

Cây chỉ số nhị phân (Binary Indexed Tree)

Cây chỉ số nhị phân (Binary Indexed Tree)

Tác giả:

- Bùi Nguyễn Đức Tân - Phổ thông Năng khiếu, Đại học Quốc gia Thành phố Hồ Chí Minh
- Lê Minh Hoàng - Phổ thông Năng khiếu, Đại học Quốc gia Thành phố Hồ Chí Minh

Reviewer:

- Nguyễn Xuân Tùng - Đại học Quốc Tế, Đại học Quốc gia Thành phố Hồ Chí Minh

Giới thiệu

Cây chỉ số nhị phân (tên tiếng Anh là Binary Indexed Tree) hay cây Fenwick là một cấu trúc dữ liệu được sử dụng khá phổ biến trong lập trình thi đấu vì có thể cài đặt nhanh, dễ dàng so với các CTDL khác.

Bài toán

Cho mảng A gồm N phần tử (đánh số từ 1). Có Q truy vấn thuộc 2 loại:

- 1 u v : cộng v vào $A[u]$.
- 2 p : tính tổng các phần tử từ $A[1], A[2], A[3], \dots, A[p]$.

Giới hạn: $N, Q \leq 2 \cdot 10^5$

Ngây thơ 1

Với truy vấn loại 1, ta đơn thuần tăng phần tử $a[u]$ thêm v .

Với truy vấn loại 2, ta duyệt qua tất cả phần tử trong đoạn $[1 \dots p]$ và cộng giá trị vào biến kết quả.

```
1 | const int N = 200003;
2 | int a[N];
3 |
4 | void update(int u, int x) {
5 |     a[u] = a[u] + x;
6 | }
7 |
8 | int getSum(int p) {
9 |     int ans = 0;
10 |    for (int i = 1; i <= p; ++i)
11 |        ans = ans + a[i];
12 |    return ans;
13 | }
```

Phân tích

- Độ phức tạp khi update: $\mathcal{O}(1)$.
- Độ phức tạp khi truy vấn: $\mathcal{O}(p) = \mathcal{O}(N)$.
- Có Q truy vấn, vì thế độ phức tạp là $\mathcal{O}(Q + Q \cdot N) = \mathcal{O}(Q \cdot N)$
- Nếu chưa biết về độ phức tạp tính toán, các bạn có thể đọc ở [đây](#).

Đối chiếu giới hạn, cách "ngây thơ" trên tỏ ra chậm chạp, không đủ để xử lý yêu cầu bài toán.

Ngây thơ 2

Nhận thấy đây là một dạng của bài toán Range Sum Query, ta có thể áp dụng mảng cộng dồn (prefix sum) để tính nhanh tổng một đoạn.

Khi cập nhật giá trị một phần tử, ta đồng thời cập nhật tất cả các prefix chứa phần tử đó.

```

1  int sum[N];
2
3  void preprocess() {
4      sum[1] = a[1];
5      for (int i = 2; i <= n; ++i) {
6          sum[i] = sum[i - 1] + a[i];
7      }
8  }
9
10 void update(int u, int x) {
11     for (int i = u; i <= n; ++i) {
12         sum[i] = sum[i] + x;
13     }
14 }
15
16 int getSum(int p) {
17     return sum[p];
18 }

