

## Chia căn (sqrt decomposition) và ứng dụng: Phần 1

# Chia căn (sqrt decomposition) và ứng dụng: Phần 1

**Tác giả:** Hoàng Xuân Nhật & Vương Hoàng Long

Chia căn là tên gọi chung của một nhóm các thuật toán thường liên quan đến việc chia các đối tượng thành  $\sqrt{N}$  phần.

Sau đây ta sẽ xét một dạng đơn giản nhất: chia mảng ra làm  $\sqrt{N}$  đoạn, thường dùng để giải quyết các bài toán truy vấn.

## Bài toán 1

Cho một mảng  $A$  gồm  $N$  phần tử là các số nguyên không âm. Bạn cần trả lời  $Q$  truy vấn, mỗi truy vấn có dạng  $(l, r, k)$  yêu cầu tìm đếm số phần tử của  $A$  nằm trong đoạn  $[l, r]$  có giá trị bằng  $k$ . Giới hạn:  $N, Q, A_i \leq 10^5$ .

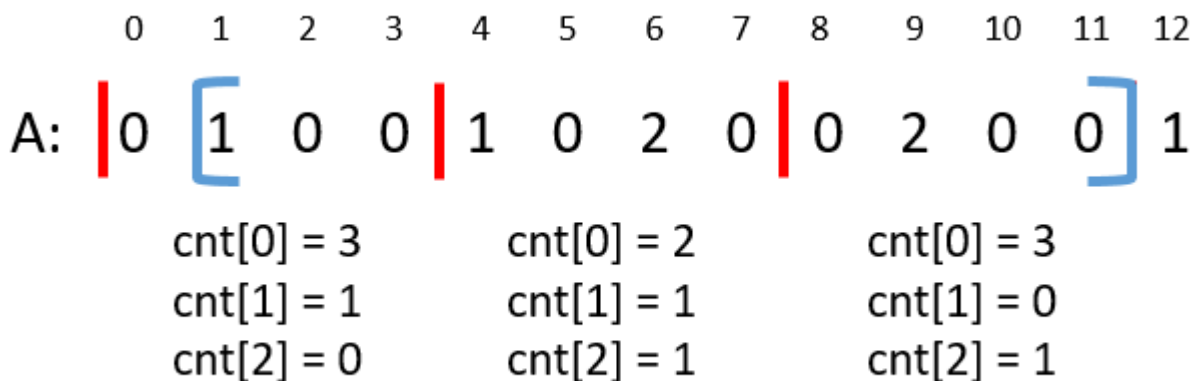
Giả sử ta luôn có  $l = 1$  và  $r = N$ , bài toán trên có thể giải đơn giản bằng cách tạo một mảng  $cnt[x] =$  số phần tử của mảng có giá trị bằng  $x$ .

Ta áp dụng ý tưởng này để giải bài toán tổng quát, bằng cách tạo ra  $\sqrt{N}$  mảng  $cnt$ , mỗi mảng quản lý một đoạn  $\sqrt{N}$  phần tử liên tiếp của  $A$ . Để hiểu rõ hơn, ta có thể xem ví dụ sau.

Ta có mảng  $A$  gồm 13 phần tử, chỉ số được đánh bắt đầu từ 0. Ta sẽ chia mảng  $A$  thành các đoạn có độ dài 4, đoạn cuối cùng sẽ chỉ chứa 1 phần tử. Nội dung mảng  $A$  và các mảng  $cnt$  đã được tính sẵn như trong hình sau:

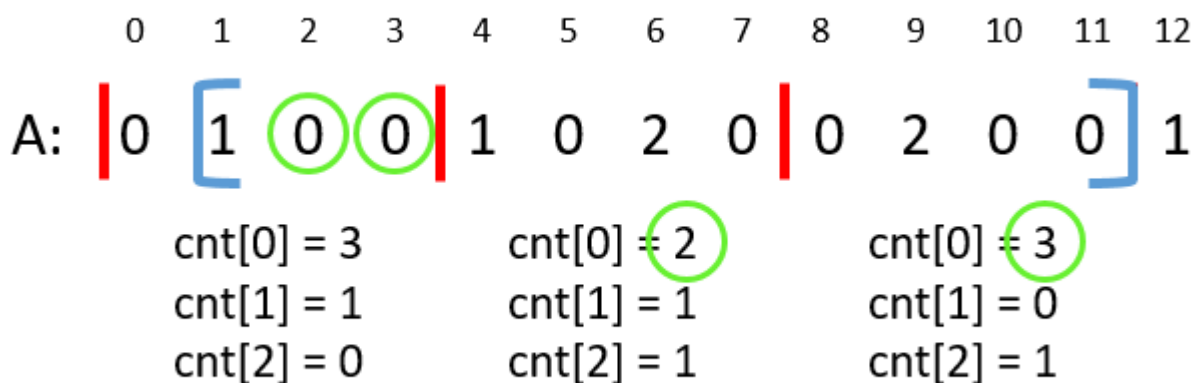
	0	1	2	3	4	5	6	7	8	9	10	11	12
A:	0	1	0	0	1	0	2	0	0	2	0	0	1
	cnt[0] = 3				cnt[0] = 2				cnt[0] = 3				
	cnt[1] = 1				cnt[1] = 1				cnt[1] = 0				
	cnt[2] = 0				cnt[2] = 1				cnt[2] = 1				

Với cấu trúc trên, ta có thể dễ dàng trả lời các truy vấn. Ví dụ, xét truy vấn  $(1, 11, 0)$ .



Có thể thấy, đoạn truy vấn sẽ luôn được chia thành các đoạn chứa đủ  $\sqrt{N}$  phần tử (trong trường hợp này là đoạn [4..7] và [8..11]), và có thể thêm một số đoạn không đầy đủ ở hai đầu (ở đây là đoạn [1..3]).

Với những đoạn đầy đủ, ta cộng  $cnt[0]$  của chúng vào kết quả. Với những đoạn không đầy đủ, ta xét từng phần tử. Phần tử nào bằng 0, ta sẽ tăng biến đếm kết quả lên 1. Với truy vấn (1, 11, 0), ta có kết quả là  $2 + 3 + 1 + 1 = 7$ .



Cấu trúc trên vẫn có thể giải bài toán này khi có thêm truy vấn **chỉnh sửa một phần tử** của  $A$ , bạn chỉ cần thay đổi giá trị  $cnt$  của một đoạn duy nhất chứa phần tử cần cập nhật.

## Phân tích

Đầu tiên, ta phải trả lời được câu hỏi: tại sao lại chia thành  $\sqrt{N}$  đoạn, mà không phải 1, 2, 10,  $N/2$ , ... ?

Gọi số đoạn ta chia ra là  $S$ . Vậy mỗi đoạn sẽ có độ dài  $N/S$  (ta tạm bỏ qua đoạn cuối).

Khi truy vấn, ta phải xét 2 thứ: một là những đoạn đầy đủ, nằm trong đoạn truy vấn. Hai là đoạn dư ra ở hai đầu của truy vấn.

Với những đoạn đầy đủ, trong trường hợp tệ nhất chúng ta phải xét cả  $S$  đoạn. Mỗi đoạn ta cộng  $cnt[k]$  vào kết quả trong  $O(1)$ , vậy tổng cộng mất  $O(S)$ .

Với đoạn dư ra ở hai đầu, ta xét riêng từng phần tử mất  $O(1)$ . Các đoạn đều có độ dài  $N/S$ , nên ta mất  $O(N/S)$  cho phần này.

Mỗi truy vấn ta mất thời gian là  $O(S + N/S)$ . Ta cần tìm giá trị  $S$  sao cho  $S + N/S$  đạt giá trị nhỏ nhất. Áp dụng [bất đẳng thức AM-GM](#)  $\square$ , giá trị này là nhỏ nhất khi  $S = N/S \iff S = \sqrt{N}$ . Thời gian để thực hiện  $Q$  truy vấn sẽ là  $O(Q\sqrt{N})$ .

## Cài đặt

Ta cần phải lưu những cấu trúc sau:

- $\sqrt{N}$  mảng  $cnt$ , mỗi mảng có độ dài  $max(A_i)$ , tốn  $O(\sqrt{N} * max(A_i))$  bộ nhớ.
- Mảng  $A$ , tốn  $O(N)$  bộ nhớ.

