Primer Problema DAA

Karlos Alejandro Alfonso Rodríguez
Karel Camilo Manresa Leon

April 3, 2023

Problema

Leandro es profesor de programación. En sus ratos libres, le gusta divertirse con las estadísticas de sus pobres estudiantes reprobados. Los estudiantes están separados en n grupos. Casualmente este año, todos los estudiantes reporbaron alguno (uno y solo uno) de los dos exámenes finales P (POO) y R (Recursividad).
Esta tarde, Leandro decide entretenerse separando a los estudiantes suspensos en conjuntos de tamaño k que cumplan lo siguiente: En un mismo conjunto, todos los estudiantes son del mismo grupo i ($1 \leq i \leq n$) o suspendieron por el mismo exámen P o R.
Conociendo el grupo y prueba suspensa de cada estudiante y el tamaño de los conjuntos, ayude a Leandro a saber cuántos conjuntos de estudiantes suspensos puede formar.

Problema en términos matemático

Se tiene una lista G, donde $G[i]$ indica el conjunto al que pertenece el entero i. Además se tiene otra lista binaria F, donde $F[i]$ indica la presencia o no de una característica en el entero i. De modo tal que para un entero i se tiene $G[i]$ y $F[i]$, que indican: conjunto al que pertenece i y presencia o no de una característica para i, respectivamente.
Dado un entero $k \geq 1$ y las listas G y F, se desea conocer cuantos conjuntos de tamaño k se pueden formar de modo que $\forall i, j \in k_t$ se cumpla $G[i] = G[j] \vee F[i] = F[j]$.
Sea:

- $|G|$ = n.
- P conjunto de todos los elementos que cumplen $F[i] = 1$.
- R conjunto de todos los elementos que cumplen $F[i] = 0$.

Primer enfoque

El primer enfoque pensado para la solución del problema es hacer un *Backtrack*, que compruebe todas las posibles formas de construir subconjuntos disjuntos válidos ($G[i] = G[j] \vee F[i] = F[j]$) de tamaño máximo k. Luego se selecciona como respuesta la forma que contenga la mayor cantidad de subconjuntos de tamaño k.
La cantidad máxima de subconjuntos de tamaño k que se pueden realizar con n elementos es $\lfloor \frac{n}{k} \rfloor$. En este caso debemos añadir dos conjuntos más ya que el resto resultante de hacer parte entera por debajo, en el peor caso tiene elementos de ambas característica, y deben agruparse en conjuntos diferentes.

Correctitud

Esta solución es correcta ya que comprueba todas las formas posibles de construir los subconjuntos y se queda con la mejor.

Pseudocódigo

```
1  def backtrack(element, G, F, k, subsets):
2      if element == |G|:
3          return KSizeSets(subsets, k)
4
5      sol = 0
6      for i from 0 to |subsets|:
7          if |subsets[i]| == k:
8              continue
9
10         if Insert(subsets[i], element):
11             sol = max(sol, backtrack(element+1, G, F, k,
                   subsets))
12             subsets[i].Remove(element)
13             if sol == |G|//k:
14                 break
15
16     return sol
```

Complejidad temporal

En el algoritmo recursivo anterior tenemos que $f(n) = 2k$ ya que *Insert* y *Remove* recorren k elementos cada uno. Luego sea $m = |G|$, tenemos que se realizan $\lfloor \frac{m}{k} \rfloor$ llamados a la recursividad. Cada llamado se realiza con un elemento menos, por lo que $T(n) = \lfloor \frac{m}{k} \rfloor T(n-1) + 2k$.
Entonces:

$$\begin{aligned} T(n) &= 2k + \lfloor \frac{m}{k} \rfloor (2k + \lfloor \frac{m}{k} \rfloor (2k + \lfloor \frac{m}{k} \rfloor (\dots))) \\ T(n) &= (\frac{m}{k})^m + 2(\frac{m}{k})^{m-1} + \cdots + 2k \\ T(n) &\leq (\frac{m}{k})^m + 2(\frac{m}{k})^m + \cdots + 2k^m = O(\frac{m^m}{k^m}) \end{aligned}$$

Segundo enfoque

El segundo enfoque pensado para la solución del problema es hacer un algoritmo greedy. Primeramente se intenta crear todos los subconjuntos posibles priorizando la existencia o no de la característica; o sea, la solución será la

cantidad de subconjuntos de tamño k que se pueden formar con elementos que cumplan la característica más la cantidad de subconjuntos que se pueden formar con elementos que no la cumplan. Luego se comprueba si con los elementos restantes se puede formar otro subconjunto basado en el conjunto inicial al que pertenecen dichos elementos.

