DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Guía 4

Ejercicios obligatorios de la práctica

14 de julio de 2020 — Algoritmos y Estructuras de Datos II

Integrante	LU	Correo electrónico
Rodriguez, Miguel	57/19	mmiguerodriguez@gmail.com

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
https://exactas.uba.ar

Ejercicio 1: Ordenamiento

Se tiene un arreglo A de N conjuntos $A[1], ..., A[N]$. El cardinal de cada conjunto es **a lo sumo** K, es decir $\#(A[i]) \leq K$ para todo $1 \leq i \leq N$. Se desea ordenar el arreglo A para obtener como resultado un arreglo de conjuntos $B[1], ..., B[N]$ de tal modo que se cumpla con la siguiente condicion:

$$B[i] \subseteq B[j] \;\Rightarrow\; i \leq j \qquad \text{para todo } i, j \text{ en el rango } 1..N$$

es decir, si un conjunto está incluido en otro, debe aparecer antes en el arreglo ordenado.

Idea:

1. Armar un bucket en el que vamos a guardar los conjuntos o sus iteradores según sus cardinales.
2. A partir de los buckets, armar una lista ordenada por el cardinal de los conjuntos, ya que, si están ordenados por tamaño, necesariamente se va a cumplir la condición de que si un conjunto está incluido en el otro, su índice va a ser menor.

Algoritmo:

1. Los `conj(nat)` dentro del vector recibido por parámetro están representados por el módulo ConjuntoDeNaturales.
2. Lo primero que hacemos es generar un `vector(vector(itConj(nat)))` en el que el tamaño del vector va a ser K, y el tamaño de los vectores internos va a ser la cantidad de conjuntos que su cardinal sea el índice. Esto va a funcionar como nuestro "bucket". Suponemos que podemos recibir conjuntos con cardinal = 0.
3. Una vez generado este vector, para calcular el resultado insertamos en el `vector(conj(nat))` una copia de cada conjunto a partir de su iterador. Suponemos en este caso que tenemos una función *copy*($itConj$) que a partir de un iterador de ConjuntoDeNaturales retorna una copia de la misma representación en tiempo lineal.

---

ordenarConjuntos(**in** A : `vector(conj(nat))`, **in** k : `nat`) $\rightarrow$ res : `vector(conj(nat))`

```
1:  bucket ← Vacia()
2:  i ← 0
3:  while i ≤ k do                                                    ▷ O(K)
4:      AgregarAtras(bucket, Vacia())
5:      i ← i + 1
6:  end while
7:  for a in A do                                                     ▷ O(N · K)
8:      it ← CrearIt(a)                                               ▷ O(1)
9:      AgregarAtras(bucket[cardinal(a)], it)                         ▷ O(K)
10: end for
11: res ← Vacia()
12: i ← 0
13: while i ≤ k do
14:     for a in bucket[i] do          ▷ Peor caso, un bucket tiene todos los conjuntos. O(N)
15:         AgregarAtras(res, copy(a))          ▷ En total se va a ejecutar N veces. O(K)
16:     end for
17:     i ← i + 1
18: end while
19: return res
```

---

cardinal(**in** A : `conj(nat)`) $\rightarrow$ res : `nat`

```
1:  i ← 0
2:  for a in A do                                                     ▷ O(#(A))
3:      i ← i + 1
4:  end for
5:  return i
```

Complejidad:

1. Crear un `vector(vector(itConj(nat)))` de tamaño K nos cuesta $\mathcal{O}(K)$.
2. Iterar por todos los elementos de la lista de entrada A de tamaño N, calcular el cardinal de cada uno, y devolver su iterador en cada paso nos cuesta $\mathcal{O}(N \cdot K)$.
3. Iterar por el bucket, en principio pareciera que nos cuesta un tiempo cuadrático, pero esto no ocurre, ya que en los buckets vacíos no se van a realizar operaciones, podemos tomarlas como $\mathcal{O}(1)$.
4. La línea 15, que inserta un conjunto en el vector de resultado se va a ejecutar exactamente N veces, y en cada una el costo va a depender de i, ya que, son la cantidad de elementos a insertar (copiar) en el arreglo de salida. En peor caso, se va a ejecutar las N veces con $i = K \Rightarrow \mathcal{O}(N \cdot K)$.

