GUÍA DE APRENDIZAJE PARA CONCURSANTES ICPC Y IOI: ÁRBOL DE RANGO VERSIONES AVANZADAS

1. Introducción

Un árbol de rango es una estructura de datos muy flexible y permite variaciones y extensiones en muchas direcciones diferentes. Intentemos categorizarlos a continuación.

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

Puede ser bastante fácil cambiar el árbol de segmentos en una dirección, de modo que calcule diferentes consultas (por ejemplo, calcular el mínimo/máximo en lugar de la suma), pero también puede ser muy no trivial.

3.1. Encontrar el máximo

El árbol tendrá exactamente la misma estructura que el árbol descrito anteriormente. Sólo tenemos que cambiar la forma $t[v]$ se calcula en el construir y actualizar funciones $t[v]$ ahora almacenará el máximo del rango correspondiente. Y también tenemos que cambiar el cálculo del valor devuelto de la función consulta.

Por supuesto, este problema se puede cambiar fácilmente para calcular el mínimo en lugar del máximo.

En lugar de mostrar una implementación de este problema, la implementación se dará a una versión más compleja de este problema en la siguiente sección.

3.2. Hallar el máximo y el número de veces que aparece

Esta tarea es muy similar a la anterior. Además de encontrar el máximo, también tenemos que encontrar el número de ocurrencias del máximo.

Para resolver este problema, almacenamos un par de números en cada vértice del árbol: Además del máximo, también almacenamos el número de ocurrencias del mismo en el rango correspondiente. Determinar el par correcto para almacenar en $t[v]$ todavía se puede hacer en tiempo constante usando la información de los pares almacenados en los vértices secundarios. La combinación de dos pares de este tipo debe hacerse en una función separada, ya que esta será una operación que haremos mientras construimos el árbol, mientras respondemos al máximo de consultas y mientras realizamos modificaciones.

3.3. Calcular el máximo común divisor/mínimo común múltiplo

En este problema queremos calcular el GCD / LCM de todos los números de rangos dados de la matriz.

Esta interesante variación del árbol de rango se puede resolver exactamente de la misma manera que los árboles de rangos que derivamos para las consultas de suma/mínimo/máximo: basta con almacenar el GCD/LCM del vértice correspondiente en cada vértice del árbol. La combinación de dos vértices se puede hacer calculando el GCD / LCM de ambos vértices.

3.4. Contando el número de ceros, buscando el k-th cero

En este problema queremos encontrar el número de ceros en un rango dado y, además, encontrar el índice de la k-th cero usando una segunda función.

Nuevamente tenemos que cambiar un poco los valores de almacenamiento del árbol: Esta vez almacenaremos el número de ceros en cada segmento en $t[]$. Está bastante claro, cómo implementar el construir, actualizar y la función cuenta_cero, simplemente podemos usar las ideas del problema de consulta de suma. Así resolvimos la primera parte del problema.

Ahora aprenderemos a resolver el problema de encontrar el k-th cero en el arreglo $a[]$. Para realizar esta tarea, descenderemos por el árbol de segmentos, comenzando en el vértice de la raíz y moviéndonos cada vez hacia el elemento secundario izquierdo o derecho, según el segmento que contenga el elemento k-th cero. Para decidir a qué niño debemos ir, basta con mirar la cantidad de ceros que aparecen en el segmento correspondiente al vértice izquierdo. Si este conteo precalculado es mayor o igual a k, es necesario descender al hijo izquierdo, y en caso contrario descender al hijo derecho. Fíjate, si elegimos al hijo de la derecha, tenemos que restar el número de ceros del hijo de la izquierda de k.

En la implementación podemos manejar el caso especial, $a[]$ que contiene menos de k ceros, devolviendo -1.

3.5. Buscando un prefijo de arreglo con una cantidad dada

La tarea es la siguiente: para un valor dado x tenemos que encontrar rápidamente el índice más pequeño i tal que la suma de los primeros i elementos del arreglo $a[]$ es mayor o igual a x (suponiendo que el arreglo $a[]$ solo contiene valores no negativos).

