images/logo_dc.jpg

Algoritmos y Estructuras de Datos (ex Algo II)

Práctica 8

19 de noviembre de 2023 Miauri :3

Plantilla de: @valnrms

images/logo_uba.jpg

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2610 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (++54 +11) 4576-3300
http://www.exactas.uba.ar

1. Ejercicio 3

Idea: Aplicar selection sort de manera parcial, hacer k recorridas lineales y luego dar la subsarray [0,k) resultante. Tiene complejidad O(n*k).
Si k >log(n) conviene ordenarlo en O(n*log(n)) y luego dar la subarray [0,k). La complejidad queda O(n*log(n)+k)) pero como k<=n se tiene que es O(n*log(n)).

```
encontrarKMásPequeños(in A: array<int>; in k: int): array<int>
# Complejidad: O(k*n)

for (i=0; i<k; i++){  // Ejecuta k veces. Complejidad del ciclo k*O(n) = O(k*n)
    indiceMin = i  // O(1)
    for (j=i; j<A.length;j++){  // Ejecuta n veces. Complejidad ciclo interno: O(n)
        if (A[i] > A[j]) then  // O(1)
            indiceMin = j  // O(1)
    }
    swap(A, i, indiceMin)  // O(1)
}

res = new array<int>(k)  // O(k)

for (i=0; i< res.length; i++){  // Ejecuta k veces. Complejidad del ciclo: // O(k)
    res[i] = A[i]  // O(1)
}

return res  // O(1)
```

```
swap(inout A: array<T>; in i: int; in j: int)
// Complejidad: O(1)
// Requiere: 0 <= i,j < A.length

temp = A[i]  // O(1)
A[i] = A[j]  // O(1)
A[j] = temp  // O(1)
```

2. Ejercicio 5

```
Sea n la longitud de la array A.
Sea h la cantidad de elementos distintos que tiene la array A.
OBS: h <= n

repeSort(inout A: array<int>)
// Complejidad: O(n*log(n)) + O(h) + O(h*log(h)) + O(n) = O(n*log(n) + h*log(h)), como h<=n, tenemos que es igual a:
# Complejidad final: O(n*log(n)).

d: DiccionarioLog<int, int>  // O(1)  // Donde las claves son un elemento de A y su valor la cantidad de apariciones que tiene.

for (i=0; i<A.length; i++){  // El ciclo se ejecuta n veces y su cuerpo es O(log(n)). Por lo tanto la complejidad del ciclo es // O(n*log(n))
    if (d.está?(A[i]) then  // Evaluar la guarda: O(log(n)), luego ambas ramas cuestan lo mismo, O(log(n))
        d.definir(A[i], A[i]+1)  // O(log(n))
    else
        d.definirRápido(A[i], 1)  // O(log(n))
}

arr_auxiliar = new array<tuple<int, int>>(d.length)  // O(h)  // Array de tuplas <elemento, repeticiones>

i = 0  // O(1)
for (key in d){  // Se ejecuta h veces. La complejidad del cilo es O(h*log(h))
    arr_auxiliar[i] = tuple<k, d.obtener(k)>  // O(log(h))
    i++  // O(1)
}

// Ordeno de menos prioritario a más prioritario.
arr_auxiliar.mergeSort() <- Por primer valor de la tupla (elemento) de forma creciente  // O(h*log(h))
arr_auxiliar.mergeSort() <- Por segundo valor de la tupla (repeticiones) de forma creciente.  // O(h*log(h))

k=0  // O(1)

for(i=0; i<arr_auxiliar.length; i++){  // Se ejecuta h veces. La sumatoria de los elementos de arr_auxiliar[i][1] con i en rango es igual a n. Por lo tanto la complejidad del ciclo es O(n)
    for(j=0; j<arr_auxiliar[i][1]; j++)  // Se ejecuta arr_auxiliar[i][1]
        A[k] = arr_auxiliar[i][0]  //O(1)
        k++  //O(1)
    }
}
```

