Secuencias Mayoritarias

Alejandro Morales Contreras[1]

[1]Departamento de Ingeniería de Sistemas, Pontificia Universidad Javeriana
Bogotá, Colombia
a.moralesc@javeriana.edu.co

11 de agosto de 2022

Índice

1. Enunciado del problema

Escriba dos algoritmos para solucionar el problema: "indicar si una secuencia es mayoritaria y, en caso de serlo, informar el elemento mayoritario". Una secuencia es mayoritaria cuando al menos la mitad de sus elementos tienen el mismo valor, por ejemplo:

$$S = \langle 1, 2, 10, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 33, 4, 15, 1, 3, -1, 1, 1, 1, 1, 1\rangle$$

es mayoritaria y el elemento mayoritario es 1. Los dos algoritmos serán:

1. Iterativo.
2. Basado en la estrategia "dividir-y-vencer".

2. Formalización del problema

El problema de secuencias mayoritarias se define a partir de:

1. una secuencia S de elementos $a \in \mathbb{T}$,
2. una relación de igualdad $a = b\ \forall a, b \in S$ y

3. una relación de orden parcial $a < b \; \forall a, b \in S$

retornar una pareja (x, y) donde x representa si la secuencia es mayoritaria o no, y y representa el elemento mayoritario.

- Entradas:
  - $S = \langle a_i \in \mathbb{T} \rangle \mid 1 \leq i \leq n \wedge n \geq 1$.
  - $a = b \in \mathbb{T} \times \mathbb{T}$, una relación de igualdad.
  - $a < b \in \mathbb{T} \times \mathbb{T}$, una relación de orden parcial.
- Salidas:
  - $x \begin{cases} 1, & \text{if } \exists y \in S \mid \sum_{i=1}^{n} [S_i = y] > \lfloor |S| \div 2 \rfloor, \\ 0, & \text{else.} \end{cases}$
  - $y \in S \cup \langle 0 \rangle$

3. Algoritmos de solución

3.1. Iterativo

3.1.1. Idea general de la solución

Una primera solución ingenua que podría surgir para este problema, sería el de recorrer la secuencia uno a uno, y por cada elemento, volver a recorrer la secuencia para contar cada vez que aparece. Esta solución tendría una complejidad $O(n^2)$, ya que consistiría de dos ciclos anidados.

De aquí nace la idea de utilizar una estructura de datos que facilite el conteo de los datos: los mapas. El mapa permitiría recorrer toda la secuencia, e ir contando cuántas veces aparece cada elemento, colocando el elemento como llave y su valor como la cantidad de veces que aparece. Finalmente, solo bastaría recorrer las llaves del mapa y encontrar una que aparezca más de $\lfloor |S| \div 2 \rfloor$ veces.

Cabe resaltar que dependiendo de la implementación de mapa que se utilice, la complejidad variaría para las inserciones y búsquedas que se realizan. Un análisis de complejidad más detallado se hace en la sección 3.1.3.

3.1.2. Escritura del algoritmo

La solución se presenta en el algoritmo 1 donde:

- EXISTS(M, k) retorna 1 si k es una llave que existe en el mapa M o 0 si no existe
- KEYS(M) retorna un arreglo con todas las llaves definidas en el mapa M

Algoritmo 1 Determinar secuencia mayoritaria de forma iterativa.

```
1: procedure IsMajorityIterative(S)
2:     let M < k, v > be a map
3:     for i ← 1 to |S| do
4:         if Exists(M, S_i) then
5:             M[S_i] ← M[S_i] + 1
6:         else
7:             M[S_i] ← 1
8:         end if
9:     end for
10:    K ← Keys(M)
11:    i ← 0
12:    is_majority ← 0 ∧ v ← 0
13:    while i < |K| ∧ is_majority = 0 do
14:        i ← i + 1
15:        if M[K_i] > ⌊|K| ÷ 2⌋ then
16:            is_majority ← 1
17:            v ← M[K_i]
18:        end if
19:    end while
20:    return is_majority, v
21: end procedure
```

3.1.3. Análisis de complejidad

Como se mencionó anteriormente, la implementación de mapa que se utilice varía la complejidad que se obtiene con esta solución.

En C++, se pueden analizar dos implementaciones de mapas que existen en la librería estándar: *map* y *unordered_map*. El primero está implementado utilizando un árbol roji-negro, y su complejidad de inserción y búsqueda es $O(\log n)$ tanto para el peor de los casos como el promedio. Debido que para la solución implementada, en el peor de los casos todos los elementos se repiten una sola vez y terminarían apareciendo una sola vez en el mapa, se recorrerían todas las llaves disponibles ($|S|$) y se buscarían dentro del mapa cada vez ($\log |S|$). Finalmente, la complejidad sería de $O(|S| \log |S|)$.

