Chia căn - Mới - Phần I

Chia Căn Phần I

Tác giả

- Lê Tuấn Hoàng - National University of Singapore

Reviewer

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

Lời nói đầu

Chia căn, một kĩ thuật thú vị, thiên biến vạn hóa qua từng bài tập.

Những kĩ thuật, thuật toán vận dụng các tính chất của phép căn thường được gọi chung là chia căn. Bài viết được biên soạn nhằm mang đến cho bạn đọc cái nhìn chi tiết nhất về chia căn và những vấn đề liên quan.

Các kĩ thuật được trình bày sẽ đi kèm một vài bài toán ví dụ, được đưa vào giúp bạn đọc dễ hình dung về ứng dụng của kĩ thuật hơn. Chia căn có thể không phải lời giải tối ưu nhất cho bài toán ví dụ. Cài đặt mẫu được cài đặt bằng ngôn ngữ C++ và được cài đặt tường minh nhất nên có thể chưa được tối ưu.

Kiến thức cần biết

Xử lí online và offline

Bài viết sẽ nhắc tới hai khái niệm này tương đối nhiều, nên xin phép bạn đọc được giải thích hai khái niệm này.

Hai khái niệm thường được sử dụng trong các bài toán xử lí truy vấn. Cụ thể:

- **Xử lí offline**: Ta có thể đọc vào tất cả các truy vấn. Sau đó có thể xử lí tuần tự từng truy vấn một hoặc có thể xử lí các truy vấn theo một thứ tự hợp lí khác để giải quyết bài toán hiệu quả.
- **Xử lí online**:
  - Ta không thể đọc được vào toàn bộ các truy vấn. Thông tin về các truy vấn sẽ được mã hóa, cần xử lí được truy vấn trước đó để giải mã.
  - Ví dụ: Khi xử lí truy vấn cần số nguyên x, đề bài sẽ cho số nguyên y. Sau đó x được tính bằng công thức $x = y + \text{ans}$ với ans là kết quả của truy vấn trước đó. Các truy vấn hoàn toàn bị phụ thuộc vào các truy vấn trước, không có cách nào ngoài việc xử lí tuần tự các truy vấn.

Quy hoạch động

Một số bài toán dưới đây sử dụng quy hoạch động. Bạn đọc nên nắm được quy hoạch động cơ bản.

Các dạng chia căn thường thấy

Phần I của bài viết xin được trình bày về các kĩ thuật chia căn thường gặp hơn.

1. Chia căn dựa vào phân tích tổng số nguyên dương

Có nhận xét: Nếu số nguyên dương n được tách thành tổng của các số nguyên dương, thì tồn tại không quá $\mathcal{O}(\sqrt{n})$ số nguyên dương khác nhau.

Chứng minh

Để cực đại số lượng số khác nhau, ta sẽ chọn những số nhỏ nhất có thể: $1, 2, 3, \ldots$ Nếu chọn các số nguyên từ 1 đến k, tổng của chúng là $\frac{k\times(k+1)}{2}$. Vậy nên số lượng giá trị khác nhau không vượt quá $k \leq \sqrt{2 \times n}$.

Dưới đây là một số bài toán sử dụng tính chất này.

Bài toán 1: Codeforces 221D - Little Elephant and Array

Tóm tắt đề bài

Cho mảng a gồm $n \leq 10^5$ phần tử, giá trị các phần tử $\leq n$. Có $m \leq 10^5$ truy vấn (l, r), hãy đếm số lượng giá trị x xuất hiện đúng x lần nằm trong đoạn $a_l, a_{l+1}, \ldots, a_r$.

Lời giải

Để một giá trị x xuất hiện đúng x lần trong một truy vấn nào đó, thì số lần xuất hiện của giá trị này trong mảng a chắc chắn không nhỏ hơn x. Ta chỉ quan tâm những giá trị này.

Số lượng giá trị ta cần quan tâm nhiều nhất khi: Có đúng giá trị 1 xuất hiện 1 lần, giá trị 2 xuất hiện 2 lần,.... Dễ thấy, số lượng giá trị cần quan tâm nhiều nhất là khoảng $\sqrt{2 \times n}$.

Từ đây, trong mỗi truy vấn, với từng giá trị x cần xét, ta chỉ cần lần lượt kiểm tra số lần xuất hiện của nó trong đoạn $a_l, a_{l+1}, \ldots, a_r$ có chính xác là x hay không.

