmo de resolución de las torres de Hanoi.

## 3. Eficiencia empírica

Para esta práctica hemos ejecutado en cada uno de los equipos los diversos algoritmos presentados previamente. Las tablas obtenidas, así como las gráficas que representan estas tablas se representan a continuación.

### 3.1. Algoritmo de inserción

### 3.2. Algoritmo de selección

### 3.3. Algoritmo HeapSort

### 3.4. Algoritmo QuickSort

### 3.5. Algoritmo de Floyd

En la siguiente tabla se encuentran los datos obtenidos en este algoritmo por cada uno de los equipos en los que hemos corrido el algoritmo.

$N_{nod}$	$t_{ASUS}$	$t_{HP}$	$t_{LENOVO}$
5	0.09	0.1	0.13
35	4.635	5.045	3.795
65	28.105	36.27	22.685
95	98.975	59.425	76.2
125	206.65	291.02	171.845
155	250.735	340.205	294.25
185	368.585	490.165	466.895
215	572.04	588.29	727.49
245	846.54	873.175	1069.24
275	1193.13	1234.555	1530.57
305	1642.2	1666.67	2058.385
335	2164.105	2205.705	2755.755
365	2826.06	2846.01	3574.31
395	3528.345	3583.4	4484.645
425	4410.26	4465.635	5606.11
455	5383.555	5485.51	6820.05
485	6554.765	6483.905	8154.31
515	7855.67	7928.895	9760.62
545	9386.4	9268.545	11542.12
575	10957.08	10777.34	13740.255
605	12786.025	12425.0	15864.585
635	14618.06	14709.94	18181.38
665	16823.92	16768.84	21031.77
695	19147.07	18815.95	23953.145
725	21807.535	21948.21	27128.6

Cuadro 1: Tiempos de ejecución (en  $\mu s$ ) del algoritmo de Floyd

En concordancia con los resultados obtenidos en las anteriores tablas, el ordenador que ofrece peores tiempos es el LENOVO, siendo el de ASUS el que ofrece resultados con mayor rapidez.

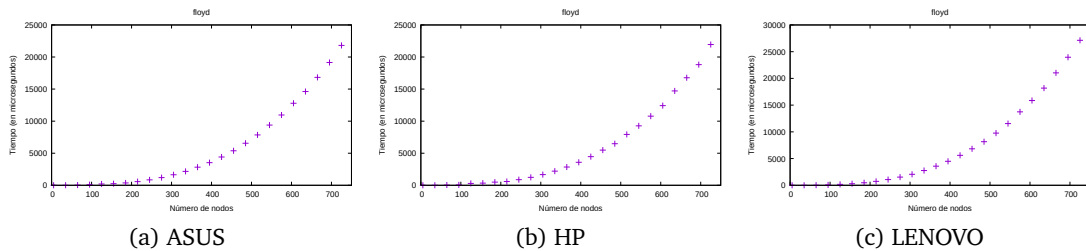


Figura 1: Representación gráfica de tiempos de ejecución del algoritmo Floyd.

## 4. Eficiencia híbrida

En esta sección trataremos los datos obtenidos de la sección anterior mediante técnicas estadísticas para comprobar la consistencia de los datos experimentales con los calculados teóricamente.

- 4.1. Algoritmo de inserción
- 4.2. Algoritmo de selección
- 4.3. Algoritmo HeapSort
- 4.4. Algoritmo QuickSort
- 4.5. Algoritmo de Floyd