Algoritmos y Estructuras de Datos II

Guía 4

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Ejercicios obligatorios de la práctica

Integrante	LU	Correo electrónico
Bruno, Patricio Damián	62/19	pdbruno@gmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

1. Ordenamiento

sortSets(**in** A: `arreglo(ConjuntoDeNaturales)`)) $\rightarrow$ res: arreglo(ConjuntoDeNaturales))

1: $K \leftarrow obtenerMaximoCardinal(A)$ $\triangleright$ $O(NK)$
2: $cardinales : arreglo\_dimensionable\ de\ lista(puntero(ConjuntoDeNaturales))$
3: $cardinales \leftarrow crearArreglo(K)$ $\triangleright$ $O(K)$
4: **for** $i \leftarrow 1$ to K **do** $\triangleright$ $O(K)$
5:     $cardinales[i] \leftarrow Vacia()$ $\triangleright$ $O(1)$
6: **end for**
7: **for** $i \leftarrow 1$ to $tam(A)$ **do** $\triangleright$ $O(N)$
8:     $cardinal \leftarrow obtenerCardinal(A[i])$ $\triangleright$ $O(K)$
9:     $AgregarAtras(cardinales[cardinal], \&A[i])$ $\triangleright$ $O(1)$
10: **end for**
11: $res : arreglo\_dimensionable\ de\ ConjuntoDeNaturales$
12: $res \leftarrow crearArreglo(tam(A))$ $\triangleright$ $O(N)$
13: $j \leftarrow 1$ $\triangleright$ $O(1)$
14: **for** $i \leftarrow 1$ to K **do** $\triangleright$ $O(K)$
15:     $it \leftarrow CrearIt(cardinales[i])$
16:     **while** $HaySiguiente?(it)$ **do** $\triangleright$ $O(N)$ en total
17:         $res[j] \leftarrow *Siguiente(it)$ $\triangleright$ $O(copy(Siguiente(it)))$
18:         $j \leftarrow j + 1$ $\triangleright$ $O(1)$
19:         $Avanzar(it)$ $\triangleright$ $O(1)$
20:     **end while**
21: **end for**
22: **return** res

Justificación: La idea es usar bucket sort, donde tengo K buckets, siendo K el mayor cardinal de todos los conjuntos en A. En cada bucket tengo una lista de punteros a los conjuntos (para evitar el costo de copia) cuyo cardinal corresponde con el índice del bucket. Voy a **suponer** que el costo de copia de un conjunto, sabiendo que tiene a lo sumo K elementos, es O(K). A continuación voy a demostrar por qué algoritmo a partir de la línea 14 tiene complejidad en peor caso de O(NK) y luego justificar la complejidad de todo el algoritmo

Sea A_i con $1 \leq i \leq N$ una sucesión o secuencia de conjuntos
Sabemos que $\forall i / 1 \leq i \leq N$, se tiene $\#A_i \leq K$
Sea C_j con $0 \leq j \leq K$ una sucesión o secuencia de conjuntos tal que
$C_j = \{\ C \mid \exists i \in \mathbb{N} / 1 \leq i \leq N \wedge A_i = C \wedge \#A_i = j\}$
Por lo tanto C_j constituye una partición de A_i (hace las veces de buckets) y por lo tanto $\sum_{j=0}^{K} \#C_j = N$
Dicho esto, la complejidad a partir de la línea 14 se puede expresar de la siguiente manera:
$\sum_{j=0}^{K} O(1) + \#C_j * (O(K) + O(1)) = \sum_{j=0}^{K} \#C_j * O(K) =$
$O(K) \sum_{j=0}^{K} \#C_j = O(K) * N = O(NK)$

$T_{peor}(N) = O(NK) + O(K) + O(K) + \sum_{i=1}^{N}(O(1) + O(K)) + O(N) + O(1) + O(NK)$
$= O(NK) + O(K) + N * O(K) + O(N) + O(1)$
$= O(max(NK, K, N, 1))$
$= O(NK)$

