GUÍA DE APRENDIZAJE PARA CONCURSANTES ICPC Y IOI: ÁRBOL DE RANGO (RANGE TREE)

1. Introducción

En múltiples problemas nos podemos encontrar en la que tenemos una colección de datos cuya cantidad ronda los 10^5 elementos. Sobre dicha colección sea realiza dos tipos de operaciónes la cuales se pueden clasificar como:

- **Actualización:** Consiste dada una posición dentro de la colección actualizar el valor de esa posición por un nuevo valor. (Tipo I)
- **Consulta:** Consiste en dado un rango de posiciones dentro de la colección ver cuales de los elementos de la colección ubicados en posiciones dentro del rango definido cumple con una determinada cualidad. (Tipo II)

La cantidad de operaciones que se pueden realizar sobre la colección ronda la cantidad de 10^5. Una idea trivial para solucionar este tipo de problema es resolverlo aplicando un poco de fuerza bruta pero esto en el peor de los casos pudiera arrojar una complejidad de O($N * M$) siendo N la candidad de elementos de la colección y M la cantidad de operaciones a realizar. En caso de las complejidades de las operaciones es facil ver como las operaciones de Tipo I su complejidad es de O(1) mientras las del tipo II en el peor caso pudiera ser de O(N) (se tenga que analizar todo el rango de la colección). Por tanto asumiendo un caso que todas las operaciones a realizar sean de tipo de II nuestro algoritmo tendría una complejidad de O($N * M$) como ya se menciono anteriormente lo que sustituyendo los valores podriamos tener un algoritmo de hasta 10^{10} operaciones.

En la guía de hoy veremos como con el uso de una estructura de datos y sacrificando tiempo en la operación de tipo I y memoria se logra reducir la complejidad para solucionar este problema con una complejidad no mayor que O($M \log N$).

2. Conocimientos previos

2.1. Arbol

Los árboles son contenedores que permiten organizar un conjunto de objetos en forma jerárquica. Ejemplos típicos son los diagramas de organización de las empresas o instituciones y la estructura de un sistema de archivos en una computadora. Los árboles sirven para representar fórmulas, la descomposición de grandes sistemas en sistemas más pequeños en forma recursiva y aparecen en forma sistemática en muchísimas aplicaciones de la computación científica. Una de las propiedades más llamativas de los árboles es la capacidad de acceder a muchísimos objetos desde un punto de partida o raíz en unos pocos pasos.

2.2. Arbol Binario

Un árbol binario es una estructura de datos en la cual cada nodo puede tener un hijo izquierdo y un hijo derecho. No pueden tener más de dos hijos (de ahí el nombre "binario"). Si algún hijo tiene como referencia a null, es decir que no almacena ningún dato, entonces este es llamado un nodo externo. En el caso contrario el hijo es llamado un nodo interno. Usos comunes de los árboles binarios son los árboles binarios de búsqueda, los montículos binarios y Codificación de Huffman.

En teoría de grafos, se usa la siguiente definición: «Un árbol binario es un grafo conexo, acíclico y no dirigido tal que el grado de cada vértice no es mayor a 3». De esta forma solo existe un camino entre un par de nodos.

Un árbol binario con enraizado es como un grafo que tiene uno de sus vértices, llamado raíz, de grado no mayor a 2. Con la raíz escogida, cada vértice tendrá un único padre, y nunca más de dos hijos. Si rehusamos el requerimiento de la conectividad, permitiendo múltiples componentes conectados en el grafo, llamaremos a esta última estructura un bosque.

3. Desarrollo

Para empezar fácil, consideramos la forma más simple de un árbol de rangos. Queremos responder consultas de suma de manera eficiente. La definición formal de nuestra tarea es: Tenemos una matriz $a[0 \dots n-1]$, y el árbol de segmentos debe poder encontrar la suma de elementos entre los índices l y r (es decir, calcular la suma $\sum_{i=l}^{r} a[i]$), y también manejar valores cambiantes de los elementos en la matriz (es decir, realizar asignaciones de la forma $a[i] = x$). El árbol de segmentos debería poder procesar ambas consultas en $O(\log n)$ tiempo.

