DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Guia 9

2do cuatrimestre 2024 — Algoritmos y Estructuras de Datos I

Integrante	LU	Correo electrónico
Federico Barberón	112/24	jfedericobarberonj@gmail.com

Facultad de Ciencias Exactas y Naturales

Universidad de Buenos Aires

Ciudad Universitaria - Pabellón I

Intendente Güiraldes 2160 - C1428EGA

Ciudad Autónoma de Buenos Aires - Argentina

Tel/Fax: (54 11) 4576-3359

http://exactas.uba.ar

Índice

1. Guia 9

1.1. Ejercicio 1

Comparar la complejidad de de los algoritmos de ordenamiento dados en la teórica para el caso en que el arreglo a ordenar se encuentre perfectamente ordenado de manera inversa a la deseada.

HACER!

1.2. Ejercicio 2

Defina la propiedad de estabilidad en un algoritmo de ordenamiento. Explique por qué el algoritmo de *heapSort* no es estable

La estabilidad hace referencia a la propiedad de los algoritmos de ordenamiento de mantener el orden relativo de los elementos de igual comparación del arreglo original. En el caso del algoritmo *heapSort*, esta propiedad no se cumple ya que al momento de corregir los elementos que estan mal posicionados en el heap, se puede potencialmente perder el orden relativo original de los elementos.

1.3. Ejercicio 3

Imagine secuencias de naturales de la forma $s = Concatenar(s', s'')$, donde s' es una secuencia ordenada de naturales y s'' es una secuencia de naturales elegidos al azar. ¿Qué método utilizaría para ordenar s? Justificar. (Entiéndase que s' se encuentra ordenada de la manera deseada)

- Primero identifico el indice donde termina la secuencia ordenada s' recorriendo s hasta encontrar un elemento desordenado // $O(|s|)$
- Luego separo s en dos subarrays s' y s'' en base al indice encontrado // $O(|s|)$
- Uso MergeSort para ordenar el subarray s'' // $O(|s''| \log(|s''|))$
- Mergeo el subarray s' con el subarray ya ordenado s'' // $O(|s|)$

Complejidad final: $O(|s| + |s| + |s''| \log(|s''|) + |s|) = O(|s| + |s''| \log(|s''|))$

1.4. Ejercicio 4

Escribir un algoritmo que encuentre los k elementos más chicos de un arreglo de dimensión n, donde $k \leq n$. ¿Cuál es su complejidad temporal? ¿A partir de qué valor de k es ventajoso ordenar el arreglo entero primero?

proc obtenerKminimos (in A: Array < int >, in k: int) : Array < int >

Complejidad: $O(nk)$ (Usando un maxHeap creo que se podía hacer en $O(n \log(k))$)

```
var minimos := new Array<int> (k);
var min: int;

for i := 0 to k do
    min := i;
    for j := i + 1 to A.size() do
        if A[j] < A[min] then
            min := j
        endif
    endfor

    minimos[i] := A[min];
    swap(A, i, min);
endfor

return minimos;
```

A partir de valores de $k > \log(n)$ tiene sentido ordenar el arreglo primero.

1.5. Ejercicio 5

Se tiene un conjunto de n secuencias $\{s_1, s_2, \ldots, s_n\}$ en donde cada $s_i (i \leq i \leq n)$ es una secuencia ordenada de naturales. ¿Qué método utilizaría para obtener un arreglo que contenga todos los elementos de la unión de los s_i ordenados? Describirlo. Justificar.

Este problema es similar a la segunda parte del algoritmo de MergeSort, ya que tenemos n subarrays ordenados y lo único que falta hacer ahora es mergearlos. Por lo que una forma de resolverlo es recorrer los subarrays de a dos, mergando uno con el siguiente, y volver a repetir el proceso con los subarrays ya mergeados hasta que quede un solo subarray, que va a tener todos los elementos ordenados y esa va a ser la respuesta. En caso de que n sea impar, se puede dejar el último subarray intacto hasta el final, donde será mergeado en el último paso.

Mergear todos los arrays en cada paso tiene una complejidad de $O(\sum_{i=1}^{n} |s_i|)$ y la cantidad de veces que tengo que mergear todos los arrays es $\log(n)$, por lo tanto la complejidad del algortimo es $O(\log(n) \cdot \sum_{i=1}^{n} |s_i|)$

Ejemplo:

$$[\ \underbrace{[1,2,3],[5,6]}_{\text{Merge}},\underbrace{[1,8,9],[2,3,7]}_{\text{Merge}}\ ]$$

$$[\ \underbrace{[1,2,4,5,6],[1,2,3,7,8,9]}_{\text{Merge}}\ ]$$

$$[\ [1,1,2,2,3,4,5,6,7,8,9]\ ]$$

1.6. Ejercicio 6

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 algortimo que relice el ordenamiento descripto, utilizando las estructuras de datos intermedias que considere necesarias. Calcule el orden de complejidad temporal del algoritmo propuesto.

