Enunciado

AED Primer Cuatrimestre 2024 / **Práctica 8 - Ordenamiento** / **Ejercicio 5**

Se tiene un arreglo de n números naturales que se quiere ordenar por frecuencia, y en caso de igual frecuencia, por su valor. Por ejemplo, a partir del arreglo [1, 3, 1, 7, 2, 7, 1, 7, 3] se quiere obtener [1, 1, 1, 7, 7, 7, 3, 3, 2]. Describa un algoritmo que realice el ordenamiento descrito, utilizando las estructuras de datos intermedias que considere necesarias. Calcule el orden de complejidad temporal del algoritmo propuesto.

Contexto del ejercicio y conocimientos previos del alumno

Este ejercicio se encuentra al comienzo de la guía. Puede ser uno de los primeros ejercicios a dar en una clase luego de introducir los algoritmos de ordenamiento y me parece un buen ejercicio para mostrar:

Como usar las estructuras de datos a la hora de resolver problemas (en particular de ordenamiento), qué son y para qué sirven los algoritmos de ordenamiento estables con un ejemplo concreto y cómo podemos reducir un problema menos general en uno más general que si sabemos resolver y podemos extrapolar con otros problemas (en particular uno de ordenamiento).

Se asume que los alumnos:

- Saben determinar el orden de complejidad temporal del peor caso de un algoritmo.
- Están familiarizados con las estructuras de datos presentadas en la guía de elección de estructuras y las complejidades de sus operaciones.
- Están familiarizados con los algoritmos de ordenamiento, no cómo implementarlos de memoria sino como funcionan, sus invariantes y complejidades.
- **Nunca** resolvieron un ejercicio de ordenamiento en el que hace falta ordenar por más de un criterio y ejercicios de ordenamiento con tuplas.

Desarrollo del ejercicio

Observación del enunciado: En este problema no solamente hay que ordenar por cantidad de apariciones, de mayor a menor, si no que tambien en caso de que dos elementos tengan la misma cantidad de apariciones tiene que estar primero el de menor valor.

Preguntas orientadoras para encaminar la resolución del ejercicio

¿Se vuelve más fácil el problema de ordenar si en vez de tener el arreglo: [1, 3, 1, 7, 2, 7, 1, 7, 3] tenemos el arreglo: [(1, 3), (3, 2),(7, 3), (2, 1)]? ¿Qué representa cada tupla respecto de los elementos del arreglo original?

Si ahora en vez de ordenar el arreglo original por los criterios del enunciado: Lo queremos ordenar por las segundas componentes de las tuplas en orden descendente y en caso de empate tiene que estar primero el elemento con un menor valor en su primer componente: ¿Se vuelve un problema más fácil? ¿Cómo pódemos hacerlo? ¿Tiene una relación ese ordenamiento con el ordenamiento que estamos buscando?

Resolución

Podemos separar el problema en tres subproblemas a resolver:

Paso 1: Transformar la lista original

Una primera intuición para llegar a la lista de tuplas puede ser recorrer la lista original, pero: ¿Cómo hacemos para guardar la cantidad de apariciones de cada elemento?

Dentro de los ejercicios de Sorting es muy común tener que apoyarnos en estructuras de datos auxiliares además del arreglo original para llegar a nuestro cometido. Por lo que podemos apoyarnos de un diccionario implementado sobre un AVL (DiccLog) para ir actualizando la cantidad de apariciones de los elementos.

Una vez que llenamos el diccionario, con los elementos del array y su cantidad de apariciones, podemos terminar de construir el array de tuplas usando el iterador del diccionario.

Modularizemos este proceso en dos funciones: Una que a partir del array va a devolver un diccionario, y otra que a partir del diccionario va a devolver un arreglo de tuplas.

Asumimos que hay aliasing, por lo que cuando pedimos el valor asociado a una clave en el diccionario, al modificar ese valor no hace falta volver a modificar el par clave valor. Ya que usamos referencias de las variables en vez de copias.

---

proc crearDiccionarioDeCantidades(**in** arr: Array<nat>) → **out** res: DiccLog<nat, nat>

```
1:  var res: DiccLog<nat, nat> = new diccionarioVacío();          ▷ O(1)
2:  for var i: int = 0 to arr.longitud() - 1 do                   ▷ O(n)
3:      var actual: int = arr[i];                                 ▷ O(1)
4:      if res.está(actual) then                                  ▷ O(Log(n))
5:          var previo: int = res.obtener(actual);                ▷ O(Log(n))
6:          previo = previo + 1;
7:      else
8:          res.definir(actual, 1);                               ▷ O(Log(n))
9:      end if
10: end for
11: return res;
```