Por lo explicado anteriormente y por el hecho de que *AgregarAtras* se va a ejecutar siempre exactamente N veces y su costo es a lo sumo $\mathcal{O}(K)$, queda demostrado que ordenarConjuntos $\in \mathcal{O}(N \cdot K)$. ■

Ejercicio 2: Dividir y conquistar

Se tiene un arreglo C de N conjuntos $C[1], ..., C[N]$. El cardinal de cada conjunto es **exactamente** M, es decir $\#(C[i]) = M$ para todo $1 \leq i \leq N$.

1. Proponer un algoritmo para determinar si los conjuntos $C[1], ..., C[N]$ son disjuntos dos a dos, es decir si para todo $i \neq j$ en el rango $1...N$ se tiene que $C[i] \cap C[j] = \emptyset$. La complejidad temporal en peor caso del algoritmo debe ser $O(M \cdot N \cdot \log N)$. Justificar la complejidad temporal obtenida.

Idea:

1. Por la forma en la que se puede iterar el módulo ConjuntoDeNaturales, podemos pensar que tenemos una lista grande de tamaño N, que adentro, tiene listas ordenadas de exactamente M elementos.
2. Mi idea es parecida al funcionamiento de un Merge Sort, ya que, estaremos dividiendo el arreglo de conjuntos en partes mas chicas, y al resultado de cada división (que va a estar ordenada), la unimos de forma ordenada.
3. Una vez que tenemos todos los elementos ordenados, lo unico que resta hacer es revisar que no hayan elementos repetidos, ya que si se encuentra algún repetido, quiere decir que hay más de un conjunto que contiene ese elemento (por el hecho de ser conjuntos y que no pueden tener repetidos) y significa que no todos los conjuntos son disjuntos dos a dos.

Algoritmo:

1. Mi forma de realizar el algoritmo parte en dividir en mitades el arreglo de tamaño N, por lo que vamos a tener $\log N$ niveles o llamadas recursivas.
2. El caso base, sera cuando los índices *start* y *end* sean iguales, por lo que vamos a tener un solo conjunto, en ese caso, me armo un nuevo vector e inserto todos sus elementos. Esto nos cuesta exactamente $\Theta(M)$ ya que insertar en un vector es $\mathcal{O}(1)$ e iterar por un único conjunto de exactamente M elementos nos cuesta $\mathcal{O}(M)$. Notar que este vector va a estar ordenado por la forma en la que se itera un ConjuntoDeNaturales.
3. Para la parte de dividir, llamo recursivamente a la función sonDisjuntosAux y así obtengo las dos mitades ordenadas ya no como un conjunto sino como un `vector(nat)`.
4. El paso de conquistar, es cuando hacemos el merge de las dos mitades ordenadas, que en tiempo lineal, nos va a retornar la unión ordenada. El costo de copia por el pasaje de los parámetros está acotado por la complejidad de la función.
5. Finalmente, a partir del resultado de la función hayRepetidos decidimos si los conjuntos son disjuntos dos a dos.

sonDisjuntos(**in** A : `vector(conj(nat))`) $\rightarrow$ res : `bool`

```
1: resVector ← sonDisjuntosAux(a, 0, Longitud(a) − 1)        ▷ O(N · M · log N)
2: res ← ¬hayRepetidos(resVector)                            ▷ O(N · M)
3: return res
```

sonDisjuntosAux(**in** A : `vector(conj(nat))`, **in** $start$: `nat`, **in** end : `nat`) $\rightarrow$ res : `vector(nat)`

```
1: if start = end then
2:     res ← Vacia()
3:     for x in a[start] do                                  ▷ O(M)
4:         AgregarAtras(res, x)                              ▷ O(1)
5:     end for
6: end if
7: half ← (end + start)/2
8: s1 ← sonDisjuntosAux(a, start, half)                      ▷ T(N/2)
9: s2 ← sonDisjuntosAux(a, half + 1, end)                    ▷ T(N/2)
10: res ← merge(s1, s2)                                      ▷ O(N · M)
11: return res
```

