Problemas de Diseño y Análisis de Algoritmos

Olivia González Peña
Juan Carlos Casteleiro Wong
Adrian Tubal Páez Ruiz

2018 - 2019

Problema 3. Subintervalos con sumas dadas

Dado un arreglo de números enteros y un valor k, se quiere determinar cuántos bloques del arreglo suman exactamente k y cuántos bloques del arreglo tienen una suma cuyo valor se encuentra en el intervalo [a, b], donde a y b son valores especificados en la entrada. Escriba un programa que resuelva esto de forma eficiente.

Solución 1: Fuerza Bruta

Una primera solución es revisar todos los posibles bloques del arreglo, calculando la suma de sus elementos, si esta está contemplada en el intervalo $[a, b]$ entonces sería una solución para este caso, y de forma análoga para los bloques cuya suma sea k. Para ello, contamos con dos índices que indicarían el inicio y final de los bloques, fijando el primero y haciendo correr el segundo a partir de esta posición hasta el final del arreglo, pudiendo iniciar los bloques en las n posiciones del mismo.

Correctitud

La correctitud de este algoritmo se basa en la idea de que se van a revisar todos los bloques que se pueden encontrar en el arreglo, cuya única restricción es que los elementos que los componen, en caso de que sean más de uno, sean consecutivos. La consecutividad se garantiza con la idea mencionada anteriormente de ir calculando la suma de todos los elementos desde un punto a otro.

Interpretaremos un intervalo como válido si el inicio del mismo corresponde a una posición que precede o coincide con la correspondiente al final.

Partiendo de que, todo bloque tiene una posición de inicio i, y una donde termina j, tal que $\forall i, j$ se cumple que $1 \leq i \leq j \leq n$, el hecho de recorrer con i todas las posiciones del arreglo desde la primera hasta la última, y a partir de cada una de ellas hacer correr j desde tomar este valor hasta el final, definimos el siguiente lema:

Lema 1

Existe una biyección entre escoger 2 de n elementos sin que importe el orden (combinaciones de n en 2) y formar todos los intervalos en un arreglo de n elementos.

Demostración Lema 1

Todo par (x, y) resultante de las combinaciones de n en 2, se puede ordenar de una única forma tal que $x < y$. Entonces podemos ver siempre x e y como el inicio y el final del intervalo, respectivamente. Como esto va a generar todas las parejas posibles a formar, tomando como conjunto las n posiciones del arreglo, se garantiza que no se obtendrá una pareja cuyos elementos no tendrán una posición correspondiente en el arreglo, y que no existirá una posición del arreglo que no esté contenida en al menos un intervalo.

Sea S la suma de los elementos de un intervalo, la solución será la tupla $(count_k, count_{ab})$ que incluye la cantidad de intervalos que cumplen que S es exactamente k y $S \in [a, b]$, respectivamente.

Complejidad Temporal
Sea una instancia del problema, donde se tiene como entrada un arreglo de longitud n, los enteros a, b, k, tales que $a < b$. Se sigue el siguiente algoritmo:

1. Recorrer con i todas las posiciones de arreglo, fijando el inicio de cada intervalo.
2. Para cada posible inicio, se recorre con j todos los posibles finales que puede tener el intervalo, comenzando en $j = i$ para contemplar la posibilidad de que el elemento en esa posición sea un intervalo válido.
3. Comprobar para todo par si la suma S de los elementos del intervalo que ellos definen en el arreglo:
   - $S \in [a, b]$: incrementar en uno $count_{ab}$
   - $S = k$: incrementar en uno $count_k$
4. Devolver $(count_k, count_{ab})$ como respuesta.

La complejidad temporal de este algoritmo es:

$$O(n^2)$$

Solución 2:

Esta solución se compone de resolver por separado los dos subproblemas que tenemos, el de calcular los intervalos cuyos elementos sumen exactamente k (I) y el de calcular los intervalos cuyos elementos sumen un valor $\in [a, b]$ (II).

Lema 2
Sean a, b, c tal que $a < b < c$, S_1 la suma de los elementos del intervalo $[a, b]$, S_2 la suma de los elementos del intervalo $[a, c]$ y S la suma de los elementos del intervalo $[b, c]$, se cumple que $S = S_2 - S_1 + b$.

