GUÍA DE APRENDIZAJE PARA CONCURSANTES ICPC Y IOI: DESCOMPOSICIÓN RAÍZ CUADRADA (*SQRT DECOMPOSITION*)

---

1. Introducción

Descomposición raíz cuadrada (*Sqrt Decomposition*) es un método (o una estructura de datos) que le permite realizar algunas operaciones comunes (encontrar la suma de los elementos del sub-array, encontrar el elemento mínimo/máximo, etc.) en operaciones, que es mucho más rápido que para el algoritmo trivial.

Anteriormente resolvimos el problema utilizando árboles binarios indexados y de rangos, que admiten ambas operaciones en tiempo O ($\log N$).

2. Conocimientos previos

2.1. Raíz cuadradada

En las matemáticas, la raíz cuadrada de un número x es aquel número y que al ser multiplicado por sí mismo da como resultado el valor x, es decir, cumple la ecuación $y^2 = x$. Tanto en C++ como Java se puede hallar el valor con el uso de la función *sqrt*.

3. Desarrollo

Primero describimos la estructura de datos para una de las aplicaciones más simples de esta idea, luego mostramos cómo generalizarla para resolver algunos problemas.

Dado un arreglo $a[0 \dots n-1]$, implemente una estructura de datos que permita encontrar la suma de los elementos $a[l \dots r]$ para arbitrarios l y r.

La idea básica de la descomposición de raíz cuadrada es preprocesamiento. Dividiremos el arreglo a en bloques de longitud aproximadamente $\sqrt{n}$, y por cada bloque i precalcularemos la suma de elementos en $b[i]$. Por ejemplo, una matriz de 16 elementos dividirse en bloques de 4 elementos de la siguiente manera:

21				15				20				13			
5	8	6	3	2	5	2	6	7	1	7	5	6	2	3	2

Podemos asumir que tanto el tamaño del bloque como el número de bloques son iguales a $\sqrt{n}$ redondeado:

$$s = \lceil \sqrt{n} \rceil$$

Luego el arreglo a se dividirá en bloques de la siguiente manera:

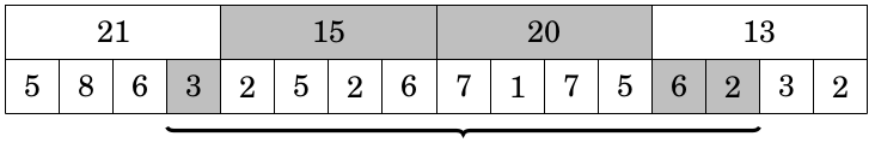
$$\underbrace{a[0], a[1], \dots, a[s-1]}_{\text{b[0]}}, \underbrace{a[s], \dots, a[2s-1]}_{\text{b[1]}}, \dots, \underbrace{a[(s-1) \cdot s], \dots, a[n-1]}_{\text{b[s-1]}}$$

El último bloque puede tener menos elementos que los demás (si n no un múltiplo de s), no es importante para la discusión (como se puede manejar fácilmente). Por lo tanto, por cada bloque k , conocemos la suma de elementos en él $b[k]$.

$$b[k] = \sum_{i=k\cdot s}^{\text{mín}\,(n-1,(k+1)\cdot s-1)} a[i]$$

Por lo tanto, hemos calculado los valores de $b[k]$ en O(n). ¿Cómo pueden ayudarnos a responder a cada consulta $[l, r]$? Observe que si el intervalo $[l, r]$ es lo suficientemente largo, contendrá varios bloques enteros, y para esos bloques podemos encontrar la suma de elementos en ellos en una sola operación. Como resultado, el intervalo $[l, r]$ contendrá partes de sólo dos bloques, y tendremos que calcular la suma de elementos en estas partes trivialmente.