```

Phân tích

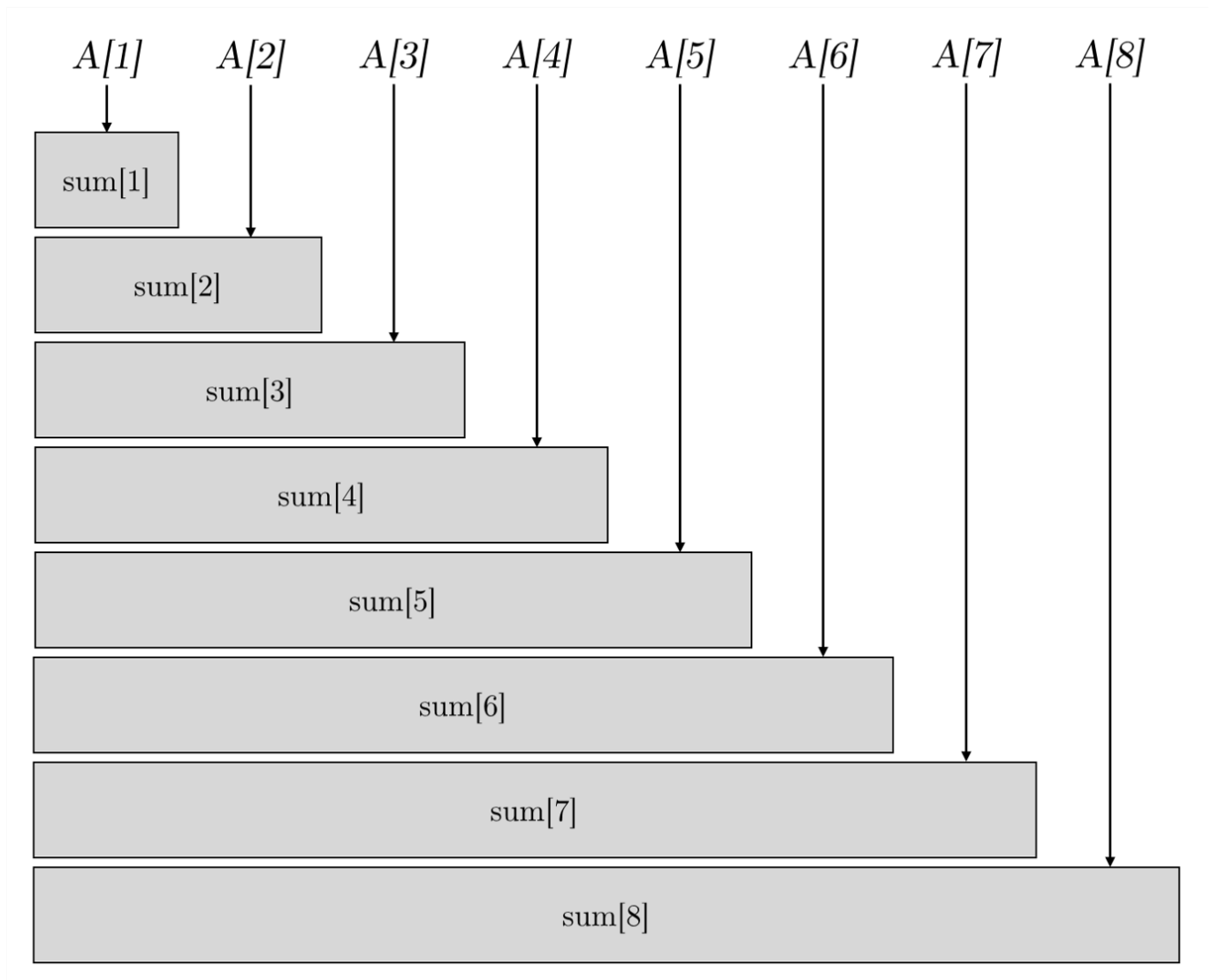
- Độ phức tạp tiền xử lý: $\mathcal{O}(N)$
- Độ phức tạp khi update: $\mathcal{O}(p) = \mathcal{O}(N)$
- Độ phức tạp khi truy vấn: $\mathcal{O}(1)$

Nếu bài toán không có truy vấn cập nhật, độ phức tạp là $\mathcal{O}(Q + N)$, đủ nhanh để giải quyết. Tuy nhiên, khi có thao tác cập nhật, độ phức tạp bị đẩy lên $\mathcal{O}(Q * N)$ - tương đương với độ phức tạp của cách ngây thơ 1.