Para llegar a las soluciones I y II tuvimos que hacer algunas definiciones:

- *sum*: arreglo tal que $sum[i]$ es la suma de los elementos desde el inicio hasta la posición i del arreglo dado.
- *sum_dec*: ordenación decreciente del conjunto de elementos de *sum* (sin repetir los que sean iguales en *sum*).
- *list_ind*: lista de igual tamaño que *sum_dec* que agrupa, crecientemente, en la posición i todos los índices en los que aparece $sum\_dec[i]$ en el arreglo *sum*.

Ambos algoritmos se basan en que, teniendo una instancia de *sum_dec* y de *list_ind* asociadas al problema en cuestión, podemos recorrerlos buscando una resta entre intervalos (aplicando el *Lema 2*) que cumpla con la restricción que se está analizando en cada caso. El orden del arreglo *sum_dec* por el que vamos a ir iterando con dos punteros (i, j que marcarían el intervalo $[a, c]$ y $[a, b]$ del *Lema 2*, respectivamente) nos permite ir acotando la resta de forma que siempre vamos a saber qué puntero incrementar para revisar un nuevo posible intervalo; y *list_ind* nos posibilita saber si es un intervalo válido, es decir, si se cumple la desigualdad de la hipótesis del *Lema 2*.

Subproblema Auxiliar I: Trabajo con los Índices

Sean A, B dos listas de enteros ordenadas decrecientemente; i, j dos enteros tal que $0 \leq i < |A|$, $0 \leq j < |B|$. Este subproblema se encarga de calcular la cantidad de parejas (A[i], B[j]) tal que $A[i] > B[j]$. La solución consistirá entonces en comparar estos valores, pudiendo tomar decisiones en cada caso sin perder soluciones, como se explica a continuación:

- $A[i] > B[j]$: $count += |B| - j$. Podemos apoyarnos en el primer punto del *Lema 3*, tomando a $k = 0$. Luego podremos aumentar en 1 el valor de la j, ya que con el valor con el que se comenzó esta iteración ya se habrán contado todas las soluciones.
- $A[i] < B[j]$: Podemos apoyarnos en el segundo punto del *Lema 3*, tomando a $k = 0$ y llegar a la conclusión de que si A[i] no aportará más parejas de las que pueda haber aportado llegado este punto, luego se puede incrementar su valor en 1 y pasar a la siguiente iteración.

Complejidad Temporal

Ya que al finalizar el algoritmo, a lo sumo se habrán recorrido los dos arreglos que se reciben como entrada, y ninguna posición se analizara mas de una vez (siempre se incrementan uno de los dos punteros que los recorren, y nunca se decrementan), la complejidad temporal del mismo es:

$$O(|A| + |B|)$$

Subproblema I

Sea S una instancia de *sum* asociada a los valores dados como entrada, supongamos que tenemos dos instancias de *sum_dec*: `A` y `B`, y dos instancias de *list_ind*: `A_Ind` y `B_Ind`, todas asociadas a S; esto solo nos facilitará la comprensión de la correctitud del algoritmo, en la implementación de la solución solo se utiliza una para cada caso ya que las operaciones a realizar solo serán consultas. De esta forma veremos a `A` como el arreglo donde se encontrarán los minuendos de las diferencias a analizar para hallar los intervalos, y `A_Ind` los índices asociados; mientras que `B` va a representar nuestros sustraendos y `B_Ind` sus índices correspondientes. En consecuencia, i y j van a iterar por `A` y `B`, `A_Ind` y `B_Ind`, respectivamente.

Lema 3

Dados un entero k y dos posiciones i, j en los arreglos `A`,`B` respectivamente, se pueden contemplar los siguientes casos según el valor de $r = A[i] - B[j]$:

1. Si $r < k$, entonces $\forall t$ tal que $i < t < |A|$ se cumple que $A[t] - B[j] < k$
2. Si $r > k$, entonces $\forall t$ tal que $j < t < |B|$ se cumple que $A[i] - B[t] > k$
3. Si $r = k$, entonces $\forall t$ tal que $j < t < |B|$ se cumple que $A[i] - B[t] \neq k$ y $\forall s$ tal que $i < s < |A|$ se cumple que $A[s] - B[j] \neq k$

```
1    def igual_k(elems,k):
2       elems = [0] + elems
3       A,A_index = init_solution(elems)
4       count_k = 0
5       i = 0
6       j = 0
7       while i < len(A) and j < len(A):
8          r = A[i] - A[j]
9          if r > k:
10            i += 1
11         elif r < k:
12            j +=1
```

```
13         else:
14            count_k += count_interv(A_index[i],A_index[j])
15            i += 1
16            j += 1
17      return count_k
```

Correctitud

Para esta solución el algoritmo sigue los siguientes pasos:

1. Construir `A` y `A_index` como instancias de *sum_dec* y *list_ind* respectivamente, a partir del arreglo `elems`.
2. Realizar un recorrido con las variables i y j por el arreglo `A` y `A_index`, basándose en el *Lema 3* y en el *Subproblema Auxiliar* para contar los intervalos cuyos elementos suman k.
3. Devolver `count_k` como resultado.