Cài đặt

```
int n, a[N];

// Mảng đếm giá trị
int count[N];

// Lưu những giá trị cần phải xét
vector<int> value;
vector<vector<int>> prefix_sums;


// Phần khởi tạo
void init(){
    for(int i = 1; i <= n; i++){
        count[a[i]]++;
    }

    for(int x = 1; x <= n; x++){
```

```
        // Điều kiện để giá trị x được xét là có số lần xuất hiện ít nhất x lầ
        if(count[x] >= x){
            value.push_back(x);

            // Mảng cộng dồn phục vụ việc đếm số lượng giá trị x nằm trong một
            vector<int> prefix_sum(n + 1);
            for(int i = 1; i <= n; i++){
                prefix_sum[i] = prefix_sum[i - 1] + (a[i] == x);
            }
            prefix_sums.push_back(prefix_sum);
        }
    }
}

// Phần truy vấn
int query(int l, int r){
    int answer = 0;

    for(int i = 0; i < value.size(); i++){
        if(prefix_sums[i][r] - prefix_sums[i][l - 1] == value[i]){
            answer++;
        }
    }

    return answer;
}
```

Độ phức tạp của phần khởi tạo là $\mathcal{O}(n \times \sqrt{n})$, độ phức tạp của một truy vấn là $\mathcal{O}(\sqrt{n})$. Độ phức tạp của thuật toán là $\mathcal{O}((n+q) \times \sqrt{n})$.

Bài toán 2: MarisaOJ - Ghép xâu

Đề bài

Cho tập S gồm n xâu khác nhau $S = \{S_1, S_2, \ldots, S_n\}$ có tổng độ dài là m và một xâu mục tiêu T. Hỏi có bao nhiêu cách để có thể ghép được xâu T từ n xâu đã cho. Một xâu có thể được sử dụng nhiều lần.

Ví dụ với xâu T = `ABAB` và $S = \{$ `A`, `B`, `AB` $\}$ thì có 4 cách ghép:

- `A + B + A + B`
- `A + B + AB`
- `AB + A + B`
- `AB + AB`

Giới hạn

- $1 < n < 10^5$.
- $|T| \leq 10^5$.

- $m \leq 5 \times 10^5$.

Quy hoạch động

Có thể sử dụng quy hoạch động để giải bài toán này: Hàm mục tiêu là $\text{count}(i)$, là số lượng cách để ghép được tiền tố $T[1...i]$.

Để giải bài toán hiệu quả, ta cần sử dụng thuật toán hash nhằm so sánh tính bằng nhau của các xâu trong độ phức tạp thời gian $\mathcal{O}(1)$.

Để tính được $\text{count}(i)$, ta xét toàn bộ các xâu con của T kết thúc tại i, kiểm tra xem nó có thuộc tập S đã cho hay không và cập nhật tương ứng.

```
// Tập H lưu mã hash của các xâu trong tập S
unoredered_set<int64_t> H;

...

// Sử dụng index từ 1 sẽ thuận tiện hơn trong bài toán này
T = "#" + T;

for(int i = 1; i < T.size(); i++){
    for(int j = i; j >= 1; j--){
        // Tính hash của xâu con T[j...i]
        int64_t hash_value = get_hash(j, i);

        // Nếu trong tập S có tồn tại xâu bằng với xâu con này
        if(H.count(hash_value)){
            count[i] += count[j - 1];
        }
    }
}
```

Thuật toán này có độ phức tạp $\mathcal{O}(|T|^2)$.

Chia căn

Ta có nhận xét là có tối đa $\mathcal{O}(\sqrt{m})$ độ dài xâu khác nhau trong tập S. Để tính $\text{count}(i)$, thay vì xét toàn bộ các xâu con kết thúc tại i, ta chỉ cần xét các xâu con có độ dài p kết thúc ở i, sao cho tồn tại ít nhất một xâu có độ dài p trong tập S. Phần cài đặt khá tương tự:

```
// Tập H[i] lưu mã hash của các xâu độ dài i trong tập S
unordered_set<int64_t> H[N];

// Danh sách các độ dài xâu khác nhau của các xâu trong tập S
vector<int> lengths;

...

T = "#" + T;
```

```
10
11  for(int i = 1; i < T.size(); i++){
12      for(int &p : lengths){
13          int64_t hash_value = get_hash(i - p + 1, i);
14
15          if(H[p].count(hash_value)){
16              count[i] += count[i - p];
17          }
18      }
19  }
```

Do không có quá $\mathcal{O}(\sqrt{m})$ độ dài khác nhau nên độ phức tạp chỉ là $\mathcal{O}(|T| \times \sqrt{m})$.

Mở rộng: Thuật toán tất định

Ngoài cách sử dụng hash, ta cũng có thể sử dụng thuật toán Aho-Corasick.

Tương tự, có $\mathcal{O}(\sqrt{m})$ độ dài xâu khác nhau. Với mỗi độ dài p, dựng máy trạng thái hữu hạn (finite state machine - FSM) gồm các xâu có độ dài p trong tập S.

Để tính được $\text{count}(i)$, sử dụng kí tự T_i để di chuyển đến trạng thái tiếp theo trên toàn bộ các FSM đã xây dựng. Nếu trạng thái của FSM gồm các xâu độ dài p là kết thúc của một xâu, có thể khẳng định xâu con $T[i-p+1..i]$ tồn tại trong S và tiến hành cập nhật tương ứng.

Bài toán 3: MarisaOJ - Cái túi

Đề bài

Cho n vật có khối lượng lần lượt là $w_1, w_2, \ldots, w_n$ và $w_1 + w_2 + \ldots + w_n = m$. Hãy kiểm tra xem có thể chọn một số vật sao cho tổng khối lượng của chúng là T không?