merge(**in** A : `vector(nat)`, **in** B : `vector(nat)`) $\rightarrow$ res : `vector(nat)`

```
1: res ← Vacia()
2: i ← 0
3: j ← 0
4: k ← Longitud(A) + Longitud(B)
5: while i < k do                                            ▷ O(|A| + |B|)
6:     if j ≥ Longitud(B) ∨ (i < Longitud(A) ∧ A[i] < B[i]) then
7:         AgregarAtras(res, A[i])                           ▷ O(1)
8:         i ← i + 1
9:     else
10:        AgregarAtras(res, B[j])                           ▷ O(1)
11:        j ← j + 1
12:    end if
13:    k ← k + 1
14: end while
15: return res
```

hayRepetidos(**in** A : `vector(nat)`) $\rightarrow$ res : `vector(nat)`

```
1: i ← 0
2: for i < Longitud(A) − 1 do                                ▷ O(|A|)
3:     if A[i] = A[i + 1] then
4:         return true
5:     end if
6:     i ← i + 1
7: end for
8: return false
```

Complejidad: En principio, si tengo que escribir la complejidad como una recurrencia, podemos notar que el caso base es cuando $N = 1$. En este caso, solamente tenemos que recorrer el único conjunto que hay de tamaño $M \Rightarrow \Theta(M)$. Para $N > 1$ el caso es distinto, ya que tenemos una función recursiva que divide problema en dos mitades (el arreglo de tamaño N) y además con el resultado de estas llamadas recursivas, realiza un merge en $\mathcal{O}(\frac{M \cdot N}{2} + \frac{M \cdot N}{2}) \in \mathcal{O}(M \cdot N)$

ya que tenemos que tener en cuenta que cada mitad tiene $\frac{M \cdot N}{2}$ elementos. La recurrencia a la que llego es:

$$T(N) = \begin{cases} \Theta(M) & si \quad N = 1 \\ \\ 2 \cdot T(\frac{N}{2}) + \mathcal{O}(N \cdot M) & si \quad N > 1 \end{cases}$$

Para demostrar que efectivamente esta complejidad es la pedida por el ejercicio, lo solucionaremos como una recurrencia típica. Veamos los parámetros que requiere la demostración:

1. $a = 2$. Cantidad de subproblemas a resolver.
2. $b = M$. Costo de unión.
3. $c = 2$. Particiones del problema.
4. $d = \log_c a = 1$.

Demostrando con la solución de la recurrencia típica[1]:

$$T(N) = a \cdot T(\frac{N}{c}) + b \cdot N^d$$

$$\text{Tomando } N = c^k$$

$$= a \cdot T(c^{k-1}) + b \cdot c^{dk}$$

$$= a^2 \cdot T(c^{k-2}) + a \cdot b \cdot c^{d(k-1)} + b \cdot c^{dk}$$

$$\dots$$

$$= a^j \cdot T(c^{k-j}) + \sum_{i=0}^{j-1} a^i \cdot b \cdot c^{d(k-i)}$$

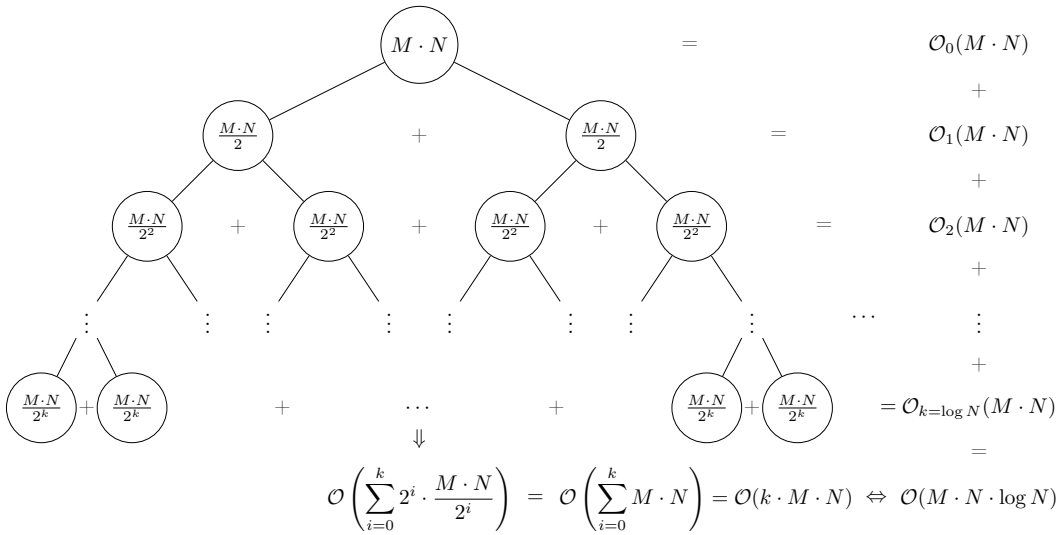
$$\text{Hasta que } c^{k-j} = 1, \text{ o sea, } j = \log_c N$$

