Práctica 2: Dividir y conquistar

Ejercitación básica

Ejercicio 1. Implementar el algoritmo de ordenamiento *mergesort* en Python. Se provee un archivo `pr02_mergesort.py` con una función `mergesort` a completar. El programa ejecuta casos de test y hace un gráfico para comparar los tiempos de ejecución de *mergesort* y *selection sort*.

Ejercicio 2. Sea A un arreglo de números enteros sin repetidos. Decimos que un elemento de A es un "pico" si es el elemento máximo del arreglo y además todos los elementos anteriores están ordenados ascendentemente y todos los elementos posteriores están ordenados descendentemente. Por ejemplo:

$$[\underbrace{-2, 9, 10, 11, 13, 14, 15, 17, 19, 25, 50}_{\text{fragmento ascendente}}, \underbrace{51}_{\text{pico}}, \underbrace{49, 29, 12, 8, 7, 5}_{\text{fragmento descendente}}]$$

a) Proponer un algoritmo que encuentre el pico de un arreglo en $O(\log n)$.

b) Implementarlo en Python y convencerse de que es correcto haciendo tests.

Ejercicio 3. **[Merge de n vías]** Suponemos dados k arreglos ordenados $A_1, A_2, \ldots, A_k$, cada uno de n elementos. Queremos conseguir un arreglo ordenado que reúna todos los elementos de $A_1, A_2, \ldots, A_k$ en orden. Un método posible para hacer esto podría ser aplicando el algoritmo Merge: primero mezclar A_1 con A_2, después mezclar el resultado con A_3, etcétera.

a) ¿Cuál es la complejidad temporal en peor caso de dicho método?

b) Proponer un algoritmo cuya complejidad temporal sea estrictamente mejor que la del método propuesto.

Ejercitación adicional

Ejercicio 4. Suponemos que los valores de $T(0)$ y $T(1)$ se encuentran fijados. Resolver las siguientes ecuaciones de recurrencia (determinando el orden de complejidad de $T(n)$ en cada caso):

a) $T(n) = T(n/2) + 1$

b) $T(n) = T(n/2) + n$

c) $T(n) = 2\,T(n/2) + 1$

d) $T(n) = 2\,T(n/2) + n$

e) $T(n) = T(n/3) + 1$

f) $T(n) = T(n/3) + n$

g) $T(n) = 2\,T(n/3) + 1$

h) $T(n) = 2\,T(n/3) + n$

Ejercicio 5. Se propone el siguiente método para determinar si un elemento x aparece en un arreglo A (no necesariamente ordenado):

1. Si el arreglo es de tamaño 0, devolver `False` (es decir, x no aparece en el arreglo).
2. Si el arreglo es de tamaño 1, comparar $A[0]$ con x.
3. Si el arreglo A es de tamaño $n > 1$, dividir A en dos mitades B, C. Procediendo recursivamente, determinar si x aparece en B o si aparece en C.

Preguntas:

a) ¿Cuál es la complejidad temporal en peor caso de este método? Proponer una ecuación de recurrencia y resolverla.

b) ¿Cómo se compara el método con la búsqueda lineal?

Ejercicio 6. Supongamos dado un arreglo A de n números enteros. Sabemos que el arreglo empieza en 0 y que está ordenado crecientemente, pero puede contener repetidos. Por ejemplo, el arreglo podría ser de la forma:

$$[0,0,0,0,1,1,1,1,2,2,2,2,2,2,3,3,4,5,5,5,5]$$

Queremos "comprimir" el arreglo generando un arreglo de pares (x, r) donde x representa un elemento del arreglo original y r el número de repeticiones de ese elemento en el arreglo original. Por ejemplo, el arreglo de arriba se puede comprimir del siguiente modo:

$$[\underbrace{(0,4)}_{\text{el 0 aparece 4 veces}}, (1,4), (2,6), (3,2), (4,1), (5,4)]$$

Proponer un algoritmo para comprimir un arreglo A en tiempo $O(k \log n)$, donde k es el valor del número más grande que aparece en A.

Ejercicio 7. [Exponenciación binaria] Dado un número entero $x \in \mathbb{Z}$ y un natural $n \in \mathbb{N}_0$. Queremos calcular la potencia x^n. Por convención, declaramos que $x^0 = 1$ para todo $x \in \mathbb{Z}$.

Un método ingenuo para calcular la potencia es realizar una sucesión de n multiplicaciones, en tiempo $O(n)$:

$$\underbrace{((x \cdot x) \cdot x) \ldots \cdot x}_{n \text{ veces}}$$

Se puede calcular x^n de manera más eficiente con el siguiente método, basado en la técnica de D&C:

- Si $n = 0$, devolver 1.
- Si n es mayor que 0 y es par, calcular $y = x^{n/2}$ y devolver $y \cdot y$.
- Si n es impar, calcular $y = x^{(n-1)/2}$ y devolver $y \cdot y \cdot x$.

Se pide:

a) Implementar el método en Python y convencerse de que es correcto haciendo tests.

b) Analizar la complejidad temporal en peor caso.