GUÍA DE APRENDIZAJE PARA CONCURSANTES ICPC Y IOI: ALGORITMO DE MO (*MO'S ALGORITHM*)

1. Introducción

El algoritmo Mo es un enfoque eficiente para resolver problemas de consulta en rangos en arreglos o secuencias. El mismo esta basado en la descomposición de raiz cuadrada (*SQRT Descomposition*), para responder consultas de rango fuera de línea, es decir se leen todas las consultas de rango primero y luego se le da respuesta a cada una.

El nombre algoritmo Mo tiene dos versiones. La primera viene del nombre del matemático y científico de la computación Joseph Mo, quien introdujo este enfoque para resolver problemas de consulta en rangos en 1977. La segunda también proviene del nombre de Mo Tao, un programador competitivo chino, pero la técnica apareció antes en la literatura.

2. Conocimientos previos

2.1. Descomposición raiz cuadrada (*SQRT Descomposition*)

La descomposición raiz cuadrada es un método (o una estructura de datos) que le permite realizar algunas operaciones comunes (encontrar la suma de los elementos de la subarreglo, encontrar el elemento mínimo o máximo, etc.) en $O(\sqrt{n})$ operaciones, que es mucho más rápido que $O(n)$ para el algoritmo trivial.

3. Desarrollo

Esto puede parecer mucho peor que los métodos o estructuras de datos, ya que tiene una complejidad ligeramente peor que la que teníamos antes y no puede actualizar los valores entre dos consultas. Pero en muchas situaciones este método tiene ventajas. Durante una descomposición normal de raíz cuadrada, tenemos que calcular previamente las respuestas para cada bloque y fusionarlas al responder consultas. En algunos problemas, este paso de fusión puede resultar bastante problemático. Por ejemplo, cuando cada consulta solicita encontrar la moda de su rango (el número que aparece con más frecuencia). Para esto, cada bloque tendría que almacenar el recuento de cada número en algún tipo de estructura de datos, y ya no podemos realizar el paso de fusión lo suficientemente rápido. El algoritmo de Mo utiliza un enfoque completamente diferente, que puede responder este tipo de consultas rápidamente, porque solo realiza un seguimiento de una estructura de datos y las únicas operaciones con ella son fáciles y rápidas.

La idea es responder las consultas en un orden especial basado en los índices. Primero responderemos todas las consultas que tengan el índice izquierdo en el bloque 0, luego responderemos todas las consultas que tengan el índice izquierdo en el bloque 1 y así sucesivamente. Y también tendremos que responder las consultas de un bloque en un orden especial, es decir, ordenadas por el índice derecho de las consultas.

El algoritmo de Mo mantiene un rango activo del arreglo, y la respuesta a una consulta sobre el rango activo se conoce en cada momento. El algoritmo procesa las consultas una por una y siempre mueve los puntos finales del rango activo insertando y eliminando elementos.

Como ya se dijo, utilizaremos una única estructura de datos. Esta estructura de datos almacenará información sobre el rango. Al principio este rango estará vacío. Cuando queremos responder la siguiente consulta (en el orden especial), simplemente ampliamos o reducimos el rango, agregando/eliminando elementos en ambos lados del rango actual, hasta que lo transformamos en el rango de consulta. De esta manera, solo necesitamos agregar o eliminar un único elemento una vez a la vez, lo que debería ser una operación bastante sencilla en nuestra estructura de datos.

Dado que cambiamos el orden de respuesta de las consultas, esto solo es posible cuando se nos permite responder las consultas en modo fuera del orden en que fue dada, en otras palabras no existe actualización entre consultas.

El truco del algoritmo de Mo es el orden en que se procesan las consultas. El arreglo se divide en bloques de $k = O(\sqrt{n})$ elementos y se realiza una consulta $[a_1, b_1]$ procesado antes de una consulta $[a_2, b_2]$ si:

- $\lfloor a_1/k \rfloor < \lfloor a_2/k \rfloor$ o
- $\lfloor a_1/k \rfloor = \lfloor a_2/k \rfloor$ y $b_1 < b_2$

Por lo tanto, todas las consultas cuyos puntos finales izquierdos están en un determinado bloque se procesan una tras otra clasificadas según sus puntos finales derechos. Usando este orden, el algoritmo solo realiza operaciones O($n\sqrt{n}$), porque el punto final izquierdo se mueve O(n) veces en O($\sqrt{n}$) pasos, y el punto final derecho se mueve O($\sqrt{n}$) veces en O(n) pasos. Por lo tanto, ambos puntos finales se mueven un total de O($n\sqrt{n}$) pasos durante el algoritmo.

