Tarea 1 - Programación y algoritmos 2

Erick Salvador Alvarez Valencia

CIMAT A.C.,
erick.alvarez@cimat.mx

Resumen En la presente tarea se analizará la solución implementada al problema: **MKTHNUM - K-th Number** de SPOJ, así como la complejidad de la misma.

1. Descripción del problema

En este problema se da un conjunto de n números enteros en un arreglo y un número de m de consultas, en cada consulta se dan tres valores **i, j, k** y consiste en: Dado el intervalo $[i, j]$, ¿cuál es el k-ésimo número más pequeño?.

Las restricciones del problema son:

1. Tiempo límite: 0.115s - 0.667s
2. Límite de memoria: 1536MB
3. $1 \leq n \leq 100000$
4. $1 \leq m \leq 5000$
5. $|a_i| \leq 10^9$ (a_i es el i-ésimo elemento del arreglo)
6. $1 \leq i \leq j \leq n, 1 \leq k \leq j - i + 1$

2. Solución implementada

Para este problema se usó el *merge sort tree* el cual es un árbol de segmentos que en cada nodo contiene un vector de enteros, dicho vector, es la mezcla ordenada de los vectores que contienen sus dos hijos, y para los nodos raíces estos solamente tienen un elemento. Este árbol es fabricado a partir de un arreglo inicial, y además este usa la función *merge* para calcular el valor de sus nados, la misma que se utiliza en el ordenamiento por mezcla. El algoritmo de construcción del árbol es el siguiente:

Algorithm 1 Construcción del árbol.

```
1: procedure BuildTree(Tree, Arr, node, lo, hi)
2:     if lo == hi then
3:         Tree[node].push(Arr[lo])
4:     else
5:         mid ← (lo + hi) / 2
6:         lc ← 2 * node
7:         rc ← 2 * node + 1
8:         BuildTree(Tree, Arr, lc, lo, mid)
9:         BuildTree(Tree, Arr, rc, mid + 1, hi)
10:        merge(Tree, node)
```

La construcción se hace de forma recursiva primero calculando el valor de cada hijo del nodo actual y usando el algoritmo de mezcla se calcula el valor del nodo actual usando el de sus hijos. Para la implementación se usó un arreglo de vectores.
Este árbol en general sirve para responder a las preguntas de la forma: Dado un intervalo $[i, j]$ y un número k, ¿cuál es el número de elementos menores/mayores que k en el mismo? Y para ello se usa el siguiente algoritmo:

Algorithm 2 Consulta del árbol.

```
1: procedure Query(Tree, Arr, node, lo, hi, i, j, k)
2:     if i > hi or j < lo then
3:         Return 0
4:     else if i ≤ lo and j ≥ hi then
5:         Return bs(node, k)
6:     else
7:         mid ← (lo + hi) / 2
8:         lc ← 2 * node
9:         rc ← 2 * node + 1
10:        q1 ← Query(Tree, Arr, lc, lo, mid, i, j, k)
11:        q2 ← Query(Tree, Arr, rc, mid + 1, hi, i, j, k)
12:        Return q1 + q2
```

La consulta al árbol se hace como se vió en clase con un árbol de segmentos, pero en este caso, cuando el intervalo que pertenece al nodo actual está contenido en el intervalo $[i, j]$ lo que se hace es correr una búsqueda binaria sobre el intervalo asociado al nodo para poder conocer cuántos números mayores/menores que k existen. La forma de la búsqueda binaria puede ser tipo *lower bound* o *upper bound* dependiendo a lo que se busque. En caso de estar parcialmente contenidos en el intervalo, se hacen las consultas recursivamente y se suman las respuestas, esto se puede hacer por la transitividad del conjunto, es decir, k tiene el mismo número de elementos menores/mayores independientemente a si estos

se encuentran en diferentes subconjuntos.
Para poder responder a la pregunta que plantea el problema tenemos que buscar el mínimo elemento e tal que el número de elementos menores o iguales a el, sea k, y para ello podemos usar otra búsqueda binaria sobre el conjunto ordenado de datos. Además de ello se usa la técnica de comprensión de datos en la que se hace un mapeo de un subconjunto de números naturales hacia otro conjunto de naturales pero sin huecos (todos los números son continuos). Esta técnica se usa para que la complejidad de la búsqueda binaria no sea tan grande.
Para poder aplicar esta técnica se requiere que el arreglo de datos se encuentre ordenado, pero de ser así ya no podríamos responder a las queries de manera correcta, por lo que se usa un arreglo auxiliar el cual si podemos ordenar, y lo que se hace es por cada elemento del arreglo sin ordenar, buscamos la posición del mismo en la versión ordenada y guardamos dicha información en otro arreglo (puede ser el mismo), de esta manera si queremos hacer el mapeo inverso simplemente consultamos al arreglo ordenado usando la posición del elemento de interés que fue guardada en el otro arreglo.
A continuación se muestra el algoritmo para hacer la compresión de caminos:

Algorithm 3 Compresión de caminos.

```
1: procedure DataCompression(Arr, Comp, N)
2:     SortAsc(Arr)
3:     for i : 1 → N do
4:         linf ← 0
5:         lsup ← n
6:         while linf ≤ lsup do
7:             mid ← (linf + lsup) / 2
8:             if Arr[mid] == Comp[i] then
9:                 Comp[i] = mid
10:                Break
```

Podemos ver en el algoritmo anterior que por cada elemento del arreglo $Comp$, buscamos la posición en la versión ordenada del mismo (Arr) y la guardamos en el mismo arreglo $Comp$, al finalizar el algoritmo podemos tener una versión comprimida de los datos de entrada así como un mapeo a los mismos. Con esto anterior se hace la búsqueda binaria que consultará varias veces al árbol para buscar el número que se mencionó anteriormente. La complejidad de este algoritmo es $O(mlog^3(n))$ donde m es el número de queries y n el tamaño del conjunto de datos. Esto anterior porque al hacer una consulta al segment tree este hace un recorrido de a lo más $log(n)$ nodos y por cada nodo se hace una búsqueda binaria para saber cuántos elementos son menores que k, esto genera una complejidad de $O(log^2(n))$ por consulta al árbol, y como en cada querie se hace una búsqueda binaria la cual hace a lo más $log(n)$ consultas al árbol, se tiene que la complejidad final por querie es de $O(log^3(n))$.

3. Resultados

La implementación del algoritmo se envió al sistema SPOJ para su evaluación y obtuvo el resultado de Aceptado con un tiempo de: 1.07 s

(a) Figura 1. Resultado del envío.