GUÍA DE APRENDIZAJE PARA CONCURSANTES ICPC Y IOI: ACTUALIZACIONES EN RANGO (LAZY PROPAGATION)

1. Introducción

Cuando se abordaron el trabajo con árbol de rango (*Range Tree*) siempre discutieron con la operación de consultas de modificación que solo afectaron a un solo elemento en el arreglo. Pero que hacer cuando la operación de consulta de modificación ya no afecta solamente a un elemento sino a varios elementos del arreglo definidos por un rango $[l \dots r]$.

Una primera idea trivial sería llamar a la función de actualización del árbol de rango por cada posición dentro del rango $[l \dots r]$ pero esto sería demasiado costoso en tiempo. Sin embargo, el árbol de rango permite aplicar consultas de modificación a un rango completo de elementos contiguos y realizar la consulta al mismo tiempo $O(\log n)$ dicha modificación se conoce como propagación peresoza (*lazy propagation*).

2. Conocimientos previos

2.1. Árbol de rango (*Range Tree*)

Es una estructura de datos basado en un árbol binario que permite responder consultas de rango sobre un marreglo de manera efectiva, sin dejar de ser lo suficientemente flexible como para permitir la modificación del arreglo. Se identifican tres operaciones:

- **Construcción:** Permite la construcción de forma recursiva del árbol.
- **Actualización:** Permite la actualización de un elemento dentro del arreglo dada la posición de este y el nuevo valor.
- **Consulta:** Permite responder una determinada pregunta para los valores comprendidos dentro de un rango de posiciones consecutivas del arreglo.

3. Desarrollo

Lo que vamos abordar no es una estructura de datos nuevas ni un algoritmo nuevo sino una modificación a una estructura ya existente a la cual le vamos a realizar algunas modificaciones es por eso que vamos a ver dicha modificación en algunos de los usos tipicos de la estructura que sirven de base para otros casos.

3.1. Suma en rangos

Comenzamos considerando problemas de la forma más simple: la consulta de modificación debe agregar un número x a todos los números del rango $a[l \dots r]$. La segunda consulta, que se supone que debemos responder, preguntaba simplemente por el valor de $a[i]$.

Para que la consulta de suma sea eficiente, almacenamos en cada vértice del árbol de rangos cuántos debemos sumar a todos los números en el rango correspondiente. Por ejemplo, si la consulta agrega 3 a todo el arreglo $a[0 \dots n-1]$ viene, luego colocamos el número 3 en la raíz del árbol. En general, tenemos que colocar este número en varios rangos, que forman una partición

del rango de consulta. Por lo tanto, no tenemos que cambiar todo $O(n)$ valores, pero sólo a lo mucho $O(\log n)$.

Si ahora surge una consulta que solicita el valor actual de una posición del arreglo en particular, es suficiente con bajar el árbol y sumar todos los valores encontrados en el camino.

3.2. Asignación en rangos

Supongamos ahora que la consulta de modificación pide asignar cada elemento de un determinado rango $a[l \dots r]$ a algún valor p. Como segunda consulta, consideraremos nuevamente leer el valor de la matriz $a[i]$.

Para realizar esta consulta de modificación en un rango completo, debe almacenar en cada vértice del árbol de rangos si el rango correspondiente está cubierto por completo con el mismo valor o no. Esto nos permite hacer una actualización perezosa: en lugar de cambiar todos los rangos en el árbol que cubre el rango de consulta, solo cambiamos algunos y dejamos otros sin cambios. Un vértice marcado significará que cada elemento del rango correspondiente está asignado a ese valor, y en realidad también el subárbol completo solo debe contener este valor. En cierto sentido, somos perezosos y demoramos en escribir el nuevo valor en todos esos vértices. Podemos hacer esta tediosa tarea más tarde, si es necesario.

Entonces, después de que se ejecuta la consulta de modificación, algunas partes del árbol se vuelven irrelevantes; algunas modificaciones permanecen sin cumplir en él.

Por ejemplo, si una consulta de modificación asigne un número a toda el arreglo $a[0 \dots n-1]$ se ejecuta, en el árbol de rangos solo se realiza un único cambio: el número se coloca en la raíz del árbol y este vértice se marca. Los rangos restantes permanecen sin cambios, aunque de hecho el número debe colocarse en todo el árbol.

Supongamos ahora que la segunda consulta de modificación dice que la primera mitad del arreglo $a[0 \dots n/2]$ debe ser asignado con algún otro número. Para procesar esta consulta debemos asignar a cada elemento del hijo izquierdo completo del vértice raíz con ese número. Pero antes de hacer esto, primero debemos clasificar el vértice de la raíz. La sutileza aquí es que la mitad derecha del arreglo aún debe asignarse al valor de la primera consulta, y en este momento no hay información almacenada para la mitad derecha.