Un árbol de rangos es una estructura de datos que permite responder consultas de rango sobre una matriz de manera efectiva, sin dejar de ser lo suficientemente flexible como para permitir la modificación de la matriz. Esto incluye encontrar la suma de los elementos consecutivos de la matriz $a[l \dots r]$, o encontrar el elemento mínimo en tal rango en el tiempo O($\log N$). Entre las respuestas a dichas consultas, el Árbol de segmentos permite modificar la matriz reemplazando un elemento, o incluso cambiar los elementos de un subsegmento completo (por ejemplo, asignando todos los elementos $a[l \dots r]$ a cualquier valor, o agregando un valor a todos los elementos en el subsegmento).

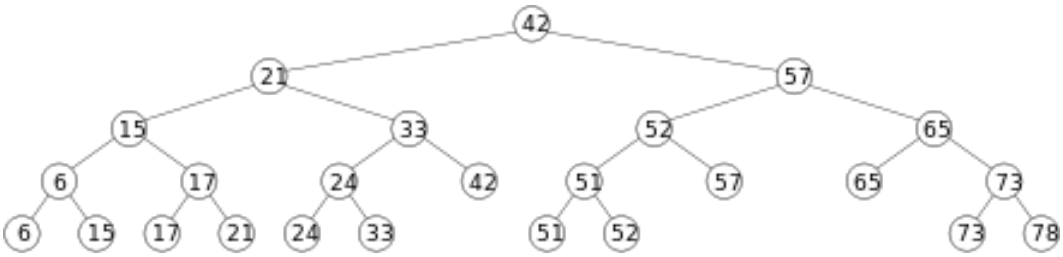

Figura 1: Representación de un árbol de rango

3.1. Operaciones

Construción Un árbol de rangos se puede construir eficientemente de la siguiente manera: Comenzamos en el nivel inferior, los vértices de las hojas. Un vértice es un vértice de hoja, si su segmento correspondiente cubre un solo valor. Por lo tanto, simplemente podemos copiar los valores de los elementos $a[i]$. Sobre la base de estos valores, podemos calcular las sumas del nivel anterior. Y en base a ellos, podemos calcular las sumas de los anteriores, y repetir el procedimiento

hasta llegar al vértice de la raíz. Es conveniente describir esta operación recursivamente: comenzamos la construcción en el vértice raíz y el procedimiento de construcción, si se invoca en un vértice que no es hoja, primero construye recursivamente los dos vértices secundarios y luego suma las sumas calculadas de estos elementos secundarios. Si se llama en un vértice de hoja, simplemente usa el valor de la matriz.

La complejidad temporal de la construcción es $O(n)$.

Actualización Ahora queremos modificar un elemento específico en la matriz, digamos que queremos hacer la asignación $a[i] = x$. Y tenemos que reconstruir el árbol de segmentos, de modo que corresponda a la nueva matriz modificada.

Esta consulta es más fácil que la consulta de suma. Cada nivel de un árbol de segmentos forma una partición de la matriz. Por lo tanto, un elemento $a[i]$ solo contribuye a un segmento de cada nivel. Por lo tanto, solo se deben actualizar los vértices $O(\log n)$.

Es fácil ver que la solicitud de actualización se puede implementar mediante una función recursiva. La función pasa el vértice del árbol actual, y recursivamente se llama a sí misma con uno de los dos vértices secundarios (el que contiene $a[i]$ en su segmento), y luego vuelve a calcular su valor de suma, similar a como se hace en el método de construcción (es decir, como la suma de sus dos hijos).

Nuevamente aquí hay una visualización que usa la misma matriz. Aquí realizamos la actualización $a[2] = 3$. Los vértices verdes son los vértices que visitamos y actualizamos.

Intervalo de consultas Por ahora vamos a responder consultas. Como entrada recibimos dos enteros l y r, y tenemos que calcular la respuesta del rango $a[l \dots r]$ en $O(\log n)$ tiempo.

Para hacer esto, recorreremos el árbol de rangos y usaremos las sumas precalculadas de los segmentos. Supongamos que actualmente estamos en el vértice que cubre el segmento $a[tl \dots tr]$. Hay tres casos posibles.