Để có thể giải quyết bài toán một cách hiệu quả, ta cần một CTDL có thể sử dụng tính chất prefix sum để trả về kết quả nhanh, đồng thời có thể nhanh chóng cập nhật giá trị cho prefix.

Cây chỉ số nhị phân

Cấu trúc prefix sum được biểu diễn qua sơ đồ sau:



Nhận xét: Mỗi phần tử $sum[i]$ chứa tổng của tất cả phần tử từ $[1 \dots i]$; vì thế, phần tử $sum[i]$ sẽ chứa phần tử $a[j]$ nếu thỏa $i \geq j$, số phần tử sum cần cập nhật là $j - i + 1$, gần tương đương độ dài của mảng.

Để tăng tốc độ cập nhật phần tử, cần bố trí lại phạm vi của từng đoạn gần với $sum[i]$ để cực tiểu số phần tử sum cần cập nhật nhưng vẫn phải đảm bảo tính liên tục để áp dụng tính chất của prefix sum.

Giới thiệu tổng quát

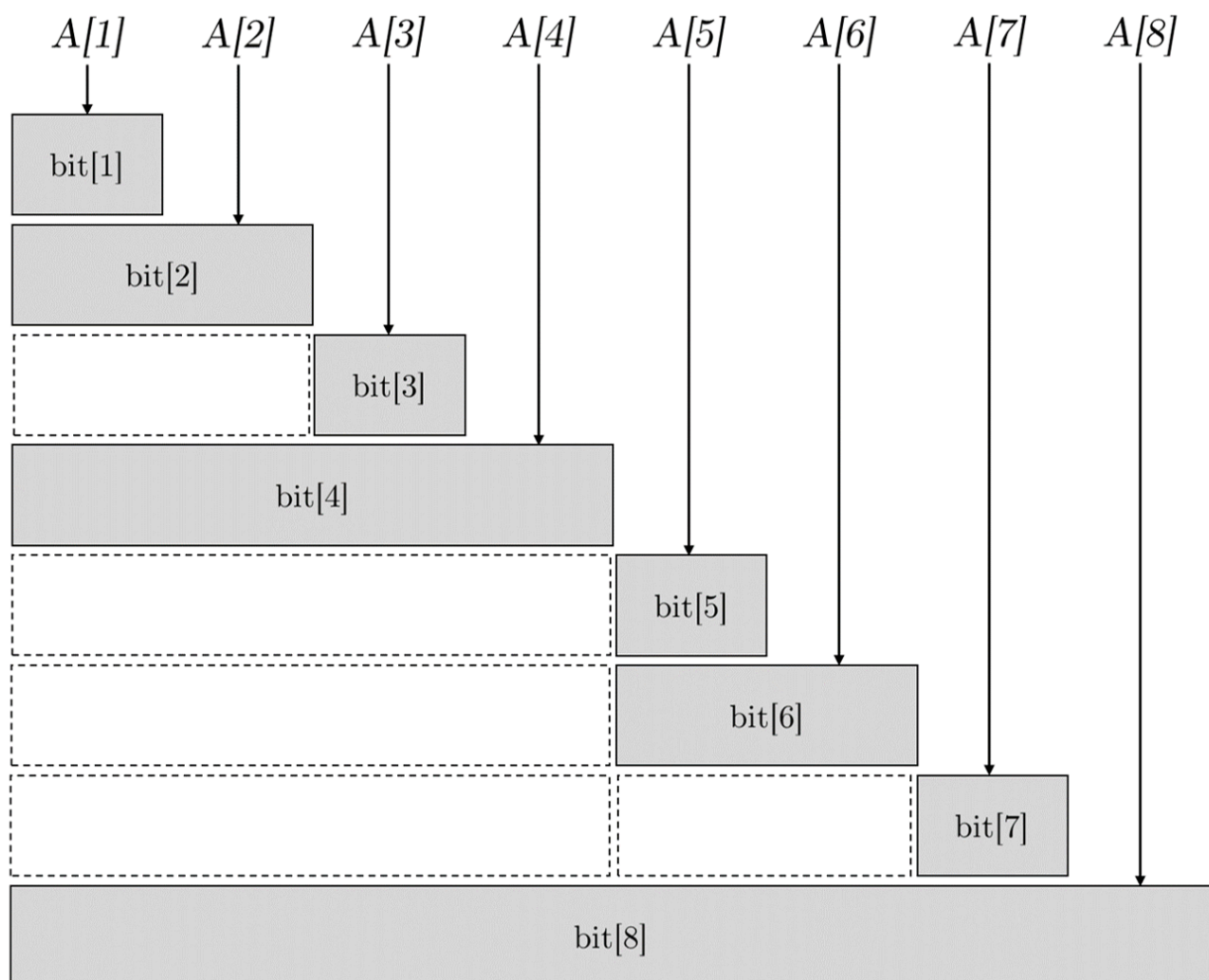
Mỗi chỉ số n đều có thể biểu diễn bằng tổng của các lũy thừa cơ số 2, vì thế, để tính tổng của các phần tử thuộc $[1 \dots n]$, ta có thể tách đoạn này thành các đoạn nhỏ hơn có độ dài 2^k và cộng lại tổng tính được trên từng đoạn.