Como ejemplo, considere un problema en el que se nos proporciona un conjunto de consultas, cada una de las cuales corresponde a un rango en un arreglo, y nuestra tarea es calcular para cada consulta el número de elementos distintos en el rango.

En el algoritmo de Mo, las consultas siempre se ordenan de la misma manera, pero dependiendo del problema cambia como se mantiene la respuesta a la consulta. En este problema, podemos mantener un recuento de arreglo donde $count[x]$ indica el número de veces que aparece un elemento x en el rango activo.

Cuando pasamos de una consulta a otra, el rango activo cambia. Por ejemplo, si el rango actual es:

4	2	5	4	2	4	3	3	4

y el siguiente rango es:

4	2	5	4	2	4	3	3	4

Habrá tres pasos: el extremo izquierdo se mueve un paso hacia la derecha y el extremo derecho se mueve dos pasos hacia la derecha.

Después de cada paso, es necesario actualizar el recuento del arreglo. Después de agregar un elemento x, aumentamos el valor de $count[x]$ en 1, y si $count[x] = 1$ después de esto, también

aumentamos la respuesta a la consulta en 1. De manera similar, después de eliminar un elemento x, disminuimos el valor de $count[x]$ por 1, y si $count[x] = 0$ después de esto, también disminuimos la respuesta a la consulta en 1.

4. Implementación

4.1. C++

```
#include <iostream>
#include <bits/stdc++.h>

using namespace std;

void remove(idx);  // Remueve el valor en idx de la estructura de datos
void add(idx);     // Adiciona el valor en idx a la estructura de datos
int get_answer();  // Retorna la respuesta actual de la estructura de datos

int block_size;
//crear una estructura de datos acorde al problema

struct Query {
   int l, r, idx;
   bool operator<(Query other) const
   {
      return make_pair(l / block_size, r) <
      make_pair(other.l / block_size, other.r);
   }
};

vector<int> mo_s_algorithm(vector<Query> queries) {
   vector<int> answers(queries.size());
   sort(queries.begin(), queries.end());

   //Incializar la estructura de datos

   int cur_l = 0;
   int cur_r = -1;
   // La estructura de datos siempre reflejar la respuesta del rango [cur_l,
      cur_r]
   for (Query q : queries) {
      while (cur_l > q.l) { cur_l--; add(cur_l); }
          while (cur_r < q.r) { cur_r++; add(cur_r); }
      while (cur_l < q.l) { remove(cur_l); cur_l++; }
      while (cur_r > q.r) { remove(cur_r); cur_r--; }
      answers[q.idx] = get_answer();
  }
  return answers;
}
```

```
void remove(int idx){
   //dependera del problema
}

void add(int idx){
   //dependera del problema
}

int get_answer(){
   //dependera del problema
}
```

4.2. Java

```
public class MosAlgorithm {
   public static class Query {
      int index;
      int a;
      int b;

      public Query(int a, int b) {
         this.a = a;
         this.b = b;
      }
   }

   static int add(int[] a, int[] cnt, int i) {
      //La implementacion depende del problema
   }

   static int remove(int[] a, int[] cnt, int i) {
     //La implementacion depende del problema
   }

   public static int[] processQueries(int[] a, Query[] queries) {
      for (int i = 0; i < queries.length; i++) queries[i].index = i;
      int sqrtn = (int) Math.sqrt(a.length);
      Arrays.sort(queries, Comparator.<Query>comparingInt(q -> q.a / sqrtn).
          thenComparingInt(q -> q.b));
      // inicializar estructura
      int[] res = new int[queries.length];
      int L = 1;
      int R = 0;
      int cur = 0;
      for (Query query : queries) {
         while (L < query.a) cur += remove(a, cnt, L++);
         while (L > query.a) cur += add(a, cnt, --L);
```

```
        while (R < query.b) cur += add(a, cnt, ++R);
        while (R > query.b) cur += remove(a, cnt, R--);
          res[query.index] = cur;
      }
    return res;
  }
}
```