El caso más sencillo es cuando el rango $a[l \dots r]$ es igual al rango correspondiente del vértice actual (es decir, $a[l \dots r] = a[tl \dots tr]$), entonces hemos terminado y podemos devolver la suma precalculada que está almacenada en el vértice.

Alternativamente, el rango de la consulta puede caer completamente en el dominio del niño izquierdo o derecho. Recuerda que el hijo izquierdo cubre el rango $a[tl \dots tm]$ y el vértice derecho cubre el rango $a[tm+1 \dots tr]$ con $tm = (tl+tr)/2$. En este caso, simplemente podemos ir al vértice secundario, cuyo segmento correspondiente cubre el rango de consulta, y ejecutar el algoritmo descrito aquí con ese vértice.

Y luego está el último caso, el rango de consulta se cruza con ambos elementos secundarios. En este caso no tenemos otra opción que hacer dos llamadas recursivas, una para cada hijo. Primero vamos al hijo de la izquierda, calculamos una respuesta parcial para este vértice, luego vamos al hijo de la derecha, calculamos la respuesta parcial usando ese vértice, y luego combine las respuestas agregándolas. En otras palabras, dado que el hijo de la izquierda representa el segmento

$a[tl \dots tm]$ y el hijo de la derecha el segmento $a[tm+1 \dots tr]$, calculamos la consulta de suma $a[l \dots tm]$ usando el hijo izquierdo, y la consulta de suma $a[tm+1 \dots r]$ usando el hijo derecho.

Entonces, procesar una consulta es una función que recursivamente se llama a sí misma una vez con el elemento secundario izquierdo o derecho (sin cambiar los límites de la consulta), o dos veces, una para el elemento secundario izquierdo y otra para el elemento secundario derecho (dividiendo la consulta en dos subconsultas). Y la recursión termina, siempre que los límites del rango de consulta actual coincidan con los límites del rango del vértice actual. En ese caso, la respuesta será el valor precalculado de la suma de este rango, que se almacena en el árbol.

En otras palabras, el cálculo de la consulta es un recorrido del árbol, que se extiende a través de todas las ramas necesarias del árbol y utiliza los valores de suma precalculados de los rangos del árbol.

Obviamente, comenzaremos el recorrido desde el vértice raíz del árbol de rangos.

4. Implementación

4.1. C++

```
/*Range Tree implementado para hallar el minimo en el rango*/
#include <stdio.h>
#include <iostream>
#include <algorithm>
#define MID (left+right)/2
#define MAX_N 1000001
#define MAX_TREE (MAX_N << 2)
#define INF 987654321
using namespace std;
typedef long long lld;

int n;
int niz[MAX_N];
int RT[MAX_TREE];

void InitTree(int idx, int left, int right){
   if(left == right){
     RT[idx] = niz[left];return;
   }
   InitTree(2*idx, left, MID);
   InitTree(2*idx+1, MID+1, right);
   RT[idx] = min(RT[2*idx], RT[2*idx+1]);
}

void Update(int idx, int x, int val, int left, int right){
   if(left == right){
     RT[idx] = val;return;
   }
```

```
   if (x <= MID) Update(2*idx, x, val, left, MID);
   else Update(2*idx+1, x, val, MID+1, right);
   RT[idx] = min(RT[2*idx], RT[2*idx+1]);
}

int Query(int idx, int l, int r, int left, int right){
   if (l <= left && right <= r) return RT[idx];
   int ret = INF;
   if (l <= MID) ret = min(ret, Query(2*idx, l, r, left, MID));
   if (r > MID) ret = min(ret, Query(2*idx+1, l, r, MID+1, right));
   return ret;
}

int main(){
  n = 6;
  niz[1] = 4;
  niz[2] = 2;
  niz[3] = 5;
  niz[4] = 1;
  niz[5] = 6;
  niz[6] = 3;

  InitTree(1, 1, n);
  printf("%d\n",Query(1, 1, 3, 1, n));
  Update(1, 4, 10, 1, n);
  Update(1, 5, 0, 1, n);
  printf("%d\n",Query(1, 4, 6, 1, n));

  return 0;
}
```