3. Ejercicio 6

La complejidad la vamos a medir en terminos de n, longitud de A y h, cantidad de escaleras. Notar que $h \leq n$
Obs: miHeap es de tipo tupla de <int,int>. El primer elemento corresponder al inicio incluido de la escalera, y el segundo elemento corresponde al fin sin incluir de la escalera, es decir, [inicio,fin). Notar que longitud del intervalo = fin - inicio.

```
escaleraSort(inout A: array<int>)
# La complejidad final es O(n + h*log(h))

miHeap = new ColaDePrioridad<tuple<int,int>> (*)  // O(1)

while(i<A.length){  // Evaluar la guarda O(1) // La complejidad del ciclo total es de O(n + h*log(h))
    info = new tuple<int, int>  // O(1)
    info[0] = i  // O(1)

    while(j<A.length-1 && A[j+1] = A[j]+1){  // Evaluar la guarda O(1) // Se ejecuta largo de la escalera actual veces.
        j++  // O(1)
    }

    j++  // O(1)
    info[1] = j  // O(1)
    i = j  // O(1)

    miHeap.encolar(info)  // O(log(h))
}

k = 0

while(!miHeap.estaVacio?()){  // Evaluar la guarda O(1) // Se ejecuta h veces. Como la sumatoria de los largo de las
     escaleras es n. El for interno tiene complejidad O(n). Luego la complejidad del while es O(h*loh(h) + n)
    info = miHeap.desencolar()  // O(log(h))
    for(i=0; i <(info[1]-info[0]); i++)  // Se ejecuta largo de la escalera actual veces.
        A[k] = info[0] + i  // O(1)
        k++  // O(1)
    }
}

(*) miHeap es una cola de prioridad que mantiene un orden interno de la siguiente manera:
Criterio princial, longitud: info[1] - info[0]
Criterio secundario, es decir, de desempate: valor inicial de la escalera, es decir, info[0]
```

4. Ejercicio 9

OBS: #gen acotado. Lo considero cte.
Mido la complejidad en terminos de n = p.length.

```
ordenarPlanilla(inout p: planilla)
# Complejidad: O(n)

// Crear bucket nota.
bucket_nota = new array <listaEnlazada<Alumno>>(11)  // O(1)
// Asumo que todas las distasEnlazadas se inicializan vacias.
for (i=0; i < p.length; i++){  // Ejecuta n veces O(1): Es O(n)
    bucket_nota[p[i].nota].agregarAlFinal(p[i])  // O(1) -cuerpo-
}

// A partir del bucket nota, creo el bucket género.
bucket_género = new array <listaEnlazada<Alumno>>(#gen - 1)  // O(1)
for (i=0; i < bucket_nota.length; i++{  // Todo el for es O(n)  (*)
    it = bucket_nota[i].iterador()  // O(1)
    while (it.haySiguiente()){
        bucket_genero[p[i].genero].agregarAlFinal(p[i])  // O(1)   // Estoy asumiendo que genero es un número.
        it.siguiente()  // O(1)
    }
}

// A partir del bucket género, le doy orden a p.
k = 0  // O(1)
for(i=0; i < bucket_género.length; i++){  // Mismo razonamiento que antes, todo el for tiene complejidad O(n)
    it = bucket_género[i].iterador()  // O(1)
    while (it.haysiguiente()){
        p[k] = it.siguiente()  // O(1)
        k++  // O(1)
    }
}

(*) El for se ejecuta bucket_nota.length veces. El while de dentro se ejecuta bucket_nota[i].length veces. Como la sumatoria de #elementos que tiene bucket_nota[i] para cada i en rango es igual a n.
Va a haber n ejecuciones de operaciones O(1). Por lo tanto, todo el for tiene complejidad O(n)

Item c) No contradice el teorema de "lower_bound" ya que este algoritmo O(n) no es un algoritmo general de ordenamiento sino que se basa en datos que sabemos del input. Estos son, una cantidad acotada de notas [desde 0 hasta 10]
y un número acotado de generos [desde 0 hasta #gen sin incluir)
```

5. Ejercicio 11

Sea A[1...n] un arreglo de números naturales en rango (cada elemento está en el rango de 1 a k, siendo k alguna constante). Diseñe un algoritmo que ordene esta clase de arreglos en tiempo O(n). Demuestre que la cota temporal es correcta.

