



UNIVERSIDAD  
DE GRANADA

*Este documento está protegido por la Ley de Propiedad Intelectual ([Real Decreto Ley 1/1996 de 12 de abril](#)).  
Queda expresamente prohibido su uso o distribución sin autorización del autor.*

# Algorítmica

2º Grado en Ingeniería Informática

## Guión de prácticas

### Cálculo de la eficiencia de algoritmos

1. Objetivo.....	2
2. Cálculo de la eficiencia teórica.....	2
3. Cálculo de la eficiencia práctica.....	4
4. Entrega de la práctica.....	6

© Prof. Manuel Pegalajar Cuéllar  
Dpto. Ciencias de la Computación e I. A.  
Universidad de Granada



DECSAI

**Departamento de Ciencias de la  
Computación e Inteligencia Artificial**

# Cálculo de la eficiencia de algoritmos

## 1. Objetivo

El objetivo de la práctica consiste en que el alumno sea capaz de analizar y calcular la eficiencia teórica, práctica e híbrida de algoritmos, tanto iterativos como recursivos. Para ello, se expone un conjunto de problemas que deberán ser resueltos por el estudiante.

## 2. Cálculo de la eficiencia teórica

### 2.1. Algoritmos iterativos

1. Diseñe un algoritmo que, teniendo como entrada un vector de números enteros de tamaño  $N$ , compruebe si existen elementos repetidos en el vector y elimine todas las ocurrencias repetidas del mismo elemento. Como salida, se devolverá el mismo vector con todos los elementos sin repetir. No se requiere que el vector de salida tenga los elementos en el mismo orden que el array original. Por ejemplo, para la entrada  $V = \{5, 1, 9, 2, 2, 5, 1, 1, 7\}$ , de tamaño  $N=9$ , una posible salida sería  $V' = \{2, 9, 5, 7, 1\}$ , de tamaño  $N=5$ . Otra posible salida alternativa sería  $V' = \{1, 2, 5, 7, 9\}$ , también de tamaño  $N=5$ . El único requisito del vector de salida es que contenga los mismos elementos que el array de entrada, sin repetir.
2. Supongamos ahora que disponemos de un vector de entrada, también conteniendo  $N$  números enteros, que cumple con la precondition de que se encuentra ordenado. Diseñe un algoritmo lo más eficiente posible para eliminar los elementos repetidos del vector y devuelva, como salida, el mismo vector modificado con elementos sin repetir. Por ejemplo, para el vector de entrada  $V = \{1, 1, 1, 2, 2, 5, 5, 7, 9\}$  de tamaño  $N=9$ , una posible salida sería  $V' = \{1, 2, 5, 7, 9\}$  de tamaño 5.
3. Identifique de qué variable, o variables, depende el tamaño del problema en cada algoritmo diseñado.
4. Identifique cuáles serían los casos mejor y peor para cada algoritmo diseñado. Exponga un ejemplo para una instancia concreta de tamaño determinado para cada caso, y justifique por qué esa instancia hace que el algoritmo analizado ejecute el máximo número de operaciones (caso peor) o el mínimo (caso mejor).
5. Calcule los órdenes de eficiencia de ambos algoritmos en el caso peor y mejor.
6. Realice una experimentación con instancias aleatorias de vectores de entrada de diferentes tamaños de caso (1000, 2000, 3000, ..., 10000, 20000, 30000, ...), midiendo tamaños de caso y tiempos de ejecución para cada uno de ellos. Calcule la constante oculta de cada algoritmo y muestre en una gráfica el tiempo real de cada algoritmo junto con la gráfica del tiempo de ejecución teórico (híbrido) con la constante calculada. Justifique los resultados que se visualizan en la gráfica para cada algoritmo.

### 2.2. Algoritmos recursivos

Se debe realizar las siguientes tareas:

1. Calcule la ecuación en recurrencias, y el orden del algoritmo en el caso peor, para las funciones **Hanoi** y **HeapSort** siguientes.
2. Compare la eficiencia del algoritmo **HeapSort** con el algoritmo **MergeSort**, explicado en clase de teoría, de forma teórica y de forma híbrida para el caso peor. Para ello, realice una experimentación con instancias aleatorias de vectores de entrada de diferentes tamaños de caso (1000, 2000, 3000, ..., 10000, 20000, 30000, ...), midiendo tamaños de caso y tiempos de ejecución para cada uno de ellos. Calcule la constante oculta de cada algoritmo y muestre en una gráfica el tiempo real de cada algoritmo junto con la gráfica del tiempo de ejecución teórico (híbrido) con la constante calculada. Justifique los resultados que se visualizan en la gráfica para cada algoritmo e indique porqué los algoritmos se comportan igual o, si se da el caso, uno es mejor que otro.

```

1  /**
2   Se trata del problema clásico de las torres de Hanoi.
3   Se tienen 3 barras, y hay que mover M anillos de la primera barra
4   a la segunda. Solo se puede mover un anillo en cada movimiento,
5   y ningún anillo de tamaño mayor puede ponerse sobre otro de tamaño
6   menor.
7   Los valores de "i" y "j" sólo pueden tomar los valores {1, 2, 3}
8
9   Si M=3, la llamada sería hanoi(3, 1, 2)
10
11  */
12  void hanoi (int M, int i, int j)
13  {
14      if (M > 0)
15      {
16          hanoi(M-1, i, 6-i-j);
17          cout << i << " -> " << j << endl;
18          hanoi (M-1, 6-i-j, j);
19      }
20  }