4.2. Java

```
class RangeTree{
  int [] niz;
  int [] RT;
  final int INF = 987654321;

  public RangeTree(int [] _array){
     niz = Arrays.copyOf(_array, _array.length);
     RT = new int [niz.length << 2];
  }

  public RangeTree(int _n){
     niz = new int [_n];
     RT = new int [_n << 2];
  }
```

```
  public void initTree(int idx, int left, int right){
     if(left == right){ RT[idx] = niz[left];}
     else{
       int MID =(left+right)/2;
       initTree(2*idx, left, MID);
       initTree(2*idx+1, MID+1, right);
       RT[idx] = Math.min(RT[2*idx], RT[2*idx+1]);
     }
  }

  public void update(int idx, int x, int val, int left, int right){
     if(left == right){ RT[idx] = val;}
     else{
       int MID =(left+right)/2;
       if (x <= MID) update(2*idx, x, val, left, MID);
       else update(2*idx+1, x, val, MID+1, right);
       RT[idx] = Math.min(RT[2*idx], RT[2*idx+1]);
     }
  }

  public int query(int idx, int l, int r, int left, int right){
     if (l <= left && right <= r) return RT[idx];
         int ret = INF;
     int MID =(left+right)/2;
     if (l <= MID) ret = Math.min(ret, query(2*idx, l, r, left, MID));
     if (r > MID) ret = Math.min(ret, query(2*idx+1, l, r, MID+1, right));
        return ret;
  }
}
```

5. Complejidad

Vamos analizar la complejidad la complejidad de esta estructura de datos acorde a cada una de las operaciones que la misma presenta:

Construción Un árbol de rango en un conjunto de n elementos es un árbol de búsqueda binario, que se puede construir en el tiempo O ($n \log n$)

Actualización Un árbol de rango en un conjunto de n elementos es un árbol de búsqueda binario, que se puede realizar una actualización en un tiempo de O($\log n$)

Intervalo de consultas Árboles de rango se puede utilizar para encontrar el conjunto de elementos que se encuentran dentro de un intervalo dado. Para informar los puntos que se encuentran en el intervalo [x_1 , x_2], comenzamos por buscar x_1 y x_2 . En algún vértice del árbol, las rutas de búsqueda a x_1 y x_2 divergirán. Sea v split el último vértice que estos dos caminos de búsqueda

tienen en común. Continúe buscando x_1 en el árbol de rangos. Para cada vértice v en la ruta de búsqueda de v dividida a x_1 , si el valor almacenado en v es mayor que x_1 , reporte cada elemento en el subárbol derecho de v . Si v es una hoja, informe el valor almacenado en v si está dentro del intervalo de consulta. Del mismo modo, reportar todos los puntos almacenados en los subárboles izquierdos de los vértices con valores menores que x_2 a lo largo de la ruta de búsqueda de v dividida a x_2 e informar la hoja de esta ruta si se encuentra dentro del intervalo de consulta.

Dado que el árbol de rango es un árbol binario balanceado, las rutas de búsqueda a x_1 y x_2 tienen longitud O ($\log n$). La generación de informes de todos los elementos almacenados en el subárbol de un vértice se puede hacer en tiempo lineal utilizando cualquier algoritmo de recorrido de árbol. Se deduce que el tiempo para realizar una consulta de rango es O ($\log n + k$), donde k es el número de elementos en el intervalo de consulta.

Una propiedad importante de los árboles de rangos es que solo requieren una cantidad lineal de memoria. El árbol de segmentos estándar requiere $4n$ vértices para trabajar en una matriz de tamaño n.

6. Aplicaciones

En general, un árbol de rangos es una estructura de datos muy flexible y se puede resolver una gran cantidad de problemas con él. Además, también es posible aplicar operaciones más complejas y responder consultas más complejas. En particular, el árbol de segmentos se puede generalizar fácilmente a dimensiones más grandes. Por ejemplo, con un árbol de rangos bidimensional, puede responder consultas de suma o mínimo sobre algún subrectángulo de una matriz dada. Sin embargo, solo en tiempo O($\log^2 N$).

7. Ejercicios propuestos

A continuación una lista de ejercicios que se resuelven utilizando esta estructura:

- DMOJ - GCD en el Arreglo.
- DMOJ - Robin Hood Curioso
- DMOJ - LLaves
- DMOJ - Vigilando el Museo