Para la segunda implementación disponible, se utilizan hash tables, y su complejidad de inserción y búsqueda es $\Theta(1)$ en el caso promedio y $O(n)$ en el peor de los casos. Sin embargo, el peor de los casos suele ocurrir solo cuando se utilizan llaves poco comunes. Siguiendo la misma lógica analizada en el punto anterior, se tendría una complejidad de $\Theta(|S|)$ en el caso promedio y $O(|S|^2)$ en el peor de los casos.

En Java, existen dos implementaciones que siguen la misma lógica: *TreeMap* y *HashMap* respectivamente. Por su parte, en Python los diccionarios siguen la misma lógica de los *unordered_map*.

3.1.4. Invariante

La invariante es que existen $1 \leq i \leq |S|$ llaves en el mapa, y al menos una de ellas tiene un valor asociado mayor a $\lfloor |S| \div 2 \rfloor$ o la secuencia no es mayoritaria.

- Inicio: $\text{Keys}(M) = \emptyset$
- Procesamiento: $M[S_i] \begin{cases} M[S_i] + 1, & \text{if Exists}(M, S_i) \\ 1, & \text{else} \end{cases}$
- Fin: $\exists x \in S \mid M[x] > \lfloor |S| \div 2 \rfloor \vee x \notin S$

3.2. Dividir y vencer

3.2.1. Idea general de la solución

Para aplicar la estrategia dividir y vencer, el hecho de que los elementos no traigan algún tipo de relación entre ellos dificulta el proceso de recorrer la secuencia para llegar al problema más sencillo. Sin embargo, una manera fácil de crear esta relación sería ordenando la secuencia con un algún algoritmo de ordenamiento adecuado para tener los elementos iguales seguidos entre sí.

Después de ordenados los datos, se puede ver que para que se cumpla la condición de que exista un elemento mayoritario que aparezca al menos $\lfloor |S| \div 2 \rfloor$ veces, este **necesariamente** debe ubicarse una vez en la posición de la mitad después del ordenamiento. Entonces podemos asumir que: a) la secuencia es mayoritaria y b) que el elemento que se ubique en la mitad de la secuencia es el elemento mayoritario. Ahora solo basta probar que el elemento mayoritario aparece más de la mitad de veces. Si esto no se cumple, la secuencia no es mayoritaria.

Podemos representar esta situación en la figura 1, en donde tenemos una secuencia S de tamaño $|S| = 13$ con punto medio en el índice 7. Decimos entonces que este elemento e es el mayoritario y los otros dentro de la secuencia los podemos marcar con X ya que no son relevantes si no son iguales a e.

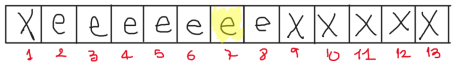

Figura 1: Secuencia inicial

El proceso consiste ahora en hacer una especie de búsqueda binaria hacia la izquierda y derecha de este primer pivote, con el fin de determinar en qué posición se encuentran el primer y último elemento mayoritario. Después de conocer esto, solo basta restar estas posiciones para saber cuántos de estos elementos hay.

Para hacer la búsqueda, determinamos el inicio y fin de la sub-secuencia, calculamos el pivote y damos un salto hacia este. Parados en este, determinamos si es o no el elemento mayoritario (equivalencia). Si lo es, nos seguimos moviendo en esa dirección. Sin embargo, si no, debemos movernos en dirección contraria porque nos pasamos. El algoritmo acaba cuando el inicio y el fin es el mismo elemento, que sería la posición del primer (o último) elemento mayoritario.

Un ejemplo de búsqueda se presenta en la figura 2.

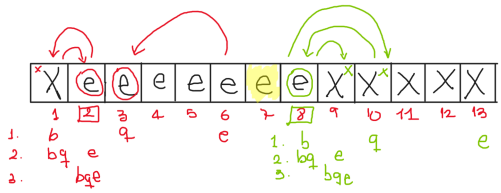

Figura 2: Proceso de búsqueda

Hacia la izquierda, el primer pivote es $(1+6)/2 = 3$ y como en esta posición está el elemento mayoritario, seguimos hacia la izquierda. El segundo pivote es $(1+2)/2 = 1$ y como en esta posición no está el elemento mayoritario, cambiamos hacia la derecha. El tercer pivote es $(2+2)/2 = 2$ y en esta posición está el elemento mayoritario.Como en esta posición el inicio y fin es el mismo, hemos acabado. La última posición conocida del elemento mayoritario fue el índice 2.