Para resolver este problema vamos a utilizar la metodología *radix* para ordenar por varios criterios. En este caso, el criterio principal es la frecuencia y el secundario es el valor.

- Ordenamos primero los numeros por valor usando MergeSort // $O(n \log(n))$
- Creamos una lista de tuplas donde la primer componente es un número del arreglo original y la segunda es la cantidad de apariciones de ese número. Al tener el arreglo ordenado, podemos crear este nuevo en tiempo lineal // $O(n)$
- Pasamos esa lista a un array y lo ordenamos usando MergeSort por la segunda componente en orden descendente // $O(n \log(n))$
- Recorremos este último array e insertamos en un nuevo array (o sobreescribimos el original) cada elemento tantas veces como repeticiones tenga // $O(n)$

Complejidad final: $O(n \log(n) + n + n \log(n) + n) = O(n \log(n))$

```
proc ordenarPorFrecuenciaYValor (inout A: Array < int >)
    MergeSort(A); // O(nlog(n))

    var repeticiones := new ListaEnlazada< Tupla<int, int> >();

    for i := 0 to A.size() do // O(n)
        if repeticiones.size() != 0 && repeticiones.ultimo()[0] == A[i] then // O(1)
            repeticiones.ultimo()[1]++; // O(1)
        else
            repeticiones.agregarAtras(new Tupla(A[i], 1)); // O(1)
        endif
    endfor

    var arrRepeticiones = List2Array(repeticiones); // O(n)
    MergeSort(arrRepeticiones); // por segunda componente en orden descendente; O(nlog(n))

    var j := 0;

    for i := 0 to A.size() do // O(n)
        if arrRepeticiones[j][1] == 1 then
            j++;
        else
            arrRepeticiones[j][1]--;
        endif

        A[i] := arrRepeticiones[j][0];
    endfor
```

1.7. Ejercicio 7

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

1. $(\forall k : \mathbb{N})\ (i \leq j < j \rightarrow A[k+1] = A[k] + 1)$ (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 cuaqluier arreglo puede ser descompuesto de manera única como una secuencia de escaleras. Se pide escribir un algortimo 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 en 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 propuedas, y justifique dicho cálculo.

Por ejemplo, el siguiente arreglo

$$[5, 6, 8, 9, 10, 7, 8, 9, 20, 15]$$

debería ser transofrmado a

$$[7, 8, 9, 8, 9, 10, 5, 6, 15, 20]$$

- Construimos un arreglo de arreglos donde cada elemento es una escalera del arreglo original. // $O(n)$
- Ordenamos este arreglo por el primer elemento de cada escalera en forma creciente con MergeSort. // $O(n \log(n))$
- Ordenamos el arreglo por la longitud de cada escalera en forma decreciente con MergeSort. // $O(n \log(n))$
- Aplanamos el arreglo de escaleras. // $O(n)$

Complejidad final: $O(n \log(n))$

```
proc ordenarEscaleras (inout A: Array < int >)
    Complejidad: O(n log(n))

    var escaleras: ListaEnlazada <ListaEnlazada< int >> := new ListaEnlazada();
    var escaleraActual: ListaEnlazada< int > := new ListaEnlazada();

    for i := 0 to A.size() do // O(n)
        if i == 0 || A[i] == A[i-1] + 1 then
            escaleraActual.agregarAtras(A[i]); // O(1)
        else
            escaleras.agregarAtras(escaleraActual); // O(1)
            escaleraActual := new ListaEnlazada< int >();
            escaleraActual.agregarAtras(A[i]); // O(1)
        endif
    endfor

    escaleras.agregarAtras(escaleraActual);  // O(1)

    var arrEscaleras: Array< ListaEnlazada< int >> := List2Array(escaleras); // O(n)

    MergeSort(arrEscaleras); // por primer elemento de manera creciente; O(nlog(n))
    MergeSort(arrEscaleras); // por longitud de manera decreciente; O(nlog(n))

    var i: int := 0;

    for j := 0 to arrEscaleras.size() do // O(n) en total
        var it: IteradorBidireccional := arrEscaleras[j].iterador();

        while it.haySiguiente() do
            A[i] := it.siguiente();
            i++;
        endwhile
    endfor
```