Cụ thể, đặt $n = 2^{i_1} + 2^{i_2} + \dots + 2^{i_k}$ ($i_1 > i_2 > \dots > i_k \geq 0$). Để tính tổng từ $[1 \dots n]$, ta tính tổng các phần tử thuộc đoạn $[1; 2^{i_1}]$, sau đó tính tiếp tổng của đoạn $[2^{i_1} + 1; 2^{i_1} + 2^{i_2}]$, lặp lại quá trình này cho đến khi ta đến đoạn cuối cùng là $[2^{i_1} + 2^{i_2} + \dots + 2^{i_{k-1}} + 1; n]$. n có thể có tối đa $\log_2 n$ bits, vì thế độ phức tạp khi tính tổng theo cách này là $\mathcal{O}(C \log n)$, trong đó $\mathcal{O}(C)$ là độ phức tạp khi lấy tổng một đoạn.

$$\begin{aligned}
 1 &= 2^0 &= [1...1] \\
 2 &= 2^1 &= [1...2] \\
 3 &= 2^1 + 2^0 &= [1...2] + [3..3] \\
 4 &= 2^2 &= [1...4] \\
 5 &= 2^2 + 2^0 &= [1...4] + [5...5] \\
 6 &= 2^2 + 2^1 &= [1...4] + [5...6] \\
 7 &= 2^2 + 2^1 + 2^0 &= [1...4] + [5...6] + [7...7] \\
 8 &= 2^3 &= [1...8]
 \end{aligned}$$

Từ cách chia block trên, ta quan sát được rằng block cuối cùng đối với mỗi n (là block tổng chứa phần tử ở chỉ số n) có độ dài bằng với bit nhỏ nhất trong biểu diễn nhị phân của n . Đây chính là ý tưởng của cây BIT, ta sẽ lưu thông tin về block cuối của từng phần tử và thực hiện thao tác truy vấn trên đây.

Dưới đây là hình ảnh minh họa cây BIT:



Trong hình trên, những đoạn được tô đậm là đoạn của phần tử chỉ số n được BIT lưu trữ; những đoạn được tô nét mảnh không được lưu trữ trực tiếp mà sẽ được truy cập gián tiếp.

Cài đặt BIT

Mặc dù có bản chất là cây, tính chất ở trên cho phép chúng ta lưu trữ BIT dưới dạng một mảng đơn giản có độ dài bằng với độ dài mảng ta đang làm việc.

```
1 | int bit[N];
```

Thao tác tìm tổng

Để tìm tổng các phần tử trong đoạn $[1 \dots n]$, ta sẽ lần lượt đi qua tất cả bit của n theo giá trị tăng dần. Mỗi lần đi qua n , ta sẽ cộng $bit[n]$ vào kết quả hiện tại, rồi trừ đi bit nhỏ nhất của n khỏi chính nó; quá trình lặp lại cho đến khi $n = 0$.

Để lấy bit nhỏ nhất của một số n , ta có thể sử dụng công thức $n \& \sim(n - 1)$ được đề cập tại bài tại [đây](#). Khi thao tác bit với số âm, C++ sử dụng phép bù 2: $\sim n = -n - 1$; vì vậy ta có phép biến đổi: $n \& \sim(n - 1) = n \& (-(n - 1) - 1) = n \& (-n)$ để sử dụng hơn.

```
1 | int getSum(int p) {
2 |     int idx = p, ans = 0;
3 |     while (idx > 0) {
4 |         ans += bit[idx];
5 |         idx -= (idx & (-idx));
6 |     }
7 |     return ans;
8 | }
```

Độ phức tạp khi truy vấn tổng: $\mathcal{O}(\log n)$.

Thao tác cập nhật

Để cập nhật phần tử tại vị trí u , ta sẽ thực hiện quá trình ngược lại so với khi truy vấn tìm tổng: cộng bit nhỏ nhất vào u cho đến khi u vượt ngoài biên của mảng.

```
1 | void update(int u, int v) {
2 |     int idx = u;
3 |     while (idx <= n) {
4 |         bit[idx] += v;
5 |         idx += (idx & (-idx));
6 |     }
7 | }
```

Chứng minh tính đúng đắn của thuật trên như sau: mỗi khi ta cộng thêm 1 lượng bằng với 2^k (k là bit nhỏ nhất của u) thì đoạn dịch qua phải một lượng 2^k thành $[l + 2^k, r + 2^k]$ (do bit nhỏ nhất lúc này vẫn có thể tính là 2^k). Đồng thời lúc đó, bit nhỏ nhất tăng ít nhất 2 lần do 2^k (mới cộng thêm) + 2^k (có sẵn trong u) tạo thành 2^{k+1} làm cho biên trái dịch trái ít nhất 2^k lần thành $[l, r + 2^k]$ (nếu có sẵn 2^{k+1} trong u thì tiếp tục gộp lại làm bit nhỏ nhất tăng lên là 2^{k+2} , ...), do đó biên trái luôn được giữ \leq biên l ban đầu.

Mỗi lần cộng thêm, bit cuối luôn bị dịch lên ít nhất 1 lần, dẫn đến có tối đa $\log n$ lần dịch bit. Vì thế độ phức tạp khi cập nhật là $\mathcal{O}(\log n)$.

Lưu ý

Bằng cây chỉ số nhị phân (BIT), ta dễ dàng tính được prefix sum và cập nhật giá trị chỉ trong $\mathcal{O}(\log n)$, mặt khác so với các CTDL khác, BIT dễ dàng cài đặt hơn rất nhiều và không tốn quá nhiều thời gian để code.

Quay lại bài toán đầu, nếu chúng ta thay đổi yêu cầu thành tìm tổng trên đoạn $[l \dots r]$, tính chất của prefix sum dễ dàng cho ta tìm được kết quả thông qua phép $sum(r) - sum(l - 1)$. Tuy nhiên, không phải tất cả phép toán nào đều cho phép chúng ta dễ dàng lấy kết quả thông qua phép hiệu như vậy. Đối với các phép \min , \gcd , không tồn tại phép hiệu cho ta phép lấy kết quả của một đoạn dễ dàng, vì thế ta không thể áp dụng BIT đối với những bài toán loại này.

Đây là một khuyết điểm mấu chốt của BIT, vì thế cần nắm rõ tính chất và những bài toán để quyết định có nên sử dụng BIT không.