$$= b \cdot N^d \cdot \sum_{i=0}^{\log N} \left(\frac{a}{c^d}\right)^i$$

$$\text{En nuestro caso, ocurre que } a = 2,\ b = M,\ c = 2,\ d = 1,\ \text{luego}$$

$$T(N) = M \cdot N \cdot \sum_{i=0}^{\log N} \left(\frac{2}{2^1}\right)^i = M \cdot N \cdot \sum_{i=0}^{\log N} 1 \in \mathcal{O}(M \cdot N \cdot \log N)$$

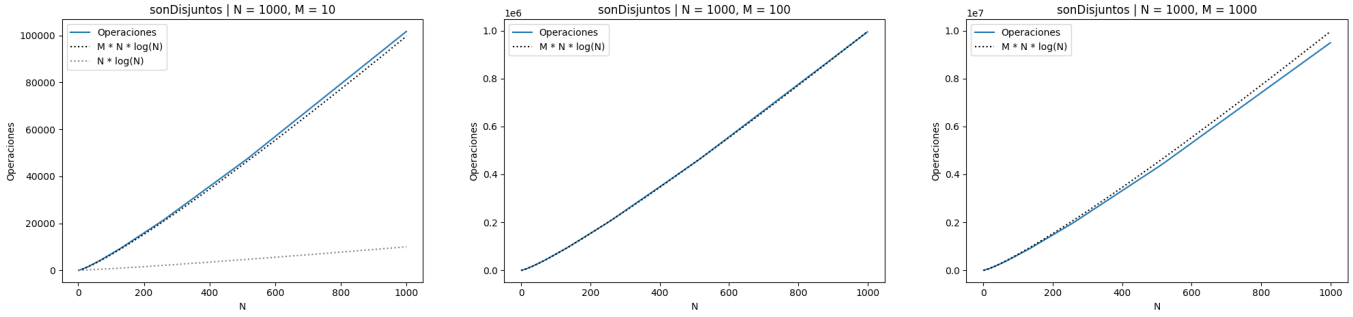
Veamos que esta complejidad se puede notar también con un árbol de recursión[2]:



[1] https://campus.exactas.uba.ar/pluginfile.php/198992/mod_resource/content/2/TeoricaDC.pdf, 8
[2] https://campus.exactas.uba.ar/pluginfile.php/200196/mod_resource/content/2/P10_DivideAndConquer-Handout.pdf, 14

Luego, queda demostrado que SONDISJUNTOS $\in \mathcal{O}(M \cdot N \cdot \log N)$. ■

EXTRA: Al ser este un examen distinto por ser a distancia me pareció interesante agregarle una sección más. Aparte de lo ya demostrado, hice la implementación de estas funciones en C++ junto con unos gráficos que comparan la complejidad calculada anteriormente con la cantidad de operaciones realizadas por el algoritmo implementado. Los resultados obtenidos fueron los esperados. Resulta que M es una constante, pero a la hora de la complejidad es muy importante y hay que tenerla en cuenta ya que es la cantidad de elementos que vamos a tener en cada conjunto. Realicé 3 experimentos, con $N = 1000$ y $M = 10$, 100 y 1000. Los gráficos obtenidos son los siguientes:



Como se puede notar en los gráficos, la cantidad de operaciones que realiza el algoritmo está en el orden de $\mathcal{O}(M \cdot N \cdot \log N)$ y además, al aumentar el M multiplicando por 10, se puede ver que la cantidad de operaciones también se multiplica por 10.