Giới hạn

- $1 \leq n, m, T \leq 10^6$.

Quy hoạch động

Dễ thấy đây là bài quy hoạch động cái túi điển hình có thể giải được trong độ phức tạp thời gian là $\mathcal{O}(n \times T)$.

Để làm tốt hơn, ta có nhận xét đầu tiên là có không quá $\mathcal{O}(\sqrt{m})$ khối lượng các nhau, phân các vật có cùng khối lượng vào cùng một nhóm.

Vẫn sử dụng quy hoạch động cái túi: Hàm mục tiêu của ta sẽ là $\text{possible}(i, p)$ là true hoặc false tương ứng với sau khi xét xong i nhóm khối lượng đầu tiên, có thể tạo ra được khối lượng p không? Ta có thể cài đặt như sau:

```
1  // d là số lượng khối lượng khác nhau
2  // W[i] là khối lương của nhóm thứ i
3  // c[i] là số lượng của nhóm thứ i
4
5  for(int i = 1; i <= d; i++){
6      for(int p = 0; p <= T; p++){
7          // k là số lượng vật lấy ở trong nhóm thứ i
8
```

```
            for(int k = 0; k * W[i] <= p, k <= c[i]; k++){
                if(possible[i - 1][p - k * W[i]])
                    possible[i][p] = true;
            }
        }
    }
```

Nếu cài đặt như này, độ phức tạp vẫn là $\mathcal{O}(m \times T)$. Cách chuyển nhãn là: $\text{possible}(i, p) = \text{true}$ khi tồn tại $0 \leq k \leq c_i$ sao cho $\text{possible}(i - 1, p - k \times W_i) = \text{true}$. Ta có một nhận xét quan trọng $p \equiv p - k \times W_i \mod W_i$, điểm đặc biệt này dẫn tới cách làm như sau:

```
// last[u] lưu lại vị trí gần nhất mà possible[i - 1][p'] = true sao cho p' mo

for(int i = 1; i <= d; i++){
    memset(last, -1, sizeof last);
    for(int p = 0; p <= T; p++){
        // Cần phải sử dụng <= c[i] vật
        if(last[p % W[i]] != -1 && (p - last[p % W[i]]) / W[i] <= c[i]){
            possible[i][p] = true;
        }
        if(possible[i - 1][p]){
            last[p % W[i]] = p;
        }
    }
}
```

Ta thu được thuật toán có độ phức tạp $\mathcal{O}(T \times \sqrt{m})$.

2. Thuật toán Mo

Bài toán 1: VNOJ - Hamilton Path

Bài toán này tuy không sử dụng thuật toán Mo, nhưng sẽ là tiền đề khi tìm hiểu về thuật toán Mo.

Đề bài

Cho $n \leq 10^6$ điểm (x_i, y_i) trên hệ trục tọa độ $(1 \leq x_i, y_i \leq 10^6)$. Khoảng cách giữa hai điểm a, b là $\text{dist}(a, b) = |x_a - x_b| + |y_a - y_b|$ (khoảng cách Manhattan).

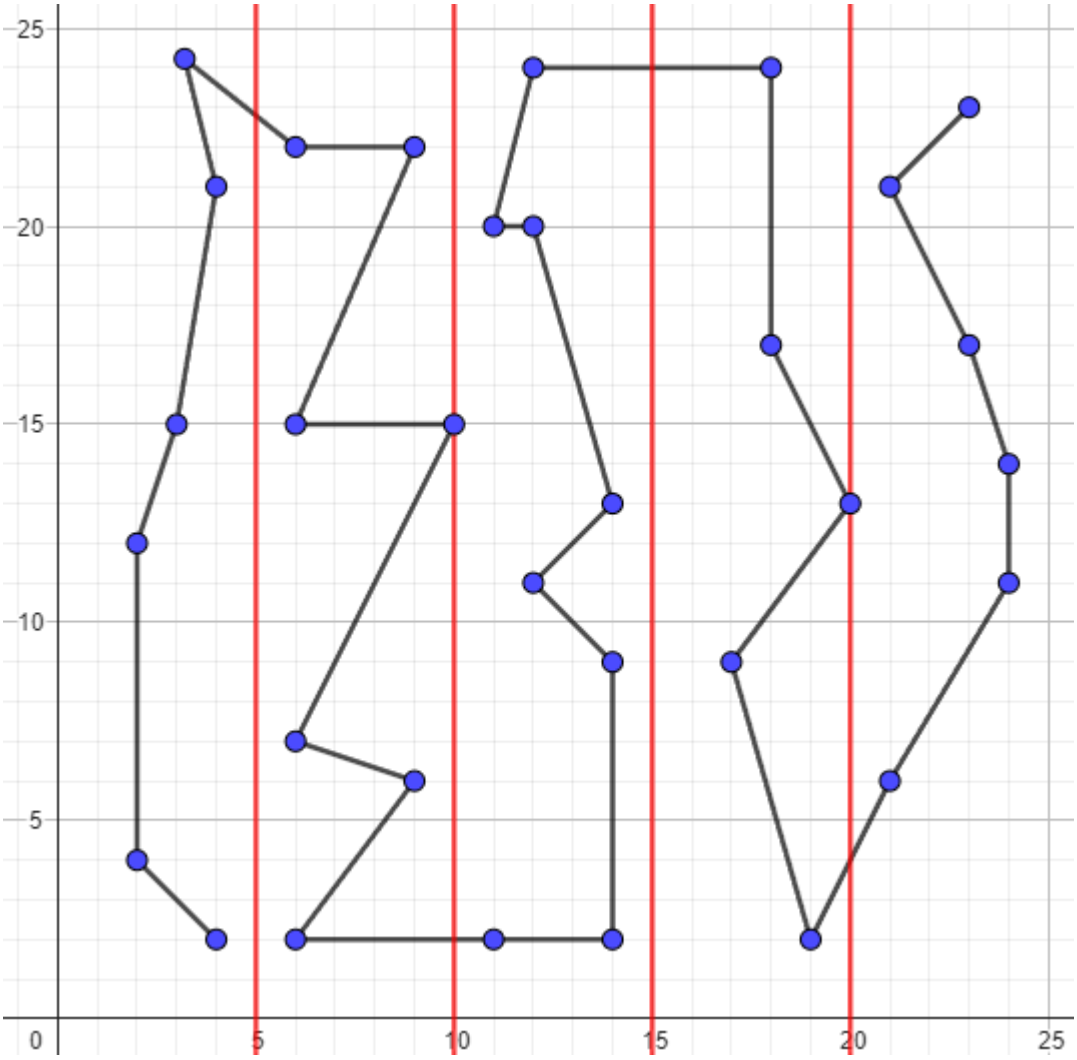
Đường đi Hamilton là đường đi đi qua toàn bộ n điểm và mỗi điểm chính xác một lần. Với p là một hoán vị của các số nguyên từ 1 tới n, độ dài đường đi Hamilton được tính bằng công thức $\sum_{i=1}^{n-1} \text{dist}(p_i, p_{i+1})$.

