Práctica 6: Ordenamiento

Ejercitación básica

Ejercicio 1. Dado un arreglo de enteros A, diseñar un algoritmo que determine si A contiene elementos duplicados. El algoritmo debe ser $O(n \log n)$ en peor caso. Está permitido reordenar el arreglo A pero se pide no usar estructuras de datos externas como AVLs.

Implementar el algoritmo en Python. Se puede usar la función `sort` de Python y asumir que su complejidad es $O(n \log n)$ en peor caso.

Ejercicio 2. Sea A un arreglo de n cadenas de texto de longitud a lo sumo k. Se quiere generar un arreglo B en el que aparezcan todas las cadenas de A, de tal modo que primero aparezcan todas las cadenas de longitud 1, después todas las de longitud 2, y así sucesivamente hasta las de longitud k.

a) Diseñar un algoritmo de ordenamiento *estable*. Es decir, el orden que tienen las cadenas de igual longitud en la salida B debe respetar el orden que originalmente tenían en A. Por ejemplo:

$$\begin{aligned} A &= [\mathit{hola}, \mathit{c}, \mathit{sí}, \mathit{a}, \mathit{no}, \mathit{b}, \mathit{chau}, \mathit{z}] \\ B &= [\mathit{c}, \mathit{a}, \mathit{b}, \mathit{z}, \mathit{sí}, \mathit{no}, \mathit{hola}, \mathit{chau}] \end{aligned}$$

La complejidad debe ser $O(n + k)$ en peor caso. *Sugerencia:* usar una variante de bucket sort.

b) Implementar el algoritmo en Python.

Ejercicio 3.

a) Sea A un arreglo de n números enteros. Sea d la diferencia entre el máximo elemento y el mínimo elemento de A. Diseñar un algoritmo para ordenar A en tiempo $O(n + d)$ en peor caso.

b) Sea A un arreglo de n números enteros comprendidos entre 0 y $n - 1$. Diseñar un algoritmo para ordenar A en tiempo $O(n)$.

c) Sean k y $n \neq 0$ números enteros. El algoritmo de división (de la escuela primaria) garantiza que k se puede escribir de manera única como $q\,n + r$ donde q es el cociente de k en la división por n, y r es el resto de k en la división por n. Recordemos que el resto cumple que $0 \leq r < n$. Además, si k está en el rango $0 \leq k < n^2$, se puede observar que el cociente también está en dicho rango, es decir, $0 \leq q < n$.

Sea A un arreglo de n números enteros comprendidos entre 0 y $n^2 - 1$. Usando las ideas de arriba, diseñar un algoritmo para ordenar A en tiempo $O(n)$. *Sugerencia:* usar una variante de radix sort.

Ejercitación adicional

Ejercicio 4. Hay n elefantes y m telarañas. Los pesos de los elefantes están dados por un arreglo $[E[0], \ldots, E[n-1]]$. Cada telaraña tiene una capacidad máxima de peso que resiste. Las capacidades de las telarañas están dadas por un arreglo $[T[0], \ldots, T[m-1]]$. Queremos saber si es posible ubicar cada elefante sobre una telaraña, con las siguientes restricciones:

- El peso de cada elefante no debe superar la capacidad de la telaraña sobre la que está ubicado.
- Cada elefante se debe ubicar sobre una telaraña distinta. Es decir, no puede haber un mismo elefante sobre dos telarañas, ni dos elefantes sobre una misma telaraña.
- Todo elefante debe estar ubicado sobre alguna telaraña (no pueden faltar telarañas) pero puede haber telarañas sin elefantes (pueden sobrar telarañas).

Diseñar un algoritmo que dados los arreglos E y T determine si es posible ubicar a los elefantes sobre las telarañas con las restricciones descriptas arriba. La complejidad debe ser $O(m \log m)$.

Ejercicio 5. Sea A un arreglo de cadenas de texto. Sabemos que cada cadena de texto tiene una longitud de a lo sumo k caracteres. Cada caracter está codificado como un número entre 0 y 255 usando la codificación ASCII (por ejemplo, la letra '`A`' se codifica como el número 65). Diseñar un algoritmo para ordenar A en tiempo $O(kn)$ en peor caso. *Sugerencia:* usar una variante de radix sort.

Implementar el algoritmo en Python.

Ejercicio 6. Dado un arreglo de enteros A, decimos que es *sin agujeros* si es una permutación de alguna secuencia de números consecutivos. Por ejemplo, $[3, 5, 6, 4, 2]$ es sin agujeros porque es una permutación de $[2, 3, 4, 5, 6]$. En cambio $[3, 5, 6, 2]$ tiene agujeros (debido a la ausencia del 4).

Recibimos como entrada n arreglos $A_1, \ldots, A_n$ **sin agujeros**, cada uno de los cuales tiene n elementos. Sabemos además, como precondición, que los arreglos no tienen elementos en común. Queremos generar un único arreglo B de tamaño n^2 que contenga a todos los elementos en orden.

a) Se propone el siguiente algoritmo: juntar todos los elementos de $A_1, \ldots, A_n$ en un nuevo arreglo B (sin ningún orden en particular) y a continuación ordenar B usando alguno de los algoritmos de ordenamiento ya conocidos. ¿Cuál sería la complejidad temporal de este algoritmo?

b) Diseñar un algoritmo con tiempo de ejecución $O(n^2)$ en peor caso.

Ejercicio 7. Suponemos que identificamos personas de manera única a través de su DNI. Se tiene un arreglo $\mathcal{P}$ cuyos elementos son pares (x, y) que representan que x es hijo de y, donde y puede ser el padre o la madre de x. Se tiene otro arreglo $\mathcal{F}$ cuyos elementos son pares (x, a) que representan que x nació en el año a. Diseñar un algoritmo para generar un arreglo $\mathcal{R}$ cuyos elementos son pares (x, a) que representan que x es hijo de alguien que nació en el año a. Por ejemplo, si la entrada está dada por los arreglos $\mathcal{P}$ y $\mathcal{F}$ de abajo, la salida debe ser el arreglo $\mathcal{R}$ de la derecha:

$$\mathcal{P} = \left(\begin{array}{r|r} \text{dni} & \text{dni padre} \\ \hline 3 & 6 \\ 2 & 5 \\ 2 & 4 \\ 1 & 3 \\ 1 & 2 \end{array}\right) \qquad \mathcal{F} = \left(\begin{array}{r|r} \text{dni} & \text{año nac.} \\ \hline 2 & 1988 \\ 5 & 1961 \\ 1 & 2013 \\ 6 & 1957 \\ 3 & 1990 \\ 4 & 1966 \end{array}\right) \qquad \mathcal{R} = \left(\begin{array}{r|r} \text{dni} & \text{año nac. padre} \\ \hline 1 & 1990 \\ 1 & 1988 \\ 2 & 1961 \\ 2 & 1966 \\ 3 & 1957 \end{array}\right)$$

Por ejemplo, la primera fila de $\mathcal{R}$ indica que 1 es hijo de alguien que nació en 1990 (lo cual es cierto porque 1 es hijo de 3), y la segunda fila indica que 1 es hijo de alguien que nació en 1988 (lo cual también es cierto porque 1 es hijo de 2). No se asume que la entrada está ordenada, ni es necesario que la salida esté ordenada. La complejidad del algoritmo debe ser $O(n \log n)$ en peor caso. Se pide no usar estructuras de datos externas como AVLs.

Nota. Este tipo de operaciones es muy común en el ámbito de bases de datos, donde se suelen llamar *joins*.