Como el arreglo `A` está ordenado decrecientemente podemos tomar una decisión en cada iteración según el resultado r obtenido en la resta de los intervalos, basándonos en lo planteado en el *Lema 3*. Por ello, si se cumple que $r < k$ o $r > k$ podemos afirmar que haciendo $j+ = 1$ o $i+ = 1$, respectivamente, no estaríamos excluyendo soluciones. Cuando $r = k$, tenemos que contar los intervalos que resulten válidos según sus índices y, como tenemos esta información almacenada en las listas `A_index[i]` y `A_index[j]` (ordenadas decrecientemente), podemos apoyarnos en el *Subproblema Auxiliar*.

Se garantiza que dicho subproblema resolverá este caso, ya que no es posible que ambos valores sean iguales, dado que las listas del problema corresponden a los índices asociados a dos sumas acumulativas distintas. (A un mismo índice no van a corresponder dos valores de sumas acumulativas diferentes.)

Para ello hacemos uso del método `count_interv(index1,index2)`; al concluir esto, podemos hacer $i+ = 1$ y $j+ = 1$ sin perder soluciones, nuevamente apoyándonos en el *Lema 3*.

Complejidad Temporal

Analizaremos la complejidad temporal según los pasos del algoritmo:

1. En las líneas `2-6` se inicializan todas las variables, incluidas `A` y `A_index` a través del método `init_solution(elems)` que es $O(n \cdot \log n)$ debido a la ordenación.
2. En las líneas `7-16` se realiza el ciclo explicado anteriormente en la correctitud del algoritmo. Notar que en cada iteración se incrementa(n) i y/o j y en ninguna se decrementa(n), haciendo a lo sumo $2n+2$ iteraciones. El método `count_interv(index1,index2)` es $O(|index1|+|index2|)$, entonces en cada iteración que se haga el llamado `count_interv(A_index[i],A_index[j])` se aumentará en $O(|A\_index[i]| + |A\_index[j]|)$ la cantidad de operaciones, pero al cumplirse que:

$$\sum_{t=0}^{|A\_index|-1} A\_index[t] = n$$

   Se puede afirmar que al final del ciclo todos los llamados al método `count_interv(A_index[i],A_index[j])` agregaron en total un máximo de $O(2n)$ operaciones. Haciendo un costo amortizado de $O(2n)$ operaciones.

La complejidad temporal del algoritmo es

$$O(n \cdot \log n) + O(2n)$$

Que por el principio de la suma es:

$$O(n \cdot \log n)$$

Subproblema Auxiliar II: Trabajo con los Índices

Dados un entero k y un conjunto C de enteros, donde se pueden eliminar e insertar elementos, se desea saber la cantidad de elementos en C menores que k.

Para la solución de este problema, nos basamos en la implementación de un AVL, dado que con dicha estructura podemos realizar las operaciones antes mencionadas. Si insertamos los elementos de C en el mismo (donde cada nodo del árbol tiene el valor del elemento), podemos contar cuántos nodos tienen valor menor que el entero k dado. Para ello proveemos el método $countSmallers(root, k)$.

Correctitud

El método $countSmallers(root, k)$ devuelve un entero res, que corresponde a la cantidad de nodos cuyo valor sea menor que k. En cada nodo del árbol incluimos un campo adicional *desc* que almacena la cantidad de nodos en su descendencia. Para calcular el número de nodos que son menores que el valor dado, simplemente recorremos el árbol, pudiéndonos encontrar con los tres casos siguientes:

1. k es mayor que el valor del nodo actual: entonces nos dirigimos al hijo izquierdo de este.
2. k es menor que el valor del nodo actual: aumentamos res en el número de descendientes del hijo izquierdo del nodo actual, y luego lo incrementamos en 2 (1 por el nodo actual y 1 por el hijo izquierdo). Luego nos movemos al hijo derecho del nodo actual.
3. k es igual al valor del nodo actual: aumentamos res en el número de descendientes del hijo izquierdo del nodo actual, y luego lo incrementamos en 1, para contar el hijo izquierdo.