Hãy tìm một đường đi Hamilton có độ dài không quá 2.5×10^9. Không bắt buộc cực tiểu hóa độ dài đường đi.

Lời giải

Ta sẽ chia hình vuông $10^6 \times 10^6$ ban đầu thành 1000 hình chữ nhật $10^3 \times 10^6$. Ta sẽ lần lượt đi qua từng hình chữ nhật một, và đi qua các điểm trong cùng một hình chữ nhật theo tung độ không giảm nếu chỉ số của hình chữ nhật là lẻ và giảm dần nếu chỉ số là chẵn.

Ví dụ như hình dưới đây. Để dễ hình dung, hình mẫu sử dụng hình vuông 25×25 và chia thành 5 hình chữ nhật 5×25. Các đường nối thể hiện **thứ tự** của các điểm, không phải khoảng cách Manhattan.



Để tính được độ dài đường đi, ta thấy số bước đi theo trục tung và số bước đi theo trục hoành là độc lập:

- Do tung độ sắp xếp không giảm, nên tổng số bước của các điểm theo trục tung trong một hình chữ nhật tối đa là 10^6. Có 1000 hình chữ nhật, nên tổng số bước đi theo trục tung không vượt quá 10^9.
- Khi di chuyển giữa các điểm ở trong cùng một hình chữ nhật sẽ không đi quá 1000 bước theo trục hoành, khi di chuyển từ một điểm ở hình chữ nhật này sang một điểm ở hình chữ nhật khác sẽ đi không quá 2000 bước. Có 10^6 điểm cũng như không chuyển hình chữ nhật quá 1000 lần, nên số bước đi theo trục hoành là $10^9 + 2 \times 10^6$.
- Vậy cận trên của độ dài đường đi Hamilton nếu đi theo cách này là $2 \times 10^9 + 2 \times 10^6$, thỏa mãn yêu cầu đề bài.

Bài toán 2: Codeforces 86D - Powerful array

Tóm tắt đề bài

Cho mảng a gồm n phần tử nguyên dương có giá trị. Cho q truy vấn l, r. Xét đoạn con $a_l, a_{l+1}, \ldots, a_r$, với K_s là số lần xuất hiện của giá trị s trong đoạn, **sức mạnh** của đoạn con này là tổng của tất cả các tích $K_s \times K_s \times s$. Với mỗi truy vấn, hãy tính sức mạnh của mảng con đã cho.

Giới hạn

- $1 \leq n, q \leq 2 \times 10^5$.
- $1 \leq a_i \leq 10^6$.

Thuật toán ngây thơ

Với mỗi truy vấn, ta dùng vòng lặp để lặp qua từng phần tử $a_l, a_{l+1}, \ldots, a_r$ nhằm mục đích đếm số lần xuất hiện của từng giá trị sử dụng mảng đếm. Cuối cùng tính sức mạnh dựa vào mảng đếm hoặc các cấu trúc dữ liệu như `std::unordered_map` . Thuật toán này là không đủ tốt với độ phức tạp $\mathcal{O}(n \times q)$.