Esta tarea se puede resolver mediante la búsqueda binaria, calculando la suma de los prefijos con el árbol de rangos. Sin embargo, esto conducirá a una $O(\log^2 n)$ solución.

En cambio, podemos usar la misma idea que en la sección anterior y encontrar la posición descendiendo del árbol: moviéndonos cada vez hacia la izquierda o hacia la derecha, dependiendo de la suma del hijo izquierdo. Encontrando así la respuesta en $O(\log n)$ tiempo.

3.6. Buscando el primer elemento mayor que una cantidad dada

La tarea es la siguiente: para un valor dado x y un rango $a[l \dots r]$ encontrar el más pequeño i en el rango $a[l \dots r]$, tal que $a[i]$ es mayor que x.

Esta tarea se puede resolver mediante la búsqueda binaria sobre consultas de prefijo máximo con el árbol de rango. Sin embargo, esto dará lugar a una $O(\log^2 n)$ solución.

En su lugar, podemos usar la misma idea que en las secciones anteriores y encontrar la posición descendiendo del árbol: moviéndonos cada vez hacia la izquierda o hacia la derecha, dependiendo del valor máximo del hijo izquierdo. Encontrando así la respuesta en $O(\log n)$ tiempo.

3.7. Encontrar subrangos con la suma máxima

Aquí nuevamente recibimos un rango $a[l \dots r]$ para cada consulta, esta vez tenemos que encontrar un subrango $a[l' \dots r']$ tal que $l \leq l'$ y $r' \leq r$ y la suma de los elementos de este rango es máxima. Como antes, también queremos poder modificar elementos individuales del arreglo. Los elementos del arreglo pueden ser negativos y el subrango óptimo puede estar vacío (por ejemplo, si todos los elementos son negativos).

Este problema es un uso no trivial de un árbol de rangos. Esta vez almacenaremos cuatro valores para cada vértice: la suma del rango, la suma máxima del prefijo, la suma máxima del sufijo y la suma del subrango máximo en él. En otras palabras, para cada rango del árbol de rangos, la respuesta ya está precalculada, así como las respuestas para los rangos que tocan los límites izquierdo y derecho del rango.

¿Cómo construir un árbol con tales datos? Nuevamente lo calculamos de forma recursiva: primero calculamos los cuatro valores para el hijo izquierdo y derecho, y luego los combinamos para archivar los cuatro valores para el vértice actual. Tenga en cuenta que la respuesta para el vértice actual es:

- La respuesta del hijo izquierdo, lo que significa que el subsegmento óptimo se coloca completamente en el segmento del hijo izquierdo.
- La respuesta del niño derecho, lo que significa que el subsegmento óptimo se coloca completamente en el segmento del niño derecho.
- La suma de la suma máxima de sufijos del hijo izquierdo y la suma máxima de prefijos del hijo derecho, lo que significa que el subsegmento óptimo se cruza con ambos hijos.

Por lo tanto, la respuesta al vértice actual es el máximo de estos tres valores. Calcular la suma máxima de prefijo/sufijo es aún más fácil.

Utilizando la función combine es fácil construir el árbol de rangos. Podemos implementarlo exactamente de la misma manera que en las implementaciones anteriores. Para inicializar los vértices de las hojas, creamos adicionalmente la función auxiliar make_data, que devolverá un objeto data que contiene la información de un solo valor.

Sólo queda, cómo calcular la respuesta a una consulta. Para responderla, descendemos por el árbol como antes, dividiendo la consulta en varios subrangoss que coinciden con los rangos del árbol de rangos, y combinamos las respuestas en ellos en una sola respuesta para la consulta. Entonces debe quedar claro, que el trabajo es exactamente el mismo que en el árbol de rangos simple, pero en lugar de sumar/minimizar/maximizar los valores, usamos la función combinar.

3.8. Guardando los subarreglos completos en cada vértice.

Esta es una subsección separada que se distingue de las demás, porque en cada vértice del árbol de rangos no almacenamos información sobre el rango correspondiente en forma comprimida (suma, mínimo, máximo, ...), pero almacenamos todos los elementos del rango Por lo tanto, la raíz del árbol de segmentos almacenará todos los elementos del arreglo, el vértice secundario izquierdo almacenará la primera mitad del arreglo, el vértice derecho la segunda mitad, y así sucesivamente.