La mágia esta en que k es una cte.
Consideramos rango [inicio, fin), es decir inicio incluido hasta fin sin incluir.
El algoritmo fue escrito en terminos de rango, inicio-fin para ser más generico.
En el ejercicio que nos piden, tomemos: inicio = 1; fin = k+1 (El +1 es para que k este incluido)

```
ordenarEnRango(inout A: array)
# Complejidad: O(n + (fin-inicio)), como tomamos fin-inicio como cte, la complejidad final es O(n).

arr_repeticiones = new array<int>(fin-inicio)  // O(fin-inicio)
// Asumo que los valores de la array se inicializan en 0.

for (i=0; i<A.length; i++){  // Ejecuta n veces. Complejidad del ciclo: O(n)
    arr_repeticiones[A[i]-inicio] += 1  // O(1)
}

indice = 0  // O(1)
for (i=0; i<arr_repeticiones.length; i++){  // El for exterior ejecuta fin-inicio veces. La complejidad del ciclo termina siendo O(n + (fin-inicio))
    for (j=0; j < arr_repeticiones[i]; j++){  // El for interno ejecuta arr_repeticiones[i] veces. La sumatoria de los elementos de arr_repeticiones es n.
        A[indice] = i + inicio  // O(1)
        indice++  // O(1)
    }
}
```

6. Ejercicio 12

Sabemos que:

- A lo sumo $\sqrt{n}$ estan fuera del rango $[20, 40]$.
- Los enteros de $[20, 40]$ son un conjunto finito acotado.
- Las muestras son enteros positivos (Por cómodidad considero el 0 como positivo, no me afecta la complejidad).

Como son enteros positivos, tenemos que:

- $[0, 19]$ y $[20, 40]$ $\longrightarrow$ Acotado $\longrightarrow$ Puedo usar countingSort.
- $[41, +inf)$ $\longrightarrow$ No acotado $\longrightarrow$ Sé que hay menos de $\sqrt{n}$ elementos, "Se banca" un algoritmo cuadrático.

Idea: Puedo ordenar con countingSort de $[0, 40]$ y luego ordenar a $[41, +inf)$ con un algoritmo de ordenamiento cuadratico como insert sort. Luego unir los resultados.
Notar que para todo n en $[0, 40]$ y para todo m en $[41, +inf)$ se cumple que $n \leq m$. Por lo tanto puedo "concatenar" los bloques y va a andar bien.