Thuật toán cải tiến

Trước tiên ta sẽ cải tiến thuật toán một chút, thay vì tính lại toàn bộ thông tin với từng truy vấn, ta lợi dụng thông tin đã tính ở truy vấn trước đó để giảm số phần tử phải xét.

Ví dụ, nếu trước đó ta xử lí truy vấn $[3, 7]$ và truy vấn kế tiếp là $[1, 6]$, ta nhận thấy hai đoạn con này có chung đoạn $[3, 6]$, ta chỉ phải thêm phần tử $1, 2$ và xóa phần tử 7 trong mảng đếm, cùng lúc đó tính lại đáp án. Phương pháp này so với khởi tạo lại mảng đếm và đếm lại từ đầu thì có tốt hơn.

```
1  // Tham số value là giá trị cần thêm hoặc xóa
2  // Tham số delta là 1 tương ứng với thêm, -1 là xóa
3
4  int current_answer;
5
6  void update(long long value, int delta){
7      // Khi thay đổi giá trị trong mảng đếm đồng thời cập nhật lại đáp án hiện
8      current_answer -= count[value] * count[value] * value;
9      count[value] += delta;
10     current_answer += count[value] * count[value] * value;
11 }
```

Một cách tổng quát, nếu truy vấn trước đó là $[l_i, r_i]$, truy vấn sau là $[l_{i+1}, r_{i+1}]$:

- Nếu $l_i < l_{i+1}$, ta cần xóa đoạn $a[l_i, l_i + 1, \ldots, l_{i+1} - 1]$.
- Nếu $l_{i+1} < l_i$, ta cần thêm đoạn $a[l_{i+1}, l_{i+1} + 1, \ldots, l_i - 1]$.

Tương tự:

- Nếu $r_i < r_{i+1}$, ta cần thêm đoạn $a[r_i + 1, r_i + 2, \ldots, r_{i+1}]$.
- Nếu $r_{i+1} < r_i$, ta cần xóa đoạn $a[r_i + 1, r_i + 2, \ldots, r_{i+1}]$.

Vậy số lượng phần tử cần thêm/xóa là giữa hai truy vấn là $|l_{i+1} - l_i| + |r_{i+1} - r_i|$. Và số lần cần thêm/xóa qua toàn bộ các truy vấn là: $\sum_{i=1}^{q-1} |l_{i+1} - l_i| + |r_{i+1} - r_i|$.

Nhưng độ phức tạp trong trường hợp tệ nhất của thuật toán trên vẫn là $\mathcal{O}(n \times q)$. Ví dụ với bộ test với $n, q = 2 \times 10^5$ và các truy vấn cố tình được sinh như sau:

$q_1 = [1, 1]$

$q_2 = [200000, 200000]$

$q_3 = [2, 2]$

$q_4 = [199999, 199999]$

...

Có thể thấy khoảng cách giữa hai đầu mút giữa hai truy vấn liên tiếp là rất lớn.

Thuật toán Mo

Bài toán có thể xử lí offline. Mấu chốt của thuật toán Mo là việc sắp xếp lại các truy vấn theo thứ tự hợp lí để đảm bảo tổng chi phí di chuyển giữa các truy vấn là đủ tốt.

Với mỗi truy vấn $[l, r]$, ta có thể coi nó như một điểm (l, r) trên hệ tọa độ. Chi phí để chuyển từ truy vấn (a, b) sang truy vấn (c, d) là $|a - c| + |b - d|$, chính là khoảng cách Manhattan giữa hai điểm (a, b) và (c, d).

Có q điểm tương ứng với q truy vấn, sắp xếp lại thứ tự các điểm sao cho tổng khoảng cách Manhattan giữa hai điểm liên tiếp là nhỏ nhất, và đó cũng chính là bài toán Hamilton Path được trình bày ở trên!

Ta sẽ sắp xếp các truy vấn giống như cách sắp xếp các điểm trong bài toán Hamilton Path:

- Đầu tiên chia các chỉ số vào các nhóm, cứ S chỉ số thì vào một nhóm (giá trị S sẽ được trình bày ở dưới), cụ thể chỉ số i sẽ vào nhóm $\lfloor \frac{i}{S} \rfloor$. Thao tác này tương ứng với thao tác chia hình chữ nhật.
- Tiếp theo sắp xếp các truy vấn theo **chỉ số nhóm** của đầu mút trái, tạm gọi giá trị này là L. Thao tác này tương ứng với việc đi qua lần lượt từng hình chữ nhật.
- Cuối cùng, các truy vấn có cùng L sẽ được sắp xếp theo đầu mút phải không giảm L lẻ, không tăng nếu L chẵn. Thao tác này tương ứng với các điểm trong cùng hình chữ nhật được sắp xếp không giảm hoặc không tăng theo tung độ tùy vào tính chẵn lẻ của hình chữ nhật.

