Enunciado: AED2 / Práctica 5 - Ordenamiento / Ejercicio 6

Sea $A[1 \dots n]$ un arreglo que contiene n números naturales. Diremos que un rango de posiciones $[i \dots j]$, con $1 \leq i \leq j \leq n$, contiene una escalera en A si valen las siguientes dos propiedades:

1. $(\forall k : nat)\ i \leq k < j \Rightarrow A[k+1] = A[k] + 1$ (esto es, los elementos no sólo están ordenados en forma creciente, sino que además el siguiente vale exactamente uno más que el anterior).
2. Si $1 < i$ entonces $A[i] \neq A[i-1] + 1$ y si $j < n$ entonces $A[j+1] \neq A[j] + 1$ (la propiedad es *maximal*, es decir que el rango no puede extenderse sin que deje de ser una escalera según el punto anterior).

Se puede verificar fácilmente que cualquier arreglo puede ser descompuesto de manera única como una secuencia de escaleras. Se pide escribir un algoritmo para reposicionar las escaleras del arreglo original, de modo que las mismas se presenten en orden decreciente de longitud y, para las de la misma longitud, se presenten ordenadas de forma creciente por el primer valor de la escalera.

El resultado debe ser del mismo tipo de datos que el arreglo original. Calcule la complejidad temporal de la solución propuesta, y justifique dicho cálculo.

Por ejemplo, el siguiente arreglo: $[5, 6, 8, 9, 10, 7, 8, 9, 20, 15]$ debería ser transformado a: $[7, 8, 9, 8, 9, 10, 5, 6, 15, 20]$

Contexto y conocimientos previos

Este ejercicio se encuentra en la guía práctica de ordenamiento. Dependiendo de la organización de la cursada, si se presenta este ejercicio en una clase práctica probablemente sea la primera vez que mostremos una solución que requiere ordenar por dos criterios distintos. Es importante hacer foco en esto durante la resolución y explicar las técnicas disponibles para éstas situaciones. En la mayoría de los casos es simplemente ordenar 2 veces, pero hay que saber deducir a partir del enunciado por cuál criterio hay que ordenar primero y por cuál después para que el ordenamiento final sea el pedido.

El ejercicio permite mostrar la estrategia de armar una estructura temporal a partir de la entrada del algoritmo para poder resolver el problema de forma eficiente, y luego desarmar esa estructura temporal para retornar el resultado en el formato pedido por el enunciado.

Se asume que las y los alumnos: 1) Conocen los algoritmos de ordenamiento presentados durante las clases previas, específicamente MergeSort. No es necesario que puedan escribir un MergeSort completo desde cero, pero sí entender la idea general del algoritmo y su complejidad. 2) Conocen estas estructuras básicas: arreglo, lista enlazada, tupla. 3) Cuentan con herramientas para diseñar estructuras y el impacto de éstas en la complejidad de los algoritmos.

La resolución no entra en mucho detalle sobre el cálculo de complejidad ya que esto se estudia en la guía práctica 2 y los cálculos necesarios para este ejercicio son relativamente sencillos.

Resolución

Antes de arrancar con la resolución mostraría algunas entradas y salidas más sencillas para validar que las y los alumnos hayan comprendido el enunciado.

¿Cuáles son los títulos de los pasos que tenemos que hacer para resolver el problema?

Podemos subdividir el problema en 3 partes: 1) encontrar todas las escaleras, 2) ordenar las escaleras según el criterio pedido, y 3) armar el arreglo final en base a las escaleras ordenadas. Apliquemos la estrategia top-down para plantear primero una solución con estos 3 pasos y luego entramos en el detalle de cada uno.

Solucion(**in** A: arreglo(nat)) $\rightarrow$ **out** res: arreglo(nat)

```
1: E: arreglo(escalera) ← ObtenerEscaleras(A)        ▷ O(n)
2: OrdenarEscaleras(E, A)                            ▷ O(nlog(n))
3: res ← ReposicionarEscaleras(E, A)                 ▷ O(n)
```

Complejidad: $O(n) + O(nlog(n)) + O(n) = O(nlog(n))$

Parte 1: encontrar escaleras

¿Qué estructura podemos usar para representar una escalera dentro del arreglo de entrada?

Podemos representar las escaleras con una tupla que contiene los índices de inicio y fin de la escalera dentro del arreglo de entrada. Para trabajar de forma más semántica, vamos a definir un nuevo tipo de dato:

escalera se representa con **tupla** $\langle$ inicio: **nat**, fin: **nat** $\rangle$

¿Cómo podemos identificar una escalera dentro del arreglo de entrada? ¿Cuántos ciclos vamos a necesitar?

El inicio y fin de una escalera define un subarreglo del arreglo de entrada en donde todos los elementos allí contenidos son consecutivos entre ellos (condición 1 del enunciado) y además la escalera es maximal (condición 2 del enunciado). Planteamos 2 ciclos: el ciclo exterior encuentra el inicio de la escalera, el ciclo interior encuentra el fin. Al salir del ciclo interior, movemos el inicio de la próxima escalera al siguiente elemento después del fin de la escalera actual.

¿Podemos saber de antemano cuántas escaleras hay? ¿Cuál es la cantidad máxima posible de escaleras en un arreglo de tamaño n?

Como máximo podríamos tener n escaleras, esto sucede cuando todos los elementos adyacentes entre sí difieren en más de 1. Pero a priori no sabemos cuántas escaleras hay. En estos casos donde no sabemos exactamente el tamaño del resultado que devuelve el algoritmo, podemos primero armar el resultado en una lista enlazada que puede agregar atras en $O(1)$. Luego, convertimos la lista enlazada en un arreglo en $O(m)$ donde m es el tamaño de la lista enlazada. Dependiendo del algoritmo, también es posible retornar la lista enlazada directamente y trabajar con ella.

ObtenerEscaleras(**in** A: arreglo(nat)) $\rightarrow$ **out** res: arreglo(escalera)

```
 1: B: lista(escalera) ← CrearLista()                          ▷ O(1)
 2: n ← tam(A)
 3: i ← 1
 4: while i ≤ n do                                             ▷ O(n)
 5:     k ← i
 6:     while k < n ∧_L A[k + 1] = A[k] + 1 do                 ▷ O(n − i)
 7:         k ← k + 1
 8:     end while
 9:     AgregarAtras(B, ⟨ inicio: i, fin: k ⟩)                 ▷ O(1)
10:     i ← k + 1
11: end while
12: res ← ListaToArreglo(B)          ▷ O(tam(B)) = O(n) pues tam(B) ≤ n
```

Complejidad: $O(n)$

La complejidad resulta $O(n)$ porque si bien hay 2 ciclos, al asignar $i = k + 1$ después del ciclo interior en efecto nos salteamos en el ciclo exterior los elementos que ya recorrimos en el ciclo interior.

Parte 2: ordenar escaleras

¿Por cuántos criterios debemos ordenar? ¿Cuáles son?

Necesitamos ordenar las escaleras por 2 criterios distintos, así que vamos a realizar 2 pasadas de ordenamiento, una por cada criterio, y la segunda pasada debe ser con algún algoritmo estable.

Importante remarcar que la segunda pasada debe ser con un algoritmo estable para no desordenar lo que ordenó la primer pasada.

1. Ordenamos las escaleras según su primer elemento. Para obtenerlo, indexamos el arreglo original con el índice de inicio de la escalera.
2. Ordenamos las escaleras según su tamaño. Para obtenerlo, hacemos $fin - inicio + 1$ (sumamos 1 porque los índices son inclusivos).

¿Qué algoritmo podríamos usar para ordenar? ¿Cambia algo que el arreglo sea de escaleras (tuplas) en vez de naturales?

Si bien el arreglo de escaleras son tuplas, podemos realizar un MergeSort sobre alguna componente en particular dentro de la tupla, o incluso determinar la clave de ordenamiento en base a ambas componentes de la tupla.

¿Cuál sería el problema si ordenamos por separado un arreglo con los primeros elementos de cada escalera, y luego otro arreglo con el tamaño de cada escalera?

Presentamos el algoritmo.

OrdenarEscaleras(**in/out** E: arreglo(escalera), **in** A: arreglo(nat))

```
1: MergeSort(E, A)      ▷ O(nlog(n)), clave de ordenamiento: A[E[i].inicio]
2: MergeSort(E)         ▷ O(nlog(n)), clave de ordenamiento: E[i].fin - E[i].inicio + 1, decreciente
```

Complejidad: $O(nlog(n))$ con $n = tam(A)$

El peor caso es que tengamos todas escaleras de tamaño 1. Entonces el arreglo E de escaleras tendrá exactamente el mismo tamaño que el arreglo original A, y todas las escaleras tendrán el mismo inicio y fin. A raíz de esto, la complejidad del peor caso es ordenar n elementos, lo cual tiene un costo de $O(nlog(n))$ porque usamos MergeSort.

Parte 3: armar el resultado final

¿Cómo pasamos del arreglo de escaleras a un arreglo de naturales para armar el resultado final?

Iteramos por el arreglo de escaleras, y por cada una, iteramos por todos las índices entre inicio y fin (inclusives). Indexamos el arreglo original en cada uno de esos índices y colocamos ese elemento en el resultado final, comenzando desde el principio del arreglo y avanzando de a 1 cada vez que colocamos un elemento.

¿Podemos armar el resultado final directamente en el arreglo original o necesitamos un arreglo nuevo?

Vamos a necesitar armar el resultado final en un nuevo arreglo ya que es necesario mantener el arreglo original intacto para poder indexar en todos los elementos de cada escalera. Recordemos que definimos las escaleras solo con sus índices de inicio y fin en el arreglo original. Sin el arreglo original intacto, no sabemos qué elementos corresponden a esos índices.

Al terminar de iterar por todas las escaleras, habremos copiado todos los elementos del arreglo original al resultado final, pero en sus posiciones correctas.

ReposicionarEscaleras(**in** E: arreglo(escalera), **in** A: arreglo(nat)) → **out** res: arreglo(nat)

```
1: res: arreglo(nat) ← CrearArreglo(tam(A))      ▷ O(tam(A)) = O(n)
2: k ← 1
3: for i ← 1 to tam(E) do
4:     for j ← E[i].inicio to E[i].fin do
5:         res[k] ← A[j]
6:         k ← k + 1
7:     end for
8: end for
```

Complejidad: $O(n)$

La complejidad resulta $O(n)$ porque la cantidad total de elementos a copiar en el arreglo res es siempre $tam(A) = n$, sin importar la cantidad de escaleras encontradas.

Para finalizar la clase mostraría algunos ejemplos de ejecución para distintas entradas. Cómo se arma el arreglo intermedio con las escaleras, cómo queda ordenado después de cada pasada, y finalmente cómo armamos el resultado final.

Conclusión

Con este ejercicio presentamos una estrategia general para resolver cierto tipo de problemas de ordenamiento. Mostramos cómo podemos ordenar otras estructuras de datos utilizando algoritmos que originalmente trabajan sobre arreglos. Además aplicamos 2 criterios de ordenamiento, y también definimos una estructura de datos intermedia que nos permite resolver el problema de forma eficiente.