La complejidad será medida con n: longitud de A y h: #elems distintos en el intervalo $[41, +inf)$, se sabe que son $\leq \sqrt{n}$.

```
ordenarDatos(inout A: array<int>)
# Requiere que todos los elementos de A sean mayores o iguales a 0.
// Complejidad: O(n) + O(h), como h en el peor caso es sqrt(n), se tiene que O(n + sqrt(n)) = O(n)
# La complejidad final es O(n) como se quería :)

dMayores: DiccionarioLog<int,int>  // O(1)  // Claves: elem, valores: #repeticiones

arrAparicionesMenores = new array<int>(41)  // O(1)  // Asumo que se inicializan en 0

for(i=0; i<A.length; i++){  // **NOTA
    if (A[i] <= 40) then  // Evaluar la guarda O(1)
        arrAparticionesMenores[i] += 1  // O(1)
    else if (dMayor.esta?(A[i]) then  // Evaluar la guarda O(log(h))
        dMayores.definir(A[i], dMayores.obtener(A[i]) + 1)  //O(log(h))
    else
        dMayores.definirRápido(A[i], 1)  //O(log(h))
}

arrMayores = new array<tuple<int,int>>(dMayores.length)  // O(h)  // Array de tuplas, el primer valor es el elem y el segundo #repeticiones

i=0  // O(1)

for(key in dMayores){  // Se ejecuta h veces. Complejidad del ciclo O(h*log(h)) = O(n) por (*)
    arrMayores[i] = tuple<key, dMayores.obtener(key)>  \\ O(log(h))
    i++  // O(1)
}

arrMayores.insertSort() <- Criterio: ordena crecientemente según el primer valor de la tupla (ordena por elem)  // O(h^2) = O((sqrt(n))^2) = O(n)

k=0  // O(1)

// En total, sumando las operaciones, entre los dos ciclos, se forma complejidad O(n)
for(i=0; i<arrAparicionesMenores.length; i++){  // Complejidad del ciclo O(#elementos menores o iguales a 40)
    for(j=0; j<arrAparicionesMenores[i]; j++){
        A[k] = i  // O(1)
        k++  // O(1)
    }
}

for(i=0; i<arrMayores.length; i++){  // Complejidad del ciclo O(#elementos mayores a 40)
    for(j=0; j<arrMayores[i]; j++){
        A[k]=i  // O(1)
        k++  // O(1)
    }
```

```
}

// **Nota: Ese ciclo se ejecuta n veces. En el peor caso se tiene que h = sqrt(n). Por lo tanto n-sqrt(n) elementos se
     realizan en O(1) y sqrt(n) elementos se realizan en O(log(sqrt(n)), entonces se tiene que O(n - sqrt(n)) +
     O(sqrt(n)*log(sqrt(n))) = O(n) + O(n) por (*), luego queda que el ciclo es O(n).

(*) Hice el límite en symbolab: f(n): sqrt(n)*log(sqrt(n)) pertenece a O(n).
```

7. Ej de parcial: tablaKPosiciones

Link al parcial

```
tablaKPosiciones(in A: array<tupla<nombre:string, puntos:nat, intentos:nat>>, in k:nat): array<string>
# Complejidad final: O(n+k+k*log(n)) = O(n + k*log(n)) como se quería :)

// Nueva arr igual a la original pero con la información de su indiceOriginal para hacer un algoritmo estable.
arr = new array<tupla<nombre:string, puntos:nat, intentos:nat, indiceOriginal:nat>>(A.length)  // O(n)

for (i=0; i< A.length; i++){  // Se ejecuta n veces. Complejidad del ciclo O(n)
    arr[i].nombre = A[i].nombre  // O(1)
    arr[i].puntos = A[i].puntos  // O(1)
    arr[i].intentos = A[i].intentos  // O(1)
    arr[i].indiceOriginal = i  // O(1)
}

minHeap = arr2Heap*(arr)  // O(n)  // Utiliza algoritmo de Floyd para crear un min_heap.

res = new array<tupla<nombre:string, puntos:nat, intentos:nat>>(k)  // O(k)

for (i=0; i<k; i++){  // Se ejecuta k veces. complejidad del ciclo O(k*log(n))
    escalador_info = minHeap.desencolar*()  // O(log(n))
    res[i].nombre = escalador_info.nombre  // O(1)
    res[i].puntos = escalador_info.puntos  // O(1)
    res[i].intentos = escalador_info.intentos  // O(1)

return res  // O(1)
```

El minHeap, sus operaciones, incluidas particularmente arr2Heap* y desencolar* utilizan la siguiente interfaz de comparación:

- Criterio 1: puntos
- Criterio 2: intentos
- Criterio 3: indiceOriginal

En código se vería así:

```
comparteTo(in escalador_1: tupla<nombre:string, puntos:nat, intentos:nat, indiceOriginal:nat>, in escalador_2: tupla<nombre:string, puntos:nat, intentos:nat, indiceOriginal:nat>): int
# Complejidad final: O(1)

if (escalador_1.puntos - escalador_2.puntos == 0) then  // O(1)
    if (escalador_1.intentos - escalador_2.intentos == 0) then  // O(1)
        return escalador_1.indiceOriginal - escalador_2.indiceOriginal  // O(1)
    else
        return escalador_1.intentos - escalador_2.intentos  // O(1)
else
    return escalador_1.puntos - escalador_2.puntos  // O(1)
```