Dưới đây là hàm so sánh khi sắp xếp các truy vấn viết bằng C++:

```
struct query{
    int l, r;  // hai đầu mút của truy vấn
    int id;    // chỉ số của truy vấn, vì sau khi sắp xếp ta sẽ mất thứ tự ban
}

bool cmp(const query &a, const query &b){
    if(a.l / S != b.l / S) // nếu đầu mút trái của hai truy vấn thuộc hai nhóm
        return a.l < b.l;  // sắp xếp dựa trên đầu mút trái

    // ngược lại nếu chỉ số nhóm của đầu mút trái lẻ thì sắp xếp không tăng th
    if((a.l / S) % 2 == 1)
        return a.r < b.r;
    else
        return a.r > b.r
}
```

Để chọn được S tối ưu: Chia mỗi nhóm S giá trị, vậy là có $\frac{n}{S}$ nhóm.

Số lần di chuyển đầu mút trái (từ l_i đến l_{i+1}):

- Nếu l_i và l_{i+1} thuộc cùng một nhóm: Số bước di chuyển là không quá S. Vậy số thao tác không vượt quá $S \times q$.
- Nếu l_i và l_{i+1} khác nhóm: Tổng số bước không vượt quá $2 \times n$.

Số lần di chuyển đầu mút phải (từ r_i đến r_{i+1}):

- Nếu l_i và l_{i+1} thuộc cùng một nhóm: Do đầu mút phải được sắp xếp tăng dần nên số thao tác di chuyển đầu mút phải không vượt quá n. Có $\frac{n}{S}$ nhóm, nên số thao tác không vượt quá $\frac{n^2}{S}$.
- Nếu l_i và l_{i+1} khác nhóm: Do điều chỉnh cách sắp xếp r_i mỗi khi đổi nhóm, tổng số thao tác trong cả hai trường hợp vẫn không vượt quá $\frac{n^2}{S}$.

Ta cần chọn S sao cho $S \times q + \frac{n^2}{S}$ nhỏ nhất. Theo bất đẳng thức AM-GM, đạt được giá trị nhỏ nhất khi $S = \frac{n}{\sqrt{q}}$, độ phức tạp thuật toán là $\mathcal{O}(n \times \sqrt{q})$.

Cài đặt mẫu

Phần cài đặt còn lại có thể như sau:

```
long long current_answer = 0;

void update(long long value, int delta){
    current_answer -= cnt[value] * cnt[value] * value * delta;
    cnt[val] += delta;
    current_answer += cnt[value] * cnt[value] * value * delta;
}

int main(){
    //...

    sort(q + 1, q + Q + 1, cmp);

    // l, r là đầu mút của đoạn đang xét hiện tại
    int l = 1, r = 0;
    for(int i = 1; i <= Q; i++){
        // Cần di chuyển hai đầu mút l, r đến truy vấn mới
        while(l < q[i].l) update(a[l++], -1);
        while(l > q[i].l) update(a[--l], 1);
        while(r < q[i].r) update(a[++r], 1);
        while(r > q[i].r) update(a[r--], -1);
        ans[q[i].id] = current_answer;
    }

    //...
}
```

Bài tập

- VNOJ - D-Query
- Codeforces 877E - Ann and books
- MarisaOJ - Tần suất
- Codeforces 617E - XOR and Favorite Number

3. Chia block

Đây là một kĩ thuật cũng thường được sử dụng để xử lí truy vấn trên mảng. Khác với thuật toán Mo, phương pháp chia block xử lí truy vấn cập nhật và trả lời truy vấn online dễ dàng hơn.

Để code được ngắn gọn, ở trong kĩ thuật này, chỉ số mảng trong các cài đặt bắt đầu từ 0. Các bài toán có chỉ số mảng bắt đầu từ 1 cần chuyển đổi phù hợp.

Bài toán 1: VNOJ - Point update, range query

Tóm tắt đề bài

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

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

Chia block

Chọn một hằng số $S = \sqrt{n}$, ta chia mảng thành các block S phần tử liên tiếp, như vậy sẽ có $\frac{n}{S} = \sqrt{n}$ nhóm. Nói cách khác, phần tử thứ i thuộc nhóm $\lfloor \frac{i}{S} \rfloor$. Phần tử đầu tiên của nhóm thứ i là $i \times S$, phần tử cuối cùng là $(i+1) \times S - 1$:

$$\underbrace{a_0\ a_1 \ldots a_{S-1}}_{\text{Block 0}}\ \underbrace{a_S\ a_{S+1} \ldots a_{2\times S-1}}_{\text{Block 1}} \ldots \underbrace{a_{t\times S}\ a_{t\times S+1} \ldots a_{t\times S-1}}_{\text{Block t}} \ldots$$

Ví dụ với dãy a có $n = 16$ phần tử, ta chọn $S = 4$ và chia thành dãy thành 4 block.

21				17				20				13			
5	8	6	3	2	7	2	6	7	1	7	5	6	2	3	2

Xử lí truy vấn cập nhật (i, x): Ta chỉ cần thay đổi a_i cũng như tổng của các phần tử trong block chứa a_i. Độ phức tạp của thao tác này là $\mathcal{O}(1)$. Ví dụ phần tử mang giá trị 7 được cập nhật thành 5, thì tổng của block đó cũng được cập nhật thành 15.