---

obtenerMaximoCardinal(**in** A: `arreglo(ConjuntoDeNaturales)`) $\rightarrow$ res: nat

$res \leftarrow -1$ $\triangleright\ O(1)$
for $i \leftarrow 1$ to $tam(A)$ **do** $\triangleright\ O(N)$
  $card \leftarrow obtenerCardinal(A[i])$ $\triangleright\ O(K)$
  $res \leftarrow max(res, card)$ $\triangleright\ O(1)$
end for
return res

Justificación: Hago una operación con complejidad temporal en peor caso perteneciente a O(K) un total de N veces, por lo tanto la complejidad final es O(NK)

---

obtenerCardinal(**in** C: `ConjuntoDeNaturales`) $\rightarrow$ res: nat

$res \leftarrow 0$ $\triangleright\ O(1)$
$it \leftarrow CrearIt(C)$ $\triangleright\ O(1)$
while $HaySiguiente?(it)$ **do** $\triangleright\ O(K)$
  $res[j] \leftarrow res + 1$ $\triangleright\ O(1)$
  $Avanzar(it)$ $\triangleright\ O(1)$
end while
return res

Justificación: Para cada elemento en C (a lo sumo K elementos) sumo 1 a res

---

2. Dividir y Conquistar

2.1.

sonDisjuntos(**in** C: `arreglo(ConjuntoDeNaturales)`) $\rightarrow$ res: bool

$res \leftarrow sonDisjuntosRec(C, 1, tam(C))$
return $res.sonDisjuntos$

sonDisjuntosRec(**in** C: `arreglo(ConjuntoDeNaturales)`, **in** $inicio$: `nat`, **in** fin: `nat`) $\rightarrow$ res: $< sonDisuntos : bool, union : lista(nat) >$

if $fin - inicio = 1$ **then** $\triangleright$ $O(1)$
  $list \leftarrow Vacia()$ $\triangleright$ $O(1)$
  for x in $C[inicio]$ **do** $\triangleright$ $O(M)$
    $AgregarAtras(list, x)$ $\triangleright$ $O(1)$
  end for
return $< true, list >$
else
  $med \leftarrow (inicio + fin)/2$ $\triangleright$ $O(1)$
  $izq \leftarrow sonDisjuntosRec(C, inicio, med)$ $\triangleright$ $O(T(n/2))$
  $der \leftarrow sonDisjuntosRec(C, med, fin)$ $\triangleright$ $O(T(n/2))$
  $disjuntos \leftarrow true$ $\triangleright$ $O(1)$
  $listaUnion \leftarrow Vacia()$ $\triangleright$ $O(1)$
  $i, j, k \leftarrow 1$ $\triangleright$ $O(1)$
  $itIzq \leftarrow CrearIt(izq.union)$ $\triangleright$ $O(1)$
  $itDer \leftarrow CrearIt(der.union)$ $\triangleright$ $O(1)$
  while $(HaySiguiente?(itIzq) \vee HaySiguiente?(itDer)) \wedge disjuntos$ **do** $\triangleright$ $O(N * M)$
    if $HaySiguiente?(itIzq) \wedge HaySiguiente?(itDer) \wedge Siguiente(itIzq) = Siguiente(itDer)$ **then**
      $disjuntos \leftarrow false$ $\triangleright$ $O(1)$
    end if
    $\triangleright$ el resto es el algoritmo clásico de merge de secuencias
    if $\neg HaySiguiente?(itDer) \vee (HaySiguiente?(itIzq) \wedge Siguiente(itIzq) < Siguiente(itDer))$ **then**
      $AgregarAtras(listaUnion, Siguiente(itIzq))$ $\triangleright$ $O(1)$
      $Avanzar(itIzq)$ $\triangleright$ $O(1)$
    else
      $AgregarAtras(listaUnion, Siguiente(itDer))$ $\triangleright$ $O(1)$
      $Avanzar(itDer)$ $\triangleright$ $O(1)$
    end if
  end while

end if
return $< disjuntos \wedge izq.disjuntos \wedge der.disjuntos, listaUnion >$