La forma de resolver esto es *empujar* la información de la raíz a sus hijos, es decir, si a la raíz del árbol se le asignó algún número, entonces asignamos los vértices secundarios izquierdo y derecho con este número y eliminamos la marca de la raíz. Después de eso, podemos asignar al hijo izquierdo el nuevo valor, sin perder la información necesaria.

Resumiendo, obtenemos: para cualquier consulta (una consulta de modificación o lectura) durante el descenso a lo largo del árbol, siempre debemos empujar la información del vértice actual a sus dos hijos. Esto lo podemos entender de tal forma, que cuando descendemos del árbol aplicamos modificaciones retrasadas, pero exactamente las necesarias (para no degradar la complejidad de $O(\log n)$.

Para la implementación necesitamos hacer una función push, que recibirá el vértice actual, y

enviará la información de su vértice a sus dos hijos. Llamaremos a esta función al comienzo de las funciones de consulta (pero no la llamaremos desde las hojas, porque no hay necesidad de extraer más información de ellas).

3.3. Agregando rangos, consultando el máximo

Ahora la consulta de modificación es agregar un número a todos los elementos en un rango, y la consulta de lectura es encontrar el máximo en un rango.

Así que para cada vértice del árbol del rango tenemos que almacenar el máximo del subrango correspondiente. La parte interesante es cómo volver a calcular estos valores durante una solicitud de modificación.

Para ello mantenemos almacenado un valor adicional para cada vértice. En este valor almacenamos los sumandos que no hemos propagado a los vértices secundarios. Antes de atravesar a un vértice hijo, llamamos push y propagar el valor a ambos hijos. Tenemos que hacer esto tanto en la función actualizar y la función consulta.

4. Implementación

Vamos a utilizar algunas implementaciones de soluciones para comentar las adecuaciones.

4.1. C++

```
#include <iostream>
#include <algorithm>
#include <math.h>
#define MID (left+right)/2
#define MOD  1000000007
#define MAX 1000010
#define MAXTREE (MAX << 2)

using namespace std;

struct Node{
   int flowers;
   bool lazy; /* va indicar si determinado nodo en
         el arbol tiene o no propagacion el tipo de dato
         puede variar segun la necesidad o el enfoque
         que se de la propagacion*/
};

Node tree[MAXTREE];

void buildRangeTree(int treeIndex, int left, int right){

  tree[treeIndex].lazy=false; /*inicialmente cada
```

```
                                nodo no tiene propagacion*/
  tree[treeIndex].flowers=0;

  if (left == right) {
     tree[treeIndex].flowers=0;return;
  }

  buildRangeTree(2*treeIndex,left,MID);
  buildRangeTree(2*treeIndex+1,MID+1,right);
}

/*Aqui cambia la actualizacion antes era una posicion
ahora es un rango (i,j)*/
void updateLazyRangeTree(int treeIndex, int left, int right, int i, int j){
   if (left > right || left > j || right < i) return;

   /*Si el rango que representa ese nodo o vertice
   en el arbol esta contenido en el rango de la actualizacion(i,j)
   detengo la actualizacion y lo marco como que existe una propagacion
   pendiente
   */
   if (i <= left && right <= j){
     tree[treeIndex].lazy=true; tree[treeIndex].flowers++; return;
   }

   /*El rango de actualizacion(i,j) es un subrango del rango que
   representa el nodo o vertice del arbol donde estoy parado que
   tiene propagaciones pendientes por tanto propago primero
   dichas actualizaciones pendientes y luego sigo el proceso de
   actualizacion en rango. Una vez propagado la actualizaciones
   pendientes ya el nodo o vertice del arbol deja de tener propagaciones
   pendientes. Funcion push*/
   if(tree[treeIndex].lazy==true){
      tree[2*treeIndex].lazy=tree[treeIndex].lazy;
      tree[2*treeIndex+1].lazy=tree[treeIndex].lazy;
      tree[2*treeIndex].flowers+=tree[treeIndex].flowers;
      tree[2*treeIndex+1].flowers+=tree[treeIndex].flowers;
      tree[treeIndex].lazy = false;
      tree[treeIndex].flowers = 0;
   }

   updateLazyRangeTree(2 * treeIndex , left, MID, i, j);
   updateLazyRangeTree(2 * treeIndex + 1, MID + 1, right, i, j);
}

int queryLazyRangeTree(int treeIndex, int left, int right, int pos){

   if(left==right){
      int answer = tree[treeIndex].flowers; tree[treeIndex].flowers=0;
      return answer;
```

```
    }

    /*El rango de consulta(left,rigth) es un subrango del rango que
    representa el nodo o vertice del arbol donde estoy parado que
    tiene propagaciones pendientes por tanto propago primero
    dichas actualizaciones pendientes y luego sigo el proceso de
    consulta en rango. Una vez propagado la actualizaciones
    pendientes ya el nodo o vertice del arbol deja de tener propagaciones
    pendientes. Funcion push*/
    if (tree[treeIndex].lazy == true){
        tree[2*treeIndex].lazy=tree[treeIndex].lazy;
        tree[2*treeIndex+1].lazy=tree[treeIndex].lazy;
        tree[2*treeIndex].flowers+=tree[treeIndex].flowers;
        tree[2*treeIndex+1].flowers+=tree[treeIndex].flowers;
        tree[treeIndex].lazy = false;
        tree[treeIndex].flowers = 0;
    }
    if (pos > MID)
        return queryLazyRangeTree(2*treeIndex+1,MID+1,right,pos);
    if (pos <= MID)
        return queryLazyRangeTree(2*treeIndex,left,MID,pos);
}
```

