HỘI CÁC TRƯỜNG TRUNG HỌC PHỔ THÔNG CHUYÊN
DUYÊN HẢI BẮC BỘ

Chuyên đề môn TIN HỌC

SQRT DECOMPOSITION

Tháng 9-2021

SQRT DECOMPOSITION

Sqrt Decompistion (Phân rã căn bậc hai) là một cấu trúc dữ liệu (hoặc một cách thức tổ chức) cho phép chúng ta thực hiện một số phép toán thông thường (tính tổng các phần tử của một dãy con, tìm phần tử cực tiểu/cực đại.v...) với thời gian thực hiện $O(n\sqrt{n})$ nhanh hơn khá nhiều so với các thuật toán tầm thường.

So với các cấu trúc khác như cây phân đoạn (segment tree - ST), BIT (Binary Indexed Tree),... thì thời gian thực hiện của Sqrt decomposiion lâu hơn. Đổi lại, giải pháp này thường cho các cài đặt đơn giản hơn.

Chuyên đề này giới thiệu về phân ra căn bậc hai (sqrt decomposition). Đầu tiên tôi mô tả cấu trúc dữ liệu cho một trong những ứng dụng đơn giản nhất của ý tưởng này, sau đó tìm cách tổng quát hóa để giải quyết các vấn đề khác và cuối cùng xem xét một cách sử dụng hơi khác của ý tưởng này: chia các yêu cầu thực hiện thành các khối căn bậc hai (thuật toán Mo).

I. Cấu trúc dữ liệu phân rã căn bậc hai cơ bản

Bài toán: Cho mảng $a[0 \dots n-1]$. Hãy xây dựng một cấu trúc dữ liệu cho phép tính tổng các phần tử của $a[l \dots r]$ với l, r tùy ý trong thời gian $O(\sqrt{n})$

Ý tưởng cơ bản của phân ra căn bậc hai là việc tiền xử lý. Ta chia mảng a thành từng khối có độ dài xấp xỷ $\sqrt{n}$, với khối thứ i tính trước tổng các phần tử của nó $b[i]$.

Giả thiết kích thước của khối bằng $\sqrt{n}$ lấy số nguyên làm tròn lên:

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

Khi đó mảng a có thể được chia thành các khối như dưới đây:

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

Khối cuối cùng có thể có ít phần tử hơn các khối khác (nếu n không phải là bội số của s). Điều này không quan trọng vì nó có thể được xử lý dễ dàng. Do vậy với khối k chúng ta tỉnh được tổng các phần tử trong khối $b[k]$:

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

Ta có thể tính toàn bộ các giá trị $b[k]$ trong thời gian $O(n)$. Điều này giúp ích gì cho chúng ta khi trả lời yêu cầu $[k, l]$? Chú ý rằng nếu đoạn $[l, r]$ đủ dài nó sẽ chứa nguyên một số khối và với các khối này ta có thể lấy tổng các phần tử của chúng chỉ bằng một phép toán. Như vậy với đoạn $[l, r]$ chúng ta cần quan tâm tính tổng các phần tử là một phần của khối đầu tiên (phần cuối) và là một phần của khối cuối cùng (phần đầu)

Như vậy để tính tổng các phần tử của đoạn $[l, r]$ chúng ta chỉ cần tính tổng các phần tử của hai "phần lẻ" : $[l \dots (k+1) \cdot s - 1]$ và $[p \cdot s \dots r]$ và tổng các khối từ $k+1$ đến $p-1$:

$$\sum_{i=l}^{(k+1)\cdot s-1} a[i] + \sum_{i=k+1}^{p-1} b[i] + \sum_{i=p\cdot s}^{r} a[i]$$

Ở đây $k = l / s, p = r / s$

Chú ý: Khi $k = p$ tức l và r ở trong cùng một khối, công thức trên không đúng và ta chỉ cần tính tổng một cách tầm thường.

Cách tiếp cận này cho phép chúng ta giảm đáng kể số lượng phép tính. Thật vậy, kích thước của mỗi phần lẻ không vượt quá s. Vì chúng ta đã chọn $s \approx \sqrt{n}$ nên tổng số phép tính cần thực hiện khi tính tổng đoạn $[l, r]$ là $O(\sqrt{n})$

Ta bắt đầu với phương án code đơn giản dưới đây:

```
// input data
int n;
vector<int> a (n);

// preprocessing
int len = (int) sqrt (n + .0) + 1; // size of the block and the number of blocks
vector<int> b (len);
for (int i=0; i<n; ++i)
    b[i / len] += a[i];

// answering the queries
for (;;) {
    int l, r;
  // read input data for the next query
    int sum = 0;
    for (int i=l; i<=r; )
        if (i % len == 0 && i + len - 1 <= r) {
            // if the whole block starting at i belongs to [l, r]
            sum += b[i / len];
            i += len;
        }
        else {
            sum += a[i];
            ++i;
        }
}
```

Mã thực thi trên chứa nhiều phép toán chia không hợp lý (chậm hơn nhiều so với các phép toán số học khác). Thay vào đó chúng ta có thể tính toán các chỉ số c_l và c_r của các khối chứa l và r sau đó thực hiện tính lặp qua các khối $c_l + 1 \dots c_r - 1$ và xử lý riêng biệt các phần lẻ trong các khối c_l và c_r. Cách tiếp cận này tương ứng với công thức trong mô tả và đưa trường hợp $c_l = c_r$ thành trường hợp đặc biệt:

```
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, end=(c_l+1)*len-1; i<=end; ++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];
}
```

Cho đến nay ta đã bàn luận về việc tính tổng của các phần tử của một mảng con liên tục. Vấn đề có thể được mở rộng nếu như cho phép cập nhật các phần tử một cách riêng lẻ. Nếu một phần tử $a[i]$ thay đổi ta chỉ cần cập nhật giá trị $b[k]$ cho khối chứa nó ($k = i / s$) trong một thao tác:

$$b[k] += a_{new}[i] - a_{old}[i]$$

Việc tính tổng các phần tử của các mảng con cũng có thể được thay thế bằng việc tìm giá trị cực tiểu/cực đại của một mảng con. Nếu như phát sinh ra yêu cầu thay đổi giá trị một phần tử thì việc cập nhật giá trị $b[k]$ cũng thực hiện được, nhưng nó yêu cầu phải duyệt qua tất cả các giá trị của khối k với thời gian $O(s) = O(\sqrt{n})$

Phân ra căn bậc hai có thể áp dụng theo cách tương tự cho các bài toán khác: tìm số phần tử 0, tìm phần tử khác 0 đầu tiên, đếm các phần tử thỏa mãn một thuộc tính nào đó....

Một vấn đề khác khi chúng ta cần cập nhật các phần tử mảng trên các khoảng: tăng các phần tử hiện có hoặc thay thế chúng bằng một phần tử nhất định.

Ví dụ (*bài toán điển hình của việc sử dụng cấu trúc segment tree*), giả sử ta cần thực hiện thao tác sau trên mảng: thêm giá trị δ vào tất cả các phần tử mảng thuộc đoạn $[l, r]$ hoặc truy vấn giá trị phần tử $a[i]$. Ta lưu các giá trị tăng thêm cho toàn bộ khối k trong biến $d[k]$ (khởi đầu $d[k] = 0$). Với thao tác "cộng" ta thêm δ vào tất cả $d[k]$ của các khối nằm trọng vẹn trọng đoạn $[l, r]$ và thêm δ vào tất cả các $a[i]$ của các phần tử thuộc phần lẻ hai đầu. Giá trị phần tử tại vị trí i lúc này là $a[i] + b[i / s]$. Thời gian thực hiện "phép cộng" là $O(\sqrt{n})$ còn thời gian thực hiện tính một giá trị là $O(1)$.

Một điều thú vị là phân ra căn bậc hai có thể được sử dụng để thay thế các cấu trúc dữ liệu như các cây quản lý phạm vi. Tuy nhiên việc cài đặt chúng đơn giản hơn cài đặt các cây quản lý phạm vi nhiều. Cái giá phải trả là tốc độ thực thi chương trình chậm hơn chút ít nhưng nói chung là chấp nhận được.

Có những vấn đề khác cũng có thể được giải quyết bằng cách sử dụng phân rã căn bậc hai. Ví dụ như duy trì một tập hợp số với các phép thêm/xóa số và tìm phần tử lớn thứ k. Để giải quyết bài toán này ta lưu trữ các số theo giá trị tăng dần, chia thành

một số khối với $\sqrt{n}$ số cho mỗi khối. Mỗi khi một số được thêm/xóa thì các khối phải được cân bằng lại bằng cách di chuyển các số giữa đầu và cuối các khối liền kề.

Bài tập 1: *Có n cái lỗ nằm dọc theo một đường thẳng đánh số 1, 2, ..., n từ trái qua phải. Trong lỗi thứ i có đặt một lò xo có độ đàn hồi a_i. Nếu như có một viên bi rơi vào lỗ i thì nó sẽ nảy lên rơi vào lỗ $i + a_i$ nếu như $i + a_i \leq n$ còn nếu không sẽ ra khỏi hàng (nhảy vượt qua lỗ n), quá trình này lại tiếp tục như vậy... cho đến khi viên bi nhảy vượt qua lỗ n. Có m thao tác được người chơi bi thực hiện lần lượt thuộc một trong hai loại sau:*

- *Thay lò xo có độ đàn hồi b vào lỗ a*
- *Bắn một viên bi vào lỗ a và đếm xem nó nhảy qua bao nhiêu lỗ trước khi vượt qua lỗ n ?. In ra giá trị này.*

Yêu cầu: *Viết chương trình thực hiện m thao tác nói trên*

Input:

- *Dòng đầu tiên chứa hai số nguyên dương n, m $(1 \leq n, m \leq 10^5)$*
- *Dòng thứ hai chứa n số nguyên dương không vượt quá n là độ đàn hồi ban đầu của các lò xo đặt ở các lỗ 1, 2, ..., n*
- *m dòng cuối, mỗi dòng mô tả một thao tác thuộc một trong hai dạng:*
  - ***0 i b**: Đặt lại độ đàn hồi của lò xo ở lỗ i thành b ($1 \leq a, b \leq n$)*
  - ***1 i:** Tính số lỗ mà viên bi nhảy vào nếu bắn vào lỗ a $(1 \leq a \leq n)$*

Output: *Với các thao tác dạng **1 i** in ra trên một dòng hai số nguyên, số thứ nhất là số hiệu của lỗ cuối cùng trong hàng mà viên bi nhảy đến, số thứ hai là số lỗ mà viên bi nhảy vào trước khi vượt qua lỗ n*

Link down test:
https://drive.google.com/file/d/1w0FfLLRO9wJ_tiL41llfkwenPv1NgZWd/view?usp=sharing

Vẫn tư tưởng phân ra căn bậc hai, ta chia dãy các lỗ thành các khối có độ dài $S = \sqrt{n}$. Nếu n không phải là số chính phương ta lấy S là giá trị nguyên làm tròn lên. Với lỗ i chúng ra duy trì độ đàn hồi a_i và chỉ số lỗ đầu tiên thuộc về khối tiếp theo mà viên bi có thể nhảy đến được từ i (gọi giá trị này là $Next[i]$), đồng thời cũng tính luôn số lỗ nhảy qua khi đến $Next[i]$ (gọi giá trị này là $count[i]$). Ta luôn giả thiết có một lỗ "giả" sau lỗ n mà các viên bi đều có thể nhảy đến.

+) Để thực hiện thao tác **1 i** chúng ta nhảy qua các lỗ từ i đến $Next[i]$ cho đến khi vượt qua n, tổng quá trình đó cộng vào câu trả lời các giá trị $count[i]$. Số bước nhảy không vượt quá n / S nên thời gian thực hiện thao tác là $O\left(\frac{N}{S}\right) = O(\sqrt{n})$

+) Để thực hiên thao tác **0 i b**: Ta cần thay đổi cac giá trị $a[u], Next[u], count[u]$ cho tất cả các lỗ $u \leq i$ nằm trong cùng khối với i. Chỉ cần một vòng lặp ta có thể cập nhật

lại tất cả các giá trị này. Có tối đa S lần thực hiện cập nhật nên thời gian thực hiện truy vấn là $O(S) = O(\sqrt{n})$.

Tổng thời gian thực hiện thuật toán $O\left((n+m)\sqrt{n}\right)$

Tham khảo chương trình dưới đây:

```
#include<bits/stdc++.h>
#define N 100010
#define B 350
using namespace std;
int n, q, a[N], las[N], cnt[N];
void get (int i) {
    if (i + a[i] >= n || i / B != (i + a[i]) / B)
        las[i] = i, cnt[i] = 0;
    else
        las[i] = las[i + a[i]], cnt[i] = cnt[i + a[i]] + 1;
}
int main() {
    scanf ("%d%d", &n, &q);
    for (int i = 0; i < n; ++i)
        cin >> a[i];
    for (int i = n - 1; i >= 0; --i)
        get (i);
    for (int o, p, v; q--;) {
        scanf ("%d", &o);
        if (o) {
            scanf ("%d", &p);
            --p;
            int dwn = 0, s = 0;
            for (; p < n; dwn = las[p], p = dwn + a[dwn])
                s += cnt[p] + 1;
            printf ("%d %d\n", dwn + 1, s);
        } else {
            scanf ("%d%d", &p, &v);
            --p;
            a[p] = v;
            for (int i = p; i >= 0 && i / B == p / B; --i)
                get (i);
        }
    }
    return 0;
}
```

Bài toán 2: *Cho dãy số nguyên $a_1, a_2, \ldots, a_n$. Hãy thực hiện Q yêu cầu, mỗi yêu cầu được cho bởi hai số nguyên L, R $(1 \le L \le R \le n)$ với ý nghĩa tính:*

$$max\{|x-y| : \ L \le x, y \le R, a_x = a_y\}$$

Input:

- *Dòng 1: Chứa hai số nguyên dương* n, Q $(1 \le n, Q \le 10^5)$
- *Dòng 2: Chứa* n *số nguyên* $a_1, a_2, \dots, a_n$ $(|a_i| \le 10^9)$
- *Dòng 3..*$Q + 2$: *Dòng* $i + 2$ *mô tả yêu cầu thứ* i *gồm hai số nguyên* L_i, R_i $(1 \le L_i \le R_i \le n,\ \ i = 1 \div n)$

Output: *In ra Q dòng, dòng thứ* i *ghi một số nguyên là câu trả lời của yêu cầu thứ* i *(in 0 nếu tất cả các giá trị trong yêu cầu tương ứng khác nhau)*

Link down test:
https://drive.google.com/file/d/17svm1L1hW9d153PMOreH1dKyA54h_6nd/view?usp=sharing

Trong bài toán này ta cũng chia dãy thành các khối. Với mỗi khối k ta xây dựng câu trả lời cho mọi đoạn $[i_k, r]$ với i_k là chỉ số đầu tiên của khối. Tương tự, cũng xây dựng câu trả lời cho mọi đoạn $[l, j_k]$ với j_k là chỉ số cuối cùng của khối. Để làm điều này, với mỗi giá trị ta nhớ vị trí xuất hiện đầu tiên của nó. Với thông tin này việc cập nhật câu trả lời khi mở rộng khoảng chỉ trong $O(1)$ và thời gian chuẩn bị trước là $O(n\sqrt{n})$
Để trả lời truy vấn $[L, R]$ ta gọi c_L, c_R lần lượt là chỉ số khối chứa L và R. Các vị trí bằng nhau có 3 trường hợp xảy ra:

- Hai vị trí nằm từ đầu khối $c_L + 1$ đến R
- Hai vị trí nằm từ L đến cuối khôi $c_R - 1$
- Vị trí thứ nhất nằm trong khối c_L vị trí thứ hai nằm trong khối c_R

Hai trường hợp đầu đã chuẩn bị sẵn. Trường hợp thứ ba có thể tính toán trực tiếp.
Chương trình tham khảo:

```
#include <bits/stdc++.h>
#define maxn 100001
#define maxB 400

using namespace std;

int n, m, Q;
int a[maxn];

// Block
int Bsize, nBlock;
int block_ans[maxB][maxn];
int block_rev[maxn][maxB];
int T = 0;

struct HASHMAP {
    int val[maxn], index[maxn];

    void Init() {
```

```
        for (int i = 1; i <= n; ++i)
            val[i] = index[i] = 0;
    }

    int Get (int x, int time) {
        if (index[x] == time)
            return val[x];
        else
            return 0;
    }

    void Set (int x, int id, int time) {
        val[x] = id;
        index[x] = time;
    }

} nho;


void precompute() {
    // Nen mang A
    int pos[n + 1];
    for (int i = 1; i <= n; ++i)
        pos[i] = a[i];
    sort (pos + 1, pos + n + 1);
    for (int i = 1; i <= n; ++i)
        a[i] = lower_bound (pos + 1, pos + n + 1, a[i]) - pos;

    Bsize = int (sqrt (n));
    nBlock = (n - 1) / Bsize + 1;



    // Xay dung cau tra loi cho cac khoi
    nho.Init();
    T = 0;
    for (int i = 1; i <= nBlock; ++i) {
        int u = (i - 1) * Bsize + 1;
        int ans = 0;
        ++T;
        for (int j = u; j <= n; ++j) {
            if (nho.Get (a[j], T))
                ans = max (ans, j - nho.Get (a[j], T));
            else
                nho.Set (a[j], j, T);
```

```
            block_ans[i][j] = ans;
        }
    }

    for (int i = nBlock; i >= 1; --i) {
        int ans = 0;
        ++T;
        int u = (i == nBlock) ? n : i * Bsize;
        for (int j = u; j >= 1; --j) {
            if (nho.Get (a[j], T))
                ans = max (ans, nho.Get (a[j], T) - j);
            else
                nho.Set (a[j], j, T);
            block_rev[j][i] = ans;
        }
    }
}

int answer (int L, int R) {
    int ans = 0;
    ++T;
    for (int i = L; i <= R; ++i) {
        if (nho.Get (a[i], T))
            ans = max (ans, i - nho.Get (a[i], T));
        else
            nho.Set (a[i], i, T);
    }
    return ans;
}

#define task "MAXLEN"
int main() {
    if (fopen (task".inp", "r")) {
        freopen (task".inp", "r", stdin);
        freopen (task".out", "w", stdout);
    }

    scanf ("%d %d", &n, &Q);
    for (int i = 1; i <= n; ++i)
        scanf ("%d", &a[i]);

    precompute();
    for (int q = 1; q <= Q; ++q) {
        int L, R;
        scanf ("%d %d", &L, &R);
```

```
        int i = (L - 1) / Bsize + 1, j = (R - 1) / Bsize + 1;
        if (i == j)
            printf ("%d\n", answer (L, R));
        else {
            int ans = max (block_ans[i + 1][R], block_rev[L][j - 1]);
            ++T;
            for (int u = L; u <= i * Bsize; ++u) {
                if (nho.Get (a[u], T))
                    ans = max (ans, u - nho.Get (a[u], T));
                else
                    nho.Set (a[u], u, T);
            }
            for (int u = (j - 1) * Bsize + 1; u <= R; ++u) {
                if (nho.Get (a[u], T))
                    ans = max (ans, u - nho.Get (a[u], T));
                else
                    nho.Set (a[u], u, T);
            }
            printf ("%d\n", ans);
        }
    }
}
```

II Thuật toán Mo

Với ý tưởng tương tự, phân rã căn bậc hai có thể được sử dụng để trả lời Q truy vấn theo hình thức offline trong thời gian $O\left((N+Q)\sqrt{N}\right)$. Điều này có vẻ tệ hơn rất nhiều so với các phương pháp trong phần trên vì đây là một độ phức tạp kém hơn một chút so với chúng ta đã có trước đó và không thể cập nhật giá trị giữa hai truy vấn. Nhưng trong rất nhiều tình huống phương pháp này có lợi thế. Trong quá trình phân rã căn bậc hai thông thường, chúng ta phải tính toán trước câu trả lời cho mỗi khối và hợp nhất chúng khi trả lời truy vấn. Với một số vấn đề, bước hợp nhất này khá rắc rối. Ví dụ với truy vấn yêu cầu tìm giá trị xuất hiện thường xuyên nhất. Để làm điều này, mỗi khối cần phải lưu trữ số lượng từng số xuất hiện theo một số loại cấu trúc dữ liệu và chúng ta không thể thực hiện bước hợp nhất đủ nhanh. Thuật toán Mo sử dụng một cách tiếp cận hoàn toàn khác, có thể trả lời các truy vấn một cách nhanh chóng bởi vì nó chỉ theo dõi một cấu trúc dữ liệu và các toán tử thực hiện thay đổi dễ dàng và nhanh chóng.

Ý tưởng là trả lời các truy vấn theo thứ tự đặc biệt dựa trên các chỉ số. Trước tiên chúng ta trả lời tất cả các truy vấn có chỉ mục bên trái nằm trong khối 0, sau đó trả lời tất cả các truy vấn có chỉ số bên trái nằm trong khối 1,... Chúng ta cũng phải trả lời các

truy vấn của một khối theo một thứ tự đặc biệt, cụ thể là được sắp xếp không giảm theo chỉ số bên phải.

Như đã nói ở trên, chúng ta sử dụng một cấu trúc dữ liệu duy nhất. Cấu trúc dữ liệu này sẽ lưu trữ thông tin của một khoảng. Khởi đầu khoảng này là rỗng. Khi chúng ta trả lời một truy vấn chúng ta cần mở rộng/thu hẹp khoảng mà cấu trúc dữ liệu lưu trữ thông tin bằng cách thêm/xóa các phần tử ở cả hai phía của khoảng hiện tại cho đến khi khoảng này trùng với khoảng truy vấn. Bằng cách này chúng ta chỉ cần thêm hoặc bớt một phần tử duy nhất một lần tại một thời điểm. Đây sẽ là các theo tác khá dễ dàng nêu như ta chọn đươc cấu trúc dữ liệu hợp lý.

Trong thuật toán Mo chúng ta sử dụng hai hàm thêm một chỉ số và xóa đi một chỉ số khỏi khoảng mà chúng ta duy trì cấu trúc dữ liệu:

```
void remove(idx);  // Xóa giá trị tại chỉ số idx khỏi CTDL
void add(idx);     // Thêm giá trị tại chỉ số idx vào CTDL
int get_answer();  // Lấy câu trả lời truy vấn cho khoảng đang duy trì

int block_size;

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());

    // Khởi tạo CTDL

    int cur_l = 0;
    int cur_r = -1;
    // CTDL luôn quản lý đoạn [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;
}
```

Tùy theo từng vấn đề ta có thể sử dụng CTDL khác nhau và sử đổi các hàm add / remove / get_answer cho phù hợp. Ví dụ nếu chúng ta được yêu cầu tính tổng trong một khoảng thì chỉ cần sử dụng một số nguyên, khởi đầu 0, làm CTDL quản lý. Hàm add chỉ cần thêm giá trị của vị trí và sau đó cập nhật biến answer. Mặt khác hàm remove sẽ trừ giá trị tại vị trí và cập nhật biến answer. Lúc này get_answer chỉ trả về một số nguyên.

Ta xem xét thời gian thực hiện theo giải thuật Mo:

+) Việc sắp xếp các truy vấn mất $O(Q \log Q)$

+) Thời gian thực hiện các phép toán như thế nào? Ta sẽ tính bao nhiêu lần hàm **add** và **remove** thực hiện? Gọi kích cỡ của khối là S. Nếu chúng ra quan sát các truy vấn có đầu bên trái trong cùng một khối thì đầu bên phải của các truy vấn này được sắp xếp tăng dần. Do vậy số lần thực hiện **add(cur_r)** và **remove(cur_r)** là $O(N)$ cho toàn bộ truy vấn trong khối. Vậy nên số lần gọi hai hàm trên cho mọi truy vấn là $O\left(\frac{N}{S}N\right)$. Giá trị **cur_l** thay đổi nhiều nhất $O(S)$ giữa hai truy vấn liên tiếp. Do vậy số lần gọi **add(cur_l)** và **remove(cur_l)** là cho mọi truy vấn là $O(SQ)$

Vì $S \approx \sqrt{N}$ nên số lần gọi các hàm là $O\left((N+Q)\sqrt{N}\right)$ và thời gian thực hiện của thuật toán là :

$$O\left((N+Q)F\sqrt{N}\right)$$

Ở đây $O(F)$ là thời gian thực hiện **add** và **remove**

Bài toán 3: *Cho dãy n số nguyên dương $a_1, a_2, \ldots, a_n$. Một dãy con (i,j) là dãy $a_i, a_{i+1}, \ldots, a_j$ $(1 \le i \le j \le n)$. Với mỗi số nguyên dương s đặt K_s là số lần s xuất hiện trong dãy con (i,j). Ta định nghĩa giá trị của dãy cọn (i,j) là tổng:*

$$\sum_{s \in \mathbb{N}} K_s^2 \cdot s$$

Yêu cầu: *Cho biết trước dãy $a_1, a_2, \ldots, a_n$. Hãy trả lời m truy vấn, mỗi truy vấn có dạng hai số nguyên L, R $(1 \le L \le R \le n)$ với yêu cầu tính giá trị của dãy con $a_L, a_{L+1}, \ldots, a_R$*

Input:

- *Dòng 1: Chứa hai số nguyên dương* n, m $(1 \le n, m \le 2 \cdot 10^5)$
- *Dòng 2: Chứa* n *số nguyên* $a_1, a_2, \ldots, a_n$ $(1 \le a_i \le 10^9)$
- *Dòng* 3 *..2+*m*: Dòng* $2 + i$ *chứa hai số nguyên* L_i, R_i $(1 \le L_i, R_i \le n)$ *mô tả truy vấn thứ* i $(i = 1 \div m)$

Output: *In ra* m *dòng, mỗi dòng một số nguyên là kết quả của một truy vấn (theo thứ tự xuất hiện trọng input)*

Link down test:
https://drive.google.com/file/d/15xV0GYs9ViNle2c2tMJ634rgBCQ6fC6S/view?usp=sharing

Giả sử với số nguyên x hiện đang có số lượng là k. Nếu như ta thêm một giá trị x nữa thì số lượng sẽ là $k + 1$. Khi đó giá trị của dãy sẽ được tăng thêm một lượng là:

$$(k+1)^2 \cdot x - k^2 \cdot x = (2k+1) \cdot x$$

Nếu bỏ bớt đi một giá trị x thì giá trị của dãy sẽ giảm đi một lượng là:

$$k^2 \cdot x - (k-1)^2 \cdot x = (2k-1) \cdot x$$

Như vậy giá trị của dãy khi tăng hoặc bớt đi một phần tử có thể được tính lại trong $O(1)$. Để có thể quản lý hiệu quả trước tiên ta nén giá trị sao cho $a_i \in [1, n]$ (để có thể dùng mảng để nhớ số lượng). Giá trị cũ của a_i được lưu lại trong b_i.

Cấu trúc dữ liệu cơ bản để xử lý mỗi truy vấn trong trường hợp này chỉ là mảng **int nho[maxn]** với nho[x] là số lượng giá trị x đã xuất hiện trong mảng.

Các hàm **add, remove, get_answer** như vậy có thể thực hiện trong $O(1)$. Giải thuật Mo lúc này có thời gian thực hiện là $O\left((n+m)\sqrt{n}\right)$

Tham khảo chương trình dưới đây:

```
#include <bits/stdc++.h>
#define maxn 200001

using namespace std;

int n, m, a[maxn], ac[maxn];
int Bsize, nBlock;
int nho[maxn];
long long res;
long long ans[maxn];

struct QUERY {
    int Left, Right, Index;

    bool operator < (const QUERY &Other) const {
        return make_pair ( (Left - 1) / Bsize + 1, Right) < make_pair ( (Other.Left - 1) /
             Bsize + 1, Other.Right);
    }
} Q[maxn];

void add (int id) {
```

```
    res += 2LL * nho[ac[id]] * a[id] + a[id];
    ++nho[ac[id]];
}

void del (int id) {
    if (nho[ac[id]]) {
        res += -2LL * nho[ac[id]] * a[id] + a[id];
        --nho[ac[id]];
    }
}

#define task "PARRAY"
int main() {
    if (fopen (task".inp", "r")) {
        freopen (task".inp", "r", stdin);
        freopen (task".out", "w", stdout);
    }

    scanf ("%d %d", &n, &m);
    for (int i = 1; i <= n; ++i)
        scanf ("%d", &a[i]);

    // Nen mang a
    int tmp[n + 1];
    for (int i = 1; i <= n; ++i)
        tmp[i] = a[i];
    sort (tmp + 1, tmp + n + 1);
    for (int i = 1; i <= n; ++i)
        ac[i] = lower_bound (tmp + 1, tmp + n + 1, a[i]) - tmp;

    // Chia khoi
    Bsize = sqrt (n);
    nBlock = (n - 1) / Bsize + 1;

    // Doc cac truy van
    for (int i = 1; i <= m; ++i) {
        int u, v;
        scanf ("%d %d", &u, &v);
        Q[i].Left = u, Q[i].Right = v, Q[i].Index = i;
    }

    sort (Q + 1, Q + m + 1);

    // Xu ly truy van offline
    res = 0;
    int L = 1, R = 0;

    for (int i = 1; i <= m; ++i) {
        QUERY q = Q[i];
        while (L > q.Left)
            add (--L);
        while (R < q.Right)
            add (++R);
        while (L < q.Left)
```

```
            del (L++);
        while (R > q.Right)
            del (R--);
        ans[q.Index] = res;
    }
    for (int i = 1; i <= m; ++i)
        printf ("%I64d\n", ans[i]);
}
```

Bài toán 4: *Cho dãy n số nguyên $a_1, a_2, \ldots, a_n$ và m truy vấn. Mỗi truy vấn có dạng hai số nguyên u, v với ý nghĩa đếm xem trong dãy con $a_u, a_{u+1}, \ldots, a_v$ có bao nhiêu giá trị khác nhau mà các giá trị này xuất hiện đúng hai lần?*

Input:

- *Dòng đầu ghi hai số nguyên dương n, m $(1 \le n, m \le 3 \times 10^5)$*
- *Dòng thứ hai chứa n số nguyên $a_1, a_2, \ldots, a_n$ $(0 \le a_i \le 10^9 : i = 1 \div n)$*
- *m dòng cuối, mỗi dòng ghi hai số nguyên u, v mô tả một truy vấn.*

Output: *Với mỗi truy vấn in ra một dòng một số nguyên - kết quả tìm được.*

Link down test:
https://drive.google.com/file/d/1ow03Htw0EelI7Kktt_2Age3cG9S3Zaa_/view?usp=sharing

Tương tự như bài tập trên, cấu trúc dữ liệu duy trì trong trường hợp này vẫn là mảng **int nho[maxn]** trong đó nho[x] là số lần xuất hiện x trong dãy được duy trì (ở đây không mất tổng quát chúng ta có thể coi $a_i \in [1, n]$ bằng cách "nén dữ liệu"). Khi thêm giá trị x ta tăng nho[x] lên 1; Nếu nho[x]=2 thì câu trả lời tăng lên 1, nếu nho[x]=3 thì câu trả lời giảm đi 1. Khi bỏ đi một giá trị x ta giảm nho[x] đi 1; Nếu nho[x]=1 thì giảm câu trả lời đi 1, nếu nho[x]=2 thì tăng câu trả lời lên 1. Các hàm **add, remove, get_answer** trong trường hợp này có thời gian thực hiện là O(1) và tổng thời gian thực hiện của chương trình là $O\left((n+m)\sqrt{n}\right)$

Tham khảo chương trình dưới đây:

```
#include <bits/stdc++.h>
#define maxn 300001

using namespace std;

int n, m, block_size;
int a[maxn];

struct Query {
    int l, r, idx;
    bool operator< (Query other) const {
        return make_pair ( (l - 1) / block_size, r) <
               make_pair ( (other.l - 1) / block_size, other.r);
```

```
    }
} Q[maxn];

int nho[maxn];
int cnt = 0;
int ans[maxn];

void add (int i) {
    ++nho[a[i]];
    if (nho[a[i]] == 2)
        ++cnt;
    if (nho[a[i]] == 3)
        --cnt;
}

void remove (int i) {
    --nho[a[i]];
    if (nho[a[i]] == 2)
        ++cnt;
    if (nho[a[i]] == 1)
        --cnt;
}

#define task "DTWICE"
int main() {
    if (fopen (task".inp", "r")) {
        freopen (task".inp", "r", stdin);
        freopen (task".out", "w", stdout);
    }
    ios::sync_with_stdio (0);
    cin.tie (0);
    cout.tie (0);

    cin >> n >> m;
    for (int i = 1; i <= n; ++i)
        cin >> a[i];
    block_size = sqrt (n);
    if (block_size * block_size < n)
        ++block_size;
    for (int i = 1; i <= m; ++i) {
        int u, v;
        cin >> u >> v;
        Q[i].l = u, Q[i].r = v, Q[i].idx = i;
    }
    sort (Q + 1, Q + m + 1);

    // Nen mang a
    int tmp[n + 1];
    for (int i = 1; i <= n; ++i)
        tmp[i] = a[i];
    sort (tmp + 1, tmp + n + 1);
    for (int i = 1; i <= n; ++i)
        a[i] = lower_bound (tmp + 1, tmp + n + 1, a[i]) - tmp;
```

```
    // Tra loi cac truy van
    int cur_l = 1, cur_r = 0;
    cnt = 0;
    for (int i = 1; i <= m; ++i) {
        int L = Q[i].l, R = Q[i].r, id = Q[i].idx;
        while (cur_l > L)
            add (--cur_l);
        while (cur_r < R)
            add (++cur_r);
        while (cur_l < L)
            remove (cur_l++);
        while (cur_r > R)
            remove (cur_r--);
        ans[id] = cnt;
    }
    for (int i = 1; i <= m; ++i)
        cout << ans[i] << '\n';
}
```

Bài toán 5: *Cho đồ thị vô hướng có n đỉnh và m cạnh. Các đỉnh của đồ thị đánh số 1, 2, ..., n còn các cạnh của đồ thị đánh số $1,2,\dots m$.*

Yêu cầu: *Hãy thực hiện Q truy vấn trên đồ thị, truy vấn thứ i có dạng L_i, R_i ($1 \leq L_i, R_i \leq m$). Với mỗi truy vấn hãy tính số thành phần liên thông của đồ thị mới nhận được từ đồ thị ban đầu bằng cách bỏ đi tất cả các cạnh ngoại trừ các cạnh có chỉ số x thỏa mãn $L_i \leq x \leq R_i$*

Input:

- *Dòng 1: Chứa ba số nguyên dương n, m, Q ($1 \leq n, m, Q \leq 10^5$)*
- *Dòng 2...$m+1$: Dòng $i+1$ chứa hai số nguyên dương u_i, v_i là số hiệu hai đầu mút của cạnh thứ i ($1 \leq u_i, v_i \leq n$). Chú ý rằng đồ thị có thể có cạnh khuyên, có nhiều cạnh cùng nối hai đỉnh của đồ thị.*

Output: *In ra Q dòng, dòng thứ i in một số nguyên là kết quả trả lời của truy vấn thứ i ($i = 1 \div m$)*

Link down test:
https://drive.google.com/file/d/1gjjhfsFD8U4GLb0X-EcZWjQQ94Rle_Lf/view?usp=sharing

Áp dụng tư tưởng phân rã căn bậc hai ta thực hiện:

1. Chia các cạnh 1, 2, ..., m thành $\sqrt{m}$ khối, mỗi khối có $\sqrt{m}$ cạnh.
2. Chia các truy vấn vào từng nhóm theo đầu mút bên trái của truy vấn
3. Với tất cả các truy vấn trong cùng một nhóm (có đầu mút bên trái cùng khối) sắp xếp theo đầu mút phải tăng dần

Bây giờ ta có thể tiến hành trả lời các truy vấn theo từng nhóm với khối của đầu mút trái tăng dần. Với mỗi truy vấn:

1. Đầu tiên duy trì một cấu trúc **disjoint set union (DSU)** để đếm số lượng các thành phần liên thông của đồ thị với các cạnh từ *cạnh có chỉ số đầu tiên của khối tiếp theo (sau khối chứa đầu mút bên trái) đến cạnh có chỉ số bằng đầu mút bên phải*.
2. Với các cạnh nằm giữa đầu mút bên trái và cạnh cuối cùng của khối chứa đầu mút bên trái (có tối đa $\sqrt{m}$ cạnh) ta chạy một thuật toán đơn giản để nhập dần từng cạnh vào DSU trên cơ sở đó có câu trả lời.
3. Sau khi có cầu trả lời khôi phục lại trạng trái DSU về trạng thái trước phần 2. Vì rằng có nhiều nhất $\sqrt{m}$ phép thêm cạnh ở phần 2 nên cũng có nhiều nhất $\sqrt{m}$ phép bỏ đi một cạnh.

Điều đặc biệt trong bài toán này là cấu trúc DSU duy trì cần có khả năng quay về trạng thái trước đó. Chú ý rằng mỗi khi thêm cạnh chỉ có tối đa 2 vị trí trên DSU thay đổi giá trị do vậy khi thêm $\sqrt{m}$ cạnh cũng chỉ có tối đa $2\sqrt{m}$ vị trí thay đổi giá trị. Ta có thể lưu vào một stack để khôi phục lại trạng thái DSU trước đó.

Độ phức tạp thuật toán là $O\left((Q+m)\sqrt{m}\right)$

Tham khảo chương trình dưới đây:

```
#include <bits/stdc++.h>
#define maxn 200001
#define ft first
#define sc second

using namespace std;
typedef pair<int,int> pII;

int n, m, Q;
pII E[maxn];
int Bsize, nBlock;
int ans[maxn];

int ReadInt() {
    char ch=getchar();
    while (ch<'0' || ch>'9') ch=getchar();
    int x=ch-'0';
    ch=getchar();
    while (ch>='0' && ch<='9') {
        x=x*10+ch-'0';
        ch=getchar();
    }
    return x;
}

void WriteInt(int x) {
    if (x<10) putchar(x+'0'); else {
        WriteInt(x/10);
        putchar(x%10+'0');
    }
```

```
}


struct QUERY {
    int Left, Right, Index;

    QUERY(int x=0, int y=0, int z=0) : Left(x), Right(y), Index(z) {
    }

    bool operator < (const QUERY &other) const {
        return Right<other.Right;
    }
};

struct DSU_WITH_ROLLBACK {
    vector<int> rnk;
    int comps;
    struct dsu_save {
        int u, urank, v, vrank, comps;
    };
    stack<dsu_save> op;

    DSU_WITH_ROLLBACK() {
    };

    DSU_WITH_ROLLBACK(int n) {
        Init(n);
    };

    void Init(int n) {
        rnk.clear();
        while (!op.empty())
            op.pop();
        rnk.resize(n+1);
        for(int i=1; i<=n; ++i) {
            rnk[i]=1;
        }
        comps=n;
    }

    int find(int x) {
        if (rnk[x]>0)
            return x;
        int y=find(-rnk[x]);
        return y;
    }


    void join(int a,int b, int save=0) {
        int ra=find(a), rb=find(b);
        if (save)
            op.push({ra,rnk[ra],rb,rnk[rb],comps});
        if (ra==rb)
            return;
```

```
        comps--;
        if (rnk[ra]<rnk[rb])
            swap(ra, rb);
        rnk[ra] += rnk[rb];
        rnk[rb] = -ra;
    }

    void rollback() {
        if (op.empty())
            return;
        dsu_save x=op.top();
        op.pop();
        comps=x.comps;
        rnk[x.u]=x.urank;
        rnk[x.v]=x.vrank;
    }

    void panic() {
        while (!op.empty())
            rollback();
    }

};

vector<QUERY> mo[505];

void Solve() {
    //scanf("%d %d %d", &n, &m, &Q);
    n=ReadInt();
    m=ReadInt();
    Q=ReadInt();
    for(int i=1; i<=m; ++i) {
        //scanf("%d %d", &E[i].ft, &E[i].sc);
        E[i].ft=ReadInt();
        E[i].sc=ReadInt();
    }

    // Chia can
    Bsize=sqrt(m)+1;
    nBlock=(m-1)/Bsize+1;

    for(int i=1; i<=Q; ++i) {
        int u, v;
        // scanf("%d %d", &u, &v);
        u=ReadInt();
        v=ReadInt();
        mo[(u-1)/Bsize+1].push_back(QUERY(u,v,i));
    }

    // Xu ly tung khoi
    for(int i=1; i<=nBlock; ++i) {
        if (!mo[i].size())
            continue;
        sort(mo[i].begin(),mo[i].end());
```

```
        DSU_WITH_ROLLBACK ds(n);
        int w=i*Bsize+1, r=w;
        for(auto it : mo[i]) {
            int x=it.Left, y=it.Right, id=it.Index;
            while (r<=y) {
                ds.join(E[r].ft,E[r].sc);
                ++r;
            }
            for(int j=min(y,w-1); j>=x; --j)
                ds.join(E[j].ft, E[j].sc,1);
            ans[id]=ds.comps;
            ds.panic();
        }
        mo[i].clear();
    }

    for(int i=1; i<=Q; ++i) {
        //printf("%d\n",ans[i]);
        WriteInt(ans[i]);
        putchar('\n');
    }
}

#define task "CGRAPH"
int main() {
    if (fopen(task".inp","r")) {
        freopen(task".inp","r",stdin);
        freopen(task".out","w",stdout);
    }

    Solve();
}
```

III.Kết luận

Phân ra căn bậc hai (sqrt decomposition) cho chúng ta một cách tiếp cận đơn giản hơn để giải quyết một lớp lớn các bài toán vốn sử dụng các cấu trúc cây quản lý phạm vi cao cấp (BST, Segment Tree, Binary Indexed Tree,...) tuy nhiên cách cài đặt đơn giản hơn và cách tiếp cận giải quyết vấn đề cũng tự nhiên hơn. Khi thực hành có thể đưa ra các bài toán mà học sinh có thể giải theo nhiều cách trong đó có cách sử dụng phân rã căn bậc hai để hình thành cách nhìn đa chiều cho học sinh mỗi khi cần phải giải quyết một vấn đề mới.

Các bài toán minh họa trong chuyên đề này là các bài toán mà cách tiếp cận phân rã căn bậc hai cho giải pháp tự nhiên hơn các cách tiếp cận sử dụng các phương pháp khác.