Khi giải bài toán, ta thường chia thành các hàm tiền xử lý để dựng ra cấu trúc dữ liệu, và các hàm trả lời các truy vấn.

```

1  const int BLOCK_SIZE = 320;
2  const int N = 1e5 + 2;
3
4  int n;
5  int cnt[N / BLOCK_SIZE + 2][N];
6  int a[N];
7
8  void preprocess()
9  {
10     for (int i = 0; i < n; ++i)
11         ++cnt[i / BLOCK_SIZE][a[i]];
12 }
```

Sau khi đã tiền xử lý, hàm trả lời truy vấn có thể cài đặt như sau. Lưu ý, ta phải xét riêng trường hợp cả hai đầu của truy vấn nằm trong cùng một đoạn. Trong code bên dưới, tác giả dùng [hàm count](#)  $\square$  của thư viện C++ STL.

```

1  int query(int l, int r, int k)
2  {
3      int blockL = (l + BLOCK_SIZE - 1) / BLOCK_SIZE;
4      int blockR = r / BLOCK_SIZE;
5      if (blockL >= blockR)
6          return count(a + l, a + r + 1, k);
7
8      int sum = 0;
9      for (int i = blockL; i < blockR; ++i)
10         sum += cnt[i][k];
11
12     for (int i = l, lim = blockL * BLOCK_SIZE; i < lim; ++i)
13         if (a[i] == k) ++sum;
14
15     for (int i = blockR * BLOCK_SIZE; i <= r; ++i)
16         if (a[i] == k) ++sum;
17
18 }
```

```

18 |     return sum;
19 | }

```

Thao tác cập nhật một phần tử có thể thêm vào như sau (với  $u$  là vị trí cần cập nhật, và  $v$  là giá trị mới):

```

1 | void update(int u, int v)
2 | {
3 |     int block = u / BLOCK_SIZE;
4 |     --cnt[block][a[u]];
5 |     a[u] = v;
6 |     ++cnt[block][a[u]];
7 | }


```

## Bài toán 2

Tiếp nối bài toán đầu tiên, chúng ta hãy cùng đi sâu hơn vào các bài toán chia mảng ra làm  $\sqrt{N}$  đoạn nhưng có truy vấn cập nhật.

**Lưu ý:** Bài tập có cách giải tối ưu nhất sử dụng Segment Tree, tuy nhiên vì mục đích của bài viết này nên bài tập sẽ được giải bằng chia căn.

### Đề bài

Các bạn có thể nộp bài ở [đây](#) 

Cho một mảng  $A$  gồm  $N$  phần tử là các số nguyên. Bạn cần thực hiện  $Q$  truy vấn có dạng  $(l, r, oval, nval)$  là với các phần tử trong đoạn từ  $l$  đến  $r$ , nếu  $A[i] == oval$ , gán  $A[i] = nval$ . Bạn cần in ra mảng sau khi thực hiện  $Q$  truy vấn. Giới hạn  $1 \leq N, Q \leq 2 * 10^5, 1 \leq A_i \leq 100$

Ghi chú:  $oval, nval$  là viết tắt cho *old value* và *new value*.

### Cách giải

Giả sử các truy vấn đều có  $l = 1, r = N$

Với giả sử trên, ta sẽ giải bài toán với đpt  $O(Q * 100 + N)$ . Ta sẽ tạo mảng  $lazy[oval]$  với ý nghĩa là các số ban đầu là  $oval$  thì hiện tại đã được đổi giá trị sang  $lazy[oval]$ . Ban đầu  $lazy[oval] = oval$  với  $1 \leq oval \leq 100$ . Với mỗi truy vấn  $(l, r, oval, nval)$ , ta sẽ làm như sau:

```

1 | for (int i = 1; i <= 100; ++i) {
2 |     if (lazy[i] == oval) lazy[i] = nval;
3 | }

```

Với thao tác cập nhật mảng lazy này, về mặt ý nghĩa, tất cả các số hiện đang có giá trị là  $oval$  sẽ được gán lại thành  $nval$ .

