

Chia căn (sqrt decomposition) và ứng dụng: Phần 2

Chia căn (sqrt decomposition) và ứng dụng: Phần 2

Tác giả: Nguyễn RR Thành Trung

Tiếp nối chuỗi bài viết về các thuật toán chia căn, trong bài viết này chúng ta sẽ bàn về kĩ thuật tăng tốc độ trả lời truy vấn bằng cách sắp xếp chúng theo một thứ tự nhất định, còn gọi là **Mo's algorithm**.

Bài toán

Cho một dãy số A gồm N phần tử. Cần thực hiện Q truy vấn, mỗi truy vấn (i, j) yêu cầu tìm $mode(A_i, \dots, A_j)$. (Mode của một tập hợp là giá trị xuất hiện nhiều lần nhất trong tập hợp đó). Giới hạn: $N, Q, A_i \leq 10^5$.

Khi đọc đề một bài toán truy vấn kiểu này, có lẽ CTDL đầu tiên mà các bạn nghĩ đến là Interval Tree. Nhưng có điều gì đó không ổn trong bài này: Khi có thông tin của 2 nút con $[l, mid]$ và $[mid + 1, r]$, rất khó để tìm được bất kỳ thông tin hữu ích nào của $[l, r]$.

Duyệt

Chúng ta xuất phát từ thuật toán duyệt hồn nhiên như sau:

- Với mỗi truy vấn, ta for từ trái sang phải, đếm số lần xuất hiện.
- Trong khi đếm thì ta cập nhật kết quả.

Code đơn giản như sau:

```
1 function mode(l, r):
2     // Khởi tạo mảng count = toàn 0
3     res = -1;
4     for i = l .. r:
5         count[A[i]] += 1;
6         if res == -1 or count[A[i]] > count[res]:
7             res = A[i];
8     return res;
```

Dễ thấy, thuật toán duyệt này có độ phức tạp $\mathcal{O}(N \cdot Q)$. Có 2 lý do chính khiến thuật toán này chạy chậm:

1. Khởi tạo mảng count mỗi lần mất $\mathcal{O}(N)$.
2. Với mỗi truy vấn, phải tính lại mảng count từ đầu.

Ta có thể cải tiến được như sau:

Sau khi trả lời truy vấn $[l_1, r_1]$, để trả lời truy vấn $[l_2, r_2]$, bạn chỉ cần thay đổi mảng đếm một cách phù hợp. Cụ thể:

- Nếu $l_2 > l_1$, giảm số lần xuất hiện của $A_{l_1}, \dots, A_{l_2-1}$
- Nếu $l_2 < l_1$, tăng số lần xuất hiện của $A_{l_2}, \dots, A_{l_1-1}$
- Tương tự với r_1 và r_2 .

Để cập nhật số lần xuất hiện lớn nhất thì có thể dùng thêm set.

Như vậy, độ phức tạp của ta là tổng $\|l_i - l_{i-1}\| + \|r_i - r_{i-1}\|$, nhân thêm $\mathcal{O}(\log N)$ để đếm và tìm phần tử lớn nhất của mảng đếm.

Thuật toán Mo

Thuật toán Mo là một cách sắp xếp lại các truy vấn, sao cho tổng $\|l_i - l_{i-1}\| + \|r_i - r_{i-1}\|$ không quá $\mathcal{O}\left((N + Q)\sqrt{N}\right)$.

Thứ tự các truy vấn được định nghĩa qua hàm so sánh dưới đây.

```

1 | S = sqrt(N);
2 | bool cmp(Query A, Query B) // so sánh 2 truy vấn
3 | {
4 |     if (A.l / S != B.l / S) {
5 |         return A.l / S < B.l / S;
6 |     }
7 |     return A.r < B.r;
8 | }
```

Giải thích:

- Ta chia dãy thành các block (nhóm) độ dài $S = \sqrt{N}$.
- Nếu đầu trái của truy vấn nằm ở 2 block khác nhau, ta sắp xếp theo đầu trái.
- Ngược lại (đầu trái của truy vấn nằm ở cùng 1 block), ta sắp xếp theo đầu phải.

Chứng minh:

Mo's algorithm có độ phức tạp là $\mathcal{O}\left((N + Q)\sqrt{N}\right)$. Để hiểu tại sao độ phức tạp của thuật toán đạt được như vậy, chúng ta hãy cùng xem việc di chuyển các đoạn $[l_1, r_1]$ thành $[l_2, r_2]$:

- Di chuyển $l_1 \rightarrow l_2$:

- Nếu l_1 và l_2 cùng block: Với mỗi thao tác, độ phức tạp không quá \sqrt{N} . Do đó, độ phức tạp trong trường hợp này của cả Q thao tác là $\mathcal{O}(Q \cdot \sqrt{N})$.
- Nếu l_1 và l_2 khác block: Vì ta ưu tiên sort theo block chứa l , nên trường hợp này xảy ra không quá \sqrt{N} lần. Ở trường hợp này, ta mất độ phức tạp tối đa là $\mathcal{O}(N)$, nên với tất cả các thao tác, độ phức tạp là $\mathcal{O}(N \cdot \sqrt{N})$.
- Di chuyển $r_1 \rightarrow r_2$:
 - Nếu l_1 và l_2 cùng block: Vì trong cùng một block r được sắp xếp tăng dần, nên với mỗi block của l , ta chỉ mất độ phức tạp tổng là $\mathcal{O}(N)$. Do có \sqrt{N} block khác nhau của l , nên tổng độ phức tạp trong trường hợp này là $\mathcal{O}(N \cdot \sqrt{N})$.
 - Nếu l_1 và l_2 khác block: Như trên đã phân tích, ta chỉ có \sqrt{N} lần đổi block, mỗi lần đổi block ta mất độ phức tạp $\mathcal{O}(N)$ để di chuyển r . Do đó tổng độ phức tạp của trường hợp này là $\mathcal{O}(N \cdot \sqrt{N})$.