En su aplicación más simple de esta técnica, almacenamos los elementos en orden ordenado. En versiones más complejas los elementos no se almacenan en listas, sino en estructuras de datos más avanzadas (conjuntos, mapas,...). Pero todos estos métodos tienen el factor común, que cada vértice requiere memoria lineal (es decir, proporcional a la longitud del segmento correspondiente).

La primera pregunta natural, al considerar estos árboles de rangos, es sobre el consumo de memoria. Intuitivamente esto podría parecer $O(n^2)$ memoria, pero resulta que el árbol completo solo necesitará $O(n \log n)$ memoria. ¿Por qué esto es tan? Sencillamente, porque cada elemento del arreglo cae en $O(\log n)$ rangos (recuerde que la altura del árbol es $O(\log n)$).

Entonces, a pesar de la aparente extravagancia de un árbol de rango de este tipo, consume solo un poco más de memoria que el árbol de rangos habitual.

3.8.1. Encuentra el número más pequeño mayor o igual a un número específico. Sin consultas de modificación.

Queremos responder consultas de la siguiente forma: para tres números dados (l, r, x) tenemos que encontrar el número mínimo en el rango $a[l \dots r]$ que es mayor o igual que x.

Construimos un árbol de rango. En cada vértice almacenamos una lista ordenada de todos los números que ocurren en el rango correspondiente, como se describe arriba. ¿Cómo construir un árbol de rango de la manera más eficaz posible? Como siempre, abordamos este problema de forma recursiva: dejemos que las listas de los hijos izquierdo y derecho ya estén construidas, y queremos construir la lista para el vértice actual. Desde esta vista, la operación ahora es trivial y se puede realizar en tiempo lineal: solo necesitamos combinar las dos listas ordenadas en una, lo que se puede hacer iterando sobre ellas usando dos punteros. El STL de C++ ya tiene una implementación de este algoritmo.

Debido a esta estructura del árbol de rango y las similitudes con el algoritmo de ordenación por fusión, la estructura de datos también se suele denominar *árbol de clasificación por fusión*.

Ya sabemos que el árbol de rango así construido requerirá $O(n \log n)$ memoria. Y gracias a esta implementación su construcción también lleva $O(n \log n)$ tiempo, después de todo cada lista se construye en tiempo lineal con respecto a su tamaño.

Ahora considere la respuesta a la consulta. Bajaremos por el árbol, como en el árbol de rango regular, rompiendo nuestro segmento $a[l \dots r]$ en varios subsegmentos (como máximo $O(\log n)$ piezas). Es claro que la respuesta de toda la respuesta es el mínimo de cada una de las subconsultas. Entonces, ahora solo necesitamos entender cómo responder a una consulta en uno de esos subrangos que se corresponde con algún vértice del árbol.

Estamos en algún vértice del árbol de rango y queremos calcular la respuesta a la consulta, es decir, encontrar el número mínimo mayor o igual a un número dado x. Dado que el vértice contiene la lista de elementos ordenados, podemos simplemente realizar una búsqueda binaria en esta lista y devolver el primer número, mayor o igual que x.

Así, la respuesta a la consulta en un segmento del árbol toma $O(\log n)$ tiempo, y toda la consulta se procesa en $O(\log^2 n)$.

3.8.2. Encuentra el número más pequeño mayor o igual a un número específico. Con consultas de modificación.

Esta tarea es similar a la anterior. El último enfoque tiene una desventaja, no fue posible modificar el arreglo entre las consultas de respuesta. Ahora queremos hacer exactamente esto: una consulta de modificación hará la asignación $a[i] = y$.

La solución es similar a la solución del problema anterior, pero en lugar de listas en cada vértice del árbol de rango, almacenaremos una lista equilibrada que le permite buscar números rápidamente, eliminar números e insertar nuevos números. Dado que el arreglo puede contener un número repetido, la elección óptima es la estructura de datos multiset.