Justificación: En este esquema de Divide & Conquer, el caso base cuesta O(M), dividir cuesta O(1) y combinar los resultados cuesta O(NM). La idea es que de cada subproblema obtengo una lista ordenada de naturales que representa la unión de todos los conjuntos que entraban en ese subproblema. Entonces para los dos resultados de los subproblemas, aplico el algoritmo de mergeo de secuencias ordenadas, con la particularidad de que si encuentro dos elementos iguales, ya se que los conjuntos no son disjuntos. La cantidad de elementos en la unión de los N conjuntos es N*M, entonces en cada llamado recursivo el costo de combinar ambos subproblemas implica iterar sobre esos N*M elementos para generar la nueva unión. A continuación, el cálculo de complejidad en peor caso del algoritmo.

$$T(N) = \begin{cases} 2T(N/2) + O(NM) & si \quad N > 1 \\ \\ O(M) & si \quad N = 1 \end{cases}$$

$T(N) = 2T(\frac{N}{2}) + O(NM)$

$= 2(2T(\frac{N}{4}) + O(\frac{N}{2}M)) + O(NM)$

$= 2^2T(\frac{N}{2^2}) + \sum_{j=0}^{2-1} O(\frac{N}{2^j}M)$

si seguimos expandiendo queda

$2^iT(\frac{N}{2^i}) + \sum_{j=0}^{i-1} O(\frac{N}{2^j}M)$

hasta que $i = log_2(N)$. También recordemos que T(1) es O(M)

$2^{log(N)}T(\frac{N}{2^{log(N)}}) + \sum_{j=0}^{log(N)-1} O(\frac{N}{2^j}M)$

$= NT(1) + \sum_{j=0}^{log(N)-1} O(\frac{N}{2^j}M)$

$= N * O(M) + \sum_{j=0}^{log(N)-1} O(\frac{N}{2^j}M)$

$= O(NM) + \sum_{j=0}^{log(N)-1} O(\frac{N}{2^j}M) \leq O(NM) + \sum_{j=0}^{log(N)-1} O(NM)$

$= O(NM) + (log(N)-1) * O(NM) = (log(N)-1+1)O(NM) = O(MNlog(N))$

2.2.

sonDisjuntos(**in** C : `arreglo(ConjuntoDeNaturales)`) $\rightarrow$ res: bool

```
elementos : arreglo_dimensionable de bool
M ← obtenerCardinal(C[1])                         ▷ O(M)
elementos ← crearArreglo(tam(C) * M))             ▷ O(N * M)
for i ← 1 to tam(elementos) do                    ▷ O(N * M)
    elementos[i] ← false                          ▷ O(1)
end for
res ← true                                        ▷ O(1)
i ← 1                                             ▷ O(1)
while i ≤ tam(C) ∧ res do                         ▷ O(N)
    it ← CrearIt(C[i])                            ▷ O(1)
    while HaySiguiente?(it) ∧ res do              ▷ O(M)
        elem ← Siguiente(it)                      ▷ O(1)
        if elementos[elem] then                   ▷ O(1)
            res ← false                           ▷ O(1)
        else
            elementos[elem] ← true                ▷ O(1)
        end if
        Avanzar(it)                               ▷ O(1)
    end while
    i ← i + 1                                     ▷ O(1)
end while
return res
```

Justificación: Ahora aprovecho que los elementos de los conjuntos tienen una cota para crear una suerte de registro de los elementos que van apareciendo. El algoritmo crea un arreglo de N*M de largo (la cantidad de elementos distintos) y lo lleno con falses (todo esto es O(NM)). Luego, para todos los conjuntos y para todos los elementos en cada conjunto, si ya había encontrado ese elemento entonces sé que no son disjuntos, y en caso contrario seteo el registro en true para denotar que ya está presente en la unión de conjuntos