Cập nhật đoạn

Ta thay đổi nội dung bài toán ban đầu như sau:

- 1 vl : cộng v vào tất cả phần tử $A[l], A[l + 1], A[l + 2], \dots, A[r]$.
- 2 u : tìm giá trị hiện tại của $A[u]$.
- 3 lr : tính tổng các phần tử từ $A[l], A[l + 1], A[l + 2], \dots, A[r]$.

Ta có thể cài đặt "ngây thơ" bằng cách áp dụng hàm `update()` trên tất cả phần tử cần được cập nhật, độ phức tạp khi này là $\mathcal{O}(Q \cdot N \log N)$. Dĩ nhiên cách này quá chậm, đòi hỏi ta cần tìm một cách cập nhật đoạn một cách nhanh hơn để giữ nguyên độ phức tạp $\mathcal{O}(N \log N)$.

Truy vấn từng phần tử

Mảng hiệu (difference array) là một loại mảng lưu hiệu giữa các phần tử liên kề với nhau.

Mảng hiệu được xây dựng bằng cách sau:

- Với $i = 1$ thì $diff[i] = A[i]$.
- Với $2 \leq i \leq N$ thì $diff[i] = A[i] - A[i - 1]$.

Bạn có thể theo dõi hình dưới và code minh họa để hiểu rõ hơn:

	1	2	3	4	5	6	7	8
A	5	1	2	8	5	9	3	5
$Diff$	5	-4	1	6	-3	4	-6	2

```

1 | int diff[N + 1];
2 |
3 | diff[1] = a[1];
4 | for (int i = 2; i <= n; ++i) {
5 |     diff[i] = a[i] - a[i - 1];
6 |     // lấy phần tử thứ i trừ cho phần tử trước nó
7 | }

```

Khi cộng tất cả phần tử $diff$ từ 1 đến i , ta có:

$$\begin{aligned}
 \sum_{j=1}^i diff[j] &= diff[1] + diff[2] + \dots + diff[i] \\
 &= a[1] + a[2] - a[1] + a[3] - a[2] + \dots + a[i] - a[i - 1] \\
 &= a[1] - a[1] + a[2] - a[2] + \dots + a[i - 1] - a[i - 1] + a[i] \\
 &= a[i]
 \end{aligned}$$

Từ tính chất này, khi tính được mảng hiệu, để tính được giá trị của $a[i]$ ta chỉ cần lấy tổng của i phần tử $diff$ đầu tiên. Khi này, bài toán của chúng ta thực chất được đưa về tính tổng trên mảng $diff$, vấn đề hiện tại là thao tác `update()` cần được xử lý như thế nào.

Hình dưới đây minh họa thao tác cập nhật trên một đoạn $[l \dots r]$ - từ mảng trên, ta cộng $\Delta = 4$ vào đoạn $[4 \dots 7]$:

A	5	1	2	8	5	9	3	5
$Diff$	5	-4	1	6	-3	4	-6	2
				↓	↓	↓	↓	
$New\ A$	5	1	2	12	9	13	7	5
$New\ Diff$	5	-4	1	10	-3	4	-6	-2
$Old\ Diff$	5	-4	1	6	-3	4	-6	2
Δ	0	0	0	4	0	0	0	-4