La construcción de un árbol de rango de este tipo se realiza prácticamente de la misma manera que en el problema anterior, solo que ahora necesitamos combinar multiset y no listas ordenadas. Esto conduce a un tiempo de construcción de $O(n \log^2 n)$ (en general, la combinación de dos árboles rojo-negro se puede hacer en tiempo lineal, pero C++ STL no garantiza esta complejidad de tiempo).

La función consulta también es casi equivalente, solo que ahora en vez de lower_bound del multiset en su lugar, se debe llamar a la función (std::lower_bound solo funciona en $O(\log n)$ time si se usa con iteradores de acceso aleatorio).

Finalmente la solicitud de modificación. Para procesarlo, debemos bajar por el árbol, y modificar todos multiset de los rangos correspondientes que contienen el elemento afectado. Simplemente eliminamos el valor anterior de este elemento (pero solo una ocurrencia) e insertamos el nuevo valor.

El procesamiento de esta consulta de modificación también toma $O(\log^2 n)$ tiempo.

3.8.3. Encuentra el número más pequeño mayor o igual a un número específico. Aceleración con *cascada fraccionada*.

Tenemos el mismo enunciado del problema, queremos encontrar el número mínimo mayor o igual que x en un segmento, pero esta vez en $O(\log n)$ tiempo. Mejoraremos la complejidad temporal utilizando la técnica de *cascada fraccionaria*.

La cascada fraccional es una técnica simple que le permite mejorar el tiempo de ejecución de múltiples búsquedas binarias, que se realizan al mismo tiempo. Nuestro enfoque anterior a la consulta de búsqueda fue que dividimos la tarea en varias subtareas, cada una de las cuales se resuelve con una búsqueda binaria. La cascada fraccional le permite reemplazar todas estas búsquedas binarias con una sola.

El ejemplo más simple y más obvio de cascada fraccionaria es el siguiente problema: hay k listas ordenadas de números, y debemos encontrar en cada lista el primer número mayor o igual al número dado. En lugar de realizar una búsqueda binaria para cada lista, podríamos fusionar todas las listas en una gran lista ordenada. Adicionalmente para cada elemento y almacenamos una lista de resultados de la búsqueda de y en cada uno de los k lista. Por lo tanto, si queremos encontrar el número más pequeño mayor o igual que x, solo necesitamos realizar una sola búsqueda binaria, y de la lista de índices podemos determinar el número más pequeño en cada lista. Sin embargo, este enfoque requiere $O(n \cdot k)$ (n es la longitud de las listas combinadas), que puede ser bastante ineficiente.

La cascada fraccional reduce esta complejidad de la memoria a $O(n)$ memoria, creando a partir de la k listas de entrada k nuevas listas, en las que cada lista contiene la lista correspondiente y, además, también cada segundo elemento de la siguiente lista nueva. Usando esta estructura solo es necesario almacenar dos índices, el índice del elemento en la lista original y el índice del elemento en la siguiente lista nueva. Entonces, este enfoque solo usa $O(n)$ memoria, y todavía puede responder a las consultas mediante una única búsqueda binaria.

Pero para nuestra aplicación no necesitamos toda la potencia de la cascada fraccionaria. En nuestro árbol de rango, un vértice contendrá la lista ordenada de todos los elementos que se encuentran en los subárboles izquierdo o derecho (como en el árbol de clasificación combinado). Además de esta lista ordenada, almacenamos dos posiciones para cada elemento. para un elemento y almacenamos el índice más pequeño i, tal que el i el elemento en la lista ordenada del hijo izquierdo es mayor o igual que y. Y almacenamos el índice más pequeño. j , tal que el j el elemento en la lista ordenada del hijo derecho es mayor o igual que y. Estos valores se pueden calcular en paralelo al paso de fusión cuando construimos el árbol.

¿Cómo acelera esto las consultas?

Recuerde, en la solución normal hicimos una búsqueda binaria en cada nodo. Pero con esta modificación, podemos evitar todos menos uno.