Por lo tanto, para calcular la suma de elementos en el intervalo $[l, r]$ sumar los elementos de las dos partes: $a[l \ldots (k + 1) \cdot s - 1]$ y $a[p \cdot s \ldots r]$, y sumar los valores $b[i]$ en todos los bloques de $k + 1$ to $p - 1$. Cuando $K = P$, es decir, L y R pertenecen al mismo bloque, la fórmula no se puede aplicar y la suma debe calcularse de manera trivial.



Este enfoque nos permite reducir significativamente el número de operaciones.De hecho, el tamaño de cada parte no es la longitud del bloque s, y el número de bloques en la suma no excede de s.Dado que se elige a $s \approx \sqrt{n}$, el número total de operaciones necesarias para encontrar la suma de elementos en el intervalo $[l, r]$ es $O(\sqrt{n})$.

4. Implementación

4.1. C++

```
// Datos de entradas
int n;
vector<int> a,b;

//lectura de datos
void readData(){
   a.resize(n);
   for(int i=0;i<n;i++) cin>>a[i];
}

// preprocesamiento
void init(){
```

```
   int len = (int) sqrt (n + .0) + 1; // longitud del bloque y numero de
      bloques
   b.resize(len);
   for (int i=0; i<n; ++i) b[i / len] += a[i];
}

int query(int l,int r){
   int sum = 0;
   int len = (int) sqrt (n + .0) + 1;
   for (int i=l; i<=r; ){
      if (i % len == 0 && i + len - 1 <= r) {
         // Si todo el bloque que comienza en i pertenece a [l, r]
         sum += b[i / len]; i += len;
      } else {
         sum += a[i]; ++i;
      }
   }
   return sum;
}

int queryv2(int l, int r){
   int len = (int) sqrt (n + .0) + 1;
   int sum = 0;
   int c_l = l / len,   c_r = r / len;
   if (c_l == c_r) for (int i=l; i<=r; ++i) sum += a[i];
   else {
      for (int i=l, ends=(c_l+1)*len-1; i<=ends; ++i) sum += a[i];
      for (int i=c_l+1; i<=c_r-1; ++i) sum += b[i];
      for (int i=c_r*len; i<=r; ++i) sum += a[i];
   }
   return sum;
}
```

4.2. Java

```
public class Main {
   // Datos de entradas
   int n;
   int[] a;
   int [] b;

   void readData(){
      a = new int [n];
      Scanner in =new Scanner(System.in);
      for(int i=0;i<n;i++) a[i]=in.nextInt();
   }

   // preprocesamiento
```

```
    public void init(){
        int len = (int) Math.sqrt (n + .0) + 1; // longitud del bloque y numero
            de bloques
        b = new int [len];
        for (int i=0; i<n; ++i) b[i / len] += a[i];
    }

    public int query(int l,int r){
        int sum = 0;
        int len = (int) Math.sqrt (n + .0) + 1;
        for (int i=l; i<=r; ){
            if (i % len == 0 && i + len - 1 <= r) {
                // Si todo el bloque que comienza en i pertenece a [l, r]
                sum += b[i / len]; i += len;
            } else {
                sum += a[i]; ++i;
            }
        }
        return sum;
    }

    public int queryv2(int l, int r){
        int len = (int) Math.sqrt (n + .0) + 1;
        int sum = 0;
        int c_l = l / len,    c_r = r / len;
        if (c_l == c_r) for (int i=l; i<=r; ++i) sum += a[i];
        else {
            for (int i=l, ends=(c_l+1)*len-1; i<=ends; ++i) sum += a[i];
            for (int i=c_l+1; i<=c_r-1; ++i) sum += b[i];
            for (int i=c_r*len; i<=r; ++i) sum += a[i];
        }
        return sum;
    }
}
```

La implementación *query* tiene demasiadas operaciones de división (que son mucho más lentas que otras operaciones aritméticas). En su lugar, podemos calcular los índices de los bloques c_l y c_r que contienen los índices l y r, y recorrer los bloques $c_l + 1 \dots c_r - 1$ con procesamiento separado de las partes en los bloques c_l y c_r. Este enfoque corresponde a la última fórmula de la descripción y hace que el caso $c_l = c_r$ sea un caso especial en este caso tenemos la implementación *queryv2*.