5. Aplicaciones

Algunas de las aplicaciones del algoritmo Mo incluyen:

1. **Problemas de consulta en rangos:** El algoritmo Mo es útil para resolver problemas que implican realizar consultas en rangos de una secuencia o arreglo. Por ejemplo, encontrar la frecuencia de un elemento en un rango dado, encontrar la suma de elementos en un rango, etc.
2. **Problemas de contadores y frecuencias:** Se puede utilizar el algoritmo Mo para resolver problemas que requieren mantener contadores o frecuencias de elementos en un rango específico.
3. **Problemas de consulta en ventanas deslizantes:** El algoritmo Mo es útil para resolver problemas que implican ventanas deslizantes, donde se necesita realizar consultas en subconjuntos continuos y superpuestos de una secuencia.
4. **Problemas de consulta en árboles:** El algoritmo Mo se puede aplicar a problemas que involucran consultas en árboles, como encontrar el LCA (antepasado común más bajo) de dos nodos en un árbol.
5. **Problemas de consulta en grafos:** Para ciertos tipos de consultas en grafos, el algoritmo Mo puede ser útil para optimizar el tiempo de respuesta.

En resumen, el algoritmo Mo es ampliamente utilizado en competencias de programación y en la resolución de problemas que involucran consultas en rangos, ya que proporciona una solución eficiente y elegante para este tipo de problemas.

6. Complejidad

Ordenar todas las consultas tomará O$(Q \log Q)$.

¿Qué pasa con las otras operaciones? ¿ Cuántas veces llamarán a *add* y *remove* ?

Digamos que el tamaño del bloque es S.

Si solo miramos todas las consultas que tienen el índice izquierdo en el mismo bloque, las consultas se ordenan por el índice derecho. Por eso llamaremos*add(cur_r)* y *remove(cur_r)* solo O(N) veces para todas estas consultas combinadas. Esto da $O(\frac{N}{S}N)$ llama a todos los bloques.

El valor de *cur_l* puede cambiar como máximo $O(S)$ durante entre dos consultas. Por lo tanto tenemos un adicional $O(SQ)$ llamadas de *add(cur_r)* y *remove(cur_r)*.

Para $S \approx \sqrt{N}$ esto da $O((N+Q)\sqrt{N})$ operaciones en total. Así, la complejidad es $O((N+Q)F\sqrt{N})$ dónde $O(F)$ es la complejidad de las funciones *add* y *remove*.

Consejos para mejorar el tiempo de ejecución

- Tamaño de bloque de precisamente $\sqrt{N}$ no siempre ofrece el mejor tiempo de ejecución. Por ejemplo, si $\sqrt{N} = 750$ entonces puede suceder que el tamaño de bloque de 700 o 800 puede funcionar mejor. Más importante aún, no calcule el tamaño del bloque en tiempo de ejecución: hágalo **const**. Los compiladores optimizan bien la división por constantes.
- En los bloques impares, ordene el índice derecho en orden ascendente y en los bloques pares, ordénelo en orden descendente. Esto minimizará el movimiento del puntero derecho, ya que la clasificación normal moverá el puntero derecho desde el final hasta el principio al comienzo de cada bloque. Con la versión mejorada este reinicio ya no es necesario.

```
bool cmp(pair<int, int> p, pair<int, int> q) {
   if (p.first / BLOCK_SIZE != q.first / BLOCK_SIZE) return p < q;
   return (p.first / BLOCK_SIZE & 1) ? (p.second < q.second) : (p.second
      > q.second);
}
```

7. Ejercicios

A continuación una lista de ejercicios que se pueden resolver con este algoritmo:

- SPOJ - DQUERY
- SPO - ZQUERY - Zero Query
- Codeforce - D. Powerful array
- Codeforce - E. XOR and Favorite Number
- CodeChef - Estimating progress
- CodeChef - Chefina and Beautiful Pairs