Para responder a una consulta, simplemente hacemos una búsqueda binaria en el nodo raíz. Esto da como el elemento más pequeño $y \geq x$ en el arreglo completo, pero también nos da dos posiciones. El índice del elemento más pequeño mayor o igual x en el subárbol izquierdo, y el índice del elemento más pequeño y en el subárbol derecho. Darse cuenta de $\geq y$ es lo mismo que

$\geq x$, ya que nuestro arreglo no contiene ningún elemento entre x y y. En la solución Merge Sort Tree normal, calcularíamos estos índices a través de la búsqueda binaria, pero con la ayuda de los valores precalculados, podemos buscarlos en $O(1)$. Y podemos repetir eso hasta que visitemos todos los nodos que cubren nuestro intervalo de consulta.

Para resumir, como de costumbre tocamos $O(\log n)$ nodos durante una consulta. En el nodo raíz hacemos una búsqueda binaria, y en todos los demás nodos solo hacemos un trabajo constante. Esto significa que la complejidad para responder una consulta es $O(\log n)$.

Pero tenga en cuenta que esto usa tres veces más memoria que un árbol de ordenación de combinación normal, que ya usa mucha memoria ($O(n \log n)$).

Es sencillo aplicar esta técnica a un problema, que no requiere consultas de modificación. Las dos posiciones son solo números enteros y se pueden calcular fácilmente contando al fusionar las dos secuencias ordenadas.

Todavía es posible permitir consultas de modificación, pero eso complica todo el código. En lugar de números enteros, debe almacenar la matriz ordenada como multisety, en lugar de índices, debe almacenar iteradores. Y debe trabajar con mucho cuidado, de modo que incremente o disminuya los iteradores correctos durante una consulta de modificación.

4. Implementación

4.1. C++

4.1.1. Hallar el máximo y el número de veces que aparece

```
pair<int, int> t[4*MAXN];

pair<int, int> combine(pair<int, int> a, pair<int, int> b) {
   if (a.first > b.first) return a;
   if (b.first > a.first) return b;
   return make_pair(a.first, a.second + b.second);
}

void build(int a[], int v, int tl, int tr) {
   if (tl == tr) { t[v] = make_pair(a[tl], 1);}
   else {
      int tm = (tl + tr) / 2;
      build(a, v*2, tl, tm); build(a, v*2+1, tm+1, tr);
      t[v] = combine(t[v*2], t[v*2+1]);
   }
}

pair<int, int> get_max(int v, int tl, int tr, int l, int r) {
   if (l > r) return make_pair(-INF, 0);
   if (l == tl && r == tr) return t[v];
   int tm = (tl + tr) / 2;
```

```
   return combine(get_max(v*2, tl, tm, l, min(r, tm)),
   get_max(v*2+1, tm+1, tr, max(l, tm+1), r));
}

void update(int v, int tl, int tr, int pos, int new_val) {
   if (tl == tr) { t[v] = make_pair(new_val, 1);}
   else {
      int tm = (tl + tr) / 2;
      if (pos <= tm) update(v*2, tl, tm, pos, new_val);
      else update(v*2+1, tm+1, tr, pos, new_val);
      t[v] = combine(t[v*2], t[v*2+1]);
   }
}
```

4.1.2. Contando el número de ceros, buscando el k-th cero

```
int find_kth(int v, int tl, int tr, int k) {
   if (k > t[v]) return -1;
   if (tl == tr) return tl;
   int tm = (tl + tr) / 2;
   if (t[v*2] >= k) return find_kth(v*2, tl, tm, k);
   else return find_kth(v*2+1, tm+1, tr, k - t[v*2]);
}
```

4.1.3. Buscando el primer elemento mayor que una cantidad dada

```
int get_first(int v, int lv, int rv, int l, int r, int x) {
   if(lv > r || rv < l) return -1;
   if(l <= lv && rv <= r) {
      if(t[v] <= x) return -1;
      while(lv != rv) {
         int mid = lv + (rv-lv)/2;
         if(t[2*v] > x) { v = 2*v; rv = mid;}
         else { v = 2*v+1; lv = mid+1;}
      }
      return lv;
    }
    int mid = lv + (rv-lv)/2;
    int rs = get_first(2*v, lv, mid, l, r, x);
    if(rs != -1) return rs;
    return get_first(2*v+1, mid+1, rv, l ,r, x);
}
```