Sau khi thực hiện tất cả các truy vấn, chúng ta có thể lấy giá trị của các số trong mảng như sau:

```

1 | for (int i = 1; i <= n; ++i) {
2 |     a[i] = lazy[a[i]];
3 | }

```

Vậy là chúng ta đã giải xong bài toán với độ phức tạp  $O(Q * 100 + N)$ .

### Giải bài toán gốc

Ta sẽ áp dụng ý tưởng trên vào để giải bài toán gốc. Ta cũng chia mảng thành  $\sqrt{N}$  đoạn. Xét một truy vấn  $(l, r, oval, nval)$  ta có:

- $blockL$  là block đầu tiên ở bên phải  $l$
- $blockR$  là block chứa  $r$
- Với mỗi block, ta sẽ có mảng *lazy* với định nghĩa như trên. Ví dụ block 3, các số đang có giá trị là *oval* sẽ được đổi thành giá trị *nval*  $\Leftrightarrow lazy[3][oval] = nval$

Vậy truy vấn của chúng ta sẽ được chia làm 3 phần (có thể rỗng) như sau:

- Phần dư bên trái:  $[l \dots blockL * BLOCK\_SIZE - 1]$
- Phần dư bên phải:  $[blockR * BLOCK\_SIZE \dots r]$
- Phần đầy đủ các block:  $[blockL * BLOCK\_SIZE \dots blockR * BLOCK\_SIZE - 1]$

### Đầu tiên, chúng ta cập nhật phần đầy đủ các block:

Ta sẽ cập nhật lần lượt cho từng block đơn lẻ. Gọi block hiện tại là  $id$ , ta sẽ làm tương tự như khi giải bài toán  $l = 1, r = N$ :

```

1 | void blockUpdate(int id, int oval, int nval) {
2 |     for (int i = 1; i <= LIM; ++i) {
3 |         if (lazy[id][i] == oval) {
4 |             lazy[id][i] = nval;
5 |         }
6 |     }
7 | }

```

Vậy là chúng ta đã cập nhật xong cho tất cả các block thuộc phần đầy đủ các block. Chú ý, việc cập nhật này chúng ta chỉ đánh dấu là các phần tử đang có giá trị là *oval* sẽ được thay đổi thành *nval*. Giá trị của các phần tử trong đoạn này sau cập nhật không có sự thay đổi nào (ý tưởng giống như [Lazy Propagation](#)).

### Tiếp theo, chúng ta cập nhật phần dư bên trái:

Gọi block của phần dư bên trái là  $id$ .

Vì phần dư bên trái không bao phủ trọn vẹn 1 block, nên chúng ta sẽ không thể dùng mảng *lazy* để cập nhật được như ở trên. Thay vào đó chúng ta sẽ phải duyệt từng phần tử trong phần này và cập nhật (xét mỗi phần tử, nếu giá trị của nó là *oval* thì gán giá trị mới là *nval*):

```

1 void manualUpdate(int L, int R, int oval, int nval) { // L R là đầu trái và đầu
2     for (int i = L; i <= R; ++i) {
3         if (a[i] == oval) {
4             a[i] = nval;
5         }
6     }
7 }

```

Tuy nhiên, các phần tử trong *phần dư bên trái* này có thể đang chịu ảnh hưởng từ mảng *lazy* của các truy vấn trước đó, nên chúng ta cần *thực sự cập nhật* các phần tử này bằng mảng *lazy*, sau đó mới thực hiện *manualUpdate* (giống như bước Propagate trong Lazy Propagation).

```

1 void doLazy(int id) { // L R là đầu trái và đầu phải của phần dư bên trái
2     int L = id * BLOCK_SIZE;
3     int R = min(n - 1, (id + 1) * BLOCK_SIZE - 1);
4     for (int i = L; i <= R; ++i) {
5         a[i] = lazy[id][a[i]]; // thay đổi giá trị các phần tử bằng mảng lazy
6     }
7     for (int i = 1; i <= 100; ++i) {
8         lazy[id][i] = i; // đã cập nhật xong, reset lại mảng lazy về ban đầu
9     }
10 }