Complejidad Temporal

Dado que la definición propia del AVL nos permite una búsqueda certera, de modo que solo recorreremos a lo sumo una rama del árbol desde la raíz hasta la hoja más lejana, la complejidad temporal del algoritmo corresponde al proceso de búsqueda en esta estructura:

$$O(logn)$$

Subproblema II

Para resolver (II) hallaremos la cardinalidad del complemento del conjunto solución ya que, tras restársela al total, obtendremos la respuesta deseada, es decir, realmente calcularemos la cantidad de intervalos cuya suma sea menor que a y la cantidad de intervalos cuya suma sea mayor que b. Para esto nos apoyaremos en los métodos `menores_que(elems,a)` y `mayores_que(elems,b)`, que se explican a continuación.

```
1    def menores_que(elems,a):
2       count = 0
3       elems = [0] + elems
4       A,A_Index = init_solution(elems)
5       I = None
6       i = 0
7       j = 0
8       while i < len(A) and j < len(A):
9          r = A[i] - A[j]
10         if r < a:
11            for item in A_Index[j]:
12               I = insert(I, item)
13            j += 1
14         else:
```

```
15            for t in range(0,len(A_Index[i])):
16               count += countSmallers(I,A_Index[i][t])
17            i += 1
18      while i < len(A_Index):
19         for t in range(0,len(A_Index[i])):
20            count += countSmallers(I,A_Index[i][t])
21         i += 1
22      return count
```

Correctitud

El algoritmo sigue los siguientes pasos:

1. Construir `A` y `A_index` como instancias de *sum_dec* y *list_ind* respectivamente, a partir del arreglo `elems`.
2. Realizar un recorrido con las variables i y j por el arreglo `A` y `A_index`, basándose en el *Lema 3* y en el *Subproblema Auxiliar II* para contar los intervalos cuyos elementos sumen menos que a.
3. Devolver `count` como resultado.

Al igual que en algoritmo `igual_k(elems,k)`, recorremos el arreglo `A` en base a lo que plantea el *Lema 3*. Lo que esta vez para contar los intervalos que son solución seguiremos los siguientes criterios en dependencia de `r = A[i] - A[j]`:

1. Si $r < a$: entonces por el *Lema 3* toda posición t tal que $i < t < |A|$ va a cumplir que $A[t] - A[j] < a$. Por tanto, es necesario contar todos los intervalos que se puedan formar con los índices de $A\_index[j]$ y $A\_index[t]$, para resolver esto, insertamos en un AVL I los índices que están en $A\_index[j]$.
2. Si $r \geq a$: entonces por el *Lema 3* toda posición t tal que $j < t < |A|$ va a cumplir que $A[i] - A[t] \geq a$. Por tanto, ninguno de los intervalos que se puedan formar con los índices de $A\_index[i]$ y $A\_index[t]$ debe ser contado. Llegado a este punto, se calcula la cantidad de intervalos que se pueden formar con los índices de $A\_index[i]$ y los acumulados en I en iteraciones anteriores, haciendo uso del método `countSmallers` planteado en el *Subproblema Auxiliar II*.
3. Una vez finalizado el ciclo anterior, se calcula la cantidad de intervalos que se pueden formar con los índices que quedaron en I y las posiciones no alcanzadas por i, también con el método `countSmallers`.

Complejidad Temporal

Analizaremos la complejidad temporal según los pasos del algoritmo:

- En las líneas `2-7` se inicializan todas las variables, incluidas `A` y `A_index` a través del método `init_solution(elems)` que es $O(n \cdot \log n)$ debido a la ordenación.
- En las líneas `8-17` se realiza el ciclo explicado anteriormente en la correctitud del algoritmo. Igual que en el método `igual_k(elems,k)`, en cada iteración se incrementa i o j y en ninguna se decrementa, haciendo a lo sumo $2n + 2$ iteraciones.
- El método `countSmallers` es $O(\log n)$ y a lo sumo se ejecuta n veces en todo el algoritmo. Por tanto, agrega una complejidad temporal $O(n \cdot \log n)$