4.2. Java

```
private class RangeTreeLP {
    private int[] tree;
    private boolean[] lz;

    public RangeTreeLP(int n) {
        tree = new int[n << 2]; lz = new boolean[n << 2];
    }

    public void buildRangeTree(int treeIndex, int left, int right) {
        /*
        * inicialmente cada nodo no tiene propagacion
        */
        lz[treeIndex] = false; tree[treeIndex] = 0;

        if (left == right) { tree[treeIndex] = 0; return; }

        int MID = (left + right) / 2;
        buildRangeTree(2 * treeIndex, left, MID);
        buildRangeTree(2 * treeIndex + 1, MID + 1, right);
    }

    /*
    * Aqui cambia la actualizacion antes era una posicion
```

```
    * ahora es un rango (i,j)
    */
    public void updateLazyRangeTree(int treeIndex,int left,int right,int i,int
        j){
       if (left > right || left > j || right < i) return;

       /*
       * Si el rango que representa ese nodo o vertice en el arbol esta
       * contenido en el rango de la actualizacion(i,j) detengo la
       * actualizacion y lo marco como que existe una propagacion pendiente
       */
       if (i <= left && right <= j) {
          lz[treeIndex] = true; tree[treeIndex]++; return;
       }

       /*
       * El rango de actualizacion(i,j) es un subrango del rango que
       * representa el nodo o vertice del arbol donde estoy parado que
       * tiene propagaciones pendientes por tanto propago primero dichas
       * actualizaciones pendientes y luego sigo el proceso de
       * actualizacion en rango. Una vez propagado la actualizaciones
       * pendientes ya el nodo o vertice del arbol deja de tener propagaciones
       * pendientes. Funcion push
       */
       if (lz[treeIndex] == true) {
          lz[2 * treeIndex] = lz[treeIndex];
          lz[2 * treeIndex + 1] = lz[treeIndex];
          tree[2 * treeIndex] += tree[treeIndex];
          tree[2 * treeIndex + 1] += tree[treeIndex];
          lz[treeIndex] = false; tree[treeIndex] = 0;
       }
       int MID = (left + right) / 2;

       updateLazyRangeTree(2 * treeIndex, left, MID, i, j);
       updateLazyRangeTree(2 * treeIndex + 1, MID + 1, right, i, j);
    }

    public int queryLazyRangeTree(int treeIndex,int left,int right,int pos){
       if (left == right){
          int answer = tree[treeIndex];
          tree[treeIndex] = 0; return answer;
       }

       /*
       * El rango de consulta(left,rigth) es un subrango del rango
       * que representa el nodo o vertice del arbol donde estoy
       * parado que tiene propagaciones pendientes por tanto propago
       * primero dichas actualizaciones pendientes y luego sigo el
       * proceso de consulta en rango. Una vez propagado la
       * actualizaciones pendientes ya el nodo o vertice del arbol
```

```
        * deja de tener propagaciones pendientes. Funcion push
        */
        if(lz[treeIndex] == true){
            lz[2 * treeIndex] = lz[treeIndex];
            lz[2 * treeIndex + 1] = lz[treeIndex];
            tree[2 * treeIndex] += tree[treeIndex];
            tree[2 * treeIndex + 1] += tree[treeIndex];
            lz[treeIndex] = false;
            tree[treeIndex] = 0;
        }

        int MID = (left + right) / 2;
        if (pos > MID)
            return queryLazyRangeTree(2 * treeIndex + 1, MID + 1, right, pos);
        if (pos <= MID)
            return queryLazyRangeTree(2 * treeIndex, left, MID, pos);
    }
}
```

5. Complejidad

A pesar de las modificaciones que se realizan al árbol de rango cuando se le incorpora esta idea su complejidad temporal no sufre ninguna modificación. Cada uno de sus operaciones siguen manteniendo la misma complejidad

6. Aplicaciones

La incorporación de esta idea a la estructura de datos árbol de rango amplia el campo de uso de la estructura de datos en sí. Dicha idea puede ser aplicada a otras estructuras en forma de árbol.

7. Ejercicios propuestos

A continuacion una lista de ejercicios que se resuelven aplicando esta técnica.

- DMOJ - Multiplicación en Rango
- DMOJ - Flores
- DMOJ - Dibujos para colorear