Correctitud

Para demostrar la correctitud de esta solución, se debe demostrar que siempre que se creen los subconjuntos basados en las características, se obtendrá un total de $\lfloor \frac{n}{k} \rfloor - 1$ o $\lfloor \frac{n}{k} \rfloor$ subconjuntos de tamaño k. En caso de obtenerse $\lfloor \frac{n}{k} \rfloor - 1$ subconjuntos y la respuesta correcta ser $\lfloor \frac{n}{k} \rfloor$, se debe probar que siempre hay una manera de armar un subconjunto basado en el conjunto inicial de los elementos. Sea $p = |P|$ y $r = |R|$, además p' y r' son la cantidad de conjuntos de tamaño k que se pueden formar con los elementos de P y R respectivamente. Demostremos que $\lfloor \frac{n}{k} \rfloor - 1 \leq p' + r' \leq \lfloor \frac{n}{k} \rfloor$

Proof. Se sabe que

$$p + r = n$$

luego si aplicamos el algoritmo de división sobre p y r con k de divisor se obtiene

$$kp' + p_r + kr' + r_r = n$$

donde p' y r' son precisamente la cantidad de subconjuntos de tamaño k que se pueden formar con los conjuntos P y R respectivamente. Si multiplicamos la ecuación anterior por k obtenemos

$$p' + r' + \frac{p_r + r_r}{k} = \frac{n}{k}$$

si aplicamos la función parte entera por debajo a la ecuación

$$\lfloor p' + r' + \frac{p_r + r_r}{k} \rfloor = \lfloor \frac{n}{k} \rfloor$$

y como p' y r' son enteros se cumple que $\lfloor p' \rfloor = p'$ y $\lfloor r' \rfloor = r'$, por tanto

$$p' + r' + \lfloor \frac{p_r + r_r}{k} \rfloor = \lfloor \frac{n}{k} \rfloor$$

Si $0 \leq \lfloor \frac{p_r + r_r}{k} \rfloor \leq 1$ entonces $\lfloor \frac{n}{k} \rfloor - 1 \leq p' + r' \leq \lfloor \frac{n}{k} \rfloor$. Demostremos entonces que $0 \leq \lfloor \frac{p_r + r_r}{k} \rfloor \leq 1$.

Proof. Como p_r y r_r son enteros positivos, su suma será un entero positivo, por tanto al hacer $\frac{p_r + r_r}{k}$ se obtiene un número positivo ya que k es un entero positivo también. Luego al aplicarle parte entera por debajo se obtiene un entero positivo, luego $0 \leq \lfloor \frac{p_r + r_r}{k} \rfloor$.

Se tiene además que

$$\begin{aligned} p_r &< k \\ r_r &< k \\ p_r + r_r &< 2k \\ \frac{p_r + r_r}{k} &< 2 \\ \lfloor \frac{p_r + r_r}{k} \rfloor &\leq 1 \end{aligned}$$

Luego $0 \leq \lfloor \frac{p_r + r_r}{k} \rfloor \leq 1$. □

Entonces se cumple que $\lfloor \frac{n}{k} \rfloor - 1 \leq p' + r' \leq \lfloor \frac{n}{k} \rfloor$. □

Ahora se debe demostrar que siempre que la respuesta arrojada por el algoritmo anterior sea $\lfloor \frac{n}{k} \rfloor - 1$ y la correcta sea $\lfloor \frac{n}{k} \rfloor$ se puede formar un nuevo subconjunto con el resto de los conjuntos P y R.
La forma de hacer esto pensada fue: por cada conjunto inicial g si se cumple que $min(g[0], r_r) + min(g[1], p_r) \geq k$ entonces se puede formar otro subconjunto válido de tamaño k en ese grupo, y será el último (en caso contrario se puedieran formar al menos dos nuevos subconjuntos, lo cual es una contradicción ya que la cantidad máxima posible es $\lfloor \frac{n}{k} \rfloor$), donde $g[0]$ es la cantidad de elementos del grupo g que no cumplen la característica y $g[1]$ los de g que la cumplen. A pesar de que esta forma de armar un nuevo subconjunto es válida, no es factible aplicarla luego de haber agrupado por característica, ya que no necesariamente con los elementos restantes se podrá lograr. A continuación un caso que prueba esto:
Sea $G = [1, 1, 2, 2, 3, 3]$, $F = [0, 1, 0, 1, 0, 1]$ y $k = 2$. Para este caso es fácil comprobar que la respuesta es $sol = 3$. Si seguimos el algoritmo planteado, primeramente agrupamos por característica, obteniendo 2 subconjuntos de tamaño 2. Luego si se intenta formar un nuevo subconjunto basado en los grupos iniciales no es posible ya que cada resto pertenece a un grupo distinto. Por tanto este enfoque no es válido.