5. Aplicaciones

Hasta ahora estábamos discutiendo el problema de encontrar la suma de los elementos de una subarrray continua. Este problema se puede extender para permitir actualizar elementos de un arreglo. Si un elemento $a[i]$ cambia, es suficiente actualizar el valor de $b[k]$ para el bloque al que pertenece este elemento ($k = i/s$) en una operación:

$$b[k] += a_{nuevo}[i] - a_{viejo}[i]$$

Por otro lado, la tarea de encontrar la suma de elementos puede reemplazarse con la tarea de encontrar un elemento mínimo/máximo de una subarreglo. Si este problema también tiene que abordar las actualizaciones de los elementos individuales, también es posible actualizar el valor de $b[k]$, pero requerirá iterarse a través de todos los valores de bloque k en $O(s) = O($ $sqrtn)$ operaciones.

La descomposición de raíz cuadrada se puede aplicar de manera similar a una clase completa de otros problemas: encontrar el número de elementos cero, encontrar el primer elemento distinto de cero, contando elementos que satisfacen una determinada propiedad, etc.

Otra clase de problemas aparece cuando necesitamos **actualizar los elementos de un arreglo en intervalos**: incrementar los elementos existentes o reemplazarlos con un valor dado.

Por ejemplo, supongamos que podemos hacer dos tipos de operaciones en un arreglo: adicionar un valor dado δ a todos los elementos del arreglo en el intervalo $[l, r]$ o consulte el valor del elemento $a[i]$. Generemos el valor que debe agregarse a todos los elementos de bloque k en $b[k]$ (inicialmente todos $b[k] = 0$). Durante cada operación de adicionar, necesitamos agregar δ a $b[k]$ para todos los bloques que pertenecen a intervalo $[l, r]$ y para agregar δ a $a[i]$ por Todos los elementos que pertenecen a las partes del intervalo. La respuesta una consulta i es simplemente $a[i] + b[i/s]$. De esta manera, la operación Adicionar tiene O($\sqrt{n}$) complejidad, y responder una consulta tiene O(1) complejidad.

Finalmente, esas dos clases de problemas se pueden combinar si la tarea requiere realizar **ambas** actualizaciones de elementos en un intervalo y consultas en un intervalo. Ambas operaciones se pueden hacer con una complejidad de O($\sqrt{n}$). Esto requerirá dos arreglos de bloque b y c: uno para realizar un seguimiento de las actualizaciones de elementos y otro para realizar un seguimiento de las respuestas a la consulta.

Existen otros problemas que se pueden resolver utilizando la descomposición de raíz cuadrada, por ejemplo, un problema sobre mantener un conjunto de números que permitirían agregar/eliminar números, verificando si un número pertenece al conjunto y encontrar K-th más grande. Para resolverlo, uno tiene que almacenar números en orden creciente, dividido en varios bloques con $\sqrt{n}$ números en cada uno. Cada vez que se agrega/elimina un número, los bloques deben reequilarse mediante números móviles entre comienzos y extremos de bloques adyacentes.

6. Complejidad

La complejidad de esta estructura de datos la podemos analizar acorde a sus operaciones clásicas que son la de construcción y respuesta a las consultas. La operación construcción tiene una complejidad de O(N) mientras la operación de consulta su complejidad sera de O($\sqrt{N}$).

7. Ejercicios

A continuación una lista de ejercicios que se pueden resolver aplicando esta estructura de datos

- CodeForces - E. Xenia and Tree
- CodeForces - C. Holidays
- CodeForces - E. Holes
- CodeForces - D. Powerful array
- CodeForces - D. Instant Messanger
- CodeForces - D. Serega and Fun