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

Xử lí truy vấn tính tổng (l, r): Ta tính tổng những block nằm gọn trong đoạn truy vấn và một số phần tử lẻ ra ở hai bên. Như ví dụ, tổng của đoạn truy vấn là $3 + (15) + (20) + 6 + 2 = 46$

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

Có không quá $\sqrt{n}$ block, và số lượng phần tử thừa ra hai bên không vượt quá $2 \times \sqrt{n}$ nên truy vấn tính tổng của độ phức tạp $\mathcal{O}(\sqrt{n})$.

Cụ thể hơn, với truy tính tổng, ta sẽ có hai bước, với L là block chứa l và R là block chứa r:

- Tính tổng các block từ $L + 1$ đến $R - 1$.
- Tính tổng các phần tử trong hai phần thừa $[l \dots L \times S - 1]$ và $[R \times S \dots r]$.
- Chú ý trường hợp đặc biệt $L = R$.

Độ phức tạp của bài toán là $\mathcal{O}(q \times \sqrt{n})$.

Cài đặt

Truy vấn cập nhật:

```
// sum[b] là tổng của block thứ b

void update(int i, int x){
    sum[i / S] -= a[i];
    a[i] = x;
    sum[i / S] += a[i];
}
```

Truy vấn tính tổng:

```
int sum(int l, int r){
    int answer = 0;

    // Block chứa l và r
    int block_l = l / S;
    int block_r = r / S;

```

```
9
10      // Trường hợp l và r nằm cùng block phải xử lí riêng
11      if(block_l == block_r){
12          for(int i = l; i <= r; i++){
13              answer += a[i];
14          }
15      }else{
16
17          // Tính tổng những phần thừa hai bên
18          for(int i = l; i < (block_l + 1) * S; i++){
19              answer += a[i];
20          }
21          for(int i = block_r * S; i <= r; i++){
22              answer += a[i];
23          }
24
25          // Tính tổng những block nằm hoàn toàn trong truy vấn
26          for(int i = block_l + 1; i < block_r; i++){
27              answer += sum[i];
28          }
29      }
30
31      return answer;
    }
```

Bài toán 2

Đề bài

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

- `1 l r k` : Đếm số lượng giá trị bằng k trong đoạn con $a_l, a_{l+1}, \ldots, a_r$.
- `2 l r x` : Tăng các phần tử trong đoạn con $a_l, a_{l+1}, \ldots, a_r$ lên x.

Giới hạn

- $1 \leq n, q, a_i \leq 10^9$.
- $0 \leq l, r \leq n - 1$.
- $0 \leq |k|, |x| \leq 10^9$.

Nếu các truy vấn đều có $l = 0$ và $r = n - 1$

Đầu tiên ta sẽ cần sử dụng một cấu trúc dữ liệu để thống kê số lần xuất hiện của từng giá trị ở trong mảng a, do các giá trị lớn nên ta chọn sử dụng `std::map` .

Với truy vấn cập nhật giá trị các phần tử, ta sẽ không trực tiếp thay đổi giá trị của n phần tử mà lưu một biến lazy, thể hiện các phần tử trong mảng đã được tăng lên bao nhiêu. Mỗi một lần cập nhật lại tăng lazy lên x. Truy vấn cập nhật có độ phức tạp $\mathcal{O}(1)$.

Với truy vấn đếm số lượng giá trị bằng k, ta biết các phần tử trong mảng đã được tăng lên lazy, nên cần đếm trong cấu trúc dữ liệu đã được chuẩn bị ban đầu có bao nhiêu giá trị bằng $k - \text{lazy}$. Truy vấn đếm có độ phức

tạp $\mathcal{O}(\log n)$.

Bài toán gốc

Chia mảng a thành các block $S = \sqrt{n}$ phần tử. Áp dụng tương tự cách làm cho trường hợp $l = 0, r = n - 1$, với mỗi block ta khởi tạo một cấu trúc dữ liệu đếm giá trị trong block đó.

Với những truy vấn cập nhật, đầu tiên ta cập nhật lại những block nằm hoàn toàn trong đoạn cần cập nhật. Ta lưu lazy_i với ý nghĩa block i đã được tăng lên bao nhiêu:

```
void update_block(int block_l, int block_r, int x){
    for(int i = l; i <= r; i++){
        lazy[i] += x;
    }
}
```