Pseudocódigo

```
1   def solve(G, F, k):
2       sol = 0
3
4       p,r = Count 1s,0s in F
5
6       sol += p//k
7       sol += r//k
8
9       p_r = p%k
10      r_r = r%k
11
12      if p_r + r_r < k:
13          return sol
14
15      fbs = FeaturesBySets(G, F)
16
17      for g in fbs:
18          if min(g[0], p_r) + min(g[1], r_r) >= k:
19              sol+=1
20
21              return sol
```

La función $FeaturesBySets(G, F)$ crea un diccionario donde dado un grupo inicial devuelve la cantidad de elementos que tiene que cumplen la característica y los que no la cumplen.

Complejidad temporal

La complejidad temporal de ejecutar la línea **4** es $O(|F|) = O(|G|) = O(n)$. Luego ejecutar las líneas **2, 6-13** tiene costo $O(1)$. El diccionario fbs se puede crear en tiempo $O(|G|) = O(n)$; y hacer el ciclo de las líneas **17-19** tiene costo $O(cantGrupos)$ y como $cantGrupos \leq n$, entonces tiene costo $O(n)$. Finalmente $T(n) = 3O(n) = O(n)$.

Tercer enfoque

El tercer enfoque consiste en construir una solución dinámica. Sea g la cantidad de grupos iniciales que contienen a los elementos, sean a y b listas que determinan la cantidad de elementos que cumplen la característica y los que no, en cada uno de los grupos. De modo que $a[i]$ devuelve la cantidad de elementos que hay en el grupo i que cumplen la característica, y $b[i]$ la cantidad que no la cumplen en el grupo i. Llamémosle elementos de tipo 1 a los que cumplen la característica y tipo 2 a los que no la cumplen.

Sea $dp[i, j]$ una matriz booleana que denota si podemos tener j elementos restantes (un elemento es restante si no ha sido ubicado en ningún subconjunto válido) de tipo 1 después de haber pasado por los primeros i grupos. No es necesario tener en cuenta los elementos de tipo 2; ya que, sabiendo que hay j de tipo 1 restantes podemos decir que hay $(n - j)\%k$ elementos de tipo 2 restantes.

Correctitud

Pseudocódigo

```
1   def sol(g, k, a, b):
2       totA = Sum(a)
3       totB = Sum(b)
4
5       dp = [g+1,k]
6       dp[0,0] = true
7
8       for i from 1 to g+1:
9           for j from 0 to k:
10              dp[i,j] = dp[i-1, (j-a[i]%k)%k]
11              for x from 0 to min(k-1, a[i]+1):
12                  if (a[i] - x)%k + b[i] >= k:
13                      dp[i,j] = dp[i,j] or dp[i-1, (j-x)%k]
14
15      result = 0
16      for i from 0 to k:
17          if dp[g,i]:
18              result = max(result, (totA + totB - i)//k)
19
20      return result
```

Complejidad temporal

El costo de calcular $totA$ y $totB$ es $O(g)$. Luego el costo del ciclo de la línea **8** es $O(g)$, el ciclo anidado en la línea **9** tiene costo $O(k)$, por lo que hasta ahora el costo va siendo $O(gk)$. Además el costo del ciclo anidado en la línea **11** tiene costo $O(k)$, ya que $k - 1 < k$ y al ser la función min se garantiza que se tomará un valor menor o igual que $k - 1$. Luego el costo temporal de ejecutar los tres ciclos es $O(gk^2)$. Luego para calcular el resultado se realiza un ciclo con costo $O(k)$. Por tanto el costo temporal total del algoritmo es $O(gk^2)$.

Tester

Generador de casos de pueba

Pseudocódigo