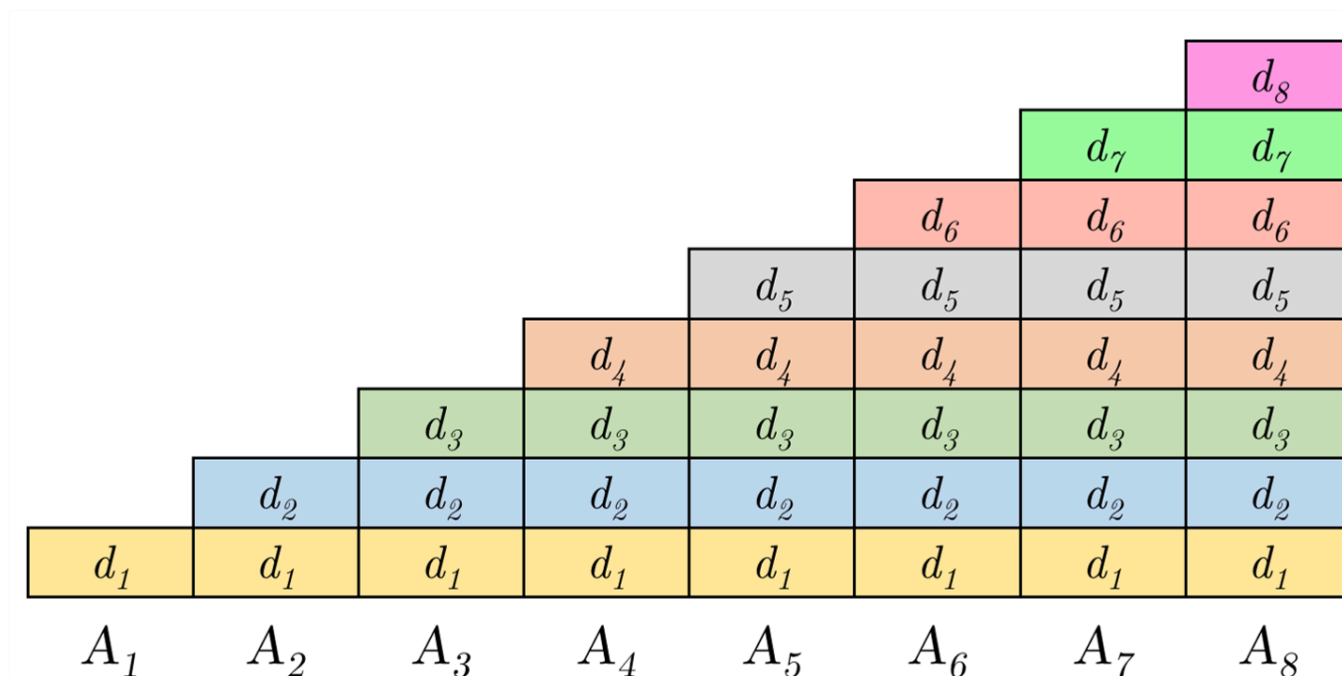
Khi cập nhật, do các phần tử liên kề trong đoạn $[l \dots r]$ đều được cộng cùng một giá trị Δ nên hiệu giữa chúng thực chất vẫn không đổi. Khác biệt duy nhất khi cập nhật nằm ở 2 biên của đoạn: giữa (a_{l-1}, a_l) và (a_r, a_{r+1}) ; vì thế ta chỉ cần cập nhật điểm tại 2 biên trên mảng hiệu và dùng truy vấn lấy tổng để tính giá trị hiện tại của a_i .

```

1  void updatePoint(int u, int v) {
2      int idx = u;
3      while (idx <= n) {
4          bit[idx] += v;
5          idx += (idx & (-idx));
6      }
7  }
8
9  void updateRange(int l, int r, int v) {
10     updatePoint(l, v);
11     updatePoint(r + 1, -v);
12 }
13
14 int get(int u) {
15     int idx = u, ans = 0;
16     while (idx > 0) {
17         ans += bit[idx];
18         idx -= (idx & (-idx));
19     }
20     return ans;
21 }

```

Truy vấn trên đoạn



Hình trên sẽ giúp ta minh họa trực quan hơn mối quan hệ về tổng các phần tử với mảng A và mảng hiệu $diff$.