Complejidad: $O(n) \times O(2Log(n)) = O(nLog(n))$ con $n = arr.longitud()$

---

proc crearArrayDeTuplas(**in** dict: DiccLog<nat, nat>) → **out** res: Array<Tupla <nat, nat>>

```
1: var res: Array<Tupla < nat, nat >> = new Array(dict.tamaño());          ▷ O(n)
2: var it : IteradorBidireccional = dict.iterador(); [1]                  ▷ O(1)
3: for var i: int = 0 to dict.tamaño() - 1 do                             ▷ O(n)
4:     res[i] = it;                                                       ▷ O(1)
5:     it = it.siguiente();                                               ▷ O(1)
6: end for
7: return res;
```

Complejidad: $O(n)$ con $n =$ dict.tamaño()

Paso 2: Ordenar el arreglo de tuplas

En este punto pasamos de un problema mucho más específico a un problema más general. Ahora nuestro problema es ordenar por las segundas componentes, de mayor a menor, y en caso de empate tiene que aparecer primero el elemento que tenga un menor valor en su primera componente.

A la hora de ordenar las tuplas es crucial elegir bien por cual criterio vamos a ordenar primero y si los algoritmos de ordenamiento que vamos a usar son estables o no.

Recordemos que lo que hacen los algoritmos de ordenamiento estables es que en caso que dos elementos tengan el mismo valor va a quedar primero el que estaba primero en el array antes de ordenar.

Como queremos que el criterio principal sea ordenar por las segundas componentes y, en caso de empate, desempatar por la primer componente: Primero ordenamos con cualquier algoritmo de ordenamiento por el criterio secundario / de desempate y luego ordenamos por el criterio principal con un algoritmo de ordenamiento **estable**.

Así, en caso de que dos elementos tengan la misma segunda componente, va a quedar primero aquel que estaba primero antes del segundo ordenamiento. Entonces, quién va a estar primero? El elemento cuya primer componente es menor, logrando el desempate que estabamos buscando!

¿Se podría ordenar primero por el criterio principal y luego por el secundario?

Una prueba de que esta idea no es correcta la podemos ver siguiendo los ordenamientos más el pasaje a array:

[(1, 3), (3, 2), (7, 3), (2, 1)] ⇝ [(1, 3), (7, 3), (3, 2), (2, 1)] ⇝ [(1, 3), (2, 1), (3, 2), (7, 3)] ⇝ [1,1,1,2,3,3,7,7,7] ✗

proc ordenarTuplas(**in** arr: Array<Tupla<nat, nat>>) → **out** arr: Array<Tupla<nat, nat>>

```
1: heapSort(arr)      ▷ De menor a mayor por las primeras componentes. O(nLog(n))
2: mergeSort(arr)     ▷ De mayor a menor por las segundas componentes. O(nLog(n))
3: return arr;
```

Complejidad: $O(nLog(n))$ con $n = arr.longitud()$

Paso 3: Reconstruir la solución

Para reconstruir la solución al ejercicio hay que recordar la semántica que le dimos a las tuplas: La primer componente representa el elemento y la segunda su cantidad de apariciones.

proc reconstruirArray(**in** arr:Array<Tupla<nat, nat>>, int tamaño)→ **out** res: Array<nat>

```
1: var res: Array<nat> = new Array(tamaño);                               ▷ O(n)
2: var indiceActual = 0;
3: for var i: int = 0 to arr.longitud - 1 do                              ▷ O(ed)
4:     for var j: int = 0 to arr[i].second - 1 do                         ▷ O(ca)
5:         res[indiceActual] = arr[i].first;                              ▷ O(1)
6:         indiceActual = indiceActual + 1;                               ▷ O(1)
7:     end for
8: end for
9: return res;
```

Complejidad: $O(n) + O(ed) \times O(ca) = O(n + ed \times ca) = O(n)$ con $ed =$ cantidad de elementos distintos en el array original, $ca =$ la cantidad de apariciones de $arr[i]$ y n la longitud del array original

La función recibe como parámetro el largo del array original para no tener que usar un vector y luego convertirlo a arreglo o tener que primero contar la cantidad de elementos totales para crear el array res.

Por más que hay un ciclo anidado en la función, la complejidad sigue siendo lineal respecto del array original. Ya que la cantidad de iteraciones totales va a ser igual a la cantidad total de elementos en el array original. Porque por cada elemento único del array vamos a hacer su cantidad de apariciones iteraciones. Por lo que la suma de iteraciones va a ser igual a la cantidad de elementos del array que estabamos buscando ordenar: $O(n)$

La complejidad final de nuestro algoritmo juntando las partes va a ser, siendo n la longitud del array:

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

[1] Recordar que el iterador está representado como una tupla, cuyo primer elemento es la clave y el segundo el valor.