Matías Eliel Waisman

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 y conocimientos previos del alumno

Este ejercicio se encuentra al principio de la guía de ordenamiento. Puede ser uno de los primeros ejercicios a dar en una clase practica 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), la importancia de los algoritmos de ordenamiento estables y como podemos reducir un problema menos general en uno más general que si sabemos resolver (en particular uno de ordenamiento).

Se asume que los alumnos:

- Saben determinar el orden de complejidad temporal del peor caso de un algoritmo.
- Estan familiarizados con las estructuras de datos presentadas en la guía de elección de estructuras y las complejidades de sus operaciones.
- Estan familiarizados con los algoritmos de ordenamiento, no como 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 ser de menor a mayor.

Preguntas orientadoras para encaminar la resolución del ejercicio

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

Si ahora en vez de ordenar la lista original por los criterios del enunciado, la queremos ordenar primero por la segunda componente de las tuplas en orden descendente y en caso de empate por la segunda en orden ascendente, se vuelve un problema más fácil? ¿Cómo pódemos hacerlo?

Resolución

Podemos separar el problema en varios subproblemas a resolver:

1. Transformar la lista original en una lista de tuplas.
2. Ordenar la lista de tuplas con los criterios que nos interesan.
3. Reconstruir la solución.

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 de la lista 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. Y una vez que tenemos para cada elemento de la lista original su cantidad de apariciones con el iterador del diccionario podemos terminar de construir la lista de tuplas.

Entonces modularizemos este proceso en dos funciones: Una que a partir de la lista va a devolver un diccionario, y otra que a partir del diccionario va a devolver una lista de tuplas.

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

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

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

*1 Asumimos que hay aliasing, por lo que cuando pedimos el valor asociada a una clave en el diccionario, al modificar el valor no hace falta volver a modificar el par clave valor. (Agregar explicación de porque pasa)

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(); *2                ▷ O(1)
3:  var i: int = 0;
4:  while i < dict.tamaño() do                                          ▷ O(n)
5:      res[i] = it;                                                    ▷ O(1)
6:      it.siguiente();                                                 ▷ O(1)
7:      i = i + 1;
8:  end while
9:  return res;
```

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

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

Agregar comentario complejidad recorrer el diccionario.

Paso 2: Ordenar el arreglo de tuplas

Una vez que llegamos al array de tuplas y es momento de ordenarlo 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.

Como queremos que el criterio principal a ordenar sea por cantidad de apariciones de mayor a menor y en caso de empate en cantidad de apariciones va a ir primero el menor numero. La solución va a ser primero ordenar el primer elemento de la tupla de menor a mayor con cualquier algoritmo de ordenamiento y luego con un algoritmo de ordenamiento **estable**.

¿Porque es crucial que sea estable el segundo ordenamiento? Porque si no vamos a perder el orden del primer

ordenamiento. Por ejemplo si primero ordenamos por la primera clave de menor a mayor y luego ordenamos la segunda de mayor a menor usando Heap Sort, que no es estable, lo que pasaría sería:

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

En cambio con un algoritmo de ordenamiento estable:

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

¿Se podría ordenar primero por el primer criterio y luego por el segundo?

Un contraejemplo de que este procedimiento no nos daría la solución correcta lo podemos ver siguiendo los resultados de 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: mergeSort(arr)                  ▷ De menor a mayor por el primer componente. O(nLog(n))
2: mergeSort(arr)                  ▷ De mayor a menor por el segundo componente. O(nLog(n))
3: return arr;
```

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

Paso 3: Reconstruir la solución

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 i: int = 0;
3:  var indiceActual = 0;
4:  while i < arr.longitud do
5:      var j: int = 0;
6:      while j < arr[i].second do
7:          res[indiceActual] = arr[i].first;
8:          indiceActual = indiceActual + 1;
9:          j = j + 1;
10:     end while
11:     i = i + 1;
12: end while
13: return res;
```

Complejidad: