Chia căn - Mới - Phần II

Chia căn II

Tác giả

- Lê Tuấn Hoàng - National University of Singapore

Reviewer

- Trần Xuân Bách - University of Chicago
- Nguyễn Minh Nhật - Georgia Tech

Phần II của bài viết xin được trình bày về các kĩ thuật chia căn nâng cao hơn.

Các kĩ thuật chia căn

1. Chia "nặng" và "nhẹ"

Kĩ thuật chia các "đối tượng", ở đây có thể là các truy vấn, đỉnh, xâu,..., thành hai nhóm "nặng" và "nhẹ" để có cách xử lí phù hợp.

Một nhận xét quan trọng thường được sử dụng trong kĩ thuật này: Nếu ta phân tích số nguyên dương n thành tổng của các số nguyên dương khác (không nhất thiết phải đôi một phân biệt), có không quá $\sqrt{n}$ số có giá trị lớn hơn hoặc bằng $\sqrt{n}$.

Bài toán 1: MarisaOJ - Truy vấn cây

Tóm tắt đề bài

Cho cây gồm n đỉnh, trên mỗi đỉnh là giá trị 0. Cho q truy vấn thuộc một trong hai dạng:

- `1 u d` : Tăng các giá trị trên các đỉnh kề với u thêm d.
- `2 u` : Tìm giá trị trên đỉnh u.

Giới hạn

- $1 \leq n, q \leq 10^5$.
- $1 \leq u \leq n$.
- $1 < d < 10^9$.

Thuật toán ngây thơ

Thuật toán đơn giản nhất chính là với mỗi truy vấn cập nhật, duyệt qua toàn bộ đỉnh kề của u và tăng giá trị trên các đỉnh này. Trong trường hợp tệ nhất, thuật toán có độ phức tạp $\mathcal{O}(n \times q)$.

Vậy trường hợp tốt hơn thì sao? Đó là khi số lượng đỉnh kề, hay còn gọi là bậc, của u không quá nhiều.

Chia "nặng" và "nhẹ"

Đầu tiên ta có nhận xét: Tổng bậc của n đỉnh là $2 \times (n - 1)$.

Từ đây phân loại các đỉnh vào hai nhóm:

- **Nặng** gồm các đỉnh có bậc lớn hơn $\sqrt{2 \times n}$. Sẽ có không quá $\sqrt{2 \times n}$ đỉnh nặng.
- **Nhẹ** gồm các đỉnh còn lại, bậc nhỏ hơn hoặc bằng $\sqrt{2 \times n}$.

Để xử lí các truy vấn cập nhật của đỉnh nhẹ, ta hoàn toàn có thể sử dụng thuật toán ngây thơ ở trên do số lượng đỉnh kề nhỏ. Thao tác xử lí những truy vấn đỉnh nhẹ có độ phức tạp $\mathcal{O}(\sqrt{n})$.

Để xử lí các truy vấn cập nhật của đỉnh nặng, ta sẽ sử dụng mảng lazy và tăng lazy_u lên d để đánh dấu đỉnh này đã được cập nhật thêm d. Thao tác xử lí cập nhật đỉnh nặng có độ phức tạp $\mathcal{O}(1)$.

Với các truy vấn trả lời giá trị trên đỉnh u, ngoài các giá trị đã được cập nhật trực tiếp qua các truy vấn đỉnh nhẹ, cần tính tổng lazy_v với v là các đỉnh kề của u. Dĩ nhiên khi tính tổng lazy, chỉ cần quan tâm đến các đỉnh v nặng, mà có không quá $\sqrt{2 \times n}$ đỉnh nặng nên các truy vấn trả lời có thể xử lí trong độ phức tạp $\mathcal{O}(\sqrt{n})$.

Bài toán 2: Codeforces 1207F - Remainder Problem

Tóm tắt đề bài

Cho mảng a gồm 500000 số nguyên được đánh số từ 1 đến 500000. Ban đầu tất cả các phần tử đều là 0.

Cho $q \leq 500000$ truy vấn thuộc một trong hai dạng:

- `1 x y` : Tăng a_x lên y.
- `2 x y` : Tính tổng các phần tử trong mảng mà có chỉ số chia x dư y.

Thuật toán ngây thơ

Với loại truy vấn đầu tiên, chỉ đơn giản tăng phần tử có chỉ số x lên y.

Trong truy vấn thứ hai, ta sẽ xét toàn bộ các chỉ số thỏa mãn (khoảng $\frac{500000}{x}$ chỉ số) để tính tổng. Có thể thấy khi x đủ lớn thì truy vấn sẽ chạy tương đối nhanh.

Chia "nặng" và "nhẹ"

Đặt $S = \sqrt{500000}$.

Từ đây phân loại các truy vấn tính tổng vào hai nhóm:

- **Nặng** gồm các truy vấn có x lớn hơn S. Các truy vấn này dễ dàng có thể được xử lí bằng thuật toán ngây thơ ở trên. Độ phức tạp thời gian là $\mathcal{O}(q \times S)$.
- **Nhẹ** gồm các truy vấn còn lại, có x nhỏ hơn hoặc bằng S.

Để xử lí các truy vấn nhẹ, ta sẽ lưu $\text{sum}(d, m)$ với $1 \le m \le S, 0 \le d < m$ là tổng của những chỉ số chia m dư d.

Khi thực hiện truy vấn cập nhật cho a_i, với từng $1 \le m' \le S$, ta cập nhật lại $\text{sum}(i \mod m', m')$. Còn khi truy vấn, nếu $x \le S$ thì in ra $\text{sum}(y, x)$.

Thuật toán có độ phức tạp $\mathcal{O}((n + q) \times S)$.

Nhận xét

Khi chia các đối tượng ra thành "nặng" và "nhẹ", ta sẽ cần hai cách xử lí khác nhau. Thông thường một trong số chúng sẽ là thuật toán ngây thơ. Từ đây ta có hướng suy nghĩ để tìm ra lời giải trong dạng bài tập này: Tìm ra thuật toán vô cùng ngây thơ, trong trường hợp nào thì nó hiệu quả, và trong trường hợp không hiệu quả thì ta cần có cách giải quyết khác là gì?

Bài tập

- VNOJ - DeMen100ns và thành phố
- MarisaOJ - Đỉnh gần nhất
- Codechef - KOL15C
- MarisaOJ - Màu thống trị
- MarisaOJ - Đếm xâu 3
- VNOJ - Demen và những truy vấn lẻ
- VNOJ - Subset sums

2. Chia căn truy vấn

Các thao tác cập nhật xuất hiện rất nhiều trong các bài toán xử lí truy vấn. Đôi khi, việc xử lí thao tác cập nhật khá khó, hoặc thậm chí không tồn tại cách cập nhật trực tiếp một cách hiệu quả. Đây chính là lúc kĩ thuật chia căn truy vấn tỏa sáng!

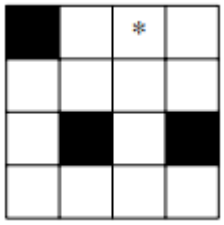
Kĩ thuật sẽ chia các truy vấn thành một số nhóm nhỏ, và sẽ chỉ thực hiện cập nhật cấu trúc (mảng, cây,...) sau khi xét xong một nhóm truy vấn.

Bài toán 1

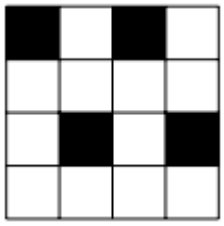
Đề bài

Cho một bảng hình chữ nhật gồm n ô. Ban đầu các ô trong bảng đều có màu trắng ngoại trừ một ô. Thực hiện $n - 1$ truy vấn, mỗi truy vấn bạn cần tính khoảng cách từ một ô trắng được chỉ định tới ô đen gần nhất, rồi tô đen ô trắng đó.

Ví dụ với bảng sau:



Khoảng cách từ ô trắng mang dấu * đến ô đen gần nhất là 2, vì có thể đi sang bên trái hai bước để đến một ô đen. Sau đó tô đen ô này:



Chia căn truy vấn

Ta có $n-1$ truy vấn, và sẽ chia chúng thành các nhóm $\sqrt{n}$ truy vấn liên tiếp: $[1, 2, \ldots, \sqrt{n}], [\sqrt{n}+1, \sqrt{n}+2 \ldots, 2 \times \sqrt{n}], \ldots$

Khi xử lí hết một nhóm truy vấn, ta mới thực hiện tô đen những ô thuộc nhóm truy vấn này. Từ đây ta có hai trường hợp.

- Với những ô đen đã được tô trên bảng: Thực hiện thuật toán BFS đa nguồn để tính với mỗi ô trắng khoảng cách gần nhất tới một ô đen. Thao tác này có độ phức tạp $\mathcal{O}(n)$.
- Với những ô đen chưa được tô nhưng nằm trước truy vấn đang xét: Chỉ cần xét toàn bộ những ô đen này và chọn ra ô có khoảng cách nhỏ nhất. Do có không quá $\sqrt{n}$ ô đen cần xét nên độ phức tạp của thao tác này là $\mathcal{O}(\sqrt{n})$.

Ta thực hiện thao tác thứ nhất $\sqrt{n}$ lần, và với mỗi truy vấn thực hiện thao tác thứ hai, lời giải này có độ phức tạp $\mathcal{O}(n \times \sqrt{n})$.

Bài toán 2: Timus - GCD 2010

Tóm tắt đề bài

Bạn được cho một tập hợp rỗng gồm các số nguyên dương và q truy vấn. Mỗi truy vấn sẽ thuộc một trong hai dạng, thêm một số vào tập hợp hoặc xóa một số khỏi tập hợp. Sau mỗi truy vấn hãy tính ước chung lớn nhất (GCD) của tất cả các số trong tập hợp. Lưu ý tập hợp có thể gồm nhiều số có giá trị giống nhau.

Giới hạn

- $1 \leq q \leq 10^5$.
- Các số trong tập hợp không vượt quá 10^{18} (đề bài gốc là 10^9, nhưng với thuật toán trình bày ở dưới có thể xử lí được tới giới hạn này).

Chia căn truy vấn

Nếu biết được GCD hiện tại của các số trong tập hợp, việc thêm một số vào và tính lại GCD rất đơn giản. Nhưng khá khó để xóa một số khỏi tập hợp và tính lại GCD trong độ phức tạp thời gian đủ tốt.

Cách giải quyết sẽ là chia truy vấn thành các nhóm $\sqrt{q}$ truy vấn.

Goi tập hợp đề bài cho là A. Xử lí lần lượt các nhóm truy vấn. Khi xử lí đến nhóm thứ p, ta tính xây dựng một tập hợp B gồm các giá trị:

- Tồn tại trong tập hợp A sau khi xử lí hết các nhóm truy vấn từ 1 đến $p-1$.
- Không tồn tại truy vấn xóa giá trị này trong nhóm p.

Xét lần lượt các truy vấn trong nhóm p: Nếu thực hiện hết các truy vấn từ đầu đến truy vấn này, có thể trong tập hợp B sẽ bị thiếu một vài giá trị so với tập A, nhưng sẽ không hề thừa giá trị nào, tránh được thao tác xóa khó thực hiện. Vì vậy chỉ cần tìm ra những giá trị bị thiếu rồi tính GCD của chúng với những giá trị trong tập B.

Số giá trị bị thiếu tối đa trong mỗi truy vấn là $\sqrt{q}$ và cứ $\sqrt{q}$ truy vấn ta lại xây tập hợp B gồm tối đa q phần tử, vậy độ phức tạp của bài toán là $\mathcal{O}(q \times \sqrt{q} \times \log 10^{18})$.

> **Nhận xét:**
>
> Bằng cách chia căn truy vấn, các bài toán đều được giải với các cách làm đơn giản và gần gũi hơn. Việc cập nhật lại cả cấu trúc tuy có độ phức tạp lớn nhưng không cần thực hiện nhiều, thu được một thuật toán khá hiệu quả.

Bài tập:

- VNOJ - Minimum Distance
- VNOJ - Dynamic connectivity
- Codeforces 455D - Serega and Fun
- Codeforces 1619H - Permutation and Queries
- VNOJ - Reversals and Sums
- VNOJ - Line queries
- Codeforces 487D - Conveyor Belts

3. Mở rộng thuật toán Mo

Mục này sẽ trình bày các ứng dụng hiếm gặp hơn của thuật toán Mo. Nếu mới làm quen với chia căn, bạn đọc có thể tạm thời bỏ qua phần này.

1. Thuật toán Mo có truy vấn cập nhật

Bài toán: VNOJ - Point Update Range Query

Cho mảng a gồm n phần tử nguyên. Cho q truy vấn thuộc một trong hai dạng:

- `1 i x` : Gán $a_i = x$.
- `2 l r` : Tính tổng các phần tử $a_l, a_{l+1}, \ldots, a_r$.

Giới hạn:

- $1 \leq n, q \leq 10^5$
- $1 < x < 10^9$
- $1 < l < r < n$

Thuật toán Mo

Giả sử bài toán cho chúng ta lần lượt các truy vấn như sau (ta chỉ cần quan tâm đến loại truy vấn là truy vấn gán hay tính tổng):

```
Truy vấn 1: 2 l r
Truy vấn 2: 1 i x
Truy vấn 3: 2 i x
Truy vấn 4: 1 l r
Truy vấn 5: 1 i x
Truy vấn 6: 2 l r
```

Tạm thời chỉ quan tâm đến truy vấn tính tổng: Các truy vấn 1, 3, và 6. Giả sử sau khi sắp xếp với thuật toán Mo, các truy vấn có thứ tự xử lí mới là: 1, 6, 3.

Truy vấn 1 có thể được xử lí dễ dàng. Khi đến với truy vấn tiếp theo là 6, có thể thấy mảng đã bị thay đổi qua các truy vấn cập nhật 2, 4, 5, nên cần tiến hành cập nhật các truy vấn này. Tiếp tục xử lí truy vấn 3, ta cần phải lần lượt đảo ngược truy vấn cập nhật 5 và 4.

Có thể thấy thuật toán Mo cũng có thể dùng để giải quyết các bài toán có truy vấn cập nhật. Nhưng nếu sắp xếp truy vấn theo thuật toán Mo bình thường thì tổng số lần cập nhật/đảo ngược giữa các truy vấn có thể rất lớn.

Thuật toán Mo có truy vấn cập nhật

Ta sẽ có cách sắp xếp mới để đảm bảo số lần cập nhật/đảo ngược không quá lớn. Tương tự thuật toán Mo, ta sẽ chia S chỉ số liên tiếp vào một nhóm. Ta sắp xếp các truy vấn với các tiêu chí lần lượt như sau:

- Chỉ số nhóm của đầu mút trái.
- Sau đó đến chỉ số nhóm của đầu mút phải.
- Và cuối cùng là số lượng truy vấn cập nhật nằm trước truy vấn này.

```
struct query{
    // t là số lượng truy vấn cập nhật đứng trước truy vấn này
    int l, r, t;
    int id;
}

bool cmp(const query &a, const query &b){
    if(a.l / S != b.l / S)
        return a.l < b.l;
    else if(a.r / S != b.r / S){
        if((a.l / S) % 2 == 1)
            return a.r < b.r;
        else
            return a.r > b.r
    }
    return a.t < b.t;
}
```

Phân tích độ phức tạp thuật toán:

- Số lần di chuyển đầu mút trái và phải là có cận trên là $2 \times (\frac{n^2}{S} + q \times S + \frac{n}{S})$.

- Với cách sắp xếp như trên, ta xét những truy vấn có đầu mút trái nằm trong cùng một nhóm, đồng thời đầu mút phải cũng nằm cùng một nhóm, chúng có thứ tự được sắp xếp tăng dần theo t. Vậy nên xử lí các truy vấn trong cùng một nhóm sẽ cần tối đa q thao tác cập nhật/đảo ngược, khi chuyển nhóm sẽ cần tối đa q thao tác cập nhật/đảo ngược khi thay đổi nhóm, vậy cần tối đa $2 \times q \times (\frac{n}{S})^2$ thao tác để xử lí các truy vấn cập nhật/đảo ngược. Có thay đổi luân phiên thứ tự sắp xếp t như sắp xếp đầu mút phải để giảm số thao tác về $q \times (\frac{n}{S})^2$.
- Nếu chọn $S = n^{\frac{2}{3}}$, ta thu được thuật toán có độ phức tạp $\mathcal{O}(q \times n^{\frac{2}{3}})$.

Cài đặt

Cách cài đặt tương đối giống với cài đặt thuật toán Mo bình thường, chỉ cần thêm thao tác xử lí các truy vấn cập nhật.

```
const int maxN = 2e5 + 5;

struct query{
    int l, r, t, id;
}

// Lưu lại thông tin cập nhật: Sau khi thực hiện truy vấn này, a[index] = old s
struct update{
    int index, old, next;
};

vector<update> update_queries;
vector<query> queries;

int n, q;
int a[maxN], b[maxN], ans[maxN];
int S = 0;
int l = 0, r = -1, t = -1;

bool cmp(const query &a, const query &b){
    if(a.l / S != b.l / S)
        return a.l < b.l;
    else if(a.r / S != b.r / S){
        if((a.l / S) % 2 == 1)
            return a.r < b.r;
        else
            return a.r > b.r
    }
    return a.t < b.t;
}
void run_update(int i, int x){
    // Nếu i nằm trong [l, r], việc cập nhật sẽ ảnh hưởng đến đáp án
    if(l <= i && i <= r){
        sum -= a[i];
        sum += x;
    }
    a[i] = x;
```

```
}

signed main(){
    ios_base::sync_with_stdio(false); cin.tie(0); cout.tie(0);

    cin >> n >> q;
    for(int i = 1; i <= n; i++){
        cin >> a[i];
        b[i] = a[i];
    }

    S = pow(n, 2. / 3.) + 1;

    for(int i = 1; i <= q; i++){
        int t, u, v;
        cin >> t >> u >> v;

        if(t == 1){
            update_queries.push_back({u, b[u], v});
            b[u] = v;
        }else{
            queries.push_back({u, v, (int)update_queries.size() - 1, i});
        }
    }

    sort(queries.begin(), queries.end(), cmp);

    for(query &q : queries){
        // Truy vấn cập nhật
        while(t < q.t) t++, run_update(update_queries[t].index, update_queries

        // Đảo ngược truy vấn cập nhật
        while(t > q.t) run_update(update_queries[t].index, update_queries[t].o

        while (l > q.l) sum += a[--l];
        while (r < q.r) sum += a[++r];
        while (l < q.l) sum -= a[l++];
        while (r > q.r) sum -= a[r--];

        ans[q.id] = sum;
    }
}
```

Bài tập

- MarisaOJ - Tắc kè
- SPOJ - XXXXXXXX

2. Thuật toán Mo trên cây

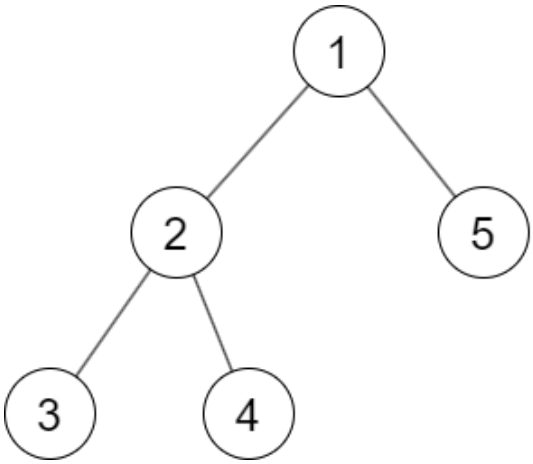
Bài toán: SPOJ COT2 - Count on a tree 2

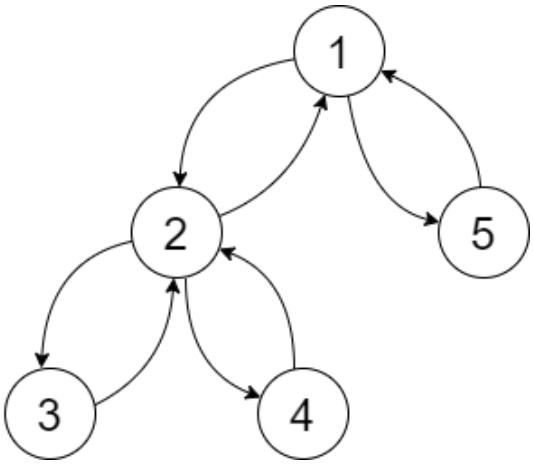
Cho một cây có n đỉnh. Trên mỗi đỉnh u gán giá trị a_u. Cho q truy vấn có dạng (u, v), hãy đếm số lượng giá trị khác nhau nằm trên đường đi từ đỉnh u đến đỉnh v.

Euler tour

Trước khi đến với thuật toán Mo trên cây, bạn đọc cần nắm rõ về Euler tour. Nếu bài toàn yêu cầu truy vấn trên cây con, ta có thể dễ dàng trải phẳng cây con đó thành một đoạn liên tiếp trên mảng với Euler tour, từ đó áp dụng trực tiếp thuật toán Mo. Xử lí truy vấn đường đi sẽ cần áp dụng khác một chút.

Để phục vụ cho phần trình bày thuật toán bên dưới, ta vẫn nhắc lại một số định nghĩa.





Với ST_u là thời điểm bắt đầu duyệt DFS cây con gốc u và EN_u là thời điểm hoàn thành duyệt cây con gốc u. Ta có thứ tự:

$$\mathrm{ST}_1 = 1;\ \mathrm{EN}_1 = 10$$
$$\mathrm{ST}_2 = 2;\ \mathrm{EN}_2 = 7$$
$$\mathrm{ST}_3 = 3;\ \mathrm{EN}_3 = 4$$
$$\mathrm{ST}_4 = 5;\ \mathrm{EN}_4 = 6$$
$$\mathrm{ST}_5 = 8;\ \mathrm{EN}_5 = 9$$
$$\mathrm{E} = \{1, 2, 3, 3, 4, 4, 2, 5, 5, 1\}$$

Thuật toán Mo trên cây

Ta có một tính chất quan trọng: Nếu v nằm trong cây con gốc u thì $\mathrm{ST}_u \leq \mathrm{ST}_v \leq \mathrm{EN}_v \leq \mathrm{EN}_u$.

Để xử lí truy vấn đường đi từ u đến v, gọi d là tổ tiên chung gần nhất (LCA) của u và v. Không mất tính tổng quát, giả sử $\mathrm{ST}_u \leq \mathrm{ST}_v$. Xét các đỉnh x nằm trên đường đi từ u đến v:

Trường hợp 1: $d = u$

Dễ thấy x phải nằm trong cây con gốc u, đồng thời v cũng phải nằm trong cây con gốc x. Từ nhận xét ở trên ta thu được:

$$\mathrm{ST}_u \leq \mathrm{ST}_x \leq \mathrm{EN}_x \leq \mathrm{EN}_u$$

$$\mathrm{ST}_x \leq \mathrm{ST}_v \leq \mathrm{EN}_v \leq \mathrm{EN}_x$$

$\Rightarrow \mathrm{ST}_u \leq \mathrm{ST}_x \leq \mathrm{ST}_v \leq \mathrm{EN}_v \leq \mathrm{EN}_x$

Từ đây có thể thấy rằng, nếu xét đoạn $[\mathrm{ST}_u, \mathrm{ST}_v]$ trên thứ tự E thì các đỉnh x nằm trên đường đi từ u đến v chỉ xuất hiện đúng một lần ở vị trí ST_x.

Trường hợp 2: $d \neq u$

- Trường hợp 2.1: Đỉnh x là tổ tiên của u và $x \neq d$.
  - $\mathrm{ST}_x \leq \mathrm{ST}_u \leq \mathrm{EN}_u \leq \mathrm{EN}_x$.
  - Do $\mathrm{ST}_u \leq \mathrm{ST}_v$, thời điểm duyệt xong cây con gốc x, ta chắc chắn chưa duyệt đến đỉnh v nên $\mathrm{EN}_x < \mathrm{ST}_v$.
  - Từ đây suy ra $\mathrm{ST}_x < \mathrm{EN}_u \leq \mathrm{EN}_x < \mathrm{ST}_v$.
- Trường hợp 2.2: Đỉnh x là tổ tiên của v và $x \neq d$.
  - $\mathrm{ST}_x \leq \mathrm{ST}_v \leq \mathrm{EN}_v \leq \mathrm{EN}_x$
  - Do $\mathrm{ST}_u \leq \mathrm{ST}_v$, thời điểm duyệt đến đỉnh x thì chắc chắn đã duyệt xong cây con gốc u nên $\mathrm{EN}_u < \mathrm{ST}_x$.
  - Từ đây suy ra $\mathrm{EN}_u < \mathrm{ST}_x \leq \mathrm{ST}_v < \mathrm{EN}_x$.
- Kết hợp hai trường hợp, ta sẽ có được đoạn cần xét là $[\mathrm{EN}_u, \mathrm{ST}_v]$, trên thứ tự E thì đỉnh x xuất hiện một lần ở vị trí EN_x trong trường hợp 2.1, và ở ST_x trong trường hợp 2.2. Với đỉnh d, dễ thấy $\mathrm{ST}_d < \mathrm{EN}_u < \mathrm{ST}_v < \mathrm{EN}_d$ nên đỉnh d không xuất hiện lần nào trong đoạn này, vì vậy cần xét riêng đỉnh d.

Bây giờ ta đã ánh xạ được các truy vấn (u, v) về một đoạn liên tiếp trên E. Những đỉnh trên đường đi từ u đến v sẽ xuất hiện **một** lần trong đoạn này. Khi cài đặt thuật toán Mo, lúc thêm/xóa các đỉnh, ta dễ dàng có thể kiểm soát số lượng của các đỉnh để biết đỉnh nào đang nằm trên đường đi.

Bài tập

- Codeforces 100962F
- VNOJ - Primitive queries : Bài toán áp dụng cả thuật toán Mo có truy vấn cập nhật.

3. Bỏ $\log$ trong thuật toán Mo

Các bài toán sử dụng thuật toán Mo thường được kết hợp với các cấu trúc dữ liệu. Việc lựa chọn cấu trúc dữ liệu phù hợp sẽ ảnh hưởng rất nhiều đến độ phức tạp cuối cùng của lời giải.

Bài toán

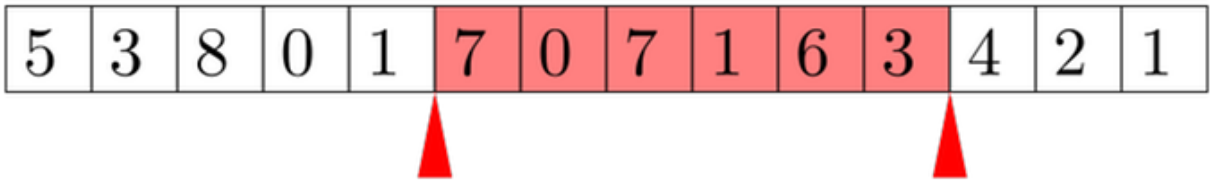
Cho mảng a gồm n phần tử. Cho q truy vấn dạng (l, r), hãy tìm MEX của $a_l, a_{l+1}, \ldots, a_r$. Có thể trả lời các truy vấn offline.

Giới hạn:

- $1 \leq n \leq 10^5$.
- $0 \leq a_i \leq n$.

Thuật toán Mo

Ta sẽ lưu một tập hợp (có thể cài đặt bằng `std::set`) những giá trị hiện đang **không tồn tại** trong đoạn đang xét. Đồng thời sẽ sử dụng thêm một mảng đếm để đếm số lượng của mỗi giá trị để thêm xóa tập hợp cho phù hợp.



$$\text{unvisited} = \{2, 4, 5, 8, 9, 10, ...\}$$

Nguồn: Codeforces

Do sử dụng `std::set` nên thuật toán có độ phức tạp $\mathcal{O}(n \times \sqrt{q} \times \log n)$, tương đối tệ. Phân tích kĩ hơn:

- Thêm một phần tử: Độ phức tạp $\mathcal{O}(\log n)$, thực hiện $\mathcal{O}(n \times \sqrt{q})$ lần.
- Xóa một phần tử: Độ phức tạp $\mathcal{O}(\log n)$, thực hiện $\mathcal{O}(n \times \sqrt{q})$ lần.
- Lấy ra giá trị nhỏ nhất trong `std::set` : Độ phức tạp $\mathcal{O}(1)$, thực hiện q lần.

Có thể thấy các thao tác thêm xóa cần thực hiện nhiều hơn nhưng lại có độ phức tạp lớn hơn, ngược lại thao tác tìm giá trị nhỏ nhất lại chạy rất nhanh trong khi thực hiện rất ít lần.

Cấu trúc dữ liệu chia căn

Ta cần một cấu trúc dữ liệu có độ phức tạp thêm/xóa phần tử nhanh, đổi lại việc truy vấn trên cấu trúc dữ liệu có thể lâu hơn. Cấu trúc dữ liệu chia căn rất phù hợp.

Quan tâm đến các giá trị $0, 1, 2, 3..., n$, chia chúng thành các nhóm mỗi nhóm $\sqrt{n}$ phần tử. Với mỗi nhóm ta sẽ lưu số lượng giá trị ở trong nhóm này xuất hiện ở đoạn đang xét.

Với thao tác thêm, xóa phần tử, ta chỉ cần cập nhật lại giá trị trong mảng đếm và trong nhóm. Hai thao tác có thể được xử lí trong độ phức tạp $\mathcal{O}(1)$.

Với thao tác tìm MEX, duyệt lần lượt qua $\sqrt{n}$ nhóm và tìm ra nhóm đầu tiên không có đầy đủ các giá trị. Sau đó lại duyệt lần lượt các giá trị trong nhóm này để tìm giá trị không xuất hiện. Thao tác này có độ phức tạp $\sqrt{n}$.

Vậy thuật toán mới có độ phức tạp $\mathcal{O}(n \times \sqrt{q} + q \times \sqrt{n})$.

Bài tập

- MarisaOJ - Truy vấn đoạn
- Codeforces 940F - Machine Learning : Bài toán áp dụng cả thuật toán Mo có truy vấn cập nhật.

Một số cách tối ưu thời gian chạy

Chỉ số nhóm

Rất nhiều bài toán chia căn sẽ sử dụng một hằng số S, kích cỡ của một nhóm, block,... và thực hiện rất nhiều phép chia cho S. Toán tử chia chạy khá chậm, nên khai báo S là biến hằng số bằng từ khóa `const` trong C++. Phép chia cho biến hằng số được compiler tối ưu rất tốt.

Trường hợp không thể khai biến báo hằng số, có thể tính trước một mảng `B[i] = i / S`.

Chọn hằng số phù hợp

Việc lựa chọn hằng số phù hợp sẽ ảnh hưởng rất lớn đến độ phức tạp thời gian, không gian. Các bài toán trình bày ở trên hầu hết sẽ được chọn hằng số là $\sqrt{n}$ hoặc $\sqrt{q}$ để đỡ rườm rà, nhưng chúng có thể không phải hằng số tối ưu.

Ví dụ ở bài toán Yếu vị được trình bày ở trên, nếu chọn mỗi nhóm có S phần tử, thì thao tác khởi tạo có độ phức tạp $\mathcal{O}(\frac{n^2}{S})$ và thao tác truy vấn có độ phức tạp $\mathcal{O}(q \times S \times \log n)$. Chọn $S = \frac{n}{\sqrt{q \times \log n}}$ sẽ cho độ phức tạp tốt hơn là $\mathcal{O}(n \times \sqrt{q \times \log n})$.

Tuy nhiên không phải hằng số trên lí thuyết này sẽ cho ra thời gian chạy tốt nhất. Lấy ví dụ bài toán Codeforces 86D - Powerful array được trình bày ở phần thuật toán Mo. Trên lí thuyết việc lựa chọn $S = \sqrt{\frac{n^2}{q}}$ cho ra độ phức tạp thời gian tốt nhất. Để kiểm chứng, người viết đã sử dụng cùng một code và thử một số hằng số S khác nhau, nộp bài sử dụng C++17:

S	Thời gian
$\sqrt{\frac{n^2}{q}}$	2806 ms
$\sqrt{2 \times \frac{n^2}{q}}$	1954 ms
$\sqrt{3 \times \frac{n^2}{q}}$	2058 ms
$2 \times \sqrt{\frac{n^2}{q}}$	2246 ms

Có thể thấy việc chọn $S = \sqrt{2 \times \frac{n^2}{q}}$ khiến code chạy nhanh hơn đáng kể. Thời gian chạy của code phụ thuộc vào nhiều yếu tố nên khá khó tính được chính xác hằng số S phù hợp.

Thay vào đó, ta có thể sinh một số test và chạy thử nghiệm với một vài hằng số S khác nhau. Sau đó chọn ra giá trị S cho thời gian chạy tốt nhất.

Danh sách bài tập

- VNOJ - Educational SQRT Contest 1
- VNOJ - Educational SQRT Contest 2
- VNOI : Các bài toán chia căn trên Codeforces được phân loại theo cách làm.

- MarisaOJ - Chia căn
- USACO - Square Root Decomposition

Đọc thêm

- Codeforces - An alternative sorting order for Mo's algorithm : Một cách sắp xếp các truy vấn trong thuật toán Mo để tối ưu hóa thời gian.
- https://codeforces.com/blog/entry/68271 : Một cách cài đặt thuật toán Mo trên cây khác.
- CP Algorithm - Sqrt tree .