Práctica 1.1: Complejidad empírica

Algoritmia y optimización

Curso 2024–25

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 sólo con arrays numéricos), midiendo el tiempo de ejecución de los distintos enfoques y mostrando los resultados obtenidos con Matplotlib.

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

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 hasta |n|:
        para j 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 ord_dos(arr):
    si |arr| <= 1:
        devuelve arr
    entonces:
        pivote = arr[0]
        para x hasta arr desde arr[1]:
            si x <= pivote:
                añadir elemento x en izq[]
            si x > pivote:
                añadir elemento x en der[]

        devuelve ord_dos(izq) + [pivote] + ord_dos(der)
```

Puedes medir los tiempos de ejecución con la librería `time` de Python. Además, para generar arrays aleatorios se recomienda emplear `numpy.random`.