El código, datos y gráficos se encuentran en las carpetas `scripts`, `data` e `img`.

2. (optativo) Suponiendo que los elementos de todos los conjuntos están en el rango comprendido entre 1 y el producto $M \cdot N$, proponer un algoritmo para resolver el mismo problema con complejidad temporal en peor caso $O(M \cdot N)$. Notar que este algoritmo sirve para determinar si los conjuntos $C[1], ..., C[N]$ constituyen una partición del conjunto $\{1, 2, ..., M \cdot N - 1, M \cdot N\}$. Justificar la complejidad temporal obtenida

Idea:

1. Al estar los conjuntos acotados entre 1 y $M \cdot N$, para que los conjuntos sean disjuntos dos a dos, entonces debe ocurrir que todos los naturales entre 1 y $M \cdot N$ estén comprendidos en la lista de conjuntos.
2. Haciendo un tipo de counting sort para ver si todos los elementos entre 1 y $M \cdot N$ están en la unión de todos los conjuntos podemos determinar si son disjuntos dos a dos.
3. Esto se debe a que si no están todos los elementos, entonces debe ocurrir que algún natural está repetido en algún conjunto ya que los conjuntos tienen todos exactamente M elementos y el total de elementos es $M \cdot N$.

Algoritmo:

1. Nos generamos un `vector(bool)` de tamaño $M \cdot N$ inicializado con todo en $false$.
2. Iteramos por todos los elementos de todos los conjuntos, y por cada natural, seteamos en $true$ su aparición en el vector generado anteriormente. Particularmente le restamos 1 a cada valor ya que los valores van entre 1 y $M \cdot N$ mientras que los índices de nuestro arreglo están entre 0 y $M \cdot N - 1$.
3. Una vez seteado todos los valores en $true$ de los elementos que se encuentran, resta revisar que ningún valor sea $false$, ya que en ese caso ese número no estaría en ningún conjunto, lo cual implica que hay alguno con repetidos y no ocurrirá que todos sean disjuntos dos a dos.
4. Si todos los números estaban en los conjuntos entonces el vector tiene todos los valores en $true$ y retornamos que son disjuntos dos a dos.

sonDisjuntosAcotado(**in** A : `vector(conj(nat))`, **in** m : `nat`) $\rightarrow$ res : `bool`

```
1:  n ← Longitud(A)
2:  nm ← n × m
3:  countingVec ← Vacia()
4:  i ← 0
5:  for i < nm do                                              ▷ O(M · N)
6:      AgregarAtras(countingVec, false)                       ▷ O(1)
7:      i ← i + 1
8:  end for
9:  i ← 0
10: for i < n do                                               ▷ O(M · N)
11:     for a in A[i] do                                       ▷ O(M)
12:         countingVec[a − 1] ← true                          ▷ O(1)
13:     end for
14:     i ← i + 1
15: end for
16: i ← 0
17: for i < nm do                                              ▷ O(M · N)
18:     if countingVec[i] = false then
19:         return false
20:     end if
21:     i ← i + 1
22: end for
23: return true
```

Complejidad:

1. Generar un vector de tamaño $M \cdot N$ con todos los elementos en $false$ nos cuesta $\mathcal{O}(M \cdot N)$.
2. Iterar por todos los elementos de todos los conjuntos tambíen nos cuesta $\mathcal{O}(M \cdot N)$. Setear un índice en $true$ tiene costo $\mathcal{O}(1)$.
3. Finalmente, iterar nuevamente por el vector nos cuesta en peor caso $\mathcal{O}(M \cdot N)$ ya que ocurre cuando todos los valores estén en $true$ y sean conjuntos disjuntos dos a dos.

Luego, queda demostrado que para conjuntos acotados, sonDisjuntosAcotado $\in \mathcal{O}(M \cdot N)$. ■