4.1.4. Encontrar subrangos con la suma máxima

```
struct data { int sum, pref, suff, ans; };

data combine(data l, data r) {
   data res;
   res.sum = l.sum + r.sum;
   res.pref = max(l.pref, l.sum + r.pref);
   res.suff = max(r.suff, r.sum + l.suff);
   res.ans = max(max(l.ans, r.ans), l.suff + r.pref);
   return res;
}

data make_data(int val) {
   data res; res.sum = val;
   res.pref = res.suff = res.ans = max(0, val);
   return res;
}

void build(int a[], int v, int tl, int tr) {
   if (tl == tr) { t[v] = make_data(a[tl]);}
   else {
      int tm = (tl + tr) / 2;
      build(a, v*2, tl, tm); build(a, v*2+1, tm+1, tr);
      t[v] = combine(t[v*2], t[v*2+1]);
   }
}

void update(int v, int tl, int tr, int pos, int new_val) {
   if (tl == tr) { t[v] = make_data(new_val);}
   else {
      int tm = (tl + tr) / 2;
      if (pos <= tm) update(v*2, tl, tm, pos, new_val);
      else update(v*2+1, tm+1, tr, pos, new_val);
      t[v] = combine(t[v*2], t[v*2+1]);
   }
}

data query(int v, int tl, int tr, int l, int r) {
   if (l > r) return make_data(0);
   if (l == tl && r == tr) return t[v];
   int tm = (tl + tr) / 2;
   return combine(query(v*2,tl,tm,l,min(r,tm)),
                  query(v*2+1,tm+1,tr,max(l,tm+1),r));
}
```

4.1.5. Encuentra el número más pequeño mayor o igual a un número específico. Sin consultas de modificación.

```
vector<int> t[4*MAXN];
```

```
void build(int a[], int v, int tl, int tr) {
   if (tl == tr) { t[v] = vector<int>(1, a[tl]);}
   else {
      int tm = (tl + tr) / 2;
      build(a, v*2, tl, tm); build(a, v*2+1, tm+1, tr);
      merge(t[v*2].begin(),t[v*2].end(),t[v*2+1].begin(),t[v*2+1].end(),
         back_inserter(t[v]));
   }
}

int query(int v, int tl, int tr, int l, int r, int x) {
   if (l > r) return INF;
   if (l == tl && r == tr) {
      vector<int>::iterator pos = lower_bound(t[v].begin(), t[v].end(), x);
      if (pos != t[v].end()) return *pos;
      return INF;
   }
   int tm = (tl + tr) / 2;
   return min(query(v*2,tl,tm,l,min(r,tm),x),
              query(v*2+1,tm+1,tr,max(l,tm+1),r,x));
}
```

La constante INF es igual a un número grande que es mayor que todos los números en el arreglo. Su uso significa que no hay número mayor o igual que x en el segmento Tiene el signifcado de *no hay respuesta en el intervalo dado.*

4.1.6. Encuentra el número más pequeño mayor o igual a un número específico. Con consultas de modificación.

```
void update(int v, int tl, int tr, int pos, int new_val) {
   t[v].erase(t[v].find(a[pos]));t[v].insert(new_val);
   if (tl != tr) {
      int tm = (tl + tr) / 2;
      if (pos <= tm) update(v*2, tl, tm, pos, new_val);
      else update(v*2+1, tm+1, tr, pos, new_val);
        } else { a[pos] = new_val; }
}
```

5. Complejidad

A pesar de las diferentes variantes y modificaciones analizadas de la estructura árbol de rango el mismo no altera la complejidad de cada una de sus operaciones clásicas en la mayoría de los casos. Solo en algunos casos la complejidad de algunas de sus operaciones puede variar a hasta $O(n \log^2 n)$.

6. Aplicaciones

No cabe duda que después de lo abordado anteriormente en esta guía las aplicaciones y versatilidad de esta estructura aumenta notablemente lo que hace que sea una herramienta importante en el arsenal de un concursante a la hora de resolver los ejercicios o problemas.

7. Ejercicios propuestos

A continuación una lista de ejercicios que se solucionan utilizando la estructura de árbol de rango con algunas de las modificaciones vistas en esta guía:

- DMOJ - Buscando compradores
- DMOJ - Mirando