Entonces la complejidad temporal del algoritmo, con un costo amortizado:

$$O(n \cdot \log n) + O(n) + O(n \cdot \log n)$$

Que por el principio de la suma es:

$$O(n \cdot \log n)$$

```
1    def mayores_que(elems,b):
2       count = 0
3       elems = [0] + elems
4       A,A_Index = init_solution(elems)
5       I = None
6       for i in range(0,len(elems)):
7          I = insert(I,i)
8       i = 0
9       j = 0
10      while i < len(A) and j < len(A):
11         r = A[i] - A[j]
12         if r > b:
13            for t in range(0,len(A_Index[i])):
14               count += countSmallers(I,A_Index[i][t])
15            i += 1
16         else:
17            for t in range(0,len(A_Index[j])):
18               I = deleteNode(I,A_Index[j][t])
19            j += 1
20      while i < len(A_Index):
21         for t in range(0,len(A_Index[i])):
22            count += countSmallers(I,A_Index[i][t])
23         i += 1
24      return count
```

Correctitud

En cada iteración en el AVL I vamos a almacenar todos los índices que pudieran definir un intervalo con los de $A\_Index[i]$. Inicialmente insertamos en I todos los índices del arreglo A, ya que todos son posibles candidatos en un primer momento.

Dado que el algoritmo sigue los mismos pasos que el anterior, la correctitud es igual, exceptuando el punto 2 donde ahora el objetivo es contar los intervalos tal que la suma de sus elementos sea mayor que b. Para esto contemplaremos los siguientes casos respecto a `r = A[i]-A[j]`:

1. Si $r \leq b$: entonces por el *Lema 3* $\forall t$ tal que $i < t < |A|$ se cumple que $A[t] - A[j] \leq b$, por tanto los índices presentes en $A\_index[j]$ no van a formar intervalos que deban ser contados, teniendo así, que eliminarlos de I. La siguiente combinación a analizar sería $A[i]$,$A[j+1]$.

2. Si $r > b$: entonces por el *Lema 3* $\forall t$ tal que $j < t < |A|$ se cumple que $A[i] - A[t] > b$, por tanto por cada índice en $A\_Index[i]$ deberán ser contados los que sean menor que él en I, a través del método `countSmallers` planteado en el *Subproblema Auxiliar II*. En este caso ya fueron contados todos los posibles intervalos para $A[i]$, por lo que la próxima combinación a analizar sería $A[i+1]$,$A[j]$.

Complejidad Temporal

Analizaremos la complejidad temporal según los pasos del algoritmo:

- En las líneas `2-7` se inicializan todas las variables, incluidas `A` y `A_index` a través del método `init_solution(elems)` que es $O(n \cdot \log n)$ debido a la ordenación.

- En las líneas `8-19` se realiza el ciclo explicado anteriormente en la correctitud del algoritmo. Igual que en el método `menores_que(elems,a)`, en cada iteración se incrementa i o j y en ninguna se decrementa, haciendo a lo sumo $2n + 2$ iteraciones.
- El método `countSmallers` es $O(\log n)$ y a lo sumo se ejecuta n veces en todo el algoritmo, por lo que agrega una complejidad temporal $O(n \cdot \log n)$

Entonces la complejidad temporal del algoritmo, con un costo amortizado:

$$O(n \cdot \log n) + O(n) + O(n \cdot \log n)$$

Que por el principio de la suma es:

$$O(n \cdot \log n)$$

Conclusión subproblema II

```
1    def menor_mayor_que(elems,a,b):
2       count_a = menores_que(elems,a)
3       count_b = mayores_que(elems,b)
4       if len(elems) == 1:
5          return max(0,1 - count_a - count_b)
6       return combinaciones(len(elems),2) - count_a - count_b
```

La correctitud del presente subproblema está determinada por la de los dos métodos que la conforman, vistos anteriormente. Siendo la complejidad temporal del subproblema **II**:

$$O(n \cdot \log n) + O(n \cdot \log n)$$

Que por el principio de la suma es

$$O(n \cdot \log n)$$

Conclusión Solución 2

```
1    def solution_2(elems,k,a,b):
2       return igual_k(elems,k),menor_mayor_que(elems,a,b)
```

Dado que el algoritmo para resolver el problema inicial es la composición de los subproblemas **I** y **II**, su correctitud se apoya en la de estos. Asimismo, la complejidad temporal resulta de aplicar el principio de la suma a la de cada uno de dichos subproblemas.

$$O(n \cdot \log n) + O(n \cdot \log n) = O(n \cdot \log n)$$