Vậy, độ phức tạp là $\mathcal{O}(N \cdot \sqrt{N} + Q \cdot \sqrt{N})$ hay $\mathcal{O}((N + Q)\sqrt{N})$.

Áp dụng

Sử dụng Mo's Algorithm, bạn đã có thể thu được một thuật toán hoàn chỉnh cho bài này với độ phức tạp $\mathcal{O}((N + Q)\sqrt{N})$:

- Sort tất cả các truy vấn theo Mo's Algorithm.
- Gọi $S(N)$ là một mảng gồm N set (có thể cài bằng hash table (bảng băm)). $S(i)$ chứa tất cả các số xuất hiện đúng i lần.
- Gọi $A(val)$ = số lần xuất hiện của val .
- Đặt max là chỉ số lớn nhất của mảng S mà $S(max)$ khác rỗng.
- Ta sẽ thêm và xóa một số trong $\mathcal{O}(1)$ như sau:
 - Thêm 1 số v :
 - Xóa v khỏi $S(A(v))$.
 - Tăng $A(v)$ thêm 1.
 - Thêm v vào $S(A(v))$.
 - Nếu $A(v) > max$, cập nhật max .
 - Xóa 1 số v :
 - Xóa v khỏi $S(A(v))$.
 - Giảm $A(v)$ đi 1.
 - Thêm v vào $S(A(v))$.
 - Nếu $S(max)$ rỗng, giảm max đi 1.

Vì tổng các thao tác thêm và xóa khi áp dụng Mo's Algorithm không quá $\mathcal{O}((N + Q)\sqrt{N})$, ta thu được một thuật toán với độ phức tạp này.

Mở rộng

Với mục đích làm bài toán khó hơn, ta xét trường hợp mà CTDL của ta chỉ cho phép thực hiện đúng 2 thao tác:

- ▶ **Insert:** Thêm 1 phần tử vào CTDL, thao tác này có độ phức tạp là $\mathcal{O}(\log N)$ hoặc $\mathcal{O}(1)$.
- ▶ **Snapshot:** Lưu lại trạng thái hiện tại của CTDL. Thao tác này có độ phức tạp $\mathcal{O}(N)$.
- ▶ **Rollback:** Hồi phục lại trạng thái của CTDL ở lần Snapshot cuối. Thao tác này cũng có độ phức tạp là $\mathcal{O}(N)$.

Một ví dụ của CTDL loại này là Disjoint set, và việc xử lý truy vấn xuất hiện trong bài toán Codechef - GERALD07.

Cách làm vẫn là áp dụng Mo's algorithm, tuy nhiên vì không thể xóa phần tử, nên ta không thể di chuyển từ l_1 sang l_2 một cách dễ dàng được.

Để đơn giản, chúng ta chỉ xét những truy vấn $[l, r]$ mà l và r rơi vào 2 block khác nhau. Để giải quyết việc không di chuyển ngược được, sau khi trả lời truy vấn $[l, r]$, chúng ta cần dùng Rollback để đưa l về cuối block chứa l . Sau đó, khi trả lời truy vấn $[l_2, r_2]$, chúng ta chỉ cần thực hiện Insert từ $r + 1$ đến r_2 và từ l_2 đến cuối block chứa l_2 .

Chi tiết cài đặt:


```

1  rt = sqrt(n);
2  init(); // this initializes our data structure (clears it)
3  snapshot();
4  for all queries q
5      if q.r - q.l + 1 <= rt + 1 // we process light queries
6          for j := q.l to q.r
7              insert(j);
8          store answer for query q;
9          rollback();
10 last_bucket = -1;
11 for all queries q
12     if q.r - q.l + 1 <= rt + 1: continue;
13     bucket = q.l / rt;
14
15     if bucket != last_bucket
16         init();
17         l = (bucket + 1) * rt; // right border of the bucket
18         r = q.r;
19         for j := l to r
20             insert(j);
21     last_bucket = bucket;
22
23     while r < q.r
24         insert(++r);
25     snapshot();
26     for j := q.l to l - 1
27         insert(j);
28



```

```
29 | store answer for query q;  
rollback();
```

Cải tiến

Các bạn có thể đọc thêm về cách cải tiến tốc độ chạy của Mo sử dụng TSP và Hilbert curve [tại đây](#) .

Bài tập áp dụng

- [Codeforces Yandex 2011 Round 2 - D](#) 
- [Codechef - GERALD07](#) 

Được cung cấp bởi [Wiki.js](#)