# Práctica 1.1: Complejidad empírica

Algoritmia y optimización

Curso 2025–26

## Índice

# 1. Introducción

En el Tema 2 de teoría denominado "Análisis de algoritmos", estudiamos la importancia de analizar la complejidad de un algoritmo, es decir, una medida de su coste. Este estudio es un paso crítico para entender cuántos recursos consume un algoritmo. Encontramos dos tipos de análisis de la complejidad de un algoritmo: (i) análisis empírico, y (ii) análisis teórico.

En esta práctica 1.1 profundizaremos en el **Análisis empírico**, el cual consiste en medir directamente la cantidad de recursos empleados por un algoritmo dados distintos valores de entrada. Uno de estos recursos podría ser el tiempo de ejecución.

# 2. Objetivos

- Comprender la complejidad empírica.
- Familiarizarse con el entorno explicado en la Práctica 0.
- Aprender a analizar empíricamente un algoritmo dado.
- Comparar los resultados entre algoritmos.

# 3. Desarrollo

Para esta práctica vamos a utilizar algoritmos de ordenación. Se verán más en profundidad en el Tema 3 de teoría ("Divide y vencerás"). Sin embargo, resultan adecuados para el objetivo de esta práctica.

Los algoritmos de ordenación consisten en, como su propio nombre indica, ordenar los elementos de una lista siguiendo algún criterio consistente (normalmente orden numérico o lexicográfico).[1] Desde los principios de la computación, los problemas de ordenación han sido siempre de gran interés entre los investigadores ya que es clave conseguir simplicidad y eficiencia.

Para esta práctica, debéis implementar los pseudocódigos proporcionados y analizar su complejidad empírica variando los tamaños y el contenido de la entrada (probando solo con arrays numéricos), midiendo el tiempo de ejecución de los distintos enfoques y mostrando los resultados obtenidos con Matplotlib.

Además, si quieres profundizar un poco más, en el material de teoría habrás visto que, además del análisis empírico, existe el análisis teórico (que pondremos en práctica en las siguientes prácticas). El análisis teórico no requiere implementar el algoritmo; sin embargo, puedes aproximarlo de manera empírica contabilizando los pasos del programa.

[1] El orden lexicográfico es comúnmente conocido como orden alfabético o el orden que se sigue en los diccionarios.

Para calcular esos pasos durante la ejecución del código, resulta conveniente utilizar una variable global que se incremente en una unidad por cada operación elemental realizada. En las diapositivas del tema 2 encontrarás una lista de ejemplos de estas operaciones.

En la práctica, este cálculo empírico basado en los pasos no suele utilizarse, ya que requiere modificar la implementación para obtenerlo. No obstante, puede ser interesante, a modo de ejercicio, compararlo con el tiempo de ejecución del algoritmo y estudiar cómo varía al incrementar la talla del problema. Ten en cuenta que, al añadir instrucciones adicionales al algoritmo, el tiempo de ejecución puede verse afectado; por ello, lo correcto sería tomar la medida del tiempo antes de añadir las instrucciones para contar los pasos.

Es importante destacar que nunca deben compararse directamente el tiempo de ejecución y los pasos de programa, ya que el primero puede variar enormemente en magnitud (desde microsegundos hasta segundos), mientras que el segundo emplea unidades que no corresponden a una medida física. Si tienes dudas sobre esta aclaración, intenta representar ambas complejidades en una misma gráfica: observarás de inmediato cómo la complejidad empírica basada en el tiempo aparece casi como una recta horizontal, mientras que la calculada a partir de los pasos de programa no lo hace.

## 4. Algoritmos

A continuación se presentan dos algoritmos de ordenación. Mide sus tiempo dependiendo de la entrada. Se aconseja, para un mismo tamaño de entrada, realizar diversas ejecuciones con contenidos diferentes y calcular el tiempo en promedio.

El primer algoritmo sería:

```
función ordenacion_uno(arr):
    n := |arr|

    para i desde 0 hasta n:
        para j desde 0 hasta n - i - 1:
            si arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]

    devuelve arr
```

El siguiente algoritmo se presenta a continuación:

```
función ordenacion_dos(arr):
    si |arr| <= 1:       # listas vacías o de 1 elemento
        devuelve arr
```

```
    pivote = arr[0]      # Se elige el primer elemento como pivote

    # Se crean dos listas auxiliares
    izq = [ ]            # elementos <= pivote
    der = [ ]            # elementos > pivote

    para i desde 1 hasta |arr|-1:
        si arr[i] <= pivote:
            añadir arr[i] a izq
        si arr[i] > pivote:
            añadir arr[i] a der

    # Ordena recursivamente izquierda y derecha
    devuelve ord_dos(izq) + [pivote] + ord_dos(der)
```

Puedes medir los tiempos de ejecución con la librería `time` de Python. Por ejemplo:

```
import time
...
inicio = time.time()
ejecutar_algoritmo()
fin = time.time()
print("Resultado:", resultado)
print("Tiempo transcurrido: ", fin - inicio, " segundos")
```

Para obtener medidas más precisas, puedes usar `time.perf_counter()` en vez de `time.time()`. Puedes multiplicar el resultado de la resta por 1000 para obtenerlo como milisegundos. Tienes más información en la documentación de la librería time.

Además, para generar arrays aleatorios se recomienda emplear `numpy.random`. Por ejemplo, puedes usar la función randint().