Tiếp theo là đến bước cập nhật các phần thừa ở hai bên, do số phần tử ít nên có thể duyệt qua từng phần tử và cập nhật. Nhưng cần lưu ý là lazy_i áp dụng cho trọn vẹn block i. Cập nhật phần thừa chỉ cập nhật lại một phần của block, vì vậy nên trước khi cập nhật phần thừa, phải cập nhật **thực sự** lazy_i vào các phần tử.

```
// count[i][j] là số lượng giá trị j nằm trong block i
map<int, int> count[];

void apply_lazy(int b){
    count[b].clear();

    // Cập nhật lazy[b] vào từng phần tử trong block b
    for(int i = b * S; i < (b + 1) * S; i++){
        a[i] += lazy[b];
        count[b][a[i]]++;
    }

    lazy[b] = 0;
}

// Cập nhật thủ công từ phần tử l đến r
void manual_update(int l, int r, int x){
    int b = l / S;
    for(int i = l; i <= r; i++){
        count[b][a[i]]--;

        if(count[b][a[i]] == 0){
            count[b].erase(a[i]);
        }

        a[i] += x;
        count[b][a[i]]++;
    }
}
```

```
31
32  void update(int l, int r, int x){
33      int block_l = l / S;
34      int block_r = r / S;
35
36      // Không quên trường hợp đặc biệt l, r nằm cùng trong một block
37      if(block_l == block_r){
38          apply_lazy(block_l);
39          manual_update(l, r, x);
40      }else{
41          update_block(block_l, block_r, x);
42
43          // Phải cập nhật lazy vào các block chứa l, r trước khi cập nhật phần
44          apply_lazy(block_l);
45          apply_lazy(block_r);
46
47          // Phần thừa
48          manual_update(l, (block_l + 1) * S - 1, x);
49          manual_update(block_r * S, r, x);
50      }
  }
```

Và khi truy vấn, ta đếm số lượng giá trị bằng $k - \text{lazy}_i$ trong từng block nguyên vẹn, và duyệt qua từng phần tử ở phần thừa.

Cả hai loại truy vấn có độ phức tạp $\mathcal{O}(\sqrt{n} \times \log \sqrt{n})$, nên độ phức tạp cuối cùng là $\mathcal{O}(q \times \sqrt{n} \times \log \sqrt{n})$.

Bài toán 3: MarisaOJ - Yếu vị

Tóm tắt đề bài

Cho một mảng a gồm n phần tử nguyên dương. Cho q truy vấn có dạng (l, r), hãy tìm số lần xuất hiện của phần tử xuất hiện nhiều nhất (số yếu vị) trong đoạn $a_l, a_{l+1}, \ldots, a_r$. **Các truy vấn phải xử lí online.**

Giới hạn

- $1 \leq n, q, a_i \leq 2 \times 10^5$.

Chia block

Ở bài toán này, ta cũng chia mảng đã cho thành $\sqrt{n}$ block liên tiếp, mỗi block gồm $\sqrt{n}$ phần tử.

Ở bài toán 1, 2, ta tính đáp án cho từng block, phần thừa rồi kết hợp chúng lại để ra đáp án cuối cùng.

Trong bài toán này, ta cũng có thể dễ dàng tìm được giá trị xuất hiện nhiều nhất trong một block cùng số lần xuất hiện của nó. Nhưng liệu có tồn tại cách để kết hợp đáp án các block lại một cách hiệu quả?

Vì vậy, ta sẽ có cách tiếp cận khác.

Định nghĩa $\text{mode}(i, j)$ giá trị xuất hiện nhiều nhất trong các **block** $i, i+1, \ldots, j$. Ta sẽ tính trước toàn bộ $\text{mode}(i, j)$ với $0 \leq i < j \leq \sqrt{n} - 1$.

Nếu truy vấn l, r bao toàn bộ các block $k, k+1, \ldots, p$. Có thể khẳng định trị xuất hiện nhiều nhất trong đoạn $A_{l \ldots r}$ sẽ là $\text{mode}(k, p)$ hoặc một giá trị nằm trong phần thừa ở hai bên.

> Chứng minh: Giả sử giá trị xuất hiện nhiều nhất là x không phải $\text{mode}(k, p)$ cũng như không nằm trong phần thừa, thì x chắc chắn chỉ xuất hiện trong các block $k, k+1, \ldots, p$. Nhưng điều này đồng nghĩa với việc x chính là $\text{mode}(k, p)$. Vậy có thể kết luận điều này không xảy ra!

Như vậy có nhiều nhất $2 \times \sqrt{n} + 1$ ứng viên cho số yếu vị của đoạn $a_{l \ldots r}$, nhiệm vụ còn lại là kiểm tra lần lượt từng giá trị này, đếm số lần xuất hiện trong đoạn truy vấn, và chọn ra giá trị xuất hiện nhiều nhất. Đếm số lần xuất hiện của một giá trị trên một đoạn con liên tiếp là bài toán cơ bản sử dụng tìm kiếm nhị phân.

Với thuật toán này, phần tính trước mode có độ phức tạp $\mathcal{O}(n \times \sqrt{n})$ và phần truy vấn có độ phức tạp $\mathcal{O}(q \times \sqrt{n} \times \log n)$.

Bài toán cũng tồn tại lời giải với độ phức tạp tốt hơn là $\mathcal{O}((n+q) \times \sqrt{n})$, cần cải tiến một chút từ thuật toán trên.

Nhận xét

Nhìn chung các bài toán sử dụng kĩ thuật chia block đều quy về việc xử lí được các block nguyên và phần thừa hai bên. Kĩ thuật có phần linh hoạt hơn thuật toán Mo do có thể hỗ trợ tốt hơn các thao tác cập nhật, cũng như trả lời được các truy vấn online.

Bài tập

- VNOJ - Inversion counting
- VNOJ - Minimum distance
- VNOJ - Vua Kẹo
- Codeforces 13E - Holes
- Codechef - FNCS

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