Nhắc lại: $A[i] = \sum_{j=1}^i diff[j]$. Dựa vào hình, ta có thể tính lần lượt tổng các phần tử từ A_1 đến A_i như sau:

- $sum[1] = diff[1]$
- $sum[2] = 2 \cdot diff[1] + diff[2]$
- $sum[3] = 3 \cdot diff[1] + 2 \cdot diff[2] + diff[3]$
- ...
- $sum[i] = i \cdot diff[1] + (i-1) \cdot diff[2] + \dots + (i-j+1) \cdot diff[j] + \dots + 2 \cdot diff[i-1] + diff[i]$

Tuy nhiên, do sự biến động của hệ số khi nhân nên cách này không thuận tiện khi ta phải truy vấn liên tục. Để dễ dàng hơn, ta sẽ cố định mỗi $diff[i]$ nhân với hệ số $n - i + 1$, khi này:

- $sum[1] = n \cdot diff[1] - (n-1) \cdot diff[1]$
- $sum[2] = n \cdot diff[1] + (n-1) \cdot diff[2] - (n-2) \cdot (diff[1] + diff[2])$
- $sum[3] = n \cdot diff[1] + (n-1) \cdot diff[2] + (n-2) \cdot diff[3] - (n-3) \cdot (diff[1] + diff[2] + diff[3])$
- ...
- $sum[i] = n \cdot diff[1] + (n-1) \cdot diff[2] + \dots + (n-j+1) \cdot diff[j] + \dots + (n-i+1) \cdot diff[i] - (n-i) \cdot (diff[1] + diff[2] + \dots + diff[i])$

Tóm lại, ta thu được:

$$sum[i] = \sum_{j=1}^i (n - j + 1) \cdot diff[j] - (n - i) \cdot \sum_{j=1}^i diff[j]$$

Cả hai giá trị $diff[j]$ và $(n - j + 1) \cdot diff[j]$ đã được đơn giản hóa, lúc này ta chỉ cần lưu toàn bộ giá trị $(n - j + 1) \cdot diff[j]$ vào mảng S_1 và $diff[j]$ vào mảng S_2 và dựng mảng cộng dồn trên hai mảng đó.

Thao tác cập nhật trên mảng S_2 giống với thao tác cập nhật đã đề cập ở trên, còn ở mảng S_1 thì khác biệt duy nhất là việc xử lý nhân hệ số $(n - j + 1)$. Tuy vậy, hệ số trên không đổi trong quá trình tính toán với từng phần tử nên ta chỉ cần nhân hệ số này với giá trị cần cập nhật và áp dụng phương thức tương tự như ở cập nhật trên S_2 .









Code tham khảo:


```

1  vector<int> bit1, bit2;
2  /*
3   * Các hàm update và sum cần làm việc trên một trong hai BIT riêng biệt.
4   * Sử dụng vector cho phép truyền BIT vào làm việc trực tiếp dễ dàng hơn.
5   */
6
7  void updatePoint(vector<int>& b, int u, int v) {
8      int idx = u;
9      while (idx <= n) {
10         b[idx] += v;
11         idx += (idx & (-idx));
12     }
13 }
14
15 void updateRange(int l, int r, int v) {
16     updatePoint(bit1, l, (n - l + 1) * v);
17     updatePoint(bit1, r + 1, -(n - r) * v);
18     updatePoint(bit2, l, v);
19     updatePoint(bit2, r + 1, -v);
20 }
21
22 int getSumOnBIT(vector<int>& b, int u) {
23     int idx = u, ans = 0;
24     while (idx > 0) {
25         ans += b[idx];
26         idx -= (idx & (-idx));
27     }
28     return ans;
29 }
30
31 int prefixSum(int u) {
32     return getSumOnBIT(bit1, u) - getSumOnBIT(bit2, u) * (n - u);
33 }
34
35 int rangeSum(int l, int r) {
36     return prefixSum(r) - prefixSum(l - 1);
37 }

```

Bài tập áp dụng

- [LQDOJ - Query-Sum](#) 
- [LQDOJ - Query-Sum 2](#) 
- [LQDOJ - Candies](#) 
- [CSES - Range Update Queries](#) 
- [CSES - Nested Range Count](#) 
- [Codeforces - Moving Points](#) 
- [VNOJ - NKINV](#) 
- [VNOJ - INCVN](#) 

VNOI Online Judge có phân loại riêng các bài tập về BIT, các bạn có thể tham khảo tại [đây](#) .