```

Vậy tổng kết lại, ta sẽ có hàm cập nhật cho *phần dư bên trái* (và cả *\*phần dư bên phải*) như sau:

```

1 void manualUpdate(int L, int R, int oval, int nval) { // L R là đầu trái và đầu
2     doLazy(R / BLOCK_SIZE); // R / BLOCK_SIZE chính là block của của phần này.
3     for (int i = L; i <= R; ++i) {
4         if (a[i] == oval) {
5             a[i] = nval;
6         }
7     }
8 }
9
10 /* Chúng ta sẽ gọi hàm như dưới đây để cập nhật cho phần dư bên trái */
11 manualUpdate(1, blockL * BLOCK_SIZE - 1, oval, nval);
12
13 /* Chúng ta sẽ gọi hàm như dưới đây để cập nhật cho phần dư bên phải */
14 manualUpdate(blockR * BLOCK_SIZE, r, oval, nval);

```

## Phân tích

*Ghi chú:* Vì hằng số của lời giải này tương đối lớn nên tác giả sẽ giữ hằng số trong độ phức tạp khi cần thiết


Ta sẽ cùng xem xét độ phức tạp của lời giải này:

Dễ thấy hàm *blockUpdate* có độ phức tạp là  $O(100)$ . Hàm này mỗi truy vấn có thể được gọi không quá  $\sqrt{N}$  lần, và có  $Q$  truy vấn nên tổng độ phức tạp của các lần gọi hàm này là  $O(100 \times Q\sqrt{N})$ . **(1)**


Hàm *doLazy* có độ phức tạp là  $O(\sqrt{N} + 100)$  do các *phần dư* có độ lớn  $\leq \sqrt{N}$ . Cộng với phần `for (int i = L; i <= R; ++i)` có độ phức tạp  $O(\sqrt{N})$ , hàm *manualUpdate* có độ phức tạp là  $O(\sqrt{N} + \sqrt{N} + 100) = O(\sqrt{N})$ .

Dễ thấy hàm *manualUpdate* sẽ được gọi đúng 2 lần trong mỗi truy vấn. Vậy tổng độ phức tạp của việc gọi hàm này là  $O(Q\sqrt{N})$ . **(2)**

Vậy độ phức tạp của lời giải chia căn này sẽ là **(1) + (2)** =  $O(100 \times Q\sqrt{N}) + O(Q\sqrt{N})$ .

Các bạn có thể xem code mẫu ở [đây](#) .

## Bài tập áp dụng

Các bạn có thể thử sức [tại đây](#) .

## Lưu ý

- Trong phần lớn trường hợp, ta nên đặt *BLOCK\_SIZE* là hằng số, chứ không nên thực sự lấy căn của  $N$  trong dữ liệu nhập vào. Lý do là việc chia cho hằng số, cũng như việc dùng mảng tĩnh sẽ giúp code của bạn chạy nhanh hơn nhiều so với việc chia cho biến và xài mảng động.
- Khi cài đặt, các bạn cần tránh việc thực hiện  $O(Q\sqrt{N})$  phép chia (cả chia lấy nguyên lẫn chia lấy dư), vì phép chia là một thao tác chậm hơn nhiều so với các phép toán khác. Các bạn dễ làm điều này khi cần tính *id* của các block lúc truy vấn/cập nhật. Phạm vào điều này nhiều khả năng sẽ khiến code bạn bị chạy quá thời gian (TLE).
- Vì thường yêu cầu bộ nhớ lớn, các bạn cần tính toán để không bị quá bộ nhớ (MLE). Cách tính như sau:  $1\text{MB} = 10^6$  byte,  $1 \text{ int} = 4$  byte,  $1 \text{ long long} = 8$  byte. Ví dụ, mảng *cnt* trong code mẫu ở trên sẽ tốn  $320 * 10^5 * 4 = 128000000 = 128\text{MB}$ .

## Mở rộng

Chia căn còn rất nhiều dạng khác. Các bạn có thể đọc tiếp về kĩ thuật này tại [Phần 2](#) .

Được cung cấp bởi [Wiki.js](#)