Hacia la derecha, el primer pivote es $(8+13)/2 = 10$ y como en esta posición no está el elemento mayoritario, vamos hacia la izquierda. El segundo pivote es $(8+9)/2 = 8$ y como en esta posición está el elemento mayoritario, vamos hacia la derecha. El tercer pivote es $(9+9)/2 = 9$ y en esta posición no está el elemento. Como en esta posición el inicio y el fin es el mismo, hemos acabado. La última posición conocida del elemento mayoritario fue el índice 8.

Finalmente, se hace la operación $8 - 2 + 1$ y se determina que el elemento aparece 7 veces. Como 7 es mayor que $|S| \div 2$, entonces la secuencia es mayoritaria y el elemento mayoritario es e.

3.2.2. Escritura del algoritmo

Donde:

- SORT(S) es un algoritmo que ordena la secuencia

Algoritmo 2 Determinar secuencia mayoritaria de forma dividir y vencer.

```
1: procedure IsMajorityDivideAndConquer(S)
2:     Sort(S)
3:     v ← S_{⌊(|S|+1)÷2⌋}
4:     i ← FindMajorityLimit(S, v, 1, ⌊(|S| + 1) ÷ 2⌋ − 1, −1, ⌊|S| ÷ 2⌋)
5:     j ← FindMajorityLimit(S, v, ⌊(|S| + 1) ÷ 2⌋ + 1, |S|, 1, ⌊|S| ÷ 2⌋)
6:     count ← j − i + 1
7:     is_majority ← 1
8:     if count ≤ ⌊|S| ÷ 2⌋ then
9:         is_majority ← 0
10:        v ← 0
11:    end if
12:    return is_majority, v
13: end procedure
```

Algoritmo 3 Encontrar posición límite del elemento mayoritario.

```
1: procedure FindMajorityLimit(S, value, b, e, dir, last)
2:     if b=e then
3:         if S_q = value then
4:             last ← b
5:         end if
6:         return last
7:     else
8:         q ← ⌊(b + e) ÷ 2⌋
9:         if S_q = value then
10:            if dir = 1 then
11:                return FindMajorityLimit(S, value, q + 1, e, dir, q)
12:            else
13:                return FindMajorityLimit(S, value, b, q − 1, dir, q)
14:            end if
15:        else
16:            if dir = 1 then
17:                return FindMajorityLimit(S, value, b, q − 1, dir, last)
18:            else
19:                return FindMajorityLimit(S, value, q + 1, e, dir, last)
20:            end if
21:        end if
22:    end if
23: end procedure
```

3.2.3. Análisis de complejidad

El algoritmo de solución consite en dos partes imporantes: el ordenamiento de la secuencia y la búsqueda de los límites, $O = O_1 + O_2$, en donde solo se preserva la de mayor complejidad. La complejidad del ordenamiento está dada por el algoritmo escogido para la secuencia, entonces $O(\text{SORT}(|S|)$.

Por su parte, la búsqueda de los límites es un algoritmo dividir y vencer, y su complejidad puede ser expresada así y hallada a través del teorema maestro:

$$T(n) = T(\frac{n}{2}) + O(1)$$

Utilizando las tres ecuaciones, se encuentra:

1. $1 = n^{\log_b(a-\epsilon)} = n^{\log_2(1-\epsilon)} \wedge \epsilon = 0$
2. $1 = n^{\log_b a} \log_2^k n = n^{\log_2 1} \log_2^k n \wedge k = 0 \rightarrow O(n^{\log_2 1} \log_2^{k+1} n) = O(\log_2 n)$
3. $1 = n^{\log_b(a-\epsilon)} = n^{\log_2(1+\epsilon)} \wedge \epsilon = 0$

Por ende la complejidad de la búsqueda de los límites es $O(\log |S|)$. Debido a que $O(\textsc{Sort}(|S|) > O(\log |S|)$ siempre, incluso para la cota inferior del ordenamiento, la complejidad dependará únicamente de la complejidad del algoritmo de ordenamiento escogido.

3.2.4. Invariante

La invariante es que después del ordenamiento, el elemento mayoritario existe una vez en la posición de la mitad y en las posiciones $i, j \mid i, j \in [1, |S|] \wedge j - i + 1 > \lfloor |S| \div 2 \rfloor$ o la secuencia no es mayoritaria.