```

```

void HeapSort(int *v, int n){
    double *apo=new double [n];
    int tamapo=0;

    for (int i=0; i<n; i++){
        apo[tamapo]=v[i];
        tamapo++;
        insertarEnPos(apo,tamapo);
    }
    for (int i=0; i<n; i++) {
        v[i]=apo[0];
        tamapo--;
        apo[0]=apo[tamapo];
        reestructurarRaiz(apo, 0, tamapo);
    }
    delete [] apo;
}

```

```

void insertarEnPos(double *apo, int pos){
    int idx = pos-1;
    int padre;
    if (idx > 0) {
        if (idx%2==0) {
            padre=(idx-2)/2;
        }else{
            padre=(idx-1)/2;
        }

        if (apo[padre] > apo[idx]) {
            double tmp=apo[idx];
            apo[idx]=apo[padre];
            apo[padre]=tmp;
            insertarEnPos(apo, padre+1);
        }
    }
}

```

```
void reestructurarRaiz(double *apo, int pos, int tamapo){
    int minhijo;
    if (2*pos+1< tamapo) {
        minhijo=2*pos+1;
        if ((minhijo+1 < tamapo) && (apo[minhijo]>apo[minhijo+1])) minhijo++;
        if (apo[pos]>apo[minhijo]) {
            double tmp = apo[pos];
            apo[pos]=apo[minhijo];
            apo[minhijo]=tmp;
            reestructurarRaiz(apo, minhijo, tamapo);
        }
    }
}
```

### 3. Cálculo de la eficiencia práctica

Para medir la eficiencia práctica de los algoritmos, tenemos que ejecutar el mismo algoritmo para diferentes tamaños de casos, y calcular su tiempo de ejecución. Para ello, y dado que los ordenadores modernos son muy veloces, haremos uso de la biblioteca **chrono** incluida en el estándar **C++11**. No olvide compilar cada programa con el compilador **g++** incluyendo este estándar antes de compilar cada programa.

El tiempo de ejecución lo mediremos como la diferencia entre el instante de tiempo justo anterior al inicio del algoritmo, y el instante justamente posterior. Para ello, necesitaremos declarar las siguientes variables en nuestro programa:

```
#include<chrono>
```

```
...
```

```
chrono::time_point<std::chrono::high_resolution_clock> t0, tf; // Para medir el tiempo de ejecución
```

Los instantes de tiempo, y la duración, los obtendremos de la siguiente forma:

```
// Comenzamos a medir el tiempo
t0= std::chrono::high_resolution_clock::now();
```

```
// Aquí vendría la ejecución del algoritmo
```

```
// Terminamos de medir el tiempo
tf= std::chrono::high_resolution_clock::now();
```

```
// Medimos la duración, en MICROSEGUNDOS
unsigned long duration;
duration= std::chrono::duration_cast<std::chrono::microseconds>(tf - t0).count();
```

Ejecute el código del ejemplo **Practica1.cpp** proporcionado por el profesor para medir el tiempo de ejecución del algoritmo de ordenación por Burbuja con diversos tiempos de ejecución.

Asumiendo que hemos calculado el orden  $O(f(n))$  de cada algoritmo propuesto, este orden  $O(f(n))$  quiere decir que existe una constante **K**, para cada algoritmo, tal que el tiempo **T(n)** de ejecución del mismo para un tamaño de caso **n** es:

$$T(n) \leq K \cdot f(n)$$

El cálculo de la constante **K** se calcula despejando e igualando la fórmula anterior:

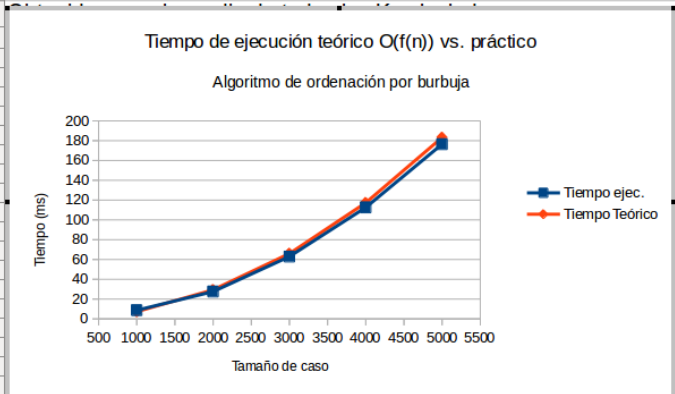
$$K = T(n)/f(n)$$

Este valor de K se calculará para todas las ejecuciones del mismo algoritmo, produciendo valores aproximados para K. Por tanto, calcularemos el valor final de K como la media de todos estos valores. La siguiente gráfica muestra un ejemplo del cálculo de este valor en LibreOffice Calc, para los resultados del algoritmo de ordenación por burbuja ejecutado para los casos 1000, 2000, 3000, 4000, 5000, utilizando una semilla de inicialización de números aleatorios igual a 123456.

	B	C	D
1	Tiempo ejec. K		
2	8,663	0,000008663	Obtenido como $T(1000)/f(1000)$ para un tiempo de ejecución $T(n)$ y orden $O(f(n))$
3	27,584	0,000006896	Obtenido como $T(2000)/f(2000)$ para un tiempo de ejecución $T(n)$ y orden $O(f(n))$
4	62,921	6,99122222222222E-006	...
5	112,555	7,0346875E-006	
6	176,439	7,05756E-006	Obtenido como $T(5000)/f(5000)$ para un tiempo de ejecución $T(n)$ y orden $O(f(n))$
7	K media=	7,32849394444445E-006	Obtenido como la media de todas las K calculadas
8			
9			
10			

**NOTA: Para importar los datos de los ficheros de salida en LibreOffice Calc o Excel, es necesario reemplazar todos los puntos (.) por comas (,) en el fichero de datos.**

Asumiendo que hemos calculado el orden de eficiencia del algoritmo de ordenación por burbuja como  $O(n^2)$ , y que el valor de la constante oculta en promedio vale  $K=7.33 \cdot 10^{-6}$ , a continuación, comprobaremos experimentalmente que el orden  $O(f(n))$  calculado con una constante oculta superior siempre será mayor que el tiempo de ejecución real del algoritmo, por lo que hemos conseguido acotar el tiempo de ejecución del mismo y estimar, en el futuro, cuánto tardará en ejecutarse para otros tamaños de casos. Se muestra la siguiente gráfica generada por LibreOffice Calc para el tiempo de ejecución real y teórico con la constante oculta calculada:

D		E
1		<b>Tiempo Teórico</b>
2	Obtenido como $T(1000)/f(1000)$ para un tiempo de ejecución $T(n)$ y orden $O(f(n))$	7,3284939444
3	Obtenido como $T(2000)/f(2000)$ para un tiempo de ejecución $T(n)$ y orden $O(f(n))$	29,3139757778
4	...	65,9564455
5		117,2559031111
6	Obtenido como $T(5000)/f(5000)$ para un tiempo de ejecución $T(n)$ y orden $O(f(n))$	183,2123486111
7	 <p>Tiempo de ejecución teórico <math>O(f(n))</math> vs. práctico</p> <p>Algoritmo de ordenación por burbuja</p>	
8		
9		
10		
11		
12		
13		
14		
15		
16		
17		
18		
19		
20		
21		
22		
23		
24		
25		
26		
27		

## 4. Entrega de la práctica

Se deberá entregar un documento por grupos de hasta 3 personas máximo, que contenga la solución a los problemas planteados, dividida en apartados. La memoria deberá contener el análisis teórico de la eficiencia de cada algoritmo requerido, y la eficiencia práctica e híbrida calculadas en cada caso requerido. Se deberá apoyar en gráficas de tiempos de ejecución y/o tiempos teóricos que avalen el análisis de resultados realizado. El código fuente se entregará en ficheros separados, y una sección de la memoria deberá indicar cómo ejecutar cada fichero para obtener los resultados que se muestran en la memoria.

La práctica deberá ser entregada por PRADO, en la fecha y hora límite explicada en clase por el profesor. No se aceptarán, bajo ningún concepto, prácticas entregadas con posterioridad a la fecha límite indicada. La entrega de PRADO permanecerá abierta con, al menos, una semana de antelación antes de la fecha límite, por lo que todo alumno tendrá tiempo suficiente para entregarla.

La práctica contribuirá con 2 puntos sobre 10, ponderado al total de la puntuación de prácticas expuesto en la guía docente de la asignatura.

El profesor, en clase de prácticas, podrá realizar auditorías de las prácticas a discreción, con el fin de asegurar de que los estudiantes alcanzan las competencias deseadas. Por este motivo, una vez finalizada la entrega de prácticas por PRADO, es recomendable repasar los ejercicios entregados para poder responder a las preguntas del profesor, llegado el caso de su defensa. La no superación de la defensa de prácticas supondrá una calificación de 0 en esta práctica. La superación de la defensa supondrá mantener la calificación obtenida.